Relational Reasoning

Constraint Solving, Deduction, and Program Verification

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Aboubakr Achraf El Ghazi

aus Casablanca

Tag der mündlichen Prüfung: 27. Oktober 2015

Erste Gutachterin:	Prof. Dr. Mana Taghdiri Karlsruher Institut für Technologie
Zweiter Gutachter:	Prof. Dr. Bernhard Beckert Karlsruher Institut für Technologie
Dritter Gutachter:	Prof. Dr. Shmuel Tyszberowicz Academic College of Tel Aviv-Yafo

Acknowledgements

In the name of God, the most compassionate, the most merciful. All praises are due to God the Lord of the worlds and peace and blessings be upon his messenger Muhammad. The Prophet Muhammad, peace and blessings be upon him, said, as reported by Ahmad, Abu Dawud, Tirmidhi and others, "He has not thanked God who has not thanked people."

In this context, I would like to express my special appreciation and thanks to my advisor Professor Dr. Mana Taghdiri for giving me the opportunity to undertake this project and for her excellent personal and scientific comportment which helps me growing as a scientist.

I would also like to thank my second reviewer Professor Dr. Bernhad Beckert for fulfilling his role with commitment and my third and external reviewer Professor Dr. Shmuel Tyszberowicz for his detailed corrections and comments. I also address my thanks to all other defense committee members, Professor Dr. Hartmut Prautzsch, Professor Dr. Ralf H. Reussner, Professor Dr. Heinz Wörn and Professor Dr. Dennis Hofheinz.

Working as a member of the formal methods groups at KIT has been always a pleasure not least because of the colleagues. In this context, my thanks go to Dr. Mattias Ulbrich, Mihai Herda, Tianhai Liu, Dr. Christoph Gladisch, Ulrich Geilmann, David Faragó, Markus Iser, Dr. Vladimir Klebanov, Thorsten Bormer, Sarah Grebing, Simon Greiner, Dr. Daniel Grahl, Dr. Olga Tveretina, Florian Merz, Dr. Stephan Falke, Dr. Carsten Sinz, Dr. Christoph Scheben, Dr. Benjamin Weiß, Alexander Sebastian and Michael Kirsten. I want also to express my gratitude to Professor Dr. Peter H. Schmitt for getting me interested in formal methods by his excellent lectures.

I would also like to express my thanks to Dr. Fouad Omri , Abdelhakim El Fadil and Dr. Martin Do for their poof readings.

I owe my deepest gratitude to my mother Zohra Laaroubi, and my father Mohamed El Ghazi as well as all members of my family.

Lastly, but most importantly, infinitely many gratitude to my wife Alev El Ghazi for her love and care and to my son Abdulmalik El Ghazi for just being there, thank you.

Karlsruhe, 23. November 2015 *Aboubakr Achraf El Ghazi*

Relational Reasoning

Solving, Deducing, and Java Verification

(Deutsche Zusammenfassung)

Software-Systeme spielen eine immer größere und wichtigere Rolle in unserem Alltag. Dadurch steigt der Bedarf an gesicherten Aussagen über ihr Verhalten. Dies betrifft nicht nur sicherheitskritische Bereiche wie die Medizintechnik, sondern auch nicht sicherheitskritische Bereiche in Betracht der immensen Auswirkung von Softwarefehlern auf die Wirtschaft.

Um Aussagen über das Verhalten von Softwaresystemen zu bewerten, benötigt man: (1) formale Sprachen für die Spezifizierung ihres erwarteten Verhaltens sowie für die Beschreibung des Softwaresystems selbst, und (2) ein Rahmenwerk und eine Methodologie um die Korrektheit des spezifizierten Verhaltens zu bemessen. Dabei kann die Korrektheitsbemessung über Testverfahren – Testing – bis hin zu formalen Korrektheitsbeweisen – formale Verifikation – erfolgen.

Diese Arbeit beschäftigt sich mit der formalen Verifikation von Softwaresystemen bezogen auf formale Spezifikationen, die auf dem *Formal-Methods* Ansatz beruhen. Dabei wird das Software-System zusammen mit seiner Spezifikation zu einer Formel transformiert, die genau dann valid ist, wenn das Softwaresystem die Spezifikation erfüllt. Im Gegensatz zum in der Industrie gängigem *Testing* Ansatz, bittet der *Formal-Methods* Ansatz die höchst mögliche Zuverlässigkeit, ist allerdings im Allgemeinen teuer. In den letzen Jahren hat der Formal-Methods Ansatz in der Industrie signifikant an Bedeutung gewonnen. Der jüngste Standard des europäischen Komitees für Steuerungs- und Schutzsysteme im Schienenverkehr [1][1] sei an dieser Stelle als Beleg für diese Entwicklung genannt. Auch der Umfang an Ausgaben für Formal-Methods, $1.5B in 2000 [15, page 5], bestätigt diesen Trend.

Zusätzlich zum eigentlichen Beweisprozess, wird die formale Softwareverifikation vom Stil und der Ausdruckmächtigkeit der Spezifikationssprache und ihrer zugrundelegenden Logik in zwei Weisen beeinflusst: (1) Je mehr das erwartete Softwareverhalten, das im Allgemeinen auf einem hohen Abstraktionslevel angesiedelt

[1] Formal methods are explicitly identified as relevant. More precisely, they are "highly recommended" for the safety integration levels 3 and 4.

ist, direkt ausgedrückt werden kann, desto besser können Spezifikationsfehler vermieden werden, (2) je mehr Implementierungsdetails abgekapselt werden können, desto genauer kann der Kern der Software erfasst und verifiziert werden.

Diese Arbeit zielt drauf ab, ein Beweis-Rahmenwerk für die funktionale Verifikation von Softwaresystemen bezogen auf relationale Spezifikationen in einer relationalen Logik erster Stufe wie Alloy [47] zu schaffen. Dabei kann die Systembeschreibung wahlweise in dem relationalen Abstraktionslevel, am Beispiel von Alloy, oder in dem detaillierten Implementationslevel, am Beispiel von Java erfolgen. Bei so einem hoch angesetzten Ziel ist der Bedarf an Experten-Interaktion in hohem Maß erforderlich. Eine ganz spezielle Anforderung an das Beweis-Rahmenwerk ist von daher die Reduktion von Interaktionen.

Um dieses Ziel zu erreichen, wurden zwei zueinander komplementäre Tools entwickelt. Das erste Tool , namens AlloyPF, bietet ein kosteneffektives Rahmenwerk für das Beweisen und Widerlegen von Alloy Spezifikationen an. Dafür werden sowohl automatisch als auch interaktiv entwickelte Beweistechniken miteinander kombiniert. Das zweite Tool, namens JKelloy, bietet einen deduktiven Beweisassistent für die Verifikation von Java Programmen bezügliche Alloy Spezifikationen.

Da der Versuch, unerfüllbare Beweisverpflichtungen zu beweisen, besonders teuer ist, startet AlloyPF zunächst in einem *begrenzten Verifikationsmodus*. Dabei wird die Kardinalität der Sorten der Alloy Beweisverpflichtung auf ein vom Benutzer gegebenes Maximum – genannt *scope* – begrenzt und die Beweisverpflichtung auf Gegenbeispiele untersucht. Dies erlaubt dem Benutzer den scope zu erhöhen und so immer mehr Konfidenz über die Korrektheit der Alloy Beweisverpflichtung zu gewinnen bevor die eigentliche *volle Verifikation* gestartet wird. Diese Korrelation zwischen der Größe des scope innerhalb dessen keine Gegenbeispiele existieren und der Konfidenz über die Korrektheit der Alloy Beweisverpflichtung beruht auf einer nicht bewiesenen aber im Kontext von Alloy verbreiteten Hypothese, die besagt, wenn Gegenbeispiele existieren, dann existieren sie auch innerhalb von kleinen scopes. Die begrenzte Verifikation wird von dem Alloy Analyzer durchgeführt. In diesem Kontext integriert AlloyPE die *vollständigkeitserhaltenden Grundtermmengen* (SufGT) Technik, die für jede universale quantifizierte Variable eine *vollständigkeitserhaltenden Grundtermmenge* ausrechnet. Dadurch kann AlloyPE Sorten der Alloy Beweisverpflichtung erkennen, die einen *vollständigkeitserhaltenden scope* aufweisen, d.h., wenn keine Gegenbeispiele innerhalb dieses scope existieren, dann existieren überhaupt keine. Falls alle Sorten einer Alloy Beweisverpflichtung einen vollständigkeitserhaltenden scope aufweisen, dann kann die begrenzte Verifikation binnen dieses scope auch die Korrektheit beweisen.

Nachdem eine initiale Konfidenz über die Korrektheit der Alloy Beweisverpflichtung durch die begrenzte Verifikation gewonnen wurde, startet der volle Verifikationsmodus mit der auf *satisfiability modulo theories* (SMT) [2] basierenden und automatischen Beweisengine AlloyPE. AlloyPE übersetzt die negierte Alloy Beweisverpflichtung zu einer erfüllbarkeitsäquivalenten Formel in einer relationalen SMT Logik namens RFOL. Die aus der Übersetzung resultierenden Formeln werden durch zwei Techniken vereinfacht.

Die erste Vereinfachungstechnik heißt *extended semantics blasting* (SB^+) und eliminiert durch die RFOL Axiome induzierte Kardinalitätseinschränkungen. Diese Simplifikation ist unabdingbar für die Analysierbarkeit der aus der Alloy Übersetzung nach RFOL resultierenden Formeln, verletzt aber im Allgemeinen die Erfüllbarkeitsäquivalenz. Um dieses Problem zu lösen wurde das logische Fragment für welche SB^+ erfüllbarkeitsäquivalent ist, zusammen mit einem kosteneffektiven Testsystem für die Prüfung von RFOL Formeln auf Enthaltensein in diesem Fragment definiert.

Die zweite Vereinfachungstechnik basiert auf der zuvor erwähnten SufGT Technik. In diesem Kontext wird die SufGT Technik benutzt, um universal-quantifizierte Variablen, deren berechnete vollständigkeitserhaltende Grundtermmenge endlich ist, zu eliminieren und so die Komplexität der RFOL Formeln zu reduzieren.

Die RFOL Axiomatisierung zusammen mit den SB^+ und SufGT Techniken erlauben eine Vielzahl von (beweisbaren) Alloy Spezifikationen mit Hilfe von SMT Beweisern automatisch zu beweisen. Für besonders komplexe Spezifikationen ist die integerbasierte RFOL Axiomatisierung für rekursive Theorien wie der transitiven Hülle nicht ausreichend. Für solche Spezifikationen setzt AlloyPE einen speziell für die Transitivehülle-Theorie entwickelten Kalkül (TCPInv) ein. Der TCPInv Kalkül basiert auf sogenannten *essentiellen Pfad-Invarianten*. Sollte auch der TCPInv Kalkül die Alloy Spezifikation nicht beweisen können, startet die deduktive Alloy Beweisengine namens Kelloy. Im Gegensatz zu AlloyPE ist Kelloy hauptsächlich interaktiv und basiert auf dem interaktiven Theorem-Beweiser KeY [12].

Im Gegensatz zu AlloyPF, beschäftigt sich JKelloy mit einem erweiterten Szenario, indem das erwartete Softwareverhalten zwar in Alloy spezifiziert ist, aber die Software selbst in Java beschrieben ist, sprich in Code-Form gegeben. Um überhaupt die relationale Spezifikation von Java Programmen unterstützen zu können, generiert JKelloy für ein gegebenes Java Programm einen *Alloy Kontext*. Der Alloy Kontext stellt für die Java Klassen und Felder zustandsabhängige, relationale Repräsentanten zur Verfügung und ermöglicht so das Schreiben einer relationalen Spezifikation für das gegebene Java Programm. Ist eine relationale Spezifikation des Java Programms gegeben, so generiert JKelloy eine entsprechende Beweisverpflichtung in *relational JavaDL* – die relationale Erweiterung der dynamischen Java Logic (JavaDL) des KeY Systems. Die Relation zwischen den relationalen Zuständen der Spezifikation und der Zustände des Java Programms wird durch automatisch generierte *Kopplungs-Axiome* festgelegt. Um den Beweisprozess in JKelloy auf dem hohen Abstraktionslevel der relationalen Spezifikation durchführen zu können und besser zu automatisieren, bietet JKelloy zwei Kalküle an.

Der erste Kalkül heißt *Heap-Resolution* und hat die Aufgabe relationale Ausdrücke über komplexe Heaps zu relationalen Ausdrücken über atomare Heaps zu reduzieren. Das Resultat dieser Reduktion ist eine relationale Interpretation des Java Programms, was den JavaDL Kalkül überflüssig macht, so dass Beweise allein mit dem relationalen Kalkül erfolgen können.

Der zweite Kalkül heißt *Override-Simplifikation* und strebt die weitere Vereinfachung und Automatisierung des Beweisprozesses in JKelloy an. Der Kalkül versucht die Re-

duktion der relationalen Operatoren auf ihren elementaren mengentheoretischen Definitionen zu reduzieren, insbesondere solche die auf Override-Ausdrücke angewendet werden. Override-Ausdrücke werden durch den relationalen Alloy update Operator "$\oplus$" (genannt *override*) gebildet und resultieren typischerweise aus dem Heap-Resolution Kalkül.

Die Arbeit fördert und unterstützt relationale Logiken für die Spezifikation von Softwaresystemen, insbesondere solche mit komplexen Manipulationen von verlinkten Datenstrukturen. Dafür entwickelte die vorliegende Arbeit geeignete und praktische Techniken für die auf relationalen Spezifikationen basierte Verifikation. Im Einzelnen sind die wesentlichen Beiträge der Arbeit die Folgenden:

- Eine relationale Logik (RFOL), die es erlaubt Alloy Beweisverpflichtungen struktur- und erfüllbarkeitserhaltend auszudrücken. Die RFOL Logik basiert auf die Prädikatenlogik erster Stufe modulo einer Gleichheits- und Integertheorie sowie einem flachen Typsystem.

- Eine vollständigkeitserhaltende Grundtermmengen Technik (SufGT), die für jede universalquantifizierte Variable eine vollständigkeitserhaltende Grundtermmenge iterativ berechnen kann. Existenzquantifizierte Variablen werden skolemisiert. Diese Grundtermmengen werden in der ersten Stelle zur Reduktion von quantifizierten Variablen benutzt. Zusätzlich werden diese Mengen, genauer ihre Kardinalitäten benutzt, um (1) die Skalierbarkeit der begrenzten Verifikation zu erhöhen und (2) die Korrektheit von einfachen Alloy Beweisverpflichtungen via begrenzter Verifikation zu beweisen.

- Eine erweiterte Semantik-Blasting Technik (SB^+) eliminiert die von RFOL Axiomen induzierte Kardinalitätseinschränkungen. Da die SB^+ Technik die Erfüllbarkeitsäquivalenz im Allgemeinen nicht erhält, wird ein logisches Fragment, für welches SB^+ erfüllbarkeitsäquivalent ist, zusammen mit einem kosteneffektiven Testsystem für die Prüfung von RFOL Formeln auf Enthaltensein in diesem Fragment definiert.

- Eine auf Pfad-Invarianten basierte Beweis-Technik für die Transitive-Hülle-Theorie (TCPInv), die in der Lage ist besonders komplexe Alloy Beweisverpflichtung automatisch zu beweisen. Dies betrifft Beweisverpflichtungen, die ansonsten Transitive-Hülle-bezogene Induktionsbeweise benötigen.

- Ein Beweisassistent für die Verifikation von Java Programmen bezogen auf relationale Spezifikationen (JKelloy). JKelloy generiert für ein gegebenes Java Programm automatisch einen Alloy Kontext, der die relationale Sicht auf Java Klassen und Felder darstellt und dadurch die relationale Spezifikation von Java Programmen ermöglicht. JKelloy generiert auch für ein gegebenes Java Programm mit relationaler Spezifikation die eigentliche Beweisverpflichtung in relational JavaDL sowie die Kopplungs-Axiome, die die Relation zwischen den relationalen Zuständen der Spezifikation und den Zuständen des Java Programms festgelegt.

- Der Heap-Resolutions-Kalkül, der relationale Ausdrücke über komplexe Heaps zu relationalen Ausdrücken über atomare Heaps reduziert. Dadurch ermöglicht der Kalkül die Verifikation von Java Programmen bezüglich relationaler Spezifikationen auf dem abstrakten Relationslevel durchzuführen.
- Der Override-Simplification-Kalkül, der die Reduktion von relationalen Operatoren auf ihre elementare, mengentheoretische Definition zu reduzieren versucht.

Contents

List of Figures

List of Tables

List of Definitions

List of Theorems

Chapter 1

Introduction

1.1 Motivation

Software systems are playing an increasingly important role in our daily life, which increases the need for guaranteed claims about their behavior. This applies not only to safety critical domains such as medical and traffic systems, but also to other domains considering the impact of software failures on the economy.

In order to check software behavior, two major components are required: (1) an intuitive *formal language*, with formal semantics, for the formulation of the expected behavior as well as for the description of the software system itself, and (2) a framework and a methodology for gaining confidence about software correctness, which can vary from random checks to a complete proof of correctness.

This dissertation exploits the *formal methods paradigm* in which the software system together with the *required behavior* are transformed to a logical formula —*proof obligation*— in a formal language, such that the formula is valid if and only if the software satisfies the required behavior. Contrary to the *testing paradigm* in which the software system is tested against a finite number of tests, the formal method approach can provide (mathematical) proofs of correctness and is thus the method of choice for safety critical systems. Over the last few years, this approach has gained more and more acceptance in industry. This is reflected, for example, by the last standard of the European Committee for Standardization for railway control and protection systems [1].[1] The amount of investments in this area, $1.5B in 2000 [15, page 5], confirms this trend.

Beside the actual reasoning, formal software verification is also deeply affected by the nature and expression power of the specification language and its underling logic in two folds: (1) the more the required behavior, which is naturally on a higher abstraction level, can be expressed directly, the more specification errors can be avoided, (2) the more implementation details can be encapsulated, the more the core of the software system can be captured and verified. Although formal verification on the detailed level of implementation is amongst the most difficult disciplines of

[1] Formal methods are explicitly identified as relevant More precisely, they are "highly recommended" for the safety integration levels 3 and 4.

formal verification, a comprehensive formal verification framework should also allow to perform verification at this level of detail, since this reflects, in fact, how software is executed.

This thesis aims at providing a reasoning framework for the *functional verification* of software systems against *relational specifications* written in a first-order relational logic such as Alloy [47]. The system description can be given either at the *abstract relational level*, e.g. using Alloy, or at the *detailed implementation level*, e.g. using Java. A distinguishing goal of this reasoning framework is to reduce the human cost. This feature will especially ease the use of formal methods in real-life applications.

We provide two complementary approaches. The first one is implemented in *AlloyPF* —a framework based on a mixture of techniques to provide both proofs and counterexamples for software systems described and specified in Alloy. The second approach is implemented in *JKelloy* —a proof assistant for verifying Java programs with respect to Alloy specifications.

1.2 Alloy and Abstractions

For complex manipulations of linked data structures, as used in most object-oriented software, the natural abstraction is relational. Alloy [47] is a declarative, typed, first-order relational logic[2] with built-in operators for transitive closure, set cardinality, and integer arithmetic. The declarative aspect of the language is not strict but reflects how Alloy is indeed used: it promotes abstractions. In addition to be the core of software development (see [47, page 1]), abstractions are especially useful for encapsulating implementation details that can cumber the application of formal methods to large-sized software.

Alloy offers a suitable set of means to easily and directly describe software systems together with their required behaviors in a declarative and implementation abstract stile. Alloy descriptions of software systems are called *models*. Alloy descriptions of required behaviors are called *assertions*. In the verification context, we refer to an Alloy model together with an Alloy assertion as Alloy specification.

The fully automatic *Alloy Analyzer* —an important reason for the success and popularity of Alloy— *checks* Alloy specifications by looking for a counterexample. This check, however, is performed with respect to a *bounded scope* in which only a small number of values is considered for each type. Therefore, although the Alloy Analyzer can find counterexamples efficiently, it cannot, in general, prove the validity of the proof obligation.

[2] We denote by logic, deviating from the Alloy book, a (formal) language equipped with semantics.

1.3 Multi-Layer Framework for Proving Relational Specifications

In order to prove Alloy specifications cost-effectively, we have developed AlloyPF—a layered framework for proving Alloy specifications with increasing costs (Figure 1.1, bottom part). Since trying to prove an invalid proof obligation is particularly costly, our framework starts in the *bounded-verification* mode first, trying to find a counterexample in a finite scope. This allows the user to increase the scope arbitrarily in order to gain more *confidence* about the correctness of the Alloy specification before switching to the *full-verification* mode. The bounded-verification is performed using the Alloy Analyzer. In this context, our framework integrates our *sufficient ground term sets* (SufGT) technique. This technique allows to detect all types of an Alloy specification admitting *sufficient maximal bounds*, i.e., if no counterexample exists within such a bound, then none will exist beyond that bound. If all types of an Alloy specification admit sufficient maximal bounds, then the bounded verification —up to that bound— will either show a counterexample or provide a proof of correctness.

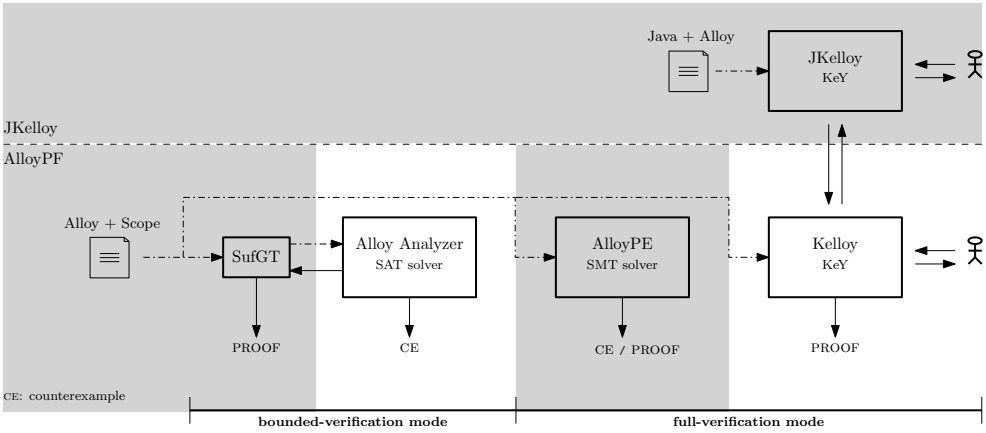


Figure 1.1: Our main tools *AlloyPF* and *JKelloy*. The main thesis contributions are shown in gray.

After gaining initial confidences in the correctness of the proof obligation through bounded verification, the full-verification starts with our satisfiability modulo theory (SMT) based verification technique, which we implement in the *Alloy proof engine* (AlloyPE) tool. AlloyPE translates the *negation* of the underling proof obligation of the Alloy specification to an SMT equisatisfiable formula. First-order operators such as union, intersection, join, etc., are axiomatized via pure first-order axioms. Non first-order operators, namely transitive closure, set cardinality, and ordering, are axiomatized with the help of the SMT integer theory. The resulting equisatisfiable

SMT formula is then simplified using multiple techniques, two of which are presented here.

The first simplification technique is called *extended semantics blasting* (SB^+) by which non-recursive operator axioms are applied on-demand, and canceled afterwards. This simplification step is crucial for the efficiency of our SMT based verification of Alloy specifications. It eliminates too restrictive cardinality constraints induced by our *complete* axiomatization. The axioms elimination step, even after exhaustive applications, can, in theory, introduce incompleteness issues. Therefore, we describe the SB^+ completeness fragment and provide a cost effective testing system for the inclusion of a formula in this fragment.

The second simplification technique is the already mentioned SufGT technique, which performs a ground term occurrence analysis to determine a set of *sufficient* ground terms for each universally quantified variable. The computed ground term sets are used to eliminate quantified variables, and thus reducing the complexity of general SMT formulas including ours.

Our experimental observations (see Chapter 6 and [30]) show that the SMT translation plus SB^+ and SufGT simplifications can prove almost all provable Alloy benchmarks which do not need induction reasoning. Since most needed inductions in Alloy specifications are concerned with the transitive closure, we developed a *path-invariants* based transitive closure reasoning technique (TCPInv). Informally, the TCPInv technique introduces a pure first-order *weak* axiomatization of the transitive closure, called WTC, together with an efficient test for inclusion in the WTC completeness fragment. Transitive closure expressions beyond the WTC-fragment, are then handled via detection and injection of relational path-invariants, these are formulas that hold for all nodes along a path in the graph of the transitive closure operand relation. Using this path-invariants, the original proof obligation is extended, equisatisfiably, until the SMT solver can show its validity.

If the SMT-based verification does not succeed, our interactive theorem proving (ITP) based verification starts. At this stage, the proof obligation is proven using *Kelloy* [75, 74], our relational extension of the KeY system [12]. For this purpose, the proof obligation is translated to KeY's typed first-order logic[3]. For the sake of completeness, but also to lower the burden of interactive proving, we define for each Alloy operator a counterpart in KeY's typed first-order logic. This requires the introduction of a complete relational first-order theory including relation and tuple types as well as semantics axioms and inference rules.

1.4 Verification of Java Programs via Relational Reasoning

In order to provide a deductive reasoning for Java programs with respect to Alloy specifications, we developed JKelloy (Figure 1.1, top part). Given a Java program,

[3] It includes an integer theory.

JKelloy provides an Alloy *context* —state depending relational counterparts for Java classes and fields— which allows to write relational specifications of the Java program following the design-by-contract paradigm [58]. Given a Java program and its relational specification, JKelloy produces a KeY proof obligation in our *relational JavaDL* —a logic that uses our relational logic embedding in KeY for the relational specification, and KeY's Java dynamic logic (JavaDL) for the program code. The relationship between both logics is established by the so called *coupling axioms*. In order to both lift the (interactive) proofs to the higher abstraction level of relations and to better automate them, two additional calculi are introduced.

The first calculus —called *heap resolution*— reduces relational expressions over composed states to relational expressions over primitive states —usually the initial state— and thus makes the proof obligation pure relational; JKelloy is equipped with a rule application strategy that automatically achieves this task.

The second calculus —called *override simplification*— aims at further simplifying and automating the reasoning process in JKelloy. It tries to reduce the need for expanding the definitions of relational operators during the reasoning, especially of those that are applied to *override expressions*. Override expressions —built by the relational override operators— typically result from applying the relational heap resolution calculus and encode relationally the effect of the individual Java program statements to the individual field relations. The calculus consists of a set of proved lemma-rules which exploit the shape of pure relational proof obligations that result from applying the heap resolution calculus to proof obligation of Java programs against relational specifications.

1.5 Contributions

This thesis aims at promoting the use of relational logics, at the example of Alloy, for the specification of software systems, especially those with complex linked data manipulations, as is the case for most object-oriented software. It thus provides a reasoning framework together with a set of reasoning techniques for relational-specification based verification tasks. In particular, the main contributions of the thesis are the following:

- A relational first-order logic (RFOL), which allows a structure preserving and equisatisfiable formulation of pure relational proof obligations. The logic is based on typed first-order logic with equality and flat type system as it is supported by most SMT solvers. First-order relational operators are axiomatized in RFOL by pure first-order axioms; non first-oder relational operators like transitive closure and set cardinality are axiomatized based on the SMT integer theory. Further details on RFOL can be found in Chapter 4 and [30].

- A sufficient ground term sets (SufGT) technique, which computes iteratively a set of sufficient ground terms of each universally quantified variable —existentially quantified variables are skolemized. We use this technique preliminary to reduce

the number of quantified variables of RFOL formulas and thus the complexity of our proof obligations. In addition, we use this technique to also (1) increase the scalability of Alloy's bounded verification —using the Alloy Analyzer— by computing maximal scopes for Alloy signatures. Signatures admitting such maximal scopes have only variables whose computed sufficient ground term sets are finite, and (2) prove the correctness of Alloy specifications via bounded verification if all quantified variables of the specification have finite sufficient ground term sets. Further details on the SufGT technique can be found in Chapter 7 and [34].

- An extended semantics blasting technique (SB^+), which eliminates cardinality constraints on the relational sorts induced by the RFOL axioms. Since in general, SB^+ does not preserve satisfiability, we have described *the* logical fragment in which our technique is complete. In addition to the theoretical description of the fragment, we developed practical tests that provide a cost effective testing system for the inclusion of an RFOL formula in the SB^+ fragment. Further details on the SufGT technique can be found in Chapter 6.

- A path-invariant based transitive closure reasoning technique (TCPInv), which can show the refutation of RFOL formulas[4] involving transitive closure for which standard SMT solving cannot show refutation. Such formulas can not be refuted via standard SMT solving (without induction) because there exists no finite instantiation of their axioms such that their ground formulas become refutable. The TCPInv technique bases on a pure first-order *weak* axiomatization of transitive closure and a procedure for the detection of path-invariants —formulas that hold for all nodes along relational paths. The fact that the base transitive closure axiomatization is integer free, reduces in general the complexity of RFOL formulas and thus improves their SMT solving. Further details on the SufGT technique can be found in Chapter 8 and [31].

- JKelloy, a tool for the deductive verification of Java programs against Alloy specifications, has been presented. It is our framework for the relational specification of Java programs and the corresponding deductive reasoning on the abstract level of relations —using a *relational view of the heap*. JKelloy automatically generates a so called *Alloy context* which encodes the relational view of the types and fields of the Java program in the pre- and post-state, and allows for writing relational specifications of Java programs following the design-by-contract paradigm. Given a relational specification of a Java program, JKelloy automatically generate the actual proof obligation using our relational JavaDL. Further details on JKelloy can be found in Chapter 9 and [32, 33].

- A set of automatically generated coupling axioms that define the link between the heap dependent relations in the Alloy specification and the Java program states.

[4] Showing the refutation of a formula F is equivalent to proving the validity of its negation $\neg F$.

- A heap resolution calculus, that normalizes relational JavaDL proof obligations over composed heaps to relational expression over constant heaps. The calculus is equipped with a rule application strategy that, after performing symbolic execution, automatically achieves the relational heap resolution task.

- An override simplification calculus, that eases the verification process in JKelloy, by reducing the need for expanding the definitions of relational operators, especially of those that are applied to override expressions.

1.6 Outline

The thesis is divided into nine chapters.

Chapter 2, the foundation chapter, provides general introductions to the Alloy specification language and its analysis, the satisfiability modulo theories problem and its analysis, and the KeY system and its Java dynamic logic. Further needed details are introduced in the chapters where they are used.

Chapter 3 gives an example-driven overview of the automatic proof process of AlloyPF. The details of the individual techniques used in the proof process are described in the next four chapters.

In Chapter 4, the syntax and semantics of our first-order relational logic RFOL are defined.

Chapter 5 defines the underling proof obligations of Alloy specifications and presents their translation to RFOL proof obligations. It shows proofs of correctness and completeness of the Alloy to RFOL translation and reports on the experimental results of proving Alloy specifications using the translation, without any further simplification.

Chapter 6 introduces our extended semantics blasting simplification technique SB^+. It presents the SB^+ procedure rules and defines its completeness fragment together with a cost effective testing system for the inclusion of RFOL formulas in the fragment. The chapter also reports on the experimental results of proving Alloy specifications using the Alloy to RFOL translation together with the SB^+ simplification technique and compares them with the experimental results of Chapter 5 and of the bounded analysis of the Alloy Analyzer.

Chapter 7 introduces our sufficient ground term sets technique SufGT. It presents the SufGT procedure rules and an algorithm that uses the computed sufficient ground terms for an efficient elimination of quantified variables. The chapter also provides a proof of correctness and completeness of the technique with respect to the elimination of quantified variables and reports on the experimental results on applying the SufGT technique to simplify general SMT benchmarks.

Chapter 8 introduces our path-invariants based transitive closure reasoning technique TCPInv. It proves the existence of effective path-invariants for each essential path in a refutable RFOL formula and presents procedures for the detection of the essential paths and their path-invariants. The chapter also reports on the experimen-

tal results of applying the TCPInv reasoning technique to prove Alloy specifications, including those which could not be proved in Chapter 6.

Chapter 9 introduces our tool for the deductive verification of Java programs against Alloy specifications JKelloy. It also reports on using JKelloy to prove three Java programs with relational specifications.

The thesis ends with a conclusion and discussion of future work in Chapter 10.

Chapter 2

Foundations: Alloy, SMT, KeY

This chapter provides a general introduction of the frameworks and tools used in this dissertation. More detailed information are introduced in the chapters where they are used.

2.1 Alloy

Alloy [47] is a modeling language based on a same named first-order relational logic with built-in operators for transitive closure, set cardinality, integer arithmetic, and comprehension. In this dissertation, we investigate the automatic verification of Alloy problems as well as the verification of Java program with Alloy annotations. For both tasks we address a subset of Alloy version 4.x[1]. This subset, called core Alloy, represents all commonly used Alloy constructors. Almost other constructors not present in core Alloy can be desugared to core Alloy.

2.1.1 Alloy Problem

As shown in Figure 2.1, an Alloy problem consists of a collection of type declarations, relation declarations, relational first order formulas marked as fact, and possibly an other formula marked as assertion to check or as predicate to run.

The type declarations introduce the global types (called signatures) which represent sets of atoms. The signature declaration

$$sig\ A\{\dots\}$$

declares a top-level type named A whereas

$$sigB\ (in \mid extend)\ A\{\dots\}$$

declares a subtype of A named B. The keyword *extend* additionally constraints B to be an interface of A —i.e, a type T is the union of all its interfaces.

[1] We started the investigation with version 4.1 and ended up with version 4.2

$problem ::= typeDcl^* \; relDcl^* \; fact^* \; (assertion \mid predicate)$
$typeDcl ::= \textbf{sig}$ identifier $[(\textbf{in} \mid \textbf{extends}) \; type]$
$relDcl ::= rel : type \; [[mult] \rightarrow [mult] \; type]^*$
$mult ::= \textbf{lone} \mid \textbf{some} \mid \textbf{one} \mid \textbf{set}$
$fact ::= formula$
$assertion ::= formula$
$predicate ::= formula$

$exp ::= type \mid var \mid rel \mid \textbf{none} \mid \textbf{iden} \mid exp + exp$
$\quad \mid exp \; \& \; exp \mid exp - exp \mid exp.exp \mid exp \rightarrow exp$
$\quad \mid exp <: exp \mid \tilde{}exp \mid \hat{}exp \mid \textbf{Int} \; intExp$

$intExp ::=$ number $\mid \#exp \mid \textbf{int} \; var$
$\quad \mid intExp \; intOp \; intExp \mid (\textbf{sum} \; [var : exp]^+ \mid intExp)$

$formula ::= exp \; \textbf{in} \; exp \mid exp = exp$
$\quad \mid intExp \; intComp \; intExp$
$\quad \mid \textbf{not} \; formula \mid formula \; \textbf{and} \; formula$
$\quad \mid formula \; \textbf{or} \; formula$
$\quad \mid \textbf{all} \; var : exp \mid formula$
$\quad \mid \textbf{some} \; var : exp \mid formula$

$intOp ::= + \mid -$
$intComp ::= < \mid > \mid =$
$type ::=$ identifier $\mid \textbf{Int}$
$rel ::=$ identifier
$var ::=$ identifier

Figure 2.1: Abstract syntax for the core Alloy logic

The relation declarations introduce global relations (called fields of signatures) which represent sets of n-ary tuples depending of the relation arity. That is,

$$sig \; A\{r : B \; m \rightarrow n \; C\}$$

declares a ternary relation named $r \subseteq A \times B \times C$ with the restriction that for each $a \in A$ the binary relation $a.r$ maps each tuple in B to n tuples of C, and each tuple in C to n tuples of B. Note that the expression B and C can be arbitrary relational expressions, and that the multiplicity keyword m and n are restricted to $\{lone, some, one, set\}$ (see Figure 2.1).

The fact formulas are usually used to express (further) global —assertion (respectively predicate) independent— constraints on the declared types and relations. The assertion (respectively predicate) formula express a property about the so far defined

system. The goal of the analysis is to check whether the property holds in the system, within a given scope.

2.1.2 Expressions

Alloy expressions represent the basic buildings blocks of Alloy formula; they always evaluate to relations[2]. Basic Alloy expressions are constant relations, this includes all declared signatures and relations as well as the built-in constants: *none* for the unary empty set, *univ* for atom set, and $iden \subseteq univ \times univ$ for the identity relation. Complex expressions are built, recursively, from basic expressions using Alloy's relational operators. This are: r + s, r++ s, r & s, r - s for union, override, intersection, and difference of same arity relations r and s, respectively; r.s, r→s for Cartesian product, and relational join of arbitrary relations r and s, respectively; ~r, ^r for transposition, and transitive closure of a binary relation *r*, respectively; s <: r for domain restriction of an arbitrary relation r to a set (unary relation) s. The semantics of this operators, except of domain restriction, is shown in Figure 5.4. A domain restriction application of the form s<: r where r is an *n*-ary relation can be, however, desugared to $(s \rightarrow \underbrace{\textbf{univ} \rightarrow ... \rightarrow \textbf{univ}}_{n-1}) \,\&\, r$.

Integer expressions denote primitive integers. The built-in type Int represents the set of all atoms carrying primitive integers. The expression **Int** ie denotes the atom carrying the integer denoted by the integer expression ie, whereas **int** v denotes the integer value of the atom represented by the variable v. Integer expressions are obtained from constant numbers (..., -1, 0, 1, ...), and set cardinality expressions #exp where exp is an arbitrary relational expression, and combined using arithmetic operators (+ , -). The arithmetic operators are distinguished from relational operators using the type information[3].

2.1.3 Multiplicity Constraints

Alloy supports the following multiplicity keywords:

- **set**: any number
- **one**: exactly one
- **lone**: at most one
- **some**: at lest one

Depending on where multiplicity keywords are placed they induce different constraints. If applied outside declarations, like in **lone** r, it constraints the relation r to have at most one tuple —with other words $|r| \leq 1$. If applied within declarations, like

[2]Since version 4.2, integer expressions can also be seen as relational expressions (cf. [48, page 82])
[3]Since version 4.2, the distinction between relational and arithmetic operators is made syntactically.

in x: **lone** r, it constrains the variable x to be a subset of r that contains at most one tuple. If applied within relation declarations, like r: A →B **some** →**one** C, it constrains for each a: A the expression a.r to associate each tuple in B with exactly one tuple in C and each tuple in C with at least one tuple in B. The default multiplicity keyword for unary relations is **one** and for multiple-arity relations is **set**.

2.1.4 Formulas

Basic Alloy formulas are formed from Alloy expressions using the subset operator **in**, the equality operator = and the integer comparison operators less than <, greater than >, and the integer equality =. Basic formulas can be combined using (usual) logical connectivities including conjunction (**and** or &&), disjunction (**or** or ||), implication (**implies** or ⇒), and negation (**not** or !). Complex Alloy formulas are build using quantifiers. Quantified Alloy formulas take the form Q x: exp | F where Q is one of the Alloy quantifier: **all**, **some**, **no**, **one**, **lone**, x a variable (usually) occurring in F and bounded by the Alloy expression exp. The semantics of the standard **all** and **some** quantifiers are shown in Figure 5.4. All other quantifiers can be desugared using standard quantifiers as shown in Table 2.1.

no x: exp | F	≡	**all** x: exp | !F
one x: exp | F	≡	**some** x: exp | F **and** (**all** y: exp | y != x ⇒ !F)
lone x: exp | F	≡	**some** x: exp | F **and** (**all** y: exp | y != x ⇒ !F) **or** **all** x: exp | !F

Table 2.1: Desugaring of non standard Alloy quantifiers

2.1.5 Analysis

The Alloy Analyzer —the original analysis tool for Alloy— provides two major analysis. The first (called predicate-running) analysis is applied to Alloy problems with a predicate and results in showing, if exists, satisfying structures (called *instances*) that satisfies the predicate together and the *Alloy model* —the Alloy problem without its predicate (respectively assertion). The second analysis form (called assertion-checking) is applied to Alloy problems with an assertion and results in showing, if exists, a structure (called *counterexample* (CE)) that falsifies the implication of the assertion from the Alloy model.

However, the Alloy Analyzer, reduces both analysis forms to the same analysis problem, namely, finding a satisfying structure (an assignment of the problem relations to values) that makes a given Alloy formula (also called constraint) true. Having such a tool, one can serve (1) the predicate-running analysis by analysing the conjunction of all formula induced by the Alloy model and the predicate formula, and (2) the assertion-checking analysis by analysing the conjunction of all formula induced by the Alloy model and the negation of assertion formula.

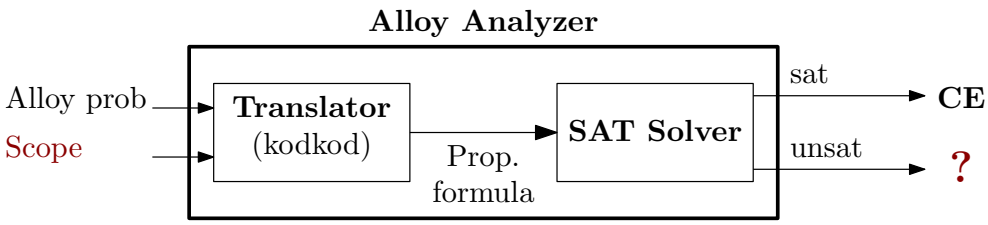


Figure 2.2: An abstract view on the Alloy analysis process of checking an assertion

To overcome the undecidability of the Alloy logic, the Alloy Analyzer performs its analysis with respect to a finite scope —a user-provided bound on the size of Alloy problem types. Using this scope the Alloy Analyzer translate Alloy problem —via kodkod— to an, within the scope, equisatisfiable propositional formula and uses a SAT solver solve it. Consequently, Alloy Analyzer can never prove the correctness of an Alloy assertion. Figure 2.2, shows the internal Alloy analysis process of checking an assertion.

2.2 SMT

Satisfiability modulo theories (SMT) refers to an extension of the well known problem of determining the satisfiability of propositional formulas. However, in the SMT case the formulas are formulated in typed first-order logic with equality and their satisfiability is considered with respect to (aka. modulo) a set of logical background theories, which usually restrict the interpretation of symbols used in the formula.

The support of background theories, especially of non finitely axiomatizable theories like the integer theory, makes SMT solvers —solvers of the SMT problem— superior to the well known automatic theorem provers (ATP) [62], which only support pure first-order logic formulas. In addition, their built-in capability of providing both proofs and counterexamples makes them especially attractive for the verification of software systems.

2.2.1 SML-LIB Language

Since its inception in 2003, the SMT-LIB initiative [2] has provided, among others, a common input and output languages for SMT solvers —supported by most solvers. We call the SMT-LIB language, shortly, SMT language, and a set of SMT declarations and formulas an SMT benchmark.

The SMT language is based on a many sorted first-order logic with equality and flat type system —no explicit subtyping. Their expressions are a sublanguage of Common Lisp's S-expressions. In addition to built-in sorts like `Bool, Int, and Real`, the language allows for the declaration of new uninterpreted sorts and functions.

The expression `(declare-sort S 0)` declares a new simple uninterpreted toplevel sort named S. The expression `(declare-fun f (S1 S2) S3)` declare the new uninterpreted total function $f : S_1 \times S_2 \to S_3$. The expression `(assert F)` adds the SMT formula F to the SMT benchmark and guaranties that each found structure of the benchmark satisfies F. Basic SMT formulas are function applications and can be combined using the usual logical connectivities `and, or, not, =>`. Universal and existentially quantified formulas are denoted by `(forall ((x S1) (y S2)) F)` and `(exists ((x S1) (y S2)) F)` respectively[4].

2.2.2 Analysis

Given an SMT benchmark composed of all sorts and functions declarations, and a set of SMT formulas, the SMT language command `(check-sat)` initiates the satisfiability check of the SMT benchmark. If the SMT benchmark is satisfiable, the SMT language command `(get-model)` can be used to provide a concrete satisfying structure of it. In order to ease the evaluation of a specific term t in a satisfiable SMT benchmark, the command `(get-value t)` is provided.

2.3 KeY

The KeY System [12] is a tool that integrates design, implementation, formal specification, and formal verification of Java software. In order to formulate and reason about the semantics of Java programs, it uses the *Java dynamic logic (JavaDL)* framework [11], an extension of first-order logic with *dynamic logic* operators [44]. We refers to the last version of JavaDL which support explicit heaps [77]. The reasoning in KeY, which is actually also a deductive theorem prover, is based on a sequent calculus for JavaDL which we do not further discuss in this introduction. Regarding specifications, KeY supports both the *Java Modeling Language (JML)* [53] and the *Object Constraint Language (OCL)* [60].

2.3.1 Syntax

JavaDL is a *multimodal* typed first-order logic. Besides standard typed first-order logic terms, JavaDL provides a Java program Prg, a set of program variables PV and three modal operators: box $[\pi]$, diamond $\langle\pi\rangle$ and update $\{U\}$, where π is a program fragment in the context of Prg (shortly denoted $\pi \in Prg$) and U an update term. Definition 1 shows the JavaDL signature Σ, Definition 2 the JavaDL set of terms Trm_Σ, Definition 3 the set of JavaDL updates Upd_Σ, and Definition 4 the set of JavaDL formulas Frm_Σ —all derived form [77].

[4] Only some of the SMT solvers do support quantified formulas. These are for instance CVC4, Yices and Z3

Definition 1 (JavaDL Signature)**.** A JavaDL signature Σ is a tuple $(\mathcal{T}, \preceq, \mathcal{V}, \mathcal{PV}, \mathcal{F}, \mathcal{F}_u, \mathcal{P}, \alpha, Prg)$ consisting of:

- a set $\mathcal{T}$ of types
- a partial order $\preceq$ on $\mathcal{T}$
- a set $\mathcal{V}$ of variables
- a set $\mathcal{PV}$ of program variables
- a set $\mathcal{F}$ of function symbols
- a set $\mathcal{F}_u \subseteq \mathcal{F}$ of unique function symbols
- a set $\mathcal{P}$ of predicate symbols
- a (static) typing function $\alpha : \mathcal{V} \cup \mathcal{PV} \cup \mathcal{F} \cup \mathcal{P} \to \mathcal{T}^*$ (where $\mathcal{T}^*$ denotes an arbitrary long Cartesian product of $\mathcal{T}$s) such that v is an instance of T (denoted by $v \sqsubseteq T$) for any $v \in \mathcal{V} \cup \mathcal{PV}$ with $\alpha(v) = T$, $f : T_1 \times \cdots \times T_{n-1} \to T_n$ for any $f \in \mathcal{F}$ with $\alpha(f) = (T_1, \ldots, T_n)$, and $P \subseteq T_1 \times \cdots \times T_n$ for any $P \in \mathcal{P}$ with $\alpha(P) = (T_1, \ldots, T_n)$
- a Java program Prg —a set of Java classes and interfaces

Definition 2 (JavaDL Terms)**.** Given a JavaDL signature Σ and a type $A \in \mathcal{T}$ different than boolean, the set Trm_Σ^A of JavaDL terms of type A is defined inductively as follow:

- v is in Trm_Σ^A if $v \in \mathcal{V} \cup \mathcal{PV}$ with $\alpha(v) = A$
- f is in Trm_Σ^A if $f \in \mathcal{F}$ with $\alpha(f) = A$
- $f(t_1, \ldots, t_n)$ is in Trm_Σ^A if $f \in \mathcal{F}$ with $\alpha(f) = (T_1, \ldots, T_n, A)$ and $t_i \in Trm_\Sigma^{T_i'}$ with $T_i' \preceq T_i$ for all $1 \leq i \leq n$
- $if(\varphi)\ then(t_1)\ else(t_2)$ is in Trm_Σ^A if $\varphi \in Frm_\Sigma$, and $t_1, t_2 \in Trm_\Sigma^A$
- $\{U\}t$ is in Trm_Σ^A if $U \in Upd_\Sigma$ and $t \in Trm_\Sigma^A$

The set Trm_Σ of all JavaDL terms is defined as $Trm_\Sigma = \bigcup_{A \in T} Trm_\Sigma^A$.

Definition 3 (JavaDL Updates)**.** Given a JavaDL signature Σ, the set of JavaDL update terms Upd_Σ is defined recursively as follow:

- $v := t$ is in Upd_Σ if $v \in PV$ and $t \in Trm_\Sigma^A$ with $\alpha(v) = A$
- $u_1 || u_2$ is in Upd_Σ if $u_1, u_2 \in Upd_\Sigma$
- $\{u_1\}u_2$ is in Upd_Σ if $u_1, u_2 \in Upd_\Sigma$

The update terms of the form $v := t$ are called elementary and, intuitively, assigns the value of the term t to the program variable v like a side effect Java assignment. Complex updates are build using parallel update terms $u_1||u_2$ and sequential update terms $\{u_1\}u_2$.

Definition 4 (JavaDL Formulas)**.** Given a JavaDL signature Σ, the set Frm_Σ of JavaDL formulas is defined inductively as follow:

- **true**, **false** are in Frm_Σ
- $P(t_1,\ldots,t_n)$ is in Frm_Σ if $P \in \mathcal{P}$ with $\alpha(P) = (T_1,\ldots,T_n)$ and $t_i \in Trm_\Sigma^{T_i'}$ with $T_i' \preceq T_i$ for all $1 \leq i \leq n$
- $\neg\varphi$, $\varphi \wedge \varphi'$, $\varphi \vee \varphi'$, $\varphi \rightarrow \varphi'$, $\varphi \leftrightarrow \varphi'$ are in Frm_Σ if $\varphi, \varphi' \in Frm_\Sigma$
- $\forall Ax;\ \varphi$, $\exists Ax;\ \varphi$ are in Frm_Σ if $A \in \mathcal{T}$, $x \in \mathcal{V}$ and $\varphi \in Frm_\Sigma$
- $\{U\}\varphi$ is in Frm_Σ if $u \in Upd_\Sigma$ and $\varphi \in Frm_\Sigma$
- $[\pi]\varphi$, $\langle\pi\rangle\varphi$ are in Frm_Σ if $\pi \in Prg$ and $\varphi \in Frm_\Sigma$

Chapter 3

AlloyPF —an Example

This chapter gives an overview of the automatic proof process in AlloyPF. The details of the individual techniques used in the proof process are described in the next four chapters. In this chapter, however, we will briefly explain, based on an example, how the actual proof process works.

3.1 Example

Figure 3.2 shows an Alloy specification of a generic file system. The specification consists of a type hierarchy, relations between the types, implicit constraints introduced by keywords, and explicit constraints introduced by fact formulas. As one can see, the syntax of Alloy is very similar to that of object-oriented programming languages: types are introduced by the class-like signatures, relations by fields of signatures.

The type hierarchy of our Alloy specification consists of the two toplevel types *Object* for file system objects (Line 1) and *DirEntry* for directory entries (Line 12). The first toplevel type *Object* is labeled with the keyword **abstract** which constraints each element of this type to belong to one of its extending subtypes *File* for files (Line 3) and *Dir* for directories (Line 5). Types are extended in Alloy using the keyword **extends**. The second toplevel type *DirEntry* is neither abstract nor extended. The last type in this type hierarchy is *Root* (Line 10). It extends the type *Dir* and is represented as singleton using the keyword **one**. Singleton sets can be and are often used in Alloy specifications as aliases of their unique element.

Based on the introduced types and the Alloy multiplicity constraints **set**, **lone**, and **one**, the specification introduces: (1) the binary relations $entries \subseteq Dir \times DirEntry$ which maps each directory to a set of directory entries (Line 6), (2) the binary relation $parent \subseteq Dir \times Dir$ which maps each directory to at most one parent directory (Line 7), and (3) the binary relation $contents \subseteq DirEntry \times Object$ which maps each directory entry to exactly one[1] file system object (Line 13). Figure 3.1 shows the corresponding class diagram of the Alloy file system specification.

[1] The default multiplicity constraint for binary relations is **one**.

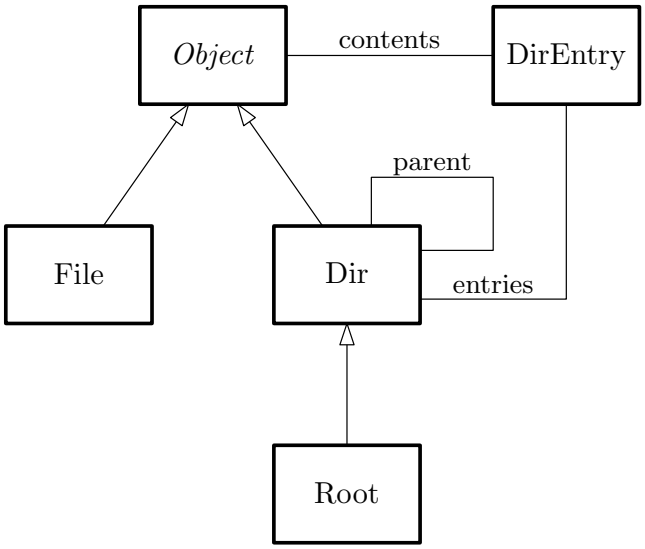


Figure 3.1: A corresponding class diagram of the Alloy file system specification

In addition to the above discussed implicit constraints, the specification introduces five explicit general constraints as fact formulas. Facts in Alloy specifications are comparable to class invariants in object-oriented programming languages. The first fact formula (Line 17) constraints each file element to be a content of at least one directory. The second fact formula (Line 18) constraints the set of parent directories of a directory d to be equal to the set of all directories which have a directory entry that has d as content. The third fact formula (Line 19) constraints the singleton set *Root* to be reachable from each directory, except itself, following the parent relation. The fourth fact formula (Line 20) constraints *Root* to not have a parent directory —"**no** e" is equivalent to "$e = \emptyset$". The last fact formula (Line 21) constraints each directory entry to belong to exactly one directory.

The part of an Alloy specification that specifies the actual system only, like in Figure 3.2, is called *model*. Given a Alloy model, the user can specify properties of that model using the **assert** keyword. The Alloy Analyzer allows to check the *bounded-correctness* of an Alloy specification[2], constituted from an Alloy model and an Alloy assertion, by checking the validity of its assertion within scopes. An assertion is valid within a given scope iff it is satisfied in each logical structure of its corresponding model within the given scope.

In order to demonstrate the proof process of AlloyPE, we use two assertions of the file system model that deduce two Alloy specifications belonging to two different classes of complexity.

[2]When speaking of the bounded-correctness of an Alloy specification, we always assume the existence of an assertion.

```
1  abstract sig Object {}
2
3  sig File extends Object {}
4
5  sig Dir extends Object {
6    entries: set DirEntry,
7    parent: lone Dir
8  }
9
10 one sig Root extends Dir {}
11
12 sig DirEntry {
13   contents: Object
14 }
15
16 fact {
17   all f: File | some d: Dir | f in d.entries.contents
18   all d: Dir | d.parent = d.~contents.~entries
19   all d: Dir | d != Root ⇒ Root in d.^parent
20   no Root.parent
21   all de: DirEntry | one de.~entries
22 }
```

Figure 3.2: An Alloy specification of a generic file system

The first assertion below asserts that if each directory has exactly one parent directory and is the content of exactly one directory entry, then there are no directory aliases.

```
1  assert noDirAliases {
2    (all d: Dir  - Root | one d.parent and one contents.d)
3    ⇒
4    (all o: Dir | lone o.~contents)
5  }
```

The second assertion below asserts that beside the root directory(s) each directory is a content of at least one directory entry.

```
1  assert someDir {
2    all o: Object  - Root | some contents.o
3  }
```

3.2 The AlloyPF Proof Process

AlloyPF starts its proof process by translating the Alloy specification (model and assertion) to an equisatisfiable RFOL formula F (details of the RFOL logic and the Alloy to RFOL translation can be found in Chapter 4 and Chapter 5, respectively). Using our SufGT technique (see Chapter 7 for details), AlloyPF first checks if there exists a general sufficient maximal bound for F and thus for the Alloy specification. If so, AlloyPF uses the Alloy Analyzer to prove the correctness of the Alloy specification via bounded verification within the computed scope. For both specifications of our assertions *noDirAliases* and *someDir*, no general sufficient maximal bound could be computed, moreover, no individual type admits a sufficient maximal bound. In such a case and in order to gain confidence in the validity of the assertion, AlloyPF applies bounded verification using the Alloy Analyzer for increasingly larger scopes until a time-out (or out-of-memory) is reached. Time-out has been set to 600 seconds. For both specifications of our assertions, the Alloy Analyzer can show their bounded-correctness up to the scope of 56.

Having this confidence in the validity of the assertions, the main automatic proof process of AlloyPF starts —using an engine named AlloyPE. This process is based on the satisfiability modulo theories (SMT) analysis of the RFOL translation of the Alloy specification. Because of the complete axiomatization of relational operators in RFOL, the resulting RFOL formulas of the Alloy to RFOL translation are, in general, too difficult for the analysis with SMT solvers. In order to overcome this limitation, AlloyPE uses our SB^+ simplification technique (see Chapter 6 for details). Since SB^+ does not preserve satisfiability, in general, AlloyPE checks a priori whether the RFOL formulas belong to the completeness fragment of SB^+ or not. Both our assertions do belong.

We will denote the SB^+-simplified RFOL translations of the Alloy specifications of the assertions *noDirAliases* and *someDir*, respectively, F_1 and F_2. In a first try, AlloyPE checks the satisfiability of their negations $\neg F_1$ and $\neg F_2$ using the Z3 SMT solver. Because of the equisatisfiability of the RFOL translation and the SB^+ simplification, AlloyPE can directly conclude the validity of the corresponding assertion if the SMT solver reports "unsat", and its invalidity if the SMT solver reports "sat"[3].

Using this process, AlloyPE proves the validity of the first assertion *noDirAliases* in 0.19 seconds but times out (after 600 seconds) trying to prove the validity of the second assertion *someDir*. These results might seem odd at the first glance, since the first assertion looks (syntactically) more complicated than the second one. The reason of this behavior, however, goes back to a general limitation of SMT solvers in handling quantified formulas. That is, an SMT solver can only refute a formula containing quantifiers if it can construct, using quantifier instantiations, a *ground* subformula of the original formula that is by itself refutable. This applies for $\neg F_1$ but not for $\neg F_2$. With respect to RFOL formulas resulting from the Alloy to RFOL translation, this is

[3]We treat "unkown" reports of the SMT solver as time-out.

due to a large extent to the use of the so called recursive theories. These are in Alloy: transitive closure, set cardinality, ordering and integer.

For such formulas, for which the first proof attempt times out, AlloyPE applies in a second attempt our TCPInv calculus (see Chapter 8 for details). The TCPInv calculus can automatically refute RFOL formulas which can not be refuted by standard SMT solving, and uses as only recursive theory the transitive closure theory. It first detects all essential relational paths, i.e., positive literals containing transitive closure, whose refutation is essential for the refutation of the overall formula. For each essential relational path, TCPInv detects and injects to the original formula so called path-invariants until the essential relational path can be refuted from the SMT solver via standard SMT solving. The overall formula gets refuted, when all essential relational paths are refuted.

Applying this technique to our second assertion *someDir*, AlloyPE proves its validity in 16.31 seconds. Out of 6 relational paths, only one was essential. In order to refute this unique essential relational path, AlloyPE needs to detect and inject 4 path-invariants. AlloyPE checks in total 14 path-invariant candidates.

Chapter 4

A First-Order Relational Logic

In order to prove the correctness of Alloy proof obligations, we translate them to a first-order relational logic (RFOL). We designed RFOL with two objectives: (1) it should express the core Alloy language almost directly[1], in order to ease the correctness argumentation, and (2) it should fit in a target logic of satisfiability modulo theories (SMT) solvers, in order to exploit their reasoning power. We first start by fixing the following basic mutually disjoint sets:

- An infinite set $\mathcal{S}$ of sort symbols s.
- An infinite set $\mathcal{X}$ of variables x.
- An infinite set $\mathcal{F}$ of function symbols f.

4.1 Sorts

In RFOL—a many-sorted language— each term is associated with a unique sort, and each sort is denoted by a sort symbol. The sorts in RFOL are neither composed nor parametrized. Also, the sort hierarchy is flat, i.e., all sorts are mutually disjoint. Consequently, a sort signature Ω of RFOL is as simple as a subset S of $\mathcal{S}$.

4.2 Terms

Having a sort system Ω, we can proceed with the definition of the terms of RFOL's terms. RFOL considers, in fact, only *well-sorted* terms, i.e., each term is associated with a unique sort. We will, contrary to the SMT standard [9], allow only for well-sorted terms, and thus avoid the need of additional *well-sortedness* rules and restrictions (cf. [9, page 47]).

Definition 5 (Term signature)**.** A (term) signature Σ is a tuple $(\Omega_\Sigma, \mathcal{Q}_\Sigma, \mathcal{F}_\Sigma, \alpha_\Sigma)$ consisting of

[1] by providing for each core Alloy operator an RFOL counterpart

- a sort signature Ω_Σ containing the sort *Bool*,
- a set $\mathcal{Q}_\Sigma = \{\exists, \forall\}$ of variable binders,
- a set $\mathcal{F}_\Sigma \subseteq \mathcal{F}$ of function symbols containing the equality symbol $=$ as well as the logical connectivities (*true*,*false*, $\neg, \wedge, \vee, \rightarrow$), and
- a sorting function $\alpha_\Sigma : \mathcal{X} \cup \mathcal{F} \rightarrow Sort(\Omega)^* \times Sort(\Omega)$.

The sorting function α determines for variables their sorts and for functions the sorts of their arguments (if they have) as well as their return sort. Function symbols that do not take any argument are called constants and their set is denoted by *Const*. We use notation $f : T_1 \times \cdots \times T_{n-1} \rightarrow T_n$ to denote a function $f \in \mathcal{F}$ with $\alpha(f) = (T_1, \ldots, T_n)$. If $T_n = Bool$, the function f is also called a predicate and the notation $f \subseteq T_1 \times \cdots \times T_{n-1}$ can be used alternatively.

We further assume that for each sort $T \in Sort(\Omega)$, there exists infinitely many variables of sort T denoted by $\mathcal{X}^T$. Variable sets are mutually disjoint and their union is $\mathcal{X}$.

Definition 6 (Terms)**.** Let $\Sigma = (\Omega_\Sigma, Q_\Sigma, F_\Sigma, \alpha_\Sigma)$ be a (term) signature and $T \in Sort(\Omega)$ a sort. Then $Term_\Sigma^T$, the set of all terms of sort T, is defined inductively as follows:

- if $x \in \mathcal{X}^T$, then x is in $Term_\Sigma^T$
- if $f \in \mathcal{F}_\Sigma$ with $\alpha(f) = T$, then f is in $Term_\Sigma^T$
- if $f \in \mathcal{F}_\Sigma$ with $\alpha(f)_{|\alpha(f)|} = T$ and $t_i \in Term_\Sigma^{\alpha(f)_i}$ for $1 \leq i \leq |\alpha(f)| - 1$, then $f(t_1, \ldots, t_{|\alpha(f)|-1})$ is in $Term_\Sigma^T$
- if $Q \in \mathcal{Q}_\Sigma$, $\varphi \in Term_\Sigma^{Bool}$ and $x \in \mathcal{X}^{T'}$ for some sort T', then $Qx : T'.\ \varphi$ is in $Term_\Sigma^{Bool}$

In RFOL only closed terms are considered. This restriction, however, does not imply any loss of generality, as far as satisfiability is concerned. A term φ with free variables $x_1, \ldots, x_n$ of respective sorts $T_1, \ldots, T_n$ can be equisatisfiably rewritten as $\exists x_1 : T_1, \ldots, x_n : T_n.\ \varphi$.

In the following we may drop the index Σ form the signature structure if it is clear from the context. We also write, henceforth, $t_{i:j}$ as shorthand for the (term) list $t_i, \ldots, t_j$, and T^i as shorthand for the i-th Cartesian product of the set T with itself.

Definition 7 (α_Σ-extention to terms)**.** Given a signature Σ, we extend the sorting function α_Σ to the set of terms $Term_\Sigma$ inductively as follows:

- $\alpha_\Sigma(x) = T$ for all $x \in \mathcal{X}^T$
- $\alpha_\Sigma(f(t_1, \ldots, t_n)) = \alpha(f)_{|\alpha(f)|}$ for all $f \in \mathcal{F}_\Sigma$

- $\alpha_\Sigma(Qx:T.\ \varphi) = \alpha_\Sigma(\varphi)$ for all $Q \in \mathcal{Q}_\Sigma$

As usual, we call terms without variables *ground terms* and denote their set by $Gr \subseteq Term$. The set $Gr(t)$ denotes all the ground terms occurring as subterms in a term t. We write $t[x_{1:n}]$ to denote that the variables $x_{1:n}$ occur in the term t. For a term $t \in Term$, a variable x and a ground term gt, the expression $t[gt/x]$ substitutes gt for all the occurrences of x in t. We apply substitutions (aka. instantiations) also to finite sets S of ground terms as $t[S/x] := \{t[gt/x] \mid gt \in S\}$. The Herbrand universe $\mathcal{H}(t)$ of a formula t is the set of all ground terms built from t. That is, all constants occurring in t, are in $\mathcal{H}(t)$, and for each function f occurring in t and $gt_1,\ldots,gt_{|\alpha(f)|-1} \in \mathcal{H}(t)$, $f(gt_1,\ldots,gt_{|\alpha(f)|-1}) \in \mathcal{H}(t)$.

4.3 Semantics

The semantics of RFOL is essentially the same as that of conventional many-sorted logics. Contrary to unsorted logics, the universe of the considered structures is partitioned with respect to sorts of the underlying sort signature.

Definition 8 (Σ-structure)**.** Let Σ be a signature as in Definition 5. Then, a Σ-structure $\mathcal{M}$ (also called Σ-model or just model) is a tuple $(\bar{M}, M)$ where

- $\bar{M}$ is a class of universes defined as $\{\bar{M}^T \mid T \in Sort(\Omega)\}$ where $\bar{M}^T$ denotes the universe of all semantical values of sort T, and
- M is an interpretation that maps every function symbol $f : T_1 \times \cdots \times T_{|\alpha(f)|-1} \to T_{|\alpha(f)|}$ in $\mathcal{F}_\Sigma$ with $|\alpha(f)| > 1$ to an interpretation $M(f) : \bar{M}^{T_1} \times \cdots \times \bar{M}^{|\alpha(f)|-1} \to \bar{M}^{|\alpha(f)|}$. If $|\alpha(f)| = 1$, then $M(f) \in \bar{M}^{\alpha(f)_1}$.

We also require that an RFOL structure satisfies the following:

- $M(=) = \{(v,v) \mid v \in \bar{M}\}$
- $\bar{M}^{Bool} = \{tt, ff\}$
- $M(true) = tt$, $M(false) = ff$
- $M(\neg) = \{ff\}$
- $M(\wedge) = \{(tt,tt)\}$
- $M(\vee) = \{(v_1,v_2) \mid v_1 = tt \text{ or } v_2 = tt\}$
- $M(\to) = \{(v_1,v_2) \mid v_1 = ff \text{ or } v_2 = tt\}$

In order to be able to define the *values* of terms, we still need to fix the interpretation of their variables. To do so, *variable assignment*s are used. A variable assignment is a function $\beta : \mathcal{X} \to \bar{M}$ which assigns to each variable a semantical value. We restrict variable assignment to be *sort compatible*, i.e., the semantical value of a variable x of sort T must be in $\bar{M}^T$. The semantic values of variables are not fixed in the structure as variables may be *universally* bound. We denote by $\beta[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ (or $\beta^{v_1 \ldots v_n}_{x_1 \ldots x_n}$) the variable assignment that maps each variable x_i to v_i and is otherwise identical to β.

Definition 9 (Term value (semantics))**.** Given a Σ-structure $\mathcal{M} = (\bar{M}, M)$, and a variable assignment $\beta : \mathcal{X} \to \bar{M}$, we assign to each terms $t \in Term^T_\Sigma$ a value $val_{\mathcal{M},\beta}(t)$ in $\bar{M}^T$. The evaluation function $val_{\mathcal{M},\beta} : Term_\Sigma \to \bar{M}$ is defined inductively as follows:

1. $val_{\mathcal{M},\beta}(x) = \beta(x)$
2. $val_{\mathcal{M},\beta}(f(t_{1:n})) = M(f)(val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_n))$
3. $val_{\mathcal{M},\beta}(\forall x : T.\ \varphi) = tt$ iff $val_{\mathcal{M},\beta[x \mapsto v]}(\varphi) = tt$ for all $v \in \bar{M}^T$
4. $val_{\mathcal{M},\beta}(\exists x : T.\ \varphi) = tt$ iff $val_{\mathcal{M},\beta[x \mapsto v]}(\varphi) = tt$ for some $v \in \bar{M}^T$

For ground terms gt we may simply use $M(gt)$ to denote $val_{\mathcal{M},\beta}(gt)$ —since is variable free and thus idependent from β. Also, for shotness, we may use for a set S of ground terms $M(S)$ to denote the set $\{M(gt) \mid gt \in S\}$ of values.

Before using the evaluation $val_{\mathcal{M},\beta}$ to introduce the satisfiability of formulas, we first show that it is sort compatible.

Theorem 1 (Sort compatibility)**.** *Given a Σ-structure $\mathcal{M}$ and a variable assignment $\beta : \mathcal{X} \to \bar{M}$, the evaluation function $val_{\mathcal{M},\beta}$ is sort compatible, i.e., for all $t \in Term_\Sigma$, $val_{\mathcal{M},\beta}(t) \in \bar{M}^{\alpha(t)}$.*

Proof. We prove the claim by induction over the term construction.

If t is a variable, the claim follows directly from the (required) sort compatibility of variable assignments.

If $t := f(t_{1:n})$, it follows from Definition 9 that

$$val_{\mathcal{M},\beta}(f(t_{1:n})) = M(f)(val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_n)).$$

Because of the induction hypothesis, all t_i fulfill the claim, i.e., $val_{\mathcal{M},\beta}(t_i) \in \bar{M}^{\alpha(t_i)}$. Thus $M(f)(val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_n))$ is well-defined and is element of $\bar{M}^{\alpha(f)_{|\alpha(f)|}}$. Finally, the claim follows for this case from Definition 7 which says $\alpha(f(t_{1:n})) = \alpha(f)_{|\alpha(f)|}$.

If $t := Qx : T.\ \varphi$ with $Q \in \mathcal{Q}_\Sigma$, it follows from Definition 9 that $val_{\mathcal{M},\beta}(t) = val_{\mathcal{M},\beta}(\varphi)$. Form Definition 6 we additionally know that $\alpha(\varphi) = Bool$. The claim holds by induction hypothesis which says that $val_{\mathcal{M},\beta}(\varphi) \in \bar{M}^{Bool}$. □

4.4 Formulas

Unlike traditional many-sorted logics, RFOL has no distinct *syntactical category* for formulas. Formulas are in the RFOL framework terms of the distinct sort *Bool* — $Term_{\Sigma}^{Bool}$. However, we will, as usual, call boolean sorted terms formulas and use the following usual convention and notation for them.

Definition 10 (Satisfiability and validity of formulas)**.** For formulas $\varphi, \varphi' \in Term_{\Sigma}^{Bool}$ we usually write

- $\mathcal{M}, \beta \models \varphi$ instead of $val_{\mathcal{M},\beta}(\varphi) = tt$ and read φ is *satisfied* by $(\mathcal{M}, \beta)$,
- $\mathcal{M} \models \varphi$ if $(\mathcal{M}, \beta) \models \varphi$ for every variable assignment β,
- $\models \varphi$ if $(\mathcal{M}, \beta) \models \varphi$ for every Σ-structure $\mathcal{M}$ and read φ is *logically valid*, and
- $\varphi' \models \varphi$ if every model that satisfies φ' satisfies φ as well, and read φ' entails φ .

Definition 11 (Formula structure)**.** A literal is an atomic formula or a negated atomic formula. A clause is a disjunction of literals. A formula is in *clause normal form* (*CNF*) if it is a conjunction $(C_1 \wedge \cdots \wedge C_n)$ of clauses where all C_i are quantifier-free and all variables are implicitly universally quantified.

We assume, unless stated otherwise, that all considered formulas are in *CNF* and all variables are unique. When required, we refer to clauses and *CNF*s as sets of literals and clauses, respectively.

4.5 Non Trivial FOL Extensions via Satisfiability Modulo Theories

Up to this point, RFOL is still a pure first-order logic. However, it is well-known that the transitive closure, for example, is not axiomatizable in pure first-order logic [50]. Moreover, adding transitive closure even to very tame logics makes them undecidable [46]. In order to bridge this gap but yet conserve the first-order nature of our RFOL framework, we use the concept of satisfiability modulo theories (SMT), especially with the integer theory.

Definition 12 (Theory)**.** Given a signature Σ, a theory $\mathcal{T}$ is a set of deductively closed formulas. The theory of the deductive closure of a set of fomulas $\mathcal{A}$ is denoted by $Cl(\mathcal{A})$. A class ω of Σ-structures induces a theory $Th(w)$, namely the theory of all formulas φ where $\mathcal{M} \models \varphi$ and $\mathcal{M} \in \omega$ —the resulting set is deductively closed by definition.

The fact that we allow in Definition 12 the definition of theories by structure classes, enables to support of non-finite theories —theories which has no finite (or recursive)

set of first-order axioms. Indeed, first the support of non-finite theories makes the satisfiability modulo theories framework essential —finite theories can be encoded in the proof obligation.

Definition 13 (Satisfiability modulo theories)**.** Given a Σ-theory $\mathcal{T}$, a formula φ is satisfiable modulo $\mathcal{T}$ iff it is satisfied by one of the structures of $\mathcal{T}$. A formula φ' entails φ modulo, written $\varphi' \models_{\mathcal{T}} \varphi$, iff every structure of $\mathcal{T}$ that satisfies φ' satisfies φ as well.

As a first example, we consider a version of McCarthy's *theory of arrays* [57] with indexes and values of given sorts A and B, respectively, and extensionality ($Array_{A\to B}$). Therefore, we assume in Ω the sort $Array_{A\to B}$ and in $\mathcal{F}_\Sigma$ the function symbols sel : $Array_{A\to B} \times A \to B$ and $sto : Array_{A\to B} \times A \times B \to Array_{A\to B}$. In this case, the theory $Array_{A\to B}$ can be completely defined by the following finite set of axioms:

$$\forall ar : Array_{A\to B}, i : A, v : B.\ sel(sto(ar,i,v),i) = v \tag{4.1}$$

$$\forall ar : Array_{A\to B}, i,j : A, v : B.\ i \neq j \to sel(sto(ar,i,v),j) = sel(ar,j) \tag{4.2}$$

$$\forall ar_1, ar_2 : Array_{A\to B}.\ (\forall i : A.\ sel(ar_1,i) = sel(ar_2,i)) \to ar_1 = ar_2 \tag{4.3}$$

However, it is well known that for other theories like the theory of integers (Int) with usual arithmetic functions ($-,+,*,<,\leq,>.\geq$), there is no finite and complete set of (first-order) axioms — even the *first-order version* of the Peano axioms is, because of the induction schema, not finite (cf. [10, page 1133]). Here the satisfiability modulo theories framework comes into play: instead of giving a set of complete axioms for Int we consider the *standard* structure for integers $\mathbb{Z}$.

4.6 Relational Extension

The central objects of a relational theory are relations. In theory, even in a flat sort system, it is possible to define all relational operators based on a general relation sort (see our suggestion for such an extension in Appendix A). However, because of the complexity of this approach and since we are more interested in the analysibility of the translated proof obligations, we decided to go for a less general, but more efficient, approach. For each $1 \leq i \leq N$, where N is the maximal needed relational arity, we introduce an RFOL sort Rel_i. The urelements (usually called atoms in the Alloy context) are represented by the RFOL sort $Atom$. The membership relation between atom tuples and relations is set by the predicates $\in_i \subseteq Atom^i \times Rel_i$, of corresponding arity. Figure 4.1 summarize this *basic* relational extension. We call it *basic*, since it is essential and enough for the definition of all (other) relational operators.

The main characteristic of this approach, in comparison to the one in Appendix A, is that the partitioning of the relations in arity classes is made explicit, thus avoiding all additional axioms needed to deduce it. A drawback of this approach is, however, that we will have to declare and axiomatize all further relational operators separately

$$\Omega \leftarrow \Omega \cup \{Rel_i \mid 1 \leq i \leq N\} \cup \{Atom\}$$
$$\mathcal{F}_\Sigma \leftarrow \mathcal{F}_\Sigma \cup \{\in_i \subseteq Atom^i \times Rel_i \mid 1 \leq i \leq N\}$$

Figure 4.1: Basic relational extension

for each arity class —sometimes for each combination. In practice, however, we will omit the arity specification of the operators, whenever it is clear from context.

4.6.1 Relational Operators

Since Alloy is our target language, we will restrict the enriching of RFOL with relational operators to the *core Alloy* relational operators[2]. We first start with the two relational comparison operators *equality* and *subset*.

$$\mathcal{F}_\Sigma \leftarrow \mathcal{F}_\Sigma \cup \{=_i \subseteq Rel_i \times Rel_i \mid 1 \leq i \leq N\} \cup \{\subseteq_i \subseteq Rel_i \times Rel_i \mid 1 \leq i \leq N\}$$
$$Ax \leftarrow Ax \cup \{\forall R, S : Rel_i.\ R =_i S \leftrightarrow (\forall a_{1:i} : Atom.\ a_{1:i} \in_i R \leftrightarrow a_{1:i} \in_i S) \mid 1 \leq i \leq N\}$$
$$Ax \leftarrow Ax \cup \{\forall R, S : Rel_i.\ R \subseteq_i S \leftrightarrow (\forall a_{1:i} : Atom.\ a_{1:i} \in_i R \rightarrow a_{1:i} \in_i S) \mid 1 \leq i \leq N\}$$

As pointed out in the last section, we introduce for each relational arity an separated operator. Ax represents the set of RFOL axioms defining the semantics of our relational operators. Ax is assumed to be included in any RFOL proof obligation. A nice side effect of this setting, is the implicit modeling of the corresponding global operators[3] as partial functions —which, indeed, they are. Thus we avoid the use of, otherwise need, techniques for handling *undefinedness* (see [43]).

The next category of relational operators we like to enhance RFOL with, are called (by the author) *relational first-order constructors*. We call them constructs, since they construct new relations based on other relations and tuples. Their definitions and axioms are presented in Figure 4.2.

As Figure 4.2 shows that the semantics of all *relational first-order constructors* could be expressed by our basic relational extension —hence the term "first-order". In order to simplify the argumentation about this *relational first-order constructors* at ones, we introduce the following generalization of their axioms. For each n-ary relational first-order constructor $op : (\bigcup_{1 \leq i \leq N} Rel_i)^n \cup Atom \rightarrow \bigcup_{1 \leq i \leq N} Rel_i$ with $ar(op(r_{1:n})) = m$, we write its axiom as follow.

[2] Widely used operators, which can not be (reasonably) desugard by others

[3] This are operators which are declared over the set of all relations, despite their arity. For example the global union operator $\cup : \{R \in Rel_i \mid i \in \mathbb{N}\} \times \{R \in Rel_i \mid i \in \mathbb{N}\} \rightarrow \{R \in Rel_i \mid i \in \mathbb{N}\}$; it is undefined for relations with different arity.

$$Ax_{op} := \forall r_{1:n} : (\bigcup_{1 \leq i \leq N} Rel_i)^n \cup Atom, a_{1:m} : Atom.\ a_{1:m} \in_m op(r_{1:n}) \leftrightarrow LAx_{op} \quad (4.4)$$

The right-hand side of an axiom, LAx_{op}, has two kind of free variables —unbounded variables. The constructor arguments $r_{1:n}$ and the atoms of a result tuple $a_{1:m}$.

The axiomatization of the transitive closure, which is basically a relational constructor, and the set cardinality operator requires, however, the integer theory. Namely, based on the satisfiability modulo theories framework introduced in Section 4.5. The translation of this —non first-order— operators is discussed in the next subsections.

4.6.2 Transitive Closure

The transitive closure operator $^+ : Rel_2 \rightarrow Rel_2$, defined for a binary relation R, constructs the smallest transitive relation that contains R. This very, intuitively, simple concept turned out to be an extremely important and on the same time difficult concept in mathematical reasoning. Avron states in his investigation of transitive closure logics [7] that the extension of first-order logic with transitive closure is the *right* intermediate level between first-order and second-order logic for the formalization and mechanization of mathematics.

Unfortunately, the transitive closure is not axiomatizable in first-order logic: Lev-Ami et al. proved in [56], using the undecidability of the halting problem, that there is no recursively enumerable set of complete first-order axioms for transitive closure; Keller proved that any complete first-oder axiomatization of the transitive closure is not compact which contradicts the compactness property of first-order logic.

In Chapter 8 we present an approach that exploits the first-order refutable fragment transitive closure logics. In this section, however, we are concerned with a complete axiomatization. Therefore, we use the following definition, where $R^{(i)} = \underbrace{R \cdot \ldots \cdot R}_{i+1}$.

$$R^+ = \bigcup_{i \geq 0} R^{(i)}$$

In this definition, the transitive closure is calculated by repeatedly joining its base relation with itself until a fix point is reached. Using this definition, we extend RFOL with a transitive closure operator as shown in Figure 4.3. As shown, among others, in 5.3, our transitive closure extension is correct and complete —with respect to the Alloy semantics— modulo integer theory.

$$\mathcal{F}_\Sigma \leftarrow \mathcal{F}_\Sigma \cup \{^+ : Rel_2 \to Rel_2, ^{()} : Rel_2 \times Int \to Rel_2\}$$
$$Ax \leftarrow Ax \cup \{\forall R : Rel_2, a, b : Atom.\ (a,b) \in_2 R^+ \leftrightarrow (\exists i : Int.\ i \geq 0 \wedge (a,b) \in_2 R^{(i)})\}$$
$$Ax \leftarrow Ax \cup \{\forall R : Rel_2.\ R^{(0)} = R\}$$
$$Ax \leftarrow Ax \cup \{\forall R : Rel_2, i : Int, a, b : Atom.\ i > 0 \to R^{(i)} = R^{(i-1)} \cup_2 R^{(i-1)} \cdot_{2,2} R\}$$

Figure 4.3: Extension of RFOL with the transitive closure operator, modulo integer theory

4.6.3 Integers and Cardinality

Although, in the Alloy context, pure integer expressions are significantly less used than cardinality expressions, one can not consider a one without the other. Fortunately, our satisfiability modulo theories framework allows for an efficient and, at the same time, simple extension of RFOL with integer theory.

$$\Omega \leftarrow \Omega \cup \{Int\}$$
$$\mathcal{F}_\Sigma \leftarrow \mathcal{F}_\Sigma \cup \{+ : Int \times Int \to Int, - : Int \times Int \to Int, * : Int \times Int \to Int\}$$
$$\mathcal{F}_\Sigma \leftarrow \mathcal{F}_\Sigma \cup \{< \subseteq Int \times Int, > \subseteq Int \times Int\}$$

Figure 4.4: Extension of RFOL with integer theory

Figure 4.4 shows the extension. Note that the newly added sort and functions are not axiomatized, and thus, with respect to standard satisfaction, uninterpreted symbols. Their intended meaning is first given by the satisfiability modulo theories framework, namely modulo the theory $\mathbb{Z}$, which interprets Int as the $\mathbb{Z}$ and $+, -, *, <, >$ as their same-named arithmetic operators.

Having the integer theory to our disposal, the axiomatization of the cardinality operator becomes possible and simple. However, since cardinality is only defined for finite sets and our sets are, so far, potentially infinite, a decision about the finiteness intention of sets has to be made.

In the context of the Alloy Analyzer, every relation is finite, even with a priori known maximal cardinality, thus cardinality is always defined for any relational expression. With respect to the Alloy language, however, there is no such restrictions. Since Alloy is in the first line concerned with the specification and analysis of concrete software systems, but also because of theoretical difficulty concerning infinite sets[4], we assume

[4]The power set of infinite sets is not axiomatizable in first-order logic.

that all relations are finite. In the context of this assumption and modulo the integer theory, the RFOL extension with the cardinality operator is shown in Figure 4.5.

$$\mathcal{F}_\Sigma \leftarrow \mathcal{F}_\Sigma \cup \{|\,|_i : Rel_i \rightarrow Int \mid 1 \leq i \leq N\}$$
$$\mathcal{F}_\Sigma \leftarrow \mathcal{F}_\Sigma \cup \{ord_i : Rel_i \times Atom^i \rightarrow Int \mid 1 \leq i \leq N\} \cup \{ordInv_1 : Rel_1 \times Int \rightarrow Atom\}$$
$$Ax \leftarrow Ax \cup \{\forall R : Rel_i, a_{1:i} : Atom.\ (a_{1:i}) \in_i R \rightarrow 1 \leq ord_i(R, a_{1:i}) \leq |R|_i\}$$
$$Ax \leftarrow Ax \cup \{\forall R : Rel_i, n : Int.\ 1 \leq n \leq |R|_i \rightarrow \exists a_{1:i} : Atom.\ (a_{1:i}) \in_i R \wedge ord_i(R, a_{1:i}) = n\}$$
$$Ax \leftarrow Ax \cup \{\forall R : Rel_i, a_{1:i}, b_{1:i} : Atom.$$
$$a_{1:i} \in_i r \wedge b_{1:i} \in_i r \wedge ord_i(R, a_{1:i}) = ord_i(R, b_{1:i}) \rightarrow (a_1 = b_1 \wedge \cdots \wedge a_i = b_i)\}$$
$$Ax \leftarrow Ax \cup \{\forall R : Rel_1, a : Atom.\ a \in_1 r \rightarrow ordInv(R, ord(R, a)) = a\}$$

Figure 4.5: Extension of RFOL with the cardinality operator

$$\mathcal{F}_\Sigma \overset{\cup}{\leftarrow}$$

empty set	$\{\emptyset_i :\rightarrow Rel_i \mid 1 \leq i \leq N\} \cup$
singleton	$\{\{\}_i : Atom^i \rightarrow Rel_i \mid 1 \leq i \leq N\} \cup$
unions	$\{\cup_i : Rel_i \times Rel_i \rightarrow Rel_i \mid 1 \leq i \leq N\} \cup$
intersections	$\{\cap_i : Rel_i \times Rel_i \rightarrow Rel_i \mid 1 \leq i \leq N\} \cup$
differences	$\{\setminus_i : Rel_i \times Rel_i \rightarrow Rel_i \mid 1 \leq i \leq N\} \cup$
overrides	$\{\oplus_i : Rel_i \times Rel_i \rightarrow Rel_i \mid 1 \leq i \leq N\} \cup$
products	$\{\times_{i,j} : Rel_i \times Rel_j \rightarrow Rel_{i+j} \mid 1 \leq i+j \leq N\} \cup$
joins	$\{\fatsemi_{i,j} : Rel_i \times Rel_j \rightarrow Rel_{i+j-2} \mid 1 \leq i+j-2 \leq N\} \cup$
transpose	$\{^{-1}\colon Rel_2 \rightarrow Rel_2\}$

$$
\begin{aligned}
Ax \overset{\cup}{\leftarrow} & \{\forall a_{1:i} : Atom.\ (a_{1:i}) \notin \emptyset_i \mid 1 \leq i \leq N\} \cup \\
& \{\forall a_{1:i}, b_{1:i} : Atom.\ (b_{1:i}) \in_i \{(a_{1:i})\}_i \leftrightarrow b_1 = a_1 \wedge \cdots \wedge b_i = a_i \mid 1 \leq i \leq N\} \cup \\
& \{\forall R, S : Rel_i, a_{1:i} : Atom.\ (a_{1:i}) \in_i R \cup_i S \leftrightarrow (a_{1:i}) \in_i R \vee (a_{1:i}) \in_i S \mid 1 \leq i \leq N\} \cup \\
& \{\forall R, S : Rel_i, a_{1:i} : Atom.\ (a_{1:i}) \in_i R \cap_i S \leftrightarrow (a_{1:i}) \in_i R \wedge (a_{1:i}) \in_i S \mid 1 \leq i \leq N\} \cup \\
& \{\forall R, S : Rel_i, a_{1:i} : Atom.\ (a_{1:i}) \in_i R \setminus_i S \leftrightarrow (a_{1:i}) \in_i R \wedge (a_{1:i}) \notin_i S \mid 1 \leq i \leq N\} \cup \\
& \{\forall R, S : Rel_i, a_{1:i} : Atom. \\
& \quad (a_{1:i}) \in_i R \oplus_i S \leftrightarrow \\
& \quad (a_{1:i}) \in_i S \vee ((a_{1:i}) \in_i R \wedge (\forall b_{2:i} : Atom.\ (a_1, b_{2:i}) \notin_i S)) \mid 1 \leq i \leq N\} \cup \\
& \{\forall R : Rel_i, S : Rel_j, a_{1:i+j} : Atom. \\
& \quad (a_{1:i+j}) \in_{i+j} R \times_{i,j} S \leftrightarrow \\
& \quad (a_{1:i}) \in_i R \wedge (a_{i+1:j}) \in_j S \mid 1 \leq i+j \leq N\} \cup \\
& \{\forall R : Rel_i, S : Rel_j, a_{1:i+j-2} : Atom. \\
& \quad (a_{1:i+j-2}) \in_{i+j-2} R \fatsemi_{i,j} S \leftrightarrow \\
& \quad \exists b : Atom. (a_{1:i-1}, b) \in_i R \wedge (b, a_{i:i+j-2}) \in_j S \mid 1 \leq i+j-2 \leq N\} \cup \\
& \{\forall R : Rel_2, a, b : Atom.\ (a,b) \in_2 R^{-1} \leftrightarrow (b,a) \in_2 R\}
\end{aligned}
$$

Figure 4.2: Relational constructor (except of transitive closure) and their axiomatization using our basic relational extension

Chapter 5

Verifying Alloy Problems

This chapter presents our full automatic approach for verifying Alloy problems. The verification bases on checking the satisfiability of *equisatisfiable* RFOL formulas to the negated proof obligation behind the Alloy problem modulo integer theory. The satisfiability check can be performed by any SMT solver supporting quantifiers and integer theory —we use the Z3 solver [20]. Section 5.1 formalize the Alloy translation to RFOL—basic translation. Section 5.3 discuses correctness and completeness of the translation.

5.1 Translating Alloy to RFOL

5.1.1 Alloy Proof Obligation

Abstractly seen, an Alloy problem consists of (1) an *Alloy model*[1] M describing the structures of the problem —i.e., signatures and fields declarations including their implicit constraints, (2) a set $\mathcal{F}$ of facts, and (3) an assertion A describing some properties about the problem. An Alloy problem is correct if any *Alloy instance* (for short *instance*) that is conform with the model M and satisfies the facts in $\mathcal{F}$, satisfies the assertion A too. An Alloy instance is the equivalent of a semantic structure in RFOL. It maps each Alloy relation symbol (including signatures) and each (bounded) variable[2] to a relation value —a set of tuples of atom values.

In the Alloy Analyzer context, Alloy instances have to additionally fulfill *scope deduced constraints*. These are (1) constraints on the cardinality of signature interpretations and (2) interpretation of Alloy integers as bit-vectors of corresponding bit-width to the scope. For the purpose of verification, however, we modified these constraints such that (1) signatures interpretations can have any, but finite, cardinality and (2) Alloy integers are interpreted as mathematical integers $(\mathbb{Z}, +, -, *, <, >)$.

In order to formulate the formal proof obligation behind an Alloy problem, we further divide the Alloy model M to a set $M_{\mathcal{S}}$ of sorting constraints on the problem

[1] not to be confused with RFOL's semantical structures, also, usually, denoted by $\mathcal{M}$

[2] In Alloy even variables represents relations, namely *singleton sub relations* of their bounding expression.

symbols, and a set $M_{\mathcal{C}}$ of Alloy formulas expressing all remaining[3] model constraints. Let $\mathcal{S}$ be the set of all Alloy instances conform to $M_{\mathcal{S}}$ of an Alloy problem. Then, the proof obligation behind that Alloy problem can be formulated as follow.

$$\models_{Th(\mathcal{S}),\mathbb{Z}} \bigwedge M_{\mathcal{C}} \wedge \bigwedge \mathcal{F} \rightarrow A \tag{5.1}$$

Consequently, an Alloy problem is correct if and only if any Alloy instance that is conform with $M_{\mathcal{S}}$ and interprets the signatures as finite sets of values, the integers as $\mathbb{Z}$, satisfies $\bigwedge M_{\mathcal{C}} \wedge \bigwedge \mathcal{F} \rightarrow A$.

Having this formulation, we inductively define in the next sections a translation function T that takes an Alloy problem, in the form of a triple $(M, \mathcal{F}, A)$, and returns an RFOL problem $(\alpha_{M_{\mathcal{S}}}, C_{M_{\mathcal{C}}}, C_{\mathcal{F}}, C_A)$, where $\alpha_{M_{\mathcal{S}}}$ represents a conform RFOL sorting function to the sorting constraints of $M_{\mathcal{S}}$, $C_{M_{\mathcal{C}}}$ represents the RFOL translation of all model constraints $M_{\mathcal{S}}$, $C_{\mathcal{F}}$ represents the RFOL translation of the facts $\mathcal{F}$, and C_A represents the RFOL translation of the assertion A. The goal of the translation is that

$$\models_{Cl(Ax),\mathbb{Z}} C_{M_{\mathcal{C}}} \wedge C_{\mathcal{F}} \rightarrow C_A \tag{5.2}$$

holds if and only if the corresponding Alloy problem is correct. The notation $(Cl(Ax),\mathbb{Z})$ stands for the theory combination of both theories $Cl(Ax)$ and $\mathbb{Z}$ —i.e., $Th(\{\mathcal{M} \mid \mathcal{M} \text{ is a } \mathbb{Z}\text{-structure} \wedge \mathcal{M} \models Ax\})$. For notation simplicity reasons and since each RFOL structure is implicitly a $\mathbb{Z}$-structure satisfying Ax, we skip indexing the deduction relation with $Cl(Ax)$ or $\mathbb{Z}$, when not need.

5.1.2 Signatures and Fields

The model M of an Alloy problem is defined by its signature and field declarations. In addition to introducing sort and relation symbols, these declarations encode restrictions on M's conform instances.

In our translation rules presented in Figure 5.1, the auxiliary function $\mathcal{D}$ isolates and translates the basic relational constants of each Alloy model declaration. For a declaration of a relation r of arity n, it introduces, regardless from further constraints expressed by the declaration, an RFOL constant of sort Rel_n with the unique name $Na[r]$[4]. All further declaration constraints are captured and translated by the auxiliary function $\mathcal{C}$.

5.1.3 Expressions

Expressions constitute the basic syntactical elements of Alloy formulas —equivalent to $Term_{\Sigma} \setminus Term_{\Sigma}^{Bool}$ in RFOL. They are translated using the auxiliary function $\mathcal{E}$ given

[3]Beside sorting of the problem symbols, all Alloy model constraints are expressible with Alloy formulas.
[4]We assume that all RFOL function symbols $Na[r]$ are already in $\mathcal{F}_{\Sigma}$.

$T : model \times \mathcal{P}(frml) \times frml \to SortFun \times Term_{\Sigma}^{Bool} \times Term_{\Sigma}^{Bool} \times Term_{\Sigma}^{Bool}$
$\mathcal{D} : decl \to SortFun$
$\mathcal{C} : decl \cup frml \to Term_{\Sigma}^{Bool}$
$Na : symbol \to symbol_{RFOL}$

$T[\langle M, \mathcal{F}, A \rangle] = \langle \alpha_{\Sigma} \bigoplus_{d \in Decl(M)} \mathcal{D}[d], \bigwedge_{d \in Decl(M)} \mathcal{C}[d], \bigwedge_{F \in \mathcal{F}} \mathcal{C}[F], \mathcal{C}[A] \rangle$
$\mathcal{D}[\texttt{sig}\ S] = \alpha(Na[S]) = (Rel_1)$
$\mathcal{D}[\texttt{sig}\ S\ (\texttt{in}|\texttt{extends})\ S'] = \alpha(Na[S]) = (Rel_1)$
$\mathcal{D}[r : S_1 \to \ldots \to S_n] = \alpha(Na[r]) = (Rel_n)$
$\mathcal{C}[\texttt{sig}\ S] = \bigwedge_{S'} Na[S] \cap_1 Na[S'] = \emptyset$ for any top-level signature $S' \neq S$
$\mathcal{C}[\texttt{sig}\ S\ \texttt{in}\ S'] = Na[S] \subseteq_1 Na[S']$
$\mathcal{C}[\texttt{sig}\ S\ \texttt{extends}\ S'] = \mathcal{C}[\texttt{sig}\ S\ \texttt{in}\ S'] \wedge$
$\bigwedge_S Na[S] \cap_1 Na[B] = \emptyset$ for any extension B of S' where $B \neq S$
$\mathcal{C}[r : S_1 \to \ldots \to S_n] = Na[r] \subseteq_n Na[S_1] \times_{1,n-1} (\ldots (Na[S_{n-1}] \times_{1,1} Na[S_n]))$

Figure 5.1: Translation rules for Alloy declarations. *SortFun* denotes the sorting functions $\alpha : \mathcal{X} \cup \mathcal{F} \to Sort(\Omega)^* \times Sort(\Omega)$.

in Figure 5.2. In Alloy, there is a distinction between relational expressions and integer expressions. The design of our RFOL makes, especially, the translation of the relational expressions straightforward. Figure 5.2 (top part) gives their translation rules.

However, for integer expressions we deviate from the Alloy's behavior in two aspects: (1) we model the Alloy set of integer values `int` as the infinite set of mathematical integers, (2) we make all implicit casts from the Alloy built-in signature `Int` to integer values explicit and forbid the opposite cast. This design decision goes on the same line, albeit not identical, with the Alloy language update made since version 4.2. Since this update, Alloy supports only the type `Int` —no direct support of integer values, and all integer operators got new distinct symbols. Figure 5.2 (bottom part) shows the translation rules for integer expressions.

The application of Alloy's `sum` operator to a unary integer expression *ie* over a variable binding in the unary expression *e* is translated using the function $sum : Rel_1 \times Rel_1 \times Int \times Int \to Int$. The axiomatization of this function is based on an approach of Leino et al. published in [55]. In this approach, they introduced an efficient first-order axiomatization for comprehensions of the form $Q\{L \leq i \leq H, T\}$ where Q is a function (e.g. sum, min), L and H are the lower and upper bounds on the integer i, and T is an integer term based on i. Alloy's `sum` expressions are computed over general variable bindings —*e* can be an arbitrary unary expression. Thus, no integer bounds are explicitly available. However, using our cardinality axioms, we can formulate Alloy's `sum` operator as $sum\{1 \leq i \leq |e|_1, \mathcal{E}[ie][ordInv_1(\mathcal{E}[e], i)/x]\}$ which makes Leino's axioms and patterns directly applicable.

$$\mathcal{E} : exp \cup intExp \to Term_\Sigma \qquad e : exp, ie : intExp, n : number$$
$$\mathcal{E}[none] = \emptyset_1$$
$$\mathcal{E}[e_1(+ \mid - \mid \texttt{\&} \mid \texttt{++})e_2] = \mathcal{E}[e_1](\cup_i \mid \setminus_i \mid \cap_i \mid \oplus_i)\mathcal{E}[e_2] \qquad \text{if } arity(e_1) = arity(e_2) = i$$
$$\mathcal{E}[e_1 \,.\, e_2] = \mathcal{E}[e_1] \cdot_{i,j} \mathcal{E}[e_2] \qquad \text{if } arity(e_1) = i \text{ and } arity(e_2) = j$$
$$\mathcal{E}[e_1\texttt{->}e_2] = \mathcal{E}[e_1] \times_{i,j} \mathcal{E}[e_2] \qquad \text{if } arity(e_1) = i \text{ and } arity(e_2) = j$$
$$\mathcal{E}[\texttt{\~}e] = \mathcal{E}[e]^{-1}$$
$$\mathcal{E}[\texttt{\^}e] = \mathcal{E}[e]^{+}$$
$$\mathcal{E}[r] = Na[r] \qquad \text{if } r \text{ is a relational symbol}$$
$$\mathcal{E}[v] = Na[v] \qquad \text{if } v \text{ is a variable}$$

$$\mathcal{E}[ie_1 +_{int} ie_2] = \mathcal{E}[ie_1] + \mathcal{E}[ie_2]$$
$$\mathcal{E}[ie_1 -_{int} ie_2] = \mathcal{E}[ie_1] - \mathcal{E}[ie_2]$$
$$\mathcal{E}[ie_1 *_{int} ie_2] = \mathcal{E}[ie_1] * \mathcal{E}[ie_2]$$
$$\mathcal{E}[\texttt{\#}e] = |\mathcal{E}[e]|_i \qquad \text{if } arity(e) = i$$
$$\mathcal{E}[\texttt{sum } x : e \mid ie] = sum(e, ie, 1, |e|_1)$$
$$\mathcal{E}[n] = n$$

Figure 5.2: Translation rules for Alloy expressions.

5.1.4 Formulas

Elementary Alloy formulas are formed by applying multiplicity quantifiers to relational expressions or by comparing expressions. Quantified formulas are formed using universal and existential quantifiers. More complex formulas are formed by combining formulas with logical connectivities. Figure 5.3 gives the rules for translating Alloy formulas to RFOL using the auxiliary function $\mathcal{C}$.

$$\mathcal{C} : formula \to Term^{Bool}_\Sigma \qquad e : exp, ie : intExp, arity(e) = i$$
$$\mathcal{C}[\texttt{lone } e] = \mathcal{C}[\texttt{all } a, b : e \mid a = b]$$
$$\mathcal{C}[e_1 \texttt{ in } e_2] = \mathcal{E}[e_1] \subseteq_i \mathcal{E}[e_2]$$
$$\mathcal{C}[e_1 \texttt{ = } e_2] = \mathcal{E}[e_1] =_i \mathcal{E}[e_2]$$
$$\mathcal{C}[ie_1 \ (\texttt{<} \mid \texttt{>} \mid \texttt{=}) \ ie_2] = \mathcal{E}[ie_1](<|>|=)\mathcal{E}[ie_2]$$
$$\mathcal{C}[\texttt{all } a : e \mid G] = \forall a_{1:i} : Atom.\ (a_{1:i}) \in_i \mathcal{E}[e] \to \mathcal{C}[G][\{(a_{1:i})\}_i / Na[a]]$$
$$\mathcal{C}[\texttt{some } a : e \mid G] = \exists a_{1:i} : Atom.\ (a_{1:i}) \in_i \mathcal{E}[e] \wedge \mathcal{C}[G][\{(a_{1:i})\}_i / Na[a]]$$
$$\mathcal{C}[\texttt{not } G] = \neg\mathcal{C}[G]$$
$$\mathcal{C}[G_1 \texttt{ and } G_2] = \mathcal{C}[G_1] \wedge \mathcal{C}[G_2]$$
$$\mathcal{C}[G_1 \texttt{ or } G_2] = \mathcal{C}[G_1] \vee \mathcal{C}[G_2]$$
$$\mathcal{C}[G_1 \texttt{ implies } G_2] = \mathcal{C}[G_1] \to \mathcal{C}[G_2]$$

Figure 5.3: Translation rules for Alloy formulas.

For an Alloy expression e of arity i, multiplicity formulas of the form `(mult e)` are desugard using standard Alloy quantified formulas where *mult* stands for the

Non principle Alloy expressions	Equivalent reduction
s <: e	(s →**univ** →...→**univ**) & e
some e	**some** x: e | true
one e	**some** e **and lone** e
no x: exp | F	**all** x: exp | !F
one x: exp | F	**some** x: exp | F **and** (**all** y: exp | y != x ⇒ !F)
lone x: exp | F	**some** x: exp | F **and** (**all** y: exp | y != x ⇒ !F) **or** **all** x: exp | !F

Table 5.1: Desugaring non principle Alloy constructors

multiplicities *one*, *some*, and *lone*. Alloy formulas comparing expression —either relational or integer— (`e1 comp e2`) are translated directly using corresponding comparators in RFOL where *comp* stands for *in*, $=$, $<$, and $>$. Quantified Alloy formulas of the form (`Q a: e| G`) are translated to quantified RFOL formulas that bound fresh RFOL variables $a_{1:i}$ to the sort *Atom* and substitute the set $\{(a_{1:i})\}_i$ for a in translation of G. The variable binding inaccuracy is corrected by guarding the consideration of the quantification body G —either using implication for universal quantifiers or a conjunction for existential quantifiers— to only valid bindings —i.e., $(a_{1:i}) \in_i \mathcal{E}[e]$. Finally, negation, conjunction, disjunction and implication in Alloy are mapped to those in RFOL.

5.2 Rewrite of non Principle Alloy Constructors

5.2.1 Core Alloy

Although core Alloy (cf. Section 2.1) focuses on principle Alloy constructors, it contains some non principle constructors —constructors that can be resolved/expressed/rewritten using others (aka. desugaring). Our translation rewrites Alloy expressions and formulas containing such constructors to equivalent ones using principle Alloy constructors before the actual translation. For the override operator, however, which can also be desugared using domain restriction, difference and union, we choose a direct translation and axiomatization in RFOL since its rewrite results in too complex expressions that can affect the efficiency of the analysis. Table 5.1 lists and resume all our rewrite rules for non principle constructors in core Alloy.

5.2.2 Ordering

Alloy provides a bunch of libraries (called modules) that offer complex and often required functionalities, like for graphs, sequences, ordered structures, etc. Most

of these functionalities serve merely the simplification of the modeling process and the conciseness of Alloy expressions, and are desugared during the analysis by the Alloy Analyzer. The ordering module, however, is hardwired into the Alloy Analyzer in order to take advantage of its symmetry breaking capability, which is especially efficient for linked data structures like ordering.

Because of its widely use in Alloy modeling, especially for complex algorithms —it is, in fact, the standard way of modeling trace executions, we were interested in supporting the proof of Alloy assertions involving ordering by our automated reasoning. However, we observed in [75], like for transitive closure, that the standard integer based axiomatization of ordering is not suitable for automation since it requires in most cases integer induction. On the other hand, we could solve a similar problem for transitive closure using our *path-invariant injection* approach (see Chapter 8) to a large extent. In this section, we show how we express functionalities of the ordering module using the transitive closure theory. This allows for a more effective exploitation of all advances we made in the automatic transitive closure reasoning.

Ordering a signature S, in Alloy with the declaration **open** util/ordering[S], defines a total order on S. The order is represented by the (implicit) relations[5]: next: S →S to denote of an element *e* the successor (by e.next) and predecessor (by next.e), first: S to denote the smallest element of the ordering and last: S to denote the largest element of the ordering. Since Alloy interprets signatures as finite sets (w.r.t. to the provided scope), the largest element of the ordering exists and has no successor. We desugar the ordering declaration of S by introducing the *explicit* relations $nextS \subseteq S \times S$ and $firstS \subseteq S$, and constraining them as follow:

one firstS (5.3)

all s: S | **lone** s.nextS (5.4)

all s: S | s !**in** s.^nextS (5.5)

all s: S | (s **=** firstS) **or** (s **in** firstS.^nextS) (5.6)

Thereby, and with respect to a transitive closure theory, we constrain the signature S to be a *strict* total order under $nextS^+$ —an irreflexive total order. The smallest element of this ordering is represented by the unary singleton relation *firstS* (constrained by Equation 5.3). Since we interpret signatures as (possibly) infinite sets, there is no largest element of the ordering. Given an element $s \in S$, its successor element is represented by the unary singleton relational expression s.nextS (constrained by Equation 5.4) and its predecessor by the unary lone relational expression nextS.s (constrained by Equations 5.4, 5.5 and 5.6). One can easily prove using Equations 5.5 and 5.6 that S is a strict total order under $nextS^+$.

Waiving the reflexivity, allows us to simplify our desugaring of the ordering module, by constraining *nextS* to be an acyclic functional relation, while preserving the actually needed properties of the ordering. Contrary to the Alloy documentation, the

[5]Based on this relations, Alloy's ordering module offers further functions and predicates to shorten expressions.

Alloy analyzer (version 4.2) also implements a strict total order. The complete Alloy description of our desugaring of the ordering module is listed in Appendix B.

5.3 Correctness and Completeness

In this section, we turn our attention to the correctness and completeness of our translation of Alloy proof obligation to RFOL, formulated in the previous section. In order to do so, we require formal semantics of Alloy formulas. Semantics of RFOL is formulated in Chapter 4.

5.3.1 Alloy Semantics

One could formulate the Alloy semantics using the same framework as we did for RFOL. However, in order to avoid confusion, we use the framework used in the Alloy book [47, Appendix C][6]. Accordingly, an Alloy formula is evaluated with respect to an Alloy *instance* $\mathcal{I}$ consisting of a set of atom values U and a binding I. The binding $I : relSym \cup var \to \bigcup_{1 \leq i \leq N} \mathcal{P}(U^i)$ assigns to each relational symbol (can be seen as free variables), and to each variable (always bounded via quantifiers) a relation value —a set of tuple of values— of corresponding arity. Figure 5.4 shows the semantics of Alloy formulas.

The meaning of an Alloy formula is given recursively by the auxiliary functions M and E. The function E extends the interpretation of relational symbols and variables to (complex) relational expressions. The function M evaluate formulas with respect to a given instance. A formula is true with respect to a given instance if and only if its expression values given by E satisfies it. Note that due to its bounded analysis, the Alloy Analyzer finitizes all signatures and types including **int**. Hence, it can check problems only with respect to fixed bitwidth integers. In the context of verification, however, integers are usually assumed to be infinite. Accordingly, we fixed the interpretation of the Alloy **int** type and its constants and operators to mirror the mathematical integers —$(\mathbb{Z}, +, -, *, <, >)$.

5.3.2 RFOL Structures Features

In this section we show some features of RFOL structures which are essential for the correctness and completeness discussion of our Alloy translation to RFOL.

The first features state that, because of the axioms Ax, the interpretation of each Rel_i with the relation $\in_i$ is isomorph to the power set of the interpretation of $Atom^i$ with the relation $\in$. In general, power set is not axiomatizable in first-order logic, however, for finite sets it is. Indeed, we show in the next theorem that the following axioms of Ax are sufficient.

[6] beside that we represent relational values as sets of tuple of atoms, instead of abstract relations as in Zermelo–Fraenkel set theory

$$M : formula \times (relSym \cup var \to \bigcup_{1 \le i \le N} \mathcal{P}(U^i)) \to \{tt, ff\}$$
$$E : exp \times (relSym \cup var \to \bigcup_{1 \le i \le N} \mathcal{P}(U^i)) \to \bigcup_{1 \le i \le N} \mathcal{P}(U^i)$$

$M[\textbf{not } \mathrm{f}]I = \neg M[\mathrm{f}]I$
$M[\mathrm{f} \textbf{ and } \mathrm{g}]I = M[\mathrm{f}]I \wedge M[\mathrm{g}]I$
$M[\mathrm{f} \textbf{ or } \mathrm{g}]I = M[\mathrm{f}]I \vee M[\mathrm{g}]I$
$M[\mathrm{f} \Rightarrow \mathrm{g}]I = M[\mathrm{f}]I \to M[\mathrm{g}]I$
$M[\textbf{all } \mathrm{x}\text{: } \mathrm{e} \mid \mathrm{f}]I = \wedge\{M[\mathrm{f}](I \oplus x \mapsto v) \mid v \subseteq E[\mathrm{e}]I \wedge |v| = 1\}$
$M[\textbf{some } \mathrm{x}\text{: } \mathrm{e} \mid \mathrm{f}]I = \vee\{M[\mathrm{f}](I \oplus x \mapsto v) \mid v \subseteq E[\mathrm{e}]I \wedge |v| = 1\}$
$M[\mathrm{e}_1 \textbf{ in } \mathrm{e}_2]I = E[\mathrm{e}_1]I \subseteq E[\mathrm{e}_2]I$
$M[\mathrm{e}_1 = \mathrm{e}_2]I = E[\mathrm{e}_1]I = E[\mathrm{e}_2]I$
$M[\mathrm{ie}_1(< \mid >)\ \mathrm{ie}_2]I = E[\mathrm{ie}_1]I(< \mid >)E[\mathrm{ie}_2]I$

$E[\textbf{none}]I = \emptyset$
$E[\mathrm{e}_1 + \mathrm{e}_2]I = E[\mathrm{e}_1]I \cup E[\mathrm{e}_2]I$
$E[\mathrm{e}_1 \ \& \ \mathrm{e}_2]I = E[\mathrm{e}_1]I \cap E[\mathrm{e}_2]I$
$E[\mathrm{e}_1 - \mathrm{e}_2]I = E[\mathrm{e}_1]I \setminus E[\mathrm{e}_2]I$
$E[\mathrm{e}_1 ++ \mathrm{e}_2]I = \{(a_{1:n}) \mid (a_{1:n}) \in E[\mathrm{e}_2]I \vee ((a_{1:n}) \in E[\mathrm{e}_1]I \wedge (\forall b_{2:n}.\ (a_1, b_{2:n}) \notin E[\mathrm{e}_2]I))\}$
$E[\mathrm{e}_1.\mathrm{e}_2]I = \{(a_{1:n-1}, b_{2:m}) \mid \exists x.\ (a_{1:n-1}, x) \in E[\mathrm{e}_1]I \wedge (x, b_{2:m}) \in E[\mathrm{e}_2]I\}$
$E[\mathrm{e}_1 \to \mathrm{e}_2]I = \{(a_{1:n}, b_{1:m}) \mid (a_{1:n}) \in E[\mathrm{e}_1]I \wedge (b_{1:m}) \in E[\mathrm{e}_2]I\}$
$E[\sim \mathrm{e}]I = \{(a,b) \mid (b,a) \in E[\mathrm{e}]I\}$
$E[\hat{} \mathrm{e}]I = \{(a,b) \mid \exists x_1, \ldots, x_n.\ (a,x_1), (x_1,x_2), \ldots, (x_n,b) \in E[\mathrm{e}]I\}$
$E[\mathrm{n}]I = n$
$E[\mathrm{x}]I = I(x)$
$E[\mathrm{r}]I = I(r)$
$E[\textbf{Atom}]I = U$
$E[\mathrm{ie}_1(+ \mid - \mid *)\ \mathrm{ie}_2]I = E[\mathrm{ie}_1]I(+ \mid - \mid *)E[\mathrm{ie}_2]I$
$E[\#\mathrm{e}]I = |E[\mathrm{e}]I|$
$E[\textbf{sum } \mathrm{x}\text{: } \mathrm{e} \mid \mathrm{ie}]I = \Sigma_{v \in E[\mathrm{e}]I} E[\mathrm{ie}](I \oplus x \mapsto v)$
$E[\textbf{int}]I = \mathbb{Z}$

Figure 5.4: Semantics of the Alloy kernel, taken from [27, 72, 47], where $e \in exp$, $ie \in intExp$, $r \in relSym$ and $n \in number$. However, for the operators override and sum, their semantics were extracted from corresponding paragraphs of the Alloy book [47].

$$\{\forall a_{1:i} : Atom.\ (a_{1:i}) \notin \emptyset_i \mid 1 \leq i \leq N\} \tag{5.7}$$
$$\{\forall a_{1:i}, b_{1:i} : Atom.\ (b_{1:i}) \in_i \{(a_{1:i})\}_i \leftrightarrow b_1 = a_1 \wedge \cdots \wedge b_i = a_i \mid 1 \leq i \leq N\} \tag{5.8}$$
$$\{\forall R, S : Rel_i, a_{1:i} : Atom.\ (a_{1:i}) \in_i R \cup_i S \leftrightarrow (a_{1:i}) \in_i R \vee (a_{1:i}) \in_i S \mid 1 \leq i \leq N\} \tag{5.9}$$
$$\{\forall R, S : Rel_i.\ R =_i S \leftrightarrow (\forall a_{1:i} : Atom.\ a_{1:i} \in_i R \leftrightarrow a_{1:i} \in_i S) \mid 1 \leq i \leq N\} \tag{5.10}$$

Definition 14 ($\mathcal{M}$-Homomorphism)**.** Each RFOL structure $\mathcal{M}$ deduces a canonical homomorphism $\simeq_M : \bigcup_{1\leq i\leq N} \bar{M}^{Rel_i} \cup \bar{M}^{Atom} \to \bigcup_{1\leq i\leq N} \mathcal{P}((\bar{M}^{Atom})^i) \cup \bar{M}^{Atom}$, with

$$\simeq_M(\bar{r}) = \begin{cases} \bar{r} & \text{if } \bar{r} \in \bar{M}^{Atom} \\ \{(\bar{a}_{1:i}) \in (\bar{M}^{Atom})^i \mid M(\in_i)(\bar{a}_{1:i}, \bar{r})\} & \text{if } \bar{r} \in \bar{M}^{Rel_i} \end{cases}$$

When more convenient, we use $\bar{r} \simeq_M s$ to denote $\simeq_M(\bar{r}) = s$.

Theorem 2 ($\mathcal{M}$-Isomorphism)**.** *For any RFOL structure $\mathcal{M}$ (compatible with α_Σ and Ax) and $1 \leq i \leq N$, the universe of the sort Rel_i with the relation $M(\in_i)$ is isomorph to the power set of the i-th self Cartesian product of the universe of $Atom$ with the relation $\in$, namely with $\simeq_M$. That is,*

1. *$\simeq_M$ is surjective*
2. *$\simeq_M$ is injective*
3. *$M(\in_i)(\bar{a}_{1:i}, \bar{r}) \iff (\bar{a}_{1:i}) \in \simeq_M(\bar{r})$*

Proof. To prove (1), we construct for each $\bar{s} \subseteq (\bar{M}^{Atom})^i$ a term r of sort Rel_i such that $\simeq_M(val_{\mathcal{M},\beta}(r))) = \bar{s}$ for some variable assignment β. We do so by induction on the cardinality of $\bar{s}$.

For $\bar{s} = \emptyset^i$, we choose $r = \emptyset_i$ and β arbitrary. Then,

$$\begin{aligned}
& (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(r)) \\
\iff & M(\in_i)(\bar{a}_{1:i}, M(\emptyset_i)) \\
\iff & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i \emptyset_i && \text{, for some variables } a_{1:i} \\
\overset{5.7}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models \mathit{false} \\
\iff & \mathit{ff} \\
\iff & (\bar{a}_{1:n}) \in \bar{s}
\end{aligned}$$

For $\bar{s} = \{(\bar{b}_{1:i})\}$, we choose $r = \{(b_{1:i})\}_i$ and β such that $\beta(b_j) = \bar{b}_j$ for $1 \leq j \leq i$. Then,

$$
\begin{aligned}
& (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(r)) \\
\iff & M(\in_i)(\bar{a}_{1:i}, val_{\mathcal{M},\beta}(\{(b_{1:i})\}_i)) \\
\iff & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i \{(b_{1:i})\}_i && \text{, for variables } a_{1:i} \\
\overset{5.8}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models a_1 = b_1 \wedge \cdots \wedge a_i = b_i \\
\overset{val}{\iff} & \bar{a}_1 = \beta(b_1) \wedge \cdots \wedge \bar{a}_i = \beta(b_i) \\
\overset{\beta}{\iff} & \bar{a}_1 = \bar{b}_1 \wedge \cdots \wedge \bar{a}_i = \bar{b}_i \\
\iff & (\bar{a}_{1:i}) \in \{(\bar{b}_{1:i})\}
\end{aligned}
$$

Let now the cardinality of $\bar{s}$ be $|\bar{s}| = n + 1$ with $0 < n$. Then, it exists $\bar{b}_{1:i} \in \bar{s}$ such that $\bar{s} = \bar{s}' \cup \{\bar{b}_{1:i}\}$ and $|\bar{s}'| = n$. By induction hypothesis, it exists an r' of sort Rel_i and a variable assignment β' such that (1) holds for $\bar{s}'$. For $\bar{s}$, we choose $r = r' \cup_i \{(b_{1:i})\}_i$ and $\beta = \beta'^{\bar{b}_{1:i}}_{b_{1:i}}$. Then,

$$
\begin{aligned}
& (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(r)) \\
\iff & M(\in_i)(\bar{a}_{1:i}, val_{\mathcal{M},\beta}(r' \cup_i \{(b_{1:i})\}_i)) \\
\iff & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i r' \cup_i \{(b_{1:i})\}_i && \text{, for variables } a_{1:i} \\
\overset{5.9}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i r' \vee (a_{1:i}) \in_i \{(b_{1:i})\}_i \\
\overset{val}{\iff} & (\mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i r') \vee (\mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i \{(b_{1:i})\}_i) \\
\overset{val}{\iff} & ((\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(r'))) \vee ((\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(\{(b_{1:i})\}_i))) \\
\overset{IH}{\iff} & ((\bar{a}_{1:i}) \in \bar{s}') \vee ((\bar{a}_{1:i}) \in \{(\bar{b}_{1:i})\}) \\
\iff & (\bar{a}_{1:i}) \in \bar{s}' \cup \{(\bar{b}_{1:i})\}_i
\end{aligned}
$$

In order to prove (2), it suffices to show $|\bar{M}^{Rel_i}| \leq |\mathcal{P}((\bar{M}^{Atom})^i)|$. Axiom 5.10 states that each element R of Rel_i is uniquely identifiable by the interpretation of $\in_i$. Consequently, there are at most as many distinct elements of Rel_i as distinct interpretations of $\in_i$. Since there are $2^{i*|\bar{M}^{Atom}|}$ distinct interpretation of $\in_i$, (2) holds.
(3) is true by definition of $\simeq_M$. □

The second RFOL structures feature extends the first to all relational RFOL operators. That is, the interpretation of Rel_i with any relational operator (e.g., $\cup_i$) is isomorph to the power set of the interpretation of $Atom^i$ with its set theoretical counterpart (here $\cup$). Lemma 1 handles the case of first-order relational constructors. Transitive closure and cardinality are handled, respectively, in lemma 3 and lemma 4.

Lemma 1. *Let $t = op(t_{1:n})$ be a complex i-ary RFOL term with a top-level first-order relational constructor $op \in \{\emptyset_i, \{\}_i, ^{-1}, \bullet_{i,j}, \times_{i,j}, \oplus_{i,j}, \setminus_i, \cup_i, \cap_i\}$. Then,*

$$val_{\mathcal{M},\beta}(op(t_{1:n})) \simeq_M \bar{op}(\simeq_M(val_{\mathcal{M},\beta}(t_1)), \ldots, \simeq_M(val_{\mathcal{M},\beta}(t_n)))$$

, where $\bar{op}$ denotes the set theoretical counterpart operator of op in prefix notation.

Proof. We conduct the proof only for some selected operators, for the rest the proof goes analog. Let consider some atom values $(\bar{a}_{1:i})$.

For $t = \emptyset_i$, we show

$$\begin{aligned}
& (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(\emptyset_i)) \\
\iff & M(\in_i)(\bar{a}_{1:i}, val_{\mathcal{M},\beta}(\emptyset_i) \\
\overset{val}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i \emptyset_i \\
\overset{5.7}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models \mathit{false} \\
\iff & \mathit{ff} \\
\iff & (\bar{a}_{1:i}) \in \emptyset
\end{aligned}$$

For $t = \{b_{1:i}\}_i$, we show

$$\begin{aligned}
& (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(\{b_{1:i}\}_i)) \\
\iff & M(\in_i)(\bar{a}_{1:i}, val_{\mathcal{M},\beta}(\{b_{1:i}\}_i) \\
\overset{val}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i \{b_{1:i}\}_i \\
\overset{5.8}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models a_1 = b_i \wedge \cdots \wedge a_i = b_i \\
\iff & \bar{a}_1 = \beta(b_1) \wedge \cdots \wedge \bar{a} = \beta(b_i) \\
\iff & (\bar{a}_{1:i}) \in \{val_{\mathcal{M},\beta}(b_1), \ldots, val_{\mathcal{M},\beta}(b_i)\}
\end{aligned}$$

For $t = t_1 \cup_i t_2$, we show

$$\begin{aligned}
& (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(t_1 \cup_i t_2)) \\
\iff & M(\in_i)(\bar{a}_{1:i}, val_{\mathcal{M},\beta}(t_1 \cup_i t_2) \\
\overset{val}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i t_1 \cup_i t_2 \\
\overset{5.9}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i t_1 \vee (a_{1:i}) \in_i t_2 \\
\iff & (\mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i t_1) \vee (\mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i t_2) \\
\overset{val}{\iff} & M(\in_i)(\bar{a}_{1:i}, val_{\mathcal{M},\beta}(t_1)) \vee M(\in_i)(\bar{a}_{1:i}, val_{\mathcal{M},\beta}(t_2)) \\
\overset{IH}{\iff} & (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(t_1)) \vee (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(t_2)) \\
\iff & (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(t_1)) \cup \simeq_M(val_{\mathcal{M},\beta}(t_2))
\end{aligned}$$

For $t = t_1 \cdot_{n,m} t_2$, we assume w.l.o.g. that $i = n + m - 2$ and show

$$
\begin{aligned}
& (\bar{a}_{1:i}) \in \simeq_M(val_{\mathcal{M},\beta}(t_1 \cdot_{n,m} t_2)) \\
\iff & M(\in_i)(\bar{a}_{1:i}, val_{\mathcal{M},\beta}(t_1 \cdot_{n,m} t_2) \\
\overset{val}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models (a_{1:i}) \in_i t_1 \cdot_{n,m} t_2 \\
\overset{Ax}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:i}}_{a_{1:i}} \models \exists b : Atom.\ (a_{1:n-1}, b) \in_n t_1 \wedge (b, a_{n:i}) \in_m t_2 \\
\iff & \mathcal{M}, \beta^{\bar{a}_{1:i}\bar{b}}_{a_{1:i}b} \models (a_{1:n-1}, b) \in_n t_1 \wedge (b, a_{n:i}) \in_m t_2 && \text{, for some } \bar{b} \\
\iff & (\mathcal{M}, \beta^{\bar{a}_{1:i}\bar{b}}_{a_{1:i}b} \models (a_{1:n-1}, b) \in_n t_1) \wedge (\mathcal{M}, \beta^{\bar{a}_{1:i}\bar{b}}_{a_{1:i}b} \models (b, a_{n:i}) \in_m t_2) && \text{, for some } \bar{b} \\
\overset{val}{\iff} & M(\in_n)(\bar{a}_{1:n-1}, \bar{b}, val_{\mathcal{M},\beta}(t_1)) \wedge M(\in_m)(\bar{b}, \bar{a}_{n:i}, val_{\mathcal{M},\beta}(t_2)) && \text{, for some } \bar{b} \\
\overset{IH}{\iff} & (\bar{a}_{1:n-1}, \bar{b}) \in \simeq_M(val_{\mathcal{M},\beta}(t_1)) \wedge (\bar{b}, \bar{a}_{n:i}) \in \simeq_M(val_{\mathcal{M},\beta}(t_2)) && \text{, for some } \bar{b} \\
\iff & \exists \bar{b}.\ (\bar{a}_{1:n-1}, \bar{b}) \in \simeq_M(val_{\mathcal{M},\beta}(t_1)) \wedge (\bar{b}, \bar{a}_{n:i}) \in \simeq_M(val_{\mathcal{M},\beta}(t_2)) \\
\iff & (\bar{a}_{1:n-1}, \bar{b}) \in \simeq_M(val_{\mathcal{M},\beta}(t_1)) \cdot \simeq_M(val_{\mathcal{M},\beta}(t_2))
\end{aligned}
$$

□

The next lemma inlines the semantics of the iterative self join operator $^{()}$ —hidden in its recursive axioms (Figure 4.3). It is used in the proof of lemma 3.

Lemma 2. *For each binary relational term r, positive number n and atom tuple (a, b),*

$$(a, b) \in_2 r^{(n)} \iff \exists x_{1:n} : Atom.\ (a, x_1) \in_2 r \wedge \cdots \wedge (x_n, b) \in_2 r.$$

Proof. Proof by induction on n. The base case of $n = 0$ is trivial. Let assume the claim is true for all $0 \leq i \leq n$. The claim is then proven by:

$$
\begin{aligned}
& (a, b) \in_2 r^{(n+1)} \\
\overset{Ax_{()}}{\iff} & (a, b) \in_2 r^{(n)} \cdot_{2,2} r \\
\overset{Ax_{\cdot}}{\iff} & \exists x_{n+1} : Atom.\ (a, x_{n+1}) \in_2 r^{(n)} \wedge (x_{n+1}, b) \in_2 r \\
\overset{IH}{\iff} & \exists x_{n+1} : Atom.\ (\exists x_{1:n} : Atom.\ (a, x_1) \in_2 r \wedge \cdots \wedge (x_n, x_{n+1}) \in_2 r) \wedge (x_{n+1}, b) \in_2 r \\
\iff & \exists x_{1:n+1} : Atom.\ (a, x_1) \in_2 r \wedge \cdots \wedge (x_{n+1}, b) \in_2 r
\end{aligned}
$$

□

Lemma 3. *Let $\mathcal{M}$ be an RFOL structure and $r \in Rel_2$ be a binary relational term. Then,*

$$val_{\mathcal{M},\beta}(r^+) \simeq_M (\simeq_M(val_{\mathcal{M},\beta}(r)))^+$$

, where the set theoretical transitive closure is on the right-hand side.

Proof. Let consider the atom values $\bar{a}_{1:2}$. Then, we show that $(\bar{a}_{1:2}) \in \simeq_M(val_{\mathcal{M},\beta}(r^+))$ iff $(\bar{a}_{1:2}) \in (\simeq_M(val_{\mathcal{M},\beta}(r)))^+$.

$$
\begin{aligned}
& (\bar{a}_{1:2}) \in \simeq_M(val_{\mathcal{M},\beta}(r^+)) \\
\iff & M(\in_i)(\bar{a}_{1:2}, val_{\mathcal{M},\beta}(r^+) \\
\overset{val}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:2}}_{a_{1:2}} \models (a_{1:2}) \in_2 r^+ \\
\overset{Ax}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:2}}_{a_{1:2}} \models \exists n : Int.\ 0 \leq n \wedge (a_{1:2}) \in_2 r^{(n)} \\
\overset{lem.\ 2}{\iff} & \mathcal{M}, \beta^{\bar{a}_{1:2}}_{a_{1:2}} \models \exists n : Int.\ 0 \leq n \wedge (\exists b_{1:n} : Atom.\ (a_1, b_1) \in_2 r \wedge \cdots \wedge (b_n, a_2) \in_2 r) \\
\iff & \mathcal{M}, \beta^{\bar{a}_{1:2}}_{a_{1:2}} \models \exists b_{1:n} : Atom.\ (a_1, b_1) \in_2 r \wedge \cdots \wedge (b_n, a_2) \in_2 r \\
\iff & M(\in_2)(\bar{a}_1, \bar{b}_1, val_{\mathcal{M},\beta}(r)) \wedge \cdots \wedge M(\in_2)(\bar{b}_n, \bar{a}_2), val_{\mathcal{M},\beta}(r)), \text{ for some } \bar{b}_{1:n} \\
\iff & (\bar{a}_1, \bar{b}_1) \in \simeq_M(val_{\mathcal{M},\beta}(r)) \wedge \cdots \wedge (\bar{b}_n, \bar{a}_2) \in \simeq_M(val_{\mathcal{M},\beta}(r)), \text{ for some } \bar{b}_{1:n} \\
\iff & \exists \bar{b}_{1:n}.\ (\bar{a}_1, \bar{b}_1) \in \simeq_M(val_{\mathcal{M},\beta}(r)) \wedge \cdots \wedge (\bar{b}_n, \bar{a}_2) \in \simeq_M(val_{\mathcal{M},\beta}(r)) \\
\overset{Fig.\ 5.4}{\iff} & (\bar{a}_{1:2}) \in (\simeq_M(val_{\mathcal{M},\beta}(r)))^+
\end{aligned}
$$

□

Lemma 4. *For each n-ary relational term r, RFOL structure $\mathcal{M}$, and variable assignment β,*

$$val_{\mathcal{M},\beta}(|r|_n) = |\simeq_M(val_{\mathcal{M},\beta}(r))|.$$

Proof. Remember that an ROFL structure is implicitly a $\mathbb{Z}$ structure satisfying Ax. Let us assume w.l.o.g. that $|\simeq_M(val_{\mathcal{M},\beta}(r))| = k$. From the cardinality definition for *finite sets*, we can further assume w.l.o.g. that $\simeq_M(val_{\mathcal{M},\beta}(r)) = \{(\bar{a}_{1_{1:n}}), \ldots, (\bar{a}_{k_{1:n}})\} \subseteq \bar{M}^{Atom}$

We first proof by contradiction that $\mathcal{M}, \beta \models |r|_n \leq k$. Let us assume $\mathcal{M}, \beta \models k < |r|_n$. From Ax (second axiom in Figure 4.5) we get that there exists at least $k+1$ atom tuples $(a_{1_{1:n}}), \ldots, (a_{k+1_{1:n}})$ with $(a_{i_{1:n}}) \in_n r$ and $ord_n(r, a_{i_{1:n}}) = i$ for all $1 \leq i \leq k+1$. Also from the cardinality axioms (this times, third axiom) we get that all $k+1$ atom tuples $(a_{i_{1:n}})$ must be distinct. Altogether we can infer that it must be at least $k+1$ distinct atom tuples belonging to r. Consequently, $\simeq_M(val_{\mathcal{M},\beta}(r))$ must have also at lest $k+1$ distinct atom value tuples which is a contradiction to our assumption.

Next, we proof, also by contradiction, that $\mathcal{M}, \beta \models k \leq |r|_n$ and thus the claim. To do so, we first show that r can be written as the union of some k unary singleton sets.

$$
\begin{aligned}
& \mathcal{M}, \beta \models \exists a_{1_{1:n}}, \ldots, a_{k_{1:n}} : Atom.\ r = \{(a_{1_{1:n}})\}_n \cup_n \cdots \cup_n \{(a_{k_{1:n}})\}_n \\
\iff & \mathcal{M}, \beta^{\bar{v}_{1_{1:n}} \ldots \bar{v}_{k_{1:n}}}_{a_{1_{1:n}} \ldots a_{k_{1:n}}} \models r = \{(a_{1_{1:n}})\}_n \cup_n \cdots \cup_n \{(a_{k_{1:n}})\}_n, \text{ for some } \bar{v}_{1_{1:n}}, \ldots, \bar{v}_{k_{1:n}} \\
\iff & val_{\mathcal{M},\beta}(r) = val^{\bar{v}_{1_{1:n}} \ldots \bar{v}_{k_{1:n}}}_{\beta_{a_{1_{1:n}} \ldots a_{k_{1:n}}}} (\{(a_{1_{1:n}})\}_n \cup_n \cdots \cup_n \{(a_{k_{1:n}})\}_n), \text{ for some } \bar{v}_{1_{1:n}}, \ldots, \bar{v}_{k_{1:n}} \\
\overset{lem.\ 1}{\iff} & val_{\mathcal{M},\beta}(r) = \bigcup_{1 \leq i \leq k} val^{\bar{v}_{1_{1:n}} \ldots \bar{v}_{k_{1:n}}}_{\beta_{a_{1_{1:n}} \ldots a_{k_{1:n}}}} (\{(\bar{a}_{i_{1:n}})\}_n), \text{ for some } \bar{v}_{1_{1:n}}, \ldots, \bar{v}_{k_{1:n}} \\
\overset{lem.\ 1}{\iff} & val_{\mathcal{M},\beta}(r) = \bigcup_{1 \leq i \leq k} \{(\bar{v}_{i_{1:n}})\}, \text{ for some } \bar{v}_{1_{1:n}}, \ldots, \bar{v}_{k_{1:n}} \\
\overset{\cup}{\iff} & val_{\mathcal{M},\beta}(r) = \{(\bar{v}_{1_{1:n}}), \ldots, (\bar{v}_{k_{1:n}})\}, \text{ for some } \bar{v}_{1_{1:n}}, \ldots, \bar{v}_{k_{1:n}}
\end{aligned}
$$

Choosing $\bar{a}_{1_{1:n}}, \bar{a}_{k_{1:n}}$ for $\bar{v}_{1_{1:n}}, \bar{v}_{k_{1:n}}$, concludes this proof.

Let us now assume that $\mathcal{M}, \beta \models |r|_n < k$. By applying the first axiom of Figure 4.5 to each $a_{i_{1:n}}$ we get that $ord_n(r, a_{i_{1:n}}) \leq |r|_n$ for all $1 \leq i \leq k$. Since $ord_n(r, .)$ is injective for all elements of r, we can conclude that $k \leq |r|_n$ which contradict the assumption. □

5.3.3 Correctness

To show the correctness of our translation, we first provide in Figure 5.5 a construction schema of an RFOL structure $\mathcal{M}_\mathcal{I}$ from a given Alloy instance $\mathcal{I}$. Note that since $\mathcal{M}_\mathcal{I}$ is an RFOL structure, it is a structure for all its axioms Ax too. Also note that $\mathcal{M}_\mathcal{I}$ is not necessary unique.

$$
\begin{aligned}
& \bar{M}^{Atom}_\mathcal{I} = U && && (5.11) \\
& \bar{M}^{Rel_i}_\mathcal{I} = \mathcal{P}(U^i) && \text{for all } 1 \leq i \leq N && (5.12) \\
& \bar{M}^{Int}_\mathcal{I} = \mathbb{Z} && && (5.13) \\
& M_\mathcal{I}(Na[r]) = I(r) && \text{for } r \in relSym && (5.14) \\
& M_\mathcal{I}(\in_i) = \{(\bar{a}_{1:i}, \bar{t}) \in U^i \times \mathcal{P}(U^i) \mid (\bar{a}_{1:i}) \in \bar{t}\} && \text{for all } 1 \leq i \leq N && (5.15)
\end{aligned}
$$

Figure 5.5: Construction of an RFOL structure from an Alloy instance.

Before we formulate and prove our main correctness theorem, we prove some general properties of the $\mathcal{M}_\mathcal{I}$ construction.

Lemma 5. *The canonical isomorphism deduced by $\mathcal{M}_\mathcal{I}$ (Definition 14) is the identity. That is, $\simeq_{\mathcal{M}_\mathcal{I}}(\bar{r}) = \bar{r}$ for all term values $\bar{r}$.*

Proof. Let $\bar{a}_{1:i}$ be some atom values in $\bar{\mathcal{M}_\mathcal{I}}^{Atom}$ and $\bar{r}$ a relational term value in $\bar{\mathcal{M}_\mathcal{I}}^{Rel_i}$. Then,

$$\begin{aligned} & (\bar{a}_{1:i}) \in \simeq_{\mathcal{M}_\mathcal{I}}(\bar{r}) \\ \overset{\simeq_{\mathcal{M}_\mathcal{I}}}{\iff} & \mathcal{M}_\mathcal{I}(\in_i)(\bar{a}_{1:i}, \bar{r}) \\ \overset{5.15}{\iff} & \bar{a}_{1:i} \in \bar{r} \end{aligned}$$

□

Lemma 6. *Let $\mathcal{I}$ be an Alloy instance of an Alloy problem $P = (M, \mathcal{F}, A)$ and $T[P] = (\alpha_{M_C}, C_{M_C}, C_F, C_A)$ its RFOL translation. Then, $\mathcal{M}_\mathcal{I}$ is a well-defined Σ-structure, where $\alpha_\Sigma = \alpha_{M_C}$.*

Proof. Since α_{M_C} only modifies the sorting of translated Alloy symbols, it is sufficient to show that for all Alloy symbols r, $M_\mathcal{I}(Na[r]) : \bar{M_\mathcal{I}}^{T_1} \times \cdots \times \bar{M_\mathcal{I}}^{T_{n-1}} \to \bar{M_\mathcal{I}}^{T_n}$ if $\alpha_{M_C}(Na[r]) = (T_1, \ldots T_n)$. This is guaranteed by the translation rules of the auxiliary translation function $\mathcal{D}$ in Figure 5.1. □

The next lemma states that the evaluation of an Alloy expression in the context of an Alloy instance $\mathcal{I}$ agrees with the evaluation of its translation with respect to the RFOL structure $\mathcal{M}_\mathcal{I}$.

Lemma 7. *Let e be an Alloy expression, $\mathcal{I}$ an Alloy instance, and $\beta : \mathcal{X} \to \bar{M}_\mathcal{I}$ an RFOL variable assignment which agrees with I on translated Alloy variables (i.e., $\beta(Na[x]) = I(x)$). Then,*

$$E[e]I = val_{\mathcal{M}_\mathcal{I}, \beta}(\mathcal{E}(e)).$$

Proof. We prove the lemma by induction on the construction of Alloy expression (Figure 5.4). However, we demonstrate the proof only for the interesting cases.

If $e = x$ is a variable or a relational symbol, then we have on the one hand $E[x]I = I(x)$. On other hand we have that $val_{\mathcal{M}_\mathcal{I}, \beta}(\mathcal{E}[x]) = \beta(Na[x])$. The claim follows, here, from the conditions on β.

If $e = r$ is a relational symbol, then we have on the one hand $E[r]I = I(r)$. On the other hand we have that $val_{\mathcal{M}_\mathcal{I}, \beta}(\mathcal{E}[r]) = M_I(Na[r])$. The claim follows, here, from the construction of $\mathcal{M}_\mathcal{I}$ (rule 5.14 of Figure 5.5).

Let now e be is a complex Alloy expression. Depending e's top-level operator op_A, we distinguish between three essential cases: (1) op_A's RFOL counterpart op_R is a first-order relational constructor (cf. Figure 4.2), (2) op_A is the transitive closure, and (3) op_A is the cardinality.

For the first case, let be $e = op_A(e_{1:n})$, where $op_A \in \{\text{non}, +, ++, -, \&, .., \to, \tilde{}, \hat{}\}$. Let furthermore $\bar{op}$ denotes the set theoretical counterpart of op_A. Then,

$$
\begin{array}{rl}
 & val_{\mathcal{M}_\mathcal{I},\beta}(\mathcal{E}[op_A(e_{1:n})]) \\
= & val_{\mathcal{M}_\mathcal{I},\beta}(op_R(\mathcal{E}[e_1],\ldots,\mathcal{E}[e_n])) \\
\overset{Lem\ 1}{=} & \bar{op}(val_{\mathcal{M}_\mathcal{I},\beta}(\mathcal{E}[e_1]),\ldots,val_{\mathcal{M}_\mathcal{I},\beta}(\mathcal{E}[e_n])) \\
\overset{IH}{=} & \bar{op}(E[e_1]I,\ldots,E[e_n]I) \\
\overset{Fig\ 5.4}{=} & E[op_A(e_{1:n})]I
\end{array}
$$

For the second case, let be $e = \hat{}\mathsf{e}_1$, where e_1 a binary relational expression. Then,

$$
\begin{array}{rl}
 & val_{\mathcal{M}_\mathcal{I},\beta}(\mathcal{E}[\hat{}\mathsf{e}_1]) \\
= & val_{\mathcal{M}_\mathcal{I},\beta}(\mathcal{E}[e_1]^+) \\
\overset{lem\ 3}{=} & (val_{\mathcal{M}_\mathcal{I},\beta}(\mathcal{E}[e_1]))^+ \\
\overset{IH}{=} & (E[e_1]I)^+ \\
\overset{Fig\ 5.4}{=} & E[\hat{}\mathsf{e}_1]I
\end{array}
$$

For the third case, let be $e = \#\mathsf{e}_1$, where $ar(e_1) = n$. In this case he claim follows directly from lemma 4 together with the induction hypothesis, as follow:

$$
\begin{array}{rl}
 & val_{\mathcal{M}_\mathcal{I},\beta}(\mathcal{E}[\#\mathsf{e}_1]) \\
= & val_{\mathcal{M}_\mathcal{I},\beta}(|\mathcal{E}[e_1]|_n) \\
\overset{lem\ 4}{=} & |val_{\mathcal{M}_\mathcal{I},\beta}(\mathcal{E}[e_1])| \\
\overset{IH}{=} & |E[e_1]I|
\end{array}
$$

□

The next, and last lemma in these series, extends the results of lemma 7 to formulas.

Lemma 8. *Let φ be an Alloy formula, $\mathcal{I}$ an Alloy instance, and $\beta : \mathcal{X} \to \bar{M}_\mathcal{I}$ an RFOL variable assignment which agree with I on translated Alloy variables (i.e., $\beta(Na[x]) = I(x)$). Then,*

$$M[\varphi]I \text{ iff } \mathcal{M}_\mathcal{I}, \beta \models \mathcal{C}[\varphi].$$

Proof. Proof by induction on the construction of Alloy formulas (see. Figure 5.4). Let $\mathcal{I}$ be an Alloy instance of φ. Cases where logical connectives are top-level, are trivial.

Case of $\varphi = \mathsf{e}_1 \mathbf{in}\ \mathsf{e}_2$: Assuming $ar(e_1) = ar(e_2) = n$

$$
\begin{aligned}
& \mathcal{M}_\mathcal{I}, \beta \models \mathcal{C}[e_1 \mathbf{in} e_2] \\
\iff & \mathcal{M}_\mathcal{I}, \beta \models \mathcal{E}[e_1] \subseteq_n \mathcal{E}[e_2] \\
\overset{Ax}{\iff} & \mathcal{M}_\mathcal{I}, \beta \models \forall a_{1:n} : Atom.\ (a_{1:n}) \in_n \mathcal{E}[e_1] \to (a_{1:n}) \in_n \mathcal{E}[e_2] \\
\overset{val}{\iff} & \mathcal{M}_\mathcal{I}, \beta^{\bar{a}_{1:n}}_{a_{1:n}} \models (a_{1:n}) \in_n \mathcal{E}[e_1] \to (a_{1:n}) \in_n \mathcal{E}[e_2], \text{ for all } \bar{a}_{1:n} \\
\overset{val}{\iff} & (\mathcal{M}_\mathcal{I}, \beta^{\bar{a}_{1:n}}_{a_{1:n}} \models (a_{1:n}) \in_n \mathcal{E}[e_1]) \to (\mathcal{M}_\mathcal{I}, \beta^{\bar{a}_{1:n}}_{a_{1:n}} \models (a_{1:n}) \in_n \mathcal{E}[e_2]), \text{ for all } \bar{a}_{1:n} \\
\overset{val}{\iff} & ((\bar{a}_{1:n}) \in val_{\mathcal{M}_\mathcal{I}, \beta}(\mathcal{E}[e_1])) \to ((\bar{a}_{1:n}) \in val_{\mathcal{M}_\mathcal{I}, \beta}(\mathcal{E}[e_2])), \text{ for all } \bar{a}_{1:n} \\
\overset{lem.\ 7}{\iff} & ((\bar{a}_{1:n}) \in E[e_1]I) \to ((\bar{a}_{1:n}) \in E[e_2]I)), \text{ for all } \bar{a}_{1:n} \\
\iff & E[e_1]I \subseteq E[e_2] \\
\overset{Fig\ 5.4}{\iff} & M[\mathsf{e}_1 \mathbf{in}\ \mathsf{e}_2]
\end{aligned}
$$

Case of $\varphi = \mathsf{e}_1 \mathbf{=}\ \mathsf{e}_2$: This case can be reduced to the first case, since the Alloy equality is translated to RFOL (relational equality) equality which is nothing else than bi-inclusion.

Case of $\varphi = \mathbf{all}$ x: e | f: Assuming $ar(e) = n$

$$
\begin{aligned}
& \mathcal{M}_\mathcal{I}, \beta \models \mathcal{C}[\mathbf{all}\ \text{x: e} \mid \text{f}] \\
\iff & \mathcal{M}_\mathcal{I}, \beta \models \forall x_{1:n} : Atom.\ (x_{1:n}) \in_n \mathcal{E}[e] \to \mathcal{C}[f][\{(x_{1:n})\}_n / Na[x]] \\
\overset{val}{\iff} & \mathcal{M}_\mathcal{I}, \beta^{\bar{x}_{1:n}}_{x_{1:n}} \models (x_{1:n}) \in_n \mathcal{E}[e] \to \mathcal{C}[f][\{(x_{1:n})\}_n / Na[x]], \text{ for all } \bar{x}_{1:n} \\
\overset{val}{\iff} & (\bar{x}_{1:n}) \in val_{\mathcal{M}_\mathcal{I}, \beta}(\mathcal{E}[e]) \to \mathcal{M}_\mathcal{I}, \beta^{\bar{x}_{1:n}}_{x_{1:n}} \models \mathcal{C}[f][\{(x_{1:n})\}_n / Na[x]], \text{ for all } \bar{x}_{1:n} \\
\overset{lem.\ 7}{\iff} & (\bar{x}_{1:n}) \in E[e]I \to \mathcal{M}_\mathcal{I}, \beta^{\bar{x}_{1:n}}_{x_{1:n}} \models \mathcal{C}[f][\{(x_{1:n})\}_n / Na[x]], \text{ for all } \bar{x}_{1:n} \\
\overset{IH}{\iff} & (\bar{x}_{1:n}) \in E[e]I \to M[f](I \oplus x \mapsto \{(\bar{x}_{1:n})\}), \text{ for all } \bar{x}_{1:n} \\
\iff & ((\bar{x}_{1:n}) \in E[e]I \wedge |\{(\bar{x}_{1:n})\}| = 1) \to M[f](I \oplus x \mapsto \{(\bar{x}_{1:n})\}), \text{ for all } \bar{x}_{1:n} \\
\iff & \bigwedge \{M[f](I \oplus a \mapsto \{(\bar{x}_{1:n})\}) \mid (\bar{x}_{1:n}) \in E[e]I \wedge |\{(\bar{a}_{1:n})\}| = 1\} \\
\overset{Fig\ 5.4}{\iff} & M[\mathbf{all}\ \text{x: e} \mid \text{f}]I
\end{aligned}
$$

Case of $\varphi = \mathbf{some}$ x: e | f: Goes similarly to the universal quantifier case. □

Now, we formulate and prove our correctness theorem.

Theorem 3 (Correctness). *Let $P = (M, \mathcal{F}, A)$ be an Alloy problem and $T[P] = (\alpha_{M_S}, C_M, C_F, C_A)$ its RFOL translation. Then, P is correct if $\models_{Cl(Ax),\mathbb{Z}} C_M \wedge C_\mathcal{F} \to C_A$.*

Proof. Proof by contradiction. We prove that for each counterexample of P —an Alloy instance of $\neg(\bigwedge M_\mathcal{C} \wedge \bigwedge \mathcal{F} \to A)$ that is conform to $M_\mathcal{S}$, a counterexample for $T[P]$

—an RFOL structure of$\neg(C_{\mathcal{M}} \wedge C_{\mathcal{F}} \rightarrow C_A)$— can be constructed. Since $\mathcal{C}[M_{\mathcal{C}}] = C_{\mathcal{M}}$, $\mathcal{C}[\mathcal{F}] = C_{\mathcal{F}}$, $\mathcal{C}[A] = C_A$, and all logical connectives has in both logics their intended semantics, it is sufficient to show that for each Alloy formula φ,

$$M[\varphi]I \text{ iff } \mathcal{M}_{\mathcal{I}}, \beta \models \mathcal{C}[\varphi]$$

, where $\mathcal{M}_{\mathcal{I}}$ is defined as in Figure 5.5 and $\beta(Na[x]) = I(x)$ for all Alloy variables. And this is exactly the proved claim in lemma 8. □

5.3.4 Completeness

In this section we discus the completeness of our translation. It does not concern the completeness of the RFOL calculus. However, for a complete RFOL calculus, the translation completeness will guarantee that the calculus proves the validity of the translation of any correct Alloy problem. With other words, the translation completeness together with the correctness ensure that the proof obligation of any Alloy problem is *equisatisfiable* to its RFOL translation.

Basically, we use a similar construction to Figure 5.5 to construct an Alloy instance from a given RFOL structure. Figure 5.6 shows the construction.

$$U_{\mathcal{M}} = \bar{M}^{Atom} \tag{5.16}$$

$$I_{\mathcal{M}}(Int) = \mathbb{Z} \tag{5.17}$$

$$I_{\mathcal{M}}(r) = \simeq_M(M(Na[r])) \qquad \text{for } r \in relSym \tag{5.18}$$

$$I_{\mathcal{M}}(\in) = \{((\bar{a}_{1:i}), \bar{r}) \in (\bar{M}^{Atom})^i \times \bar{M}^{Rel_i} \mid (\bar{a}_{1:i}, \bar{r}) \in M(\in_i)\} \qquad \text{for } 0 \leq i \leq N \tag{5.19}$$

Figure 5.6: Construction of an *Alloy* instance from an RFOL structure.

Since, like in the $\mathcal{M}_{\mathcal{I}}$-construction (rule 5.14), $\mathcal{I}_{\mathcal{M}}$ agrees with $\mathcal{M}$ on the interpretation of relational symbols modulo the isomorphism $\simeq_M$ (rule 5.18), lemma 7 holds for the $\mathcal{I}_{\mathcal{M}}$-construction correspondingly[7]. Note that, unlike for the $\mathcal{M}_{\mathcal{I}}$-construction, $\simeq_M$ is for the $\mathcal{I}_{\mathcal{M}}$-construction not necessary the identity.

Corollary 1. *Let e be an Alloy expression, $\mathcal{M}$ an RFOL structure, and $\beta : \mathcal{X} \rightarrow \bar{M}$ an RFOL variable assignment. Then,*

$$\simeq_M(val_{\mathcal{M},\beta}(\mathcal{E}(e))) = E[e]I_{\mathcal{M}}$$

, where $I_{\mathcal{M}}$ agrees with β on the translated Alloy variables (i.e., $I(x) = \beta(Na[x])$).

Having corollary 1, the corresponding claim of lemma 8 for the Alloy instance construction of Figure 5.6, follows directly.

[7] Rule 5.14 is the only $\mathcal{M}_{\mathcal{I}}$-construction related argumentation used for proving Lemma 7

Corollary 2. *Let φ be an Alloy formula, $\mathcal{M}$ an RFOL structure, and $\beta : \mathcal{X} \to \bar{M}$ an RFOL variable assignment. Then,*

$$M[\varphi]I_{\mathcal{M}} \text{ iff } \mathcal{M}, \beta \models \mathcal{C}[\varphi]$$

, where $I_{\mathcal{M}}$ agrees with β on the translated Alloy variables (i.e., $I(x) = \beta(Na[x])$).

Theorem 4 (Completeness)**.** *Let $P = (M, \mathcal{F}, A)$ be an Alloy problem and $T[P] = (\alpha_{M_S}, C_M, C_F, C_A)$ its RFOL translation. If $T[P]$ has a counterexample, then P has a counterexample too.*

Proof. For a given counterexample $\mathcal{M}$ of $T[P]$ —an RFOL structure of $\neg(C_{\mathcal{M}} \wedge C_{\mathcal{F}} \to C_A)$, we show that $\mathcal{I}_{\mathcal{M}}$ is a counterexample of P —an Alloy instance of $\neg(\bigwedge M_{\mathcal{C}} \wedge \bigwedge \mathcal{F} \to A)$ that is conform to $M_{\mathcal{S}}$. Since $\mathcal{C}[M_{\mathcal{C}}] = C_{\mathcal{M}}$, $\mathcal{C}[\mathcal{F}] = C_{\mathcal{F}}$, $\mathcal{C}[A] = C_A$, and all logical connectives and arithmetic operations has in both logics their intended semantics, it is sufficient to show that

(1) $\mathcal{I}_{\mathcal{M}}$ is a well-defined Alloy instance for P, and

(2) for each Alloy formula φ, $M[\varphi]I_{\mathcal{M}}$ iff $\mathcal{M}, \beta \models \mathcal{C}[\varphi]$

, where $\beta(Na[x]) = I_{\mathcal{M}}(x)$ for all Alloy variables.

The sorting function α_{M_S} together with the formula $C_{\mathcal{M}}$ guarantee that any RFOL structure of $T[P]$ is conform to all model constraints of P on the translated Alloy symbols. Consequently, construction rule 5.18 guarantees the same for $\mathcal{I}_{\mathcal{M}}$. Together with Theorem 2, $\mathcal{I}_{\mathcal{M}}$ is a well-defined Alloy instance for P.

(2) follows directly from corollary 2. □

5.4 Evaluation

The main feature of the Alloy to RFOL translation presented here is that it guarantees equisatisfiability of the produced RFOL proof obligation with respect to its Alloy counterpart. Consequently, the result of analysing the RFOL proof obligation of an Alloy problem with an SMT solver, whether showing validity —denoted "proved"— or providing a counterexample —denoted "CE", is faithful. However, as we will show in this section, this is done at the higher expense of efficiency, especially for incorrect Alloy problems.

Table 5.2 shows the results of analyzing our RFOL translation of a set of Alloy problems with the Z3 SMT solver. The time (in second) is measured on an Intel Core2Quad, 2.8GHz, 8GB memory. Time out (TO) is 600 seconds. We have analyzed 20 assertions in 8 *Alloy system*[8] specifications: the address book of an email client where aliases and groups are allowed, the query interface and aggregation mechanism of Microsoft COM, the operations of a memory accessed by abstract addresses, a system for managing media files, the mark and sweep garbage collection algorithm, the own-grandpa puzzle, a hand shaking protocol among spouses, and the queens

[8] an Alloy problem without the assertion to prove —i.e., model plus facts

arrangement puzzle for an $n \times n$ chessboard. Beside the n-Queens problem, all considered problems are included in the Alloy 4 distribution, and represent various combinations of hierarchical types, nested relational joins, transitive closure, nested quantifiers, set cardinality, and arithmetic operations.

Whereas, three of the Alloy problems expected to be valid (top part of the table) were proven correct —i.e., the RFOL translation of their negated Alloy proof obligations are unsatisfiable, none of the problems expected to be invalid (bottom part of the table) could be proven invalid —i.e., the RFOL translation of their negated Alloy proof obligations have satisfiable structures. To further investigate these results, we additionally analysed all 8 Alloy systems with the empty assertion and noticed that even with the empty assertion all RFOL proof obligations times out. This confirms the hypothesis that the limited success of the basic RFOL translation is due to its *general* completeness feature.

The completeness feature of our translation, as proven in Theorem 4, restricts the valid structures $\mathcal{M}$ to those where the cardinality of the universe of each sort Rel_i is the power set of $(\bar{M}^{Atom})^i$, where $\bar{M}^{Atom}$ is the universe of the sort $Atom$. This radically hampers the main task of the SMT solver —finding satisfiable structures— in two respects: (1) it makes a huge number of the structures constructed and checked by the solver —since not known a priory— invalid only because of the required cardinality restriction and (2) the universes of structures fulfilling the cardinality restriction are in general so huge such that checking their validity with respect to the Ax axioms and the input formula is practically impossible. The fact that this approach could prove three of the Alloy problems correct, is, however, due to the potential ability of SMT solvers of deducing *general* contradictions while constructing and checking structures.

In the next two chapters, we will present two approaches for reducing this structure restriction, while preserving the completeness feature of the translation. In Chapter 6, we present our *Semantics Blasting* technique which is tailored to the exact structure restriction problem presented in this section, albeit in a general setting. In Chapter 7 we present our *Sufficient Ground Term Sets* technique which handles the more general task of quantified variables elimination in non arithmetic theories.

		RFOL + Z3	
Problem	Assertion	Time (sec)	Result
address book	delUndoesAdd	0.02	proved
	addIdempotent	0.02	proved
abstract memory	writeRead	TO	–
	writeIdempotent	TO	–
COM	theorem1	TO	–
	theorem2	TO	–
	theorem3	TO	–
	theorem4a	TO	–
	theorem4b	TO	–
mark sweep	soundness1	TO	–
	soundness2	TO	–
	completeness	TO	–
media assets	hidePreservesInv	0.01	proved
	pasteAffectsHidden	TO	–
n-Queen	solCondition	TO	–
abstract memory	empty	TO	–
address book	empty	TO	–
	addLocal	TO	–
COM	empty	TO	–
handshake	empty	TO	–
	puzzle	TO	–
mark sweep	empty	TO	–
media assets	empty	TO	–
	cutPaste	TO	–
n-Queen	empty	TO	–
	15Queens	TO	–
own grandpa	empty	TO	–
	ownGrandpa	TO	–

Table 5.2: Evaluation results

Chapter 6

Semantics Blasting

Some theories make strong constraints on the cardinality on sort universes of their (valid) structures. The *complete* axiomatization of such theories —by providing complete axiomatization of their symbols, encodes this universe cardinality constraints automatically in the axiomatization. However, for most such theories, there exists an importantly large fragment of formulas for which the waiving of the universe cardinality constraints does not affect the completeness —i.e., if there exists a structure that satisfies a formula f but the universe cardinality constraints, then there exists another structure that satisfies f together with the universe cardinality constraints.

A prominent example of such theories is the Array theory modeling the heap of Java programs [77]. This theory consists of the sorts $Heap$ for heaps, $Field$ for fields, Obj for objects, and Val for the super sort of all possible fields values; two operators $sto : Heap \times Obj \times Field \times Val \rightarrow Heap$, for constructing the resulting heap of a store statement and $sel : Heap \times Obj \times Field \rightarrow Val$ for reading the field value of an object at a given heap with usual semantics. In this theory example the cardinality of the universe of the sort $Heap$ is required to be $|\bar{Val}|^{|\bar{Obj}|*|\bar{Field}|}$ which becomes for reasonably large —even for the smallest— universes of Obj, $Field$, and Val impractically large.

This theory characteristic becomes important when it comes to checking the satisfiability of formulas modulo such theories using an SMT solver. Abstractly seen, the structure search of SMT solvers can be divided into three major steps: (1) the construction of sort universes of preferably minimal cardinalities with respect to the constants of the input formula f and theory axioms Ax, (2) the search of an interpretation that satisfies f together with the theory axioms Ax, and (3) the enlarging of some sort universes, if no satisfying interpretation can be found in the previous step. Consequently, the universes cardinality constrains hamper the structure search of the SMT solver in two respects: (1) a huge number of structures become invalid *only* because of the universes cardinality constraints —they are although considered, since not known a priory— and (2) the structures satisfying the cardinality constraints have such huge universes, that checking their satisfaction of f together with the theory axioms Ax is practically impossible.

To solve this problem we developed a preprocessing[1] which in a first step applies a selected set of theory axioms on-demand to all theory corresponding symbols of the input formula f and in a second step reduces the theory axioms Ax by the selected and applied axioms. We call the first step *semantics blastings* (SB) and the second *freeness*. This way of modeling the semantics of theory symbols on-demand was implicitly used in several approaches such as [47, 30, 45], but without discussing its generality and the condition under which it is complete. This is important since in general the extensionality property is required and the freeness step is indeed a source of incompleteness.

In the following we describe the SB transformation as a prioritized rule set and describe a complete fragment for the SB transformation together with the freeness step.

6.1 Semantics Blasting Rules

We first define theories *admitting* semantics blasting and provide a general rule based procedure to perform semantics blasting for them.

Let T be a sort, E and V two further sorts different from T and $X = (E \times V) \cup (T \times E \times V) \cup (T \times T)$ a union of sorts. Let further Con be the set of all T-constructors $con_i : X \to T$ and $\epsilon : T \times E \to V$ the T-observer function[2]. The sort T is called depending on the constructors axiomatization *Inductive* or *Co-Inductive* [68].

Definition 15 (SB theories)**.** A theory $Cl(Ax)$ admits semantics blasting of Con if

- for each constructor $con_i \in Con$ there exists a single semantics axiom $A_i \in Ax$ of the form $\forall x : X, e : E.\ \epsilon(con_i(x), e) = \Phi_i$ where con_i do not occurs in Φ_i, neither directly or indirectly —the set of all constructor axioms is denoted by Ax_{Con},
- T is extensional with respect to its observer ϵ modulo $Cl(Ax)$ —i.e, $Ax \models \forall t_1, t_2 : T.\ (\forall e : E.\ \epsilon(t_1, e) = \epsilon(t_2, e)) \to t_1 = t_2$, and
- for each structure $\mathcal{M}$ of Ax_{Con} and $f : \bar{M}^E \to \bar{M}^V$, there exists $\bar{A} \in \bar{M}^T$ such that $f = \lambda \bar{e}.M(\epsilon)(\bar{A}, \bar{e})$[3].

Although the last condition of the definition is not mandatory to apply our semantics blasting procedure, it first causes the problem of strong cardinality constraints on universes observed in 5.4 and motivated in the introduction of this chapter.

Given a theory admitting SB we provide a procedure that constructs from formula F a T-constructors-free formula F^{SB}, such that, F and F^{SB} are equisatisfiable modulo $Cl(Ax)$. Figure 6.1 shows a system of prioritized transformation rules of our SB procedure.

[1] Before SMT solving

[2] Since it is the usual case, we restrict w.l.o.g. the number of T-observers to one.

[3] Henceforth, we use lambda abstractions to introduce unnamed functions.

DefCons$_i$:

$$\epsilon(con_i(x'),e') \rightsquigarrow \Phi_i[x'/x,e'/e]$$

PullOut$_i$:

$$F \rightsquigarrow freshT = con_i(x) \rightarrow F[freshT/con_i(x)]$$

, where $con_i(x)$ occurs in F and $freshT$ is fresh skolem function[4]

Extens:

$$t_1 = t_2 \rightsquigarrow \forall e : E.\, \epsilon(t_1,e) = \epsilon(t_2,e)$$

Figure 6.1: Rules of the SB procedure

The first rule DefCons$_i$ is nothing else than the application of the axioms A_i of the T-constructor con_i —by replacing its left-had side with the right-hand side. The second rule PullOut$_i$, handles the case in which a T-constructor con_i occurs in F not as argument of the T-observer ϵ. In this case, the rule replace the occurrence of con_i with a fresh T function $freshT$ and adds its definition —equality to con_i— as an assumption of F. The last rule Extens reduces the equality between T terms —especially the one introduced from the second rule— to the equality of all their observations with ϵ. Our SB-procedure consists of applying this rules exhaustively and in the presented order, i.e., each rule is only applied if the previous rule is not applicable. One can easy see that applying this rules exhaustively in the presented order guaranties termination and the result is T-constructor free.

Theorem 5 (SB procedure)**.** *Given a theory $Cl(Ax)$ admitting SB and formula F, the result formula F^{SB} of applying the SB-procedure to F is T-constructor free and equisatifiable to F modulo $Cl(Ax)$.*

Proof. Since we already know that the SB-procedure always terminates and its result F^{SB} is T-constructor free, it suffices to prove that all rules of the SB-procedure are equisatisfable transformation modulo $Cl(Ax)$. This is true for the first rule since $A_i \in Ax$ for all $con_i \in Con$ and for the second rule since T is extensional with respect to ϵ modulo $Cl(Ax)$. The second rule is an equisatisiable transformation even modulo the empty theory (first-order logic with equality). □

In the following we present some concrete theories which admit semantics blasting:

1. The extensional array theory for Java heaps:

$$\begin{aligned}
T &:= Heap, E := (Obj \times Field), V := Val \\
\epsilon &:= sel : Heap \times (Obj \times Field) \to Val \\
Con &:= \{sto : (Heap \times Obj \times Field \times Val) \to Heap\} \\
Ax &:= \{\forall h : Heap, o, o' : Obj, f, f' : Field, v : Val. \\
&\qquad (o' = o \wedge f' = f) \to sel(sto(h,o,f,v),o',f') = v \wedge \\
&\qquad (o' \neq o \vee f' \neq f) \to sel(sto(h,o,f,v),o',f') = sel(h,o',f'), \\
&\quad \forall h, h' : Heap.\ (\forall o : Obj, f : Field.\ sel(h,o,f) = sel(h',o,f)) \to h = h'\}
\end{aligned}$$

2. Any sub theory of our RFOL (see Chapter 4) for fixed relational arity i:

$$\begin{aligned}
T &:= Rel_i, E := Atom^i, V := Bool \\
\epsilon &:= \in : Rel_i \times Atom^i \to Bool \\
Con &:= \{\mathrm{sin}_i : Atom^i \to Rel_i, \cup_i : Rel_i \times Rel_i \to Rel_i, \cdots\} \\
Ax &:= \text{ RFOL axioms for relational arity } i \text{ (see Figure 4.2)}
\end{aligned}$$

6.2 The SB$^+$ Complete Fragment

So far our SB procedure only performs equisatisfiable transformations, and thus does not affect completeness. But, in order to reduce the discussed universes cardinality constraints —possibly encoded in Ax— a non completeness preserving transformation is further needed. Namely, the reduction of the background theory $Cl(Ax)$ by all T-constructors axioms Ax_{Con}. We call this step *freeness* and denote the extension of the SB procedure with this step SB$^+$.

Hereafter, F denotes the Skolem normal form of the negation of the original input formula F^{org} and F^{SB} the results of applying SB to F. We further assume without loss of generality that $F := H \wedge G$ with G is T-quantifier-free.

One can already see that SB$^+$ is correct in the sense that if F^{SB} is unsatisfiable modulo $Cl(Ax \setminus Ax_{Con})$ then F is unsatisfiable too modulo $Cl(Ax)$ —i.e., F^{org} is valid. However, if F^{SB} is satisfiable modulo $Cl(Ax \setminus A_{Con})$, we cannot transfer this result to F, because of the non guaranteed completeness of SB$^+$. Therefore, it is curial to determine the complete fragment of SB$^+$ in order to use it in the general context.

Before starting with the description of *the* complete fragment of SB$^+$, we give an example of a formula which is not in the SB$^+$ fragment of the set theory —i.e., RFOL with $N = 1$. Let us consider the formula

$$F := \forall s : Rel_1. \exists x : Atom.\ \epsilon(\cup_1(A,B),x) \wedge \neg\epsilon(s,x)$$

where A and B are constant sets. One can easy see that F is unsatisfiable —even modulo the empty theory, since instantiating s with $\cup_1(A,B)$ is equivalent to $false$. However, applying our SB$^+$ procedure to F results in

$$F^{\text{SB}} := \forall s : Rel_1.\exists x : Atom.\ (\epsilon(A,x) \vee \epsilon(B,x)) \wedge \neg\epsilon(s,x)$$

which is satisfiable modulo $Cl(Ax \setminus A_{\cup_i})$. For example the structure $\mathcal{M}$ with

$$\bar{M}^{Rel_1} := \{\bar{A},\bar{B}\}, \bar{M}^{Atom} := \{\bar{a},\bar{b}\}$$
$$M(\epsilon)(\bar{A},v) := \begin{cases} tt & \text{if } v = \bar{a} \\ ff & \text{else} \end{cases}$$
$$M(\epsilon)(\bar{B},v) := \begin{cases} tt & \text{if } v = \bar{b} \\ ff & \text{else} \end{cases}$$

is a satisfying structure for F^{SB} modulo $Cl(Ax \setminus A_{\cup_i})$.

In the following, we define some auxiliary definitions and lemmas needed for the description and proof of the complete fragment of SB$^+$.

Definition 16 (Structure enlargin w.r.t. Ax_{Con})**.** Suppose F has a structure $\mathcal{M}$, we construct the structure $\mathcal{M}_L$ as follow:

$$\bar{M}_L^S := \begin{cases} \{f \mid f : \bar{M}^E \to \bar{M}^V\} & \text{if } S = T \\ \bar{M}^S & \text{otherwise} \end{cases}$$
$$M_L(f)(v) := \begin{cases} \lambda\bar{e}.M(\epsilon)(M(f),\bar{e}) & \text{if } \alpha(f) = T \\ g(\bar{e}) & \text{if } f = \epsilon \wedge v = (g,\bar{e}) \\ \lambda\bar{e}.val_{M_L,\beta^{v\bar{e}}_{x\bar{e}}}(\Phi_i) & \text{if } f = con_i \\ M(f)(v) & \text{otherwise} \end{cases}$$

Definition 16 provides a tool for an isomorphic enlargement $\mathcal{M}_L$ of a given structure $\mathcal{M}$ with respect to the T-universe. It targets: (1) the satisfaction of all constructor axioms in Ax_{con} and consequently the last condition of SB theories (see Definition 15); (2) the preservation of the satisfaction of T-quantifier free formulas with respect to $\mathcal{M}$. These two properties of the $\mathcal{M}_L$ structure are essential for the proof of the SB$^+$ fragment.

Lemma 9. *For each structure $\mathcal{M}$, $\mathcal{M}_L$ is a structure for $(\bigwedge_{A_i \in Ax_{Con}} A_i)$.*

Proof. It is sufficient to prove that $\mathcal{M}_L \models A_i$ for an arbitrary $A_i \in Ax_{Con}$. Let $\beta : Var \to \bar{M}_L$ be an arbitrary variable assignment, then we need to prove that $val_{M_L,\beta}(\epsilon(con_i(x),e)) = val_{M_L,\beta}(\Phi_i)$.

$$\begin{aligned}
& val_{M_L,\beta}(\epsilon(con_i(x),e)) \\
\overset{val}{\Longleftrightarrow}\ & M_L(\epsilon)(M_L(con_i)(\beta(x)),\beta(e))) \\
\overset{\mathcal{M}_L}{\Longleftrightarrow}\ & M_L(\epsilon)(\lambda e.val_{M_L,\beta}(\Phi_i),\beta(e))) \\
\overset{\mathcal{M}_L}{\Longleftrightarrow}\ & [\lambda e.val_{M_L,\beta}(\Phi_i)](\beta(e)) \\
\overset{\lambda}{\Longleftrightarrow}\ & val_{M_L,\beta}(\Phi_i)
\end{aligned}$$

□

In order to discuss the preservation of the satisfaction of formulas with $\mathcal{M}$ in $\mathcal{M}_L$, we first need to stablish an embedding of $\mathcal{M}$ in $\mathcal{M}_L$.

Definition 17 ($\mathcal{M}$ embedding in $\mathcal{M}_L$)**.** For each structure $\mathcal{M}$, the $\mathcal{M}_L$ construction deduces a canonical embedding $emb : \bar{M} \to \bar{M}_L$ from $\mathcal{M}$ to $\mathcal{M}_L$ with:

$$emb(v) := \begin{cases} \lambda \bar{e}.M(\epsilon)(v,\bar{e}) & \text{if } v \in \bar{M}^T \\ v & \text{else} \end{cases}$$

Lemma 10. *For each structure $\mathcal{M}$ satisfying $Ax \setminus Ax_{Con}$, the embedding emb of $\mathcal{M}$ in $\mathcal{M}_L$ is injective.*

Proof. The claim follows directly from the extensionality of the sort T with respect to its ϵ in $\mathcal{M}$. Since T is extensional with respect to ϵ modulo $Cl(Ax)$ (see. Definition 15) and Ax_{Con} only contains T-constructor axioms, T is extensional with respect to ϵ modulo $Cl(Ax \setminus Ax_{Con})$ and thus in $\mathcal{M}$. □

Lemma 11. *For each structure $\mathcal{M}$ of Ax, the embedding emb of $\mathcal{M}$ in $\mathcal{M}_L$ is even a bijection.*

Proof. Because of lemma 10, it suffices to prove the surjectivity of *emb*. The surjectivity of *emb*, however, follows directly from the last condition on SB theories (see. Definition 15), since $\mathcal{M}$ satisfies Ax. □

Lemma 12. *For each T-quantifier and T-constructor free term s, $val_{\mathcal{M}_L,\beta}(s) = emb(val_{M,\beta}(s))$.*

Proof. By induction on the complexity of s.

Case $s = f$, where $\alpha(f) = T$:

$$\begin{array}{ll} & val_{\mathcal{M}_L,\beta}(f) \\ \overset{val}{=} & M_L(f) \\ \overset{M_L}{=} & \lambda\bar{e}.M_L(\epsilon)(M(f),\bar{e}) \\ \overset{emb}{=} & emb(M(f)) \\ \overset{val}{=} & emb(val_{\mathcal{M},\beta}(f)) \end{array}$$

Case $s = \epsilon(f,e)$, where $\alpha(f) = T$:

$$\begin{array}{ll} & val_{\mathcal{M}_L,\beta}(\epsilon(f,e)) \\ \overset{val}{=} & M_L(\epsilon)(val_{\mathcal{M}_L,\beta}(f),val_{\mathcal{M}_L,\beta}(e)) \\ \overset{M_L}{=} & val_{\mathcal{M}_L,\beta}(f)(val_{\mathcal{M}_L,\beta}(e)) \\ \overset{\alpha(f)=T}{=} & \mathcal{M}_L(f)(val_{\mathcal{M}_L,\beta}(e)) \\ \overset{M_L}{=} & [\lambda\bar{e}.M(\epsilon)(M(f),\bar{e})](val_{\mathcal{M}_L,\beta}(e)) \\ \overset{\lambda}{=} & M(\epsilon)(M(f),val_{\mathcal{M}_L,\beta}(e)) \\ \overset{val}{=} & val_{\mathcal{M},\beta}(\epsilon(f,e)) \\ \overset{emb}{=} & emb(val_{\mathcal{M},\beta}(\epsilon(f,e))) \end{array}$$

Otherwise w.l.o.g. $s = f(t_{1:n})$ with $f \notin \{\epsilon, con_i\}$, $n > 0$, and $t_i \notin Term^T_\Sigma$ for $1 \leq i \leq n$:

$$\begin{array}{ll} & val_{\mathcal{M}_L,\beta}(f(t_{1:n})) \\ \overset{val}{=} & M_L(f)(val_{\mathcal{M}_L,\beta}(t_1),\ldots,val_{\mathcal{M}_L,\beta}(t_n)) \\ \overset{IH}{=} & M_L(f)(emb(val_{\mathcal{M},\beta}(t_1)),\ldots,emb(val_{\mathcal{M},\beta}(t_n))) \\ \overset{emb}{=} & M_L(f)(val_{\mathcal{M},\beta}(t_1),\ldots,val_{\mathcal{M},\beta}(t_n)) \\ \overset{M_L}{=} & M(f)(val_{\mathcal{M},\beta}(t_1),\ldots,val_{\mathcal{M},\beta}(t_n)) \\ \overset{val}{=} & val_{\mathcal{M},\beta}(f(t_{1:n})) \\ \overset{emb}{=} & emb(val_{\mathcal{M},\beta}(f(t_{1:n}))) \end{array}$$

□

Lemma 13. *If a structure $\mathcal{M}$ satisfies Ax, then $val_{\mathcal{M}_L,\beta}(s) = emb(val_{M,\beta'}(s))$ for $\beta' = \lambda x.emb^{-1}(\beta(x))$ and s an arbitrary term.*

Proof. By induction on the complexity of s. Because of lemma 12, it suffices to consider the following cases:

Case $s = x$, a variable of type T:

$$\begin{array}{rl} & val_{\mathcal{M}_L,\beta}(x) \\ \overset{val}{=} & \beta(x) \\ \overset{inj.\ emb}{=} & emb(emb^{-1}(\beta(x))) \\ \overset{\beta'}{=} & emb(\beta'(x)) \\ \overset{val}{=} & emb(val_{\mathcal{M},\beta'}(x)) \end{array}$$

Case $s = con_i(t_{1:n})$:

$$\begin{array}{rl} & val_{\mathcal{M}_L,\beta}(con_i(t_{1:n})) \\ \overset{val}{=} & \mathcal{M}_L(con_i)(val_{\mathcal{M}_L,\beta}(t_1),\ldots,val_{\mathcal{M}_L,\beta}(t_n)) \\ \overset{M_L}{=} & \lambda\bar{e}.val_{\mathcal{M}_L,\beta'^{\bar{e}}_{\bar{e}}}(\Phi_i[t_1/x_1,\ldots,t_n/x_n]) \\ \overset{IH}{=} & \lambda\bar{e}.emb(val_{\mathcal{M},\beta'^{\bar{e}}_{\bar{e}}}(\Phi_i[t_1/x_1,\ldots,t_n/x_n])) \\ \overset{emb}{=} & \lambda\bar{e}.val_{\mathcal{M},\beta'^{\bar{e}}_{\bar{e}}}(\Phi_i[t_1/x_1,\ldots,t_n/x_n]) \\ \overset{M\models A_i}{=} & \lambda\bar{e}.val_{\mathcal{M},\beta'^{\bar{e}}_{\bar{e}}}(\epsilon(con_i(t_{1:n}),e)) \\ \overset{val}{=} & \lambda\bar{e}.M(\epsilon)(val_{\mathcal{M},\beta'}(con_i(t_{1:n})),\bar{e}) \\ \overset{emb}{=} & emb(val_{\mathcal{M},\beta'}(con_i(t_{1:n}))) \end{array}$$

□

Corollary 3 (Theory equalization)**.** *In order to show that a formula F modulo a theory $Cl(S_1)$ is equisatifiable to an other formula F' modulo the theory $Cl(S_1 \setminus S_2)$, it is sufficient to show that $(\bigwedge_{\Phi\in S_2}\Phi) \wedge F$ modulo $Cl(S_1 \setminus S_2)$ is equisatifiable to F' modulo $Cl(S_1 \setminus S_2)$.*

Proof. This corollary is a simple result of the deduction theorem (ded-theo) applied to the satisfiability of F modulo $Cl(S_1)$.

$$\begin{aligned}
& \exists \mathcal{M} \in Cl(S_1).\ \mathcal{M} \models F \\
\iff\ & \neg(S_1 \models \neg F) \\
\iff\ & \neg((S_1 \setminus S_2) \cup S_2 \models \neg F) \\
\overset{ded\text{-}theo}{\iff}\ & \neg(S_1 \setminus S_2 \models (\bigwedge_{\Phi \in S_2} \Phi) \rightarrow \neg F) \\
\iff\ & \exists \mathcal{M} \in Cl(S_1 \setminus S_2).\ \mathcal{M} \models \neg((\bigwedge_{\Phi \in S_2} \Phi) \rightarrow \neg F) \\
\iff\ & \exists \mathcal{M} \in Cl(S_1 \setminus S_2).\ \mathcal{M} \models (\bigwedge_{\Phi \in S_2} \Phi) \wedge F
\end{aligned}$$

□

Theorem 6 (The SB$^+$ completeness fragment)**.** *Let $Cl(Ax)$ be a first-order theory admitting SB, and $F := H \wedge G$ a first-order formula with G is T-quantifier-free. Then, F modulo $Cl(Ax)$ is equisatisfiable to $F^{SB} := SB(H) \wedge SB(G)$ modulo $Cl(Ax \setminus Ax_{Con})$ iff*

- *F^{SB} is unsatisfiable modulo $Cl(Ax \setminus Ax_{Con})$ or*
- *$\mathcal{M}_L \models SB(H)$ for some structure $\mathcal{M}$ of F^{SB}.*

Proof. Using lemma 3 it is sufficient to proof (1) if $SB(H) \wedge SB(G)$ is unsatisfiable so $F' := (\bigwedge_{Ax_i \in Axs} Ax_i) \wedge H \wedge G$ is unsatisfiable too and (2) if $SB(H) \wedge SB(G)$ has a model modulo $Cl(Ax \setminus Ax_{Con})$ so F' has a model modulo $Cl(Ax \setminus Ax_{Con})$ too. The first case is trivial since $F' \rightarrow SB(H) \wedge SB(G)$. For the second case, we assume a structure $\mathcal{M}$ in $Cl(Ax \setminus Ax_{Con})$ of $SB(H) \wedge SB(G)$ and prove that $\mathcal{M}_L$ is a structure for F'.

Because of lemma 9, $\mathcal{M}_L$ is a satisfying structure for Ax. It remains to prove that $\mathcal{M}_L$ is also a satisfying structure for $SB(G)$ and $SB(H)$. Since $SB(G)$ is T-quantifier and T-constructor free, we get from lemma 12 that $val_{\mathcal{M}_L,\beta}(SB(G)) = emb(val_{M,\beta}(SB(G)))$. Since $\mathcal{M}$ satisfies $(Ax \setminus Ax_{Con})$, we get from lemma 10 that *emb* is injective and thus $\mathcal{M}_L$ must also be a model of $SB(G)$. For $SB(H)$, even though, it is T-constructors-free, we cannot use lemma 12 since it can contain T-quantifiers. Also lemma 13 cannot be used, since $\mathcal{M}$ is not required to satisfy Ax. However, the claim holds from the conditions of the theorem, which require for this case that $\mathcal{M}_L \models SB(H)$.

So far we have proved that our fragment conditions are sufficient (from right to left). In order to show that the conditions are essential, we consider two cases: (1) F' and F^{SB} are both unsatisfiable modulo $Cl(Ax \setminus Ax_{Con})$ and (2) F' and F^{SB} are both satisfiable modulo $Cl(Ax \setminus Ax_{Con})$. The first case is trivial since explicitly covered by the first fragment condition. For the second case, we assume that $\mathcal{M}$ is a satisfying structure of F' and F^{SB} and show that the second fragment condition $\mathcal{M}_L \models SB(H)$

must hold. $\mathcal{M}$ is especially a satisfying structure of Ax and $SB(H)$. Because of lemma 13, we have that $val_{\mathcal{M}_L,\beta}(SB(H)) = emb(val_{M,\beta'}(SB(H)))$ for arbitrary β and for $\beta' = \lambda x.emb^{-1}(\beta(x))$. Since *emb* is injective —for this case even bijective— and $\mathcal{M}$ satisfies $SB(H)$, $\mathcal{M}_L$ satisfies $SB(H)$ too. □

Theorem 6 provides an entire description of the SB$^+$ fragment which guarantees the equisatisfiability of our SB$^+$ transformation with respect to the input formula. However, the conditions of the fragment are based rather on model checking than on SMT solving.

6.3 Practical Tools for the SB$^+$ Fragment

In the following, we present some practical tools for detecting formulas of the SB$^+$ fragment described in Theorem 6.

Proposition 1. *Let $Cl(Ax)$ be a first-order theory admitting SB. If F^{SB} is unsatisfiable modulo $Cl(Ax \setminus Ax_{Con})$, then F is in the SB$^+$ fragment of $Cl(Ax)$.*

Proof. This condition satisfies the SB$^+$ fragment conditions as in Theorem 6 and is thus trivially a sufficient condition for the fragment. □

Proposition 2. *Let $Cl(Ax)$ be a first-order theory admitting SB. Each T-quantifier-free formula F is in the SB$^+$ fragment of $Cl(Ax)$.*

Proof. The proposition follows directly from Theorem 6, since for T-quantifier free formulas, H and consequently $SB(H)$ are equivalent to true. □

Proposition 2 represents a simple and sufficient condition of the SB$^+$ fragment. The condition is purely syntactic. Although it captures only a small sub fragment of the SB$^+$ fragment, it has been shown that Proposition 2 describes one of the most frequently used fragment in practice (cf. Section 6.4).

Proposition 3. *Let $Cl(Axs)$ be a first-order theory admitting SB, and $F := H \wedge G$ a first-order formula with G is T-quantifier-free. Then, F is in the SB$^+$ fragment if*

$$(\bigwedge_{A_i \in Ax} A_i) \wedge SB(H) \text{ is satisfiable modulo } Cl(Ax \setminus Ax_{Con}).$$

Proof. We assume a satisfying structure $\mathcal{M}$ of $(\bigwedge_{A_i \in Ax} A_i) \wedge SB(H)$. Consequently, $\mathcal{M}$ is a satisfying structure for $(\bigwedge_{A_i \in Ax} A_i)$ and for $SB(H)$. Because $\mathcal{M}$ is a structure for $(\bigwedge_{A_i \in Ax} A_i)$, we conclude by means of lemma 13 that $val_{\mathcal{M}_L,\beta}(SB(H)) = emb(val_{M,\beta'}(SB(H)))$ where *emb* is bijective and β arbitrary. Since $\mathcal{M}$ is a structure for $SB(H)$, $\mathcal{M}_L$ is also a structure for $SB(H)$ which satisfies the SB$^+$ fragment conditions as in Theorem 6. □

Proposition 3 provides especially a suitable interpretation of the second condition of Theorem 6 for SMT solving.

All three propositions together propose a cost effective testing system for inclusion in the SB$^+$ fragment. Namely: (1) if F^{SB} is unsatisfiable modulo $Cl(Ax \setminus Ax_{Con})$, then the result can be directly trusted and transfered to the original proof obligation, (2) if F^{SB} is satisfiable modulo $Cl(Ax \setminus Ax_{Con})$, then we first need to show the inclusion of F in the SB$^+$ fragment by successively using the checks of proposition 2 and proposition 3 respectively, (3) if F passes one of these tests, then the result can be trusted and transfered to the original proof obligation, otherwise not.

6.4 Evaluation

We have evaluated our SB$^+$ technique on RFOL proof obligations resulting from the RFOL translation of Alloy problems described in Chapter 5. Thereby, all RFOL first-order relational theories of fixed arity —i.e., all Rel_i with corresponding operators and axioms— were subjects to semantics blasting. Therefore, we applied the SB$^+$ transformation to the RFOL proof obligations of all 28 Alloy problems considered and described in Section 5.4 —including the one with empty assertions— and solved the transformation outcomes using the Z3 solver.

Table 6.1 shows the results and also (1) compares them to solving the RFOL proof obligations without applying the SB$^+$ transformation (cf. Table 5.2) and (2) reports on the performance of the Alloy Analyzer. The time (in seconds) is measured on an Intel Core2Quad, 2.8GHz, 8GB memory. The Alloy analysis time is the total time spent on generating CNF and solving it using the SAT4J solver. The Z3 analysis time is what it reports using the -st option. The assertions at the top part of the table are expected to be valid[5] and the ones at the bottom part are expected to have counterexamples. The scope column denotes for the valid assertions the maximum scope for which the Alloy Analyzer can check the assertion before reaching the time-out[6] of 600 seconds and for the invalid assertions it gives the smallest scope required by the Alloy Analyzer to find a counterexample. The result column gives the outcome of the Z3 analysis: *proved* if it returns "unsat" when looking for a counterexample, implying that the assertion is successfully proven, *sound CE* if it returns a sound counterexample, and *false CE* if the counterexample is spurious.

Out of the 15 valid assertions, 12 were proven correct using the SB$^+$ technique in less than 1 second, whereby only 3 of them could be proven correct without using the SB$^+$ technique. The same observation holds for the invalid assertions. In this case the SB$^+$ technique could help finding *sound* counterexamples for all assertions except

[5]Their Alloy models contain developer's comments that no counterexamples are expected.

[6]We set the memory and stack usage for the Alloy Analyzer to the maximum and do not distinguish between out of memories and actual out of times.

		ALLOY ANALYZER		RFOL		RFOL + SB$^+$	
PROBLEM	ASSERTION	SCOPE	TIME	TIME	RESULT	TIME	RESULT
abstract memory	writeRead	44	179.44	0.02	proved	0.00	proved
	writeIdempotent	29	98.67	0.02	proved	0.03	proved
address book	delUndoesAdd	31	80.91	TO	–	0.00	proved
	addIdempotent	31	112.66	TO	–	0.01	proved
COM	theorem1	17	538.11	TO	–	0.00	proved
	theorem2	17	552.44	TO	–	0.00	proved
	theorem3	17	513.11	TO	–	0.00	proved
	theorem4a	17	534.78	TO	–	0.00	proved
	theorem4b	17	507.40	TO	–	0.00	proved
mark sweep	soundness1	16	112.75	TO	–	TO	–
	soundness2	9	341.36	TO	–	TO	–
	completeness	8	164.22	TO	–	TO	–
media assets	hidePreservesInv	58	13.02	0.01	proved	0.00	proved
	pasteAffectsHidden	47	511.79	TO	–	0.00	proved
nQueen	solCondition	73	173.51	TO	–	0.05	proved
abstract memory	empty	1	0.00	TO	–	0.01	sound CE
address book	empty	1	0.01	TO	–	0.01	sound CE
	addLocal	3	0.05	TO	–	0.10	sound CE
COM	empty	1	0.03	TO	–	0.01	sound CE
handshake	empty	1	0.01	TO	–	0.25	sound CE
	puzzle	10	2.47	TO	–	TO	–
mark sweep	empty	1	0.01	TO	–	0.01	sound CE
media assets	empty	1	0.02	TO	–	0.01	sound CE
	cutPaste	3	0.19	TO	–	0.06	sound CE
nQueen	empty	1	0.07	TO	–	0.01	sound CE
	15Queens	15	4.95	TO	–	13.53	sound CE
own grandpa	empty	1	0.01	TO	–	0.01	sound CE
	ownGrandpa	4	0.01	TO	–	0.12	sound CE

Table 6.1: Evaluation results

for the puzzle assertion of the handshake system, whereby without using SB$^+$ non of them could be found. It should be noted that the use of the SB$^+$ technique can lead to spurious counterexamples if the considered proof obligation is not in the SB$^+$ fragment (see Section 6.2 and 6.3). However, all considered benchmarks fit in the SB$^+$ fragment and to our surprise even in the easy to test sub fragment defined in proposition 2. This property of the benchmarks together with proposition 2, allows us to guarantee the soundness of all found counterexamples without the need of checking them.

Overall, the results clearly shows that, first, the use of the SB$^+$ technique makes the analysis of the RFOL proof obligations —resulting from RFOL translation of Alloy problems— practical and efficient. However, the 3 non-proved assertions of the mark sweep system, reveal an other restriction of RFOL proof obligations which is more general than SMT solving efficiency. It concerns, namely, the general limitation of SMT solvers in reasoning about recursive theories, in our case these are the transitive

closure theory and the cardinality theory. In Chapter 8, we discuss and present an approach capable to solve the case of the transitive closure theory.

Although the development of the SB^+ technique was mainly motivated and conducted by the RFOL relational theories, the technique is general enough to be applied to any theory admitting SB (see Definition 15). In order to confirm this claim, we report on the results of applying the SB^+ technique to the array theory of Java Heap in the context of *bounded* SMT proof obligations of proof branches of the KeY System [12, 77] — a tool that integrate design, implementation, formal specification, and formal verification of Java software. This SB^+ application was done by Mihai Herda in the context of his master thesis [45] in our group. He applied the SB^+ technique to 38 bounded SMT proof obligations of 29 provable proof branches and 9 open proof branches —expected to be invalid— of 11 different Java methods. He set the bound —the maximal number of elements for sorts including integers— to 8 and the time-out of the Z3 SMT solver to 180 seconds.

Also, for this theory and setting, the efficiency and need of the SB^+ technique was clearly demonstrated. Using SB^+, out of the 29 valid benchmarks, the SMT solver could proof the *bounded validity*[7] of 27 benchmarks, whereas without using the SB^+ technique, the SMT solver times out on all benchmarks. For the invalid benchmarks, the use of the SB^+ technique could help finding sound counterexamples for all 9 benchmarks. Also here, it has been observed that all benchmarks belong to the SB^+ sub fragment described in proposition 2 and thus the soundness of all found counterexamples was automatically guaranteed. However, without using the SB^+ technique, the SMT solver proved the bounded validity of all 9 invalid benchmarks. At the first sight, this would appear wrong, but it is not, it is a consequence of the already discussed implicit cardinality constraints on valid structures universes induced by the theory axioms. This means that without applying SB^+, one has to choose for the sort Heap a bound of 8^{64} —assuming that the bound of the other sorts is 8, which can, in practice, never be handled by an SMT solver, regardless of time-out.

6.5 Related Work

Several approaches have addressed the verification of Alloy problems in general. Due to the undecidability of the Alloy language, most general approaches are based on interactive theorem proving.

Prioni [5] translates Alloy formulas into proof obligations of the first-order logic of the Athena [6] theorem prover. The relational reasoning calculus is build using polymorphic sets for relations, polymorphic functions for relational operations and first-order axioms to stat the semantics.

Dynamite [37] uses proper fork algebras [36] —algebras of binary relations over a structured set— as an alternative semantics for relational binary fragment of Alloy. Higher arity Alloy relations are reduced injectively to binary relations by means of

[7] Denotes the validity of a formula within a given bound –i.e., no counterexample exists within the given bound.

an injective function, leading intermediate expressions that are hard to understand. Dynamite uses the higher-order interactive theorem prover PVS [63] to perform reasoning in the proposed semantics.

Kelloy [75], developed in our group, is the last one in this series of interactive theorem provers for Alloy. Kelloy translates Alloy to first-order logic of the KeY system with built-in support of integer theory. Unlike Prioni and Dynamite, Kelloy targets the support of the complete Alloy language including important libraries as the ordering library. Kelloy also verifies Alloy assertions in both finite and infinite domains.

Compared to interactive theorem provers that provide refutationally complete calculi but are not fully automatic, SMT solvers are fully automatic, but may fail to refute refutable quantified formulas. Recent SMT solvers, however, have shown significant advances in handling quantifiers. Solvers like haRVey and Z3, integrated the quantifier handling of saturation based theorem provers in the DPLL framework [19, 21], which makes them refutationally complete for finitely axiomatizable theories. The most recent and promising approaches in this area is [39], which combines a refutational instantiation procedure with model checking based prioritization heuristics. However, as our experiments showed, the straightforward extension of the SMT solver with a relational theory, evaluates in practically non analyzable proof obligations. This makes techniques like our SB^+ indispensable for proving Alloy problems with structure-finding based solvers like the SMT solvers.

Other approaches, like Abadi et al. [3] and Lev-Ami et al. [56], focus on the reasoning on the fragment of pure first order-logic extended with transitive closure. These are discussed in more details in Chapter 8.

Suter, et al. [70] presented a decision procedure for the quantifier-free Boolean Algebra with Presburger Arithmetic (QFBAPA) capable of handling sets and their cardinalities. They reduce QFBAPA to integer linear arithmetic (QFPA) which is solved by the decision procedures of Z3. Set cardinality is computed using the integers that represent the cardinality of Venn regions —the regions built by the maximal overlapping degree of a finite collection of sets. Since Alloy cardinality can be applied to arbitrary expressions (possibly containing variables) with arbitrary arities, this technique is not readily applicable to our translation.

6.6 Conclusion

In this chapter, we presented our SB^+ technique. This technique together with our translation of Alloy proof obligations to RFOL proof obligations (see Chapter 4 and Chapter 5) constitutes the base of AlloyPE —our tool for the automatic verification of Alloy problems. An extension of the tool including its prove power is discussed and presented in Chapter 8. It should be noted that the AlloyPE results presented in this chapter first became possible after the introduction of our SB^+ technique. Among others, the SB^+ technique provides for a given RFOL formula when and how

to remove (after their applications) the axioms of all first-order relational operators while preserving the satisfiability of the formula.

To our knowledge, AlloyPE is the first tool capable of verifying Alloy assertion full automatically, a capability totally missing from the Alloy Analyzer. Although AlloyPE avoids type finitization altogether, it is also capable of finding counterexamples in case the Alloy assertion is not valid. However, we suggest our analysis to be used to complement Alloy Analyzer: when Alloy Analyzer fails to find a counterexample, our tool can be used to try to prove the assertion correct.

Due to Alloy's undecidability and our arbitrary use of quantifiers, resulting SMT formulas can be undecidable. However, among different ways of axiomatizing an Alloy construct, we have carefully chosen the one that performs best in practice. The current results show that Z3 can correctly handle most of our valid and invalid assertions, witnessing the effectiveness of the approach. Improving the cases that AlloyPE failed to handle is done in Chapter 8.

Although we focused on Alloy, our technique demonstrate a general approach that can be applied in various contexts. In particular, we described how to axiomatize transitive closure using the theory of linear integer arithmetic, and cardinality of (possibly cyclic) relations using bijective integer functions.

Our current translation deviates from Alloy semantics in handling arithmetic using infinite integers. While we believe that this is more suitable for most system descriptions, we will also provide in the future an alternative fixed bitwidth arithmetic using bit-vectors.

Chapter 7

Variable Elimination via Sufficient Ground Term Sets Computation

Trying to solve the problem of cardinality constraints on structure universes induced by SB theories (see chapter 6), we encountered the more general problem of quantified variables elimination. That is, given a universally quantified variable x in a formula A, find an equisatisfiable formula A' to A such that A' has less quantified variables including x. It should be noted that this problem is different from the related and well known problem of quantifier elimination (QE) in the following respects: (1) it focuses on the empty theory, but allows some forms of combination with arithmetic theories, (2) it focuses on the elimination of individual variables, (3) it bases on finite instantiation with ground terms, and (4) it targets small instantiation sets that let A' be more efficient in terms of Satisfiability Modulo Theories (SMT) solving.

Point (4) in the above paragraph, reveals implicitly that in some cases the SMT solvers are more efficient in handling formulas with more quantified variables when the instantiation sets are too large. This is due to the significant progress made by modern SMT solvers in handling general first-order formulas. SMT solvers such as CVC4 [8], haRVey [21], Yices [26], and Z3 [20] do not only operate on quantifier-free fragments but successfully address general FOL formulas —including quantified formulas. Most of these solvers solve quantified formulas using heuristic quantifier instantiation based on the E-matching instantiation algorithm —originally introduced by Simplify [22]. Although E-matching, because of its heuristic nature, is not complete, not even refutationally, it is best suited for integration into the DPLL(T) framework. Some techniques (e.g. [67, 39]) have extended E-matching in order to make it complete for some fragments of first-order logic.

In spite of all the advances, the presence of quantifiers still poses a challenge to the solvers. In this chapter, we propose a simplification of quantified SMT formulas that can be applied as a pre-process before the SMT solving. Given a skolemized SMT formula A, our simplification returns an equisatisfiable SMT formula A' with

potentially fewer universally quantified variables. Our simplification approach is syntactic in the sense that it extracts a set of set-valued constraints from the structure of A whose solution is a set of *sufficient ground terms* for every variable. Those variables whose sets of sufficient ground terms are finite can be eliminated by instantiating them with the computed ground terms. If the resulting formula A' is unsatisfiable, A is guaranteed to be unsatisfiable too. However, if A' has a structure, it is not necessarily a structure of A. We describe how any structure of A' can be modified into a structure for A without any significant overhead. This requires a special treatment of the interpreted functions. Our simplification procedure can also be applied if the logic of the input formula is not decidable; it can still reduce the number of quantifiers, thus simplifying the proof obligation.

Although our elimination process reduces the number of quantifiers, it may increase the number of occurrences of the remaining quantified variables (if any). The following example illustrates a case where eliminating one variable can result in increasing the occurrences of the other variables.

Example 1. *Let $\forall x.\,(\psi(x) \vee \forall y,z.\,\varphi(x,y,z))$ be the input formula A and $S_y = \{gt_1,\ldots,gt_n\}$ be a set of sufficient ground terms for the variable y. Suppose that the sets of sufficient ground terms of x and z are infinite. In this case, instantiating and eliminating y will result in the formula A'*

$$\forall x.\,(\psi(x) \vee \forall z.\,(\varphi(x,gt_1,z) \wedge \ldots \wedge \varphi(x,gt_n,z)))$$

which has less quantified variables but a higher number of occurrences of the variables x and z.

Depending on the complexity of the involved terms, this phenomenon may introduce additional overhead for the solver. Therefore, in order to apply our simplification as a general preprocessing step, it is important to balance the number of eliminated variables and the number of newly introduced variable occurrences. We define a metric that aims for estimating the cost of variable elimination, and allows the user to provide a threshold for the estimated cost.

7.1 Example

Figure 7.1(a) shows an SMT formula (as a set of implicitly conjoined subformulas) in which c_1 and c_2 represent constants, f is a unary function, and p is a binary predicate. Figure 7.1(b) shows the same formula after conversion to conjunctive normal form (CNF) (denoted by A^{CNF}) where the constants c_3 and c_4 denote the skolems for the formulas (3) and (4), respectively. Instead of solving the original formula (denoted by A), we produce an *instantiated formula* A^{inst} in which the x and y variables are instantiated with certain ground terms. A^{inst} is given in Figure 7.1(c) where the numbers correspond to the lines in the CNF (and original) formula. Formula A^{inst} has fewer quantifiers than A (in fact, it has zero quantifiers), and thus is easier to solve. Yet, it can be shown that A^{inst} and A are equisatisfiable (see Section 7.2).

(a)	(b)	(c)
(1) $c_1 \neq c_2$ (2) $\forall x.\ f(x) = f(c_1)$ (3) $\exists z.\ \forall y \mid \neg p(y,z) \vee f(y) = c_2$ (4) $\exists z.\ f(z) = c_1$	(1) $c_1 \neq c_2$ (2) $\forall x.\ f(x) = f(c_1)$ (3) $\forall y.\ \neg p(y,c_3) \vee f(y) = c_2$ (4) $f(c_4) = c_1$	(1) $c_1 \neq c_2$ (2) $f(c_1) = f(c_1)$ (2) $f(c_4) = f(c_1)$ (3) $\neg p(c_1,c_3) \vee f(c_1) = c_2$ (3) $\neg p(c_4,c_3) \vee f(c_4) = c_2$ (4) $f(c_4) = c_1$

$$\bar{M} = \{1,2,3,4\}, M(c_1) = 1, M(c_2) = 2, M(c_3) = 3, M(c_4) = 4$$

$$M(f)(v) = \begin{cases} 1 & \text{if } v = 1 \\ 1 & \text{if } v = 4 \\ \text{any value} & \text{else} \end{cases} \qquad M(p)(v,3) = \begin{cases} \mathit{ff} & \text{if } v = 1 \\ \mathit{ff} & \text{if } v = 4 \\ \text{any value} & \text{else} \end{cases}$$

(d)

$$\bar{M}^\pi = \bar{M}, M^\pi(c_1) = M(c_1), M^\pi(c_2) = M(c_2), M^\pi(c_3) = M(c_3), M^\pi(c_4) = M(c_4)$$

$$M^\pi(f)(v) = \begin{cases} M(f)(v) & \text{if } v \in \{1,4\} \\ M(f)(M(c_1)) & \text{else} \end{cases} = 1 \quad \text{for all } v$$

$$M^\pi(p)(v,c_3) = \begin{cases} M(p)(v,M(c_3)) & \text{if } v \in \{1,4\} \\ M(p)(M(c_1),M(c_3)) & \text{else} \end{cases} = \mathit{ff} \quad \text{for all } v$$

(e)

Figure 7.1: Example of safe elimination of quantified variables. (a) original SMT formula, (b) CNF transformation, (c) instantiated formula, (d) a structure for the instantiated formula, and (e) a structure for the original formula.

We use $vGT(x)$ to represent the set of ground terms that is used to instantiate a variable x. The variable x (in Formula 2 of A^{CNF}) refers to the first argument of f, and thus we instantiate it with all the ground terms that occur in that position of f, namely $\{c_1, c_4\}$. We call this the set of ground terms of f for argument position 1, and denote it by $fGT(f,1)$. The variable y (in Formula 3 of A^{CNF}), on the other hand, refers to both the first argument of p and the first argument of f. Therefore, we require that $vGT(y) = fGT(p,1) \cup fGT(f,1)$. In order to guarantee equisatisfiability of A^{inst} and A, if two functions are applied to the same variable, their ground terms set for the application position should be equal (see Section 7.2). Therefore, in this example, $fGT(p,1) = fGT(f,1) = \{c_1, c_4\}$ although p is not directly applied to any constants.

The instantiated formula A^{inst} is an implication of the original formula. Hence, if A^{inst} is unsatisfiable, A is also unsatisfiable. However, not every structure of A^{inst} satisfies A. But the instantiation was chosen in such a way that each structure of A^{inst} can be modified to satisfy A. Figure 7.1(d) gives a sample structure $\mathcal{M}$ for A^{inst} which

does not satisfy A. Since in A^{inst}, f is only applied to c_1 and c_4, and p only to (c_1, c_3) and (c_4, c_3), $\mathcal{M}$ may assign arbitrary values to f and p applied to other arguments. Although these values do not affect satisfiability of A^{inst}, they affect satisfiability of A. Therefore, we modify $\mathcal{M}$ to a structure $\mathcal{M}^\pi$ by defining acceptable values for the function applications that do not occur in A^{inst}. Figure 7.1(e) gives the modified structure $\mathcal{M}^\pi$ that our algorithm constructs. It is easy to show that this structure satisfies A.

The basic idea of modifying a structure is to fix the values of the function applications that do not occur in A^{inst} to some arbitrary value of a function application that does occur in A^{inst}. This works well for this example as f and g are uninterpreted symbols and thus their interpretations are not restricted beyond the input formula. Were they interpreted symbols, this would be different. As an example, assume that p is the interpreted operator "$\leq$". In this case, the original formula $A_\leq$ becomes unsatisfiable[1], but its instantiation $A_\leq^{inst}$ stays satisfiable[2]. To guarantee the equisatisfiability in the presence of interpreted symbols, we require the ground term sets to contain some terms that make their enclosing literals false —henceforth, we call literals containing variables as argument of interpreted functions *interpreted* otherwise *uninterpreted*[3]. This makes the solver explore the cases where clauses become satisfiable regardless of the interpreted literals. In this example, the interpreted literal $\neg(y \leq c_3)$ becomes false if y is instantiated with the ground term $c_3 - 1$. Instantiating $A_\leq$ with the ground terms $\{c_1, c_4, c_3 - 1\}$ reveals the unsatisfiability.

In the following, we formalize the ideas used in the example above. We first define sufficient ground term sets for quantified variables. Then, give a syntactic rule system that derives a set constraints system capable of finding out if the sufficient ground term set of a given variable is finite and compute it otherwise.

7.2 Sufficient Ground Term Sets

Definition 18 (Sufficient ground term sets)**.** Given a variable x in a clause C of an SMT formula A (in CNF), a set of ground terms $S \subseteq \mathcal{H}(A)$ is *sufficient* for x w.r.t. a theory $\mathcal{T}$ if A and $A[S/x]$[4] are equisatisfiable modulo $\mathcal{T}$.

A variable x in a formula A can have more than one sufficient set of ground terms. $\mathcal{H}(A)$, for example, is always a sufficient set of ground terms as a result of the Gödel-Herbrand-Skolem theorem which states that a formula A in Skolem Normal Form (SNF) is satisfiable iff its Herbrand expansion $\{A[gt_1/x_1] \ldots [gt_n/x_n] \mid gt_{1:n} \in \mathcal{H}(A)\}$ is satisfiable where $x_{1:n}$ are the variables of A [69]. But, $\mathcal{H}(A)$ is usually infinite, and our goal is to determine whether a *finite* set of sufficient ground terms exists, and to

[1](2) and (4) imply $f(c_1) = c_1$. $y \leq z$ holds for some pair of integers, thus (3) implies $f(y) = c_2$ for some y. But $f(y) = f(c_1)$ by (2) and so $f(c_1) = c_2 = c_1$. This contradicts, however, (1).

[2]A structure is $M'(c_1) = 1, M'(c_2) = 2, M'(c_3) = 0, M'(c_4) = 4, M'(f) \equiv 1$

[3]A more convenient notation would be *essentially interpreted literals*. However, for the sake of simplicity we use the shorter notation.

[4]In practice, one will use $A[C[S/x]/C]$ instate where C is the enclosing clause of x.

compute it if one exists. This is done, in our approach, by generating and solving a system of set constraints $\mathcal{S}_A$ over sets of ground terms.

Figure 7.2 presents our (syntactic) rules to generate the set constraints system $\mathcal{S}_A$ for a formula A in CNF. The notation $t \dot{\in} C$ denotes that a term t occurs as a subterm of a clause C of A. We use $\mathcal{S}_A$ to denote the set constraints system that results from applying these rules exhaustively to all the clauses of A. The constraints range over the sets $vGT(x) \subseteq Gr$ for all variables x in A. These sets denote the relevant instantiations for the respective variables. Auxiliary sets $fGT(f,i) \subseteq Gr$ are introduced to denote the set of relevant ground terms for an uninterpreted function $f \in \mathcal{F}$ at an argument position $i \in \mathbb{N}$. We assume that the theory of integers is part of the considered $\mathcal{T}$, and that integers are included in the universe of every $\mathcal{T}$-structure $\mathcal{M}$, i.e. $\mathbb{Z} \subseteq \bar{M}$. The integer operators $<, \leq, +, -, \geq, >$ are fixed with their obvious meaning.

Rule R_1 establishes a relationship between sets of ground terms for variables and function arguments. Rule R_2 ensures that the ground terms that occur as arguments of a function f are added to the corresponding ground term set of f. Rule R_3 states that if a term $t[x_{1:n}]$ with variables $x_{1:n}$ occurs as the i-th argument of f, then all the instantiations of t with the respective sets $vGT(x_i)$ must be in $fGT(f,i)$. Rules R_0, R_4 and R_{14} state that our approach does not *currently* handle the case where a variable x occurs as atom of a literal or as an argument of an unsupported interpreted function (supported operators are $\{=, <, \leq, >, \geq\}$, where "$=$" refers to integer equality), thus sets $vGT(x)$ to infinity. Moreover, we do not handle the case where a supported interpreted operator has more than one variable argument (rule R_5). The remaining rules infer additional constraints for $vGT(x)$ where x occurs as an argument of a supported interpreted function. They constrain $vGT(x)$ to contain at least one ground term that falsifies the corresponding (interpreted) literal.

It should be noted that, in fact, we use infinity as a label to denote that our (current) approach cannot compute a finite sufficient ground term set for a variable. This happens in three main cases: (1) if the ground term set of a variable is explicitly set to infinity (using rules R_0, R_4, R_5 and R_{14}), (2) if we can conclude that the ground term set of a variable *subsumes* an infinite ground term set (using rules R_1 and R_3) and (3) if we can conclude that the ground term set of a variable *subsumes* itself. A ground term set S *subsumes* a ground term set R, denoted by $R \dot{\subseteq} S$, if for every ground term $gt_1 \in R$ there exists a ground term $gt_2 \in S$ such that gt_1 is a subterm of gt_2.

Before formulating and proving our main theorem, we first motivate the intuition behind our rules in the example of R_1, R_2 and R_3. Therfore consider the formula A

$$A' \wedge$$
$$\varphi_1[f(gt_1)] \wedge (\forall x.\ \varphi_2[f(x)]) \wedge \tag{7.1}$$
$$(\forall y.\ \varphi_3[f(g(y))] \wedge h(y))) \wedge h(gt_2) \tag{7.2}$$

where φ_1, φ_2 and φ_3 are three formulas containing (among others) the terms $f(gt)$, $f(x)$ and $f(g(y))$, respectively, and f, g, and h are function symbols. Let further assume that A is refutable.

$$\mathrm{R}_0: \frac{x \in C \;||\; \neg x \in C}{vGT(x) = \infty} \qquad \mathrm{R}_1: \frac{f(\cdots, \overbrace{x}^{\text{i-th}}, \cdots) \dot{\in} C}{vGT(x) = fGT(f,i)} \qquad \mathrm{R}_2: \frac{f(\cdots, \overbrace{gt}^{\text{i-th}}, \cdots) \dot{\in} C}{gt \in fGT(f,i)}$$

$$\mathrm{R}_3: \frac{f(\cdots, \overbrace{t[x_{1:n}]}^{\text{i-th}}, \cdots) \dot{\in} C}{t[vGT(x_1)/x_1, \cdots, vGT(x_n)/x_n] \subseteq fGT(f,i)}$$

$$\mathrm{R}_4: \frac{op(\cdots, x, \cdots) \in C,\ op \notin \{=, \neq, <, \leq, >, \geq\}}{vGT(x) = \infty} \qquad \mathrm{R}_5: \frac{op(\ldots, x, \ldots, y, \ldots) \in C}{vGT(x) = \infty,\ vGT(y) = \infty}$$

$$\mathrm{R}_6: \frac{(x \leq gt) \in C}{gt + 1 \in vGT(x)} \qquad \mathrm{R}_7: \frac{(x \geq gt) \in C}{gt - 1 \in vGT(x)} \qquad \mathrm{R}_8: \frac{\neg op(x, gt) \in C, where\ op \in \{\leq, \geq\}}{gt \in vGT(x)}$$

$$\mathrm{R}_9: \frac{\neg(x < gt) \in C}{gt - 1 \in vGT(x)} \qquad \mathrm{R}_{10}: \frac{\neg(x > gt) \in C}{gt + 1 \in vGT(x)} \qquad \mathrm{R}_{11}: \frac{op(x, gt) \in C, where\ op \in \{<, >\}}{gt \in vGT(x)}$$

$$\mathrm{R}_{12}: \frac{\neg(x = gt) \in C}{gt \in vGT(x)} \qquad \mathrm{R}_{13}: \frac{(x = gt) \in C, x \in \mathbb{Z}}{\{gt - 1, gt + 1\} \subseteq vGT(x)} \qquad \mathrm{R}_{14}: \frac{(x = gt) \in C, x \notin \mathbb{Z}}{vGT(x) = \infty}$$

Figure 7.2: The syntactic rules for generating the set constraints system ($\mathcal{S}_A$). C denotes a clause of A, gts denote ground terms, f, and op denote uninterpreted and interpreted function symbols, $t[x_{1:n}]$ denotes a term with variables $x_{1:n}$.

Considering the sub formula 6.1, A restricts, abstractly seen, the interpretation of f on the value of gt_1 with φ_1 and on all values with φ_2. Now having this view on A, one can easily see that if the variable x is eliminated without considering its instantiation with gt_1, one could miss a possible inconsistency of A —namely between φ_1 and φ_2— and thus possibly report a spurious counterexample. This case is exactly prevented by the exhaustive application of the rules R_1 and R_2.

Considering the sub formula 6.2, A further restricts the interpretation of f for all values of the image of g with φ_3. In this case and in order to prevent missing a possible inconsistency between φ_2 and φ_3 one should guarantee that x gets instantiated (at least) with all ground terms of the form $g(gt)$ where $gt \in vGT(y)$ (in our example $g(gt_2)$). This is guaranteed by the exhaustive application of R_1, R_2 and R_3. This argumentation also holds even if g is the identity and this is exactly the reason for the (at first sight not intuitive) left to right inclusion of the equality of rule R_1.

Let $vGT_{\mathcal{S}_A}$ denote a collection of finite sets of ground terms which satisfies the constraints of $\mathcal{S}_A$. We show that, if finite, $vGT(x)_{\mathcal{S}_A}$ is a sufficient ground term set for x in A. The variable x can hence be eliminated by instantiating it with all the ground

terms in $vGT(x)_{\mathcal{S}_A}$. The resulting formula $A[vGT(x)_{\mathcal{S}_A}/x]$ is equisatisfiable to A and does not contain x anymore.

Theorem 7 (Main theorem)**.** *Let x be a variable in A with $vGT(x)_{\mathcal{S}_A} \neq \infty$, then A and $A[vGT(x)_{\mathcal{S}_A}/x]$ are equisatisfiable.*

Proof. If $A[vGT(x)_{\mathcal{S}_A}/x]$ is unsatisfiable, so is A since the former is an implication of the latter. If $A[vGT(x)_{\mathcal{S}_A}/x]$ is satisfiable with a structure $\mathcal{M}$, then we construct a modified structure $\mathcal{M}^{\pi_x}$ (as defined below) and show in lemma 16 that $\mathcal{M}^{\pi_x}$ satisfies A. □

Given a structure $\mathcal{M}$ for the formula $A[vGT(x)_{\mathcal{S}_A}/x]$, we construct a modified structure $\mathcal{M}^{\pi_x}$ as follows: $\bar{M}^{\pi_x} := \bar{M}$. For any constant $c \in Const$, $M^{\pi_x}(c) := M(c)$. For any interpreted operator op, $M^{\pi_x}(op) := M(op)$. For any uninterpreted function f, $M^{\pi_x}(f)(v_{1:n}) := M(f)(\pi_x(f,1)(v_1), \cdots, \pi_x(f,n)(v_n))$, where the *value projection* with respect to the i^{th} argument of f $\pi_x(f,i)$ is defined as in Equation 7.3.

Intuitively, if the ground term set of x does not *subsume* the ground term set of the i^{th} argument of f, or if v_i is a value that M assigns to a ground term for the i^{th} argument of f, then $M^{\pi_x}(f)(..,v_i,..) := M(f)(..,v_i,..)$. Otherwise, $\pi_x(f,i)$ maps v_i to a value that M assigns to some ground term for the i^{th} argument of f. Integers must be mapped to the closest such value (see the proof of Lemma 15).

$$\pi_x(f,i)(v) = \begin{cases} v & \text{if } fGT(f,i)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A} \\ v & \text{else if } v \in M(fGT(f,i)_{\mathcal{S}_A}) \\ v' \in M(fGT(f,i)_{\mathcal{S}_A}) & \text{else if } v \notin \mathbb{Z} \\ v' \in M(fGT(f,i)_{\mathcal{S}_A}), s.t.\ |v - v'| \text{ is minimal} & \text{otherwise} \end{cases} \tag{7.3}$$

$$\pi_x(v) = \begin{cases} v & \text{if } v \in M(vGT(x)_{\mathcal{S}_A}) \\ v' \in M(vGT(x)_{\mathcal{S}_A}) & \text{else if } v \notin \mathbb{Z} \\ v' \in M(vGT(x)_{\mathcal{S}_A}), s.t.\ |v - v'| \text{ is minimal} & \text{otherwise} \end{cases} \tag{7.4}$$

We also define the value projection with respect to a variable x π_x as in Equation 7.4. If $vGT(x)_{\mathcal{S}_A} = fGT(f,i)_{\mathcal{S}_A}$, for instance because x occurs as the i^{th} argument of f, then $\pi_x = \pi_x(f,i)$.

Before showing the proof of lemma 16 used in our main theorem, we introduce and prove some auxiliary corollaries and lemmas.

Corollary 4. *If $vGT(x)_{\mathcal{S}_A} \neq \infty$, then $\pi_x(v) \in M(vGT(x)_{\mathcal{S}_A})$, for all $v \in \bar{M}$.*

Proof. The claim follows directly from the definition of π_x □

Corollary 5. *For all ground terms t and variables x occurring in A, $val_{\mathcal{M}^{\pi_x},\beta}(t) = val_{M,\beta}(t)$ for any variable assignment β.*

Proof. We perform the proof by induction over the structure of t.

If $t \in Const$, the claim follows directly from the definition of M^{π_x}.

If, without loss of generality, $t := f(s)$, where $f \in \mathcal{F}$ and $s \in Gr(A)$, we get by the induction hypothesis,

$$\begin{aligned} & val_{\mathcal{M}^{\pi_x},\beta}(f(s)) \\ \overset{val}{=} \; & M^{\pi_x}(f)(val_{\mathcal{M}^{\pi_x},\beta}(s)) \\ \overset{IH}{=} \; & M^{\pi_x}(f)(val_{\mathcal{M},\beta}(s)) \\ \overset{val}{=} \; & M^{\pi_x}(f)(M(s)) \end{aligned}$$

Now, we have to distinguish between interpreted and uninterpreted functions. If f is interpreted, the claim follows directly from the definition of M^{π_x}. If f is uninterpreted, we first get $M^{\pi_x}(f)(M(s)) = M(f)(\pi_x(f,1)(M(s)))$. Furthermore, we know, because of rule R_2 and since $f(s)$ occurs in A, that $s \in fGT(f,1)_{\mathcal{S}_A}$ and consequently $M(s) \in M(fGT(f,1)_{\mathcal{S}_A})$. Now, we can conclude the proof using the definition of $\pi_x(f,1)$ for values in $M(fGT(f,1)_{\mathcal{S}_A})$. □

The following lemmas show that if a structure $\mathcal{M}$ together with the variable assignment $\beta' = \lambda y.$ *if* $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$ *then* $\pi_y(\beta(y))$ *else* $\beta(y)$ for some variable assignment β satisfies a literal l in a CNF formula A, then the modified structure $\mathcal{M}^{\pi_x}$ together with β satisfies l. These lemmas are insofar important as they describe how our structure modification M^{π_x} can be reduced to a modification of variable assignments. Lemma 14 gives a stronger variant (with value equality rather than implication) for uninterpreted literals, and Lemma 15 formulates the claim for interpreted literals.

Lemma 14. *Let x be a variable with $vGT(x)_{\mathcal{S}_A} \neq \infty$, $\mathcal{M}$ a structure, β a variable assignment, and $\beta' = \lambda y.$ if $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$ then $\pi_y(\beta(y))$ else $\beta(y)$. Then $val_{\mathcal{M},\beta'}(l) = val_{M^{\pi_x},\beta}(l)$ for all uninterpreted literals l in A.*

Proof. To prove the claim, we show the statement $val_{\mathcal{M},\beta'}(l) = val_{M^{\pi_x},\beta}(l)$ for all uninterpreted terms $l \in Term_\Sigma$ occurring as literals of the CNF of A using structural induction.

If l is a ground term in A, then the claim follows directly from corollary 5.

If $l = y$ is a variable, then because of rule R_0 $vGT(y)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A}$. Consequently, $val_{\mathcal{M},\beta'}(l) = \beta'(y) = \beta(y) = val_{M^{\pi_x},\beta}(l)$.

Let $l = f(t_{1:n})$ be a function application in A with f an uninterpreted function. The evaluations of l are

$$\begin{aligned} val_{\mathcal{M}^{\pi_x},\beta}(f(t_{1:n})) &= M^{\pi_x}(f)(val_{\mathcal{M}^{\pi_x},\beta}(t_1),\ldots,val_{\mathcal{M}^{\pi_x},\beta}(t_n)) \\ &= M(f)(\pi_x(f,1)(val_{\mathcal{M}^{\pi_x},\beta}(t_1)),\ldots,\pi_x(f,n)(val_{\mathcal{M}^{\pi_x},\beta}(t_n))) \\ val_{\mathcal{M},\beta'}(f(t_{1:n}) &= M(f)(val_{\mathcal{M},\beta'}(t_1),\ldots,val_{\mathcal{M},\beta'}(t_n)) \end{aligned}$$

It suffices to show that $\pi_x(f,i)(val_{\mathcal{M}^{\pi_x},\beta}(t_i)) = val_{M,\beta'}(t_i)$ for $1 \leq i \leq n$. We do this by a case distinction over the structure of the terms t_i.

If $t_i = y$ is a variable with $vGT(y)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A}$, then $\beta'(y) = \beta(y)$. Because of rule R_1 we additionally get $fGT(f,i)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A}$, which implies that $\pi_x(f,i)$ is the identity.

If $t_i = y$ is a variable with $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$, then $\beta'(y) = \pi_y(\beta(y))$. Because of rule R_1 we get $fGT(f,i)_{\mathcal{S}_A} = vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$, which implies that $\pi_x(f,i) = \pi_y$.

If t_i is a function application, we assume $t_i = s[x_{1:m}]$ for some term s. By induction hypothesis,

$$\pi_x(f,i)(val_{\mathcal{M}^{\pi_x},\beta}(s[x_{1:m}])) \overset{IH}{=} \pi_x(f,i)(val_{\mathcal{M},\beta'}(s[x_{1:m}])).$$

It suffices now to show that

$$\pi_x(f,i)(val_{\mathcal{M},\beta'}(s[x_{1:m}])) = val_{M,\beta'}(s[x_{1:m}]).$$

With respect to $fGT(f,i)_{\mathcal{S}_A}$, there is two possible cases to consider.

1. $fGT(f,i)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A}$: in this case, $\pi_x(f,i)$ is the identity and the claim follows directly.

2. $fGT(f,i)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$: in this case, $vGT(x_i)_{\mathcal{S}_A} \dot{\subseteq} fGT(f,i)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$, for all $1 \leq i \leq m$ (because of rule R_3). This implies that $\beta'(x_i) = \pi_{x_i}(\beta(x_i))$ for all $1 \leq i \leq m$. Using this fact together with corollary 4, there exists for each x_i a ground term gt_i, with $\pi_{x_i}(\beta(x_i)) = M(gt_i)$ and $gt_i \in vGT(x_i)_{\mathcal{S}_A}$. Therefore, we can write, w.l.o.g., $\pi_x(f,i)(val_{\mathcal{M},\beta'}(s[x_{1:m}])) = \pi_x(f,i)(M(s[gt_{1:m}]))$. Because of rule R_3 we know that $s[gt_{1:m}] \in fGT(f,i)_{\mathcal{S}_A}$, and so $M(s[gt_{1:m}]) \in M(fGT(f,i)_{\mathcal{S}_A})$. Finally, the claim follows from the definition of $\pi_x(f,i)$ for values in $M(fGT(f,i)_{\mathcal{S}_A})$ and the assumption that $fGT(f,i)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$.

Let $l := f(t_{1:n})$ be a function application with f an uninterpreted function. Since l is uninterpreted, all t_is are non-variables and we can use the induction hypotheses on them. Having this together with the definition of M^{π_x} for interpreted functions, we can now conclude the proof for this last case as follow:

$$\begin{aligned}
& val_{\mathcal{M}^{\pi_x},\beta}(f(t_{1:n})) \\
\overset{val}{=}\ & M^{\pi_x}(f)(val_{\mathcal{M}^{\pi_x},\beta}(t_1),\ldots,val_{\mathcal{M}^{\pi_x},\beta}(t_n)) \\
\overset{M^{\pi_x}}{=}\ & M(f)(val_{\mathcal{M}^{\pi_x},\beta}(t_1),\ldots,val_{\mathcal{M}^{\pi_x},\beta}(t_n)) \\
\overset{IH}{=}\ & M(f)(val_{\mathcal{M},\beta'}(t_1),\ldots,val_{\mathcal{M},\beta'}(t_n)) \\
\overset{val}{=}\ & val_{\mathcal{M},\beta'}(f(t_{1:n}))
\end{aligned}$$

□

Lemma 15. *Let x be a variable with $vGT(x)_{\mathcal{S}_A} \neq \infty$, $\mathcal{M}$ a structure, β a variable assignment, and $\beta' = \lambda y.$ if $vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$ then $\pi_y(\beta(y))$ else $\beta(y)$. Then $(M, \beta') \models l$ implies $(M^{\pi_x}, \beta) \models l$ for all interpreted literals l in A.*

Proof. We prove the claim by induction on the structure of uninterpreted literals l. The structure of uninterpreted literals, however, has only one case more than the structure of uninterpreted literals, namely the case of $l = op(y_{1:n})$ where op is an interpreted function symbol and $y_{1:n}$ are variables. Having already proved Lemma 14, we can focus, w.l.o.g., on this case.

Fist, we considre the case in which $vGT(y_i)_{\mathcal{S}_A} \dot{\not\subseteq} vGT(x)_{\mathcal{S}_A}$ for all $1 \leq i \leq n$. In this case, $\beta'(y_i) = \beta$ for all $1 \leq i \leq n$. Having this, the claim follows directly using the defintion of M^{π_x} for interpreted function symbols,

$$\begin{aligned} & val_{\mathcal{M}^{\pi_x},\beta}(op(y_{1:n})) \\ \overset{val}{=} \; & M^{\pi_x}(op)(\beta(y_1),\ldots,\beta(y_n)) \\ \overset{M^{\pi_x}}{=} \; & M(op)(\beta(y_1),\ldots,\beta(y_n)) \\ \overset{\beta'}{=} \; & M(op)(\beta'(y_1),\ldots,\beta'(y_n)) \\ \overset{val}{=} \; & val_{\mathcal{M},\beta'}(op(y_{1:n})). \end{aligned}$$

For the remaining case, we can assume $vGT(y_i)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A}$ for some $1 \leq i \leq n$ and consequently $\beta'(y_i) = \pi_{y_i}(\beta(y_i))$. In addition to that, we can because of rules R_4, R_5 and R_{14} further restrict, without loss of generality, the form of l to $l := op(y_i, gt)$ where $op \in \{=, \neq, <, \leq, >, \geq\}$ and $\alpha_\Sigma(y_i) = \mathbb{Z}$. Let us now assume that $(\mathcal{M}, \beta') \models l$ and $(\mathcal{M}^{\pi_x}, \beta) \not\models l$ and show by case distinction on $op \in \{=, \neq, <, \leq, >, \geq\}$ that this assumption is wrong and consequently the claim holds.

1. For $op \in \{<, >\}$, we get from rule R_{11}, $gt \in vGT(y_i)_{\mathcal{S}_A}$ and from the assumption the inequality system $(\pi_{y_i}(\beta(y_i)) < gt) \wedge (\beta(y_i) \geq gt)$, which implies that $|\beta(y_i) - \pi_{y_i}(\beta(y_i))|$ is not minimal, since $|\beta(y_i) - gt|$ is strictly smaller.

2. For $op \in \{\leq, \geq\}$, the proofs go similar to the previous case using rule R_6 for the "$\leq$"-case and rule R_7 for the "$\geq$"-case.

3. For $op :=$ "$=$", we get from rule R_{13}, $\{gt - 1, gt + 1\} \subseteq vGT(y_i)_{\mathcal{S}_A}$ and from the assumption, the inequality system $(\pi_{y_i}(\beta(y_i)) = gt) \wedge (\beta(y_i) \neq gt)$, which is equivalent to $(\pi_{y_i}(\beta(y_i)) = gt) \wedge ((\beta(y_i) \leq gt - 1) \vee (gt + 1 \leq \beta(y_i)))$ and implies that $|\beta(y_i) - \pi_{y_i}(\beta(y_i))|$ is not minimal, since in the case $(\beta(y_i) \leq gt - 1)$, $|\beta(y_i) - (gt - 1)|$ is strictly smaller and in the case $(gt + 1 \leq \beta(y_i))$, $|\beta(y_i) - (gt + 1)|$ is strictly smaller.

4. For $op :=$ "$\neq$", the proof goes similar to the previous case using rule R_{12}.

□

Lemma 16. *Let x be a variable in A with $vGT(x)_{\mathcal{S}_A} \neq \infty$ and $\mathcal{M}$ a structure of $A[vGT(x)_{\mathcal{S}_A}/x]$, then $\mathcal{M}^{\pi_x}$ is a structure of A.*

Proof. Let A' denote $A[vGT(x)_{\mathcal{S}_A}/x]$. Since $\mathcal{M}$ is a (satisfiable) structure of A', $(\mathcal{M},\beta) \models A'$ for every variable assignment $\beta : \mathcal{X} \to \bar{M}$. Let β_0 be an arbitrary variable assignment. By corollary 4, we know that it exists some ground term $gt \in vGT(x)_{\mathcal{S}_A}$ such that $\pi_x(\beta_0(x)) = M(gt)$. The resulting formula of instantiating x with gt $A[gt/x]$ is included in A' and thus $(\mathcal{M},\beta) \models A[gt/x]$ for any β.

Let us now consider the variable assignment modification of β_0

$$\beta_0' = \lambda y.\ \textit{if } vGT(y)_{\mathcal{S}_A} \dot{\subseteq} vGT(x)_{\mathcal{S}_A} \textit{ then } \pi_y(\beta_0(y)) \textit{ else } \beta_0(y).$$

For β_0' holds: (1) β_0' maps x to $\pi_x(\beta_0(x)) = M(gt)$ and (2) $(M,\beta_0') \models A[gt/x]$. From (1) and (2), we get $(\mathcal{M},\beta_0') \models A$.

Assuming that A is in CNF, there must be for every clause C in A at least one literal l^C in C with $(\mathcal{M},\beta_0') \models l^C$. Using lemma 15 for interpreted literals and lemma 14 for uninterpreted literals, we know that also $(\mathcal{M}^{\pi_x},\beta_0) \models l^C$. Hence, M^{π_x} is a structure for l^C, C and finally for A. Since β' was chosen arbitrary, the proof is done. □

7.3 Practical Optimizations

In the previous section we have introduced and proved a general approach that can reduce the number of universally quantified variables of a given SMT formula. However, in practice, our approach may introduce a notifiable overhead that can prevent from attaining the expected increase in SMT solver time. The sources of this overhead are: (1) the cost of converting the SMT formula to CNF and (2) the increase of occurrences of some quantified variables, during the elimination of others. In the following subsections, we present our approaches to reduce the practical effect of this problems.

7.3.1 Simulating NNF

We have formulated and proved our quantified variables elimination approach totally based on the CNF representation of SMT formulas. This includes in addition to the proofs (1) the computation of sufficient ground terms *vGT* and (2) the elimination the eliminable variables via instantiation.

Both tasks, however, do not require necessarily the formula to be in CNF. For the first task of constructing and computing the constraint system of Figure 7.2, only the CNF polarity of the literals of the input formula is needed (see rules R_6 to R_{13}). For the second task of instantiating a variable x with a set S of ground terms in a formula A —not necessarily in CNF, one only needs to adjust the instantiation definition to

$$A[S/x] := A[\bigwedge_{gt \in S} B_x[gt/x]/B_x],$$

where B_x is the smallest subformula containing x.

Therefore, instead of actually converting the original formula to CNF, we (1) *simulate* the NNF (negation normal form) conversion (without actually changing the formula) to compute polarity, and (2) skolemize all existential quantifiers. This computation does not introduce any considerable overhead. It should be noted that conversion to CNF using distribution (as opposed to Tseitin encoding [73, 64]) has the additional advantage that it minimizes the scope of each variable. This can significantly improve our simplification approach. Distribution, however, is very costly in practice. Computing minimal variable scopes without performing distribution is left for future work.

7.3.2 Limiting Instantiations

Our variable elimination approach eliminates those variables that have finite sets of sufficient ground terms by instantiating them with the computed ground terms. In practice, such instantiation may increase the occurrences of non-eliminable variables (see Example 1 of the previous section). Our experiments with Z3 and CVC4 show that this increase in the number of variable occurrences can considerably increase the solving time, specially for nested quantifiers.

In order to estimate the elimination cost $C(S)$ of a set of eliminable variables S, we use definition 19, where *occurr* returns the occurrence positions of a variable in a term. The definition estimates the elimination cost based on the maximal number of newly introduced variable occurrences.

Definition 19 (Occurrence increase of quantified variables)**.** Let A be a formula and $S := \{x_1, \ldots, x_n\} \subseteq \{x \doteq A \mid vGT(x)_{\mathcal{S}_A} \neq \infty\}$ a set of eliminable variables. The occurrence increase of quantified variables caused by eliminating all x_is in A is defined as:

$$C(S) := \max_{y \doteq A} \Big(\frac{|occurr(y, A[vGT(x_1)/x_1, \ldots, vGT(x_n)/x_n])|}{|occurr(y, A)|} \Big)$$

Using Algorithm 1, we estimate and limit the cost of variable elimination based on the number of variable occurrences that it introduces. The algorithm tries to maximize the number of eliminated variables S while keeping the cost C_S low. Given a formula A and a threshold cost C_{max}, this algorithm returns a set of variables S whose elimination cost do not exceed C_{max}. Line 2 initializes the set of expensive —w.r.t. C_{max}— variables *ExpVar* to the set of all variables whose sets of sufficient ground terms are infinite, and thus will *never* be eliminated by our approach. Lines 5-10 evaluate the cost of eliminating all variables y in S. The set S of candidate variables for elimination contains eliminable that are not yet marked expensive. Eliminating all variables y of S with their sufficient ground terms, in the worst case, replicates each expensive variable x in $ExpVar$ $\prod_{y \in bindVars(x) \setminus ExpVar} |vGT(y)_{\mathcal{S}_A}|$ times, where $bindVars(x)$ denotes the elimination candidate variables that scopes x. A variable y scopes a variable x if x occurs in the quantified formula that binds y. If the estimated

Algorithm 1: Heuristic detection of expensive variables with respect to a threshold

Data: $A : Term_{\Sigma}, C_{max} : \mathbb{N}$
Result: S : *Set<Var>*

begin
  $ExpVar \leftarrow \{x \in vars(A) \mid vGT(x)_{\mathcal{S}_A} = \infty\}$
  $S \leftarrow vars(A) \setminus ExpVar$
  repeat
    for $x \in ExpVar$ **do**
      $cost_S \leftarrow \prod_{y \in bindVars(x) \setminus ExpVar} |vGT(y)_{\mathcal{S}_A}|$
      if $cost_S > C_{max}$ **then**
        select $m \in bindVars(x) \setminus ExpVar$ *s.t.* $|vGT(m)_{\mathcal{S}_A}|$ *is maximum*
        $ExpVar \leftarrow ExpVar \cup \{m\}$
        $S \leftarrow vars(A) \setminus ExpVar$
  until *ExpVar is unchanged*;
  return S

cost of eliminating all elimination candidate variables C_S exceeds the given threshold, then a variable m with the maximum number of instantiations ($vGT(m)_{\mathcal{S}_A}$) will be marked as expensive —by adding it to *ExpVar* and consequently moving it from S. The process then starts over [5].

7.4 Evaluation

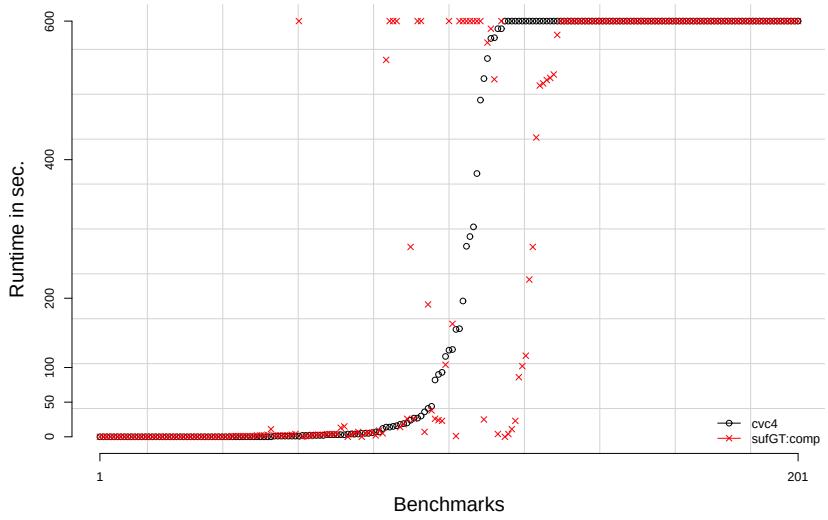
We have implemented our approach in a prototype tool and performed experiments on the SMT competition (SMT-COMP) benchmarks of 2012 in the AUFLIA-p division, using CVC4 (version 1.0) and Z3 (version 4.1) solvers. We ran both solvers on all benchmarks on an AMD DualCore Opteron Quad, 2.6GHz with 32GB memory.

For each benchmark, we compare the original runtime of each solver (with no simplification) against (1) a complete variable elimination and (2) a limited variable elimination where $C_{max} = 100$. Figures 7.3a and 7.3b give the comparison results for CVC4, and Figures 7.4a and 7.4b give the results for Z3. The x-axis of each plot shows the benchmarks, sorted according to the original runtime of the solvers, and the y-axis gives the runtime in seconds. Time-outs and "unknown" outputs are represented identically. The time-out limit is 600 seconds.

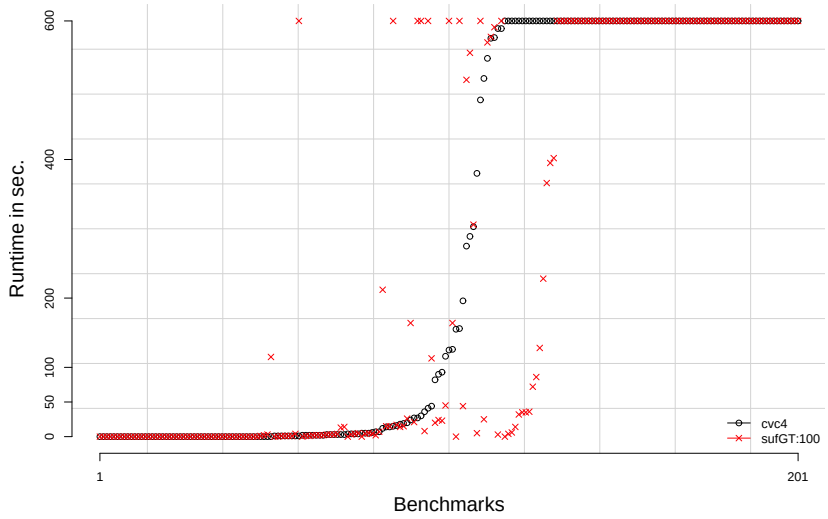
For CVC4, the complete variable elimination improves the solving time of 37 cases (18%) —average speedup[6] 49x— out of which 16 were originally unsolvable, and

[5] Our actual implementation optimizes this algorithm by exploiting the structure of the abstract syntax tree.

[6] Speedup = old solving time / new solving time, where 0 second is changed to 0.5 second.

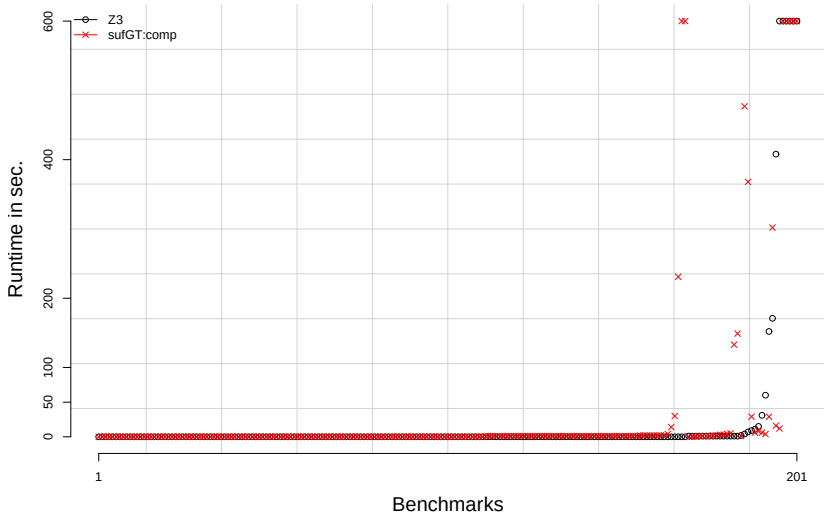


(a) CVC4, original vs. simplified (complete)

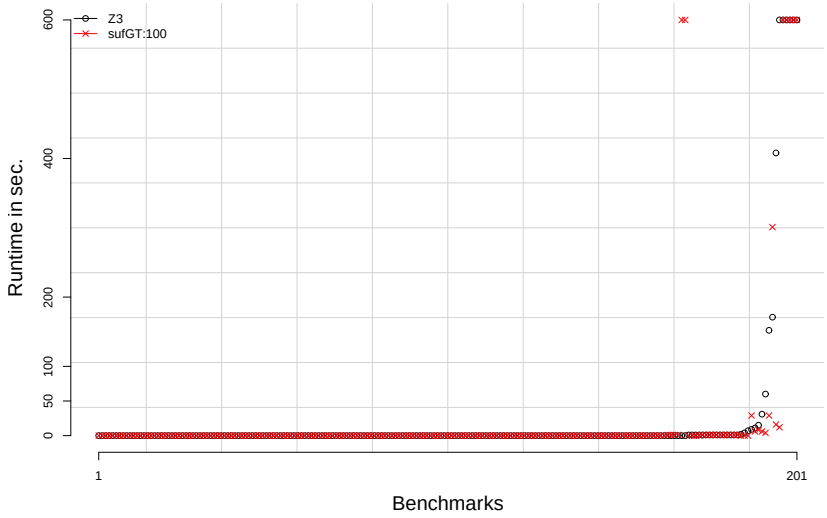


(b) CVC4, original vs. simplified ($C_{max} = 100$)

Figure 7.3: CVC4 experimental results on the benchmarks of SMT-COMP/AUFLIA-p



(a) Z3, original vs. simplified (complete)



(b) Z3, original vs. simplified ($C_{max} = 100$)

Figure 7.4: Z3 Experimental results on the benchmarks of SMT-COMP/AUFLIA-p

worsens 55 cases (27%) —average speedup 0.45. The limited variable elimination, on the other hand, improves 39 cases (19%) —average speedup 57x— out of which 15 were originally unsolvable, and worsens 32 cases (15%) —average speedup 0.48. Z3 is known to be highly efficient in the AUFILA divisions (winner since 2008); its original runtime on many benchmarks is zero. The complete variable elimination, however, worsens 70 of these benchmarks (34%) —average speedup 0.38— and improves 11 cases (5%) —average speedup 10x— out of which one was originally unsolvable. The limited variable elimination, on the other hand, worsens only 8 cases (4%) —average speedup 0.35— and improves 14 cases (7%) —average speedup 9.4x— out of which one was originally unsolvable.

The main reason for slow down is the introduction of too many variable occurrences when not all variables are eliminable. Thus, as shown by these plots, for both solvers, the limited variable elimination produces stronger results. However, even when *all* variables are eliminated, it is still possible that the solving time worsens as the number of instantiations that we produce can be higher than the number of instantiations that the solver would generate while solving the quantified formula. Although feasible in theory, this case was never observed in our experiments.

Although variable elimination with a limited cost can result in significant improvements of solving time, the experiments show that in some cases such as the two new time-outs of Figure 7.4b, a finer-grained limitation decision is needed. Investigating such cases is left as future work.

The individual experiments results of the plots as well as the results of further experiments are available online [29].

7.5 Related Work

Quantifier elimination in its traditional sense (aka. QE) refers to the property that a Σ-theory $\mathcal{T}$ admits QE if for each formula $\phi \in Term_\Sigma$, there exists —algorithmically— a quantifier-free formula $\phi' \in Term_\Sigma$ so that for all $\mathcal{T}$-structures $\mathcal{M}$, $\mathcal{M} \models \phi \Leftrightarrow \phi'$. Most applications of QE either provide decision procedures for fragments of FOL, or only prove their decidability. An example of the later use, is the original decidability proof of the Presburger arithmetic theory which is totally based on the QE technique (see [35, page 197]). An example for the former use, is the Fourier-Motzkin QE procedure for linear rational arithmetic (see [65]).

Given a formula ϕ, the main idea, of almost all QE procedures, is to exploit the arithmetic properties of the theory $\mathcal{T}$ in order to partition the theory sort —e.g., $\mathbb{R}$ for the theory of reals— into a finite number of intervals with periodic behavior and use the intervals boundary as sufficient ground terms. Having this view on QE, one can see our technique as a theory independent QE procedure that also woks if not all variables are eliminable.

Another approach to eliminate quantifiers was proposed in [41] where partial FOL structures are represented as programs. A program generation technique tries to heuristically generate a program P_i for a quantified formula ϕ_i in $F := \phi_1 \wedge \ldots \wedge \phi_n$

such that the proof obligation $[P_i](\phi_1, \ldots, \phi_n \Rightarrow \phi_i)$ can be discharged using a theorem prover. If such a program is found, F is modified to $\phi'_1 \wedge \ldots \wedge \phi'_n$ (without ϕ_i) where $\phi'_j \equiv [P_i]\phi_j$. The program generation and verification loop can be repeated until all quantified formulas are eliminated. Such an approach is very different from ours and is sound only for satisfiable formulas.

Our work was motivated by [18] and [39] in which quantifiers are eliminated via instantiation. In [18], a decision procedure is proposed for the *Array Property* fragment of FOL which supports a combination of Presburger arithmetic for index terms, and equality with uninterpreted functions and sorts (EUF) for array terms. Similar to ours, this work instantiates universally quantified variables with a finite set of ground terms to generate an equisatisfiable formula. They prove the existence of such sets for their target fragment. Our approach, however, targets general FOL formulas and leaves a variable uninstantiated if its set of ground terms is infinite. We believe that we can successfully handle the Array Property fragment. Experiments are left for future work.

In [39], the Model-based Quantifier Instantiation (MBQI) is proposed for Z3. Similar to ours, this work constructs a system of set constraints Δ_F to compute sets of ground terms for instantiating quantified variables. Unlike us, however, they do not calculate a solution upfront, but instead, propose a fair enumeration of the (least) solution of Δ_F with certain properties. Assuming such enumeration, one can incrementally construct and check the quantifier-free formulas as needed[7]. If Δ_F is *stratified*, F is in a decidable fragment, and termination of the procedure is guaranteed. Otherwise the procedure can fall back on the quantifier engine of Z3 and provide helpful instantiation ground terms. Consequently, this technique can only act as an internal engine of an SMT solver and cannot provide a stand-alone formula simplification as ours does.

Variable expansion has also been proposed for quantified boolean formulas (QBF). In [14], a reduction of QBF to propositional conjunctive normal form (CNF) is presented where universally quantified variables are eliminated via expansion. Similar to our approach, they introduce cost functions, but with the goal of keeping the size of the generated CNF small.

7.6 Conclusion

In this chapter, we described a general simplification approach for quantified SMT formulas. Based on an analysis of the ground term occurrences at function applications, we compute *sufficient ground term sets* for each universally quantified variable in the input formula. We proved that instantiating (thus eliminating) any variable whose computed set is finite, results in an equisatisfiable formula. Elimination of each variable is independent of the others. Thus we improve the performance of our technique by restricting the set of eliminable variables: we defined a prioritization

[7] In practice, they guide the quantifier instantiation using model checking which, in turn, uses an SMT solver.

algorithm that estimates the overhead of variable elimination, and tries to maximize the number of eliminable variables while keeping the estimated elimination cost below a threshold. We evaluated our approach using two configurations and two solvers on a large subset of the SMT-COMP benchmarks. Our results show that (1) SMT benchmarks contain many variables that can be eliminated by our technique, (2) our complete variable instantiation may introduce significant overhead and thus slow down the solvers, (3) instantiation along with prioritization shows improvement of the solving time and score.

We believe that our technique can provide an easy framework for extending arbitrary SMT solvers with quantifier support. If we ignore termination and performance related rules when generating the set constraint system, we will have an incremental and fair procedure for building ground term sets. Using a finite model checker, like in [39], can then provide a framework for extending SMT solvers with quantifier support. Investigating this idea is left for future work.

Chapter 8

Transitive Closure Axiomatization via Invariant Injections

Among all non first-order operators of Alloy (respectively RFOL), the transitive closure (TC) is the most important operator. This is because (1) it is the most frequently used operator in existing Alloy problems, and (2) all challenging Alloy problems, whether according to our SMT solving (see Chapter 6) or to interactive theorem proving (see [75]), involve transitive closure. Both observations are, actually, not surprising; Avron states in his investigation of transitive closure logics [7] that the extension of first-order logic with transitive closure is the *right* intermediate level between (pure) first-order and second-order logic for the formalization and mechanization of mathematics; Immerman et al. show that adding transitive closure even to very tame logics makes them undecidable [46].

In Chapter 6 we demonstrated the capability of our axiomatization of RFOL operators, including the integer based axiomatization of transitive closure, together with the SB^+ technique in proving an important number of valid Alloy assertions —including those that involve transitive closure. However, and although the axiomatization is TC-complete —any structure of the axiomatization is a TC-structure, the analysis of the non proved assertions reveals a fundamental restriction of our axiomatization of the non first-order RFOL operators. This restriction goes back to the general limitation of SMT solvers in handling quantified formulas, especially, those axiomatizing recursive definitions, as in the case of transitive closure and set cardinality.

In order to explain this limitation of SMT solvers, we consider an abstract, but for most SMT solvers correct, representation of the internal engine of SMT solvers with quantifier support. Figure 8.1 shows the principle building blocks of this representation. Each formula F —a set of implicitly conjuncted formulas (usually clauses)— is divided into a set G of quantifier free formulas and a possibly non empty set H of quantified formulas —also called axioms. The *ground solver*, constituted of the *core solver* and the *theory solvers*, tries to find a satisfiable structure of the set G modulo the

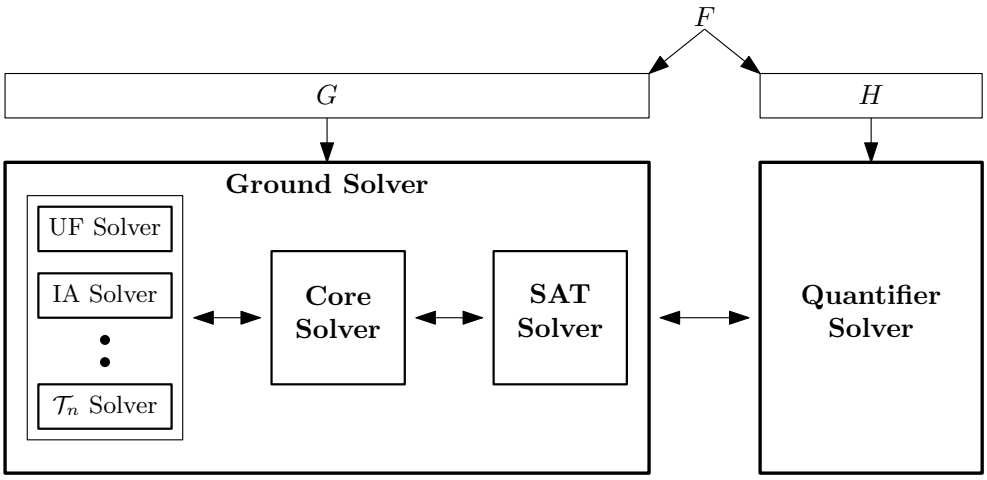


Figure 8.1: An abstract architecture of SMT solvers with quantifier support

theories $\mathcal{T}_{1:n}$ using a SAT solver. If the ground solver can deduce the unsatisfiability of G modulo $\mathcal{T}_{1:n}$ then it reports the refutation of F —and consequently the validity of its negation, otherwise it checks if the found (candidate) structure $\mathcal{M}$ also satisfies the axioms in H using the *quantifier solver*. If the quantifier solver can establish the validity of the structure candidate $\mathcal{M}$ for H, then a satisfiable structure is found for F —and consequently a counterexample for its negation, otherwise the quantifier solver determines for each axiom Ax in H the *ground instantiations* —i.e., instantiations that instantiate all free variables– that contradict the candidate structure $\mathcal{M}$ and adds them to the set G. The process then starts over.

Let us consider the two RFOL formulas $F = G \wedge H$ and $F' = G' \wedge H$ where

$$G = (a,b) \in R \wedge (b,c) \in R \wedge (a,c) \notin R^+$$

and

$$G' = P(a) \wedge \neg P(b) \wedge (a,b) \in R^+,$$

a and b are constants and H the following set of axioms ($Ax_1, \ldots, Ax_4$):

$$\forall x,y.\ (x,y) \in R^+ \iff \exists i.\ 0 \leq i \wedge (x,y) \in R^{(i)} \tag{8.1}$$

$$\forall x,y,i.\ (x,y) \in R^0 \iff (x,y) \in R \tag{8.2}$$

$$\forall x,y,i.\ 0 < i \rightarrow \big((x,y) \in R^{(i)} \iff \exists z : Atom, (x,z) \in R^{(i-1)} \wedge (z,y) \in R\big) \tag{8.3}$$

$$\forall x,y.\ P(x) \wedge (x,y) \in R \rightarrow P(y) \tag{8.4}$$

Both formulas F and F' are refutable (modulo integer theory), however, current SMT solvers can only show the refutation of F but not of F'. In the case of F, and

with respect to the above introduced abstraction of the SMT solver engine, the SMT solver will first find a satisfying structure $\mathcal{M}$ for G, which for example interprets R^+ to be the empty set. This structure is obviously not a structure for the axioms Ax_1 to Ax_3 (Equations 8.1-8.3), namely for $x = a$, $x = b$, $i = 1$ for Ax_1 and Ax_3, and for $x = a$, $x = b$ for Ax_2. However, after adding this contradicting ground instantiations of the three axioms, i.e., $Ax_1[a/x, b/y, 1/i], Ax_2[a/x, b/y], Ax_3[a/x, b/y, 1/i]$, to G, the SMT solver can show the refutation of the new G and consequently of F. For F', the situation is different since there is no finite number of instantiations of the axioms in H such that G become refutable. The reason, therefore, is that the solver can always construct a satisfying structure for G by choosing a *path* $c_1 \overset{R}{\rightarrow} \ldots \overset{R}{\rightarrow} c_n$ in the graph of R to satisfy the literal $(a, b) \in R^+$ (with respect to Ax_1-Ax_3) such that at least the instantiation $Ax_4[c_i/x, c_{i+1}/y]$ of Ax_4 is in not (yet) in G for some $1 \leq i < n$.

In order to show the refutation of formulas like F', for which there exists no finite number of axioms instantiations that can be added to G' to show its refutation, the principle of induction has been successfully used for a long time in mathematics as well as in induction based first-oder theorem provers like ACL2 [49] and is, in fact, the method of choice for this kind of problems. For our example F', it would suffice to proof by induction the axiom $Ax_5 = \forall y.\ (a, y) \in R^+ \rightarrow P(y)$ which states that the formula P is an *invariant* for any R-path that starts with a —i.e., it holds for all reachable atoms from a in the R-graph. Adding this new intermediate axiom to H, will allow the SMT solver to close the proof by only one axiom instantiation, namely $Ax_5[b/y]$. We call literals of the form $(a, b) \in R^+$ R-paths since they state the existence of a path in the graph interpretation of R that starts with a and ends with b.

In this chapter, we present our approach of using the induction principle to increase the proof capability of our tool for formulas in the *transitive closure fragment* —i.e., first-order logic with all first-order relational operators and transitive closure. Therefore, we investigate the following questions:

1. when can the integer based axiomatization of the transitive closure refute formulas without requiring the integer induction principle?

2. for the logic fragment of question 1, can one use an integer-free axiomatization? and

3. to refute a formula outside the fragment of question 1, what kind of integer-free axiomatization can be used?

Let F be a refutable first-order relational formula in which the semantics of all symbols except for transitive closure are precisely encoded, and the transitive closure of a relation R is encoded by an uninterpreted binary relation tc_R. To answer questions (1) and (2), we use a pure first-order, weak axiomatization (WTC) which constrains tc_R to a transitive relation containing R but not the smallest one. We prove that WTC is complete for any *negative* transitive closure occurrence in the clause normal form (CNF) of F. Therefore, if the solver (falsely) reports F as satisfiable modulo WTC, it is only because of *positive* transitive closure occurrences. To extend the WTC fragment

F :
(1) $h_1 \,.\, mark = \emptyset$
(2) $h_0 \,.\, ref \subseteq h_1 \,.\, ref$
(3) $\forall n. \neg((root, n) \in tc_{H.ref}(h_1)) \vee n \in h_2 \,.\, mark$
(4) $h_1 \,.\, ref \subseteq h_2 \,.\, ref$
(5) $\forall n. \neg(n \notin h_2 \,.\, mark) \vee n \,.\, (h_3 \,.\, ref) = \emptyset$
(6) $\forall n. \neg(n \in h_2 \,.\, mark) \vee n \,.\, (h_3 \,.\, ref) = n \,.\, (h_2 \,.\, ref)$
(7) $(root, live) \in tc_{H.ref}(h_0)$
(8) $live \,.\, (h_0 \,.\, ref) \not\subseteq live \,.\, (h_3 \,.\, ref)$

WTC :
(9) $\forall h. h \,.\, ref \subseteq tc_{H.ref}(h)$
(10) $\forall h. Transitive(tc_{H.ref}(h))$

(a)

Essential R-path p:
$(root, live) \in tc_{H.ref}(h_0)$
Path invariant for p:
$n \in h_2 \,.\, mark$

(b)

F' :
$F \wedge WTC \wedge$
$\forall x. (root, x) \in tc_{H.ref}(h_0) \rightarrow x \in h_2 \,.\, mark$

(c)

Figure 8.2: Example. (a) Original formula and a weak transitive closure theory, (b) an unsafe R-path in F and its invariant, (c) augmented formula.

and to answer question (3), we introduce a technique which automatically detects relevant invariants about the R-paths in F and adds them as additional assumptions to F. If any of such invariants cause contradiction, F has been refuted and the process stops. Otherwise, more invariants will be detected and added to F.

8.1 Example

Figure 8.2(a) gives an RFOL formula F in CNF form —lines correspond to clauses. Symbols h_0 to h_3 are constants of type H that represents the system state; *root* and *live* are two constants of type *Obj* that represents objects; the binary relation $mark \subseteq H \times Obj$ represents the marked objects in each state; the ternary relation $ref \subseteq H \times Obj \times Obj$ represents references between objects in each state; and $tc_{H.ref} : H \rightarrow Obj \times Obj$ is a function that maps each state h to a binary relation $tc_{H.ref}(h) \subseteq Obj \times Obj$ which aims at representing the transitive closure of the relation $h \,.\, ref$[1]. The last two lines (WTC) give a weak semantics for $tc_{H.ref}(h)$. They constrain it to be transitive and to include the base relation, but not necessarily the smallest such relation. F gives the negated proof obligation of a safety property of an extremely simplified version of mark-and-sweep algorithm. The state transition (h_0–h_1) resets all the marks (Lines 1-2), (h_1–h_2) marks objects reachable from *root* (Lines 3-4), and (h_2–h_3) sweeps references of non-marked objects (Lines 5-6). The safety property is negated, thus it checks if in the final state, there is a *live* object that was originally reachable from *root* in the beginning state (Line 7), but some of its references have been swept (Line 8).

[1] As already mentioned, we omit the arity specification of the relational operators, but also the singleton operator for atoms, i.e., $h \,.\, ref$ is an abbreviation of $\{h\} \,._{1,3}\, ref$.

In our work [75], for example, we solved such formulas by adding general axioms about transitive closure. Here, for example, F can be refuted using the *subset preservation axiom* of transitive closure, namely $R \subseteq S \rightarrow R^+ \subseteq S^+$ for binary relations R and S. The only state transition in F that allows for sweeping object references is (h_2–h_3) —Line 5. Since (5) is guarded by the condition that the objects are not marked at h_2, to refute the formula, it is sufficient to show that all live objects are marked at h_2. Applying the subset preservation axiom above to Line 2, and using Line 7, we get $(root, live) \in tc_{H.ref}(h_2)$ which allows for closing the proof by any SMT solver. In [75] we collected more than 100 such transitive closure axioms, proved and added them as further deduction rules. Although the approach was useful for interactive and semi-interactive solving, the results of [13] suggest that this approach does not scale for automatic provers such as SMT solvers. [13] proposes to add these lemmas only on-demand based on some heuristics. In this approach, we go one step further and detect and add only the actually needed properties on-the-fly (as opposed to always include some general properties).

Our new approach refutes F by first solving $F \wedge WTC$ using an SMT solver. In this example, the safety property holds, and thus F must be unsatisfiable. The solver, however, (falsely) reports F as satisfiable. This is because WTC only fixes the semantics of transitive closure for *negative* R-paths in F —negated literals in the CNF of F of the form $(a, b) \in tc_R$— (thus called *safe* R-paths); *positive* R-paths remain as sources of incompleteness (thus called *unsafe* R-paths). However, the refutation of an unsafe R-path is not always mandatory for the refutation of F, this depends on F. Therefore, we are more interested in those unsafe R-paths whose refutation is mandatory for refuting F (we call them *essential* R-paths).

Our formula F in Figure 8.2(a) contains only one unsafe R-path p, namely $(root, live) \in tc_R$ (Line 7); p is essential since it is a unit clause in F[2]. We refute p by searching for some property $\varphi[x]$, called p-invariants, that

1. holds for all objects reachable from the beginning of p, namely *root*, by one R-step —we call this property the p-step test— and

2. if it holds for a node x, it holds for all nodes reachable from x by one R-step —we call this property the R-invariant test.

Given a p-invariant $\varphi[x]$, the induction principle allows us to add the assumption $\forall x_2.(root, x_2) \in tc_R \rightarrow \varphi[x_2/x]$ as an additional clause to F without affecting its validity. If one of the p-invariants is known to not hold for *live* (the end node of p), then p is refuted and we are done. Figure 8.2(b) shows the p-invariant which is sufficient to refute our essential R-path p. It is a subclause of clause (3) which passes both p-step and R-invariant tests. Details on the search procedure for p-invariants are presented in Section 8.4. After adding the p-invariant assumption to F (Figure 8.2(c)), the SMT solver reports it as unsatisfiable, and thus the example has been verified, namely fully automatically and using any SMT solver with standard quantifier support.

[2] In general the test for essential R-paths is not trivial.

8.2 Weak TC Axiomatization and its Fragment

In this section, we discuss a general[3], weak, first-order, integer-free axiomatization for transitive closure (denoted by WTC) and describe a fragment for which it is complete. The WTC axioms are given by the Equations in 8.5. They constrain the symbol tc_R to be a transitive relation that contains R (denoted by $tr(R)$). Therefore, their deductive closure $Cl(WTC)$ describes the theory of structures that interpret tc_R as $tc(R)$ (denoted by $\mathcal{T}^{tr(R)}_{tc_R}$). Generally, we use $\mathcal{T}^{g}_{f}$ to denote the theory which agrees with $\mathcal{T}$ except for the interpretations M of f , where it is interpreted the same way as $M(g)$. We also use $R|_u$ to denote the restriction of a relational term R on a tuple u of same arity, i.e., $R|_u = R \cap \{u\}$.

$$\begin{aligned} &\forall x_1, x_2.\ (x_1, x_2) \in R \rightarrow (x_1, x_2) \in tc_R \\ &\forall x_1, x_2, x_3.\ (x_1, x_2) \in tc_R \wedge (x_2, x_3) \in tc_R \rightarrow (x_1, x_3) \in tc_R \end{aligned} \tag{8.5}$$

Although the WTC axiomatization is very weak, there exists a *non-trivial* fragment for which this axiomatization is complete.

Theorem 8 (WTC completeness fragment)**.** *Let F be a first-order relational formula, R and tc_R two binary relations, and u a tuple such that the R-path $u \in tc_R$ occurs only as negative literal in $CNF(F)$. Then, F is unsatisfiable modulo $\mathcal{T}^{R^+}_{tc_R|u}$ iff it is unsatisfiable modulo $\mathcal{T}^{tr(R)}_{tc_R|u}$.*

Proof. Let u denote a tuple (a, b) and the R-path $(a, b) \in tc_R$ be denoted by p. Assuming that p occurs only as negative literal in $CNF(F)$, we need to prove that (1) if F is unsatisfiable modulo $\mathcal{T}^{tr(R)}_{tc_R|u}$, then it is unsatisfiable modulo $\mathcal{T}^{R^+}_{tc_R|u}$ too, and (2) if F has a $\mathcal{T}^{tr(R)}_{tc_R|u}$-structure, it has a $\mathcal{T}^{R^+}_{tc_R|u}$-structure too. Case (1) is trivial since $R^+ \subseteq tr(R)$. For case (2) we assume that $\mathcal{M}$ is a $\mathcal{T}^{tr(R)}_{tc_R|u}$-structure of F. For all clauses in $CNF(F)$ in which a literal other than $\neg p$ is satisfied, $\mathcal{M}$ is especially a $\mathcal{T}^{R^+}_{tc_R|u}$-structure because $\mathcal{T}^{tr(R)}_{tc_R|u}$ and $\mathcal{T}^{R^+}_{tc_R|u}$ coincide in symbols other than $tc_R|_u$. For all other clauses C, we can assume that $C := \neg p \vee C_{rest}$ and $\mathcal{M} \models \neg(a, b) \in tc_R$. Since $\mathcal{T}^{tr(R)}_{tc_R} = Cl(WTC)$, $\mathcal{M}$ is especially a structure for the second WTC axiom instantiated with a and b; $\mathcal{M} \models \forall x_2.\ (a, x_2) \notin tc_R \vee (x_2, b) \notin tc_R$. By induction, using the first axiom, there is no R-path from a to b in $\mathcal{M}$. Therefore $\mathcal{M}$ is a $\mathcal{T}^{R^+}_{tc_R|u}$-model for $\neg p$ and thus for C. □

In other words, theorem 8 states that if all R-paths in $CNF(F)$ are negative literals, then WTC is a correct and complete R^+-axiomatization of tc_R in F. It describes, therefore, a WTC complete fragment. The fragment conditions are syntactic and allow categorizing R-paths into safe —with only negative literals in the $CNF(F)$— and unsafe —otherwise. Hereafter, we denote the set of all unsafe R-paths by UP.

[3]Independent of the considered formula

8.3 R-Invariants for Axiomatizing Unsafe R-Paths

This section introduces *R-invariants* as a means for providing a transitive closure axiomatization that is *context-complete*, i.e., complete with respect to the context in which the transitive closure is used. This axiomatization handles unsafe R-paths, those for which the WTC axiomatization is not complete, and thus provides a proof possibility for formulas beyond the WTC-fragment described in Section 8.2. However, not all unsafe R-paths are indeed essential for the refutation of a given formula. Therefore, we base our approach on those unsafe R-paths which are *essential*.

Definition 20 (Essential unsafe R-paths)**.** Let R be a binary relation and F be a refutable first-order relational formula modulo $\mathcal{T}^{R^+}_{tc_R}$. Then, an unsafe R-path $(a,b) \in tc_R$ in UP is essential —for refuting F— if there exists a structure $\mathcal{M}$ where

$$\begin{aligned}&\forall (b,c) \in tc_Q : UP \setminus \{(a,b) \in tc_R\}.\ M(tc_Q|_{(b,c)}) = M(Q^+|_{(b,c)}),\\ &M(tc_R|_{(a,b)}) = M(tr(R)|_{(a,b)}) \text{and}\\ &\mathcal{M} \models F.\end{aligned}$$

The set of all essential (unsafe) R-paths is denoted by EP.

Definition 20 describes unsafe R-paths that require further axiomatization in order to refute F. The definition condition, however, requires a complete axiomatization of unsafe relational paths, which is in fact our ultimate goal. Therefore, we will later give a practical *heuristic* to check for essential R-paths.

Definition 21 (R-invariant)**.** Let F be a first-order formula and R a binary relation. Then, a formula $\varphi[x]$, containing a variable x, is a forward (resp. backward) R-invariant with respect to x, F and a theory $\mathcal{T}$ if

$$F \models_{\mathcal{T}} \forall x_1, x_2.\ \varphi[x_1/x] \wedge (x_1,x_2)^d \in R \rightarrow \varphi[x_2/x]$$

for $d = 1$ (resp. $d = -1$) , where $(x_1,x_2)^{-1} = (x_2,x_1)$.

Definition 22 (p-invariant)**.** Let F be a first-order formula, R a binary relation and p an R-path of the form $(a,b) \in tc_R$. Then, a forward (resp. backward) R-invariant formula $\varphi[x]$, containing a variable x, is forward (resp. backward) p-invariant with respect to x, F and a theory $\mathcal{T}$ if a (resp. b) is ground and

$$F \models_{\mathcal{T}} \forall x_2.\ (c,x_2)^{-d} \in R \rightarrow \varphi[x_2/x]$$

for $c = a$ and $d = 1$ (resp. $c = b$ and $d = -1$).

When using Definition 21 and 22, we may skip mentioning x, F and $\mathcal{T}$ when clear from the context. Unless explicitly stated, the forward definitions are meant.

Definition 23 (TC induction schema)**.** The first-order relational version of the induction axiom, denoted by IND, is a schema of axioms which states that for any closed first-order formula $\varphi[z_1, z_2]$, containing variables z_1 and z_2, the following hold:

$$\big(\forall x_{1:2}.\ (x_1, x_2) \in R \rightarrow \varphi[x_1/z_1, x_2/z_2] \wedge \tag{8.6}$$

$$\forall x_{1:3}.\ \varphi[x_1/z_1, x_2/z_2] \wedge (x_1, x_2) \in tc_R \wedge (x_2, x_3) \in R \rightarrow \varphi[x_1/z_1, x_3/z_2]\big) \tag{8.7}$$

$$\rightarrow$$

$$\forall x_{1:2}.\ (x_1, x_2) \in R^+ \rightarrow \varphi[x_1/z_1, x_2/z_2]$$

Definition 24 (Backward TC induction schema)**.** The first-order relational backward version of the induction axiom, is a schema of axioms which states that for any closed first-order formula $\varphi[z_1, z_2]$, containing variables z_1 and z_2, the following hold:

$$\big(\forall x_{1:2}.\ (x_1, x_2) \in R \rightarrow \varphi[x_1/z_1, x_2/z_2] \wedge \tag{8.8}$$

$$\forall x_{1:3}.\ \varphi[x_1/z_1, x_2/z_2] \wedge (x_1, x_2) \in tc_R \wedge (x_3, x_1) \in R \rightarrow \varphi[x_3/z_1, x_2/z_2]\big) \tag{8.9}$$

$$\rightarrow$$

$$\forall x_{1:2}.\ (x_1, x_2) \in R^+ \rightarrow \varphi[x_1/z_1, x_2/z_2] \tag{8.10}$$

For any refutable formula F modulo $\mathcal{T}^{R^+}_{tc_R|(a,b)}$ that contains an essential R-path p of the form $(a, b) \in tc_R$, we would like to claim the existence of a p-invariant formula φ, such that $(\forall x.(a, x) \in tc_R \rightarrow \varphi) \wedge F$ is refutable modulo $\mathcal{T}^{tr(R)}_{tc_R|(a,b)}$. We found it difficult to prove this claim using a $\mathcal{T}^{R^+}_{tc_R}$ theory, especially since any refutation proof of F has to be considered in a second-order proof system. Instead, we consider the $\mathcal{T}^{ind}_{tc_R}$ theory, which consists of the extension of $\mathcal{T}^{tr(R)}_{tc_R}$ with our induction schema for transitive closure (Definitions 23 and 24). This is indeed a restriction, since $\mathcal{T}^{ind}_{tc_R}$ only covers a recursively-enumerable set of properties —similar argument as in [50]. This is comparable to the gap between the first- and second-order Peano axiomatization of arithmetic (cf. [10, page 1133]). In practice, however, it imposes no restriction to the proof power and this is the common practice in literature (cf. [75, 5]).

Theorem 9 (Main theorem)**.** *Let R be a binary relation, F a first-order relational formula and p an unsafe R-path of the form $(a, b) \in tc_R$ in a clause C of F. If F is refutable modulo $\mathcal{T}^{ind}_{tc_R|(a,b)}$ but satisfiable modulo $\mathcal{T}^{tr(R)}_{tc_R|(a,b)}$, then there exists a forward (resp. backward) p-invariant $\varphi[x]$ w.r.t. x, $F \setminus C$ and $\mathcal{T}^{tr(R)}_{tc_R|(a,b)}$, such that*

$$(\forall x_2.\ (a, x_2) \in tc_R \rightarrow \varphi[x_2/x]) \wedge F$$

$\big($resp. $(\forall x_2.\ (x_2, b) \in tc_R \rightarrow \varphi[x_2/x]) \wedge F\big)$ is refutable modulo $\mathcal{T}^{tr(R)}_{tc_R|(a,b)}$.

Proof. For simplicity and without loss of generality we restrict our self to the forward case, i.e., all applied TC inductions are forward. Without loss of generality, we can assume that $\mathcal{T}^{tr(R)}_{tc_R|(a,b)}$ differs from $\mathcal{T}^{ind}_{tc_R|(a,b)}$ only in the interpretation of $tc_R|_{(a,b)}$, and p

only occurs in C. Therefore, $F \setminus C$ must be satisfiable modulo $\mathcal{T}^{ind}_{tc_R|(a,b)}$. This means that since F is refutable modulo $\mathcal{T}^{ind}_{tc_R|a,b}$ but satisfiable modulo $\mathcal{T}^{tr(R)}_{tc_R|(a,b)}$, for each $\mathcal{T}^{ind}_{tc_R|(a,b)}$-structure $\mathcal{M}$ of $F \setminus C$, $\mathcal{M} \models (a,b) \notin tc_R$, which in turn means

$$F \setminus C \models_{\mathcal{T}^{ind}_{tc_R|(a,b)}} (a,b) \notin tc_R.$$

Let us further consider a proof object pr (for example, in sequent style) for $F \setminus C \models_{\mathcal{T}^{ind}_{tc_R|(a,b)}} (a,b) \notin tc_R$, then the set $IP := \{\varphi_1[x_1,x_2],\dots \varphi_n[x_1,x_2]\}$ of all formulas of all essential IND applications in pr is not empty. Let $\Gamma := \{\phi_i[x_1,x_2] := \forall x_1,x_2.\ (x_1,x_2) \in tc_R \rightarrow \varphi_i[x_1,x_2] \mid \varphi_i \in IP\}$. Since IP contains all formulas of all essential IND applications in pr, we can conclude that

$$\Gamma, F \setminus C \models_{\mathcal{T}^{tr(R)}_{tc_R|(a,b)}} (a,b) \notin tc_R.$$

Note, that a proof pr' of the last sequent does not contain any essential IND application. Therefore, we can assume, without los of generality, the existence of a formula $\varphi_k[x_1,x_2] \in IP$ where $F \setminus C \models_{\mathcal{T}^{tr(R)}_{tc_R|(a,b)}} \neg\varphi_k[a/x_1,b/x_2]$. Now we construct φ as follow

$$\varphi := \bigwedge_{\varphi_i \in IP} \varphi_i[a/x_1]$$

and prove that φ fulfills all the conditions of the theorem.

All $\varphi_i \in IP$ have to fulfill the first and second IND conditions (Equations 8.6 and 8.7). By instantiating x_1 with a in the IND conditions, we get directly that all φ_is are p-invariants w.r.t. x_2, F and $\mathcal{T}^{R^+}_{tc_R|(a,b)}$, which also holds especially for the $\varphi_i[a/x_1]$s. Now it is easy to see that φ is a p-invariant since it is a conjunction of p-invariants.

In order to proof that φ fulfill the last condition of the theorem, we assume that

$$\big(\forall x_2.\ (a,x_2) \in tc_R \rightarrow (\bigwedge_{\varphi_i \in IP} \varphi_i[a/x_1])\big) \wedge F$$

has a satisfying $\mathcal{T}^{tr(R)}_{tc_R|(a,b)}$-structure $\mathcal{M}$. Consequently, and especially,

$$(a,b) \in tc_R \rightarrow (\bigwedge_{\varphi_i \in IP} \varphi_i[a/x_1,b/x_2])$$

is also true in $\mathcal{M}$. Since $(a,b) \in tc_R$ is essential in F, $\varphi_k[a/x_1,b/x_2]$ is, especially, true in $\mathcal{M}$. But this is a contradiction to our earlier result $F \setminus C \models_{\mathcal{T}^{tr(R)}_{tc_R|(a,b)}} \neg\varphi_k[a/x_1,b/x_2]$. □

Theorem 9 offers a basis for a framework capable of proving the validity of transitive closure formulas beyond the WTC fragment. Especially, for each essential R-path p, the theorem guaranties the existence of a p-invariant which is deducible from F modulo $\mathcal{T}^{ind}_{tc_R}$ and can together with F refute p. In the next section we show how the conditions of the theorem on φ can be turned into practical rules and heuristic algorithms to direct the search for p-invariants.

8.4 Algorithm for Detecting p-invariants

In order to provide an automatic procedure capable of proving transitive closure formula beyond the WTC fragment, we developed an algorithm which tries to bring the theoretical results of the previous sections into action. Before discussing the actual algorithm, some definitions and lemmas are needed.

We first discuss two concepts introduced and used in the last section: (1) *essential R-paths*, and (2) *R-path isolation*, i.e., the consideration of F modulo the theory $(\mathcal{T}^{R^+}_{tc_R})^{tr(R)}_{tc_R|(a,b)}$ which except of a specific R-path $(a,b) \in tc_R$ interprets all other R-paths as edges in R^+ (cf. proof of Theorem 9). The latter concept —R-path isolation— subsumes the former one and is of particular importance for the automation process. It allows for detecting essential R-paths and for handling the WTC incompleteness for each R-path individually regardless of other relational paths. However, the second concept requires $\mathcal{T}^{R^+}_{tc_R}$ which is our actual goal. In order to overcome this, we introduce in Definition 25 the idea of *n-confident R-paths isolation*.

Definition 25 (n confident R-path isolation)**.** Let R be a binary relation, F a first-order relational formula, p an unsafe R-path in F and n a positive natural number. Then, the n confident isolation of p in F is

$$F|^n_p := F\big[\{(c,d) \in \bigcup_{i \leq n} Q^{(i)} \ / \ (c,d) \in tc_Q \mid ((c,d) \in tc_Q) \in UP \setminus \{p\}\}\big].$$

Here, the isolation $F|^n_p$ of a given R-path p in F, is a formula obtained by replacing each Q-path $(c,d) \in tc_Q$ —of an arbitrary length— different than p in F with a corresponding Q-path of length less equal n, where n is the isolation confidence and $Q^{(i)}$ denotes joining Q with itself i-times. Note that for the definition of $F|^n_p$, we extend the notation for variable substitution to describe a set of term replacement.

Having $F|^n_p$ in disposition, we can check its satisfiability using an SMT solver and conclude:

1. if $F|^n_p$ is satisfiable, then p is certainly and regardless of n —even for $n := 1$— essential in F, and otherwise
2. p is not essential in F with the confidence n —the greater the n, the less essential the p.

Algorithm 2 shows the main procedure of our approach. Given a refutable formula F modulo $\mathcal{T}^{R^+}_{tc_R}$, it will first detect all essential R-paths by checking the satisfiability of the n confident isolation $F|^n_p$ of all unsafe R-paths p (line 3). The isolation confidence n, is only increased if $F|^n_p$ is unsatisfiable for all essential R-paths in EP but F is not (lines 20-21).

For each essential R-path p of the form $(p_s, p_e) \in tc_R$, we search for forward p-invariants with respect to its start boundary p_s and backward p-invariants with respect to its end boundary p_e. If the currently handled path boundary, p_g, is ground, which

Algorithm 2: Main Procedure

Data: $F : Term$
Result: $Term$

$F^{ini} \leftarrow CNF(\neg F)$; $F \leftarrow F^{ini}$; $n \leftarrow 1$
repeat
  $EP \leftarrow \{p \in UP(F^{ini}) \mid sat(F^{ini}|_p^n)\}$
  for $p := ((p_s, p_e) \in tc_R) \in EP$ **do**
    for $\langle p_g, d\rangle \in \{\langle p_s, 1\rangle, \langle p_e, -1\rangle\}$ **do**
      if $p_g \in Gr$ **then**
        $F \leftarrow pathInv(p, p, p_g, F, F^{ini}, R, d, n)$
        if $unsat(F)$ **then**
          return F
      else
        $x_{1:n} \leftarrow Var(p_g)$
        for $p' := (p'_s, p'_e) \in \{p[a_{1:n}/x_{1:n}] \mid a_i \in sufGT^1(x_i)\}$ **do**
          if $sat(F[p'/p]|_{p'}^n)$ **then**
            $p'_g \leftarrow d \; ? \; p'_s : p'_e$
            $F \leftarrow pathInv(p, p', p'_g, F, F^{ini}, R, d, n)$
            if $unsat(F)$ **then**
              return F
        if $(\forall p'.\, unsat(F[p'/p]|_{p'}^n)) \wedge sat(F|_p^n)$ **then**
          Further/General techniques are needed
  if $\forall p : EP.\, unsat(F|_p^n)$ **then**
    $n \leftarrow n + 1$
until *F and n are unchanged;*
return F

Algorithm 3: *pathInv*

Data: $p, p', p_g, F, F^{ini} : Term, R \subseteq T \times T, d, n : Int$
Result: $Term$

for $\varphi[x_{1:n}] \in (F^{ini} \setminus C_p)$ *with* $p_g \sqsubseteq type(x_i)$ **do**
  for $x_i \in \{x_{1:n}\}$ **do**
    $F \leftarrow concPathInv(\varphi, p, p', p_g, F, F^{ini}, x_i, R, d, n)$
    if $unsat(F[p'/p]|_{p'}^n)$ **then**
      return F
return F

Algorithm 4: *concPathInv*

Data: $\varphi, p, p', p_g, F, F^{ini} : Term, x : Var, R \subseteq T \times T,$
$d, n : Int$
Result: $Term$

```
for φ_i[x] ⊆ φ do
    F ← checkPathInv(φ_i, x, p_g, F, R, d)
    if unsat(F[p'/p]|^n_{p'}) then
        return F
    for φ'_i[x] ∈ abst(φ_i, F^ini, x, R, n) do
        F ← checkPathInv(φ'_i, x, p_g, F, R, d)
        if unsat(F[p'/p]|^n_{p'}) then
            return F
return F
```

Algorithm 5: *abst*

Data: $\varphi, F : Term, x : Var, R \subseteq T \times T, n : Int$
Result: $Set < Term >$

```
S ← {φ}; A ← ∅
for φ_i ∈ S do
    for abst ∈ {applicable abstraction rules to φ_i} do
        A ← A ∪ abst(φ_i, x, R, n); S ← S ∪ abst(φ_i, x, R, n)
    S ← S \ {φ_i}
return A
```

Algorithm 6: *checkPathInv*

Data: $\varphi, t, p_g, F : Term, R \subseteq T \times T, d : Int$
Result: $Term$

```
begin
    PO_ini ← ∀x_2. (p_g, x_2)^d ∈ R → φ[x_2/t]
    PO_ind ← φ[x_2/t] ∧ (p_g, x_2)^d ∈ tc_R ∧ (x_2, x_3)^d ∈ R
    PO_ind ← ∀x_2, x_3. PO_ind → φ[x_3/t]
    if unsat(F ∧ ¬PO_ini) ∧ unsat(F ∧ ¬PO_ind) then
        F ← (∀x_2. (p_g, x_2)^d ∈ tc_R → φ[x_2/t]) ∧ F
        return F
```

corresponds exactly to the considered case in Theorem 9, the search is performed for the original R-path p by Algorithm 3. Otherwise, instances of p are used (line 12-13). The p instances are generated by instantiating the variables of p_g with their essential ground terms of complexity 1 —constants— using a slightly modified version of our sufficient ground term sets framework presented in Chapter 7 and published in [34][4]. The R-path instantiation approach is motivated by the guess that probably only a *small* finite set of p instances are refutable. When considering an R-path instance p' of an R-path p, one has to perform the n-confident path isolation of p' in $F[p'/p]$ instate of F, since p' originally does not exists in F.

In Algorithm. 3, each clause φ of $CNF(F)$ —after excluding p's clauses— that contains a non empty set of variables $x_{1:n}$ of a type compatible to p_g is considered for the p-invariant search, namely with respect to each x_i in $\{x_{i:n}\}$ (line 1-2). Since all variables in φ are universally quantified, φ is obviously a p-invariant with respect to any variable x_i, however, we are interested in more concrete forms of φ. This is described in Algorithm 4, where, each sub clause φ_i of φ that contains x_i is considered a candidate. The actual check for p-invariance is performed in Algorithm 6. Depending on weather p_g is a start or end boundary, the forward or backward definition of p-invariants is used respectively. If the p-invariant check fails for a candidate φ_i, syntactically-driven abstractions are generated and tried (Algorithm 5).

Our abstraction rules are shown in Figure 8.3. The first rule abstracts a φ_i by instantiating their variables —x_i excluded— with their essential ground terms of complexity equal to the current calculation round r. The second rule relaxes positive literals —conclusions— in φ_i by their *syntactic* consequences in F. The third rule is only used if a p-invariant candidate passes the p-step test (cf. PO_{ini} in Algorithm 6) but fails in the R-invariant test. It then relaxes *unary* assumptions on a single path boundary such that they hold for all reachable nodes from that boundary including itself —reachability direction is stated by d. Let's assume a clause C in F of the form $(a, x_2) \in R \wedge \phi(a) \rightarrow \varphi_{rest}[x_2]$ and an R-path p of the form $(a, x_2) \in tc_R$. Then, the p-invariant candidate φ equal to $\phi(a) \rightarrow \varphi_{rest}[x_2]$ will pass the p-step check using C only. If φ does not pass the R-invariant test, then our third abstraction rule can abstract it such that it passes both tests using C only.

Abst_1:	Variable instantiations with essential ground terms of complexity r, using [34]
Abst_2:	$\varphi := (l \vee \varphi_{rest}), (\neg l \vee C_{rest}) \in CNF(F) \Longrightarrow \varphi \rightsquigarrow \varphi[C_{rest} \ / \ l]$
Abst_3:	$\varphi := (\neg\phi(t) \vee \varphi_{rest}), t := p_g \Longrightarrow \varphi \rightsquigarrow \varphi[(\forall x.\ x = t \vee (t,x)^d \in tc_R \rightarrow \phi(x)) \ / \ \phi(t)]$

Figure 8.3: Abstraction rules

[4]The essential ground terms are calculated in rounds with increasing term complexity, regardless of whether the set is finite or not.

If, in the case of a non-ground p_g, all R-path instances p' can be refuted but not the original path p, we directly switch to a more general technique (Algorithm 2 line 18-19). Basically, the technique is a natural extension of the framework presented in Section 8.3 to explicitly consider R-paths with non-ground boundaries. This technique was employed in only one of our benchmarks.

8.5 Evaluation

We have implemented a prototype version of the procedure described in Section 8.4. In the current implementation, we fixed both the isolation confidence (Algorithm 2 line 20-21) and the ground term complexity (fig. 8.3 $Abst_1$) parameters to 1. To evaluate our technique, we checked 20 Alloy assertions that were expected to be correct. These benchmarks were taken from the Alloy Analyzer 4.2 distribution and involve transitive closure of varying complexities. In order to provide a fair evaluation of the technique, we have restricted the considered benchmarks to those that require the semantics of transitive closure for their correctness proof.

Since most Alloy benchmarks that involve transitive closure also involve trace specifications (based on the Alloy ordering library), we developed a reduction of Alloy trace specifications to transitive closure specifications. That is, we represent any ordered signature S which forms the base of a trace specification, as the set $\mathit{first} \cup \mathit{first} \,.\, \mathit{next}^+$ where *first* denotes the starting atom of the trace and $\mathit{next} \subseteq S \times S$ is a fresh acyclic relation denoting the ordering. If a trace invariant is known, we divide the original benchmark to (1) an invariant proof and (2) an invariant use benchmark. Such reduction is used for two of our Alloy benchmarks: *addrbooktrace* and *hotelroom*.

Table 8.1 shows the experimental results[5] performed using Z3 4.3.1 on an Intel Xeon, 2.7 GHz, 64GB memory. For each checked benchmark, we collect the number of R-paths, unsafe R-paths, essential R-paths, checked p-invariant candidates, proved and injected p-invariants and the total analysis time (in seconds). Time-out is set to 12 hours for the entire analysis and to 1 minute for each call to the SMT solver. Out of 20 benchmarks assumed to be valid, 18 were proven correct by our tool. It should be noted that these benchmarks are absolutely not trivial. For example, our previous (fix) axiomatization based on integer theory and semantics blasting could not prove any of the benchmarks with essential R-paths at all (cf. Chapter 6), and although Kelloy could prove all benchmarks, it required substantial human interactions, even for the *com* benchmarks, which do not contains essential R-paths at all (cf. [75]).

A surprising observation is that quite a large number, 13 out of 20, of Alloy problems that involve transitive closure, do not contain any essential R-paths, which lets them be *effectively* in the WTC fragment, although not syntactically. This fully answers our question of why in our earlier investigation (in Chapter 6), some transitive closure benchmarks could be proven but not others. It shows that only a very small part of

[5] Benchmarks, results and tool are available at `http://i12www.ira.uka.de/~elghazi/tcAx_via_p-inv/`

that integer based transitive closure axiomatization, namely the WTC axioms, was actually responsible for the success.

All 13 benchmarks with no essential R-paths could be proven fully automatically in less than 2 seconds using WTC and without the need of any p-invariant injection. For these examples, according to Theorem 8, if the SMT solver had reported a satisfying structure, it would have been a valid one. Out of the remaining 7 benchmarks containing essential R-paths, our tool could prove 5. The number of injected p-invariants varies between 1, for *soundness1*, and 159, for *completeness*. The number of injected p-invariants is not guaranteed to reflect the number of needed p-invariants since it depends very much on the ordering of essential R-paths and CNF clauses. However, it does reflect that for all of our proven benchmarks except the last two. The benchmarks *hotelroom-locking* and *javatypes-soundness* could not be proven by our tool. For both benchmarks, the main difficulty lies in the complexity of our generated SMT formulas which makes them too difficult to solve by Z3. For *hotelroom-locking*, the proof obligations for the essential R-path checks could be handled, but none of the p-invariant checks, whereas for *javatypes-soundness* every single call of the solver times-out. This shows the dependency of the current version of our approach on analysable SMT representations.

Benchmarks	Result	All/Dif/Ess Paths	Che. p-inv	Inj. p-inv	Time
addrbook-addIdempotent	proved	5 / 2 / 0	0	0	0,08
addrbook-delUndoesAdd	proved	5 / 2 / 0	0	0	0,10
addrbooktrace-addIdempotent	proved	23 / 17 / 0	0	0	0,25
addrbooktrace-delUndoesAdd	proved	20 / 14 / 0	0	0	0,21
addrbooktrace-lookupYields-use	proved	22 / 13 / 0	0	0	0,24
grandpa-noSelfFather	proved	6 / 3 / 0	0	0	0.09
grandpa-noSelfGrandpa	proved	6 / 3 / 0	0	0	0.09
com-theorem1	proved	5 / 2 / 0	0	0	0,18
com-theorem2	proved	5 / 2 / 0	0	0	1.73
com-theorem3	proved	5 / 2 / 0	0	0	0.24
com-theorem4a	proved	5 / 2 / 0	0	0	0.25
com-theorem4b	proved	5 / 2 / 0	0	0	0.13
filesystem-noDirAliases	proved	7 / 4 / 0	0	0	0.12
filesystem-someDir	proved	5 / 3 / 1	2	1	0.15
marksweepgc-soundness1	proved	15 / 9 / 1	38	1	9,29
marksweepgc-soundness2	proved	16 / 10 / 2	75	2	5,92
marksweepgc-completeness	proved	16 / 8 / 2	1021	159	66,58
addrbooktrace-lookupYields-proof	proved	18 / 11 / 2	271	41	79,67
hotelroom-locking	timeout	6 / 3 / 1	–	–	–
javatypes-soundess	timeout	116 / 19 / –	–	–	–

Table 8.1: Evaluation results

8.6 Related Work

Several approaches have addressed the verification of Alloy problems in general. Due to the undecidability of the Alloy language, most of these approaches are based on interactive solving. Prioni [5] and Kelloy [75] rely on reasoning in first-order logic and integer arithmetic, Dynamite [37] chose a reasoning in fork algebras —a higher-order logic. In all these general approaches the verification of transitive closure formulas is in general *interactive*. In addition to definition rules, an induction schema is involved either directly or indirectly —for proving general lemmas.

Closer to our approach, are the works of Nelson [61] and Ami [56]. Nelson proposes a set of first-order axioms for axiomatizing the reachability between two objects following a *functional* relation f. To handle the presence of cycles he uses a ternary predicate $a \overset{f}{\underset{c}{\rightarrow}} b$ stating that b is reachable from a via arbitrary f applications, but never going through c. Later works, as in [54, 52, 28], revisited and extended Nelson's ideas. The main problem with such *fixed* first-order axiomatizations of transitive closure is that it is unlikely that they are complete. Ami proves in [56] that Nelson's axioms are not complete even in the functional setting. More directly, we can provide a very simple refutable formula modulo transitive closure which is satisfiable in Nelson's axioms, i.e., $a \overset{f}{\underset{b}{\rightarrow}} b \wedge \forall x.\, f(x) \neq b$. In our approach, however, the f-path from a to b can be easily refuted since the empty clause —$false$— is a backward invariant for this path. Ami's work, also motivated by Nelson's work, proposes, instead, three axiom *schemas*, which follow from a transitive closure induction schema. This is similar to our approach in that the axiom set is not fixed, but generated on-demand. However, their approach differs significantly from ours in that: (1) only *unary* predicates and their boolean combinations are considered as instantiation formulas for the axiom schemas, (2) the search for instantiation formulas is not essential R-path directed —not even R-path directed, (3) no R-path isolation criteria is involved —for example, to detect already refuted R-paths, and finally (4) no abstractions are used, even not variable instantiations.

Other tools such as ACL2 [49] and IsaPlanner [24] are well established in the automation of general induction schemas, for years. We think that our procedure and implementation can definitively profit from their ideas, especial their lemma discovering routine, called *lemma calculation*, and lemma abstraction ideas.

8.7 Conclusion

In this chapter, we have presented an approach capable of proving Alloy specifications that involve transitive closure full automatically. For all transitive closure occurrences the WTC axiomatization is introduced. In case the Alloy specification includes neither *unsafe* R-paths —syntactical check— nor essential R-path —semantical check— we have proved that WTC is a complete axiomatization of transitive closure and thus the solver result —either *sat* or *unsat*— can be trusted. Otherwise, each essential

R-path can be handled on its own thanks to our bounded *R-path isolation* concept. The incompleteness of WTC is adjusted for an essential *R*-path *p* by a directed detection and injection of so called *p-invariants*.

Although in theory our *p*-invariant detection procedure is guaranteed to terminate, this has little significance in practical terms, as we could observe for some benchmarks. From both, the conceptual as well as the engineering point of view, there is plenty room for improvement. This includes (1) the reduction of redundancy with respect to *p*-invariant candidates, and instantiation of paths and formulas, (2) the introduction of heuristics for the prioritization of paths, clauses, instantiations and abstractions, and (3) the further, also conceptual, investigation of essential *R*-paths with non-ground boundaries. At least for (1) and (2) we think that we can profit from well established tools in the area of induction automation like ACL2 [49], and IsaPlanner [24], even though their focus is different.

Chapter 9

JKelloy – A Deductive Relational Engine for Verifying Java Programs

So far, we have presented in the previous chapters our approach (respectively approaches) for the automatic verification of Alloy problems. Here, an Alloy problem describes a safety property of a software system, where both the safety property as well as the software system are specified using Alloy [47]—a *declarative* first-order relational logic with built-in operators for transitive closure, set cardinality, integer arithmetic, and set comprehension. Thereby, only an abstract version of the software system is considered.

The consideration of software abstractions allows especially for capturing the core of the software system (data structures, algorithms, etc.) while encapsulating implementation details that may cumber the verification process. For this task, we believe, and think have demonstrated, that Alloy is a suitable logic, language and tool, particularly for software systems with complex manipulation of linked data structures.

However, whereas an abstract specification of the safety property is sufficient —even perfect, a software system has ultimately to be considered in its detailed implementation level since this is how it is in fact used. The standard way of verifying complex software implementations, let say in Java, is to use (1) a Java related language for the specification of the safety property such as the Java modeling language (JML), and (2) a deductive verification tool that can establish the semantics of the safety property and the Java program and reason over them such as the KeY system [12, 77]. Thereby, the safety property, which in general and naturally is abstract, has to be written against the program implementation. This let safety property specifications tend to be unnecessary complex and erroneous, especially for complex manipulation of linked data structures.

However, the efficiency of specifying and verifying properties about linked data structures depends to a large extent on both the level of abstraction of that data structure and the conciseness of expressing properties over its reachable elements.

A suitable formalism for expressing such properties that can also be utilized in the context of deductive reasoning is relational logic with a transitive closure operator. In this logic, the links of the data structures can be modeled as binary relations, and thus reachability can be expressed using transitive closure. Furthermore, relational specifications allow the user to easily abstract away from the exact order and connection of elements in a data structure by viewing it as a set. This reduction of precision, when applicable, pays off in simplification of proofs as well as in better readability of the specifications and the intermediate verification conditions, which is important for user interaction.

In this chapter we describe JKelloy, our extension of the deductive Java verification tool KeY [12, 77], to support specifications written in the relational specification language Alloy [47]. To the best of our knowledge, this work is the first attempt in this direction; other related approaches either restrict the analysis to bounded domains (e.g. [76, 78, 38, 4]) or focus only on the Alloy models of systems without considering their implementations (e.g. [37, 75, 5]).

In our previous work [75, 74], done in the context of the master thesis of Ulrich Geilmann [40], we extended the *Java Dynamic Logic (JavaDL)*, the input logic of KeY, to a relational Java dynamic logic (relational JavaDL) capable of reasoning over general Alloy expressions. The extension, called Kelloy, is almost similar to our RFOL logic (cf. Chapter 4) but is rule based in order to fit within KeY's deductive reasoning. This, however, is not sufficient for handling Alloy as a specification language for Java programs since it has no explicit model of program state change.

JKelloy assumes a *relational view* of the Java heap: classes are modeled as Alloy signatures and fields as binary relations. To evaluate Alloy expressions in different program states, e.g., pre- and post-state of a method, we translate Alloy relations into functions which take the heap (representing the program state) as an argument. We define the relationship between Alloy relations and Java program states using pre-defined *coupling axioms*. This eliminates the need for the user to provide coupling invariants manually. Changes to program states are aggregated as heap expressions. We introduce an automatic transformation of those heap expressions to relational expressions using a set of *heap resolution rules* that normalizes all intermediate heap expressions. The transformation allows us to reason about verification conditions in the relational logic. To simplify the reasoning process, we further introduce a set of *override simplification rules* that exploit the specific shape of the resulting conditions. To increase the degree of automation, we have developed two *proof strategies* that control the application of our rules. We have proved the correctness of all rules using KeY.

Given a Java program, JKelloy can also generate an *Alloy context* that maps the class hierarchy of the program to a semantically equivalent Alloy type hierarchy. This allows the user to check the consistency of the specifications using the automatic, lightweight Alloy Analyzer before starting the full, possibly interactive, verification process. Building on top of KeY enables the user to take advantage of the supported SMT solvers to prove simple subgoals. It also lets the user provide additional lemmas. Complex lemmas, e.g., those that contain transitive closure over update expressions,

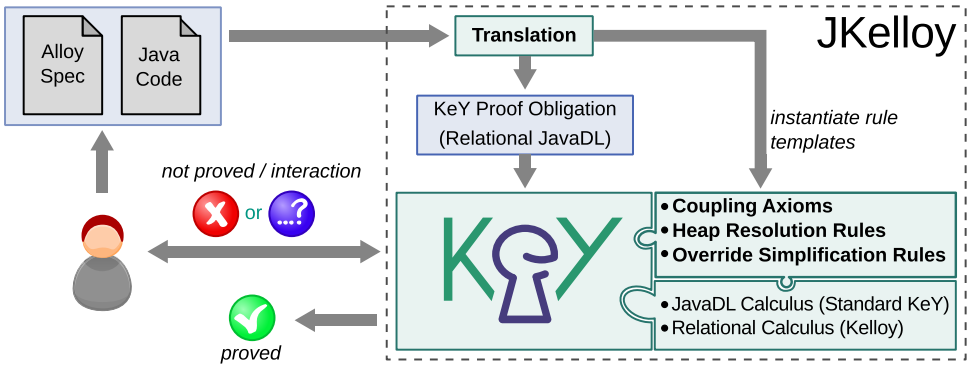


Figure 9.1: Overall Framework. Contributions highlighted in a boldface font.

can be proved by using induction in side-proofs, and then be reused to automatically prove non-trivial verification conditions without requiring induction.

9.1 Overall Framework

Our verification tool JKelloy extends KeY [12], a deductive verification engine that supports both automatic and interactive verification of Java programs. Figure 9.1 presents the general structure of JKelloy as well as the user's workflow. The input of the tool is a Java program together with its specification written in Alloy [47]. JKelloy follows the *design-by-contract* [58] paradigm in which every method is specified individually with pre- and post-conditions and a modifies-clause. Verification is performed method by method, in a modular way. For simpler programs and properties, the verification may run through automatically. In other cases, some user interaction may be required, in which the user guides the steps taken by the prover.

JKelloy extends KeY with a translation front-end that converts Alloy specifications of Java methods to relational JavaDL, the relationally extended input logic of KeY. JKelloy augments KeY with heap-dependent relations for modeling Java classes and fields. Furthermore, JKelloy introduces a set of calculus rules that facilitates verification of relational specifications. Some of these rules are program-dependent, and are generated for each program during the translation by instantiating pre-defined templates. The verification process for a method contract typically proceeds as follows:

1. The Alloy pre- and post-conditions are translated to relational JavaDL. The relations in the conditions become relational symbols depending on a heap-state. Their evaluation in a heap state is defined by *coupling axioms*.

2. The code of the Java method is symbolically executed, computing the post-heap-state in relation to the pre-heap.

3. *Heap resolution rules* are applied to normalize the resulting heap-dependent expressions so that all heap arguments become constant.

4. The resulting proof obligation is relational and can be discharged using the relational calculus. *Override simplification rules* simplify this process by providing additional lemmas in relational logic, exploiting the shape of the resulting conditions.

9.2 Alloy as Specification Language for Java Programs

Alloy [47] is a first-order relational logic with built-in operators for transitive closure, set cardinality, integer arithmetic, and set comprehension, which make it particularly suitable for concisely specifying properties of linked data structures. Properties of object-oriented programs can be specified in Alloy using the *relational view of the heap* [76]. That is, every class is viewed as a set of objects, and every field as a relation from the class in which the field is declared to its type. Our representation of this relational view differs from other approaches [76, 78] in that it provides an *explicit* encoding of the Java types in the pre- and post-state.

Given a Java program, JKelloy automatically generates an *Alloy context* which encodes the type hierarchy of that program, and declares all the relations accessible to the user for writing the specifications. The user can then add the specifications to this context in order to check their consistency using the Alloy Analyzer before starting the verification process using JKelloy. Although the Alloy Analyzer checks Alloy models only for bounded domains, it helps users detect flaws automatically: under-specifications and errors can be detected using the visualizer tool in the Alloy Analyzer, whereas over-specification can be detected using the unsat-core generator tool.

Figure 9.2(a) provides a sample Java program. It implements a singly linked list that stores `Data` objects, where `Data` is declared as an interface with two sample implementations. The method `prepend` adds a `Data` object to the beginning of the list.

Figure 9.2(b) presents the corresponding Alloy context. A signature declaration `sig A{}` declares `A` as a top-level type (set of uninterpreted atoms); `sig B in A{}` declares `B` as a subtype (subset) of `A`. The `extends` keyword has the same effect as the keyword `in` with the additional constraint that extensions of a type are mutually disjoint. An attribute `f` of type `B` declared in signature `A` represents a relation $\texttt{f} \subseteq \texttt{A} \times \texttt{B}$. The multiplicity keyword `one`, when followed by a set, constrains that set to be a singleton, and when used as a type qualifier of a relation, constrains that relation to be a total function.

The generated Alloy context always contains a singleton `Null` (Fig. 9.2(b) Line 1) which represents the Java `null` element. Every Java class `C` is represented by two signatures, `C` and `C'`, that give the set of atoms corresponding to the allocated objects of type `C` in the pre- and post-state, respectively. The top-level Java class `Object` is always included. The Alloy signature `Object` is constrained to be a subset of `Object'` (Line 3).

```
1  class List {
2    Entry head;
3
4  /*@ requires true;
5    @ ensures self . head' . *next' . data'
6    @  = self . head . *next . data + d;
7    @*/
8   void prepend(Data d) {
9     Entry oldHead = head;
10    head = new Entry();
11    head . next = oldHead;
12    head . data = d;
13  }
14 }
15
16 class Entry {
17   Data data;
18   Entry next;
19 }
20
21 interface Data { . . }
22 class ID implements Data { . . }
23 class Name implements Data { . . }
```

(a)

```
1  one sig Null {}
2  sig Object' {}
3  sig Object in Object' {}
4  sig List' extends Object' {
5    head': one (Entry'  + Null)
6  }
7  sig List in Object {
8    head: one (Entry  + Null)
9  }
10 sig Entry' extends Object' {
11   data': one (Data'  + Null),
12   next': one (Entry'  + Null)
13 }
14 sig Entry in Object {
15   data: one (Data  + Null),
16   next': one (Entry  + Null)
17 }
18 sig ID' extends Object' {..}
19 sig ID in Object {..}
20 sig Name' extends Object' {..}
21 sig Name in Object {..}
22 sig Data' in Object' {..}
23 sig Data in Object {..}
24 fact {
25   List = List'  & Object
26   Entry = Entry'  & Object
27   ID = ID'  & Object
28   Name = Name'  & Object
29   Data' = Name'  + ID'
30   Data = Name  + ID
31 }
32 pred pre[self: one List, d: one (Data  + Null)] {}
33 pred post[self: one List, d: one (Data  + Null)] {
34   self.head'. * next'.data' = self.head. * next.data  + d
35 }
```

(b)

Figure 9.2: (a) Sample code (b) Alloy context along with pre- and post-conditions

This allows new objects to be created, but created objects cannot be deallocated. That is, garbage collection is not considered. If a Java class `B` extends a class `A` (immediate parent), the signature `B'` will be an extension of `A'`, and `B` a subset of `A`[1]. Furthermore, any pre-state signature `C` is constrained to be the intersection of its corresponding post-state signature `C'` and the signature `Object` (e.g. Lines 25–28). This ensures both $\texttt{C} \subseteq \texttt{C'}$ and $\texttt{C} \subseteq \texttt{Object}$. Signatures for interfaces denote the union of the signatures for the classes that implement them (Lines 29-30).

A Java field `f` of type `T` declared in a class `C` is represented by two functional relations $\texttt{f: C} \rightarrow \texttt{(T} \cup \texttt{Null)}$ for the pre-state, and $\texttt{f': C'} \rightarrow \texttt{(T'} \cup \texttt{Null)}$ for the post-state (e.g. Lines 5, 8). Since $\texttt{C} \subseteq \texttt{C'}$, the domain of `f'` includes `C` as well.

[1]Semantically, the pre-state signature `B` must be an extension of the pre-state signature `A` rather than a subset. However, pre-state signature hierarchy goes up to the `Object` signature which is the subset of `Object'`, and Alloy does not allow subset signatures (in this case, `Object`) to have extensions. Nonetheless, it is easy to show that our other constraints imply subclasses of a class to be disjoint in the pre-state.

Pre-conditions of a method can access the receiver object (`self`) and that method's arguments. Post-conditions can additionally access the method's return value (`ret`) if any exists. These are given as parameters of the predicates `pre` and `post`, respectively (Lines 32-33). The user can copy the pre- and post-conditions of the analyzed method as bodies of these predicates, and check their consistency by providing Alloy assertions or by simply running the visualizer to see their satisfiable instances.

Specifications must be legal Alloy formulas. Basic formulas are constructed using subset (`in`) and equality (`=`) operators over Alloy expressions, and are combined using the usual logical connectives as well as universal (`all`) and existential (`some`) quantifiers. Alloy expressions evaluate to relations. Sets are unary relations and scalars are singleton unary relations. The operators `+`, `-`, and `&` denote union, difference, and intersection, respectively. For relations `r` and `s`, relational join (forward composition), Cartesian product, and transpose are denoted by `r.s`, `r -> s`, and `~r`, respectively. The relational override `r++s` contains all tuples in `s`, and any tuples of `r` whose first element is not the first element of a tuple in `s`. The transitive closure `^r` denotes the smallest transitive relation that contains `r`, and `*r` denotes the reflexive transitive closure of `r`. The expressions `s<:r` and `r:>s` give domain and range restriction of `r` to `s`, respectively.

We assume that pre- and post-conditions are annotations marked as `requires` and `ensures` clauses, respectively. Assume that the specifications of `prepend` read as follows:

```
1 /*@ requires true;
2   @ ensures self.head'.*next'.data' = self.head.*next.data + d;
3   @*/
```

In this case, the method has no pre-conditions, and the post-condition ensures that the set of `Data` objects stored in the receiver list in the post-state (given by the expression `self.head'.*next'.data'`) augments that of the pre-state (given by the expression `self.head.*next.data`) with the prepended data (namely `d`). This example shows that Alloy specifications for linked data structures tend to be concise. It also shows that the specifications can be arbitrarily partial.

9.3 Relational Java Dynamic Logic

JavaDL, the verification logic of KeY, extends typed first-order logic with dynamic logic [44] operators over Java program fragments. Besides propositional connectives and first-order quantifiers, it introduces modal operators. The formula $\{p := t\}\varphi$ in which p is a constant symbol, t is a term whose type is compatible with that of p, and φ is a JavaDL formula, is true iff φ is true after the assignment of t to p —more precisely, $\{p := t\}\varphi$ is true in a state s iff φ is true in the state s' which coincides with s for all symbols but p, for which $s'(p) = s(t)$. The modal operator $\{p := t\}$ is called an *update*. The formula $[\pi]\varphi$ in which π is a sequence of Java statements and φ is a

$$store(h,p,g,v)[o.f] = (\text{if } o = p \wedge f = g \wedge g \neq \langle \texttt{created} \rangle \text{ then } v \text{ else } h[o.f])$$

$$create(h,p)[o.f] = (\text{if } o = p \wedge f = \langle \texttt{created} \rangle \text{ then } \textbf{true} \text{ else } h[o.f])$$

$$anon(h_1,l,h_2)[o.f] = (\text{if } (o,f) \in l \wedge f \neq \langle \texttt{created} \rangle \vee o \in free(h_1) \text{ then } h_2[o.f] \text{ else } h_1[o.f])$$

Figure 9.3: Definitions of heap constructors

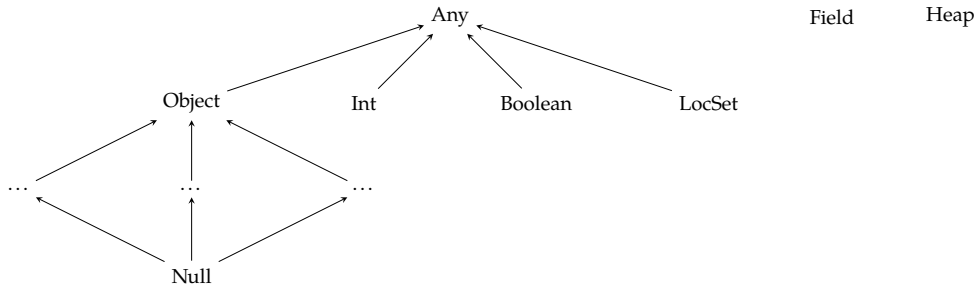


Figure 9.4: Abstract structure of JavaDL type hierarchies

formula, is true in a state s iff φ is true in the post-state (if any exists) of the program π. The formula $\langle \pi \rangle \varphi$ additionally requires π to terminate.[2]

JavaDL is based on an *explicit heap model* [77]: a dedicated program variable *heap* of type *Heap* stores the current heap state. The heap stores are modeled based on a modified versions of McCarthy's theory of arrays [57]. A read access `o.f` in Java is encoded as the heap term $select(heap, o, \texttt{f})$, abbreviated as $heap[o.f]$. Heap modifications are modeled using *heap constructors*, as defined in Fig. 9.3. The *store* function is used to encode changes to a field other than $\langle \texttt{created} \rangle$. The boolean field $\langle \texttt{created} \rangle$ is implicitly added to the class `Object` to distinguish between created and uncreated objects. A Java assignment of a variable v to a field `f` of a non-null object o can be interpreted as an update:

$$[\texttt{o.f} = \texttt{v;}]\varphi \;\leftrightarrow\; \{heap := store(heap,\ o,\ f,\ v)\}\varphi \tag{9.1}$$

The *create* function is used to set the $\langle \texttt{created} \rangle$ field of an object to **true**. The *anonymizing* function *anon* modifies a set of locations rather than a single location. It is used to summarize the effects on the heap made by code in loops or method invocations. The heap denoted by the term $anon(h_1, l, h_2)$ coincides with h_2 (the anonymous heap) in all fresh locations and those in the location set l, and coincides with h_1 (the base heap) on the remaining ones.

[2] $[\pi]\varphi$ and $\langle \pi \rangle \varphi$ correspond to $wlp(\pi, \varphi)$ and $wp(\pi, \varphi)$ in the wp-calculus [23].

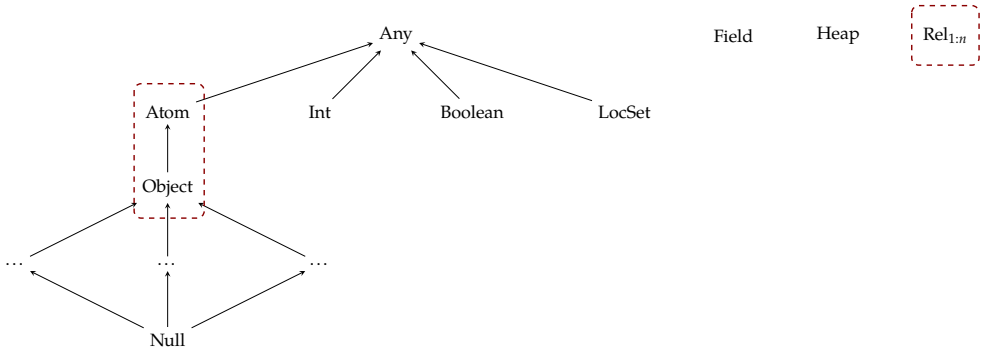


Figure 9.5: Abstract structure of relational JavaDL type hierarchies

JavaDL's type system is shown in Figure 9.5. It includes the hierarchy of Java reference types, with the root type *Object* which denotes an infinite set of objects (including the `null` object), whether or not created. The expression

$$free(h) = \{o : Object \mid \neg h[o.\langle \texttt{created} \rangle] \wedge o \neq \texttt{null}\}$$

gives the set of all uncreated objects different than `null` in the heap h. The types *Boolean* and *Integer* have their usual meanings, the type *Field* consists of all Java fields declared in the verified program, and *LocSet* consists of sets of locations, which are binary relations between *Object* and *Field*. As shown in Figure 9.5, all types except of *Field* and *Heap* are subtypes of the type *Any*. For a type T, the type predicate $x \sqsubseteq T$ evaluates to true iff x is of type T.

KeY performs *symbolic execution* [51] of the given Java code. The effects of this execution on the program state are recorded as JavaDL updates. The equivalence (9.1), for instance, is used to encode the effect of the Java assignment `o.f=v`. Similar equivalences are used for all other Java statements. Branching statements (conditional or looping) cause the proof obligation to split into cases; corresponding path conditions are assumed in each case. Consequently, symbolic execution resolves the original proof obligation

$$pre \rightarrow [p]post$$

of a program p, that given the precondition *pre* it ensures the postcondition *post*, into a conjunction of formulas of the form

$$pre \wedge path_i \rightarrow \{\mathcal{U}_i\}post$$

, in which $path_i$ stands for the accumulated path condition of an execution path in p, and $\mathcal{U}_i$ for the accumulated state updates in that path.

In order to support deductive reasoning for Alloy problems in KeY, relational JavaDL, a relational extension of JavaDL, was developed (cf. [75, 74, 40]). Like our relational first-order logic RFOL, this extension included new (toplevel) JavaDL types,

namely *Atom* for elements of relations, and a Rel_i type for all i-ary relations for each $1 \leq i \leq N$, where N is the maximal needed relational arity. New function symbols for Alloy operators were introduced and defined, however, using deductive rules, instead of axioms like in RFOL. For our proposes of using Alloy as specification logic for Java programs, we additionally make the type *Atom* as supertype of the *Object* and subtype of *Any*. This allows Java objects to be directly elements of relations without the need of mapping bijections, and thus simplifies intermediate expressions. The type system of relational JavaDL is shown in Figure 9.5 —differences to JavaDL type system are circled. We use the same symbols as in RFOL to denote the symbols in relational JavaDL that correspond to the Alloy operators, i.e., we use $\cup$, $\setminus$, $\oplus$, $\times$, $\lhd$, $\mathbin{.}$, *, $^+$ (ascending precedence order) for the Alloy operators `+` , `-` ,`++` , $\rightarrow$, `<:` , `.`, `*` , `^`, respectively. Also here, we skip the arity notations when it is clear for context.

9.4 Coupling Axioms

Although the embedding of Alloy into relational JavaDL is just perfect for deductive reasoning over Alloy problems in KeY, it is not sufficient for verifying Java programs against Alloy specifications as it lacks a model of program state. To encode a relational view of the heap, we translate relations for Java classes and fields as heap-dependent function symbols and fix their semantics with so called coupling axioms. A Java class `C` is translated to a function symbol $C_{rel} : Heap \rightarrow Rel_1$ such that the expression $C_{rel}(h)$ gives the set of all created objects of type C in the heap h, as given by the first coupling axiom:

$$C_{rel}(h) := \{o \mid h[o.\langle \texttt{created} \rangle] \wedge o \sqsubseteq C \wedge o \neq \texttt{null}\}. \tag{9.2}$$

It should be noted that first the embedding of *Atom* a supertype of *Object* and subtype of *Any* (cf. Figure 9.5), allows Java objects to be directly elements of relations as in Axiom 9.2. A Java field `f` of type `R` declared in a class `C` is translated to a function symbol $f_{rel} : Heap \rightarrow Rel_2$ where $f_{rel}(h)$ gives the set of all pairs (o_1, o_2) such that, in heap h, the created object o_1 points to the object o_2 via `f`, as given by the second coupling axiom:

$$f_{rel}(h) := \{(o_1, o_2) \mid o_1 \in C_{rel}(h) \wedge (o_2 = \texttt{null} \vee o_2 \in R_{rel}(h)) \wedge o_2 = h[o_1.\texttt{f}]\}. \tag{9.3}$$

Following the design-by-contract paradigm, Alloy specifications (for Java programs) can access only the pre- and post-state. Thus we provide two sets of relations (unprimed for pre- and primed for post-state) instead of introducing an explicit notion of state. Heap arguments are introduced when Alloy specifications are translated into relational JavaDL: references to C and f are translated to $C_{rel}(preheap)$ and $f_{rel}(preheap)$, respectively, referring to the heap in the pre-state; references to C′ and f′ are translated to $C_{rel}(postheap)$ and $f_{rel}(postheap)$, referring to the heap in the post-state. Null

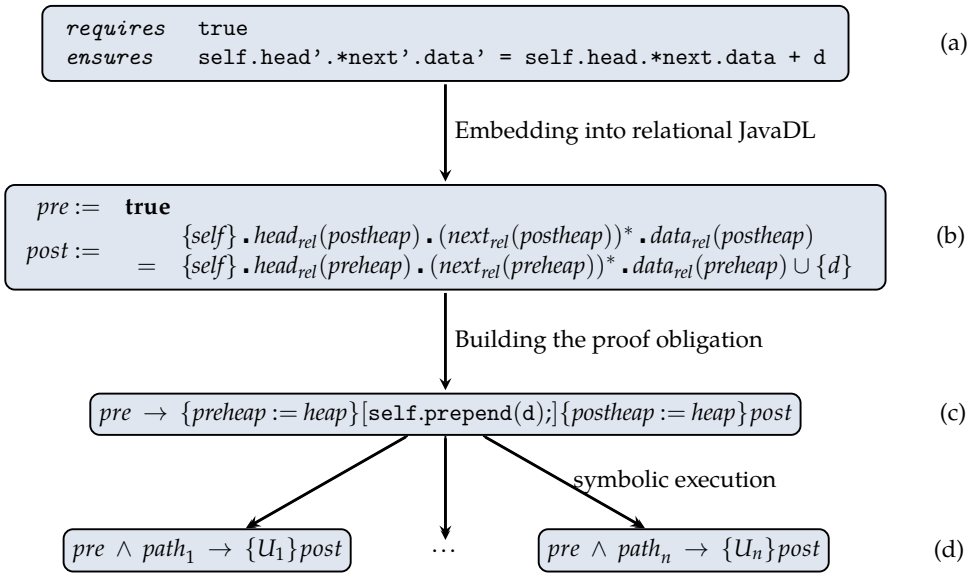


Figure 9.6: The verification process for the method List.prepend as running example

signature (see the example in Figure 9.2) is translated as $Null_{\text{rel}}(h) := \{\texttt{null}\}$ for every heap h.

Figure 9.6 shows how JKelloy processes the example of Figure 9.2. Figure 9.6(a) gives the original Alloy specification, Fig. 9.6(b) gives its translation into our relational JavaDL, and Fig. 9.6(c) the relational JavaDL proof obligation for the method `List.prepend`. In addition to the program modality $[\texttt{self.prepend(d);}]$, two updates $\{\mathit{preheap} := \mathit{heap}\}$ and $\{\mathit{postheap} := \mathit{heap}\}$ are used to store the respective current heap. Symbolic execution then resolves the code of the method. Several formulas as shown in Fig. 9.6(d) are produced for the various execution paths of the code. The example is continued in Section 9.5.

The above coupling axioms are defined such that they preserve the meaning of the Alloy relations used in the specifications. For instance, relation `head'` in the example of Fig. 9.2 is a total binary relation containing the references from all created `List` objects to `Entry` objects (or `null`) after the method call. Axiom (9.3) ensures that $\mathit{head}_{\mathit{rel}}(\mathit{postheap})$ contains precisely those elements.

9.5 Calculus

The coupling axioms (9.2) and (9.3) fix the semantics of the relation function symbols. Together with relational JavaDL's relational calculus, they suffice to conduct proofs for Java programs against Alloy specification. In practice, however, verification using these low-level axioms alone is inefficient since it requires to always expand the

definitions of the relations. In order to both lift proofs to the higher abstraction level of relations (i.e. without expanding their definitions) and to automate them, we introduce two sets of rules described in the following subsections.

9.5.1 Relational Heap Resolution Calculus

Figure 9.7 lists the rules for the relational resolving of heap constructor occurrences as argument of field relations (R_1–R_3) and class relations (R_4–R_6). All rules reduce relational expressions over composed heaps to expressions over their heap argument —constant heaps. They are applied to the verification conditions after symbolic execution and eliminate all heap constructors from arguments of relational function symbols. Rules R_1, R_2 and R_5, for instance, make case distinctions between the cases when the relation needs to be updated and when it remains untouched. R_3 is special since it updates a set of elements and not only one element in the relation. The heap resolution rules are program specific rules. They are thus automatically generated, by the translation front-end, for each class and field of the considered program. All rules (schemas) were proved correct with respect to the coupling axioms (see Section 9.4) using KeY.

We explain the idea of heap resolution using the example in Figure 9.6. The update U in the formulas of Figure 9.6(d) encodes the successive heap modifications performed by the program, during an execution path. After some simplifications, the heap modification of the method body is encoded as

$$\begin{array}{rl} & \overset{h_5}{\downarrow}\ \overset{h_4}{\downarrow}\ \overset{h_3}{\downarrow}\ \overset{h_2}{\downarrow}\ \overset{h_1}{\downarrow} \\ \mathit{postheap} := & \mathit{store}\ (\ \mathit{store}\ (\ \mathit{store}\ (\ \mathit{create}\ (\ \mathit{preheap}\ ,e), \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{self},\texttt{head},e), \\ & \qquad\qquad\qquad\qquad\quad e,\texttt{next},\mathit{preheap}[\mathit{self}.\texttt{head}]), \\ & \qquad\quad e,\texttt{data},d) \end{array}$$

where $h_1,\ldots,h_5$ are abbreviations for the intermediate heap expressions and e is a reference to the freshly created `Entry` object. In Figure 9.6(b), some of the field relations take *postheap* as argument (like $\mathit{head}_{\mathit{rel}}(\mathit{postheap})$) in which, under the influence of U, *postheap* is replaced by the nested term h_5. Heap modifications in h_5 affect the value of $\mathit{head}_{\mathit{rel}}(h_5)$ only if they are related to the field `head`. Rule R_1, which is responsible for the resolution of this term, translates the *store* expression into an if-then-else term resulting either in an overridden relation ($f_{\mathit{rel}}(h) \oplus \{o_1\} \times \{o_2\}$) or in the original relation $f_{\mathit{rel}}(h)$. The relations $\mathit{head}_{\mathit{rel}}(h_5)$, $\mathit{head}_{\mathit{rel}}(h_4)$ and $\mathit{head}_{\mathit{rel}}(h_3)$, for instance, are equivalent as the modified Java fields `data` and `next` are different from `head`. But the *store* expression of h_3 modifies the field `head`; hence, the relation $\mathit{head}_{\mathit{rel}}$ must be updated for the arguments of *store* and we obtain:

$$\mathbf{R_1}\text{:}\quad f_{rel}(store(h,o_1,g,o_2)) \rightsquigarrow \begin{array}{ll}\text{if} & g = \quad f \wedge h[o_1.created] \wedge o_1 \sqsubseteq C \wedge o_1 \neq \texttt{null} \\ \text{then} & f_{rel}(h) \oplus \{o_1\} \times \{o_2\} \\ \text{else} & f_{rel}(h)\end{array}$$

assuming $wellformed(store(h,o_1,g,o_2))$

$$\mathbf{R_2}\text{:}\quad f_{rel}(create(h,o)) \rightsquigarrow \begin{array}{ll}\text{if} & o \neq \texttt{null} \wedge o \sqsubseteq C \\ \text{then} & f_{rel}(h) \oplus \{o\} \times \{h[o.\texttt{f}]\} \\ \text{else} & f_{rel}(h)\end{array}$$

$$\mathbf{R_3}\text{:}\quad f_{rel}(anon(h_1,l,h_2)) \rightsquigarrow f_{rel}(h_1) \oplus (((l \,.\, \{\texttt{f}\}) \cup free(h_1)) \lhd f_{rel}(h_2))$$

$$\mathbf{R_4}\text{:}\quad C_{rel}(store(h,o_1,g,o_2)) \rightsquigarrow C_{rel}(h)$$

$$\mathbf{R_5}\text{:}\quad C_{rel}(create(h,o)) \rightsquigarrow \begin{array}{ll}\text{if} & o \neq \texttt{null} \wedge o \sqsubseteq C \\ \text{then} & C_{rel}(h) \cup \{o\} \\ \text{else} & C_{rel}(h)\end{array}$$

$$\mathbf{R_6}\text{:}\quad C_{rel}(anon(h_1,ls,h_2)) \rightsquigarrow C_{rel}(h_1) \cup C_{rel}(h_2)$$

Figure 9.7: Heap Resolution Calculus. The term rewrite relation "$\rightsquigarrow$" represents an equivalence transformation. In R_1 and R_2 the field `f` is defined in class C.

$$\begin{aligned} & head_{rel}(h_5) = head_{rel}(store(h_4,e,\texttt{data},d)) \\ \overset{R_1}{=}\ & head_{rel}(h_4) = head_{rel}(store(h_3,e,\texttt{next},preheap[self.\texttt{head}])) \\ \overset{R_1}{=}\ & head_{rel}(h_3) = head_{rel}(store(h_2,self,\texttt{head},e)) \\ \overset{R_1}{=}\ & head_{rel}(h_2) \oplus \{self\} \times \{e\}\ . \end{aligned} \tag{9.4}$$

Relation $head_{rel}(h_2)$ is finally simplified to $head_{rel}(h_1)$ by rule R_2 since the creation of the `Entry` element e does not affect the relation $head_{rel}$ for the field `head` declared in class `List`. Equation (9.4) shows the main idea of the heap resolution calculus: the heap state changes are transformed into relational operations. In particular, an assignment `o1.f=o2` in Java resolves into a relational override of the form $f_{rel}(heap) \oplus \{o_1\} \times \{o_2\}$.

Rule R_1 has been used three time in the above example. Twice (for h_5 and h_4) the condition $g = \quad f$ was false and the else branch had been taken. For h_3, the 4 parts of the condition were all true and the then-branch has been taken. For the soundness of R_1 with respect to the coupling axioms, it is required that the heap argument is

wellformed, i.e., all locations point to a created object of their declared type. If this formula is not present in the verification condition, it is automatically introduced as a lemma by the strategy. When creating a new object o of class C in heap h and the field `f` is defined in C, rule R_2 extends the relation $f_{rel}(h)$ with a tuple for defining the value of $o.f_{rel}(h)$. Since the object o did not exist before, no such tuple was present in f_{rel}. Rule R_3 handles the *anon* constructor, where the memory locations l are assigned new values from h_2. Hence, the rule overrides the relation $f_{rel}(h_1)$ with tuples from $f_{rel}(h_2)$, but restricts this override to the relevant tuples. These are the locations in l that belong to field `f` (i.e., $l.\{\texttt{f}\}$[3]) and the not yet created objects ($free(h_1)$). The selection is done using the domain restriction operator $\lhd$. Rule R_4 is relatively simple since the constructor *store* cannot modify the $\langle\texttt{created}\rangle$ field, the set of created objects cannot be enlarged. Rule R_5 extends the relation C_{rel} when a new object of class `C` is created. Rule R_6 considers the possibility that *anon* introduces new objects, of any class, from h_2. The set of created objects in the *anon* heap contains all created objects of the first heap argument (objects can never be deallocated) and all objects created in the second (which have been introduced in the anonymization).

Applying these rules exhaustively leads to a normal form where all heap arguments are constants. JKelloy extends KeY with a proving strategy that always achieves this task automatically. The final result of applying the heap resolution rules to the post-condition of the running example (Fig. 9.6(b)) is the following relational verification condition:

$$\begin{aligned} & \{self\}.(head_{rel}(h_1) \oplus \{self\}\times\{e\}) \\ & \quad .(next_{rel}(h_1) \oplus \{e\}\times\{self\}.head_{rel}(h_1))^*.(data_{rel}(h_1) \oplus \{e\}\times\{d\}) \\ = \quad & \{self\}.head_{rel}(h_1).next_{rel}(h_1)^*.data_{rel}(h_1) \cup \{d\} \end{aligned} \tag{9.5}$$

After all heap terms have been resolved, further reasoning can proceed on the relational level.

9.5.2 Override Simplification Calculus

The normalized proof obligations that result from applying heap resolution rules can be proved on the relational level using the relational calculus of our relational JavaDL. However, this relational calculus only provides definition axioms for relational operators and a set of lemmas for general relational expressions. To make proofs easier and to increase the automation level, we introduce a set of lemma rules which exploit the shape of the relational expressions that result from verifying Java programs. These lemmas do not increase the power of the calculus but ease the verification by reducing the need for expanding the definitions of relational operators. That is particularly costly for the transitive closure as it leads to quantified integer formulas that generally require user interaction in form of manual induction. Out of more than 220 new lemmas we have introduced, we present in this section the subset that is most relevant

[3] To unify the presentation we apply relational operators also to type *LocSet*.

$\mathbf{R_7}$: $\{a\} \,.\, (R \oplus \{a\} \times \{b\}) \rightsquigarrow \{b\}$

$\mathbf{R_8}$: $S_1 \,.\, (R \oplus S_2 \times S_3) \rightsquigarrow$ if $S_2 = \emptyset$ then $S_1 \,.\, R$ else $S_1 \,.\, (S_2 \times S_3) \cup (S_1 \setminus S_2) \,.\, R$

$\mathbf{R_9}$: $S_1 \,.\, (R \oplus \{a\} \times S_2)^+ \rightsquigarrow$ if $S_2 = \emptyset \vee a \notin S_1$ then $S_1 \,.\, R$ else $S_2 \cup ((S_1 \setminus \{a\}) \,.\, R^+) \cup (S_2 \,.\, R^+)$
assuming $R \,.\, \{a\} = \emptyset$

$\mathbf{R_{10}}$: $\{a\} \,.\, (R \oplus \{b\} \times \{c\})^+ \rightsquigarrow$ if $b \in \{c\}.R^+ \vee b = c$ then $(\{a\} \,.\, R^+ \cup \{c\} \cup \{c\} \,.\, R^+) \setminus \{b\} \,.\, R^+$ else $(\{a\} \,.\, R^+ \setminus \{b\} \,.\, R^+) \cup \{c\} \cup \{c\} \,.\, R^+$
assuming $b \in a \,.\, R^+$, $parFun(R)$ and $acyc(R)$

$\mathbf{R_{11}}$: $\{a\} \,.\, (R \oplus \{b\} \times \{c\})^+ \rightsquigarrow$ if $b \neq a$ then $\{a\} \,.\, R^+$ elseif $a \in \{c\} \,.\, R^+ \vee c = a$ then $(\{c\} \cup \{c\} \,.\, R^+) \setminus \{a\} \,.\, R^+$ else $\{c\} \cup \{c\} \,.\, R^+$
assuming $b \notin a \,.\, R^+$ and $parFun(R)$

$\mathbf{R_{12}}$: $S_1 \,.\, f_{rel}(h) \,.\, (R_1 \oplus S_2 \lhd R_2) \rightsquigarrow S_1 \,.\, f_{rel}(h) \,.\, R_1$ assuming $S_2 \subseteq free(h)$

$\mathbf{R_{13}}$: $S_1 \,.\, f_{rel}(h) \,.\, (g_{rel}(h) \oplus S_2 \lhd R)^+ \rightsquigarrow S_1 \,.\, f_{rel}(h) \,.\, g_{rel}(h)^+$ assuming $S_2 \subseteq free(h)$

$\mathbf{R_{14}}$: $(f_{rel}(h) \oplus S_2 \lhd R)^+ \rightsquigarrow f_{rel}(h)^+ \oplus S_2 \lhd R$ assuming $S_2 \subseteq free(h)$

Figure 9.8: A sampling of our override driven calculus rules

to the examples of Figure 9.2 and Section 9.6; not all of them are used in the presented examples. All lemmas have been proved correct using KeY.

Equation (9.5) is typical for our approach: its right-hand side (RHS) refers to the base relations of the pre-state, whereas its left-hand side (LHS) refers to the post-state and thus includes override-updates on the field relations. To prove such formulas, we bring the LHS closer to the shape of the RHS by resolving or pulling out the override operations that occur below other operators such as join and transitive closure.

Figure 9.8 lists a number of lemmas dealing with this override resolution to give an idea of the process. The most simple case is R_7 which says that retrieving the *image* of an atom a from a relation which has been overridden at the very same a results precisely in the updated atom b. In other, more composed cases, the resolution is not as simple. Rules R_9, R_{10} and R_{11}, e.g., allow us to resolve the override beneath a transitive-closure operation under certain conditions at the cost of larger replacement

expressions without override. Rules R_{12}–R_{14} resolve override operations which only modify objects not yet created in the base heap ($S_2 \subseteq \mathit{free}(h)$).

In our example, the subexpression $\{\mathit{self}\} \,.\, (\mathit{head}_{\mathit{rel}}(h_1) \oplus \{\mathit{self}\} \times \{e\})$ in (9.5) can be simplified to $\{e\}$ using R_7 as the left argument $\{\mathit{self}\}$ of the join equals the domain of the overriding relation $\{\mathit{self}\} \times \{e\}$. After this simplification, the LHS contains the subexpression

$$\{e\} \,.\, (\mathit{next}_{\mathit{rel}}(h_1) \oplus \{e\} \times \{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1))^* \,. \tag{9.6}$$

To resolve the override operation in this expression, we first transform reflexive transitive closure to transitive closure using the equality $S.R^* = S \cup S.R^+$, and then apply rule R_9. The assumption of R_9 holds because e is not yet created in h_1, and the if-condition evaluates to **false**. Altogether the subterm 9.6 of the relational verification condition of our running example (Equation 9.5) result in:

$$\begin{aligned}
& \{e\} \,.\, (\mathit{next}_{\mathit{rel}}(h_1) \oplus \{e\} \times \{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1))^* \\
\overset{R^*\ def}{\rightsquigarrow}\ & \{e\} \cup \{e\} \,.\, (\mathit{next}_{\mathit{rel}}(h_1) \oplus \{e\} \times \{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1))^+ \\
\overset{R_9}{\rightsquigarrow}\ & \{e\} \cup \{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1) \cup ((\{e\} \setminus \{e\}) \,.\, \mathit{next}_{\mathit{rel}}(h_1)^+) \cup \\
& (\{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1) \,.\, \mathit{next}_{\mathit{rel}}(h_1)^+) \\
\overset{\emptyset\ def}{\rightsquigarrow}\ & \{e\} \cup \{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1) \cup \{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1) \,.\, \mathit{next}_{\mathit{rel}}(h_1)^+ \\
\overset{R^*\ def}{\rightsquigarrow}\ & \{e\} \cup \underline{\{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1) \,.\, \mathit{next}_{\mathit{rel}}(h_1)^*}
\end{aligned} \tag{9.7}$$

The underlined subexpression above appears also on the RHS of (9.5). We have thus reached our goal of resolving the override and bringing the LHS closer to the RHS, to a larger extent. Based on this result, the proof can be easely concluded using R_7 together with other general relational rules of our calculus as the distributive properties of join over union.

The rules of Figure 9.8 work as follows. Rule R_8 is a generalization of R_7 (explained above) where an arbitrary relation is joined with an overridden expression. Pulling override out of a transitive closure operation is particularly important due to the complexity of verifying transitive closure. We introduce a number of rules for various forms of such expressions. Rule R_9, for instance, is applicable when the singleton in the domain of the overriding relation (a) is not in the range of the overridden relation (R)[4]. Rule R_{10}, which is one of the most general cases that our calculus can handle, is applicable under three assumptions: (1) the first element of the overriding pair (b) must be reachable from the joining singleton (a) via R, (2) the overridden relation (R) must be a partial function, (3) R must be be acyclic. Rule R_{11} complements R_{10}. It handles the case where the first element of the overriding pair (b) is *not* reachable

[4]Such information is inferred from the path condition of the proof obligation.

$\mathbf{R_{15}}$: $\vdash parFun(f_{rel}(h))$
$\mathbf{R_{16}}$: $\vdash parFun(R) \rightarrow parFun(R \oplus \{a\} \times \{b\})$
$\mathbf{R_{17}}$: $\vdash parFun(R_1) \wedge parFun(R_2) \rightarrow parFun(R_1 \oplus R_2)$

$\mathbf{R_{18}}$: $\vdash acyc(R) \wedge R \,.\, \{a\} = \emptyset \wedge a \neq b \rightarrow acyc(R \oplus \{a\} \times \{b\})$
$\mathbf{R_{19}}$: $\vdash acyc(R) \wedge \{b\} \,.\, R = \emptyset \wedge a \neq b \rightarrow acyc(R \oplus \{a\} \times \{b\})$
$\mathbf{R_{20}}$: $\vdash acyc(R) \wedge a \notin \{b\} \,.\, R^+ \wedge a \neq b \rightarrow acyc(R \oplus \{a\} \times \{b\})$

$\mathbf{R_{21}}$: $S_2 \in S_1 \,.\, R^+ \rightsquigarrow \mathbf{false}$ assuming $S_1 \,.\, R = \emptyset$
$\mathbf{R_{22}}$: $\{a\} \in R \,.\, \{b\} \rightsquigarrow \mathbf{true}$ assuming $\{a\} \,.\, R = \{\}b$
$\mathbf{R_{23}}$: $\{a\} \in R \,.\, \{b\} \rightsquigarrow \mathbf{false}$ assuming $parFun(R)$ and $\{a\} \,.\, R \neq \{\}b$

Figure 9.9: A selection of auxiliary rules for the override simplification

from the joining singleton (a) via R. In this case, our rule is more general than the previous case since it does not require acyclicity of the overridden relation. Rules R_{12}–R_{14} resolve override operations which only modify the uncreated objects in the base heap ($S_2 \subseteq free(h)$). Such cases arise when occurrences of the *anon* constructor are resolved by applying rule R_3.

The rules in Figure 9.8 focus on resolving override operations, yet further rules are required to reason about expressions that occur in the rules' assumptions, if-conditions, and results. Figure 9.9 shows such rules divided into three categories. The first involves partial functionality of relations: every relation corresponding to a field is a partial function by construction (R_{15}); R_{16} and R_{17} allow the propagation of this property over the override operator. Similarly, the second propagates the acyclicity of relations over the override operator, the rules R_{18}–R_{20} show some examples. The last category lists some rules for handling reachability between objects effectively.

The general shape of the Alloy expressions which our override-driven calculus can effectively simplify (i.e., their override operators can be pulled out to the top) is described in the grammar of Figure 9.10. The fragment has three main restrictions:

1. override expressions are built by successively applying one or more override operations to field relations in which every overriding relation contains at most one element,

2. (reflexive) transitive closure operator can be applied to override expressions only if the left subexpression of the outermost override is acyclic,

3. override expressions may be joined from left only with a set that has at most one element.

The heap resolution rules resolve assignments to heap locations into override expressions for which the first restriction holds. These expressions have exactly the same form as $overE_1$ in Figure 9.10. Anonymised heaps (using *anon*) do not belong to

this fragment. The second and third restrictions are important in order to develop efficient resolution rules for override. Acyclicity is a property that is often assumed in linked data structures. Using the calculus rules, e.g. R_{18}–R_{20}, the acyclity of override expressions can be deduced from the acyclity of their base field relations. Specifications adhere to the third restrictions if they denote sets of objects reachable from a particular starting point (like the post-condition in Figure 9.6(a) for instance). However, this last restriction does not hold for general relations which may also appear in specifications. The weakening of the last restriction is left for future work.

$$
\begin{aligned}
&\mathit{expr} ::= \mathbf{self} \mid \mathbf{var} \mid \mathbf{none} \mid \mathbf{univ} \mid \mathbf{iden} \mid \mathbf{C_{rel}}(\mathbf{h}) \mid \mathbf{f_{rel}}(\mathbf{h}) \mid \mathit{overFreeE}(^{\sim} \mid ^{+} \mid ^{*}) \\
&\quad \mid \mathit{overFreeE}\ \mathit{binOp}\ \mathit{overFreeE} \mid \mathit{overE}\ \mathit{binOp}\ \mathit{overFreeE} \mid \mathit{overFreeE}\ \mathit{binOp}\ \mathit{overE} \\
&\mathit{overFreeE} ::= \texttt{any override free expression} \\
&\mathit{overE} ::= \mathit{loneS} \mathbin{.} \mathit{overE}_1 \mid \mathit{loneS} \mathbin{.} \mathit{overE}_2 \mid \mathit{loneS} \mathbin{.} (\mathit{overE} \cup \mathit{overE}) \\
&\mathit{overE}_1 ::= \mathbf{f_{rel}}(\mathbf{h}) \oplus \mathit{loneS} \times \mathit{loneS} \mid \mathit{overE}_1 \oplus \mathit{loneS} \times \mathit{loneS} \\
&\mathit{overE}_2 ::= (\mathit{overE}_1 \oplus \mathit{loneS} \times \mathit{loneS})(^{+} \mid ^{*})\ \texttt{where acyc(}\mathit{overE}_1\texttt{)} \\
&\mathit{loneS} ::= \mathbf{none} \mid \mathbf{\{self\}} \mid \mathbf{\{var\}} \mid \mathit{loneS} \mathbin{.} \mathbf{f_{rel}}(\mathbf{h}) \\
&\mathit{binOp} ::= \cup \mid \cap \mid \setminus \mid \times \mid \lhd \mid \rhd \mid .
\end{aligned}
$$

Figure 9.10: Target fragment of the override driven calculus

9.6 Evaluation

Proofs in KeY are conducted by applying calculus rules either manually or automatically, using KeY's proof search strategy. We extend the existing strategy by incorporating two new strategies that assign priorities to heap resolution rules and override simplification rules[5] , and apply them consecutively. In this section we report on the use of our tool to verify Java programs against Alloy specifications.

The `List.prepend` example[6] verifies fully automatically within 5.4 seconds[7] using 1546 rule applications although its post-condition involves transitive closure.

We have also verified a slightly different example (`List.append`) where the `Data` argument is added to the end of the list. The proof contains a total of 2850 rule applications out of which 28 are interactive. These include 6 applications of proof-branching rules, and 6 rule applications to establish the assumptions for rule R_{10}. Automatic rule applications take 20.3 seconds. The `append` method is more complex than `prepend` as it contains a loop that traverses the list to the end, thus requires handling loop invariants. The proof requires the more complex transitive closure rule R_{10} since the code updates already-created objects.

For a more rigorous evaluation of the JKelloy capability of verifying programs which manipulate rich heap data structures we have verified the example of Figure 9.11.

[5]Rules with looping potential (e.g., R_{15}-R_{20}) and branching rules are excluded from the strategy.

[6]All examples and proofs can be found at `http://i12www.ira.uka.de/~elghazi/jkelloy/`

[7]On an Intel Core2Quad, 2.8GHz with 8GB memory

```
1  public class Graph {
2   NodeList nodes;
3   /*@ requires acyc(next);
4     @ requires not n = null;
5     @ ensures self.nodes'.first'.*next' = self.nodes.first.*next - n;
6     @ ensures Object <: left' = left++ ((left.n & self.nodes.first.*next) → null);
7     @ ensures Object <: right' = right++ ((right.n & self.nodes.first.*next) → null);
8     @*/
9   void remove(Node n) {
10   if (nodes != null) {
11    Node curr = nodes.first;
12    /*@ loop_invariant
13      @ curr in self.nodes.first.*next and
14      @ Object<:left' =
15      @   left++ ((left.n & (self.nodes.first.*next - curr.*next)) → null) and
16      @ Object<:right' =
17      @   right++ ((right.n & (self.nodes.first.*next - curr.*next)) → null)
18      @ assignable
19      @ (self.nodes.first.*next → left) + (self.nodes.first.*next → right);
20      @*/
21    while (curr != null) {
22     if (curr.left == n)  { curr.left = null; }
23     if (curr.right == n) { curr.right = null; }
24     curr = curr.next;
25    }
26    nodes.remove(n);
27  } } }
28  class NodeList { Node first;  void remove(Node n) { . . . } }
29  class Node {  Node next, left, right; }
```

Figure 9.11: Specification and implementation of the graph remove example

This example also illustrates that structurally complex specifications can be concisely expressed by exploiting combinations of relational operators in Alloy. The `Graph` class implements a binary graph[8] where each node stores its two (possibly `null`) successors (`left` and `right`, Line 29). The graph keeps a linked list of its nodes (Line 2) using the `next` field (Line 29). The method `Graph.remove` removes a node `n` from the receiver graph by removing all of its incoming edges (Lines 21–25), and then removing `n` (and thus its outgoing edges) from the node list (Line 26).

The method requires the node list to be acyclic (Line 3) and the argument node `n` to be non-null (Line 4). It ensures that `n` is removed from the graph's node list (Line 5), and that the `left` and `right` fields of all nodes in this list that used to point to `n`, point to `null` at the end of the method (Lines 6 and 7). This example also illustrates that structurally complex specifications can be concisely expressed by exploiting

[8] A directed graph with an outgoing degree of at most two for every node

combinations of relational operators in Alloy. In particular, sets of nodes with a particular property can be easily expressed using Alloy operators. For example, using the join operator from the right side of a field relation, the expression left.n concisely gives the set of all nodes whose left field points to `n`. The domain restriction to Object restricts the relation in the post-state to those objects already existing in the pre-state. The relational override operator denotes exactly what locations are modified and how, thus also implicitly specifies which locations do not change.

The example requires additional intermediate specifications which are not part of the contract. This includes a *loop specification* for the method loop (Lines 12–20) describing the state after the execution up to the current loop iteration. Primed relations in the loop invariant refer to the state of the heap after the current loop iteration, whereas unprimed relations refer to the pre-state of the method. The assignable clause specifies the set of heap locations which may be modified by the loop. `Graph.remove` calls `NodeList.remove` which removes `n` from the linked list; the call is abstracted by the callee's contract which is omitted here for the sake of simplicity.

Though the specification in the example is concise, it extensively combines relational operators including, in particular, transitive closure. In the code, the nested method call and the loop result in complex composed heap expressions after symbolic execution. Brought together, these two technical points make this example difficult to verify. The proof required 6973 rule applications distributed over 157 subgoals, where 1201 of the rule applications were interactive. Amongst them, 116 were heap resolution rules, 309 apply override simplification rules and 224 general relational rules. Our rules for handling transitive closure proved to be very effective; they were applied 43 times, and allowed us to conduct the proof without any explicit induction. Relational operations were never expanded to their definitions. Thus the proof was completely conducted in the abstraction level of relations. The rules introduced with JKelloy made up 37% of all rule applications; the rest were default KeY rules. The whole proof, including specification adjustments, was conducted by an Alloy and KeY expert in one week; the total time spent by the automatic rule applications was 6.3 minutes. Other comparable examples in KeY (using the JML specification language) require $50k$ to $100k$ proof steps (see, for instance, [42]).

9.7 Related Work

In this section we describe works that used the Alloy language as a specification language to specify programs. We also describe approaches that bear resemblance to ours in the specification and verification of heap data structures.

Several approaches (e.g. [76, 25, 71]) support Alloy as a specification language for Java programs. To check the specifications, however, they bound the analysis domain by unrolling loops and limiting the number of elements of each type. Thus although they find non-spurious counterexamples automatically, they cannot, in general, provide correctness proofs. The JForge specification language [78] is another

lightweight language for specifying object-oriented programs. It is a behavioral interface specification language like JML, but uses a relational view of the heap, that allows some Alloy operators. So far it has been used only for bounded program checking.

Galeotti [38] introduced a bounded, automatic technique for the SAT-based analysis of JML-annotated Java sequential programs dealing with linked data structures. The annotations are translated to SAT using Alloy as an intermediate language. It incorporates (i) DynAlloy [4], an extension of Alloy to better describe dynamic properties of systems using actions, in the style of dynamic logic; (ii) DynJML, an intermediate object-oriented specification language; and (iii) TACO, a prototype tool which implements the entire tool-chain.

A few approaches [75, 59, 5] support full verification of Alloy models. Since they do not model program states, they cannot be readily applied for verifying code with Alloy specifications. Dynamite [59], for example, extends PVS to prove Alloy assertions and incorporates Alloy Analyzer for checking hypotheses.

Other approaches (e.g. [79, 42, 66]) also verify properties of linked data structure implementations. In contrast to ours, in [79], for example, specifications are written in classical higher-order logic (including set comprehension, λ-expressions, transitive closure, set cardinality) and are verified using Jahob [16] which integrates several provers. A decision procedure based on inference rules for a quantifier-free specification language with transitive closure is presented in [66]. In [42] the focus is to write specifications in JML so that they can be used for both deductive program verification and runtime checking.

Similar to our approach, [52, 56] handle reachability of linked data structures using a first-order axiomatization of transitive closure. Their general idea, however, is to use a specialized induction schema for transitive closure, to provide useful lemmas for common situations. [52] focuses on establishing a relatively complete axiomatization of reachability, whereas [56] focuses on introducing as complete schema lemmas as possible and adding their instantiations to the original formula. The main difficulty of schema rules is to find the right instantiation (analogous to induction hypothesis).

9.8 Conclusion

In this chapter, we have presented an approach for verifying Java programs annotated with Alloy specifications. Alloy operators (for instance, relational join, transitive closure, set comprehension, and set cardinality) let users specify properties of linked data structures concisely. Our tool, JKelloy —built on top of KeY— translates Alloy specifications into relational Java Dynamic Logic and proves them using KeY. It introduces coupling axioms to bridge between specifications and Java states, and two sets of calculus rules and strategies that facilitate interactive and automatic reasoning in relational logic. Verification is done on the level of abstraction of the relational specifications. JKelloy lets relational lemmas be proved ones beforehand, and reused

to gain more automation. Our calculus rules are proved lemmas that exploit the shape of the relational expressions that occur in proof obligations of Java programs.

Although further experiments are needed to better perform the proof strategies and to better evaluate the approach, our examples, especially the *Graph* example, shows that our approach can already handle arbitrary shapes of linked data structures. Furthermore, they illustrate (1) how the liberal combinations of transitive closure and relational operators in Alloy can be exploited for concise specifications and (2) the benefit of our calculi in producing shorter proofs —especially in terms of user interactions. The sizes of proofs are an order of magnitude smaller compared to other similar proofs using standard KeY.

KeY supports the Java Modeling Language (JML), a behavioral specification language for Java. A combination of the specification concepts of JML and Alloy has the potential to bring together the best of both paradigms. Furthermore, the symbolic execution engine of KeY along with our calculus rules can produce relational summaries of Java methods which can be checked for bugs using the Alloy Analyzer before starting a proof attempt. Investigating these ideas is left for future work.

Chapter 10

Conclusion

10.1 Summary

The primary goal of this thesis has been to enable the automatic verification of software systems against relational specifications written in Alloy —a first-order relational language— and to make the verification process more effective. Thereby, the software system description should be supported at the abstract relational level, using Alloy, as well as at the detailed implementation level, using Java. We have addressed in this context two major challenges: (1) the automatic verification of pure relational proof obligations and (2) the verification of proof obligations in which the specification is relational but the software system is described in Java.

In order to address the first challenge, a relational first-order logical framework RFOL has been presented which allows a structure preserving and equisatisfiable formulation of pure Alloy proof obligations. RFOL consists of an extension of first-order logic with relational sorts and operators. The axiomatization of non first-oder relational operators such as transitive closure and set cardinality is based on the integer theory. The reasoning in RFOL is based on satisfiability modulo theories (SMT) solving. The SMT reasoning in RFOL, however, can be resource-intensive and does not always succeed. The main reasons for the SMT solving limitations in RFOL are: (1) the excessive and arbitrary use of quantifiers in Alloy problems as well as in the RFOL axiomatization, (2) the high cardinality constraints on relational sorts deduced by the RFOL axioms ($|Rel_i| = 2^{Atom \times ... \times Atom}$), and (3) the absence of an efficient and effective reasoning for the non first-order relational operators. We addressed these problems respectively by the following further contributions:

- We developed a sufficient ground term sets (SufGT) technique which computes iteratively a set of sufficient ground terms of each universally quantified variable —existentially quantified variables are skolemized. We use this technique preliminary to eliminate —via instantiation— all variables whose computed sufficient ground term sets are finite and thus reduce the complexity of RFOL formulas. In addition to that, we also use this technique to (1) increase the scalability of Alloy's bounded verification —using the Alloy Analyzer— by computing maximal scopes for Alloy signatures; signatures admitting such maximal

scopes have only variables whose computed sufficient ground term sets are finite, and (2) prove the correctness of Alloy assertions via bounded verification if all quantified variables of the problem have finite sufficient ground term sets with respect to our technique.

- We developed an extended semantics blasting technique (SB^+) that eliminates cardinality constraints on the relational sorts induced by the RFOL axioms. Since SB^+ does not preserve satisfiability, in general, we have described *the* logical fragment in which our technique is complete. In addition to the theoretical description of the fragment, we developed practical tests that provide a cost effective testing system for the inclusion of an RFOL formula in the SB^+ fragment. It should be noted, in this context, that all considered Alloy benchmarks in the thesis fit in the SB^+ fragment, even in the most easy to test subfragment.

- We developed a path-invariant based transitive closure reasoning technique (TCPInv) that can show the refutation of RFOL formulas[1] involving transitive closure for which standard SMT solving cannot show refutation. Such formulas can not be refuted via standard SMT solving (without induction) because there exists no finite instantiation of their axioms such that their ground subformulas become refutable. The TCPInv technique bases on a pure first-order *weak* axiomatization of transitive closure and a procedure for the detection of invariant formulas of essential relational paths —transitive closure literals in CNF whose refutation is essential. The fact that the base transitive closure axiomatization is integer free, reduces in general the complexity of RFOL formulas and thus improves their SMT solving.

For the second main challenge of the thesis, JKelloy, a tool for the deductive verification of Java programs against Alloy specifications, has been presented. In order to support and promote the specification of Java programs at the abstract level of relations —using a *relational view of the heap* [76], JKelloy automatically generates a so called *Alloy context* which encodes the relational view of the types and fields of the Java program in the pre- and post-state, following the design-by-contract paradigm [58].

Given a Java program with an Alloy specification written with respect to the Alloy context, JKelloy translates the Alloy specification into our relational Java dynamic logic — a relational extension of KeY's Java dynamic logic [11]. The resulting proof obligation combines two (independent) logics, namely the pure relational logic and the pure JavaDL logic. In order to enable and facilitates the interactive and automatic reasoning for such proof obligations, we developed and implemented in JKelloy the following contributions:

- A set of coupling axioms that automatically define the link between the heap dependant relations in the Alloy specification and the Java program states.

[1] Showing the refutation of a formula F is equivalent to proving the validity of its negation $\neg F$.

- A heap resolution calculus that normalizes relational JavaDL proof obligations over composed heaps to relational expression over their heaps —i.e., only constant heaps remain. The calculus rules are applied to the verification conditions after symbolic execution and eliminate all heap constructors from arguments of relational function symbols. We equipped JKelloy with a rule application strategy that always achieves this task automatically.

- An override simplification calculus that eases the verification process in JKelloy, by reducing the need for expanding the definitions of relational operators, especially of those over override expressions. Override expressions typically result from applying the relational heap resolution calculus and encode relationally the effect of the individual Java program statements to the individual field relations. Expanding definitions of relational operators is particularly costly for non first-order operators like transitive closure since it leads to quantified integer formulas that generally require user interaction in form of manual induction.

10.2 Future Work

Our focus on core Alloy, regarding the language support, and on the automatic reasoning in the first-order logic augmented with transitive closure theory (TCFOL), regarding the reasoning support, restricts the set of considered Alloy benchmarks for our automatic verification approach for Alloy assertions. The achieved results so far are, however, promising and suggest the extension of the approach to support more language constructs and more non first-order theories such as Alloy's set cardinality. The main challenge in this respect, however, is the problem of theory combination – a well known problem in SMT solving [22, 17]. Rewrite based extensions are also worthy of investigation, even for set cardinality, as we have demonstrated for Alloy's ordering module —using a reduction to TCFOL logic.

Beside extending the support of the Alloy language, the individual techniques developed in the thesis give rise to several interesting theoretical and practical research questions.

Our sufficient ground term sets technique (SufGT) used to improve SMT solving for RFOL formulas on the one hand and to increase the efficiency of bounded verification in the Alloy Analyzer on the other hand, lacks of completeness and efficiency investigation. That is, if SufGT computes that the sufficient ground term set S for a variable x is infinite then there is no guarantee that x does not have yet a finite sufficient ground term set; if it computes that S is finite, although we proved that in this case x is equisatisfiably eliminable via instantiation with S, there is no guarantee that S is the minimal such set. Especially, the results of our extended semantics blasting technique (SB^+) which could eliminate variables of formulas that the SufGT technique could not, prove partially[2] the incompleteness of SufGT and show the potential of investigating the completeness of the SufGT technique.

[2]There is a small difference in the way of eliminating variables between both techniques.

Regarding our path-invariant based transitive closure reasoning technique (TCPInv) which can show the refutation of TCFOL formulas for which the standard SMT solving cannot show refutation, there exists several improvement possibilities which we already have discussed in the corresponding chapter. Here, we want to mention and emphasis two special research questions in this respect. The TCPinv technique is based on the claim that for any essential path p in a refutable formula modulo transitive closure theory there exists a p-invariant that helps refuting p with standard SMT solving. Since the transitive closure theory $\mathcal{T}^{R^+}_{tc_R}$ is only axiomatizable in second-order logic and in order to avoid consideration of second-order proof systems, we have proved the claim only for a bit weaker transitive closure theory $\mathcal{T}^{ind}_{tc_R}$. The $\mathcal{T}^{ind}_{tc_R}$ theory consists of the axiomatization of the transitive closure relation tc_R of relation R to be the transitive relation containing R and a transitive closure induction schema. Since $\mathcal{T}^{ind}_{tc_R}$ is weaker than $\mathcal{T}^{R^+}_{tc_R}$ two interesting research questions arise: (1) can our proof be extended to refutable formula modulo $\mathcal{T}^{R^+}_{tc_R}$ but not modulo $\mathcal{T}^{ind}_{tc_R}$ and (2) can one construct a formula that is refutable modulo $\mathcal{T}^{R^+}_{tc_R}$ but not modulo $\mathcal{T}^{ind}_{tc_R}$. Especially, the second question is of a high theoretical importance. Such a formula would play a similar role for the transitive closure theory as the well known Paris-Harrigton theorem [10, page 1133] for the integer theory, namely it exhibits with a concrete example the gap between the first- and second-order Peano axiomatization of integer theory.

Regarding the verification of Java programs against relational specifications using JKelloy, and beside the need of more experiments to further confirm the so far reached results, two research directions have in this respect high potential:

- The combination of specification concepts of the Java modeling language (JML) with our relational logic based specification approach. This combination has the potential to bring together the best of both paradigms and to make our approach tempting to more users, by supporting well known and established specification concepts.

- The use of the symbolic execution engine of KeY along with our relational heap resolution calculus to produce relational summaries of Java methods. Having such summaries, corresponding Java methods can be checked for bugs using the Alloy Analyzer before starting a proof attempt in JKelloy. For simpler loop-free Java methods, all needed tools are already in place. In this case, it suffices to (1) compute the post-heap as a function of the pre-heap using KeY's symbolic execution engine —straightforward for loop-free code, (2) use (1) along with the relational heap resolution calculus to compute field relations at the post-heap as a function of the field relations at the pre-heap and (3) apply the result of (2) to the corresponding Alloy assertions.

Appendix

Appendix A

An Arity Independent first-order Relational Framework

In this chapter we introduce a general logical framework for the arity-independent relational extension of first-order logic. Its target is to enable the use of (1) general sorts for relations and tuples and (2) general functions for the relational operators. This framework (respectively extension), called GRFOL, is a generalization of RFOL (introduced in Section 4.6).

Therefore, we assume in Ω two arity independent sorts, *Rel* for all relations and *Tuple* for all tuples. Figure A.1 shows the basic functions provided by the GRFOL framework together with their axioms. For functions that do not use the functional notation, we use dots as place holder for their arguments.

The boolean valued function $\in$ (aka. predicate) fixes the membership relation between tuples and relations and is initially uninterpreted. The function *ar* denotes the arity of tuples and relations. The third function written as $e_{i:j}$ where e is a relation (respectively tuple), i and j two natural number with $1 \leq i \leq j \leq ar(e)$ returns a relation (respectively tuple) of arity $j - i + 1$ representing the projection of e on its columns (respectively elements) i to j. The fourth function written as $\{t\}$ where t is a tuple returns the singleton relation containing t. The last symbol written as $t \parallel t'$ where t and t' are tuples returns the result tuple of concatenating t and t'.

In order to demonstrate the differences in axiomatizing Alloy relational operators in the GRFOL framework in comparison to RFOL, we list in Figure A.2 the arity independent relational operators of RGFOL and in Figure A.3 their axiomatization.

$$
\begin{aligned}
\mathcal{F}_\Sigma \overset{\cup}{\leftarrow} \{ & . \in . \subseteq Tuple \times Rel, \\
& ar : Rel \cup Tuple \to \mathbb{N}, \\
& ._{:.:} : Rel \cup Tuple \times \mathbb{N} \times \mathbb{N} \to Rel \cup Tuple, \\
& . \parallel . : Tuple \times Tuple \to Tuple \}
\end{aligned}
$$

$$
\begin{aligned}
Ax \overset{\cup}{\leftarrow} \{ & \forall R : Rel, t : Tuple.\ t \in R \to ar(t) = ar(R), && \text{(A.1)} \\
& \forall R : Rel, t : Tuple, i, j : \mathbb{N}.\ R_{i:j} \in Rel \wedge t_{i:j} \in Tuple, && \text{(A.2)} \\
& \forall x : Rel \cup Tuple, i, j : \mathbb{N}.\ \neg(1 \leq i \leq j \leq ar(x)) \to ar(x_{i:j}) = 0, && \text{(A.3)} \\
& \forall x : Rel \cup Tuple, i, j : \mathbb{N}.\ 1 \leq i \leq j \leq ar(x) \to ar(x_{i:j}) = j - i + 1, && \text{(A.4)} \\
& \forall R : Rel, t : Tuple.\ t \in R \to (\forall i, j : \mathbb{N}.\ 1 \leq i \leq j \leq ar(t) \to t_{i:j} \in R_{i:j}), && \text{(A.5)} \\
& \forall t, t' : Tuple.\ ar(t \parallel t') = ar(t) + ar(t'), && \text{(A.6)} \\
& \forall t, t' : Tuple.\ (t \parallel t')_{1:ar(t)} = t \wedge (t \parallel t')_{ar(t)+1:ar(t')} = t', && \text{(A.7)} \\
& \forall t, t' : Tuple.\ t = t' \to (ar(t) = ar(t') \wedge \forall i : \mathbb{N}.\ 1 \leq i \leq ar(t) \to t_{i:i} = {t'}_{i:i}), \} && \text{(A.8)}
\end{aligned}
$$

Figure A.1: A logical framework for a general relation sort based relational extension of first-order logic

$$
\begin{aligned}
& \mathcal{F}_\Sigma \overset{\cup}{\leftarrow} \{ \\
& \textbf{empty set} && \emptyset :\to Rel, \\
& \textbf{singleton} && \{\} : Atom^i \to Rel, \\
& \textbf{union} && \cup : Rel \times Rel \to Rel, \\
& \textbf{intersection} && \cap : Rel \times Rel \to Rel, \\
& \textbf{difference} && \setminus : Rel \times Rel \to Rel, \\
& \textbf{override} && \oplus : Rel \times Rel \to Rel, \\
& \textbf{product} && \times : Rel \times Rel \to Rel, \\
& \textbf{join} && \mathbf{.} : Rel \times Rel \to Rel, \\
& \textbf{transpose} && {}^{-1} : Rel \to Rel \}
\end{aligned}
$$

Figure A.2: The arity independent relational operators of GRFOL

$$\forall t : Tuple. \quad t \notin \emptyset \tag{A.9}$$

$$\forall t, t' : Tuple. \quad t' \in \{t\} \leftrightarrow t' = t \tag{A.10}$$

$$\forall R, S : Rel, t : Tuple. \quad t \in R \cup S \leftrightarrow t \in R \vee t \in S \tag{A.11}$$

$$\forall R, S : Rel, t : Tuple. \quad t \in R \cap S \leftrightarrow t \in R \wedge t \in S \tag{A.12}$$

$$\forall R, S : Rel, t : Tuple. \quad t \in R \setminus S \leftrightarrow t \in R \wedge t \notin S \tag{A.13}$$

$$\forall R, S : Rel, t : Tuple. \quad t \in R \oplus S \leftrightarrow t \in S \vee (t \in R \wedge (\forall t' : Tuple.\ ar(t') = ar(S) - 1 \rightarrow t_{1:1} \parallel t' \notin S)) \tag{A.14}$$

$$\forall R, S : Rel, t : Tuple. \quad t \in R \times S \leftrightarrow t_{1:ar(R)} \in R \wedge t_{ar(R)+1:ar(R)+ar(S)} \in S \tag{A.15}$$

$$\forall R, S : Rel, t : Tuple. \quad t \in R \,.\, S \leftrightarrow \exists u : Tuple.\ t_{1:ar(R)-1} \parallel u \in R \wedge u \parallel t_{ar(R):ar(R)+ar(S)-2} \in S \tag{A.16}$$

$$\forall R : Rel, t : Tuple. \quad t \in R^{-1} \leftrightarrow ar(t) = ar(R) = 2 \wedge t_{2:2} \parallel t_{1:1} \in R \tag{A.17}$$

Figure A.3: The arity independent axiomatization of GRFOL relational operators

Appendix B

A Transitive Closure based Rewrite of Alloy's Ordering Module

This appendix lists the Alloy file of our transitive closure based reduction of the Alloy ordering functionality applied to a signature S. The file contains also the original correction assertions of the original Alloy ordering module. The here shown reduction is made finite to allow for its analysis with the Alloy analyzer (AA).

```
//
// A transitive closure based axiomatization of a signature to a strict total order.
// Given the Alloy declaration "open util/ordering[elem]", our desugaring consists of:
//

//
// (1) Adding the fresh relation nextS: elem x elem
//
sig elem {
  // Only for the finite case and the analysis with AA, otherwise use one instead of lone
  nextS: lone elem
}

//
// (2) Adding the unary singleton relation firstS
//
one sig firstS in elem {}

//
// (3) Adding, only for the finite case and the analysis with AA, a singleton relation lastS
//
one sig lastS in elem {}
```

```
//
// (4) Axiomatizing nextS, firstS and lastS as follow
//
fact {
 all x: elem | x !in x.^nextS
 all x: elem | x = firstS or x in firstS.^nextS
 //Only for the finite case and the analysis with AA
 no lastS.nextS
}

//
// Some helper functions as provided by the original Alloy orderingmodule
//
fun yfirst: one elem {
  firstS
}
fun ylast: one elem {
  lastS
}
fun yprev : elem→elem {
  ~(nextS)
}
fun ynext : elem →elem {
  nextS
}
fun yprevs [s: elem]: set elem {
  s.^(~(nextS))
}
fun ynexts [s: elem]: set elem {
  s.^(nextS)
}

//
// Assertions of the original Alloy orderingmodule
//
assert correct {
 ( all b:elem | (lone b.ynext) and (lone b.yprev) and (b !in b.^ynext) )
 ( (no yfirst.yprev) and (no ylast.ynext) )
 ( all b:elem | (b!= yfirst and b!= ylast) ⇒ (one b.yprev and one b.ynext) )
 ( !one elem
    ⇒
    (one yfirst and one ylast and yfirst!= ylast and one yfirst.ynext and one ylast.yprev) )
 ( one elem ⇒ (yfirst= elem and ylast= elem and no yprev and no ynext) )
```

```
 ( yprev= ~ynext )
 ( elem = yfirst. * ynext )
 (all disj a,b:elem | a in b.^ynext or a in b.^yprev)
 (no disj a,b:elem | a in b.^ynext and a in b.^yprev)
 (all disj a,b,c:elem | (b in a.^ynext and c in b.^ynext) ⇒ (c in a.^ynext))
 (all disj a,b,c:elem | (b in a.^yprev and c in b.^yprev) ⇒ (c in a.^yprev))
}
check correct for 5
```

Bibliography

[1] Railway applications - communication, signalling and processing systems - software for railway control and protection systems. Standard EN 50128:2011, European Committee for Standardization, Brussels, Belgium, 2011.

[2] SMT-LIB The Satisfiability Modulo Theories Library. `http://smtlib.cs.uiowa.edu/`, 2015-07-09.

[3] Aharon Abadi, Alexander Rabinovich, and Mooly Sagiv. Decidable fragments of many-sorted logic. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 17–31. Springer-Verlag, October 2007.

[4] Nazareno Aguirre, Marcelo F. Frias, Pablo Ponzio, Brian J. Cardiff, Juan P. Galeotti, and Germán Regis. Towards abstraction for DynAlloy specifications. In *Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM)*, pages 207–225. Springer-Verlag, October 2008.

[5] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating model checking and theorem proving for relational reasoning. In *Proceedings of the 7th International Seminar on Relational Methods in Computer Science and 2nd International Workshop on Applications of Kleene Algebra (RelMICS)*, pages 21–33. Springer-Verlag, May 2003.

[6] Konstantinos Arkoudas. *Denotational Proof Languages*. PhD thesis, Massachusetts Institute of Technology, 2000.

[7] Arnon Avron. Transitive Closure and the Mechanization of Mathematics. In Fairouz D. Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, number 28 in Applied Logic Series, pages 149–171. Springer-Verlag, 2003.

[8] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, pages 171–177. Springer-Verlag, July 2011.

[9] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at: www.smt-lib.org.

[10] Jon Barwise and H. Jerome Keisler. *Handbook of mathematical logic*. Studies in logic and the foundations of mathematics ; 90. North-Holland, January 1989.

[11] Bernhard Beckert. A dynamic logic for the formal verification of Java card programs. In *Revised Papers from the First International Workshop on Java on Smart Cards: Programming and Security*, JavaCard '00, pages 6–24. Springer-Verlag, 2001.

[12] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[13] Jonathan Best. Proving Alloy models by introducing an explicit relational theory in SMT. Studienarbeit, Karlsruhe Institute of Technology, December 2012.

[14] Armin Biere. Resolve and expand. In *Proceedings of the 7th international conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 59–70. Springer-Verlag, May 2004.

[15] Robin E. Bloomfield, Dan Craigen, Frank Koob, Markus Ullmann, and Stefan Wittmann. Formal methods diffusion: Past lessons and future prospects. In Floor Koornneef and Meine van der Meulen, editors, *Computer Safety, Reliability and Security*, number 1943 in Lecture Notes in Computer Science, pages 211–226. Springer-Verlag, January 2000.

[16] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using first-order theorem provers in the Jahob data structure verification system. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 74–88. Springer-Verlag, January 2007.

[17] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, pages 335–349. Springer-Verlag, July 2005.

[18] 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)*, pages 427–442. Springer-Verlag, January 2006.

[19] Leonardo De Moura and Nikolaj Bjørner. Efficient E-matching for smt solvers. In *Proceedings of the 21st International Conference on Automated Deduction (CADE)*, pages 183–198. Springer-Verlag, 2007.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer-Verlag, March/April 2008.

[21] David Déharbe and Silvio Ranise. Satisfiability solving for software verification. *International Journal on Software Tools for Technology Transfer*, 11(3):255–260, June 2009.

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

[23] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[24] Lucas Dixon and Jacques Fleuriot. IsaPlanner: A prototype proof planner in isabelle. In Franz Baader, editor, *Proceedings of the 19st International Conference on Automated Deduction (CADE)*, number 2741 in Lecture Notes in Computer Science, pages 279–283. Springer-Verlag, January 2003.

[25] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE, pages 195–204. ACM, September 2007.

[26] Bruno Dutertre and Leonardo de Moura. The yices SMT solver. Technical report, SRI International, 2006.

[27] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A Type System for Object Models. In *Proceedings of the 12th International Symposium on Foundations of Software Engineering*, SIGSOFT/FSE, pages 189–199. ACM, November 2004.

[28] Jan Van Eijck. Defining (reflexive) transitive closure on finite models. `http://homepages.cwi.nl/~jve/papers/08/pdfs/FinTransClosRev.pdf`, 2008. Unpublished manuscript.

[29] Aboubakr Achraf El Ghazi. Experiment results of the sufficient ground terms simplification (SufGT). `http://i12www.ira.uka.de/~elghazi/sufGT_smt13_expData`, 2013.

[30] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via SMT solving. In *Proceedings of the 17th International Symposium on Formal Methods (FM)*, pages 133–148, Limerick, June 2011. Springer-Verlag.

[31] Aboubakr Achraf El Ghazi, Mana Taghdiri, and Mihai Herda. First-order transitive closure axiomatization via iterative invariant injections. In *Proceedings of the 7th NASA Formal Methods Symposium (NFM)*, pages 143–157, Pasadena, April 2015. Springer-Verlag.

[32] Aboubakr Achraf El Ghazi, Mattias Ulbrich, Christoph Gladisch, Shmuel Tyszberowicz, and Mana Taghdiri. JKelloy: A proof assistant for relational specifications of Java programs. In *Proceedings of the 6th NASA Formal Methods Symposium (NFM), 2014*, pages 173–187, Houston, April-May 2014. Springer-Verlag.

[33] Aboubakr Achraf El Ghazi, Mattias Ulbrich, Christoph Gladisch, Shmuel Tyszberowicz, and Mana Taghdiri. On verifying relational specifications of Java programs with JKelloy. Karlsruhe Reports in Informatics 2014-3, Karlsruhe Institute of Technology, 2014.

[34] Aboubakr Achraf El Ghazi, Mattias Ulbrich, Mana Taghdiri, and Mihai Herda. Reducing the complexity of quantified formulas via variable elimination. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories (SMT)*, pages 87–99, Helsinki, July 2013.

[35] Herbert Enderton and Herbert B. Enderton. *A Mathematical Introduction to Logic, Second Edition*. Academic Press, 2 edition, January 2001.

[36] Marcelo F. Frias, Armando M. Haeberer, and Paulo A. S. Veloso. A Finite Axiomatization for Fork Algebras. *Logic Journal of IGPL*, 5(3):1–10, May 1997.

[37] Marcelo F. Frias, Carlos Lopez Pombo, and Mariano Moscato. Alloy Analyzer+PVS in the analysis and verification of Alloy specifications. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 587–601. Springer-Verlag, March/April 2007.

[38] Juan Pablo Galeotti. *Software Verification using Alloy*. PhD thesis, Universidad de Buenos Aires, 2010.

[39] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, pages 306–320. Springer-Verlag, June/July 2009.

[40] Ulrich Geilmann. Verifying Alloy models using KeY. Diplomarbeit, Karlsruhe Institute of Technology, August 2011.

[41] Christoph Gladisch. Satisfiability solving and model generation for quantified first-order logic formulas. In *Proceedings of the 2nd International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*, pages 76–91. Springer-Verlag, October 2011.

[42] Christoph Gladisch and Shmuel Tyszberowicz. Specifying a linked data structure in JML for formal verification and runtime checking. In *Proceedings of the 16th Brazilian Symposium on Formal Methods (SBMF)*, volume 8195 of *LNCS*, pages 99–114. Springer-Verlag, September/October 2013.

[43] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today*, number 1000 in Lecture Notes in Computer Science, pages 366–373. 1995.

[44] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.

[45] Mihai Herda. Generating bounded counterexamples for KeY proof obligations. Master thesis, Karlsruhe Institute of Technology, January 2014.

[46] Neil Immerman, Alex Rabinovich, Tom Reps, Mooly Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. January 2004.

[47] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, April 2006.

[48] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, January 2012.

[49] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[50] Uwe Keller. Some remarks on the definability of transitive closure in first-order logic and datalog. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.8266&rep=rep1&type=pdf`, 2004. Unpublished manuscript.

[51] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[52] Shuvendu K Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 115–126. ACM, 2006.

[53] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[54] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programs. In *Proceedings of the 7th European Symposium on Programming Languages and Systems*, number 1381 in Lecture Notes in Computer Science, pages 170–184. Springer-Verlag, March/April 1998.

[55] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of the 24th ACM symposium on Applied Computing*, pages 615–622. ACM, March 2009.

[56] Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Mooly Sagiv, Srivastava Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Proceedings of the 20st International Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science, pages 99–115. Springer-Verlag, January 2005.

[57] John McCarthy. Towards a Mathematical Science of Computation. In *Proceedings of the IFIP Congress*, pages 21–28. North-Holland, August/September 1962.

[58] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[59] Mariano Moscato, Carlos Lopez Pombo, and Marcelo F. Frias. Dynamite 2.0: New features based on UnSAT-core extraction to improve verification of software requirements. In *Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 275–289. Springer-Verlag, September 2010.

[60] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, October 2006.

[61] Greg Nelson. Verifying reachability invariants of linked structures. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL, pages 38–47. ACM, 1983.

[62] Monty Newborn. *Automated Theorem Proving: Theory and Practice*. Springer-Verlag, 2001 edition, December 2000.

[63] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer-Verlag, June 1992.

[64] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[65] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[66] Zvonimir Rakamaric, Jesse Bingham, and Alan J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI, pages 106–121. Springer-Verlag, January 2007.

[67] Philipp Rümmer. E-matching with free variables. In *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 359–374. Springer-Verlag, March 2012.

[68] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.

[69] Uwe Schöning. *Logic for Computer Scientists*. Birkhäuser, January 2008.

[70] Philippe Suter, Robin Steiger, and Viktor Kuncak. Sets with cardinality constraints in satisfiability modulo theories. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 403–418. Springer-Verlag, January 2011.

[71] Mana Taghdiri. *Automating Modular Program Verification by Refining Specifications*. PhD thesis, Massachusetts Institute of Technology, 2008.

[72] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 632–647. Springer-Verlag, March/April 2007.

[73] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, pages 466–483. Springer-Verlag, 1983.

[74] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. On proving Alloy specifications using KeY. Karlsruhe Reports in Informatics 2011-37, Karlsruhe Institute of Technology, 2011.

[75] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. A proof assistant for Alloy specifications. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 422–436, Tallinn, March 2012. Springer-Verlag.

[76] Mandana Vaziri-Farahani. *Finding bugs in software with a constraint solver*. PhD thesis, Massachusetts Institute of Technology, 2004.

[77] Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.

[78] Kuat T. Yessenov. A Lightweight Specification Language for Bounded Program Verification. Master thesis, Massachusetts Institute of Technology, 2009.

[79] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 349–361. ACM, June 2008.