Karlsruhe Institute of Technology

Diploma Thesis

Verifying Alloy Models Using KeY

Ulrich Geilmann

August 16, 2011

Department of Informatics
Institute for Theoretical Computer Science

Responsible Supervisors:	JProf. Dr. Mana Taghdiri
	Prof. Dr. Peter H. Schmitt
Supervisors:	Mattias Ulbrich
	Aboubakr Achraf El Ghazi

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst zu haben und keine weiteren als die angegebenen Hilfsmittel verwendet zu haben.

Ulrich Geilmann
Karlsruhe, den 16. August 2011

Abstract

Alloy is a declarative specification language suitable for modeling structurally rich systems. It provides a relational first-order logic augmented with built-in operators for transitive closure, set cardinality, and integer arithmetics. The Alloy Analyzer automatically analyzes Alloy models by bounding the size of the modeled system. Therefore, although capable of finding counterexamples, it cannot prove the model correct. In this thesis, an interactive first-order theorem prover, namely the KeY system, is used to provide such proving capability.

Alloy specifications are translated to KeY's first-order logic. For this purpose, a relational first-order theory is defined that resembles the relational calculus offered by Alloy, and supports most of the language features including all relational, set, and numerical operators. Unlike the Alloy Analyzer that bounds the size of a model's instances, our translation allows instances to be infinite. We prove that our translation is correct for the Alloy kernel, a subset of the Alloy language for which a formal semantics is available.

A translated Alloy model is loaded in the KeY system for proving. KeY's automatic proof search strategy is extended to allow efficient relational reasoning. We evaluate our approach by several experiments. As a case study, we prove Dijkstra's solution to the dining philosophers problem.

Contents

1. Introduction

1.1 Overview

The Alloy specification language [19, 2] is a relational first-order logic, enriched with a number of features to provide a simple, yet efficient modeling notation. Its application to a wide range of diverse problems demonstrates the flexibility of the language. Examples include the analysis of the flash filesystem [21], checking functional properties of Java programs (see e.g. [28]), and modeling of network protocols (see e.g. [29]).

One reason for Alloy's popularity is its fully automatic analysis engine, called the Alloy Analyzer. Since Alloy's logic is undecidable, the automation comes at a price: models are checked with respect to a finite scope – a bound on the size of the model's instances – that is provided by the user. The Alloy Analyzer finds counterexamples, if there are any, within the given scope. This analysis helps the user find flaws in the model or gain confidence about its correctness. However, the Alloy Analyzer is not capable of a complete verification.

Critical systems rely on a correct design and a formal correctness proof is therefore valuable. Since Alloy is a first-order language, it is a natural approach to translate Alloy models to a first-order logic for the verification task. For proving, we use the KeY system [5], a semi-automatic first-order theorem prover. Alloy models are translated to KeY's first-order logic KeYFOL. Using KeY as a proving back-end for Alloy models is appealing for several reasons:

- The KeY system offers a simple, yet powerful way of defining first-order theories. Lemmas of a theory can be shown to follow from the axioms, thus keeping the set of axioms in a theory small and, as a consequence, reducing the risk of inconsistencies being introduced.
- Lemmas and axioms are conveniently written as taclet rules [4] for the underlying sequent calculus and can easily be incorporated into the existing automation strategy.
- KeY supports for integer arithmetic.

In order to translate Alloy models to KeYFOL, we developed a relational first-order theory for KeY that resembles the relational calculus offered by Alloy: each of Alloy's operators has a counterpart in KeYFOL. While most of the operators can be defined in a natural way, the semantics of some of Alloy's features, for example the cardinality operator, are only understood for finite relations. In our translation, however, relations are potentially infinite. We therefore included an axiomatization of finiteness which allows us to define such features distinctly for the finite and the infinite cases. Having the relational theory at hand, Alloy formulas and expressions are translated in a natural way.

Our translation has been proven correct for the kernel, a subset of the Alloy language for which a formal semantics is available. The translation is implemented in the Alloy2KeY tool. We call the collective of the translation tool and the KeY prover "Kelloy". There are a few language constructs that are currently not handled by Kelloy, see Section 7.3.

KeY's automatic proof search strategy has been extended to support the relational theory. The strategy allows for efficient semi-automatic reasoning about Alloy models: experiments showed that user interaction can often be narrowed down to the central steps of a proof while most subgoals are automized.

The applicability of our approach is demonstrated by means of a case study: We prove Dijkstra's solution [6] to the famous dining philosophers problem [18].

1.2 An Example

In this section we demonstrate the use of Kelloy by the means of a simple, yet motivating example to give the reader a rough overview without going into detail.

Our example is a very simple Alloy model of a filesystem, shown in Figure 1.1. There are two kinds of objects in the modeled filesystem: Files and directories. Each object in the filesystem has at most one parent directory (Line 2) and every directory is associated with the objects it contains – its entries (Line 8). A fact postulates that each directory is the parent of its entries (Line 13). Furthermore, there is unique root directory (Lines 11 and 14) that has no parent (Line 15) and from which all objects in the filesystem are reachable (Line 16).

We want to prove that except for the root directory, all objects in the filesystem have exactly one parent directory. The assertion `OneParent` formulates this property. The Kelloy tool generates a proof obligation for this assertion that is loaded into the KeY system for proving. A proof in the KeY system is conducted by applying deduction rules to a *proof sequent*.

An excerpt of the proof sequent for the example after some automatic simplification steps is shown in Figure 1.2. The sequent reads as follows: the formulas before the sequent sign $\vdash$ are assumed to hold, while the formulas following $\vdash$ are assumed to be false. The proof objective is to infer a contradiction from these assumptions.

The numbers next to the formulas indicate their origin in the Alloy model. Naturally, the translated facts are assumed to hold and thus appear on the left-hand side of the sequent. The assertion (Line 19) translates to

$$\forall o\colon Atom \mid in(o, diff_1(Object, Root)) \Rightarrow one(join_{1\times 2}(sin(o), parent))$$

```
1  abstract sig Object {
2    parent : lone Dir
3  }
4
5  sig File extends Object {}
6
7  sig Dir extends Object {
8    entries : set Object
9  }
10
11 sig Root extends Dir {}
12 fact {
13   all o: Object, d: Dir |
       o in d.entries ⇒ o.parent = d
14   one Root
15   no Root.parent
16   Object = Root.*entries
17 }
18 assert OneParent {
19   all o: Object - Root | one o.parent
20 }
21 check OneParent for 5
```

Figure 1.1: Simple Alloy model of a filesystem.

$$
\begin{aligned}
&13\ \forall o, d\text{: } Atom \mid in(o, Object) \land in(d, Dir) \\
&\quad \Rightarrow (subset(sin(o), join_{1\times2}(sin(d), entries)) \Rightarrow join_{1\times2}(sin(o), parent) \doteq sin(d)) \\
&16\ Object \doteq join_{1\times2}(Root, reflTransClos(entries)) \\
&\quad \vdash \\
&19\ in(o_0, diff_1(Object, Root)) \Rightarrow one(join_{1\times2}(sin(o_0), parent))
\end{aligned}
$$

Figure 1.2: Excerpt of the proof sequent.

Note that the bounding expression of the Alloy quantifier is explicitly included in the translation's quantification body. Since we conduct a proof-by-contradiction, we assume that the assertion does not hold. For some element o_0, the quantification body is thus violated and appears on the right-hand side of the sequent.

The KeY system automatically applies deduction rules to the sequent in order to find a proof. In this case, however, it does not find the right quantifier instantiations to finish the proof completely automatically, so we have to assist the prover with some interaction. Fortunately, providing one quantifier instantiation suffices: We instantiate the formula "13" with o_0. This adds the formula

$$
\begin{aligned}
&\forall d\text{: } Atom \mid in(o_0, Object) \land in(d, Dir) \\
&\quad \Rightarrow (subset(sin(o_0), join_{1\times2}(sin(d), entries)) \Rightarrow join_{1\times2}(sin(o_0), parent) \doteq sin(d))
\end{aligned}
$$

to the left-hand side of the sequent and KeY can now finish the proof automatically.

1.3 Outline

The next chapter covers the prerequisites needed to follow this thesis. In Chapter 3, we define the relational first-order theory for KeY and presents in detail how Alloy models are translated.

A correctness proof for our translation of the Alloy kernel is conducted in Chapter 4. We furthermore examine the relationship between the analysis performed by the Alloy Analyzer and the verification performed by Kelloy.

Chapter 5 gives an overview of how reasoning about an Alloy model takes place within the KeY system and presents the automation strategy.

Chapter 6 evaluates our approach and demonstrates its applicability by means of a case study.

2. Background

2.1 Alloy

Alloy [19, 2] is a modeling language based on a first-order relational logic. In this work we address a subset of Alloy version 4.1.10. Its syntax is shown in Figure 2.1. Further features of the language are treated as syntactic sugar by Kelloy (i.e., they are rewritten to the language of Figure 2.1). Some language constructs, however, are currently not handled, see Section 7.3 for a comprehensive list.

2.1.1 Expressions

The value of every Alloy expression is a relation. The simplest expressions are constants: `none` is the empty set, `univ` the universal set, and `iden` denotes the identity relation.

Besides the usual set operations union (+), intersection (`&`), and difference (-), Alloy offers several relational operators:

- The join operator "." denotes relational composition: `r.s` contains the tuple $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$ when there is an a such that `r` and `s` contain a tuple $\langle x_1, \ldots, x_n, a \rangle$ and $\langle a, y_1, \ldots, y_m \rangle$, respectively.
- `r->s` is the cartesian product of `r` and `s`.
- `~r` is the transposition of a binary relation `r`.
- The transitive closure of a homogenous, binary relation `r` is denoted by `^r` and the reflexive transitive closure by `*r`.
- For a unary relation `s`, `s<:r` denotes the domain restriction of `r` to `s` that contains those tuples from `r` that start with an element in `s`. Similarly, the range restriction `r:>s` contains those tuples of `r` that end with an element in `s`.
- The override operator `r ++ s` is similar to the union of `r` and `s`, but replaces a tuple of `r` with a tuple of `s` when they start with the same atom.

```
specification ::= open* paragraph*
path ::= ID / [path]
open ::= open [path] ID [ [ ref,+ ] ] [ as ID ]
paragraph ::= factDecl | assertDecl | funDecl
            | predDecl | cmdDecl | sigDecl

sigDecl ::= [abstract] sig ID [sigExt] { decl,* }
sigExt ::= extends ref | in ref [+ ref]*

factDecl ::= fact [ID] block
assertDecl ::= assert [ID] block
funDecl ::= fun ID [ decl,* ] : declExpr { expr }
predDecl ::= pred ID [ decl,* ] block
cmdDecl ::= check ref [ for number ]

decl ::= ID : declExpr
declExpr ::= [mult] expr | declRelExpr
declRelExpr ::= declRelExpr' [mult] → [mult] declRelExpr'
declRelExpr' ::= declRelExpr | expr
mult ::= lone | one | some | set

expr ::= ref | this | none | univ | iden
       | (~ | * | ^) expr | expr binOp expr | ref [ [ expr,* ] ]
       | formula [⇒ | implies] expr else expr
       | { decl,+ blockOrBar }
       | Int [ intExpr ] | ( expr )
binOp = & | + | - | ++ | <: | :> | . | →

intExpr ::= number | # expr | sum expr | int expr
          | intExpr (+ | -) intExpr | ( intExpr )

formula ::= expr (= | in) expr | expr in declExpr
          | (no | some | lone | one) expr
          | intExpr (< | > | =< | >=) intExpr
          | (! | not) formula | formula logicOp formula
          | formula [⇒ | implies] forumal else formula
          | quant decl blockOrBar
          | ref [ [ expr,* ] ]
          | ( formula ) | block
logicOp = || | or | && | and | ⇔| iff | ⇒ | implies
quant ::= all | no | some | lone | one
block ::= { forumla* }
blockOrBar ::= block | | formula

ref ::= [moduleRef] ID | Int | univ
moduleRef ::= [path] ID [ [ ref,+ ] ] /
```

Figure 2.1: Grammar of the Alloy language.

2.1.2 Formulas

Alloy offers two comparison operators that construct formulas from expressions: the subset operator `in` and the equality operator `=`. Quantified formulas take the form `Q x: e | F` where `Q` is one of the quantifiers `all`, `some`, `no`, `lone`, and `one`. The unary expression `e` bounds the quantification variable `x`, and `F` is a formula based on `x`. This is a first-order quantification, so `x` denotes a single element of `e`. However, all Alloy expressions are relational, so `x` is in fact a singleton subset of `e`. The meanings of the quantifiers are as follows:

- `all x: e | F` holds when `F` holds for every `x` in `e`
- `some x: e | F` holds when `F` holds for at least one `x` in `e`,
- `no x: e | F` holds when there is no `x` in `e` such that `F` holds,
- `lone x: e | F` holds when `F` holds for at most one `x` in `e`,
- `one x: e | F` holds when `F` holds for exactly one `x` in `e`.

The quantifiers `some`, `no`, `lone`, and `one` can also be applied to an expression in order to constraint the cardinality of its value. For example, the formula `one Root` in Line 14 of Figure 1.1 holds when `Root` has exactly one element and Line 15 states that `Root.parent` is empty.

Alloy offers all the standard propositional connectives including conjunction (`and`, `&&`), disjunction (`or`, `||`), negation (`not`, !), and implication (⇒ , `implies`)[1]. The block notation `{ ... }` that is used in Lines 14 to 16 of Figure 1.1 is a shorthand for conjunction.

2.1.3 Multiplicity Constraints

A formula of the form `r in e` denotes that `r` is a subset of `e`. In this particular case, however, `e` can contain *multiplicity annotations* that additionally restrict the value of `r`.

When `e` is a unary expression, it may be prefixed with one of the multiplicity keywords `set`, `one`, `lone`, and `some` in order to restrict the size of `r`:

- `r in (set e)` does not induce any restriction and holds when `r` is a subset of `e`.
- `r in (one e)` holds when `r` is a singleton subset of `e`.
- `r in (lone e)` holds when `r` is a subset of `e` that contains at most one element.
- `r in (some e)` holds when `r` is a non-empty subset of `e`.

Expressions of higher arity can induce restrictions by annotating the product operator → with multiplicity keywords on either sides. The allowed keywords are `one`, `lone` and `some`. `set` is also allowed in this context, but has the same effect as no keyword. The following examples illustrate the meanings of the annotations:

[1] Contrary to some programming languages like Perl, both versions of each operator are of the same precedence and thus completely interchangeable.

- `r in A → one B` holds when `r` is a total function from `A` to `B`: a binary relation that maps every element in `A` to exactly one element in `B`.
- `r in A → lone B` holds when `r` is a partial function from `A` to `B`.
- `r in A some → B` holds when `r` is a surjective, binary relation with domain `A` and range `B`: every element in `B` is mapped from some element in `A`
- `r in A one → one B` holds when `r` is a bijective function from `A` to `B`.

2.1.4 Signatures

A central part of Alloy's modeling notation are *signatures* which resemble classes in object-oriented programming languages like Java. The model of Figure 1.1 declares four signatures: `Object`, `File`, `Dir`, and `Root`. Each signature denotes a set[2] of uninterpreted elements, the *atoms*.

A signature can *extend* another signature: an extension is a subset of its parent. For example, `Dir` and `File` are subsets of `Object`. Extensions of a common signature are mutually disjoint. The signature `Object` is *abstract*: Every element in `Object` is an element in an extension of `Object`.

A signature can also be declared as a *subset signature*. In contrast to extensions, subset signatures of a common parent are not necessarily disjoint. A subset signature can have multiple parents. It is then a subset of the union of its parents. We can, for example, declare

```
sig Foo in File + Dir {}
```

2.1.5 Fields

Relations are declared as signature *fields*.

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

declares a relation `f` with domain `A`. The expression `e` may contain multiplicity annotations. For every element `this` in `A`, `e` bounds the value of `this.f`. We can write this equivalently as

```
all this: A | this.f in e
```

There is one exception: when `e` is unary and not prefixed with a multiplicity keyword, the declaration defaults to `one`; `f: e` is then equivalent to `f: one e`.

The signature `Dir` has a field `entries` that associates every element in `Dir` with a subset of `Object`. `entries` is thus a binary relation with domain `Dir` and range `Object`. The declaration of `parent` contains the multiplicity keyword `lone` which makes `parent` a partial function: A binary relation that associates every `Object` with at most one `Dir`.

When a field of the same signature appears in another field's declaration, it is interpreted in the context of that signature. We can, for example, add a field `hidden` to the `Dir` signature to capture that directory entries might be hidden:

[2] more precisely a unary relation

```
sig Dir extends Object {
  entries: set Object,
  hidden: set entries }
```

This declaration makes `this.hidden` a subset of `this.entries` for every element `this` in `Dir`.

2.1.6 Functions and Predicates

An Alloy model can declare *functions* and *predicates* which are reusable, parameterized expressions and formulas, respectively. For example, the following predicate is true when a given file is contained in a given directory or one of its subdirectories:

```
pred contains [d: Dir, f: File] {
  f in d.*entries }
```

The declaration of a function is similar, except that it also specifies the return value. We can, for example, declare a function `files` to return all files that are contained in a given directory:

```
fun files [d: Dir] : set File {
  d.entries & File }
```

To use a function or predicate, we instantiate each of its arguments with an expression. We can, for example, state the obvious:

```
all d: Dir, f: File | f in files[d] ⇒ contains[d,f]
```

2.1.7 Modules

An Alloy model can be split amongst several modules defined in separate files. One module can import the declarations of another one using the `open` directive.

Modules can be parameterized. A module parameter is a signature that is not declared in the module itself, but in the one importing it. The importing module therefore has to instantiate the parameters. The same parameterized module can be imported more than once, with different instantiations. To resolve ambiguity, an alias can be defined when a module is imported. The declarations of that module can then be accessed by prefixing their names with the alias and "/". For example,

```
open util/ordering[Dir] as do
```

imports the predefined ordering module, instantiates its parameter with the `Dir` signature, and defines the alias `do`.

The ordering module defines a linear ordering for its parameter signature and provides several functions to access the ordering. For example, `do/first` returns the smallest element of `Dir` and `do/next` returns a binary relation that associates every `Dir` with its direct successor in the ordering.

2.1.8 Integers

Alloy supports simple integer expressions which can be constructed from (1) literals `1`, `2`, etc., (2) the arithmetic operators `+` and `-`, (3) the cardinality expression `#e`, where `e` is an arbitrary relational expression.

Integer expressions are syntactically separated from relational expressions. Thus, integer values are no atoms. To use integers in relational expressions, Alloy provides the built-in signature `Int`. For every integer value `i`, `Int` contains exactly one atom corresponding to that value which can be obtained by `Int[i]`. The cast in the other direction is performed by `int[e]` for a relational expression `e`. If the value of `e` contains more than one integer-carrying atom, `int` returns the sum of all these values.

Alloy also offers the usual comparison operators `=`, `<`, `>`, `<=`, and `>=` to construct formulas from integer values.

2.1.9 Comprehensions

Relations can be built from formulas using comprehension expressions. The comprehension $\{x_1: e_1, \ldots, x_n: e_n \mid F\}$ denotes the n-ary relation that contains all tuples $\langle x_1, \ldots, x_n \rangle$ of $e_1 \to \ldots \to e_n$ for which `F` holds. Alloy only allows unary bounding expressions e_i, which may not be prefixed with a multiplicity keyword. For example,

```
{f1: File, f2: File | f1.parent = f2.parent}
```

constructs the binary relation that relates two files if and only if they have the same parent directory.

2.1.10 Analysis

An *instance* of an Alloy model is an assignment of sets and relations to the signatures and fields of the model, respectively. A *fact* is an axiomatic constraint that holds in every instance of the model, while an *assertion* is a property of the model that is expected to hold. The model of Figure 1.1 declares one fact and the assertion `OneParent`. The command `check OneParent for 5` instructs the Alloy Analyzer to look for *counterexamples* (i.e., instances that satisfy the facts but violate the assertion) in a *scope* of 5. The scope bounds the size of the sets being assigned to the signatures. Bounding the size of instances makes the logic decidable and the analysis can thus be performed automatically. When the Alloy Analyzer does not find a counterexample, the assertion is guaranteed to be valid within the analyzed scope. It does not, however, ensure that there are no counterexamples in a larger scope. Furthermore, the analysis performed by the Alloy Analyzer gives no information about the validity of an assertion in an infinite scope.

2.2 KeY

KeY [5] is a verification system for the Java programming language, based on the first-order dynamic logic JavaDL. KeY's first-order logic KeYFOL is a subset of JavaDL. At its heart, the KeY system is a deductive theorem prover based on a sequent calculus for JavaDL.

2.2.1 First-Order Logic

In order to prove an Alloy model correct, it is translated to the first-order logic as supported by the KeY system [5, 16], which we refer to as KeYFOL. The logic is typed and supports subtyping. We denote the set of types with $\mathcal{T}$, and the subtype relation with $\sqsubseteq$. We furthermore use F for the set of function symbols, P for the set of predicate symbols, and V_{FO} for the set of variables which we assume to be infinitely large.

The typing function α determines the type of each variable, the number and type of arguments of each function and predicate symbol, as well as the return type of function symbols:

- $\alpha(v) \in \mathcal{T}$ for all $v \in V_{FO}$
- $\alpha(f) \in \mathcal{T}^* \times \mathcal{T}$ for all $f \in F$
- $\alpha(p) \in \mathcal{T}^*$ for all $p \in P$

We call functions that do not take any arguments constants. We use the notation $f : T_1 \times \cdots \times T_n \to T_r$ to declare a function $f \in F$ such that $\alpha(f) = \langle\langle T_1, \ldots, T_n\rangle, T_r\rangle$ and $p \subseteq T_1 \times \cdots \times T_n$ to declare a predicate $p \in P$ such that $\alpha(p) = \langle T_1, \ldots, T_n\rangle$.

KeYFOL supports the standard propositional connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, the negation operator $\neg$, and the boolean constants **true** and **false**. It furthermore provides a conditional operator for terms and formulas, denoted by $\texttt{if}(c)\ \texttt{then}(a)\ \texttt{else}(b)$, and an equality predicate $\doteq$ which we use in infix form.

Quantification ranges over the values of a given type. We write the universal and existential quantification as $\forall x\colon T \mid \phi$ and $\exists x\colon T \mid \phi$, respectively, for some type T and formula ϕ. As a shorthand for nested quantification, we sometimes write $\forall x, y\colon T \mid \phi$ instead of $\forall x\colon T \mid \forall y\colon T \mid \phi$.

We denote the substitution of t for x in a formula ϕ with $\phi[x \leftarrow\!\shortmid\ t]$. The same notation is used for substitution in Alloy formulas.

KeYFOL formulas are evaluated in the context of *first-order states*. A first-order state $S = \langle \mathcal{D}, \delta, \mathcal{I}\rangle$ consists of a non-empty set $\mathcal{D}$, called the domain, a typing function δ, and an interpretation $\mathcal{I}$. The typing function $\delta : \mathcal{D} \to \mathcal{T}$ assigns a type to every element of the domain. We write $\mathcal{D}^T$ for the set of elements that are of type T (i.e. $\mathcal{D}^T = \{x \in \mathcal{D} : \delta(x) \sqsubseteq T\}$). The interpretation $\mathcal{I}$ maps each function symbol f to a function $f^{\mathcal{I}}$ of the appropriate type, and each predicate symbol p to a relation $p^{\mathcal{I}}$. The interpretation of the equality predicate is fixed in every first-order state: $\doteq^{\mathcal{I}} = \{\langle x, x\rangle \mid x \in \mathcal{D}\}$.

An *assignment* is a function $\beta : V_{FO} \to \mathcal{D}$ that maps each variable to a value of the appropriate type: $\beta(v) \in \mathcal{D}^{\alpha(v)}$. We write β^d_v for the assignment that maps v to d but otherwise agrees with β.

For every first-order state $S = \langle \mathcal{D}, \delta, \mathcal{I}\rangle$ and assignment β, we associate every term t with a value from the domain $t^{S,\beta} \in \mathcal{D}$, by defining

$$v^{S,\beta} = \beta(v) \text{ for every } v \in V_{FO}$$
$$[f(t_1, \ldots, t_n)]^{S,\beta} = f^{\mathcal{I}}(t_1^{S,\beta}, \ldots, t_n^{S,\beta})$$

We ultimately define the models relation $\models$ analogously to [5]. For a predicate invocation, that definition is

$$S, \beta \models p(t_1, \ldots, t_n) \text{ iff } \langle t_1^{S,\beta}, \ldots, t_n^{S,\beta} \rangle \in p^{\mathcal{I}}$$

$S, \beta \models \phi$ denotes that the formula ϕ holds in the first-order state S and the assignment β. The evaluation of closed formulas (i.e. formulas that do not contain free variables) is independent from the assignment and we write $S \models \phi$ in that case. We say that a formula ϕ is *valid* and write $\models \phi$ if it holds in every first-order state.

2.2.2 Sequent Calculus

A *sequent* consists of two sets of closed formulas, Γ and Δ, and is written as

$$\Gamma \vdash \Delta$$

We call Γ the *antecedent* and Δ the *succedent*. The semantics of the sequent is given by the formula

$$\bigwedge_{\phi \in \Gamma} \phi \Rightarrow \bigvee_{\psi \in \Delta} \psi$$

We can thus read the sequent as follows:

> It cannot be that all formulas in the antecedent are true while all formulas in the succedent are false.

To prove the validity of a formula ϕ, we therefore start with the sequent $\vdash \{\phi\}$ and try to construct a proof by applying deduction rules to the sequent. For brevity, we write Γ, ϕ instead of $\Gamma \cup \{\phi\}$, and ϕ instead of $\{\phi\}$ for a set of formulas Γ and a formula ϕ.

As an example[3], we demonstrate how to prove the formula $p \wedge q \Rightarrow q \wedge p$ for some constant predicates p and q. We start with

$$\vdash p \wedge q \Rightarrow q \wedge p$$

During the proof, this sequent is altered by rule applications. The following rules handle an implication in the succedent, respectively a conjunction in the antecedent.

$$\mathsf{impRight}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \Rightarrow \psi, \Delta} \qquad \mathsf{andLeft}\ \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}$$

Here, ϕ, ψ, Γ, and Δ are *schematic variables* that are instantiated when a rules is applied to a sequent. Throughout this thesis, the name of a schematic variable determines how it can be instantiated as specified in Table 2.1. All names of schematic variables might additionally be decorated with subscripts.

The meaning of the impRight rule is that if the premiss $\Gamma, \phi \vdash \psi, \Delta$ is a valid sequent, then the conclusion $\Gamma \vdash \phi \Rightarrow \psi, \Delta$ is a valid sequent. The application of the rule follows the opposite direction: The rule is applicable when the proof sequent matches

[3] which we borrowed from [5]

Schematic Variable	Instantiations
Δ, Γ	Sets of closed formulas
ϕ, ψ	Formulas
t, u, v, w	Terms of type *Tuple*[4]
a, b, c, d	Terms of type *Atom*
$t^{(2)}$, $u^{(2)}$	Terms of type *Tuple2*
$t^{(3)}$, $u^{(3)}$	Terms of type *Tuple3*
r, s	Terms of type *Relation*
$r^{(1)}$, $s^{(1)}$	Terms of type *Rel1*
$r^{(2)}$, $s^{(2)}$	Terms of type *Rel2*
$r^{(3)}$, $s^{(3)}$	Terms of type *Rel3*
i, j	Terms of type *int*

Table 2.1: Schematic variables and their instantiations.

the conclusion, that is, there are instantiations for the schematic variables such that the proof sequent is equal to the conclusion. The proof sequent can then be replaced by the premiss of the rule. To apply the impRight rule to our proof sequent, ϕ and ψ are instantiated with $p \wedge q$ and $q \wedge p$, respectively, while Γ and Δ are empty. The result of the application is the new proof sequent 2.2. Applying the andLeft rule then leads to 2.3:

$$\vdash p \wedge q \Rightarrow q \wedge p \tag{2.1}$$

$$p \wedge q \vdash q \wedge p \tag{2.2}$$

$$p, q \vdash q \wedge p \tag{2.3}$$

A sequent rule can have more than one premiss. The rule to handle a conjunction in the succedent is an example:

$$\text{andRight}\ \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi}$$

The meaning of a rule with several premisses is that if all premisses are valid, then the conclusion is valid. The application of the above rule to 2.3 therefore splits the proof into two branches:

$$p, q \vdash q \text{ and } p, q \vdash p$$

A rule without any premisses is called a closure rule and states the validity of the sequent that it is applied to. The closure rule

$$\text{close}\ \frac{}{\Gamma, \phi \vdash \phi, \Delta}$$

can be applied to the sequents of both branches by instantiating ϕ with q and p. We say that the branches are closed. When all branches of a proof are closed, the initial proof sequent has been proven valid.

[4]The mentioned types are defined in Chapter 3

The notation of sequent rules can be cumbersome. We therefore introduce some notational shorthands to be used throughout this thesis. An *inference rule*

$$\frac{\phi}{\psi}$$

allows us to infer ψ from the premiss ϕ. Its semantics is $\phi \Rightarrow \psi$. To apply the inference rule to a sequent, the premiss has to be matched in the antecedent. The conclusion can then be added to the antecedent. The inference rule is thus a shorthand for

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \vdash \Delta}$$

A premiss can also be negated. In this case it is matched in the succedent. An inference rule can also have multiple premisses. For example, the inference rule

$$\frac{\phi_1 \quad \neg\phi_2}{\psi}$$

is a shorthand for the sequent rule

$$\frac{\Gamma, \phi_1, \psi \vdash \phi_2, \Delta}{\Gamma, \phi_1 \vdash \phi_2, \Delta}$$

A *rewrite rule* allows us to replace some term l with another term r anywhere in the sequent. A rewrite rules is written as $l \rightsquigarrow r$ and has the semantics $l \doteq r$. Likewise, there are rewrite rules for formulas $\phi \rightsquigarrow \psi$ with the semantics $\phi \Leftrightarrow \psi$. There are also conditional rewrite rules which can only be applied when their premisses are matched in the sequent. The following rule, for example, can be used to replace a term with respect to an equation:

$$\frac{t \doteq s}{t \rightsquigarrow s}$$

We occasionally have to define the semantics of functions and predicates. We do this by axiomatic rewrite rules. To emphasize their character as definitions, we write

$$t := s \text{ and } \phi :\Leftrightarrow \psi$$

for $t \rightsquigarrow s$ and $\phi \rightsquigarrow \psi$, respectively.

3. Translating Alloy to First-Order Logic

This chapter presents how Alloy models are translated to KeY's first-order logic KeYFOL. Sections 3.1 to 3.7 cover the essential parts of the language, whilst Section 3.8 addresses more advanced features.

The translated model will be proved correct using KeY which potentially requires user interaction. To lower this burden, the translation should be transparent; the correspondence between the original model and its translation should be obvious. Section 3.1 therefore defines a first-order relational theory to which formulas are translated while preserving their structure.

An *instance* of an Alloy model is an assignment of sets and relations to the signatures and fields of that model. An instance is *admissible* if its assignments conform to the declarations of the signatures and fields. Sections 3.3 and 3.4 discuss the generation of *model constraints* that characterize the admissible instances of a particular Alloy model.

The translation of Alloy expressions and formulas is covered in sections 3.2, 3.5, and 3.6. Section 3.7 focusses on the translation of Alloy functions and predicates.

An Alloy model is regarded as correct if, for any admissible instance, the facts imply the assertions. Formally speaking, for a set of model constraints $\mathcal{M}$, facts $\mathcal{F}$, and assertions $\mathcal{C}$, we generate the following proof obligation in order to verify the model:[1]

$$\bigwedge \mathcal{M} \wedge \bigwedge \mathcal{F} \Rightarrow \bigwedge \mathcal{C}$$

Throughout this chapter, we use the following conventions:

- For any Alloy expression or formula `x`, x' denotes its translation.

[1] We only consider those assertions that are marked for proving by a `check` command and ignore the scope declaration that might be given.

- When considering named Alloy entities, for example variables, we need to create a proper identifier for its translation. For the purpose of presentation, we simply use the same name as in the original model. In the implementation, however, this is not always suitable, because (1) Alloy identifiers are not always legal identifiers in KeY, (2) Alloy allows (at least to some extent) overloading of names, (3) identifiers used in Alloy might be reserved names in KeY.

3.1 A First-Order Relational Theory

Our relational theory declares several first-order types, predicates and functions, as well as a set of rules for the sequent calculus, the axioms. This section introduces these parts and motivates the design choices that were made.

Alloy's signatures serve as basic types [20], so a crucial part of translating Alloy to KeYFOL is the representation of signatures. We have examined two orthogonal approaches for this.

The first approach is to introduce a type for every Alloy signature. Since KeYFOL offers subtyping, signature extensions can be expressed conveniently. However, the type system of KeYFOL is not powerful enough to express substantial parts of Alloy: For example, since no product or union types are provided, it is not clear what type should be assigned to a binary relation or the union of two distinct signatures.

We therefore selected the second, more flexible approach that captures the signature hierarchy by explicit constraints. We define one type for relations (including unary relations, i.e. sets), and another for tuples, called *Relation* and *Tuple*, respectively. In the translation, Alloy signatures and relations become constant function symbols of type *Relation*. The uninterpreted predicate $in \subseteq Tuple \times Relation$ relates the two types and denotes the membership of a tuple to a relation.

While the membership predicate is sufficient to define the usual set operations like union and intersection, expressing the semantics of relational operators, such as Alloy's join operator, requires access to the components of a tuple. For this purpose, we introduce subtypes of *Tuple* and *Relation* to capture the arity information:

$$Atom, Tuple2, Tuple3, \ldots \sqsubseteq Tuple$$
$$Rel1, Rel2, Rel3, \ldots \sqsubseteq Relation$$

Furthermore, tuples are built from atoms. To denote this, constructor functions are introduced:

$$binary : Atom \times Atom \to Tuple2$$
$$ternary : Atom \times Atom \times Atom \to Tuple3$$
$$\ldots$$

We state by axioms that (1) any tuple of at least arity two has a representation using the constructor function for that arity, and (2) constructor invocations are equal iff their parameters are equal. For arity two, these axioms are:

$$\forall t{:}\ Tuple2 \mid \exists a, b{:}\ Atom \mid t \doteq binary(a, b)$$
$$\forall a, b, c, d{:}\ Atom \mid binary(a, b) \doteq binary(c, d) \Leftrightarrow a \doteq c \wedge b \doteq d$$

The described approach allows us to define relational operators in a simple way (see Section 3.1.1). A drawback, however, is that relations of different arities have to be treated separately. For every arity, a distinct set of operators has to be defined. Chapter 4 presents a different approach, in which operators are defined arity-independently. The arity of tuples is captured by a function instead of a type, and the components of a tuple are accessed through a projection function. While the operator definitions are arity-independent, the representation of tuples is cumbersome and hardly intuitive. We value the simplicity of the translation over the redundant work that our approach requires. The implementation of Kelloy currently supports arities up to three, which is sufficient for the majority of Alloy models.

3.1.1 Operators

Alloy provides two ways for comparing relations: The subset operator `in`, and the equality operator `=`. In our relational theory, we define a subset predicate $subset \subseteq Relation \times Relation$ and axiomatize equality of relations. We furthermore introduce a predicate for disjointness $disj \subseteq Relation \times Relation$. The semantics of these predicates are defined by axioms. This is done separately for every arity, leaving comparison of relations with different arities undefined. For arity two, these axioms are:

$$\begin{aligned} subset(r^{(2)}, s^{(2)}) &:\Leftrightarrow \forall t\colon Tuple2 \mid in(t, r^{(2)}) \Rightarrow in(t, s^{(2)}) \\ disj(r^{(2)}, s^{(2)}) &:\Leftrightarrow \forall t\colon Tuple2 \mid \neg(in(t, r^{(2)}) \wedge in(t, s^{(2)})) \\ r^{(2)} \doteq s^{(2)} &:\Leftrightarrow \forall t\colon Tuple2 \mid in(t, r^{(2)}) \Leftrightarrow in(t, s^{(2)}) \end{aligned}$$

We only specify comparison of relations with matching arities. This principle of leaving arity mismatches undefined (known as underspecification [17]) is also applied to all other operator definitions and the membership predicate.

In Alloy, singleton sets[2] occur frequently. It is therefore useful to have an explicit notion for them. The function $sin : Atom \to Rel1$ serves this purpose by constructing the set containing only one atom. It is defined by

$$in(b, sin(a)) :\Leftrightarrow a \doteq b$$

Alloy provides several relational operators, as well as the standard set operations union, intersection, and difference. Similar to the comparison operators, we enhance the relational theory with a counterpart for each of these operators. As mentioned earlier, this has to be done separately for any arity. We denote this by subscripting the function names with the arity they are defined for. Table 3.1 shows the Alloy operators and their counterparts.

To fix the semantics of an operator, we specify the membership predicate applied to that operator. For example, the cartesian product of two sets $r^{(1)}$ and $s^{(1)}$ is defined by

$$in(binary(a, b), prod_{1\times1}(r^{(1)}, s^{(1)})) :\Leftrightarrow in(a, r^{(1)}) \wedge in(b, s^{(1)})$$

[2] which are expressed in Alloy by multiplicities

Name	Alloy	KeYFOL
Join	`r.s`	$join_{1\times2}(r', s')$, $join_{2\times2}(r', s')$, ...
Product	`r -> s`	$prod_{1\times1}(r', s')$, $prod_{1\times2}(r', s')$, ...
Union	`r + s`	$union_1(r', s')$, $union_2(r', s')$, ...
Intersection	`r & s`	$inter_1(r', s')$, $inter_2(r', s')$, ...
Difference	`r - s`	$diff_1(r', s')$, $diff_2(r', s')$, ...
Domain restriction	`r <: s`	$domRestr_2(r', s')$, $domRestr_3(r', s')$, ...
Range restriction	`r :> s`	$rangeRestr_2(r', s')$, $rangeRestr_3(r', s')$, ...
Override	`r ++ s`	$overr_2(r', s')$, $overr_3(r', s')$, ...
Transposition	`~r`	$transp(r')$
Transitive closure	`^r`	$transClos(r')$
Reflexive transitive closure	`*r`	$reflTransClos(r')$

Table 3.1: Alloy operators and their counterparts in first-order logic.

Note again that we do not define arity mismatched cases. The definition of the join operator is slightly more complex:

$$in(binary(a, c), join_{2\times2}(r^{(2)}, s^{(2)})) \\ :\Leftrightarrow \exists b\colon Atom \mid in(binary(a, b), r^{(2)}) \wedge in(binary(b, c), s^{(2)})$$

Most of the other operators have similar definitions, see Appendix B for their axiomatizations. Defining the transitive closure operator, however, is not that straightforward.

Transitive closure is defined for binary and homogenous relations, and can be obtained by repeatedly joining the relation with itself, until a fix point is reached: `^r = r + r.r + r.r.r + ...`

Unfortunately, such inductive definitions can not be expressed in first-order logic, so there is no recursively enumerable set of axioms that characterizes transitive closure [22]. The same problem arises when defining the natural numbers in a first-order language [12]. Nevertheless, KeY offers an approximation of the integers which we use to define the transitive closure operator.

The function $iterJoin : Rel2 \times int \to Rel2$ captures the above iteration. It is defined by a base case

$$iterJoin(r^{(2)}, 0) := r^{(2)}$$

and a recursive step

$$\frac{i > 0}{iterJoin(r^{(2)}, i) := union_2(iterJoin(r^{(2)}, i-1), join_{2\times2}(r^{(2)}, iterJoin(r^{(2)}, i-1)))}$$

Using the iterative join operator, the transitive closure is now defined by

$$in(t^{(2)}, transClos(r^{(2)})) :\Leftrightarrow \exists i\colon int \mid i \geq 0 \wedge in(t^{(2)}, iterJoin(r^{(2)}, i))$$

$$one(r^{(1)}) :\Leftrightarrow some(r^{(1)}) \wedge lone(r^{(1)})$$
$$lone(r^{(1)}) :\Leftrightarrow \forall a, b\colon Atom \mid in(a, r^{(1)}) \wedge in(b, r^{(1)}) \Rightarrow a \doteq b$$
$$some(r^{(1)}) :\Leftrightarrow \exists a\colon Atom \mid in(a, r^{(1)})$$

Figure 3.1: Definition of the multiplicity predicates

3.1.2 Constants

The simplest relational expressions are constants of which three are offered by Alloy: The empty set `none`, the universal set `univ`, and the identity relation `iden`. For each of these constants, we define a counterpart:

$$in(a, none) :\Leftrightarrow \textbf{false}$$
$$in(a, univ) :\Leftrightarrow \textbf{true}$$
$$in(binary(a, b), iden) :\Leftrightarrow a \doteq b$$

Although Alloy's `none` constant denotes a set, it is convenient to also have a constant for empty relations of higher arity. Analogously to *none*, we therefore define the constants $none_2$, $none_3$, et cetera.

3.2 Multiplicity Constraints

Alloy allows multiplicity annotations on the right-hand side of the subset operator `in`. A formula `r in e` holds when `r` is a subset of `e` and furthermore satisfies the multiplicity restrictions induced by `e`. There are two forms of multiplicity annotations, depending on whether the expression `e` is unary (i.e. set valued) or a higher-arity relational expression.

When `e` is a set valued expression, it may be prefixed with one of Alloy's multiplicity keywords `set`, `one`, `lone`, and `some` in order to restrict the size of `r`. To translate these multiplicity restrictions, we introduce a predicate for each case (with the exception of `set`). Figure 3.1 shows their definition for the unary case. The other cases are similar. Most interesting is the definition of *lone*, which states that the predicate holds iff there are no distinct elements in the set.

With these predicates at hand, we can translate the multiplicity restrictions of formulas like `r in (lone e)`. The formula translates to $subset(r', e') \wedge lone(r')$.

The second form of multiplicity annotations is an annotated product operator $\rightarrow$. For multiplicity keywords `n` and `m`, the multiplicity restriction of `r in A n ->m B` can be expressed by the following formulas:

```
all a: A | m a.r
all b: B | n r.b
```

We use these formulas to translate the multiplicity restrictions of an annotated product operator. For example, to capture the multiplicity restriction of `r in A ->one B`, we translate `all a: A | one a.r`. The translation of quantified formulas is covered in Section 3.6.

Annotated product operations may also be nested. These can be desugared in a similar way, using consecutive join operations. For example, the multiplicity restriction of `r in A →one B →lone C` is expressed by

```
all a: A | one a.r && all a: A, b: B | lone b.(a.r)
```

3.3 Signatures

Every Alloy model centers around its signatures. The model of Figure 1.1, declares four signatures:

```
abstract sig Object { ... }
sig File extends Object {}
sig Dir extends Object { ... }
sig Root extends Dir {}
```

For every signature declared in the model, a constant function symbol is introduced, for example *Object* : *Rel1*. These constants are further restricted by model constraints:

- The signatures `File` and `Dir` are extensions of `Object`, and `Root` extends `Dir`: $subset(File, Object)$, $subset(Dir, Object)$, $subset(Root, Dir)$
- `Object` is declared to be abstract, so every element in `Object` is an element in one of its extensions: $\forall a\text{: } Atom \mid in(a, Object) \Rightarrow in(a, File) \vee in(a, Dir)$[3]
- Top-level signatures as well as extensions of a common signature are mutually disjoint: $disj(File, Dir)$

A subset signature is a subset of the union of its parents. Thus,

```
sig Foo in File + Dir {}
```

translates to the constant function symbol *Foo* : *Rel1* and a model constraint $subset(Foo, union_1(File, Dir))$.

3.4 Fields

Alloy relations are declared as signature fields. The Alloy model of Figure 1.1 declares two fields:

```
abstract sig Object {
  parent : lone Dir }

sig Dir extends Object {
  entries : set Object }
```

[3]Note that, if `Object` had no extension, the constraint had constant **false** in place of the disjunction, and thus stated the emptiness of the signature.

Alloy		KeYFOL
F && G	F **and** G	$F' \wedge G'$
F || G	F **or** G	$F' \vee G'$
F ⇔ G	F **iff** G	$F' \Leftrightarrow G'$
F ⇒ G	F **implies** G	$F' \Rightarrow G'$
!F	**not** F	$\neg F'$
F ⇒ G **else** H	F **implies** G **else** H	$\texttt{if}(F')\ \texttt{then}(G')\ \texttt{else}(H')$

Table 3.2: Translating Alloy's logical connectives.

Like for signatures, the translation introduces a constant function symbol of the appropriate type for every field being declared. In this case *parent* : *Rel2* and *entries* : *Rel2*. The declaration of fields appear in the context of a signature. The above declaration, for example, defines `parent` to be a binary relation with domain `Object`. The context of the declaration is an element `this` of `Object` and the declaration expression bounds the value of `this.parent`. The declaration of `parent` can thus be expressed by the following formulas:

```
parent in Object→ Dir
all this: Object | this.parent in (lone Dir)
```

We generate the model constraints for `parent` by translating these formulas. When a field of the same signature appears in the declaration expression, it is also interpreted in the context of that signature. In Section 2.1.5, we introduced the field `hidden`:

```
sig Dir extends Object {
  entries: set Object,
  hidden: set entries }
```

This declaration is expressed by

```
hidden in Dir→ Object
all this: Dir | this.hidden in (set this.entries)
```

3.5 Expressions

Every constant and operator in Alloy has a counterpart in the relational theory, so its translation is straightforward. Other expressions include signatures and fields which are translated to constant function symbols. Variables (e.g. as introduced by quantifiers) are translated to logic variables of the appropriate type.

Alloy also features conditional expressions: $\texttt{c} \Rightarrow \texttt{e}_1$ `else` $\texttt{e}_2$ evaluates to $\texttt{e}_1$ when the constraint `c` holds, and to $\texttt{e}_2$ otherwise. Conveniently, KeY also offers a conditional operator. Thus, the expression is translated to $\texttt{if}(c')\ \texttt{then}(e_1')\ \texttt{else}(e_2')$.

3.6 Formulas

Alloy features several logical connectives, each of which has an equivalent operator in KeYFOL. Table 3.2 illustrates their translation.

Alloy offers two comparison operators to construct formulas from expressions: The equality operator that is translated to the equality in KeYFOL as described in Section 3.1.1, and the subset operator `in` which is translated as described in Section 3.2.

The multiplicity keywords `one`, `lone` and `some` can also be used in a formula to express that some set or relation has exactly one, at most one, or at least one element. Translating these formulas is simple using the predicates *one*, *lone* and *some* described in Section 3.2. Alloy also supports the formula `no e` to express that the expression `e` is empty, which is equivalent to the negation of `some e`, so we translate it to $\neg some(e')$.

3.6.1 Quantification

Section 2.1 introduced quantifications of the form `Q x: e | F` where `e` is a set-valued expression without a multiplicity keyword. The Alloy language also allows arbitrary declarations in a quantification, not just such declarations that bind the variable to a singleton set. These quantifications, however, can only be analyzed by the Alloy Analyzer if they can be eliminated by skolemization.

Since relations are first-order citizens in the translation, there is no need to adopt this limitation. Consider therefore a quantified formula of the form `Q x: e | F`, where `e` is an arbitrary expression that may contain multiplicity annotation, and `Q` is one of the quantifiers `all`, `some`, `one`, `lone`, and `no`. The quantification ranges over all relations `x` that satisfy `x in e`, with one exception: like for field declarations, the default multiplicity for a unary expression `e` is `one`; in this case, the quantification range is characterized by `x in (one e)`.

Let now B denote the translation of `x in e` (see Section 3.2) and α the arity of the expression `e`. KeYFOL offers universal and existential quantification, so translating the `all` and `some` quantifiers is rather straightforward: We include the restriction on the variable in the body and choose the appropriate sort to quantify over. The formula `all x: e | F` is thus translated to $\forall x\colon Rel\alpha \mid B \Rightarrow F'$ and `some x: e | F` becomes $\exists x\colon Rel\alpha \mid B \wedge F'$.

The `no` quantifier is just a shorthand for the negated existential quantifier. So instead of translating `no x: e | F`, we translate `not(some x: e | F)`.

The other quantifiers, namely `one` and `lone`, can be rewritten to `all` and `some` but this requires a little more effort. Consider the formulas `one x: e | F` and `lone x: e | F`. Let `y` be a variable that does not occur in `F`. The quantified formulas are then, respectively, equivalent to

`some x: e | (F && all y: e | (F[x ↤ y] ⇒ x = y))`
`all x: e | all y: e | ((F && F[x ↤ y]) ⇒ x = y)`

So far, we have quantified over relations. In a quantification where the declaration bounds the variable to a scalar (i.e. singleton and unary)[4], this is not very desirable and we rather want to quantify over atoms. This case is therefore treated specially. For a unary expression `e`, we translate `all x: e | F`[5] to

$$\forall x\colon Atom \mid in(x, e') \Rightarrow F'[x \leftarrow sin(x)]$$

and proceed similarly for existential quantification.

[4] which is the most frequent case

[5] and equally `all x: one e | F`

3.7 Functions and Predicates

Functions and predicates denote parameterized expressions and formulas, respectively. Recall the predicate `contains` we defined in Section 2.1:

```
pred contains [d: Dir, f: File] {
  f in d.*entries }
```

This predicate takes two unary parameters. We therefore introduce a declaration $contains \subseteq Rel1 \times Rel1$. Each predicate use `contains[d,f]` can now be translated to $contains(d', f')$.

The predicate's body translates to a formula ϕ with two free variables d and f. This formula is used to define a rewrite rule for the *contains* predicate:

$$contains(r^{(1)}, s^{(1)}) \rightsquigarrow \phi[d \leftarrow\!\shortmid\, r^{(1)}, f \leftarrow\!\shortmid\, s^{(1)}]$$

Note that we do not enforce the declaration constraint of the predicate's parameters. This conforms to the behavior of the Alloy Analyzer. Once the model is successfully typechecked, the parameter declarations are ignored. Alloy's typechecker guarantees that the call respects each parameter's arity, so our translation is appropriate for any well-typed model.

The translation of functions is similar. In Section 2.1, we defined the `files` function:

```
fun files [d: Dir] : set File {
  d.entries & File }
```

It translates to the function $files : Rel1 \rightarrow Rel1$ and the rewrite rule

$$files(r^{(1)}) \rightsquigarrow t[d \leftarrow\!\shortmid\, r^{(1)}]$$

where t is the translation of the function's body.

3.8 Advanced Features

3.8.1 Integers and Cardinalities

Alloy supports simple integer expressions, which can be constructed from (1) literals `1`, `2`, etc., (2) the arithmetic operators `+` and `-`, and (3) the cardinality expression `#e`, where `e` is an arbitrary relational expression.

The Alloy Analyzer analyzes integer expressions with respect to a finite bitwidth, which are therefore subject to overflow. When verifying a model, however, overflow is usually not intended and integers are assumed to be infinite. We take this into account by translating integer expressions using KeY's *int* type that has the semantics of mathematical integers.

With the exception of the cardinality operator, the translation of integer expressions is straightforward since literals and arithmetic operators are supported by the KeY system. KeY also supports the comparison operators offered by Alloy.

The `Int` signature is translated to a constant function symbol $Int : Rel1$, just like all other signatures. Of course, we also define it to be disjoint from all other top-level

signatures. We connect the elements of *Int* with integer values using the bijection $i2a : int \to Atom$. We also define $a2i : Atom \to int$ to invert $i2a$:

$$in(a, Int) :\Leftrightarrow \exists i\colon int \mid a \doteq i2a(i) \qquad i2a(i) \doteq i2a(j) :\Leftrightarrow i \doteq j$$

$$a2i(i2a(i)) := i \qquad \frac{in(a, Int)}{i2a(a2i(a)) := a}$$

When analyzing a model using the Alloy Analyzer, every relation is finite and the cardinality operator is therefore defined for any expression. In our translation, however, relations are potentially infinite, so we can only define a cardinality operator for those relations that are known to be finite. For this purpose, we introduce a finiteness flag $finite \subseteq Relation$, and a function $card : Relation \to int$ to yield the cardinality of a finite relation. An inference system for the finiteness flag is introduced in Section 5.3. Kelloy also allows the user to explicitly finitize signatures at translation time.

The next step is to explicitly finitize the relations that are flagged as finite. This is done by defining an *enumerator*, a bijection from an integer interval to the finite relation, thus making the relation isomorphic to that interval. We do this separately for each arity. In the unary case, $elem_1 : Rel1 \times int \to Atom$ provides the enumerator for every finite set:

$$\frac{finite(r^{(1)}) \quad in(a, r^{(1)})}{\exists i\colon int \mid i \geq 0 \wedge i < card(r^{(1)}) \wedge a \doteq elem_1(r^{(1)}, i)}$$

$$\frac{finite(r^{(1)})}{\forall i\colon int \mid i \geq 0 \wedge i < card(r^{(1)}) \Rightarrow in(elem_1(r^{(1)}, i), r^{(1)})}$$

$$\frac{finite(r^{(1)})}{\forall i, j\colon int \mid i, j \geq 0 \wedge i, j < card(r^{(1)}) \Rightarrow (elem_1(r^{(1)}, i) \doteq elem_1(r^{(1)}, j) \Leftrightarrow i \doteq j)}$$

This makes every set $r^{(1)}$, for which $finite(r^{(1)})$ holds, a finite set with $card(r^{(1)})$ elements. There are several different ways to axiomatize finiteness and cardinality. This one, however, comes in handy when translating Alloy's ordering module (see Section 3.8.2).

Being able to enumerate the elements of a set is also convenient for the translation of the `int` operator, which sums up the values of all `Int` atoms in a set. We define the function $sum : Rel1 \to int$ using KeY's bounded sum operator.

$$\frac{finite(r^{(1)})}{sum(r^{(1)}) := \sum_{i:int}^{0 \ldots card(r^{(1)})} (\texttt{if}(in(elem_1(r^{(1)}, i), Int)) \ \texttt{then}(a2i(elem_1(r^{(1)}, i))) \ \texttt{else}(0))}$$

The bounded sum operator Σ replaces the variable i in the given term by every integer value from the defined interval, and sums up all values.

The enumerator functions associate each element of a finite relation with an ordinal number. We introduce the function $ord : Relation \times Tuple \to int$ to provide us with uniform access to this number. It is defined to invert the enumerator functions:

$$\frac{finite(r^{(1)}) \quad i \geq 0 \quad i < card(r^{(1)})}{ord(r^{(1)}, elem_1(r^{(1)}, i)) := i}$$

3.8.2 Module Imports

A module can import the declarations of another one using the `open` directive. Translating a module that (possibly indirectly) imports other modules is theoretically trivial, and performed as if all imported declarations were placed in the root module (with potential name clashes resolved).

Translating a parameterized module import is done similarly, with the module's parameters being replaced by their instantiations.

The Alloy Analyzer provides several predefined modules. Most of them can be translated just like user-defined modules. The only exception is the frequently used ordering module, which we thus treat specially.

The Ordering Module

In Alloy, every signature is finite for the sake of analysis. It is therefore possible to define a total ordering on the elements of a signature, thus identifying a smallest and a largest element for non-empty signatures. The ordering module does exactly this for its parameter signature by defining the binary relation `next` that relates an element with its direct successor in the ordering. Although that property could also be stated in pure Alloy, the module uses a built-in predicate for that purpose, which enables the Alloy Analyzer to perform special optimizations. As a result, the usual translation approach is not applicable to the ordering module.

If the signature `S` being ordered is finite (i.e. $\mathit{finite}(S)$ holds), translating the `next` relation is rather simple since the enumerator elem_1 assigns an ordinal number to the elements of S. In that case, we define a counterpart for the `next` relation $\mathit{nextS} : \mathit{Rel2}$ by

$$\frac{\mathit{finite}(S)}{\mathit{subset}(\mathit{nextS}, \mathit{prod}_{1\times 1}(S, S))} \tag{3.1}$$

$$\frac{\mathit{finite}(S) \quad \mathit{in}(a, S) \quad \mathit{in}(b, S)}{\mathit{in}(\mathit{binary}(a, b), \mathit{nextS}) := \mathit{ord}(S, a) + 1 \doteq \mathit{ord}(S, b)} \tag{3.2}$$

For an infinite signature `S`, the `next` relation should intuitively relate every element with its direct successor. Defining such a relation makes the set countable, hence isomorphic to the natural numbers. Similar to the finitization shown in Section 3.8.1, we define elem_1 to provide such an isomorphism. We also define *ord* to invert elem_1. Figure 3.2 shows the rules for this. We can then define *nextS* the same way as in the finite case, so we drop the premiss $\mathit{finite}(S)$ from 3.1 and 3.2.

The built-in Alloy predicate that is used by the ordering module to define `next` also defines the smallest element of `S`, which can be obtained using the function `first`. Naturally, its counterpart $\mathit{firstS} : \mathit{Rel1}$ should yield the element associated with 0 if the set is not empty. It is, however, unclear what the semantics of `first` should be, if `S` is in fact empty. In the Alloy Analyzer, the instantiation of the ordering module is forced to contain exactly as many elements as permitted by the scope. However, defining a scope of zero makes the model inconsistent, thus any assertion becomes vacuously true. To overcome this problem, our translation leaves that case undefined. The definition of *firstS* is then given by following rules:

$$\frac{\mathit{finite}(S) \quad \mathit{card}(S) > 0}{\mathit{firstS} := \mathit{sin}(\mathit{elem}_1(S, 0))} \qquad \frac{\neg\mathit{finite}(S)}{\mathit{firstS} := \mathit{sin}(\mathit{elem}_1(S, 0))}$$

$$\frac{in(a,S) \quad \neg \mathit{finite}(S)}{\exists i\colon \mathit{int} \mid i \geq 0 \wedge a \doteq \mathit{elem}_1(S,i)} \qquad \frac{\neg \mathit{finite}(S)}{\forall i\colon \mathit{int} \mid i \geq 0 \Rightarrow \mathit{in}(\mathit{elem}_1(S,i),S)}$$

$$\frac{\neg \mathit{finite}(S)}{\forall i,j\colon \mathit{int} \mid i,j \geq 0 \Rightarrow (\mathit{elem}_1(S,i) \doteq \mathit{elem}_1(S,j) \Leftrightarrow i \doteq j)}$$

$$\frac{i \geq 0 \quad \neg \mathit{finite}(S)}{\mathit{ord}(S, \mathit{elem}_1(S,i)) \rightsquigarrow i}$$

Figure 3.2: Rules to linearly order an infinite set S.

The ordering module also declares several functions and predicates that are derived from `next` and `first`. We can translate their definitions as usual. An interesting case is the function for the largest element, `last`, which yields the set of elements that have no successor in `next`. In the finite case, this set contains exactly one element. In the infinite case, however, every element has a successor, so the empty set is returned.

Proving properties about all elements of a linearly-ordered set is often performed using induction. Conveniently, KeY already offers an induction theorem for its integers which can be used for that.

3.8.3 Comprehensions

A comprehension $\{x_1\colon e_1, \ldots, x_n\colon e_n \mid F\}$ denotes the n-ary relation that contains all tuples $\langle x_1, \ldots, x_n \rangle$ of $e_1 \to \ldots \to e_n$ for which F holds. We illustrate the translation of comprehensions for the binary case, the others are analogous.

To translate the binary comprehension $\{x_1\colon e_1, x_2\colon e_2 \mid F\}$, we define an operator $\mathit{compr}_2 : \mathit{Formula} \to \mathit{Rel2}$. It takes a formula as an argument which indicates whether a tuple belongs to the relation. Consider the parameterized formula G that is obtained from the comprehension:

$$G(a,b) \equiv \mathit{in}(a, e_1') \wedge \mathit{in}(b, e_2') \wedge F'[x_1 \leftarrow\!\shortmid \mathit{sin}(a), x_2 \leftarrow\!\shortmid \mathit{sin}(b)]$$

Intuitively, the tuple $\mathit{binary}(a,b)$ should be a member of the relation that compr_2 constructs, iff $G(a,b)$ holds. To implement parameterized formulas in KeYFOL, we introduce an operator to perform a λ-abstraction: The symbol $\texttt{bind}\ v : T \mid \phi$ binds a variable v of type T that is free in ϕ. The formula

$$\texttt{bind}\ u : \mathit{Atom} \mid \texttt{bind}\ v : \mathit{Atom} \mid G(u,v) \tag{3.3}$$

can then be used as a parameterized formula by substituting the parameter terms for the variables bound by `bind`. We define the compr_2 operator in this way:

$$\mathit{in}(\mathit{binary}(a,b), \mathit{compr}_2(\texttt{bind}\ u : \mathit{Atom} \mid \texttt{bind}\ v : \mathit{Atom} \mid \phi)) :\Leftrightarrow \phi[u \leftarrow\!\shortmid a, v \leftarrow\!\shortmid b]$$

We now use 3.3 as the argument to compr_2 and translate the binary comprehension to

$$\mathit{compr}_2(\texttt{bind}\ u : \mathit{Atom} \mid \texttt{bind}\ v : \mathit{Atom} \mid G(u,v))$$

4. Theoretical Evaluation

In this chapter, we discuss the translation of Alloy models from a theoretical viewpoint. We formalize the translation approach for a subset of the Alloy formulas, known as the Alloy kernel [8, 19]. For this purpose, we introduce arity-independent versions of the operators from the relational theory of Chapter 3 Section 4.2 shows the formalization of the translation and Section 4.3 conducts a proof of its correctness.

In Section 4.4, we discuss the impacts of a successful verification attempt and examine the relationship between the verification performed by Kelloy and the analysis performed by the Alloy Analyzer.

4.1 Arity-Independent Operators

The relational theory developed in Chapter 3 is well suited for translating and reasoning in a simple and effective way. For a theoretical evaluation, however, it is not suitable since it handles every arity separately. Hence, general correctness properties cannot be proven. To formalize the translation of the Alloy kernel in an arity-independent way, we define a set of new, arity-independent operators that are generalizations of the corresponding operators discussed in Chapter 3.

The declarations and axioms for the new operators are shown in Figure 4.1. To distinguish the newly defined operators from the old ones, we annotate them with a prime, for example $join'$. In Chapter 3, we used the types *Tuple2*, *Tuple3*, et cetera to capture the arity of a tuple and constructor functions provided us with access to its components. For the arity-independent operators, we declare functions to serve this purpose: *ar* assigns an arity to each tuple and the projection function *proj* provides access to the components of a tuple.[1] These functions fully characterize each tuple, as it is expressed by the equality axiom. We define some auxiliary functions to work with tuples in a more convenient way: *conc* returns the concatenation of two tuples, *drop* and *tail* remove the last, respectively the first component of a tuple.

[1] It may not seem intuitive that the components of a tuple – having type *Atom* – are again tuples. This does, however, not impose any theoretical problems but is closer to the definitions from Chapter 3.

$$
\begin{array}{ll}
ar : \mathit{Tuple} \to \mathit{int} & \mathit{join}' : \mathit{Relation} \times \mathit{Relation} \to \mathit{Relation} \\
\mathit{proj} : \mathit{Tuple} \times \mathit{int} \to \mathit{Atom} & \mathit{prod}' : \mathit{Relation} \times \mathit{Relation} \to \mathit{Relation} \\
 & \mathit{union}' : \mathit{Relation} \times \mathit{Relation} \to \mathit{Relation} \\
\mathit{conc} : \mathit{Tuple} \times \mathit{Tuple} \to \mathit{Tuple} & \mathit{inter}' : \mathit{Relation} \times \mathit{Relation} \to \mathit{Relation} \\
\mathit{drop} : \mathit{Tuple} \to \mathit{Tuple} & \mathit{diff}' : \mathit{Relation} \times \mathit{Relation} \to \mathit{Relation} \\
\mathit{tail} : \mathit{Tuple} \to \mathit{Tuple} & \mathit{transp}' : \mathit{Relation} \to \mathit{Relation} \\
 & \mathit{iterJoin}' : \mathit{Relation} \times \mathit{int} \to \mathit{Relation} \\
\mathit{subset}' \subseteq \mathit{Relation} \times \mathit{Relation} & \mathit{transClos}' : \mathit{Relation} \to \mathit{Relation} \\
\mathit{one}' \subseteq \mathit{Relation} & \mathit{none}' : \mathit{Relation} \\
 & \mathit{sin}' : \mathit{Tuple} \to \mathit{Relation}
\end{array}
$$

$$ar(t) \geq 0 \qquad ar(a) := 1 \qquad proj(a, 0) := a$$

$$t_1 \doteq t_2 :\Leftrightarrow ar(t_1) \doteq ar(t_2) \wedge \forall i\colon int \mid i \geq 0 \wedge i < ar(t_1) \Rightarrow proj(t_1, i) \doteq proj(t_2, i)$$

$$\frac{ar(t) \geq 1}{ar(drop(t)) := ar(t) - 1} \qquad \frac{ar(t) \geq 1 \quad n \geq 0 \quad n < ar(t) - 1}{proj(drop(t), n) := proj(t, n)}$$

$$\frac{ar(t) \geq 1}{ar(tail(t)) := ar(t) - 1} \qquad \frac{ar(t) \geq 1 \quad n \geq 0 \quad n < ar(t) - 1}{proj(tail(t), n) := proj(t, n+1)}$$

$$ar(conc(t_1, t_2)) := ar(t_1) + ar(t_2) \qquad \frac{n \geq 0 \quad n < ar(t_1)}{proj(conc(t_1, t_2), n) := proj(t_1, n)}$$

$$\frac{n \geq ar(t_1) \quad n < ar(t_1) + ar(t_2)}{proj(conc(t_1, t_2), n) := proj(t_2, n - ar(t_1))}$$

$$in(t, join'(r_1, r_2)) :\Leftrightarrow \exists u, v\colon Tuple \mid in(u, r_1) \wedge in(v, r_2) \wedge conc(drop(u), tail(v)) = t \wedge proj(u, ar(u) - 1) = proj(v, 0)$$

$$in(t, prod'(r_1, r_2)) :\Leftrightarrow \exists u, v\colon Tuple \mid in(u, r_1) \wedge in(v, r_2) \wedge conc(u, v) \doteq t$$

$$in(t, union'(r_1, r_2)) :\Leftrightarrow in(t, r_1) \vee in(t, r_2)$$

$$in(t, inter'(r_1, r_2)) :\Leftrightarrow in(t, r_1) \wedge in(t, r_2)$$

$$in(t, diff'(r_1, r_2)) :\Leftrightarrow in(t, r_1) \wedge \neg in(t, r_2)$$

$$in(t, transp'(r)) :\Leftrightarrow ar(t) \doteq 2 \wedge in(conc(tail(t), drop(t)), r)$$

$$in(t, transClos'(r)) :\Leftrightarrow \exists i\colon int \mid i \geq 0 \wedge in(t, iterJoin'(r, i))$$

$$in(t, iterJoin'(r, 0)) :\Leftrightarrow in(t, r) \wedge ar(t) \doteq 2$$

$$\frac{n \geq 1}{\begin{array}{l} in(t, iterJoin'(r, n)) :\Leftrightarrow ar(t) \doteq 2 \wedge \\ \quad in(t, union'(iterJoin'(r, n-1), join'(r, iterJoin'(r, n-1)))) \end{array}}$$

$$subset'(r_1, r_2) :\Leftrightarrow \forall u\colon Tuple \mid in(u, r_1) \Rightarrow in(u, r_2)$$

$$r_1 \doteq r_2 :\Leftrightarrow \forall u\colon Tuple \mid in(u, r_1) \Leftrightarrow in(u, r_2)$$

$$in(t, none') :\Leftrightarrow \mathbf{false}$$

$$in(t, sin'(s)) :\Leftrightarrow t \doteq s$$

$$one'(r) :\Leftrightarrow \exists t_1\colon Tuple \mid (in(t_1, r) \wedge \forall t_2\colon Tuple \mid (in(t_2, r) \Rightarrow t_1 \doteq t_2))$$

Figure 4.1: Declarations and Axioms for the arity-independent operators.

Having these functions at hand, it is not a challenge to define all the operators needed to translate the Alloy kernel, although the definitions are not as intuitive as in Chapter 3. Note that there is a unique[2] tuple with arity zero. Permitting this saves us some special cases when defining the *join′* operator. Note further that it is not necessary to restrict the tuples in a relation to have the same arity.

So far, the functions *ar* and *proj* are uninterpreted. To establish a relationship between the newly defined and the former operators, however, we have to connect the semantics of *ar* and *proj* to the arity-capturing types and the constructor functions. For *Tuple2*, *Rel2*, and *binary*, this is achieved by the following axioms:

$$\forall r\colon \mathit{Rel2} \mid \forall t\colon \mathit{Tuple} \mid \mathit{in}(t, r) \Rightarrow \mathit{ar}(t) \doteq 2 \qquad \forall t\colon \mathit{Tuple2} \mid \mathit{ar}(t) \doteq 2$$
$$\forall a, b\colon \mathit{Atom} \mid \mathit{proj}(\mathit{binary}(a, b), 0) \doteq a \qquad \forall a, b\colon \mathit{Atom} \mid \mathit{proj}(\mathit{binary}(a, b), 1) \doteq b$$

Using these axioms, one can prove that the definitions of the arity-dependent operators agree with the corresponding arity-independent operators. We show this exemplary for the $\mathit{prod}_{1\times 2}$ operator.

Theorem 1. The operators *prod′* and $\mathit{prod}_{1\times 2}$ coincide in the following sense:

$$\begin{aligned}&\forall r\colon \mathit{Rel1} \mid \forall s\colon \mathit{Rel2} \mid \forall a, b, c\colon \mathit{Atom} \mid \\ &\quad \mathit{in}(\mathit{ternary}(a, b, c), \mathit{prod}_{1\times 2}(r, s)) \Leftrightarrow \mathit{in}(\mathit{ternary}(a, b, c), \mathit{prod}'(r, s))\end{aligned}$$

Proof. With the definitions of both operators applied, we get the following objective:

$$\begin{aligned}&\forall r\colon \mathit{Rel1} \mid \forall s\colon \mathit{Rel2} \mid \forall a, b, c\colon \mathit{Atom} \mid \mathit{in}(a, r) \wedge \mathit{in}(\mathit{binary}(b, c), s) \\ &\quad \Leftrightarrow \exists t_1, t_2\colon \mathit{Tuple} \mid \mathit{in}(t_1, r) \wedge \mathit{in}(t_2, s) \wedge \mathit{conc}(t_1, t_2) \doteq \mathit{ternary}(a, b, c)\end{aligned}$$

"⇒": To show the first direction of the equivalence, we show $\mathit{conc}(a, \mathit{binary}(b, c)) \doteq \mathit{ternary}(a, b, c)$ using the equality axiom for tuples. We thus have to prove that (1) the arities match, and (2) the components of both tuples are equal. We omit the details here.

"⇐": Let t_1 and t_2 be some elements of *Tuple*, such that $\mathit{in}(t_1, r)$, $\mathit{in}(t_2, s)$, and $\mathit{conc}(t_1, t_2) \doteq \mathit{ternary}(a, b, c)$. From the two former assumption, we infer $\mathit{ar}(t_1) \doteq 1$ and $\mathit{ar}(t_2) \doteq 2$ using the axioms for the *Rel1* and *Rel2* types. Using the last assumption and the equality axiom for tuples, we can show $t_1 \doteq a$ and $t_2 \doteq \mathit{binary}(b, c)$, thus concluding the proof. □

4.2 Formalization of the Translation

Proving correctness properties requires the translation to be formally defined. This section presents such a formalization for the Alloy kernel. The kernel is a subset of Alloy's logical formulas and has a formal semantics [19, 8]. Most other features of the Alloy language can be desugared to the kernel. However, a formal description of such desugaring is – to the best of our knowledge – not available.

The syntactical category `relationName` represents the signatures and fields that are declared in a model, which we formally denote as the set N. For any signature and

[2] due to the equality axiom

formula ::= *elemFormula* | *compFormula* | *quantFormula*
elemFormula ::= *expr* **in** *expr* | *expr* **=** *expr*
compFormula ::= **not** *formula* | *formula logicop formula*
logicop ::= **and** | **or** | ⇒
quantFormula ::= *quantifier var* : *expr* | *formula*
quantifier ::= *all* | *some*

expr ::= *relationName* | *var* | **none** | *expr binop expr* | *unop expr*
binop ::= **-** | **+** | **&** | **.** | →
unop ::= **^** | **~**

relationName ::= ID
var ::= ID

Figure 4.2: Syntax of the Alloy kernel.

relation $r \in N$, there is a constant function symbol in KeYFOL associated with it, which we write as r'. Consequently, the set of these constants is named N'. The set of Alloy variables, represented by the syntactical category `var`, is called V_A.

The denotational semantics of the Alloy kernel are shown in Figure 4.3. The function E assigns relation values to expressions, and M boolean values to formulas. Both functions take a *binding* as a parameter. A binding b over some universe U is a function that assigns relation values to the relation names and variables: $b : N \cup V_A \to \mathfrak{P}(U^*)$. We use U^* to denote the set of all finite tuples (including the empty tuple) over some non-empty set U, and $\mathfrak{P}(U^*)$ is the powerset of U^*. Note that the semantics of the Alloy kernel do not require relations to be finite or uniform.

The binding is a total function and thus assigns relation values to all variables. The semantics of a particular formula or expression, however, only depend on the binding of those variables occurring free in it. It is straightforward, and therefore omitted here, to inductively define the set $F_{\mathtt{x}}$ of free variables in a formula or expression `x`.

The formalization of the translation in denotational style is shown in Figure 4.4. We inductively define two translation functions $\mathcal{E}$ and $\mathcal{C}$ to translate Alloy expressions and formulas, respectively. Both functions take a *variable mapping* as a parameter. A variable mapping m is a partial function from Alloy variables to the KeYFOL variables: $m : V_A \rightharpoonup V_{FO}$. We write $dom(m)$ to denote the domain of m.

The translation of expressions is straightforward using the operators defined in the previous section. Since the variable mapping m is partial, $\mathcal{E}[\![\mathtt{e}]\!]m$ is only well-defined if m is total on the free variables of the expression `e`, i.e. $F_{\mathtt{e}} \subseteq dom(m)$. The logical connectives offered by Alloy are all available in KeYFOL, so the definition of $\mathcal{C}$ for these cases is simple. The most interesting case of the definition is the quantification. Naturally, we translate the `all` and `some` quantifiers to universal and existential quantification, respectively. The quantification variable for the translation is required not to be in use yet[3]. The operator $\oplus$ denotes map update.

[3]recall that we assumed an infinite set of first-order variables V_{FO}

$M[\textbf{not } \mathtt{f}]b = \neg M[\mathtt{f}]b$
$M[\mathtt{f} \textbf{ and } \mathtt{g}]b = M[\mathtt{f}]b \wedge M[\mathtt{g}]b$
$M[\mathtt{f} \textbf{ or } \mathtt{g}]b = M[\mathtt{f}]b \vee M[\mathtt{g}]b$
$M[\mathtt{f} \Rightarrow \mathtt{g}]b = M[\mathtt{f}]b \Rightarrow M[\mathtt{g}]b$
$M[\textbf{all } \mathtt{x}\textbf{: } \mathtt{e} \textbf{ | } \mathtt{f}]b = \wedge\{M[\mathtt{f}](b \oplus x \mapsto v) \mid v \subseteq E[\mathtt{e}]b \wedge |v| = 1\}$
$M[\textbf{some } \mathtt{x}\textbf{: } \mathtt{e} \textbf{ | } \mathtt{f}]b = \vee\{M[\mathtt{f}](b \oplus x \mapsto v) \mid v \subseteq E[\mathtt{e}]b \wedge |v| = 1\}$
$M[\mathtt{p} \textbf{ in } \mathtt{q}]b = E[\mathtt{p}]b \subseteq E[\mathtt{q}]b$
$M[\mathtt{p} \textbf{ = } \mathtt{q}]b = E[\mathtt{p}]b = E[\mathtt{q}]b$

$E[\textbf{none}]b = \emptyset$
$E[\mathtt{p} + \mathtt{q}]b = E[\mathtt{p}]b \cup E[\mathtt{q}]b$
$E[\mathtt{p} \ \& \ \mathtt{q}]b = E[\mathtt{p}]b \cap E[\mathtt{q}]b$
$E[\mathtt{p} - \mathtt{q}]b = E[\mathtt{p}]b \setminus E[\mathtt{q}]b$
$E[\mathtt{p}.\mathtt{q}]b =$
$\quad \{\langle p_1, \ldots, p_{n-1}, q_2, \ldots, q_m\rangle \mid \langle p_1, \ldots, p_n\rangle \in E[\mathtt{p}]b \wedge \langle q_1, \ldots, q_m\rangle \in E[\mathtt{q}]b \wedge p_n = p_1\}$
$E[\mathtt{p} \rightarrow \mathtt{q}]b = \{\langle p_1, \ldots, p_n, q_1, \ldots, q_m\rangle \mid \langle p_1, \ldots, p_1\rangle \in E[\mathtt{p}]b \wedge \langle q_1, \ldots, q_m\rangle \in E[\mathtt{q}]b\}$
$E[\tilde{}\mathtt{p}]b = \{\langle p_2, p_1\rangle \mid \langle p_1, p_2\rangle \in E[\mathtt{p}]b\}$
$E[\hat{}\mathtt{p}]b = \{\langle x, y\rangle \mid \exists p_1, \ldots, p_n \mid \langle x, p_1\rangle, \langle p_1, p_2\rangle, \ldots, \langle p_n, y\rangle \in E[\mathtt{p}]b\}$
Variables: $E[\mathtt{x}]b = b(x)$
Relations: $E[\mathtt{r}]b = b(r)$

Figure 4.3: Semantics of the Alloy kernel, taken from [8] and [19].

Variables: $\mathcal{E}[\![\texttt{v}]\!]m = m(v)$
Relations: $\mathcal{E}[\![\texttt{r}]\!]m = r'$
$\mathcal{E}[\![\textbf{none}]\!]m = none'$
$\mathcal{E}[\![\texttt{e}_1 \texttt{->} \texttt{e}_2]\!]m = prod'(\mathcal{E}[\![\texttt{e}_1]\!]m, \mathcal{E}[\![\texttt{e}_2]\!]m)$
$\mathcal{E}[\![\texttt{e}_1\texttt{.}\texttt{e}_2]\!]m = join'(\mathcal{E}[\![\texttt{e}_1]\!]m, \mathcal{E}[\![\texttt{e}_2]\!]m)$
$\mathcal{E}[\![\texttt{e}_1\texttt{+}\ \texttt{e}_2]\!]m = union'(\mathcal{E}[\![\texttt{e}_1]\!]m, \mathcal{E}[\![\texttt{e}_2]\!]m)$
$\mathcal{E}[\![\texttt{e}_1\texttt{-}\ \texttt{e}_2]\!]m = diff'(\mathcal{E}[\![\texttt{e}_1]\!]m, \mathcal{E}[\![\texttt{e}_2]\!]m)$
$\mathcal{E}[\![\texttt{e}_1\texttt{\&}\ \texttt{e}_2]\!]m = inter'(\mathcal{E}[\![\texttt{e}_1]\!]m, \mathcal{E}[\![\texttt{e}_2]\!]m)$
$\mathcal{E}[\![\texttt{\~{}e}]\!]m = transp'(\mathcal{E}[\![\texttt{e}]\!]m)$
$\mathcal{E}[\![\texttt{\^{}e}]\!]m = transClos'(\mathcal{E}[\![\texttt{e}]\!]m)$

$\mathcal{C}[\![\texttt{e}_1\textbf{in}\ \texttt{e}_2]\!]m = subset'(\mathcal{E}[\![\texttt{e}_1]\!]m, \mathcal{E}[\![\texttt{e}_2]\!]m)$
$\mathcal{C}[\![\texttt{e}_1\texttt{=}\ \texttt{e}_2]\!]m = \mathcal{E}[\![\texttt{e}_1]\!]m \doteq \mathcal{E}[\![\texttt{e}_2]\!]m)$
$\mathcal{C}[\![\textbf{not}\ \texttt{c}]\!]m = \neg\mathcal{C}[\![\texttt{c}]\!]m$
$\mathcal{C}[\![\texttt{c}_1\textbf{and}\ \texttt{c}_2]\!]m = \mathcal{C}[\![\texttt{c}_1]\!]m \wedge \mathcal{C}[\![\texttt{c}_2]\!]m$
$\mathcal{C}[\![\texttt{c}_1\textbf{or}\ \texttt{c}_2]\!]m = \mathcal{C}[\![\texttt{c}_1]\!]m \vee \mathcal{C}[\![\texttt{c}_2]\!]m$
$\mathcal{C}[\![\textbf{all}\ \texttt{x}\texttt{:}\ \texttt{e}\ \texttt{|}\ \texttt{c}]\!]m = \forall y\texttt{:}\ Relation \mid one'(y) \wedge subset'(y, \mathcal{E}[\![\texttt{e}]\!]m) \Rightarrow \mathcal{C}[\![\texttt{c}]\!](m \oplus x \mapsto y)$
where $y \in V_{FO} \setminus m(V_A)$
$\mathcal{C}[\![\textbf{some}\ \texttt{x}\texttt{:}\ \texttt{e}\ \texttt{|}\ \texttt{c}]\!]m = \exists y\texttt{:}\ Relation \mid one'(y) \wedge subset'(y, \mathcal{E}[\![\texttt{e}]\!]m) \wedge \mathcal{C}[\![\texttt{c}]\!](m \oplus x \mapsto y)$
where $y \in V_{FO} \setminus m(V_A)$

Figure 4.4: Formalization of the translation of the Alloy kernel.

We mentioned in Section 3.6.1 that the case of a unary and singleton bounding expression is treated specially, and `all x: e | F` is translated to[4]

$$\forall x\texttt{:}\ Atom \mid in(x, e') \Rightarrow F'[x \leftarrowtail sin(x)]$$

This is not reflected by the formalization which translates the formula to

$$\forall x\texttt{:}\ Relation \mid one'(x) \wedge subset'(x, e') \Rightarrow F'$$

It can, however, be shown that this optimization is sound:

Theorem 2. Let ϕ be a formula and γ a term of type *Rel1* with no free occurrences of x. Then, the following formula holds:

$$\begin{aligned} &\forall x\texttt{:}\ Relation \mid (one'(x) \wedge subset'(x, \gamma) \Rightarrow \phi) \\ \Leftrightarrow\ &\forall x\texttt{:}\ Atom \mid (in(x, \gamma) \Rightarrow \phi[x \leftarrowtail sin(x)]) \end{aligned}$$

[4] like in Chapter 3, we denote the translations of `e` and `F` with e' and F', respectively, and assume that the Alloy variable `x` is translated to the logic variable x.

4.3 A Correctness Proof

The evaluation of an Alloy formula is performed in the context of a binding, while the evaluation of a KeYFOL formula is performed in the context of a first-order state. A KeYFOL formula ϕ is valid if it holds in every first-order state that satisfies the axioms, which we write as $\models \phi$. We say that an Alloy formula $\mathtt{f}$ is valid if it holds for all bindings, written as $\models \mathtt{f}$. The intuition of a correct translation is, that an Alloy formula is valid if its translation is valid. In this section, we prove that property for the translation of the Alloy kernel as formalized in the previous section.

To conduct a proof for the correctness property, we need to establish a relationship between first-order states and bindings. Let $b : N \cup V_A \to \mathfrak{P}(U^*)$ be some binding over some universe U. We construct first-order states S that agree with b: An Alloy formula holds in b iff its translation holds in S.

We define $\mathcal{S}_b$ to be the set of all first-order states $S = \langle \mathcal{D}, \delta, \mathcal{I} \rangle$ that satisfy the following constraints:

$$\mathcal{D}^{Relation} = \mathfrak{P}(U^*) \qquad \mathcal{D}^{Tuple} = U^* \qquad \mathcal{D}^{Atom} = \{\langle x \rangle \mid x \in U\} \qquad \mathcal{D}^{int} = \mathbb{Z}$$

$$in^{\mathcal{I}} = \{\langle t, r \rangle \in \mathcal{D}^{Tuple} \times \mathcal{D}^{Relation} \mid t \in r\} \tag{4.1}$$

$$proj^{\mathcal{I}}(\langle x_1, \ldots, x_n \rangle, i) = x_{i+1} \text{ for } \langle x_1, \ldots, x_n \rangle \in \mathcal{D}^{Tuple}, i \in 0 \ldots n-1 \tag{4.2}$$

$$ar^{\mathcal{I}}(\langle x_1, \ldots, x_n \rangle) = n \text{ for } \langle x_1, \ldots, x_n \rangle \in \mathcal{D}^{Tuple} \tag{4.3}$$

$$(r')^{\mathcal{I}} = b(r) \tag{4.4}$$

$$0^{\mathcal{I}} = 0 \qquad 1^{\mathcal{I}} = 1$$

$$-^{\mathcal{I}}(x, y) = x - y \text{ and } +^{\mathcal{I}}(x, y) = x + y \text{ for } x, y \in \mathcal{D}^{int}$$

$$\geq^{\mathcal{I}} = \{\langle x, y \rangle \in \mathcal{D}^{int} \times \mathcal{D}^{int} \mid x \geq y\} \tag{4.5}$$

Note that we fixed the interpretation of the *int* type and its constants and operators to exactly mirror the mathematical integers. We furthermore note that $\mathcal{S}_b$ is not empty.

It is easy to show that, in every $S \in \mathcal{S}_b$, the functions *drop*, *tail*, and *conc* have the intended semantics:

$$drop^{\mathcal{I}}(\langle x_1, \ldots, x_n \rangle) = \langle x_1, \ldots, x_{n-1} \rangle \tag{4.6}$$

$$tail^{\mathcal{I}}(\langle x_1, \ldots, x_n \rangle) = \langle x_2, \ldots, x_n \rangle \tag{4.7}$$

$$conc^{\mathcal{I}}(\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_m \rangle) = \langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle \tag{4.8}$$

We now prove that the first-order states in $\mathcal{S}_b$ agree with b in the evaluation of Alloy expressions. For this property to hold, we have to restrict the variable assignment performed by b.

Lemma 1. Let $b : N \cup V_A \to \mathfrak{P}(U^*)$ be a binding and $\mathtt{e}$ an Alloy expression. For every first-order state $S = \langle \mathcal{D}, \delta, \mathcal{I} \rangle \in \mathcal{S}_b$, assignment $\beta : V_{FO} \to \mathcal{D}$, and injective variable mapping $m : V_A \rightharpoonup V_{FO}$, if

$$b(x) = \beta(m(x))$$

for all $x \in dom(m)$ and $F_{\mathtt{e}} \subseteq dom(m)$, then

$$E[\![\mathtt{e}]\!]b = \mathcal{E}[\![\mathtt{e}]\!]m^{\mathcal{I},\beta}$$

Proof. The lemma is proven by induction on the syntax of the Alloy kernel. We show the most interesting cases here. The base case of a variable follows directly from the lemma's assumption and similarly the case of a relation name from 4.4:

$$E[\mathtt{v}]b = b(v) \overset{\text{Ass.}}{=} \beta(m(v)) = m(v)^{\mathcal{I},\beta} = \mathcal{E}[\![\mathtt{v}]\!]m^{\mathcal{I},\beta}$$
$$E[\mathtt{r}]b = b(r) \overset{4.4}{=} (r')^{\mathcal{I}} = (r')^{\mathcal{I},\beta} = \mathcal{E}[\![\mathtt{r}]\!]m^{\mathcal{I},\beta}$$

Join: For relations $R, S \subseteq U^*$, we first prove

$$\begin{aligned} join'^{\mathcal{I}}(R,S) = \{\langle x_1,\ldots,x_{n-1},y_2,\ldots,y_m\rangle \mid \langle x_1,\ldots,x_n\rangle \in R \\ \wedge \langle y_1,\ldots,y_m\rangle \in S \\ \wedge x_n = y_1\} \end{aligned} \tag{4.9}$$

Consider an element $t \in join'^{\mathcal{I}}(R,S)$. Let u be a variable of type *Tuple*, and r, s variables of type *Relation*. Then, for the assignment $\gamma = \beta^{t\,R\,S}_{u\,r\,s}$,

$$\begin{aligned} & S,\gamma \models in(u, join'(r,s)) \\ \overset{\text{Axiom}}{\iff} & S,\gamma \models \exists u,v\colon Tuple \mid in(u,r) \wedge in(v,s) \wedge conc(drop(u), tail(v)) \doteq w \\ & \qquad\qquad \wedge proj(u, ar(u)-1) \doteq proj(v,0) \\ \iff & S, \gamma^{\langle x_1,\ldots,x_n\rangle\,\langle y_1,\ldots,y_m\rangle}_{u\qquad\quad v} \models in(u,r) \wedge in(v,s) \\ & \qquad\qquad \wedge conc(drop(u), tail(v)) \doteq w \\ & \qquad\qquad \wedge proj(u, ar(u)-1) \doteq proj(v,0) \\ & \text{for some } \langle x_1,\ldots,x_n\rangle, \langle y_1,\ldots,y_m\rangle \in U^* \\ \iff & \langle x_1,\ldots,x_n\rangle \in R \text{ and } \langle y_1,\ldots,y_m\rangle \in S \\ & \text{and } conc^{\mathcal{I}}(drop^{\mathcal{I}}(\langle x_1,\ldots,x_n\rangle), tail^{\mathcal{I}}(\langle y_1,\ldots,y_m\rangle)) = t \\ & \text{and } proj^{\mathcal{I}}(\langle x_1,\ldots,x_n\rangle, ar^{\mathcal{I}}(\langle x_1,\ldots,x_n\rangle)-1) = proj^{\mathcal{I}}(\langle y_1,\ldots,y_m\rangle, 0) \\ \iff & \langle x_1,\ldots,x_n\rangle \in R \text{ and } \langle y_1,\ldots,y_m\rangle \in S \\ & \text{and } \langle x_1,\ldots,x_{n-1},y_2,\ldots,y_m\rangle = t \text{ and } x_n = y_1 \\ \iff & t \in \{\langle x_1,\ldots,x_{n-1},y_2,\ldots,y_m\rangle \mid \langle x_1,\ldots,x_n\rangle \in R \wedge \langle y_1,\ldots,y_m\rangle \in S \wedge x_n = y_1\} \end{aligned}$$

which shows the equality of both sets. From 4.9, we immediately infer

$$join'^{\mathcal{I}}(E[\mathtt{p}]b, E[\mathtt{q}]b) = E[\mathtt{p.q}]b$$

and use the induction hypothesis (IH) to conclude

$$\begin{aligned} \mathcal{E}[\![\mathtt{p.q}]\!]m^{\mathcal{I},\beta} &= join'(\mathcal{E}[\![\mathtt{p}]\!]m, \mathcal{E}[\![\mathtt{q}]\!]m)^{\mathcal{I},\beta} \\ &= join'^{\mathcal{I}}(\mathcal{E}[\![\mathtt{p}]\!]m^{\mathcal{I},\beta}, \mathcal{E}[\![\mathtt{q}]\!]m^{\mathcal{I},\beta}) \\ &\overset{\text{IH}}{=} join'^{\mathcal{I}}(E[\mathtt{p}]b, E[\mathtt{q}]b) = E[\mathtt{p.q}]b \end{aligned}$$

Transitive Closure: To prove the case of transitive closure, we need an inner induction on the natural numbers. For relations $R \subseteq U^*$, we define an iteration by

$$\begin{aligned} R^0 &= \{\langle x,y\rangle \in R\} \\ R^{n+1} &= \{\langle x,z\rangle \mid \exists y \mid \langle x,y\rangle \in R, \langle y,z\rangle \in R^n\} \text{ for all } n \in \mathbb{N}_0 \end{aligned}$$

and prove by induction on the natural numbers

$$iterJoin'^{\mathcal{I}}(R,n) = \bigcup_{i=0..n} R^i \tag{4.10}$$

For $n = 0$, the objective becomes

$$iterJoin'^{\mathcal{I}}(R,0) = R$$

Consider two variables u and r of type *Tuple* and *Relation*, respectively, and the assignment $\gamma = \beta^{t\,R}_{ur}$. The objective is then proven by:

$$\begin{aligned}
& t \in iterJoin'^{\mathcal{I}}(R,0) \\
\overset{4.1}{\iff}\ & S,\gamma \models in(u, iterJoin'(r,0)) \\
\overset{\text{Axiom}}{\iff}\ & S,\gamma \models in(u,r) \wedge ar(u) \doteq 2 \\
\overset{4.1}{\iff}\ & t \in R \text{ and } ar^{\mathcal{I}}(t) = 2 \\
\overset{4.3}{\iff}\ & t \in R \text{ and } t = \langle x,y\rangle \text{ for some } x,y \in U \\
\iff\ & t \in R^0
\end{aligned}$$

In the step case, we get the following objective:

$$iterJoin'^{\mathcal{I}}(R,n+1) = \bigcup_{i=0..n} R^i \cup R^{n+1}$$

Let now be u, r, and k be variables of type *Tuple*, *Relation*, and *int*, respectively, and $\gamma = \beta^{t\,R\,n}_{u\,r\,k}$. The objective is then proven by:

$$\begin{aligned}
& t \in iterJoin'^{\mathcal{I}}(R,n+1) \\
\overset{4.1}{\iff}\ & S,\gamma \models in(u, iterJoin'(r,k+1)) \\
\overset{\text{Axiom}}{\iff}\ & S,\gamma \models ar(u) \doteq 2 \wedge in(u, union'(iterJoin'(r,k), join'(r, iterJoin'(r,k)))) \\
\overset{4.3}{\iff}\ & t = \langle x,y\rangle \text{ for some } x,y \in U \\
& \text{and } S,\gamma \models in(u, union'(iterJoin'(r,k), join'(r, iterJoin'(r,k)))) \\
\overset{\text{Axiom}}{\iff}\ & t = \langle x,y\rangle \text{ for some } x,y \in U \\
& \text{and } S,\gamma \models in(u, iterJoin'(r,k)) \vee in(u, join'(r, iterJoin'(r,k))) \\
\iff\ & \langle x,y\rangle \in iterJoin'^{\mathcal{I}}(R,n) \text{ or } \langle x,y\rangle \in join'^{\mathcal{I}}(R, iterJoin'^{\mathcal{I}}(R,n)) \\
\overset{\text{IH}}{\iff}\ & \langle x,y\rangle \in \cup_{i=0..n} R^i \text{ or } \langle x,y\rangle \in join'^{\mathcal{I}}(R, iterJoin'^{\mathcal{I}}(R,n)) \\
\overset{4.9}{\iff}\ & \langle x,y\rangle \in \cup_{i=0..n} R^i \\
& \text{or } \langle x,y\rangle \in \{\langle x_1,\ldots,x_{n-1},y_2,\ldots,y_m\rangle \mid \langle x_1,\ldots,x_n\rangle \in R \\
& \qquad\qquad \wedge \langle y_1,\ldots,y_m\rangle \in iterJoin'^{\mathcal{I}}(R,n) \\
& \qquad\qquad \wedge x_n = y_1\} \\
\iff\ & \langle x,y\rangle \in \cup_{i=0..n} R^i \\
& \text{or } \langle x,z\rangle \in R \wedge \langle z,y\rangle \in iterJoin'^{\mathcal{I}}(R,n) \text{ for some } z \in U
\end{aligned}$$

$$
\begin{aligned}
&\overset{\text{IH}}{\iff} \langle x, y\rangle \in \cup_{i=0..n} R^i \\
&\quad \text{or } \langle x, z\rangle \in R \text{ and } \langle z, y\rangle \in \cup_{i=0..n} R^i \text{ for some } z \in U \\
&\iff \langle x, y\rangle \in \cup_{i=0..n} R^i \\
&\quad \text{or } \langle x, z\rangle \in R \text{ and } \langle z, y\rangle \in R^j \text{ for some } z \in U, 0 \geq j \geq n \\
&\iff \langle x, y\rangle \in \cup_{i=0..n} R^i \\
&\quad \text{or } \langle x, y\rangle \in R^{j+1} \text{ for some } 0 \geq j \geq n \\
&\iff \langle x, y\rangle = t \in \bigcup_{i=0..n} R^i \cup R^{n+1}
\end{aligned}
$$

We now use 4.10 to prove the objective of the outer induction. It is easy to see that

$$E[\hat{}\mathtt{p}]b = \bigcup_{i\in\mathbb{N}_0} (E[\mathtt{p}]b)^i \tag{4.11}$$

Let u and k be logic variables of type *Tuple* and *int*, respectively, that do not occur free in $\mathcal{E}[\![\mathtt{p}]\!]m$. We then prove the equality $E[\hat{}\mathtt{p}]b = \mathcal{E}[\![\hat{}\mathtt{p}]\!]m^{\mathcal{I},\beta}$ by

$$
\begin{aligned}
& t \in E[\hat{}\mathtt{p}]b \\
&\overset{4.11}{\iff} t \in \bigcup_{i\in\mathbb{N}_0} (E[\mathtt{p}]b)^i \\
&\iff t \in \bigcup_{i=0..n} (E[\mathtt{p}]b)^i \text{ for some } n \in \mathbb{N}_0 \\
&\overset{4.10}{\iff} t \in \mathit{iterJoin}'^{\mathcal{I}}(E[\mathtt{p}]b, n) \text{ for some } n \in \mathbb{N}_0 \\
&\overset{\text{IH}}{\iff} t \in \mathit{iterJoin}'^{\mathcal{I}}(\mathcal{E}[\![\mathtt{p}]\!]m^{\mathcal{I},\beta}, n) \text{ for some } n \in \mathbb{N}_0 \\
&\overset{4.1}{\iff} S, \beta^{t\,n}_{uk} \models \mathit{in}(u, \mathit{iterJoin}'(\mathcal{E}[\![\mathtt{p}]\!]m, k)) \text{ for some } n \in \mathbb{N}_0 \\
&\overset{4.5}{\iff} S, \beta^{t\,n}_{uk} \models k \geq 0 \wedge \mathit{in}(u, \mathit{iterJoin}'(\mathcal{E}[\![\mathtt{p}]\!]m, k)) \text{ for some } n \in \mathcal{D}^{\mathit{int}} \\
&\iff S, \beta^t_u \models \exists k\colon \mathit{int} \mid k \geq 0 \wedge \mathit{in}(u, \mathit{iterJoin}'(\mathcal{E}[\![\mathtt{p}]\!]m, k)) \\
&\overset{\text{Axiom}}{\iff} S, \beta^t_u \models \mathit{in}(u, \mathit{transClos}'(\mathcal{E}[\![\mathtt{p}]\!]m)) \\
&\overset{4.1}{\iff} t \in \mathcal{E}[\![\hat{}\mathtt{p}]\!]m^{\mathcal{I},\beta}
\end{aligned}
$$

□

With Lemma 1 proven, we can show the equivalence between Alloy formulas and its translation.

Theorem 3. Let $b : N \cup V_A \to \mathfrak{P}(U^*)$ be a binding and $\mathtt{f}$ an Alloy formula. For every first-order state $S = \langle \mathcal{D}, \delta, \mathcal{I}\rangle \in \mathcal{S}_b$, assignment $\beta : V_{FO} \to \mathcal{D}$, and injective variable mapping $m : V_A \rightharpoonup V_{FO}$, if

$$b(x) = \beta(m(x))$$

for all $x \in \mathit{dom}(m)$ and $F_{\mathtt{f}} \subseteq \mathit{dom}(m)$, then

$$S, \beta \models \mathcal{C}[\![\mathtt{f}]\!]m \text{ iff } M[\mathtt{f}]b$$

Proof. We again use induction on the syntax of the Alloy kernel to proof the theorem. The cases of the logical connectives are trivial.

Subset: The case of the subset operator `in` is proven using Lemma 1:

$$\begin{aligned}
& S,\beta \models \mathcal{C}[\![\mathsf{e}_1 \ \mathtt{in}\ \mathsf{e}_2]\!]m \\
\iff & S,\beta \models \mathit{subset}'(\mathcal{E}[\![\mathsf{e}_1]\!]m, \mathcal{E}[\![\mathsf{e}_2]\!]m) \\
\overset{\text{Axiom}}{\iff} & S,\beta \models \forall u\text{: } \mathit{Tuple} \mid \mathit{in}(u, \mathcal{E}[\![\mathsf{e}_1]\!]m) \Rightarrow \mathit{in}(u, \mathcal{E}[\![\mathsf{e}_2]\!]m) \\
\iff & S,\beta^t_u \models \mathit{in}(u, \mathcal{E}[\![\mathsf{e}_1]\!]m) \Rightarrow \mathit{in}(u, \mathcal{E}[\![\mathsf{e}_2]\!]m) \text{ for all } t \in U^* \\
\iff & \text{if } t \in \mathcal{E}[\![\mathsf{e}_1]\!]m^{\mathcal{I},\beta} \text{ then } t \in \mathcal{E}[\![\mathsf{e}_2]\!]m^{\mathcal{I},\beta} \text{ for all } t \in U^* \\
\overset{\text{Lem. 1}}{\iff} & \text{if } t \in E[\mathsf{e}_1]b \text{ then } t \in E[\mathsf{e}_2]b \text{ for all } t \in U^* \\
\iff & E[\mathsf{e}_1]b \subseteq E[\mathsf{e}_2]b \\
\iff & M[\mathsf{e}_1 \ \mathtt{in}\ \mathsf{e}_2]b
\end{aligned}$$

Since the equality operator is just a shorthand for mutual inclusion, the proof for this case is analogous to the case of the `in` operator.

Quantification: We first state two auxiliary lemmas. Let y be a variable that is not free in $\mathcal{E}[\![\mathsf{e}]\!]m$ and $v \subseteq U^*$ a relation. Then,

$$S,\beta^v_y \models \mathit{one}'(y) \iff |v| = 1 \tag{4.12}$$

$$S,\beta^v_y \models \mathit{subset}'(y, \mathcal{E}[\![\mathsf{e}]\!]m) \iff v \subseteq E[\mathsf{e}]b \tag{4.13}$$

The first one can easily be shown by contradiction, and 4.13 can be proven using Lemma 1. We now infer:

$$\begin{aligned}
& S,\beta \models \mathcal{C}[\![\mathtt{all\ x}\text{: e | f}]\!]m \\
\iff & S,\beta \models \forall y\text{: } \mathit{Relation} \mid \mathit{one}'(y) \wedge \mathit{subset}'(y, \mathcal{E}[\![\mathsf{e}]\!]m) \Rightarrow \mathcal{C}[\![\mathsf{f}]\!](m \oplus x \mapsto y) \\
& \text{for some } y \notin m(V_A) \\
\iff & \text{if } S,\beta^v_y \models \mathit{one}'(n) \text{ and } S,\beta^v_y \models \mathit{subset}'(y, \mathcal{E}[\![\mathsf{e}]\!]m) \\
& \text{then } S,\beta^v_y \models \mathcal{C}[\![\mathsf{f}]\!](m \oplus x \mapsto y) \text{ for all } v \subseteq U^*
\end{aligned}$$

From $y \notin m(V_A)$, we see that y is not free in $\mathcal{E}[\![\mathsf{e}]\!]m$. Thus, 4.13 is applicable and we further infer

$$\begin{aligned}
& \text{if } |v| = 1 \text{ and } v \subseteq E[\mathsf{e}]b \\
& \text{then } S,\beta^v_y \models \mathcal{C}[\![\mathsf{f}]\!](m \oplus x \mapsto y) \text{ for all } v \subseteq U^*
\end{aligned} \tag{4.14}$$

Since m is an injective variable mapping and $y \notin m(V_A)$, $m \oplus x \mapsto y$ is also injective. We furthermore see that $\mathcal{S}_{b \oplus x \mapsto v} = \mathcal{S}_b$. The induction hypothesis thus yields

$$S,\beta^v_y \models \mathcal{C}[\![\mathsf{f}]\!](m \oplus x \mapsto y) \iff M[\mathsf{f}](b \oplus x \mapsto v)$$

which we use to infer from 4.14:

$$\begin{aligned}
& \text{if } |v| = 1 \text{ and } v \subseteq E[\mathsf{e}]b \\
& \text{then } M[\mathsf{f}](b \oplus x \mapsto v) \text{ for all } v \subseteq U^* \\
\iff & \wedge \{M[\mathsf{f}](b \oplus x \mapsto v) \mid v \subseteq E[\mathsf{e}]b \wedge |v| = 1\} \\
\iff & M[\mathtt{all\ x}\text{: e | f}]b
\end{aligned}$$

The proof for existential quantification is similar. □

A closed Alloy formula is translated to a closed formula in KeYFOL and is thus independent from the variable mapping. If we use the empty mapping ε for the translation, the premisses of Theorem 3 are always satisfied. We formulate this as a corollary.

Corrolary 1. For a closed Alloy formula $\mathtt{f}$, a binding $b : N \cup V_A \to \mathfrak{P}(U^*)$, and a first-order state $S \in \mathcal{S}_b$

$$S \models \mathcal{C}[\![\mathtt{f}]\!]\varepsilon \text{ iff } M[\mathtt{f}]b$$

From Corollary 1, we ultimately prove the correctness theorem:

Theorem 4 (Correctness Theorem)**.** A closed Alloy formula $\mathtt{f}$ is valid if its translation is a valid KeYFOL formula.

Proof. If the theorem did not hold, there was a closed Alloy formula $\mathtt{f}$ such that $\mathtt{f}$ is not valid but its translation is: $\models \mathcal{C}[\![\mathtt{f}]\!]\varepsilon$. Since $\mathtt{f}$ is not valid, there is a binding b in which $\mathtt{f}$ does not hold: $M[\mathtt{f}]b = \bot$. Using of Corollary 1, we can thus construct a first-order state $S \in \mathcal{S}_b$ such that $S \not\models \mathcal{C}[\![\mathtt{f}]\!]\varepsilon$ hence contradicting the assumption that the translation of $\mathtt{f}$ is valid. □

Theorem 4 states the soundness of our translation of the Alloy kernel, but it says nothing about its completeness. It does therefore not obviate that some closed and valid Alloy formula translates to an KeYFOL formula that is not valid. In fact, due to the built-in transitive closure operator of Alloy, our translation cannot be complete [24].

The correctness proof conducted in this section is based on different operators than used for the translation in Chapter 3. However, we saw in Section 4.1 that the new operators agree with the old ones but are defined for a larger domain, namely the types *Relation* and *Tuple* rather than *Rel2* and *Tuple*3, et cetera. When we translate well-typed Alloy models, the arity of the tuples in a relation is always uniform and the arity of every expression is statically determined by Alloy's type system. Thus, the arity-capturing types are applicable and the broader domain of the new operators is not significant to the validity of formulas. The findings of this section therefore also apply to the translation of the kernel as described in Chapter 3.

4.4 Model Correctness

The Alloy kernel that was considered in the previous sections is a subset of Alloy's formulas and omits several features. Furthermore, the full Alloy language enriches the logic with declarations such as signatures and fields, and embeds formulas as facts and assertions.

The signature and field declarations restrict the admissible instances of a model. A model is considered correct if the assertions hold in any admissible instance that satisfies the facts. We addressed this in Chapter 3 by generating model constraints and including these in the proof obligation, together with the translation of the facts and assertions. When the translation of formulas is correct in the sense of Theorem 4 and the model constraints correctly identify the admissible instances, it is easy to see that a valid proof obligation implies the correctness of the model. In

this setting, where Theorem 4 holds, when a model has been proven correct by the KeY system, the bounded analysis performed by the Alloy Analyzer will not find any counterexample, regardless of the defined scope. The reverse, however, is not true because Kelloy does not restrict the model's instances to be finite. Consider, for example, the following model:

```
sig A {
  foo: A }

fact {
  some A
  foo in A some→ A }

assert {
  some a: A | a in a.^foo }
```

The fact states that `A` is not empty and the functional relation `foo` is surjective. The assertion that `foo` is a cyclic relation holds in all finite instances. If `A` was infinite, however, the assertion can be violated when `foo` represents an infinite chain.

We observe that the correctness of Theorem 4 is rooted in the fact that the semantics of the Alloy kernel does – in contrast to the Alloy Analyzer – not impose the finiteness of relations. Most of Alloy's features can be desugared to the kernel and it is therefore not a theoretical challenge to expand the theorem to cover also these features. However, there are features whose the semantics cannot be directly adopted to infinite instances.

The most natural of such features is the cardinality operator, which is only evaluated if the argument is guaranteed to be finite and unspecified otherwise (see Section 3.8.1). A model might be proven correct although it uses the cardinality operator for a potentially infinite relation. Since the operator is unspecified in that case, it is ensured that the model is in fact correct for all finite and infinite instances. It can, however, be intentional that parts of a model are always finite. For example, it might be safe to assume a finite number of clients in the model of a network protocol. In such cases, the user can choose to explicitly finitize signatures at translation time. This should be done with care, since it can make the model inconsistent. For example, finitizing the `Int` signature (possibly indirectly), contradicts with its axiomatization and the model might thus be unintentionally proven correct.

Integer expressions are analyzed with respect to a finite bitwidth by the Alloy Analyzer and are thus subject to overflow. This can lead to the somehow awkward behavior that an increased scope vanishes counterexamples. Overflow is usually not intended for the verification of a model, so we use mathematical integers for the translation of integer expressions. In the presence of integer expressions, the Alloy Analyzer might therefore produce a counterexample for a verified model. This counterexample, however, is then due to overflow and disappears when choosing a sufficiently large bitwidth.

We mentioned in Section 3.6.1 that the Alloy language allows quantification over arbitrary relations, but the Alloy Analyzer might not be able to analyze such models. Since relations are first-order citizens in our translation, Kelloy can potentially prove models correct that the Alloy Analyzer does not handle.

In Section 3.8.2 we extended the meaning of the ordering module to infinite sets in a natural way. In a proof, however, we might need to separate the infinite and the finite cases. There are, for example, two different induction principles. In the case of a finite set being linearly ordered, Kelloy conforms to the Alloy Analyzer.

5. Reasoning

Once that an Alloy model is successfully translated, it should be proven correct. This chapter presents how reasoning about Alloy models is performed within the KeY system.

The KeY system can verify the correctness of Alloy models, but it is, in general, not capable of refuting incorrect ones. Therefore, we suggest that a model is tested extensively using automatic tools like the Alloy Analyzer before it is handed to the interactive prover. We can thus presume that the user is confident about the model's correctness and, more importantly, has some understanding why it is correct. For the verification task, these correctness arguments have to be formalized in the KeY system in order to conduct a proof. To keep this step simple, we follow the design principle that the reasoning should be performed on a similar abstraction level as the modeling. Due to the relational theory introduced in Chapter 3, Alloy formulas and expressions are translated in a structure preserving way.

Using solely the axioms of the relational theory, we can eliminate all relational operators from a formula by rewriting it to an equivalent one, in which only the uninterpreted symbols (i.e. the membership predicate and the constructor functions) appear. While this might be appropriate for purely automatic approaches, we consider this not suitable for interactive proving since it breaks the correspondence between the original model and its translation, hence violating the design principle stated above. Moreover, this approach tends to produce large proof sequents that contain a lot of quantifiers which are difficult to handle efficiently, and KeY achieves only poor automation.

Consider for example the formula $union_1(r, r) \doteq s$ appearing in the antecedent. The union operator is idempotent and the term on the left-hand side should be simplified to r. The equation could then be used to replace occurrences of r with s throughout the sequent. However, since the union operator is defined through the membership predicate, the axioms cannot be applied here without rewriting the equation first. Evidently, the theory's axioms alone are not suitable for reasoning on the abstraction level of relations. We therefore define a set of deduction rules to provide us with the necessary abstraction. All rules that are shown here are lemmas and have been proven to follow from the axioms.

The next sections present a reasoning strategy for proving Alloy models. Reading them should enable the user to efficiently conduct proofs for Alloy models with KeY. They give an overview of the strategy, rather than an exhaustive definition. Section 5.1 outlines the general approach and presents the deduction rules the reasoning is based on. Sections 5.2 to 5.4 extend the approach for the more special features of Alloy. The strategy has been implemented to be applied automatically by the KeY system, but we omit the technical details for that.

Compared to a simple automation strategy that only applies the axioms from the relational theory, the strategy presented here achieves a much higher degree of automation: for one of the simplest models that is distributed with Alloy, the birthday book example, our strategy found a proof after 82 rule applications. The simple strategy needed 680 steps. Another rather simple example is the "Self-Grandpa" example from [19]. While our strategy finished the proof in 718 steps, the simple strategy did not succeed to prove the model in a reasonable amount of time.

5.1 Deduction Rules

The reasoning strategy we develop here is integrated in KeY's proof search strategy for KeYFOL[1]. We do therefore not consider the logical connectives and quantifiers, but are only interested in the Alloy specific parts.

5.1.1 Predicates

The Alloy specific formulas that we are interested in, are composed of a predicate from the relational theory. The predicate *one* is defined using *lone* and *some*. Its treatment is therefore redundant and the presentation hence omitted.

The semantics of most predicates is defined by universally quantified formulas (the *some* predicate is an exception that is discussed later). Whether it is desirable to expand these predicates to their definitions depends on their occurrence in the sequent. Universal quantification in the succedent can be eliminated by skolemization using the δ-rules[2] of the calculus. It is therefore suitable to rewrite predicate invocations on the right-hand side of the sequent to their definitions.

Contrary to that, universally quantified formulas in the antecedent need to be instantiated. Since providing a suitable instantiation automatically is a complex and heuristic task, expanding predicates on the left-hand side of the sequent is not desirable and the axioms should not be applied. Instead, we define rules to exploit the semantics of the predicate occurring in the antecedent, without rewriting it. For the predicates *subset*, *disj*, and *lone*, such rules are:

$$\mathsf{useSubsetTrue}\ \frac{subset(r,s) \quad in(t,r)}{in(t,s)} \qquad \mathsf{useSubsetFalse}\ \frac{subset(r,s) \quad \neg in(t,s)}{\neg in(t,r)}$$

$$\mathsf{useDisj1}\ \frac{disj(r,s) \quad in(t,r)}{\neg in(t,s)} \qquad \mathsf{useDisj2}\ \frac{disj(r,s) \quad in(t,s)}{\neg in(t,r)}$$

$$\mathsf{replWithSin}\ \frac{lone(r) \quad in(t,r)}{r \rightsquigarrow sin(t)}$$

[1] actually, JavaDL which is a superset of KeYFOL

[2] δ-rules replace a quantification with its body while substituting the quantification variable with a *skolem term*: a previously unused constant.

1 $subset(A, B)$,
2 $lone(B)$,
3 $in(x, A)$,
4 $in(y, B)$,
5 $in(z, C)$
$\vdash$
$some(C)$

Figure 5.1: An example proof sequent.

When applying the replWithSin rule, we have to ensure that the replacement is not performed within the premisses since that can be destructive: the application might result in a sequent that is not valid although the preceding sequent was.

In a nutshell, we rewrite predicates on the right-hand side of the sequent, but leave them on the left-hand side. This approach is implemented by defining a rewrite rule to match only in the succedent, while the axioms of the predicates are not to be applied.

As mentioned before, the *some* predicate is an exception since its semantics is defined by existential quantification. We therefore rewrite it in the antecedent, since the quantifier is then eliminated by skolemization. When the *some* predicate appears in the succedent (and is thus assumed to be false), we can conclude that the relational argument is empty, hence add equality to the empty relation to the left-hand side of the sequent. In the unary case, the rule for this is

$$\text{someRight}\ \frac{\Gamma,\ r^{(1)} \doteq none \vdash \Delta}{\Gamma \vdash some(r^{(1)}),\ \Delta}$$

The KeY system uses equations in the antecedent to replace occurrences of the left-hand term anywhere else in the sequent. For this purpose, the equation is ordered so that the simpler term (determined by a lexicographical order) is to the right of the equality sign. This behavior is particularly useful when the right-hand side of the equation is a constant, like the equality to an empty relation above. However, equality of two relations has the same semantics as their mutual inclusion. We also want to leverage this like it is done by the useSubsetTrue and useSubsetFalse rules. We therefore rewrite equalities to two *subset* invocations, and dynamically add a rewrite rule to maintain the replacement facility. Obviously, this rewrite rule should not be applied within the newly introduced formulas.

To illustrate the reasoning approach, we apply some of the rules to the example proof sequent shown in Figure 5.1: from lines 1 and 3, we can infer $in(x, B)$ using the useSubsetTrue rule. Together with line 2, replWithSin lets us replace B with $sin(x)$ in line 4, thus inferring $x \doteq y$. The someRight rule adds $C \doteq none$ to the antecedent, which is then rewritten to $subset(C, none)$ and $subset(none, C)$.[3] At this step, the rewrite rule $C \rightsquigarrow none$ is dynamically created and can be used to replace

[3]Section 5.1.3 introduces simplification rules for such trivial predicate invocations.

$$union_2(r^{(2)}, r^{(2)}) \rightsquigarrow r^{(2)} \qquad join_{1\times2}(r^{(1)}, iden) \rightsquigarrow r^{(1)}$$
$$inter_1(none, r^{(1)}) \rightsquigarrow none \qquad lone(none) \rightsquigarrow \mathbf{true}$$
$$subset(sin(a), r) \rightsquigarrow in(a, r) \qquad \frac{subset(r, s)}{union_1(r, s) \rightsquigarrow s}$$
$$join_{1\times2}(r^{(1)}, transp(s^{(2)})) \rightsquigarrow join_{2\times1}(s^{(2)}, r^{(1)})$$

Figure 5.2: Selected simplification rules.

C with *none* in line 5, which can then be replaced with **false** by the definition of *none*. Since **false** now appears in the antecedent, the sequent has been proven valid.

5.1.2 Tuple Terms

The value of every Alloy expression is a relation. There is no notion for the elements of relations. Contrary to that, tuples are explicit in the translation. Tuple terms (i.e., terms of type *Atom*, *Tuple2*, etc.) that appear in the sequent are variables, constants, and constructor invocations. Constants usually appear as skolem terms when a δ-rule is applied.

Axioms that require access to the components of a tuple match on constructor invocations rather than general terms. It is therefore not desirable to have terms of type *Tuple2*, *Tuple3*, etc. in the sequent that are not constructor invocations. We therefore carefully define all the rules to prevent this.[4] For example, rather than using quantification over *Tuple2* to rewrite a *subset* invocation for two binary relations in the succedent, we use quantification over *Atom* twice:

$$\frac{\Gamma \vdash \forall a, b\colon Atom \mid in(binary(a, b), r^{(2)}) \Rightarrow in(binary(a, b), s^{(2)}), \Delta}{\Gamma \vdash subset(r^{(2)}, s^{(2)}), \Delta}$$

We can therefore presume that tuple terms appearing in the sequent are constants or variables of type *Atom*, or constructor invocations. We say that these tuple terms are in *normal form*.

5.1.3 Relational Terms

The relational terms appearing in a proof sequent are composed of the theory's operators and constants. We introduce a number of rules to efficiently support the relational operators. The treatment of transitive closure, however, is postponed to Section 5.2.

In some cases, relational terms can be simplified. Figure 5.2 shows representatives of simplification rules to be applied greedily. Similarly, formulas can sometimes be rewritten to simpler ones. The figure also shows some rules for this.

When discussing the treatment of predicates in Section 5.1.1, we did not examine the membership predicate. Following the previous section, the first argument to an invocation of *in* is a tuple term in normal form. The axioms for the top-level operator

[4] We can of course not prevent the user from introducing such terms interactively.

of the relational argument are thus applicable, and all operators can be eliminated. The resulting formula is composed of *elementary memberships*: invocations of the membership predicates where the first argument is a tuple term in normal form, and the relational argument is a constant or a variable. For example, the term

$$in(b, inter_1(join_{1\times2}(sin(a), foo), bar))$$

is rewritten by the axioms to

$$in(binary(a, b), foo) \wedge in(b, bar)$$

In Section 5.1.1 we introduced rules to capture the semantics of predicates appearing in the antecedent, that match on membership invocations. Since these membership invocations are subject to rewriting, the rules might not be applicable. For example, in a sequent

$$subset(diff_1(A, B), C),\ in(a, A) \vdash in(a, B)$$

we want to infer $in(a, C)$, but the useSubsetTrue rule can not be applied. To overcome this problem, we define rules that match on predicate/operator combinations and their elementary membership representation, for example

$$\textsf{useSubsetTrueDiff}\ \frac{subset(diff_1(r^{(1)}, s^{(1)}), t^{(1)}) \quad in(a, r^{(1)}) \quad \neg in(a, s^{(1)})}{in(a, t^{(1)})}$$

Using the useSubsetTrueDiff rule, we can now infer $in(a, C)$ as intended.

The operators in a relational term may be nested, but the rules that we just discussed only match on the top-level operator of the predicate's arguments. For nested operators, the same problem as before arises again. Apparently, this argument can be applied to an arbitrary depth of operator nesting, so no set of rules can ever handle all possible cases. However, we rarely observe deeply nested operators. In those cases where the lemmas do not suffice, the predicate should be expanded to its definition by user interaction.

Alloy models frequently talk about the value of a field for a single element of a signature, which is expressed by joining a field with a singleton set. Thus, terms like $join_{1\times2}(sin(x), foo)$ and $join_{1\times2}(sin(y), join_{1\times3}(sin(x), bar)$ appear frequently and should therefore be handled efficiently. We address this by introducing further lemmas like the following:

$$\frac{subset(join_{1\times2}(sin(b), join_{1\times3}(sin(a), r^{(3)})), s^{(1)}) \quad in(ternary(a, b, c), r^{(3)})}{in(c, s^{(1)})}$$

5.2 Transitive Closure

Transitive closure of a binary relation cannot be expressed in first-order languages [22]. Consequently, a universally efficient reasoning strategy is not achievable. Many models, however, only rely on certain properties of transitive closure which are expressible in first-order logic. In this section, we define a set of rules to capture such properties. While some of them are suitable for automatic application, others require the user to provide an instantiation.

The problem of reasoning about transitive closure in first-order languages has been tackled in several ways. Some approaches are only applicable in a limited context: Nelson [25] addresses the transitive closure of functional relations, while Dong and Su [7] maintain transitive closure under unit changes to a relation. A more general approach is proposed by Lev-Ami et al. [24]. Similar to our approach, several first-order axioms capturing essential properties of transitive closure are defined.

The rules that we introduce here are lemmas and have been proven using the induction principle for KeY's integers. Like the other relational operators, transitive closure can be simplified in some cases. We therefore introduce simplification rules like the following, that are applied greedily:

$$\frac{lone(r^{(2)})}{transClos(r^{(2)}) \rightsquigarrow r^{(2)}} \qquad transClos(transp(r^{(2)})) \rightsquigarrow transp(transClos(r^{(2)}))$$

We furthermore introduce rules to capture some simple properties about transitive closure: (1) Transitive closure is transitive, (2) the transitive closure of a relation is a superset of that relation, and (3) the transitive closure of a subset of some relation is a subset of the transitive closure of that relation, and (4) an element in the domain of the transitive closure of a relation is also in the domain of that relation (and analogous for the range). These properties are implemented by the following rules:

$$\frac{in(binary(a,b), transClos(r^{(2)})) \quad in(binary(b,c), transClos(r^{(2)}))}{in(binary(a,c), transClos(r^{(2)}))}$$

$$\frac{}{\Gamma, in(t^{(2)}, r^{(2)}) \vdash in(t^{(2)}, transClos(r^{(2)})), \Delta}$$

$$\frac{subset(r^{(2)}, s^{(2)})}{subset(transClos(r^{(2)}), transClos(s^{(2)}))}$$

$$\frac{in(binary(a,b), transClos(r^{(2)}))}{\exists c\colon Atom \mid in(binary(a,c), r^{(2)})} \qquad \frac{in(binary(a,b), transClos(r^{(2)}))}{\exists c\colon Atom \mid in(binary(c,b), r^{(2)})}$$

While these rules suffice in some cases, they are usually not capable of proving functionally complex properties about a model involving transitive closure. For such cases, we introduce two lemmas for interactive application. The first one is an induction principle:

$$\frac{\begin{array}{l}\forall a,b\colon Atom \mid in(binary(a,b), r^{(2)}) \Rightarrow \phi(a,b) \\ \forall a,b,c\colon Atom \mid in(binary(a,b), r^{(2)}) \wedge in(binary(b,c), transClos(r^{(2)})) \wedge \phi(b,c) \\ \qquad\qquad \Rightarrow \phi(a,c)\end{array}}{\forall a,b\colon Atom \mid in(binary(a,b), transClos(r^{(2)})) \Rightarrow \phi(a,b)}$$

To apply this rule, the user has to provide an instantiation for the parameterized formula ϕ. The rule can then be used to show $\phi(a,b)$ for every a and b that are connected in $r^{(2)}$ by proving a base and a step case. The second rule is one of the coloring axioms from [24], namely the NoExit axiom:

$$\frac{\forall a,b\colon Atom \mid \phi(a) \wedge \neg\phi(b) \Rightarrow \neg in(binary(a,b), r^{(2)})}{\forall a,b\colon Atom \mid \phi(a) \wedge \neg\phi(b) \Rightarrow \neg in(binary(a,b), transClos(r^{(2)}))}$$

Again, the parameterized formula ϕ has to be instantiated interactively. The intuition of the rule is the following: when the class of nodes for which ϕ holds is never left by an edge of $r^{(2)}$, then no path in $r^{(2)}$ leaves this class.

$$\mathit{finite}(\mathit{union}_1(r^{(1)}, s^{(1)})) \rightsquigarrow \mathit{finite}(r^{(1)}) \wedge \mathit{finite}(s^{(1)})$$

$$\frac{\mathit{finite}(r^{(1)})}{\mathit{finite}(\mathit{diff}_1(r^{(1)}, s^{(1)})) \rightsquigarrow \mathbf{true}}$$

$$\mathit{finite}(\mathit{none}) \rightsquigarrow \mathbf{true} \qquad \mathit{finite}(\mathit{sin}(a)) \rightsquigarrow \mathbf{true}$$

$$\frac{\mathit{lone}(r)}{\mathit{finite}(r) \rightsquigarrow \mathbf{true}} \qquad \frac{\mathit{subset}(r, s) \quad \mathit{finite}(s)}{\mathit{finite}(r) \rightsquigarrow \mathbf{true}}$$

$$\mathit{card}(\mathit{none}) \rightsquigarrow 0$$

$$\mathit{card}(\mathit{sin}(a)) \rightsquigarrow 1$$

$$\frac{\mathit{finite}(r^{(1)}) \quad \mathit{finite}(s^{(1)})}{\mathit{card}(\mathit{union}_1(r^{(1)}, s^{(1)})) \rightsquigarrow \mathit{card}(r^{(1)}) + \mathit{card}(s^{(1)}) - \mathit{card}(\mathit{inter}_1(r^{(1)}, s^{(1)}))}$$

$$\frac{\mathit{finite}(r^{(1)}) \quad \mathit{finite}(s^{(1)})}{\mathit{card}(\mathit{diff}_1(r^{(1)}, s^{(1)})) \rightsquigarrow \mathit{card}(r^{(1)}) - \mathit{card}(\mathit{inter}_1(r^{(1)}, s^{(1)}))}$$

Figure 5.3: Inference rules the finiteness and cardinality.

5.3 Finiteness and Cardinality

The cardinality operator is defined for every relation that can be shown to be finite. The *card* function and the *finite* predicate can be applied to arbitrary relational terms. However, if the finiteness flag has not been shown to hold, the cardinality function is unspecified. We therefore define a set of rules to infer the finiteness flag for relational terms. Some representatives are shown in Figure 5.3. This inference system can be used to show that the value of relational terms is guaranteed to be finite. However, the inference system is not complete, that is, it may not be capable to infer the finiteness flag for an in fact finite relation. Akin to the inference system for the finiteness flag, we define a set of inference rules for the cardinality operator. Representatives of these are also shown in the figure.

Our inference systems for finiteness and cardinality are very similar to those implemented in the Rodin tool [26] for the Event-B language [1]. The inference is straightforward and intuitive, but incomplete. Verifying Alloy models that use the cardinality operator might thus require explicit finitization. Kelloy therefore allows the user to finitize a signature at translation time. This feature, however, has to be used with care since it can make the model inconsistent (see Section 4.4).

When the finiteness flag was inferred for some relation, all of its tuples are associated with an integer value by the enumerator (e.g. elem_1, see Section 3.8.1). A tuple of a finite relation can thus have two representations in the sequent, namely a tuple term in normal form and an enumerator invocation. To have a uniform representation for the tuples, we therefore maintain equations in the antecedent that relate normal form tuple terms with their ordinal numbers. Consider for example a finite binary relation *foo*, and an element $\mathit{binary}(a, b)$ of that relation. We introduce two equations to the antecedent: $\mathit{elem}_2(\mathit{foo}, i) \doteq \mathit{binary}(a, b)$ is used to replace occurrences of the enumerator invocation with $\mathit{binary}(a, b)$, and $\mathit{ord}(\mathit{foo}, \mathit{binary}(a, b)) \doteq i$ captures the ordinal number for the tuple.

5.4 Ordering

For a signature S that is linearly ordered by the ordering module, the function *ord* is defined to assign an ordinal number to each element of S (see Section 3.8.2). In this section, we present lemmas to efficiently handle the ordering module's successor relation `next`. All functions and predicates in the module are defined by means of this relation. We only consider the infinite case here, the finite case is similar.

The successor relation `next` is frequently joined with a singleton set to obtain the direct successor of an element. We introduce a lemma for this case:

$$\frac{in(a, S) \quad \neg finite(S)}{join_{1\times 2}(sin(a), nextS) \rightsquigarrow sin(elem_1(S, ord(S, a) + 1))}$$

A similar lemma covers the case of obtaining the predecessor of an element. Some functions and predicates of the ordering module are defined using the transitive closure of `next`, for example the comparison predicates `lt` and `gt`. We therefore introduce a lemma for the transitive closure of $nextS$, which is independent from the finiteness of S:

$$\frac{in(a, S) \quad in(b, S)}{in(binary(a, b), transClos(nextS)) \rightsquigarrow ord(S, a) < ord(S, b)}$$

As mentioned in Section 3.8.2, the induction rule for integers can be used to prove properties about all elements of a linearly ordered signature S. However, we introduce a more convenient induction rule for this case:

$$\frac{\begin{array}{l}\neg finite(S)\\ \phi(elem_1(S, 0))\\ \forall i\colon int \mid i \geq 0 \wedge (\phi(elem_1(S, i)) \Rightarrow \phi(elem_1(S, i + 1)))\end{array}}{\forall a\colon Atom \mid in(a, S) \Rightarrow \phi(a)}$$

6. Experiments

In this chapter, we evaluate the applicability of our verification approach. Section 6.1 summarizes our experiences with the automation strategy. As a case study, a proof of Dijkstra's solution to the dining philosophers problem is presented in Section 6.2.

6.1 Automation

Throughout the development of Kelloy, numerous models that are distributed with the Alloy Analyzer have been proven, including the address book model from [19] and the mark-and-sweep garbage collection algorithm. In this section, we summarize our experiences with the automation strategy.

In the absence of transitive closure, relational reasoning is efficient and simple models are usually proven automatically. When transitive closure is involved, the problem is potentially hard. Many problems, however, only rely on simple properties of transitive closure. The filesystem model of Figure 1.1 is such an example: the model uses transitive closure to postulate that every object in the filesystem is reachable from the root directory. For the proof of the assertion, KeY automatically infers from this fact that every object besides the root directory is an entry of some directory.

Besides transitive closure, universal quantifiers in the antecedent can pose a challenge to the automation. KeY's proof search strategy heuristically instantiates quantifiers with terms from the sequent. The performance of the heuristic depends on the complexity of the quantified formula. Complex formulas therefore usually require interactive instantiation, like the doubly quantified formula we manually instantiated in our example of Section 1.2. When a quantified formula is known not to be needed anymore, it is advisable to "hide" it from the sequent so the automation strategy won't bother with unnecessary instantiations.

The automation strategy reduces the user interaction necessary to conduct a proof. For interesting correctness properties of non-trivial models, however, complete automation is rarely achieved and the user has to perform central steps of the proof interactively, for example providing an induction hypothesis. In such proofs, the strategy can be used to prove the simpler branches automatically while the user

focusses on the hard parts. Interaction can often be narrowed down to a few central interactive steps. The complete proof of the mark-and-sweep garbage collection algorithm took 20839 rule applications, out of which only 10 were interactive steps, mostly instantiations of lemmas about transitive closure (like the NoExit coloring axiom, see Section 5.2).

6.2 Case Study

In this section, we conduct a case study with the Kelloy tool. The purpose of the case study is to apply the presented approach to a real-world problem, thus providing evidence of its practical applicability. In our case study we prove that, in a system where multiple processes concurrently allocate resources, Dijkstra's ordering criterion [6] prevents deadlocks.

6.2.1 The Problem and its Solution

The context of the problem is a system with multiple concurrently executed processes, and a set of resources that may be allocated through mutexes. A process can grab the mutexes it needs, use the resources and ultimately release the mutexes again. When a process tries to grab a mutex that is held by a different process, it gets stalled, that is, it waits for that mutex to get released. The system just described is subject to deadlocks: it can run into a state in which all processes are stalled.

The problem has several solutions, one of them is Dijkstra's ordering criterion [6] that assumes all mutexes to be ordered. Mutexes have to be grabbed in order. A process can thus only grab a mutex that is larger than those it already holds. In our case study, we consider an existing formalization of the system in Alloy and prove that deadlocks are in fact prevented.

6.2.2 The Model

A model for the problem is included in the Alloy distribution. That model[1], however, has only trivial instances because of an overrestrictive assumption that prevents mutexes from being grabbed. We therefore present a corrected version here. Its declaration part is the following:

```
1 open util/ordering [State] as so
2 open util/ordering [Mutex] as mo
3
4 sig Process {}
5 sig Mutex {}
6
7 sig State { holds, waits: Process → Mutex }
```

The processes and mutexes of the system are represented by the signatures `Process` and `Mutex`, respectively. An ordering for the mutexes is introduced by instantiating the ordering module for the `Mutex` signature (Line 2).

[1] shipped along with the Alloy Analyzer, Version 4.1.10

```
9  pred Initial [s: State] { no s.holds + s.waits }
10 pred IsFree [s: State, m: Mutex] { no m.~(s.holds) }
11 pred IsStalled [s: State, p: Process] { some p.(s.waits) }
12
13 pred GrabMutex [s: State, p: Process, m: Mutex, s': State] {
14    !s.IsStalled[p]
15    m !in p.(s.holds)
16    all m': p.(s.holds) | mo/lt[m',m]
17    s.IsFree[m] ⇒ {
18       p.(s'.holds) = p.(s.holds) + m
19       no p.(s'.waits)
20    } else {
21       p.(s'.holds) = p.(s.holds)
22       p.(s'.waits) = m }
23    all otherProc: Process - p {
24       otherProc.(s'.holds) = otherProc.(s.holds)
25       otherProc.(s'.waits) = otherProc.(s.waits) }
26 }
27
28 pred ReleaseMutex [s: State, p: Process, m: Mutex, s': State] {
29    !s.IsStalled[p]
30    m in p.(s.holds)
31    p.(s'.holds) = p.(s.holds) - m
32    no p.(s'.waits)
33    no m.~(s.waits) ⇒ {
34       no m.~(s'.holds)
35       no m.~(s'.waits)
36    } else {
37       some lucky: m.~(s.waits) {
38          m.~(s'.waits) = m.~(s.waits) - lucky
39          m.~(s'.holds) = lucky }
40    }
41    all mu: Mutex - m {
42       mu.~(s'.waits) = mu.~(s.waits)
43       mu.~(s'.holds)= mu.~(s.holds) }
44 }
45
46 pred GrabOrRelease {
47    Initial[so/first]
48    all pre: State - so/last | let post = so/next [pre] |
49          (post.holds = pre.holds and post.waits = pre.waits)
50       or (some p: Process, m: Mutex | pre.GrabMutex [p, m, post])
51       or (some p: Process, m: Mutex | pre.ReleaseMutex [p, m, post])
52 }
```

Figure 6.1: Predicate definitions of the model.

The typical methodology to describe dynamic systems in Alloy is to model them as execution traces (e.g. proposed in [20]): a snapshot of the system is described as a state, and valid state transitions are defined. Our model directly follows this methodology and introduces the signature `State`, which is made a sequence using the ordering module (Line 1). The fields `holds` and `waits` describe which processes hold, respectively wait for what mutexes. The predicates that are defined in the model are shown in Figure 6.1. The predicate `GrabOrRelease` describes the execution trace: The system starts with all mutexes being free, so `holds` and `waits` are empty in the first state (Line 47). The transition between two adjacent states is one of the following:

- A process tries to grab a mutex. If the mutex was previously free, the process now holds that mutex and otherwise waits for it (Line 50).
- A process releases one of the mutexes it holds. If there is a process waiting for that mutex, the mutex is passed on (Line 51).
- Nothing happens at all (Line 49).

The first transition is modeled by the `GrabMutex` predicate. Its parameters are a pre- and a post-state, `s` and `s'`, a process, `p`, and the mutex, `m`, that `p` tries to grab. In the pre-state, `p` must not be stalled and not already hold `m` (Lines 14 and 15). Line 16 ensures that mutexes are grabbed in order: the process may only grab the mutex `m` if it is larger than the mutexes already held by `p`. The next lines describe the transition from `s` to `s'`. If `m` is free, then `p` holds `m` in the post-state, along with all the mutexes previously held, and is not waiting for any mutex (Lines 18 and 19). When `m` is already occupied by another process, `p` holds the same mutexes as in the pre-state and waits for `m` in the post-state (Lines 21 and 22). Lines 23 to 25 express that things remain unchanged for the processes other than `p`.

The predicate `ReleaseMutex` describes the transition between the states `s` and `s'` in which the process `p` releases the mutex `m`. In the pre-state, `p` is not stalled and holds the mutex `m` (Lines 29 and 30). In the post-state, `p` holds the same mutexes as before, except for `m` (Line 31). If no process waits for the released mutex, it is free in the post-state (Line 34). Otherwise, it is passed on to some process that waits for `m` (Lines 37 to 39). Lines 41 to 43 describe that things remain unchanged for all mutexes but `m`.

When the system runs into a deadlock, all processes wait for some mutex. Thus no process can act and only the trivial transition is possible. This situation is expressed by the predicate `Deadlock`:

```
54 pred Deadlock {
55    some Process
56    some s: State | all p: Process | some p.(s.waits)
57 }
```

The following assertion checks that a system as described by `GrabOrRelease` never runs into a deadlock and the ordering criterion thus prevents deadlocks.

```
59 assert DijkstraPreventsDeadlocks {
60    GrabOrRelease ⇒ not Deadlock
61 }
62 check DijkstraPreventsDeadlocks
```

The analysis of the Alloy Analyzer does not find a counterexample in a scope that bounds the size of `State` to 8, and the sizes of `Process` and `Mutex` to 5. This analysis finished in about 6 minutes. When increasing the scope for `State` to 9, the analysis is no longer feasible: we aborted the solver after 30 minutes.

6.2.3 The Proof

The system is modeled as an execution trace that describes legal transitions between adjacent states. To prove properties about the system, we therefore need induction on the `State` signature. For the proof as it is shown here, we assume this signature to be infinite. For the finite case, the proof is similar.

For a complex proof, it is advisable to partition it into simpler subproofs. We therefore formulate and prove several auxiliary lemmas. Such lemmas are introduced by a "cut": a case distinction on a user provided formula. In this proof, we introduced seven such auxiliary lemmas. We present the most central ones here and sketch how they are used to prove the assertion.

In order to prove the assertion, we need to find for every state a witness process that is not stalled. Intuitively, the process that holds the largest mutex fulfills this property. We use the *maxMutex* function from the translation of the ordering module to express this property:

$$\begin{aligned}\forall s, p, m\colon Atom \mid\ & in(s, State) \wedge in(p, Process) \wedge in(m, Mutex) \\ & \wedge in(m, maxMutex(join_{1\times 2}(Process, join_{1\times 3}(sin(s), holds)))) \\ & \wedge in(ternary(s, p, m), holds) \\ & \Rightarrow \neg some(join_{1\times 2}(sin(p), join_{1\times 3}(sin(s), waits)))\end{aligned} \tag{6.1}$$

We use induction over the elements of `State` to prove (6.1). In the step case, we distinguish between the three possible transitions. The case of a mutex being released required two auxiliary lemmas, which were again proven by induction on `State`. The first one states that, if a process waits for some mutex, then all mutexes it holds are smaller than that mutex:

$$\begin{aligned}\forall s, p, m\colon Atom \mid\ & in(s, State) \wedge in(p, Process) \wedge in(m, Mutex) \\ & \wedge in(ternary(s, p, m), waits) \\ \Rightarrow\ & \forall mh\colon Atom \mid in(mh, Mutex) \wedge in(ternary(s, p, mh), holds) \\ & \qquad \Rightarrow ltMutex(sin(mh), sin(m))\end{aligned} \tag{6.2}$$

The second auxiliary lemma expresses that no process waits for a free mutex:

$$\begin{aligned}\forall s, p, m\colon Atom \mid\ & in(s, State) \wedge in(p, Process) \wedge in(m, Mutex) \\ & \wedge in(ternary(s, p, m), waits) \\ & \Rightarrow some(join_{2\times 1}(join_{1\times 3}(sin(s), holds), sin(m)))\end{aligned} \tag{6.3}$$

For the proof of the assertion, we get the assumption that the predicate `Deadlock` holds. So there is a process p_0 and in some state s_0, all processes are stalled. The latter is expressed by:

$$\forall p\colon Atom \mid in(p, Process) \Rightarrow some(join_{1\times 2}(sin(p), join_{1\times 3}(sin(s_0), waits))) \quad (6.4)$$

We want to use (6.1) to refute this assumption. We thus have to find the maximal mutex m_{max} that is being held in the state s_0. To achieve this, we first prove a set theoretical theorem that every non-empty and finite subset of *Mutex* (which is thus ordered) has a maximal element:

$$\forall r\colon Rel1 \mid finite(r) \wedge card(r) > 0 \wedge subset(r, Mutex) \Rightarrow some(maxMutex(r)) \quad (6.5)$$

We prove this theorem by induction on the set's cardinality. To use the theorem, we infer from (6.4) and (6.3) that the set $join_{1\times 2}(Process, join_{1\times 3}(sin(s_0), holds))$ is not empty. To show that the set is also finite, we show that in every state, only finitely many mutexes are held:

$$\forall s\colon Atom \mid in(s, State) \Rightarrow finite(join_{1\times 3}(sin(s), holds)) \quad (6.6)$$

We again use induction on `State` to show the correctness of (6.6): Since every state is reached by a finite number of state transitions, only finitely many mutexes can be grabbed. From (6.5) and (6.6), we obtain m_{max} and ultimately use (6.1) to show that the process holding m_{max} is not stalled, thus contradicting to (6.4).

6.2.4 Conclusion

During the proof, seven auxiliary lemmas have been proven. Despite (6.5), the proofs used induction on the execution trace. Before conducting a proof for these lemmas in the KeY system, confidence about their correctness can be gained by formulating them in Alloy and use the Alloy Analyzer for automatic analysis. Providing suitable correctness properties requires in-depth knowledge of the model and is the central part of the required user interaction.

Proving the auxiliary lemmas needed guidance by the user, mostly quantifier instantiations. Providing these instantiations requires the user to follow the proof and identify the difficult cases that the automation strategy does not handle. The user can increase the performance of the automation by hiding unnecessary (usually quantified) formulas from the sequent. While this requires some experience, large parts of the proof can be automated.

In total, the proof took 18875 rule applications, out of which 291 were performed manually. Most of the manual steps were quantifier instantiations (92) and hiding of formulas (80). An experienced user can conduct such a proof in roughly one work day.

The Alloy model makes extensive use of the ordering module. Kelloy leverages KeY's integers to handle the module. The case study provides evidence for the efficiency of this approach: no considerable user interaction was needed to handle the ordering module.

While the case study shows some of the strengths of Kelloy, it also shows some deficiencies. Most notably was the proof of (6.6) cumbersome and required some user interaction in order for the inference rules about finiteness to be applicable. In contrast to that, the proof of (6.5) was quite elegant. However, this set-theoretical theorem is not directly related to the model being verified.

7. Conclusion

7.1 Summary

Based on the KeY system, we developed the Kelloy tool that is capable of verifying Alloy models. A first-order theory was defined to permit relational reasoning within the KeY system. Kelloy uses this theory to translate Alloy specifications to KeY's first-order logic. The translated model can then be verified using KeY. Restrictions on the size of the model's instances, as imposed by the Alloy Analyzer, are dropped and instances are potentially infinite. Some parts of the Alloy language, however, are only understood for finite instances, most notably the cardinality operator. Their semantics in an infinite setting had to be defined separately from the finite case. For a subset of Alloy's relational logic, the translation has been proven correct.

KeY's automated proof search strategy was extended to the newly defined relational theory. Several hundred lemmas have been written to allow for efficient relational reasoning. We evaluated the performance of the approach in several experiments and a case study.

7.2 Related Work

The verification of Alloy models has been addressed earlier in several ways. The approach closest to ours is Prioni [3] that translates Alloy models to a polymorphic multi-sorted first-order logic. Similar to this work, Prioni provides a first-order relational theory as abstraction layer. In contrast to Kelloy, however, it regards only finite instances.

The Dynamite tool [10] verifies Alloy models by translating them to PVS [27]. It is based on fork algebras [11] and a complete proof calculus [9]. In contrast to Prioni and our approach, the target of the translation is a higher-order language. Thus, the degree of user interaction is presumably higher.

Both system, Prioni and Dynamite, integrate the Alloy Analyzer into the process of interactive proving, for example to check a user-provided hypotheses. Such ideas might also be incorporated into Kelloy, see Section 7.3. To the best of our knowledge,

Prioni and Dynamite both only cover a rather small Alloy subset and lack support for several language features like integers.

El Ghazi and Taghdiri [14, 15] use SMT solvers to verify Alloy models automatically. However, since the Alloy logic is undecidable, the solver occasionally fails to verify correct models. The approach is thus complementary to this work. A framework coupling their approach with Kelloy is presented in [13].

In [23], Event-B [1] proof obligation are translated to KeY. Event-B is a set theoretical language that also supports binary relations. Similarly to the relational theory in this work, a first-order theory that resembles the language constructs of Event-B is developed. Relations of higher arity, however, are not supported by Event-B and therefore not covered.

7.3 Future Work

In this section, we outline some possible extensions to the Kelloy tool that might be addressed in the future.

The automatic analysis performed by the Alloy Analyzer might be leveraged by Kelloy to make interactive proving more efficient. Formulas from the proof sequent could be translated back to Alloy and analyzed automatically, thus saving the user from introducing false hypotheses to the proof. One might further use the Alloy Analyzer to generate and visualize instances of the model that correspond to a particular proof sequent, thus helping the user to keep track of the proof while sequences become more complex.

The KeY system is a software verification tool. As specification languages, OCL, JML, and dynamic logic are supported. In contrast to Alloy, these languages are designed to specify an implementation rather than an abstract model of the system being implemented. Future work might explore how Alloy can be used as an alternate specification language for Java programs to be verified in the KeY system: refinement steps towards the implementation might be carried out as proof obligations in KeY, thus providing a uniform framework for the proof of the abstract model, as well as its implementation.

Kelloy does currently not handle all of Alloy's language constructs. Adding support for most of the missing features, however, does not present a theoretical challenge. The unsupported features are:

- Multiple variable bindings for the `one` and `lone` quantifiers as in `one x,y: e | F`.
- Sequences of atoms.
- The `disj` keyword on the right-hand side of a field declaration.
- The `sum` quantifier `sum x: e | i`.
- The shift operators $<<$, $>>$, and $>>>$. They manipulate the binary representation of an integer value. For mathematical integers, however, they do not seem to be very useful.
- The maximal and minimal integer values are undefined in the translation.

A. Implementation Notes

Kelloy consists of two parts: The translation tool Alloy2KeY and the KeY prover. Alloy2KeY uses the API of the Alloy Analyzer to parse and typecheck Alloy models, and desugar several syntactical features.

The translation tool gets an Alloy specification as input and generates a `.key` problem file containing the proof obligation to be loaded into the KeY system. Along with the problem file, a directory called `theory` is created that contains the rules and declarations of the relational theory, as well as the lemmas of the reasoning strategy.

To load the problem file created by Alloy2KeY, a special version of the KeY system is needed. This version is based on a current development version of KeY (1.7.2205) and implements the strategy as described in Chapter 5. It furthermore allows formulas as parameters to functions and predicates which we use for the translation of comprehensions.

Kelloy currently supports relations up to an arity of three. The Alloy2KeY tool, however, handles arbitrary arities. The limitation arises from the relational theory that lacks the necessary declarations for higher arities.

We document the usage of the translation tool Alloy2KeY here. Documentation on how to use the KeY system can be found elsewhere[1]. The distribution of Alloy2KeY comes with an executable startup script named `alloy2key`. It takes the Alloy specification to be translated as an argument. For example,

```
alloy2key model.als
```

translates `model.als`. It creates the problem file `model.als.key` and the directory `theory` in the same place as the input model. To change this behavior, an output file might be specified: Calling the script with

```
alloy2key model.als /path/to/model.key
```

creates the problem file `model.key` in the `/path/to/` directory, and also stores the `theory` directory there. By default, Alloy2KeY does not overwrite any existing files. Using the `--force` switch changes this behavior:

[1] `http://www.key-project.org`

```
alloy2key --force model.als
```

overwrites the file `model.als.key` when it exists. It is occasionally desired that signatures in an Alloy model are finite. Alloy2KeY allows the user to explicitly finitize signatures through the `--finite` switch. For example,

```
alloy2key model.als –finite=Foo,Bar
```

makes `Foo` and `Bar` finite.

B. Operator Axiomatizations

Chapter 3 introduced numerous relational operators. We show their axiomatizations here, each exemplary for one arity.

Union

$$in(a, union_1(r^{(1)}, s^{(1)})) :\Leftrightarrow in(a, r^{(1)}) \vee in(a, s^{(1)})$$

Intersection

$$in(a, inter_1(r^{(1)}, s^{(1)})) :\Leftrightarrow in(a, r^{(1)}) \wedge in(a, s^{(1)})$$

Difference

$$in(a, diff_1(r^{(1)}, s^{(1)})) :\Leftrightarrow in(a, r^{(1)}) \wedge \neg in(a, s^{(1)})$$

Join

$$in(b, join_{1\times 2}(r^{(1)}, s^{(2)})) :\Leftrightarrow \exists a\colon Atom \mid in(a, r^{(1)}) \wedge in(binary(a, b), s^{(2)})$$

Product

$$in(ternary(a, b, c), prod_{1\times 2}(r^{(1)}, s^{(2)})) :\Leftrightarrow in(a, r^{(1)}) \wedge in(binary(b, c), s^{(2)})$$

Domain Restriction

$$in(binary(a, b), domRestr_2(r^{(1)}, s^{(2)})) :\Leftrightarrow in(a, r^{(1)}) \wedge in(binary(a, b), s^{(2)})$$

Range Restriction

$$in(binary(a, b), rangeRestr_2(r^{(2)}, s^{(1)})) :\Leftrightarrow in(b, s^{(1)}) \wedge in(binary(a, b), r^{(2)})$$

Override

$$\begin{aligned} in(binary(a,b), overr_2(r^{(2)}, s^{(2)})) :\Leftrightarrow\ & in(binary(a,b), s^{(2)}) \vee \\ & (in(binary(a,b), r^{(2)}) \wedge \\ & \forall c\colon Atom \mid \neg in(binary(a,c), s^{(2)})) \end{aligned}$$

Transpose

$$in(binary(a,b), transp_2(r^{(2)})) :\Leftrightarrow in(binary(b,a), r^{(2)})$$

Transitive Closure

$$\frac{\begin{array}{c} iterJoin(r^{(2)}, 0) := r^{(2)} \\ i > 0 \end{array}}{\begin{array}{c} iterJoin(r^{(2)}, i) := union_2(iterJoin(r^{(2)}, i-1), join_{2\times 2}(r^{(2)}, iterJoin(r^{(2)}, i-1))) \\ in(t^{(2)}, transClos(r^{(2)})) :\Leftrightarrow \exists i\colon int \mid i \geq 0 \wedge in(t^{(2)}, iterJoin(r^{(2)}, i)) \end{array}}$$

Reflexive Transitive Closure

$$reflTransClos(r^{(2)}) := union_2(iden, transClos(r^{(2)}))$$

Bibliography

[1] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 2007.

[2] Alloy Analyzer 4. `http://alloy.mit.edu/alloy4`.

[3] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating model checking and theorem proving for relational reasoning. In *Seventh International Seminar on Relational Methods in Computer Science*, 2003.

[4] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas*, 2004.

[5] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Springer-Verlag, 2007.

[6] Edsger W. Dijkstra. Cooperating sequential processes. In *Programming Languages: NATO Advanced Study Institute*. Academic Press, 1968.

[7] Guozhu Dong and Jianwen Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Information and Computation*, 1995.

[8] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A type system for object models. In *Foundations of Software Engineering*, 2004.

[9] Marcelo Frias, Carlos Lopez Pombo, and Nazareno Aguirre. An equational calculus for Alloy. In *ICFEM 2004: International conference on formal engineering methods*, 2004.

[10] Marcelo Frias, Carlos Lopez Pombo, and Mariano Moscato. Alloy Analyzer+PVS in the analysis and verification of Alloy specifications. In *13th. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[11] Marcelo Fabian Frias. *Fork Algebras in Algebra, Logic and Computer Science*. World Scientific Publishing Co., Inc., 2002.

[12] Stefan Geschke. Model theory. `http://www.hausdorff-center.uni-bonn.de/people/geschke/teaching/ModelTheory.pdf`.

[13] Aboubakr Achraf El Ghazi, Ulrich Geilmann, Mattias Ulbrich, and Mana Taghdiri. A dual-engine for early analysis of critical systems. In *Workshop on Dependable Software for Critical Infrastructures (DSCI)*, 2011.

[14] Aboubakr Achraf El Ghazi and Mana Taghdiri. Analyzing alloy constraints using an smt solver: A case study. In *5th International Workshop on Automated Formal Methods (AFM)*, 2010.

[15] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via SMT solving. In *17th International Symposium on Formal Methods (FM)*, 2011.

[16] Martin Giese. A calculus for type predicates and type coercion. In *Automated Reasoning with Analytic Tableaux and Related Methods, Tableaux 2005*, 2005.

[17] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In *Computer Science Today: Recent Trends and Developments, number 1000 in Lecture Notes in Computer Science*, 1995.

[18] Charles A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 1978.

[19] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[20] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proceedings of the 9th ACM SIGSOFT international symposium on Foundations of software engineering / 8th European software engineering conference*, 2001.

[21] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, 2008.

[22] Uwe Keller. Some remarks on the definability of transitive closure in first-order logic and datalog, 2004.

[23] Christopher Köker. Discharging Event-B proof obligations. Studienarbeit, Universität Karlsruhe (TH), 2008.

[24] Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. *Computing Research Repository*, 2009.

[25] Greg Nelson. Verifying reachability invariants of linked structures. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1983.

[26] *User Manual of the RODIN Platform. Version 2.3*, October 2007.

[27] Natarajan Shankar, Sam Owre, John Rushby, and David Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, September 1999.

[28] Mana Taghdiri and Daniel Jackson. Inferring specifications to detect errors in code. *Journal of Automated Software Engineering (JASE)*, 2007.

[29] Pamela Zave. Lightweight modeling of network protocols in Alloy. `www2.research.att.com/~pamela/chord.pdf`, 2009.