Relif: a relation algebra specification tool

Marcelo M. Lynch

December 20, 2019

Abstract

Relation algebras are algebras arising from the study of binary relations. They form a part of the field of algebraic logic, and have applications in proof theory, modal logic, and computer science. An interesting problem in relation algebras is the *representation problem*, which is to give a canonical representation of a given relation algebra, in the form of binary relations. This problem doesn't have a solution for all algebras.

This paper presents *Relif*, a specification tool that allows the user to explore relation algebras satisfying a set of constraints defined by the user, and provides a way of looking for representations.

Resumen

Las álgebras de relaciones son álgebras surgidas a partir del estudio de las relaciones binarias. Forman parte del campo de la lógica algebraica y tienen aplicaciones en la teoría de demostraciones, lógicas modales y ciencias de la computación. Un problema interesante en el marco de las álgebras de relaciones es el *problema de la representación*, que consiste, dada un álgebra de relaciones, en proveer una representación canónica de la misma en forma de relaciones binarias. Este problema no siempre tiene solución.

Este trabajo presenta *Relif*, una herramienta de especificación que permite al usuario explorar álgebras de relaciones que satisfagan un conjunto de restricciones definidas por el usuario, y provee una forma de buscar representaciones de esas álgebras.

Keywords: *relation algebra, model finding, representation, Alloy, Kodkod.*

Introduction

The importance of *relationships* in logic has been recognized since the time of Aristotle. In the second half of the nineteenth century, logicians such as Augustus de Morgan and Charles Sanders Peirce embarked in the effort of formalizing the characteristics of relations and give them a mathematical framework. In the early 1900s Alfred Tarski decided to give an axiomatic characterization of relations: thus *relation algebras* were born [Tar41].

Numerous papers on relation algebras have been published since 1950, including papers in areas of computer science, and the subject has had a strong impact on such fields as universal algebra, algebraic logic, and modal logic [Giv17]. Extensions of relation algebras also show promise as tools for program verification and derivation [FVB04].

Relation algebra is an area of active research. Of particular interest is the characterization of the so-called *representable relation algebras*, which are a subclass of all *relation algebras* that can be given a canonical form (a *representation*) using binary relations [HH02].

In this paper we present *Relif*, an interactive tool that allows a user to specify properties that they wish hold within the elements of some relation algebra and, given a bound on the number of elements of the algebra, either find an example or assure that no such algebra exists within those bounds. Relif offers a user experience inspired in light-weight model checkers such as *Alloy*. Additionally, Relif can try to build representations of the relation algebras that it finds.

This document is organized as follows:

- **Chapter 1** introduces universal algebras, boolean algebras and relation algebras, and gives the main theoretical results that justify Relif's implementation details.
- **Chapter 2** gives an overview of different techniques in software verification and focuses on the model finding tools *Alloy*, which served as motivation for the tool we developed, and *Kodkod*, which is the main workhorse behind both Alloy and Relif.
- **Chapter 3** describes Relif, both in usage, semantics and inner workings. The correctness of the implementation is justified based on the results of chapter 1. Finally, an example problem with interesting conclusions is explored.
- A final **Conclusion** summarizes the work and points towards possible future developments and extensions.

Chapter 1

Algebras

In this chapter we will develop the theory of relation algebras that is necessary to understand the project. We begin by briefly describing the topic of *universal algebra* in general, then introduce *boolean algebras* as a mean to pave the way to the main definition that concerns us: relation algebras. In what follows we assume the reader is familiar with basic set theory.

1.1 Universal algebra

Universal algebra is the field of mathematics that studies the *structure* of mathematical objects and operations in and of themselves. Mathematicians and computer scientists do their work with a variety of theoretical constructions: as different kinds of objects and operations between those objects arose in history it became clear that certain *underlying structures* were common between them. For a simple example: comparing the objects "integer number" with their "integer addition" operation, and the objects we call "matrix" with their respective "matrix addition" we can recognize common properties in the structural behavior of the objects with the operation (such as "the operation is commutative"), even though the nature of the objects is quite different: these two examples are structurally what we call a *ring*.

Universal algebra is then concerned with *describing* these structures, characterizing them and studying their properties without regard of the actual *examples* that possess them. Conversely, given a set of *rules* (or *axioms*) for the behavior of certain objects and operations between them, we can try and find examples of known objects that satisfy this structure.

1.1.1 Basic definitions

Let A be a set and n any non-negative integer. We denote by A^n the cartesian product with A with itself, convening $A^0 = \{\emptyset\}$. An **n-ary operation** on A is a mapping $f : A^n \to A$. Note that a 0-ary (or ***nullary***) operation is fully determined by $f(\emptyset)$, hence we can identify it with a single element from A and we call them ***constants***. We call 1-ary operations ***unary*** and 2-ary operations ***binary***.

An ***algebra*** is a pair (A, F) where A is a set and F is a collection of operations on A. A is called the ***universe*** and the elements of F the ***fundamental operations*** of the algebra. When $F = \{f_1, \cdots, f_n\}$ we sometimes note $(A, f_1, \cdots, f_n)$ instead of (A, F). If the universe of an algebra $\mathfrak{A}$ is finite, we say that $\mathfrak{A}$ is a ***finite algebra***.

Given two algebras $\mathfrak{A} = (A, F)$ and $\mathfrak{B} = (B, G)$, with $F = \{f_i : i \in I\}$ and $G = \{g_i : i \in I\}$ (where I is some index set), a ***homomorphism*** between $\mathfrak{A}$ and $\mathfrak{B}$ is a mapping $h : A \to B$ satisfying, for all $i \in I$:

$$h(f_i(x_1, \cdots, x_n)) = g_i(h(x_1), \cdots, h(x_n))$$

Note that we are assuming that for all $i \in I$, f_i and g_i have the same arity. We say in this case that $\mathfrak{A}$ and $\mathfrak{B}$ have the same *similarity type*. This is a precondition for isomorphism.

In other words, h is an operation-preserving mapping. We shall abuse notation saying that h is a mapping between $\mathfrak{A}$ and $\mathfrak{B}$ when we mean a mapping between the universes.

Two algebras with the same intrinsic structure are often identified, even though the elements in the two algebras may in fact be different. As discussed in the previous section, this is at the core of the subject of universal algebra, where we care only about structure. This identification is made precise with the notion of an isomorphism. A homomorphism is called an ***isomorphism*** when h is a bijection. In this case, h is a structure-preserving mapping. If there exists an isomorphism between $\mathfrak{A}$ and $\mathfrak{B}$ we say they are ***isomorphic*** and we note $\mathfrak{A} \cong \mathfrak{B}$. Isomorphism is an equivalence relation.

1.2 Fields of sets and boolean algebras

We call **field of sets over** $\mathcal{U}$, where $\mathcal{U}$ is a set, to a structure $(A, \cup, \cap, \overline{\cdot}, \emptyset, \mathcal{U})$ where

- $A \subset \mathcal{P}(\mathcal{U})$, where $\mathcal{P}(\mathcal{U})$ is the powerset of $\mathcal{U}$
- $\cup$ is the union of sets
- $\cap$ is the intersection of sets
- $\overline{x}$ is the complement of x with respect to $\mathcal{U}$
- $\emptyset$ is the empty set
- A is closed under the above operations

Following our previous discussion on universal algebra, we can try to characterize the behavior of this structure with an *abstract* algebraic characterization, describing the rules governing the operations such that the operations in the field of sets satisfy them exactly. This characterization comes in the form of *boolean algebras.*

A **boolean algebra** is an algebra $(A, +, \cdot, -, 0, 1)$ (in which $+$ and $\cdot$ are binary operations, $-$ is a unary operation, and 0 and 1 are constants), that satisfies the following axioms for all $x, y, z \in A$:

$$x + y = y + x \tag{BA1}$$

$$x \cdot y = y \cdot x \tag{BA2}$$

$$x + (y \cdot z) = (x + y) \cdot (x + z) \tag{BA4}$$

$$x \cdot (y + z) = (x \cdot y) + (x + z) \tag{BA4}$$

$$x + 0 = x \tag{BA5}$$

$$x \cdot 1 = x \tag{BA6}$$

$$x + (-x) = 1 \tag{BA7}$$

$$x \cdot (-x) = 0 \tag{BA8}$$

The class of all boolean algebras is noted **BA**.

1.2.1 The representation problem for boolean algebras

It is clear from the above definitions that any field of sets is in fact a boolean algebra. A natural question to ask is: are field of sets the *only* (up to isomorphism) kinds of structures that satisfy the boolean algebra axioms? This would mean that any boolean algebra can be *canonically represented* by a field of sets. To make this question clear, let us give a simple example.

Consider the two-element boolean algebra $(\mathbb{B}, \vee, \wedge, \neg, \bot, \top)$ given by $\mathbb{B} = \{\top, \bot\}$, and the operations described in Table 1.1.

	$\bot$	$\top$
$\neg$	$\top$	$\bot$

$\vee$	$\bot$	$\top$
$\bot$	$\bot$	$\top$
$\top$	$\top$	$\top$

$\wedge$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$

Table 1.1: Definition of the operations $\neg$, $\vee$ and $\wedge$

And consider further the boolean algebra given by the field of sets over a singleton set $\mathcal{U} = \{x\}$, with $A = \{\emptyset, \mathcal{U}\}$. The set operations on the field of sets similarly yield Table 1.2.

	$\emptyset$	$\mathcal{U}$
$\overline{\cdot}$	$\mathcal{U}$	$\emptyset$

$\cup$	$\emptyset$	$\mathcal{U}$
$\emptyset$	$\emptyset$	$\mathcal{U}$
$\mathcal{U}$	$\mathcal{U}$	$\mathcal{U}$

$\cap$	$\emptyset$	$\mathcal{U}$
$\emptyset$	$\emptyset$	$\emptyset$
$\mathcal{U}$	$\emptyset$	$\mathcal{U}$

Table 1.2: Operations on the field of sets

We can see that the two algebras have esentially the same structure, which is preserved if we "rename" the elements and operations. This is precisely what an isomorphism does: the mapping $h : \mathbb{B} \to A$ with $h(\bot) = \emptyset$ and $h(\top) = \mathcal{U}$ is an isomorphism between the algebras.

Again, our question is: can we find an isomorphic field of sets for any boolean algebra? This question is known as a ***representation problem***, and it arises in the context of different kind of algebraic structures (not only boolean algebras). The answer for boolean algebras is positive, and given by Stone in [Sto38]:

Theorem 1.2.1 (Stone's representation theorem)**.** *Every boolean algebra is isomorphic to the boolean algebra of a certain field of sets.*

1.3 Binary relations

Binary relations are a mathematical way of expressing relationship between objects. These relationships can be mathematical, like "*is a multiple of*" or colloquial "*is the mother of*". In any case, relationships are everywhere, and the importance of relatives in logic has been recognized since the time of Aristotle. In the nineteenth century, a *calculus of binary relations* was developed mainly through the work of Augustus de Morgan, Charles Sanders Peirce and Ernst Schröder [Mad91].

This calculus formalizes what we could call *natural operations* between relations[1]: for instance, consider relations M and F described respectively by the phrases *is the mother of* and *is the father of.* We can think of the relation P, described by *is the parent of* as the *union* of M and F. There is also a notion of composition between relations: the relation G, "*is a grandfather of*", can be expressed by *composing* the relation P with the relation P (a grandfather is a *parent of a parent*). We can also include the intersection and complement of relations.

The calculus of binary relations

Let U be a fixed non-empty set, called the **universe of discourse**. A ***binary relation*** $\mathcal{R}$ on U is a set of ordered pairs of elements of U, this is: $\mathcal{R} \subset U \times U$.

[1] We follow the examples given in [Giv17]

Operations

The basic operations on binary relations are, for all binary relations R and S:

- *Union*: $R \cup S = \{(\alpha, \beta) : (\alpha, \beta) \in R \text{ or } (\alpha, \beta) \in S\}$
- *Intersection*: $R \cap S = \{(\alpha, \beta) : (\alpha, \beta) \in R \text{ and } (\alpha, \beta) \in S\}$
- *Complement*: $-R = \{(\alpha, \beta) : (\alpha, \beta) \in U \times U \text{ and } (\alpha, \beta) \notin R\}$
- *Converse*: $R^{-1} = \{(\beta, \alpha) : (\alpha, \beta) \in R\}$
- *Composition*: $R|S = \{(\alpha, \gamma) : \exists \beta \in U \text{ such that } (\alpha, \beta) \in R \text{ and } (\beta, \gamma) \in S\}$

Union, intersection and complement are exactly the set theoretic operations (if we think of binary relations as sets of ordered pairs). Note that the complement is relative to the relation $U \times U$. The converse relation is akin to the inverse of a function: it “flips” the ordered pairs of a relation. Relational composition is akin to the composition of functions.

Special relations

We can also point out some special relations:

- $\emptyset$ is a relation, called the ***empty relation***
- $U \times U$ is a relation, called the ***universal relation***
- $id_U = \{(x, x) : x \in U\}$ is the ***identity relation*** on U

Relational laws

A main concern of the calculus of relations as developed in the nineteenth century was to study the properties of the relational operators. Let’s see some examples of these *relational laws*. In what follows R, S, T are binary relations and the universe of discourse is U.

$R\|(S\|T) = (R\|S)\|T$	(Associativity of composition)
$R\|id_U = id_U\|R = R$	(id_U is identity of composition)
$(R^{-1})^{-1} = R$	(First involution law)
$(R\|S)^{-1} = S^{-1}\|R^{-1}$	(Second involution law)
$(R \cup S)\|T = (R\|T) \cup (S\|T)$	(Right-hand distributive law)
$T\|(R \cup S) = (T\|R) \cup (T\|S)$	(Left-hand distributive law)
$(R \cup S)^{-1} = R^{-1} \cup S^{-1}$	(Distributive law of converse over union)
$(R \cap S)^{-1} = R^{-1} \cap S^{-1}$	(Distributive law of converse over intersection)

These laws can be established by simple set-theoretic arguments.

1.4 Relation algebras

While de Morgan, Peirce and Schröder were interested in the relational operations and their properties, they didn’t pursue an axiomatic approach to the calculus. It would take several decades until Alfred Tarski proposed, in 1940, an axiomatization of a fragment of the calculus of relations. Tarski and his students subsequently developed the theory of relation algebras[2], which is ultimately the universal algebraic version of the calculus of relations. This bears a certain parallel

[2] A historical account of these developments can be found in [Mad91]

with the boolean algebras being the abstract algebraic version of fields of sets: we'll have something to say about this shortly, but let's give the pertinent definitions first.

1.4.1 Definition of relation algebras

A ***relation algebra***, also (*abstract relation algebra*) can be defined ([Giv17]) as an algebra

$$\mathfrak{A} = (A, +, -, ;, {}^{\smile}, 1')$$

where $+$ and ; are binary operations, $-$ and ${}^{\smile}$ are unary operations and $1'$ is a constant, satisfying the following axioms for all elements r, s, t in A:

$$r + s = s + r \tag{R1}$$

$$r + (s + t) = (r + s) + t \tag{R2}$$

$$-(-r + s) + -(-r + -s) = r \tag{R3}$$

$$r; (s; t) = (r; s); t \tag{R4}$$

$$r; 1' = r \tag{R5}$$

$$r^{\smile\smile} = r \tag{R6}$$

$$(r; s)^{\smile} = s^{\smile}; r^{\smile} \tag{R7}$$

$$(r + s); t = r; t + s; t \tag{R8}$$

$$(r + s)^{\smile} = r^{\smile} + s^{\smile} \tag{R9}$$

$$r^{\smile}; -(r; s) + -s = -s \tag{R10}$$

(R1) is the *commutative law for addition*, (R2) is the *associative law for addition*, (R3) is *Huntington's law*, (R4) is the *associative law for relative multiplication*, (R5) is the *(right-hand) identity law for relative multiplication*, (R6) is the *first involution law*, (R7) is *the second involution law*, (R8) is the *(right-hand) distributive law for relative multiplication*, (R9) is the *distributive law for converse*, and (R10) is *Tarski's law.*

Readers will note the similarity of axioms (R4) to (R9) with the corresponding laws of the calculus of binary relations that were described in the previous section.

We can define the special constants 0 and 1, and a binary operation $\cdot$ from the fundamental operations, by:

$$\begin{aligned}
&1 = 1' + (-1') \\
&0 = -1 \\
&r \cdot s = -(-r + -s)
\end{aligned}$$

Axioms (R1), (R2) and (R3) are actually equivalent to saying that $(A, +, \cdot, -, 0, 1)$ is a boolean algebra. The algebra $(A, +, \cdot, -, 0, 1)$ is called the *boolean reduct* of $\mathfrak{A}$. The operations $+$, $-$ (and $\cdot$) are the *boolean operations*, while ; and ${}^{\smile}$ are the *relational operations.*

We denote the class of all relation algebras by **RA**.

1.4.2 Proper relation algebras

The classic example of a relation algebra is precisely the one that motivated its definition and study, that is, the algebra of all binary relations on a set U: the algebra $(\mathcal{P}(U \times U), \cup, \overline{\cdot}, \mid, \cdot^{-1}, id_U)$

is a relation algebra.

More generally, a ***proper relation algebra***, also called a *set relation algebra* or *algebra of binary relations*, is an algebra

$$\mathfrak{A} = (A, \cup, \sim, |, .^{-1}, id_U)$$

Where:

- U is any set, called the ***base set***
- $\emptyset \in A$ and $id_U \in A$
- A is a set of binary relations in U, containing a largest relation E (i.e, $E = \bigcup A$)
- $\sim$ is the complement with respect to E: $\sim R = \{(\alpha, \beta) : (\alpha, \beta) \in E \text{ and } (\alpha, \beta) \notin R\}$
- $\cup$, $|$ and $.^{-1}$ are the operations previously defined for binary relations
- A is closed under $\cup$, $|$, $.^{-1}$ and $\sim$

It can be shown that E is an equivalence relation on U (not necessarily $U \times U$). A proper relation algebra is called ***square*** if $E = U \times U$, and ***full*** if $A = \mathcal{P}(S \times S)$ for some S. The full (and square) proper relation algebra on U (which, incidentally, is the first example we gave in this section) is noted $\mathfrak{Re}(U)$.

The class of all proper relation algebras is noted **PRA**.

1.4.3 The meaning of the operations

With the introduction of as relation algebras the intended meaning of the operations and constants in the abstract relation algebra become clear:

- $+$ is analogue to *union*
- $-$ is analogue to *complement*
- $\cdot$ is analogue to *intersection*
- ; is analogue to *relational composition*
- $\smile$ is analogue to *converse*
- $1'$ is analogue to *the identity relation*
- 1 is analogue to the *unit*, the largest relation on the algebra
- 0 is analogue to *the empty relation* (the empty set)

1.4.4 The representation problem for relation algebras

Any proper relation algebra is a relation algebra: **PRA** $\subset$ **RA**. When introducing relation algebras, we noted that they arose as an attempt to axiomatize the calculus of binary relations. A reasonable goal, then, is that the axiomatization is precise in the sense that it captures *exactly* the properties of binary relation. This is the representation problem for relation algebras: **are all relation algebras isomorphic to a proper relation algebra?**.

An isomorphism $h : \mathfrak{A} \to \mathfrak{B}$, where $\mathfrak{B} \in$ **PRA** is called a ***representation*** of $\mathfrak{A}$, and we say that $\mathfrak{A}$ is ***representable*** if such an isomorphism exists. The class of all representable relation algebras (which is the closure of **PRA** under isomorphisms) is noted **RRA**. The representation problem can then be stated as: **RA** $\overset{?}{=}$ **RRA**.

As it turns out, **the answer is negative**, and it was given by Roger Lyndon in [Lyn50]. Lyndon showed this constructively, building a relation algebra and showing it couldn't be representable.

Theorem 1.4.1 (Lyndon, 1950)**.** *There exists a relation algebra which is not isomorphic to any proper relation algebra:*

In other words, there exist *non-representable relation algebras*, and the immediate corollary is that $\mathbf{RA} \neq \mathbf{RRA}$.

1.5 Some additional definitions and results

We finish this chapter by giving a few more definitions and results that we will refer to in the rest of the work.

1.5.1 Atomic algebras

A partial order $\leq$ is defined in the universe of a relation algebra, given by:

$$r \leq s \text{ if and only if } r + s = s$$

.

Note that in a proper relation algebra $\leq$ is the same as set inclusion.

An ***atom*** in a relation algebra is a minimal non-zero element. In other words: a is an atom if and only if:

- $a \neq 0$
- $s \leq a \Rightarrow s = 0$ or $s = a$

The set of atoms of a relation algebra $\mathfrak{A}$ is noted $At(\mathfrak{A})$.

A relation algebra $\mathfrak{A}$ is ***atomic*** if for every non-zero element r exists some $a \in At(\mathfrak{A})$ such that $a \leq r$. We say in this case that a is *under* r or that r is *above* a.

All these definitions apply to boolean algebras (note that $\leq$ is defined in terms of $+$, and thus atomicity is a "boolean" property): the boolean reduct of an atomic relation algebra is an atomic boolean algebra (with exactly the same atoms). We will use the same terminology and the notation $At(\mathfrak{A})$ also in the context of boolean algebras.

The following theorem is a well known result:

Theorem 1.5.1. *Every finite relation algebra is atomic.*

In an atomic relation algebra $\mathfrak{A}$:

- any element x can be expressed as a sum of atoms. We will express this as $x = \sum X$, where $X = \{a \in At(\mathfrak{A}) : a \leq x\}$ (the set of atoms under x).
- if a is an atom, then $a^{\smile}$ is also an atom.
- if a is an atom and r any element, $a \leq r$ or $a \cdot r = 0$.
- if a and b are atoms, either $a = b$ or $a \cdot b = 0$.

All operations in a boolean or relation algebra $\mathfrak{A}$, except for complement, are ***completely additive***, meaning that, for all x and y elements of $\mathfrak{A}$, and noting $X = \{a \in At(\mathfrak{A}) : a \leq x\}$ and $Y = \{a \in At(\mathfrak{A}) : a \leq y\}$:

- $x + y = \sum X + \sum Y = \sum_{a\in X, b\in Y} a + b$
- $x \cdot y = \sum X \cdot \sum Y = \sum_{a\in X, b\in Y} a \cdot b$
- $x ; y = \sum X ; \sum Y = \sum_{a\in X, b\in Y} a ; b$
- $-x = -\sum X = \sum_{a\in X} -a$
- $x^{\smile} = (\sum X)^{\smile} = \sum_{a\in X} a^{\smile}$

1.5.2 Consistent triples

A triple of atoms (a, b, c) of some relation algebra $\mathfrak{A}$ is said to be a ***consistent triple*** or a ***cycle*** if $c \leq a; b$.

1.5.3 A finite relation algebra is determined by its atoms

The facts presented above suggest that the behavior of the atoms in an atomic relation algebra determine the behavior of the operations in all of the algebra. We will show that this is the case.

Proposition 1.5.1. *Let $\mathfrak{A}$ be a finite relation algebra and $c : At(\mathfrak{A}) \times At(\mathfrak{A}) \to \mathcal{P}(At(\mathfrak{A}))$ defined by*

$$c(a_1, a_2) = \{a \in At(\mathfrak{A}) : a \leq a_1; a_2\}.$$

and consider the structure $\mathfrak{B} = (\mathcal{P}(At(\mathfrak{A})), \cup, \overline{\cdot}, \circ, ^{\smile}, Id)$ where:

- *$\circ : \mathcal{P}(At(\mathfrak{A})) \times \mathcal{P}(At(\mathfrak{A})) \to \mathcal{P}(At(\mathfrak{A}))$ is defined by*
$$X \circ Y = \bigcup_{x\in X, y\in Y} c(x, y)$$
- *$^{\smile} : \mathcal{P}(At(\mathfrak{A})) \to \mathcal{P}(At(\mathfrak{A}))$ is defined by $X^{\smile} = \{x^{\smile} : x \in X\}$*
- *$Id = \{a \in At(\mathfrak{A}) : a \leq 1'\}$*

Then $\mathfrak{B}$ is a relation algebra and the function $h : A \to \mathcal{P}(At(\mathfrak{A}))$ given by $h(x) = \{a \in At(\mathfrak{A}) : a \leq x\}$ is an isomorphism between $\mathfrak{A}$ and $\mathfrak{B}$.

The proof of proposition 1.5.1 is straightforward, using basic relation algebra arithmetic.

Corollary 1.5.1. *A finite relation algebra $\mathfrak{A}$ is determined by its set of consistent triples, the behavior of $^{\smile}$ restricted to $At(\mathfrak{A})$, and the atoms under 1'.*

Characterizing RA only with atoms

A characterization of **RA**, equivalent to the axiomatization we gave in section 1.4.1, is given by Frías and Maddux in [FM97]:

Theorem 1.5.2 ([FM97], Lem 2.2)**.** *$\mathfrak{A} = (A, +, -, ;, ^{\smile}, 1')$ is in **RA** if and only if:*

$(A, +, \cdot, -, 0, 1)$ is a boolean algebra, and

for all $v, w, x, y, z \in A$:

$$x = x; 1' \qquad \text{(FM1)}$$

$$(x; y) \cdot z \neq 0 \iff x^{\smile}; z \cdot y \neq 0 \qquad \text{(FM2)}$$

$$(x; y) \cdot z \neq 0 \iff z; y^{\smile} \cdot x \neq 0 \qquad \text{(FM3)}$$

$$(v; x) \cdot (w; y) \neq 0 \iff (v^{\smile}; w) \cdot (x; y^{\smile}) \neq 0 \qquad \text{(FM4)}$$

The following lemma characterizes finite relation algebras with only the behavior *of its atoms* with respect to the relational operations. This proves what we were stating before.

Lemma 1.5.2.1. *An algebra* $\mathfrak{A} = (A, +, -, ;, ^\smile, 1')$ *(where* $+$, $;$ *are binary operations,* $-$ *and* $^\smile$ *are unary, and* $1'$ *is a constant) is a finite relation algebra if and only if:*

$(A, +, \cdot, -, 0, 1)$ *is a finite boolean algebra (where* $\cdot$, 0, 1 *are defined as usual)*

The operations $;$ *and* $^\smile$ *are completely additive.*

For all $v, w, x, y, z \in At(\mathfrak{A})$*:*

$$x = x; 1' \qquad \textit{(A1)}$$

$$z \leq x; y \iff y \leq x^\smile; z \qquad \textit{(A2)}$$

$$z \leq x; y \iff x \leq z; y^\smile \qquad \textit{(A3)}$$

$$(v; x) \cdot (w; y) \neq 0 \iff (v^\smile; w) \cdot (x; y^\smile) \neq 0 \qquad \textit{(A4)}$$

Proof. $(\Rightarrow)$: Let $\mathfrak{A} = (A, +, \cdot, -, 0, 1)$ be a finite relation algebra. Then by theorem 1.5.2 $(A, +, \cdot, -, 0, 1)$ is a (finite) boolean algebra. Also:

$(FM1)$ implies $(A1)$: If x is an atom, then it is an element of $\mathfrak{A}$, then $x = x; 1'$

$(FM2)$ implies $(A2)$: Let x, y, z be arbitrary atoms. Then:

$$\begin{array}{ll} z \leq x; y & \\ \iff (x; y) \cdot z \neq 0 & \text{definition of } \leq \\ \iff (x^\smile; z) \cdot y \neq 0 & \text{(FM2)} \\ y \leq x^\smile; z & \text{because } y \text{ is an atom} \end{array}$$

$(FM3)$ implies $(A3)$: Let x, y, z be arbitrary atoms. Then:

$$\begin{array}{ll} z \leq x; y & \\ \iff (x; y) \cdot z \neq 0 & \text{definition of } \leq \\ \iff (z; y^\smile) \cdot x \neq 0 & \text{(FM3)} \\ x \leq z; y^\smile & \text{because } x \text{ is an atom} \end{array}$$

(FM4) implies (A4): Trivially, given that the atoms are elements of the algebra.

$(\Leftarrow)$: For the converse, let $\mathfrak{A} = (A, +, \cdot, -, 0, 1)$ be a finite algebra (where $+$, $;$ are binary operations, $-$ and $^\smile$ are unary, and $1'$ is a constant), such that $(A, +, \cdot, -, 0, 1)$ is a boolean algebra (where $\cdot$, 0, 1 are defined as usual), and $;$ and $^\smile$ are completely additive. We have:

$(A1)$ implies $(FM1)$. If x is an element of the algebra, then because $(A, +, \cdot, -, 0, 1)$ is a finite (thus atomic) boolean algebra, $x = \sum X$, where $X = \{a \in At(\mathfrak{A}) : a \leq x\}$. Then:

$$\begin{aligned}
&x;1' \\
&=\sum X;1' \\
&=\sum_{a\in X}(a;1') && \text{(Complete additivity)} \\
&=\sum_{a\in X} a && \text{(A1)} \\
&=x
\end{aligned}$$

$(A2)$ implies $(FM2)$. Let $x=\sum X$, $y=\sum Y$, $z=\sum Z$ be elements of $\mathfrak{A}$.

$$\begin{aligned}
&(x;y)\cdot z\neq 0 \\
\iff\ &(\sum X;\sum Y)\cdot\sum Z\neq 0 \\
\iff\ &\sum_{a\in X,b\in Y,c\in Z}(a;b\cdot c)\neq 0 && \text{(Complete additivity, twice)} \\
\iff\ &(a';b'\cdot c')\neq 0 \text{ for some } a'\in X,\ b'\in Y,\ c'\in Z && \text{(Boolean algebra)} \\
\iff\ &c'\leq(a';b') && \text{(c' is an atom)} \\
\iff\ &b'\leq(a'^{\smile};c') && \text{(A2)} \\
\iff\ &(a'^{\smile};c')\cdot b'\neq 0 \\
\iff\ &\sum_{a\in X,b\in Y,c\in Z}(a^{\smile};c)\cdot b\neq 0 && \text{(Boolean algebra)} \\
\iff\ &(\sum X^{\smile};\sum Z)\cdot\sum Y\neq 0 && \text{(Complete additivity, twice)} \\
\iff\ &x^{\smile};z\cdot y\neq 0
\end{aligned}$$

$(A3)$ implies $(FM3)$. Analogous to $(A2)$ implies $(FM2)$.

$(A4)$ implies $(FM4)$. Let $x=\sum X$, $y=\sum Y$, $v=\sum V$, $w=\sum W$ be elements of $\mathfrak{A}$. Then:

$$\begin{aligned}
&(v;x)\cdot(w;y)\neq 0 \\
\iff\ &(\sum V;\sum X)\cdot(\sum W;\sum Y)\neq 0 \\
\iff\ &(\sum_{a\in V,b\in W,c\in X,d\in Y}(a;c)\cdot(b;d)\neq 0 && \text{(Complete additivity)} \\
\iff\ &(a';c')\cdot(b';d')\neq 0 \text{ for some } a'\in V,b'\in W,c'\in X,d'\in Y && \text{(Boolean algebra)} \\
\iff\ &(a'^{\smile};b')\cdot(c';d'^{\smile})\neq 0 && \text{(A4)} \\
\iff\ &\sum_{a\in V,b\in W,c\in X,d\in Y}(a^{\smile};b)\cdot(c;d^{\smile})\neq 0 && \text{(Boolean algebra)} \\
\iff\ &(\sum V^{\smile};\sum W)\cdot(\sum X;\sum Y^{\smile})\neq 0 && \text{(Complete additivity)} \\
\iff\ &(v^{\smile};w)\cdot(x;y^{\smile})\neq 0
\end{aligned}$$

This completes the proof. □

1.5.4 Representations as labelled graphs

Given that not every algebra is representable, it is interesting, given an algebra, to try and find a representation of it. For finite relation algebras, we can build representations by labelling directed graphs.

The following theorem gives the construction: the idea is to consider the ordered pairs of a potential base set as *edges on a directed graph.* The set of all the pairs (x_1, x_2) labelled with the same atom a of $\mathfrak{A}$ will be the corresponding atom in the representation. Of course, the labelling must respect certain conditions as to constitute a representation of the algebra in question.

Theorem 1.5.3. *Let $\mathfrak{A} = (A, +, -, ;, \smile, 1')$ be a finite relation algebra. If there exists a directed graph $G = (V, E)$, where V is a set of vertices and $E \subset V \times V$ a set of directed edges, and a labelling function $\lambda : E \to At(\mathfrak{A})$, such that:*

1. *E is an equivalence relation on V:*
   $(v, v) \in E$ for all $v \in V$
   If $(u, v) \in E$ then $(v, u) \in E$
   If $(u, v) \in E$ and $(v, w) \in E$, then $(u, w) \in E$
2. *λ is surjective (i.e, every atom appears in the labelling)*
3. *$\lambda(u, v) \leq 1' \iff u = v$*
4. *$\lambda(u, v) = \lambda(v, u)^\smile$*
5. *If (u, v), (v, w) and (u, w) are edges, $\lambda(u, w) \leq \lambda(u, v); \lambda(v, w)$*
6. *For all $a, b \in At(\mathfrak{A})$, if $\lambda(u, w) \leq a; b$ then there is $v \in V$ with $\lambda(u, v) = a$ and $\lambda(v, w) = b$.*

Then let $h : A \to \mathcal{P}(E)$ such that

$$h(x) = \bigcup_{a \in X} \lambda^{-1}(a),$$

where $\lambda^{-1}(a) = \{(v_1, v_2) : \lambda(v_1, v_2) = a\}$ and $X = \{a \in At(\mathfrak{A}) : a \leq x\}$.

Then $\mathfrak{A}$ is representable in the proper relation algebra $\mathfrak{B} = (B, \cup, \sim, |, .^{-1}, id_V)$, where B is defined as the range of the function h.

Proof. First, let's note that $\mathfrak{B}$ truly is a proper relation algebra, according to the definition we gave in Section 1.4.2.

- $h(0) = \emptyset$ (because 0 has no atoms under it) and $h(1') = id_V$ (because of condition (3)), so both $\emptyset \in B$ and $id_V \in B$.
- B is closed under $\cup$: if $\alpha \in B$ and $\beta \in B$, then $\alpha = h(x)$ for some $x \in A$ and $\beta = h(y)$ for some $y \in A$. It is clear from the definition that $h(x + y) = \alpha \cup \beta$, and so $\alpha \cup \beta \in B$.
- B is closed under $.^{-1}$: If $\alpha \in B$, then $\alpha = h(x)$ for some $x \in A$.

  Because of condition (1), given any $(v_1, v_2) \in \alpha$, the edge (v_2, v_1) exists. We have that $\lambda(v_1, v_2) \leq x$ but then, by condition (2), $\lambda(v_2, v_1) = \lambda(v_1, v_2)^\smile$ and then $\lambda(v_2, v_1) \leq x^\smile$, which in turn means that $(v_2, v_1) \in h(x^\smile)$. Then $\alpha^{-1} \subset h(x^\smile)$.

  Next, consider any edge (v_1, v_2) labelled a, where a is an arbitrary atom under $x^\smile$ (this means $a^\smile$ is an atom under x). Then (because of condition (4)) $\lambda(v_2, v_1) = a^\smile$, and so $(v_2, v_1) \in \alpha$: equivalently, $(v_1, v_2) \in \alpha^{-1}$. This means that $h(x^\smile) \subset \alpha^{-1}$.

  We conclude that $h(x^\smile) = \alpha^{-1}$ (thus $\alpha^{-1} \in B$).
- B is closed under $\sim$, the complement with respect to E. If $\alpha \in B$, then $\alpha = h(x)$ for some $x \in A$. Now, the atoms under $-x$ are precisely the atoms that are *not* under x. Then every edge *not labelled* by an atom under x will be labelled by an atom under $-x$. Then $h(-x) = E - \alpha = \sim \alpha$, and so $\sim \alpha \in B$.

- B is closed under $|$: if $\alpha \in B$ and $\beta \in B$, then $\alpha = h(x)$ for some $x \in A$ and $\beta = h(y)$ for some $y \in A$.

  Let (u, w) be an element of $\alpha|\beta$. This means that $(u, v) \in \alpha$ and $(v, w) \in \beta$ for some $v \in V$. We know that $\lambda(u, v) \leq x$ and $\lambda(v, w) \leq y$. Because of condition (1), $(u, w) \in E$, and, because of condition (5), $\lambda(u, w) \leq \lambda(u, v); \lambda(v, w)$. Then (by additivity) $\lambda(u, w) \leq x; y$, and so $(u, w) \in h(x; y)$. This means that $\alpha|\beta \subset h(x; y)$.

  Next, consider any edge (u, w) labelled a, with $a \leq x; y$. Because of additivity, $a \leq x_a; y_a$ for some **atoms** $x_a \leq x$, $y_a \leq y$. Now, because of condition (6), as $\lambda(u, w) \leq x_a; y_a$ then there is a $v \in V$ such that $\lambda(u, v) = x_a$ and $\lambda(v, w) = y_a$. This means $(u, v) \in h(x) = \alpha$ and $(v, w) \in h(y) = \beta$, and then $(u, w) \in \alpha|\beta$. Then $h(x; y) \subset \alpha|\beta$.

  We conclude $h(x; y) = \alpha|\beta = h(x)|h(y)$, then $\alpha|\beta \in B$ and so B is closed under $|$.

Consider the function $\tilde{h} : A \to B$ such that $\tilde{h}(x) = h(x)$ (in other words, $\tilde{h}$ is just h with the codomain restricted to its range). Note that whilst proving the closure under operations we have also proven the following:

- $h(1') = id_V$
- $h(x + y) = h(x) \cup h(y)$
- $h(x^{\smile}) = h(x)^{-1}$
- $h(-x) = \sim h(x)$
- $h(x; y) = h(x)|h(y)$

and thus $\tilde{h}$ is a homomorphism.

Additionally, $\tilde{h}$ is a bijection. It is surjective by definition (B is the range of h), and it is injective: if $x, y \in A$ with $x \neq y$, then without loss of generality there is an atom $a \leq x$ such that $a \not\leq y$. Because λ is surjective, then some $(x_1, x_2) \in E$ is labelled a. This means $(x_1, x_2) \in \tilde{h}(x)$ but $(x_1, x_2) \notin \tilde{h}(y)$, then $h(x) \neq h(y)$.

$\tilde{h}$ is a bijective homomorphism, so it is an isomorphism between $\mathfrak{A}$ and $\mathfrak{B}$, and thus a representation of $\mathfrak{A}$.

□

Chapter 2

Model finding

In this chapter we describe model finding in the context of software verification, introducing *Alloy* and *Kodkod* as tools for that purpose.

2.1 Software verification

In a world in which software is increasingly present and in which people entrust more and more of their responsibilities to automatic systems the need to guarantee that software *does what we intend it to do* is similarly increasing. We can easily think of many examples in which a software error (or *bug*) could cost human lives:

- Avionics
- Nuclear facilities
- Medical equipment software (a notable example can be found in [LT93])

And even when the bug isn't life-threatening, the financial consequences can be huge [Har03]. In any case, the importance of *correctness* regarding programs and algorithms is clear. This correctness must be with respect to some *specification*: a set of rules about the functionality and effects of the software that the programmers define.

As software systems become ubiquitous, they become more complex: this in turn means that their correct, error-free implementation becomes non-trivial. This leads to the necessity of tools and techniques that can assist the programmer in the task. The theory and development of these techniques is the setting of the subfield of computer science known as *software engineering.* This checking of correctness is the activity called *software verification.*

2.1.1 Verification techniques

In this section we will give an overview of different software verification techniques. We don't pretend an exhaustive analysis but merely a birds-eye view, aiming to contextualize the tools that are relevant to this work.

Testing and peer review

In practice the most widely used verification methods are *testing* and *peer review.* The latter is simply the acceptance of correctness from both the programmer who wrote the code and her peers, before checking in the code.

Testing is a way of checking the behaviour of a program at runtime. A *test suite* exercises different sections of the code, different components of the system, and checks that it works as expected in various scenarios, by validating certain assertions of the state of the system after running the component.

Testing is useful to find bugs and for consistency (ensuring that the system's behaviour doesn't change with time), but it's non-exhaustive, in the sense that there's usually no way of covering all possible executions. Naturally, peer review suffers from the same limitation.

Dynamic and static analyisis to find bugs

Besides testing we can name other non-exhaustive techniques, which don't guarantee correctness under every possible scenario but are useful for bug finding. These methods can be based on *static analysis* of the code (i.e, the analysis is carried out without executing the program) or *dynamic analysis* (involving execution with different inputs, as in testing). We won't describe these techniques in detail, but as paradigmatic examples we can mention *symbolic execution* and *fuzzing*, respectively.

Formal methods

The need to actually *prove correctness* of programs instead of only partially exploring the space of executions leads us to formal methods, which can be described as "the applied mathematics for modeling and analyzing software systems" [BK08]. The aim is to establish system correctness with mathematical rigor: this is achieved by providing formal proofs on top of a mathematical model of the system.

One of the common approaches is *deductive verification.* These techniques are carried out by endowing the system with formal semantics (which can be expressed with logical formulae). Specifications are similarly expressed. Finally, dedicated systems such as *theorem provers* are used to formally show that the system is valid with respect to the specification.

A different approach, which will be our focus in the next section, are *model based verification techniques.* These are based on models describing the possible system states and behaviors in a mathematically precise and unambiguous manner. These models are accompanied by algorithms that systematically explore all states of the model. This leads to various techniques of both exhaustive exploration (model checking) and partial exploration (simulation, or even testing) [BK08].

2.1.2 Model checking and lightweight methods

Given a mathematical model of the system, and some property we expect to hold in the system, we call **model checking** to the set of verification techniques that can check the property in all possible states of the system, exploring them in a brute-force (but systematic) manner. Model checkers can also provide counterexamples when properties do not hold.

It is important to notice that these techniques assume that the model itself (which must provided by the user) is a faithful representation of the system: *any verification using model checking techniques is only as good as the model of the system*, so an essential part of the model checking process is then the design and refinement of the model [BK08].

When the space of states of the model is large, model checkers may require considerable computational resources (or may fail to terminate). This problem is called *state explosion*, and arises because model checkers are often used to analyze models with components that work in parallel, and thus states that grow exponentially with the number of components. [Jac19]. Motivated by this, and the practical issues regarding the modeling of systems that are both large and still in development, the formal method communities have developed ***lightweight approaches*** [AL98] to formal methods. The term lightweight is used in the sense of "less-than-completely formal" or "partial", where the methods can be used to perform "partial analysis on partial specifications, without a commitment to developing and baselining complete formal specifications" [ELC$^+$98].

2.2 Alloy

Alloy, developed by Daniel Jackson at MIT [Jac02] was designed as a small specification language aimed at lightweight formal verification. It was designed as an attempt to provide “the smallest modelling notation that can express a useful range of structural properties, is easy to read and write, and can be analyzed automatically” [Jac02]. Together with the specification language comes an *analyzer*, that given the specification, can:

- Generate a model that satisfies the specification (*simulation*)
- Given an assertion that turns out to be false in some model that satisfies the implementation, generate such model, this is, give a counterexample to the assertion (*checking*)

We can call Alloy a **model finder** (or **instance finder**) [Jac12], in the sense that its primary use case is to find *instances* given certain constraints (where the instances can be, given a specification, models of such specification or counterexamples to an assertion). The latest versions of Alloy come with an interactive graphic interface which lets the user graphically explore the models, and not only verify assertions but also *iteratively build* more and more refined specifications of their system.

The verification strategy in Alloy is also *partial*, because it works within a **scope**: it will search exhaustively for models or counterexamples but the amount of objects that it considers is **always bounded from above**.

2.2.1 An example

Let’s illustrate the characteristics and usage of Alloy and its analyzer with a simple example. Suppose we want to model a universe of people and dogs, in which every person has a set of friends and a pet.

Our first attempt at a specification that captures this model could be as follows:

```
-- The model: a person has a set of friends and an animal pet

-- There are persons
sig Person {
   friends: set Person, -- with a set friends
   pet: Animal  -- and a pet
}

-- ... and animals
sig Animal {
  owner: lone Person    -- An animal may have an owner
}


-- Show me a model with up to 3 elements
run {} for 3
```

Figure 2.1: A first attempt at an Alloy model

In Alloy our objects like `Person` and `Animal` are called *signatures* and declared with the keyword `sig`. To a *signature* we can associate fields, like `friends`, `pet` and `owner`, in a style reminiscing of object oriented programming languages. Associated with this fields are *cardinality constraints*: in our example, we are saying that a `Person` has a `set` of `friends` (one or more), exactly one `pet`, and that an `Animal` has one or zero `owners` (`lone` stands for *less than or equal* *to one*).

Finally, with the *command* "`run {} for 3`" we are telling the analyzer to look for models with up to 3 elements of each signature (3 is the *bound* of the search).

If we run the analyzer with the specification of Figure 2.1 we get (among other valid instances) the result shown on Figure 2.2:

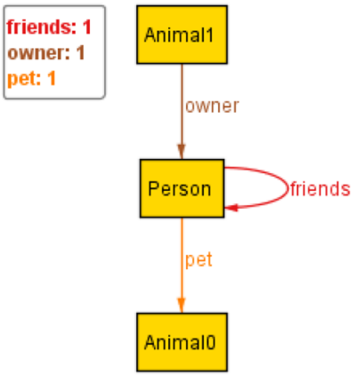


Figure 2.2: An instance found by the analyzer

We can see right away that our specification has some problems: there is an issue of consistency with pets and owners (`Animal1`'s owner is `Person` but `Person`'s pet is `Animal0`), and we find that our model allows `Persons` to be friends with themselves, a situation we may want to avoid. To solve these problems we add some constraints to our model in the form of facts, as shown in Figure 2.3:

```
-- Constraints of the model

-- 1. No person is their own friend
fact { all p: Person | p not in p.friends }

-- 2. Ownership and petship consistency
fact { all a: Animal , p: Person | a.owner = p iff p.pet = a}
```

Figure 2.3: Constraints to the model.

We can see that the constraints are expressed in a straightforward manner in the form of logical formulae, with quantifiers, logical operators such as `iff` (*if and only if*) and set-theoretical predicates (such as `not in` or equality). Running the same command as before gives us a better looking result, shown in Figure 2.4. Increasing the scope leads to more complex instances that we can explore to continue polishing our model: for example, looking at the model shown in Figure 2.5 (found with a scope of 4) we can see that friendship are not always symmetric. If we wish to correct this, we add another constraint, and the process continues.

This is the value of the style of modelling encouraged by Alloy: the iterative exploration and refinement of our model leads us to different insights about our design, which are useful not only for verification purposes but for the actual design and development of the systems [Jac12].

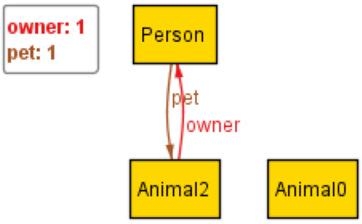


Figure 2.4: A better looking model

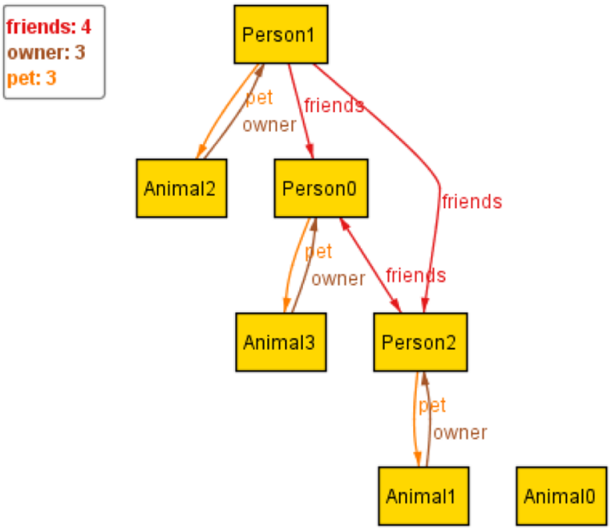


Figure 2.5: An example found with scope 4 shows that friendship is not symmetric in our specification

Finally, we can check assertions in our model. If instead of the `run` command we give a `check` command, like the one given in Figure 2.6:

```
-- Do different people have different pets?
-- Check this assertion or find a counterexample
-- (within the scope of 4)
check { all p1, p2: Person | p1 != p2 implies p1.pet != p2.pet }
    ↪  for 4
```

Figure 2.6: An assertion

the analyzer will try to find a counterexample within the given scope. In this case it can't find any, so it shows the message:

```
No counterexample found. Assertion may be valid.
```

Where the word *may* is indicating that there *could* be counterexamples within a larger scope.

The semantics of Alloy

Alloy uses a form of **relational logic** for expressing the constraint, which is basically a first-order logic augmented with operators of relational calculus. Every object in an Alloy model is ultimately encoded as a relation, and the relational calculus is similar to the one described in chapter 2, with some additional operations and extended to relations of any *arity* (not just binary relations). Signatures define sets of objects (unary relations), while fields like the ones we defined are binary relations: in the previous example `Person` and `Animal` are sets, and `pet` is a binary relation, pairs `(p, a)` where `p` is an element of `Person` and `a` an element of `Animal`.

To actually find the instances, Alloy uses a *relational model finder engine* called **Kodkod**, which we will describe next. Alloy translates the specifications to *Kodkod problems*, and, when a solution comes back from Kodkod, presents it to the user in the manner that we shown.

2.2.2 Kodkod

Kodkod [TJ07] is the solving engine used as back end for the Alloy analyzer. Kodkod was developed by Emina Torlak for her doctoral thesis at MIT [Tor09], and it ultimately replaced the original back end (i.e., the constraint solver) of the Alloy analyzer. As described in Torlak's thesis (the emphasis is ours):

> Kodkod extends the relational logic of Alloy with the notion of **relational bounds**. A bounded relational specification is a collection of constraints on relational variables of any arity that are bound above and below by relational constants (i.e. sets of tuples). All bounding constants consist of tuples that are drawn from the same finite universe of uninterpreted elements. The upper bound specifies the tuples that a relation *may* contain; the lower bound specifies the tuples that it *must* contain. [Tor09]

A Kodkod problem consists of a *universe declaration*, a set of *relation declarations*, and a *formula* in which the declared relations appear as free variables [TJ07]. The universe is esentially a set of uninterpreted elements, and solving a Kodkod problem entails constructing relations as sets of tuples of the elements of the universe in a way that the relational formula is satisfied. Additionaly, the user specifies a set of bounds for each relational variable, which constraints the tuples that may or must appear in the relation associated with that variable, as described above.

The possibility of specifying bounds for the relations makes Kodkod more flexible than Alloy, as it allows expressing *partial instances*. An illustrative example of a partial instance given in [TD06] is that of a half-filled Sudoku puzzle: in Kodkod it is straightforward to specify not only the rules of how a Sudoku puzzle is supposed to be filled, but also information on *filled slots* (thus partially realizing an instance of the specification), while doing this in Alloy is more cumbersome (and inefficient, because it means encoding the partial solution as constraints of the specification, which makes the final formula more complex).

The solving of the formula itself is carried out by translating the relational formulas into a propositional formula by means of encoding into propositional variables the statements such as "the tuple t belongs to relation R". After this translation the problem becomes a classical boolean satisfiability problem that can be solved with off-the-shelf SAT solvers [TD06].

The Kodkod engine itself consists of a rich API that allows the user to construct a Kodkod problem, invoke the solver and ultimately iterate the solutions. This makes Kodkod a valuable tool to use as back end for any application wishing to make use of relational constraint satisfaction problems, (such as Alloy itself), and this is precisely what the tool presented in this work, Relif, does.

Chapter 3

Relif

In this chapter we bring together relation algebras and model finding, introducing **Relif**, a tool that could be classified as a *relation algebra model finder*. We describe the usage and inner workings of the tool in its current state justifying its correctness, and outline possible future work.

3.1 Introduction

The main features of Relif are two:

- The ability for a user to **provide a specification**, consisting of constraints on relational variables, and, given a scope on the number of atoms, try to find relation algebras that satisfy such specification. The relational variables will be bound to elements in the universe of the algebras.
- Additionaly, given a found instance (which is a relation algebra), the user can instruct the tool to **try to find representations** of such instance. The search is bounded by a certain cardinality of the base set.

Relif provides an interactive experience not unlike Alloy's, in which the user can iterate through different specifications, adding and removing constraints and exploring the different results.

3.2 Usage and semantics

```
// Declarations
rel R, Q    // R and Q are relations
atom a      // a is an atom

// Facts
a in iden
a in Q
R;(Q + R) = R

// Command
run {} for 3
```

Figure 3.1: A simple Relif specification

A *Relif specification* consists of:

- One or more **declarations** of relational variables

- A list of **facts**, formulas in which the relational variables are free variables
- A **command**, which can be one of two

  A *run* command, which instructs the tool to try and find instances that satisfy all of the facts and an optional additional predicate

  A *check* command, which given an assertion tries to find a counterexample
- A set of **bounds**, which will limit the number of atoms of the relation algebras that the tool will find

Figure 3.1 shows a simple example of a specification. In this example, the user declares three relational variables, R, Q and a. R and Q will be bound to arbitrary elements of the universe, while a will be bound to an atom.

3.2.1 Semantics

Let us briefly outline the grammar of the Relif specification language by giving the different statements that can be written and their semantics.

Declarations

The user can declare variables that will represent relations or single atoms, as shown in table 3.1

Statement	**Meaning**
`rel R`	*R is a relation*
`atom a`	*a is an atom*

Table 3.1: Declarations

Relational expressions

A relational expression is either a previously declared relational variable or an operation between relational expressions. Table 3.2 describes the meaning of the operators, where R and S are relational expressions. The meaning is given with the notation of chapter 2 for the operations of relation algebras.

Expression	**Meaning**
`R + S`	$R + S$
`R & S`	$R \cdot S$
`R;S`	$R;S$
`R.S`	$R;S$
`-R`	$-R$
`~R`	$R^{\smile}$
`R - S`	$R + (-S)$

Table 3.2: Relational expressions

Facts

The facts can either be *comparison facts*, which predicate about equality and $\leq$ (i.e., atomic first-order formulas with those predicates), or formulas with the usual logical connectives. Table 3.3 shows the possibilities for *comparison facts*, where R and S are relational expressions. Table 3.4 shows the different connectives (where P and Q can be any fact).

Formula	**Meaning**
`R = S`	$R = S$
`R != S`	$R \neq S$
`R in S`	$R \leq S$
`R not in S`	$R \not\leq S$
`no R`	$R = 0$

Table 3.3: Comparison formulas

Formula	**Meaning**
`not P`	$\neg P$
`! P`	$\neg P$
`P and Q`	$P \wedge Q$
`P && Q`	$P \wedge Q$
`P or Q`	$P \vee Q$
`P || Q`	$P \vee Q$
`P implies Q`	$P \Rightarrow Q$
`P => Q`	$P \Rightarrow Q$
`P iff Q`	$P \iff Q$
`P <=> Q`	$P \iff Q$

Table 3.4: Formulas with connectives

Special constants

Relif has special constants corresponding to the relation algebra constants. Table 3.5 shows the different keywords corresponding to such constants. These constants are also relational expressions.

Expression	**Meaning**
`iden`	$1'$
`univ`	1

Table 3.5: Special constants

There is no special constant for 0, but it can be bound to any relational variable `Z` by stating the fact "`no Z`", as described in table 3.3.

Commands

As mentioned above, there are two types of commands, *run* commands and *check* commands. Both commands have the form

type { *F* } **for** *bound*

Where:

- *type* is one of `run` or `check`,
- *F* is an optional formula (with the same syntax as outlined before for facts),
- *bound* is a bounding expression, which will be described next

If the type of the command is `run`, the formula F (if present) is added as a fact of the specification, and an instance satisfying the resulting constraints (and within the bounds) will be searched. If the command is `check`, then the negation of F is added to the specification: if an instance is found, it serves as a counterexample for the assertion F.

Bounding expressions

Relif bounds the search by **bounding the number of atoms** of the solutions that it will consider. The bound is specified separately for three types of atoms: ***identity*** atoms are atoms under the $1'$ relation, ***symmetric*** atoms are all atoms a such that $a^{\smile} = a$ but are not under the identity. Finally, ***asymmetric*** atoms are the atoms a such that $a^{\smile} \neq a$.

The most general bounding expression is then:

x `id,` y `sym,` z `asym`

This tells Relif to look for algebras with **at most** x identity atoms, **at most** y symmetric atoms and **at most** z asymmetric atoms.

If the keyword `exactly` appears before the numbers x, y or z, then the number of atoms of that type in the solutions (if any) will be exactly that (the bound is tight).

A bounding expression can be also a single number `x`, which is shorthand for `x id, x asym, x sym`, or the keyword `default`, which is shorthand for `1 id, 3 asym, 3 sym`.

Finally, these shorthand expressions can be partially overridden using the keyword `but`. For example, the bounding expression “`3 but 1 sym`” is equivalent to “`3 id, 1 sym, 3 asym`”, and the expression “`default but exactly 2 sym`” is equivalent to “`1 id, 3 asym, exactly 2 sym`”.

3.3 Implementation

Relif uses Kodkod as its solving back-end. This means that the Relif specification is translated to a Kodkod problem, which is then solved by the Kodkod engine as described in chapter 3. Recall that a Kodkod problem consists of a *universe declaration*, a set of *relation declarations*, a *formula* which the engine will try to satisfy, and *bounds* on the different relational variables.

3.3.1 “Relation algebra finding” as a relational algebra problem

Ultimately, what we must ask Kodkod is for a set of relations that satisfy a given formula. We will see how, using the results from chapter 2, we turn the problem of **finding a relation algebra that satisfies a Relif specification** into the problem of **finding certain ($n-$ary) relations that satisfy a Kodkod formula**.

We only need to speak about atoms

Proposition 1.5.1 lets us turn the problem of describing a relation algebra by just describing its set of atoms, how $^{\smile}$ acts upon the atoms, and the set of consistent triples (this is, how ; acts upon the atoms). On the other hand, we proved with Lemma 1.5.2.1 that we can axiomatize a finite relation algebra **using only statements that speak of atoms** (provided that they already form a boolean algebra).

This two facts are the key to the translation, and with them in mind we can describe what the Kodkod problem resulting from a Relif specification looks like:

- The *universe* will consist of elements which we will interpret as the **atoms of a relation algebra**.
- Our *relation declarations* are as follows:

  A binary relation `conv`, which we will interpret as relating an atom to its converse.

A ternary relation `cycles`. This relation will contain the consistent triples: we will interpret a tuple $(e_1, e_2, e_3) \in$ `cycles` as the statement $a_3 \leq a_1;a_2$, where a_i is the atom that we associate to the element e_i.

Unary relations (sets) named `Ids`, `Syms` and `Asyms`, which will hold the identity, symmetric and asymmetric atoms.

A unary relation `At`, which corresponds to the unit relation 1 in relation algebra, and will hold all the elements that correspond to atoms of the solutions. Thus `At = Ids + Syms + Asyms`.

One unary relation (set) for every relational variable in the Relif specification: every element of the relation algebra is determined by the set of atoms under it, which will correspond to the elements in these sets in the Kodkod solution. Additionally, `Ids` is the Kodkod relation that corresponds to the Relif relation `iden`, and `At` corresponds to the Relif relation `univ`.

- Our *formulas* must express both the axiomatization which will result in a correct set of atoms, consistent triples, etc., and the constraints expressed by the user in the Relif specification. We also make use of Kodkod's *relational bounds* for this purpose. We will describe this at length below.

Translating expressions

In what follows we will write Kodkod formulas with the Alloy syntax: even though Kodkod itself doesn't have a language (the formulas are built through its API), the translation is direct, as the API was designed specifically to work with Alloy's grammar.

Every statement in a Relif specification will have its counterpart in the Kodkod translation. Table 3.6 summarizes the translation of the relational expressions, (`E, E1, E2` are arbitrary relational expressions). In the translation we use the special Kodkod relations that we described above and are common to all Relif problems (`Ids, At, conv, cycles`). Every relational expression results, after translation, in an expression that reduces to a Kodkod set included in `At`, this is, in a set of atoms. Thus, we characterize an element of the algebra with the atoms under it (which is sound in finite relation algebras).

Relif expression e	**Kodkod translation** $t(e)$
`R` (a relational variable)	`R’` (a corresponding Kodkod relational variable)
`iden`	`Ids`
`univ`	`At`
`E1 + E2`	t(`E1`) + t(`E2`)
`E1 & E2`	t(`E1`) & t(`E2`)
`E1;E2`	`cycles[`t(`E1`),t(`E2`)`]`
`-E`	`At -` t(`E`)
`~E`	`conv[`t(`E`)`]`

Table 3.6: Translating the expressions to Kodkod

An expression of the form $R[E_1, \cdots, E_n]$, where R is a $(n+1)-$ary relation and $E_1 \cdots E_n$ are expressions which reduce to a set, is equivalent to the set (unary relation):

$$\{y : (x_1, \cdots, x_n, y) \in R, \text{ where } x_i \in E_i \text{ for } 1 \leq i \leq n\}$$

With this in mind (and that our elements are atoms) we can interpret `conv` and `cycles` applied this way as *totally additive* operations on the atoms (i.e, of the underlying boolean algebra given by the sets and set-theoretic operations of Kodkod).

Translating facts

The facts are translated to corresponding formulas in Kodkod in the obvious manner (Kodkod can construct formulas with the same logical connectives that are used in the Relif facts).

Enforcing the relation algebra conditions

To ensure that we will end up with a valid relation algebra we impose the conditions from Lemma 1.5.2.1. The condition that the set of atoms with the boolean operations will constitute a boolean algebra, and that ; and $\smile$ will be completely additive are guaranteed by our translation and the relational semantics of Kodkod: the + will correspond to set unions, − to complement with respect to `At` (the unit), And $\smile$ and ; will be given by the `conv` and `cycle` "operations" (which, as we mentioned, behave additively).

The conditions $(A1)$ to $(A4)$ from lemma 1.5.2.1, which complete the characterization, are added as part of the *Kodkod formula.*

Axiom	Kodkod Formula
$x = x; 1'$	`x = cycles[x, Ids]`
$z \leq x; y \iff y \leq x^{\smile}; z$	`z in cycles[x,y] iff y in cycles[conv[x], z]`
$z \leq x; y \iff x \leq z; y^{\smile}$	`z in cycles[x,y] iff x in cycles[z, conv[y]]`

$(v; x) \cdot (w; y) \neq 0 \iff (v^{\smile}; w) \cdot (x; y^{\smile}) \neq 0$	`some (cycles[v,x] & cycles[w,y]) iff`
	`some (cycles[conv[v],w] & cycles[x,conv[y]])`

Table 3.7: Kodkod formulas expressing the conditions from lemma 1.5.2.1

The variables on the left column in Table 3.7 are quantified over all atoms. We respectively quantify the variables in the Kodkod formulas of Table 3.7 over the set `At`, which is the one that will include all the atoms in the Kodkod solution.

By including these formulae in every translation from Relif to Kodkod we can guarantee that the solution that comes back from the solver will effectively represent a relation algebra.

Describing the converse relation

When we described Relif's bounding expression we noted that the identity, symmetric and asymmetric atoms are bound separately: the reason for this is that internally we explicitly differentiate the atoms in those categories. The reason is that this lets us determine $\smile$ on the atoms beforehand: we know that $a^{\smile} = a$ for any identity and symmetric atoms, and we explicitly relate elements of the asymmetric group as converses.

For example, given a bounding expression "`1 id, 2 sym, 4 asym`", the Relif backend generates three different groups of atoms: `{I0}`, with the sole element we will interpret as the identity atom, `{S0, S1}`, interpreted as the symmetric atoms, and `{A0, A1, A2, A3}`, the asymmetric atoms. We then explicitly determine the converses:

$$I0^{\smile} = I0$$

$$S0^{\smile} = S0$$

$$S1^{\smile} = S1$$

$$A0^{\smile} = A2$$

$$A1^{\smile} = A3$$

$$A2^{\smile} = A0$$

$$A3^{\smile} = A1$$

The universe of our Kodkod problem will then have the elements `I0, S0, S1, S2, A0, A1, A2, A3`. As the Relif bound is not exact, it is acceptable if some of these elements are not present in the solutions, so what we do is **bound from above** the `conv` relation with the corresponding tuples: `(I0, I0), (S0, S0), (S1, S1), (A0, A2), (A1, A3), (A2, A0), (A3, A1)`. This will guarantee that the `conv` relation in every solution will be faithful to our (external) convention of converses.

Putting it all together

Table 3.8 summarizes the bounds imposed to the different relations. `Ids, Syms, Asyms` and `conv` are bounded as discussed, with the explicit categorization of the atoms and its converses. If the Relif bounding expression specifies tight bounds (with the use of `exactly`), then they are also bounded from below. The relation `Ids` is always bounded from below with a single element, because the identity relation in any relation algebra must have at least one atom under it ($1' \neq 0$).

The relation `At`, (which will be the unit of the relation algebra found as a solution) may have any element of the universe, as can the relations associated with the user-defined variables.

Kodkod relation	**Upper bound**
`At`	*(none)*
`Ids`	Identity atoms
`Syms`	Symmetric atoms
`Asyms`	Asymmetric atoms
`conv`	Externally defined converses
`cycles`	*(none)*
User defined relational variables	*(none)*

Table 3.8: Upper bounds for Kodkod relations

The final Kodkod formula is a conjunction of

- The formulas for (A1)-(A4) given in table 3.7
- Formulas expressing the fact that `At` is the unit:

  A formula expressing `At = Ids + Syms + Asyms`

  Formulas expressing the fact that the tuples of `conv` and `cycles` must have elements of `At` (which means the closure of the solution under ; and $\smile$)

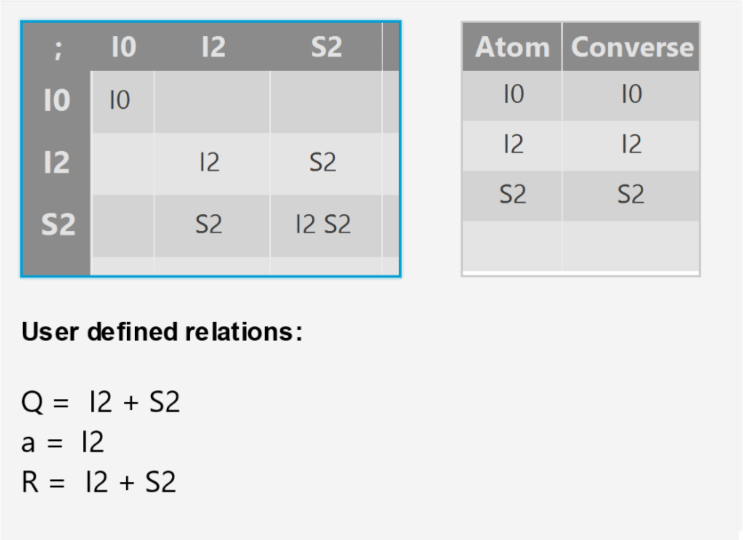
  For every user-defined relational variable `R`, a formula expressing that `R` is included in `At` (which surmounts to say $R \leq 1$).
- Every formula resulting from a Relif fact, i.e., the actual user's specification

Finding and showing solutions

With the previous explanation the description of the translation from a Relif specification to a Kodkod problem is complete. The resulting problem is promptly given to the Kodkod solver, which returns an iterator of the solutions (which can be empty). Any solution found will form a relation algebra with universe `At`, identity element `Ids`, the converse defined by `conv`, and the consistent triples found in the ternary relation `cycles`. Together, the algebra and the user-defined relations, which are elements of the algebra (and are represented by the atoms under them) satisfy the specification.

The user is presented with this information in the form of a table representing the composition of atoms (the consistent triples), and information on the user-defined elements and converses.

The identity atoms always named starting with an I, symmetric atoms with an S and asymmetric atoms with an A. For the example specification of Figure 3.1, one of the solutions found is presented as shown in Figure 3.2.



;	I0	I2	S2
I0	I0		
I2		I2	S2
S2		S2	I2 S2

Atom	Converse
I0	I0
I2	I2
S2	S2

User defined relations:

Q = I2 + S2
a = I2
R = I2 + S2

Figure 3.2: A solution to Figure 3.1 found by Relif.

3.3.2 Finding representations

The second main feature of Relif is the ability to search for representation of the instances found from specifications. This is done with a different invocation to Kodkod, where the problem is now the representation of the graph labeling problem of Theorem 1.5.3.

The problem is bounded by setting a maximum cardinality for the base set of the representation, that is, a maximum number of vertices for the graph to be labelled.

Graph labelling as a Kodkod problem

Specifying the graph labelling problem is fairly simple:

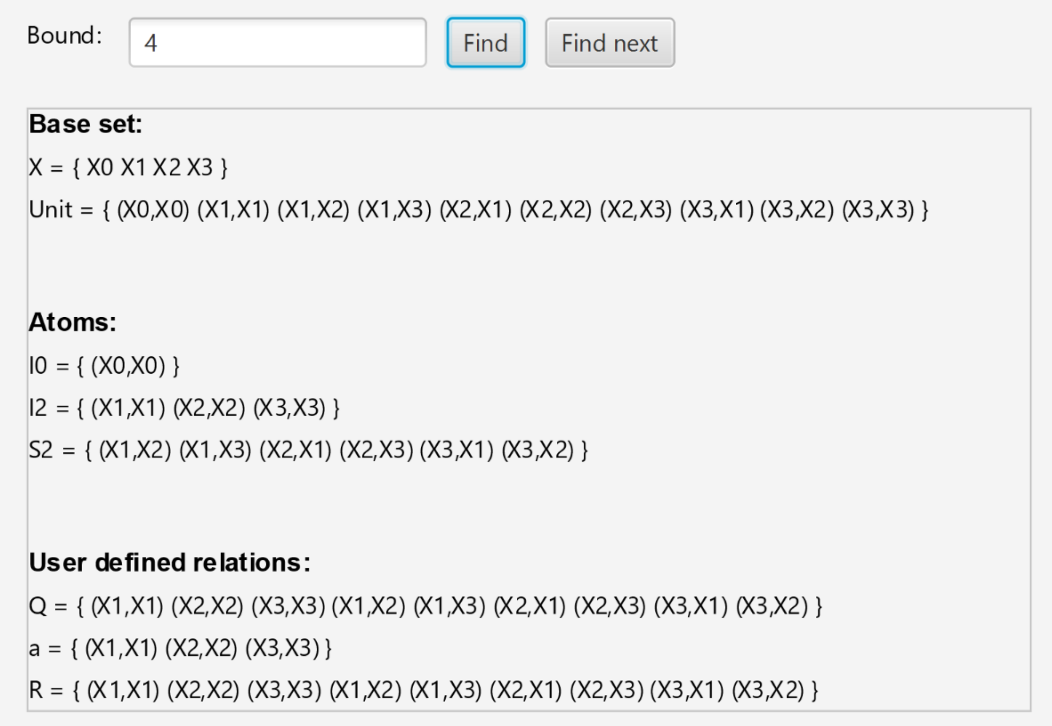
- The *universe* will consist of the atoms of the relation algebra (for labelling purposes) and the possible elements of the base set `X_1, ... , X_k` where k is the upper bound for the cardinality of the base set.
- The *relation declarations* are as follows:

  The relations `At, cycles, conv` that we carry from the relation algebra instance. We need these relations in the problem to express the labelling constraints, but we already know the tuples they contain, so we **bound them exactly** with those tuples.

  A unary relation `X`, which represents the base set. It is bounded from above by `{ X_1, ... , X_k }` which are the possible elements of the base set.

  A ternary relation `labels`. This relation will represent both the edges of the graph and the labelling: if $(x_1, x_2, a) \in$ `labels`, then we will know that (x_1, x_2) is an edge of the graph and it is labelled with a. In the upper bound for `labels` we put all the triples (x, y, a) with $x, y \in$ `X`, and $a \in$ `At`.
- The *formulas* express all the conditions given in Theorem 1.5.3, such that the labelling corresponds to a representation.

There may be multiple solutions for a given bound (and some of them may be isomorphic). The user can iterate through the solutions, and is presented with all the information of the representation, associated with the original specification: not only the elements of each atom are shown but also the elements from all the user-defined relations. Figure 3.3 shows one of the solutions found for the relation algebra given in Figure 3.2.



Base set:

X = { X0 X1 X2 X3 }

Unit = { (X0,X0) (X1,X1) (X1,X2) (X1,X3) (X2,X1) (X2,X2) (X2,X3) (X3,X1) (X3,X2) (X3,X3) }

Atoms:

I0 = { (X0,X0) }

I2 = { (X1,X1) (X2,X2) (X3,X3) }

S2 = { (X1,X2) (X1,X3) (X2,X1) (X2,X3) (X3,X1) (X3,X2) }

User defined relations:

Q = { (X1,X1) (X2,X2) (X3,X3) (X1,X2) (X1,X3) (X2,X1) (X2,X3) (X3,X1) (X3,X2) }

a = { (X1,X1) (X2,X2) (X3,X3) }

R = { (X1,X1) (X2,X2) (X3,X3) (X1,X2) (X1,X3) (X2,X1) (X2,X3) (X3,X1) (X3,X2) }

Figure 3.3: A representation of the algebra in Figure 3.2.

3.3.3 Technical notes

Relif is implemented in Java, and it requires JRE 8 or superior. The scanner/lexer was implemented with **JFlex** (`https://www.jflex.de/`) and the grammar was written for the LALR parser generator **Java CUP** (`http://www2.cs.tum.edu/projects/cup/`).

The code is open source and available on GitHub at `http://github.com/marcelolynch/relif`.

3.4 A non-trivial example

We conclude this chapter with an example that shows that Relif can find instances of specifications that Alloy could never find.

3.4.1 The problem: an unbounded total order

Figure 3.4 shows a specification in Alloy of the fact that the relation R is a total order on some set A. The domain of R is conveniently defined as the pairs $(x, x) \in A$ where $(x, y) \in R$ for some y. The analyzer is then asked to find counterexamples to the assertion "`some iden - Dom[R - iden]`", which is equivalent to the statement "the total order R has a maximum element" (because there is some element outside of the non-reflexive domain of R).

$;$	I	$<$	$>$
I	I	$<$	$>$
$<$	$<$	$<$	$I <>$
$>$	$>$	$I <>$	$>$

Table 3.9: The point algebra, a representable relation algebra

Naturally, there are total orders for which neither minimum nor maximum elements exist. An example is the $\leq$ ordering on the integers. Nevertheless, if the Alloy Analyzer is supplied with such specification and a scope, it will not succeed in finding a counterexample because **finite total orderings** always have maximum elements, and Alloy is only capable of constructing finite instances.

The same specification is given in Relif in Figure 3.5. This time a counterexample *is* found, and in the form of a relation algebra with *only three atoms*. In fact, this relation algebra is called *the point algebra*, and is representable. Table 3.9 shows the composition table for the point algebra, where I is the identity element (which is an atom) and the atoms $<$ and $>$ are converses of each other. This algebra is representable, for example considering $<$ as the usual ordering on the rational numbers (that makes $I + < = \leq$, which is an unbounded total order!) It is clear that the algebra found by Relif is isomorphic to the point algebra, and that the relation R could in fact express the unbounded total order $\leq$.

As a final remark, it is a known fact that the point algebra has no finite representations, and thus Relif, when asked, and even though the algebra *is* representable, will never be able find a representation for the instance in Figure 3.5, because the base set for the representations is always finite. This situation is depicted in Figure 3.7.

```
sig A {
    -- R is a relation on A
    R: set A
}

-- Unit is AxA, the universal relation
fun Unit[] : set A->A { A->A }

-- This function gives the domain of a
-- binary relation on A
fun Dom[r: set A->A] : set A->A { r.~r & iden }

-- Specification: R is a total order
fact { iden in R } -- Reflexivity
fact { R & ~R in iden } -- Antisymmetry
fact { R.R in R } -- Transitivity
fact { ~R + R = Unit[] } -- Totality


-- Alloy can never find a counterexample
check { some iden - Dom[R - iden] } for 5
```

Figure 3.4: An assertion on a total order, in Alloy

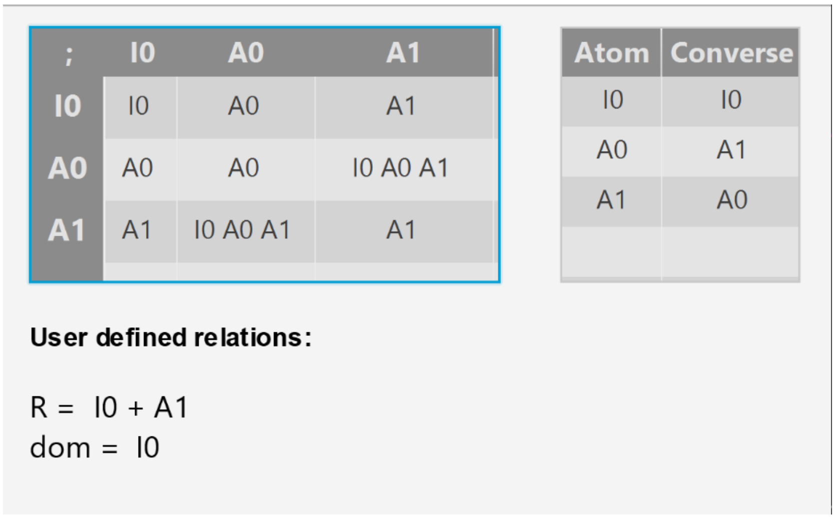
```
rel R, dom

iden in R         // Reflexivity
R & ~R in iden    // Antisymmetry
R.R in R          // Transitivity
R + ~R = univ     // Totality

// Bind dom to the domain of R - iden
dom = ((R - iden).~(R - iden) & iden)

// The same assertion
check { some iden - dom } for 2
```

Figure 3.5: An assertion on a total order, in Relif



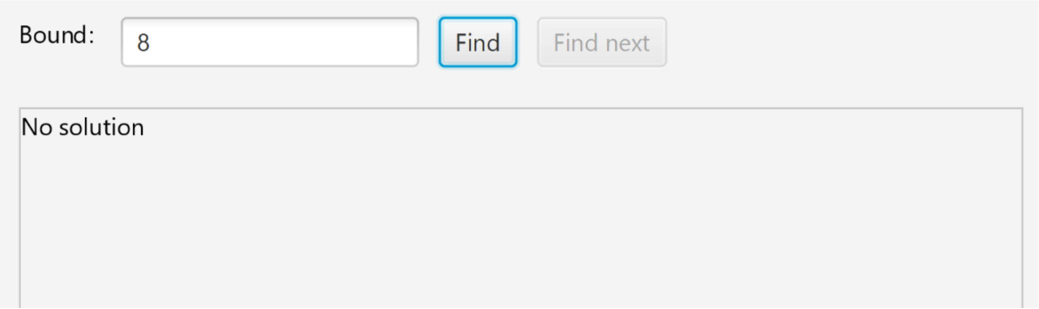
;	I0	A0	A1
I0	I0	A0	A1
A0	A0	A0	I0 A0 A1
A1	A1	I0 A0 A1	A1

Atom	Converse
I0	I0
A0	A1
A1	A0

User defined relations:

R = I0 + A1
dom = I0

Figure 3.6: Relif finds an algebra from the specification in Figure 3.5.



Bound: 8

```
No solution
```

Figure 3.7: Relif will never find a representation for the algebra in Figure 3.6.

Conclusions

Summary

We have presented the Relif tool and justified the correctness of its implementation. Relif's user experience, inspired by Alloy's, has proven useful even for developing and debugging purposes. We have also seen that the expressive power of the specification tool is enough to solve problems intractable by Alloy, which is encouraging.

Another interesting (though tangential) insight gained during the development was the power and flexibility of Kodkod as a problem-solving tool (fittingly based on relational semantics), with a very well documented and well designed API.

Future work

There seems to be little work done with relation algebras and SAT solvers. During the development of Relif we explored the possibility of translating directly to SAT (ultimately opting for going through Kodkod because of the generality of our problem). The boolean nature of SAT solving provides an interesting framework for future work, perhaps with more specific purposes. In particular, the graph labelling problem as a way to find representations seems to be very well suited for an efficient straight-to-SAT implementation.

There is also room for improving Relif. Possible improvements and extensions include:

- Symmetry breaking: reducing the number of isomorphic instances found from a specification and the isomorphic representations given by the representation finder.
- Optimizations, such as better bounds on the `cycle` relation in the Kodkod problem.
- Provide a format for importing and exporting the relation algebras.
- Extend the representation finder to allow searching starting from arbitrary relation algebras and not only the ones found via specifications.
- Add support for other *algebras of relational type* with different or additional operations (Kleene closures, fork algebras).

References

[AL98] Sten Agerholm and Peter Gorm Larsen. A lightweight approach to formal methods. In *International Workshop on Current Trends in Applied Formal Methods*, pages 168–183. Springer, 1998.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[ELC$^+$98] Steve Easterbrook, Robyn Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, 1998.

[FM97] MF Frias and RD Maddux. Non-embeddable simple relation algebras. *algebra universalis*, 38(2):115–135, 1997.

[FVB04] Marcelo Frias, Paulo Veloso, and Gabriel Baum. Fork algebras: past, present and future. *Journal on Relational Methods in Computer Science*, 1:181–216, 2004.

[Giv17] Steven Givant. *Introduction to Relation Algebras: Relation Algebras*, volume 1. Springer, 2017.

[Har03] John Harrison. Formal verification at intel. In *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pages 45–54. IEEE, 2003.

[HH02] Robin Hirsch and Ian Hodkinson. *Relation algebras by games*, volume 147. Elsevier, 2002.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[Jac12] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[Jac19] Daniel Jackson. Alloy: a language and tool for exploring software designs. *To Appear) Communications of the ACM*, 2019.

[LT93] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[Lyn50] Roger C Lyndon. The representation of relational algebras. *Annals of mathematics*, pages 707–729, 1950.

[Mad91] Roger D Maddux. The origin of relation algebras in the development and axiomatization of the calculus of relations. *Studia Logica*, 50(3-4):421–455, 1991.

[Sto38] Marshall H Stone. The representation of boolean algebras. *Bulletin of the American Mathematical Society*, 44(12):807–816, 1938.

[Tar41] Alfred Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3):73–89, 1941.

[TD06] Emina Torlak and Greg Dennis. Kodkod for alloy users. In *First ACM Alloy Workshop, Portland, Oregon*, 2006.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[Tor09] Emina Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications.* PhD thesis, Massachusetts Institute of Technology, 2009.