Automated Verification of Relational While-Programs

Rudolf Berghammer[1], Peter Höfner[2,3], and Insa Stucke[1]

[1] Institut für Informatik, Christian-Albrechts-Universität zu Kiel, Germany
[2] NICTA, Australia
[3] Computer Science and Engineering, University of New South Wales, Australia

Abstract. Software verification is essential for safety-critical systems. In this paper, we illustrate that some verification tasks can be done fully automatically. We show how to automatically verify imperative programs for relation-based discrete structures by combining relation algebra and the well-known assertion-based verification method with automated theorem proving. We present two examples in detail: a relational program for determining the reflexive-transitive closure and a topological sorting algorithm. We also treat the automatic verification of the equivalence of common-logical and relation-algebraic specifications.

1 Introduction

Many discrete structures of mathematics and computer science, such as orders, lattices, certain classes of graphs, Petri nets, and games, are relations or can easily be modelled by means of relations. In such cases computational tasks frequently reduce computations on relations and the correctness proofs of the corresponding algorithms to proofs of statements over relations.

In the past, various techniques for programming with relations have been proposed. In this paper, we follow an approach that considers relations only as data structures and manipulates them with a simple, imperative programming language. It is straightforward to translate the relational programs into more efficient programming languages such as Java or C. The approach also bears methodical advantages: if problem specifications are expressed via relation-algebraic formulae, then the correctness proofs allow to intertwine approved program verification steps with formal and precise relation-algebraic calculations. This mathematical rigour drastically reduces errors in the programs. Moreover, this approach is supported by tools for (a) prototyping and testing, (b) interactive theorem proving, and (c) automatic theorem proving. An example for prototyping and testing relation-algebraic specifications is RelView (cf. [30]), which allows the evaluation of relation-algebraic expressions and the formulation of relational programs. With regard to interactive theorem proving either special purpose systems, such as RALF (see [14]), can be used, or relation-algebraic techniques can be integrated into existing provers (for example into Isabelle/HOL as described in [28,12]). Full automatisation of proofs can frequently be achieved by off-the-shelf automated theorem provers, such as Prover9 (see [31]). We refer to [16] for such an application. In the present paper, we will follow the latter approach and use Prover9 for automated program verification of while-programs.

Formal verification of imperative programs is often done by use of pre- and post-conditions as problem specifications, and loop-invariants; see e.g., [10,11,13]. This so-called assertion-based technique is particularly useful for while-loops, where it is sufficient to show that the loop-invariant is established and maintained (under the assumption that the pre-condition holds) and that the post-condition is valid as soon as the while-loop terminates. The combination of program verification and relation algebra we are going to use is not new; it was applied in several case studies, for instance in [1,2,3,4].

Encouraged by the practicability and elegance of the latter results and the positive experiences of [16], the combination of assertion-based program verification and relation algebra was combined with automated theorem proving using Prover9; see [5]. This paper is a continuation as well as a step further of this idea. We consider two new and more sophisticated examples, viz. the computation of reflexive-transitive closures by means of decomposition and the computation of topological sortings in case of cycle-free relations. We further demonstrate how the equivalences of the logical specifications and their relation-algebraic counterparts can automatically be verified using Prover9. The paper closes with a short discussion on the lessons we have learned from the two case studies.

2 Preliminaries

In this paper, we formalise data structures and assertions of imperative programs by homogeneous relation algebra, as axiomatised by Tarski in [26]. The pre-conditions, post-conditions, loop-invariants and proof obligations will be formalised via expressions and formulae in relation algebra and implemented in Prover9. In this section, we recapitulate the basic concepts of the relational calculus and its automation via automated theorem proving, which are needed later on. For more details we refer to [22,23] concerning relation algebra, to [31] concerning Prover9, and to [7,24] concerning the use of automated theorem proving in general software engineering.

2.1 (Homogeneous) Relation Algebra

Homogeneous relation algebra was first axiomatised in [26] and further developed in [8,27]. A relation R over a set $\mathcal{X}$, the *universe*, is a subset of the direct product $\mathcal{X} \times \mathcal{X}$. Relation algebra offers five operations on relations, viz. $R \cup S$ (union), $R \cap S$ (intersection), $\overline{R}$ (complement), $R;S$ (composition) and R^{T} (transposition), two predicates to compare relations, viz. $R \subseteq S$ (inclusion) and $R = S$ (equality), and three special relations: O (empty relation), L (universal relation), and I (identity relation). Except composition, transposition and the identity relation all concepts are defined by standard set theory. The composition $R;S$ of two relations R and S is the set of all pairs $(x, y) \in \mathcal{X} \times \mathcal{X}$ such that

$(x,z) \in R$ and $(z,y) \in S$ for some $z \in \mathcal{X}$, the transposition R^T is the set of all pairs $(x,y) \in \mathcal{X} \times \mathcal{X}$ with $(y,x) \in R$, and the identity relation I is the set of all pairs $(x,y) \in \mathcal{X} \times \mathcal{X}$ with $x = y$.

These definitions form the base of *concrete relation algebras*. An *(abstract) relation algebra* abstracts from set theory and is axiomatised as follows, where we follow the axiomatisation of [22] instead of [8,26,27].

1. With regard to $\overline{}$, $\cup$, $\cap$, the order $\subseteq$, and the constants O and L the relations form a Boolean algebra.
2. With regard to composition and the identity relation I the relations form a monoid.
3. The *Dedekind rule* holds, i.e., for all relations Q, R and S we have

$$Q;R \cap S \subseteq (Q \cap S;R^\mathsf{T});(R \cap Q^\mathsf{T};S)\ . \tag{1}$$

Since all axioms are first-order, it is easy to encode them in any off-the-shelf automated theorem provers.

From the Dedekind rule we obtain the so-called *Schröder equivalences* (also known as "Theorem K" of de Morgan). They state that

$$Q;R \subseteq S \ \Leftrightarrow\ Q^\mathsf{T};\overline{S} \subseteq \overline{R} \qquad\qquad Q;R \subseteq S \ \Leftrightarrow\ \overline{S};R^\mathsf{T} \subseteq \overline{Q} \tag{2}$$

for all relations Q, R, and S. The Schröder equivalences are equivalent to the Dedekind rule (see e.g., [22]).

Using relation algebra, we now recapitulate some fundamental classes of relations. These will be used in the remainder of the paper.

A relation R is called *reflexive* if $\mathsf{I} \subseteq R$ and *transitive* if $R;R \subseteq R$. The least reflexive and transitive relation containing R is its *reflexive-transitive closure* R^*, specified by the laws $\mathsf{I} \cup R;R^* = R^*$ and $R;Q \cup S \subseteq Q \Rightarrow R^*;S \subseteq Q$ or, equivalently, by the laws $\mathsf{I} \cup R^*;R = R^*$ and $Q;R \cup S \subseteq Q \Rightarrow S;R^* \subseteq Q$ to hold for all relations Q, R, and S. A relation R is *antisymmetric* if $R \cap R^\mathsf{T} \subseteq \mathsf{I}$ and in combination with the above formulae this allows to characterise *partial order relations* R by $\mathsf{I} \subseteq R$, $R;R \subseteq R$, and $R \cap R^\mathsf{T} \subseteq \mathsf{I}$. A partial order relation R is called a *linear order relation* if additionally $R \cup R^\mathsf{T} = \mathsf{L}$ holds. A relation v satisfying $v = v;\mathsf{L}$ is called a *vector*. In case of a set-theoretic (i.e., concrete) relation $v \subseteq \mathcal{X} \times \mathcal{X}$ this equation means that an element $x \in \mathcal{X}$ is either in relationship to none of the elements of $\mathcal{X}$ or to all elements of $\mathcal{X}$. Due to this property, vectors can be used to model subsets of the universe $\mathcal{X}$. We say that $v \subseteq \mathcal{X} \times \mathcal{X}$ models the subset Y of $\mathcal{X}$ if for all $x, y \in \mathcal{X}$ we have that $x \in Y$ iff $(x,y) \in v$. By definition, a *point* is an injective and surjective vector, i.e., a vector p such that the two properties $\mathsf{L};p = \mathsf{L}$ and $p;p^\mathsf{T} \subseteq \mathsf{I}$ hold. In case of a set-theoretic point $p \subseteq \mathcal{X} \times \mathcal{X}$ these properties mean that it models a singleton subset $\{x\}$ of $\mathcal{X}$, i.e., the element x of the universe if we identify the singleton set $\{x\}$ with the only element x it contains.

2.2 Automating Relation Algebra

Automated/mechanised reasoning is not a new challenge, but has been performed since more than 20 years. Interactive theorem provers for relation algebras have been implemented (see e.g., [28,17]) and relational techniques have been integrated into various proof checkers for B or Z. Special purpose first-order proof systems for relation algebras, including tableaux and Rasiowa-Sikorski calculi, have been proposed as well (e.g., in [20]). Translations of relation-algebraic formulae into (undecidable) fragments of predicate logics have been implemented (see [25]) and integrated into the theorem prover SPASS (see [29]). However, it has been shown that automated reasoning with relation algebra does not need special-purpose tools nor interaction. As demonstrated in [16,5], an off-the-shelf automated theorem prover, such as Prover9, is often sufficient.

In this paper, we follow the latter approach and encode relation algebra in Prover9, which is a saturation-based automated theorem prover for first-order logic with equality. An evaluation of various automated theorem provers has shown that in our context Prover9 is currently best suited for verifying properties in relation algebra; see [9]. We also have experimented with the interactive theorem prover Isabelle/HOL. However, for our specific purpose the proof-effort of interactive theorem provers presently seems to be too high. Moreover, we believe that they often require a rather deep understanding of the used tool and hence experienced user, whereas our approach also can be used by people mainly interested in relation algebra and not in theorem proving.

Prover9 implements a first-order resolution and paramodulation calculus. Equalities are handled via rewriting rules and Knuth-Bendix completion. The tool suite also offers the counterexample generator Mace4, which is very useful in practice. The encoding of relation-algebraic formulae in Prover9 is straightforward. For example the Dedekind rule (1) can be written as follows:

```
all Q all R all S (Q * R /\ S <= (Q /\ R * S^) * (R /\ Q^ * S)).
```

Since Prover9 allows only ASCII symbols as input, we use the symbols `\/`, `/\`, `*`, `_^`, `_'`, `<=` and `rtc(_)` for union, intersection, composition, transposition, complement, inclusion, and the operator for reflexive-transitiv closure, respectively. An entire input template can be found in the appendix.

Prover9 does not support types. Hence we define the following two predicates to characterise relations as vectors and points; they are nothing else than the translations of the definitions of Section 2.1 into the language of Prover9:

```
vector(R) <->  R = R*L.
point(R)  <-> (R = R*L & L*R = L & R*R^ <= I).
```

Prover9, as any other automated theorem proving system, heavily depends on the axioms given as input. In case one only uses the few axioms of relation algebra given in Section 2.1, an automated theorem prover has to derive each and every relation-algebraic fact used in a proof. For example, if a distributivity law is needed for a proof, Prover9 has to derive it first. This fact does not only increase the running times of the theorem prover, but sometimes even yields failure in the proof search. Due to this, suitable and well-known facts, such as the following distributivity laws, should be added as axioms.

```
all R all S all T ((R \/ S)*T = R*T \/ S*T).
all R all S all T (T*(R \/ S) = T*R \/ T*S).
```

Other examples for useful relation-algebraic facts concern transposition, such as the following formulae:

```
all R (R^^ = R).
all R all S ((R * S)^ = S^ * R^).
```

For the proof automatisation we use a suitable (fixed) set of axioms. In case we need some special fact as additional input, we will state it. All input files can be found at the webpage `http://hoefner-online.de/ramics14/`. Running times presented in this paper are w.r.t. a standard desktop PC equipped with a 3.1 GHz Intel Pentium 5 CPU, 16 GB main memory, running a Mac OS operating system.

3 Automation of Proof Obligations

We start with a description of our general approach to the automation of the assertion-based verification of relational programs. Then we consider two examples. All formulae appearing in the verifications are relation-algebraic ones, but usually the notions in question are specified by predicate-logical means. To connect these two kinds of specifications, we finally show how to automatically verify the equivalence of the relation-algebraic and the common-logical specifications.

3.1 Verification of Relational While-Programs

In the present paper, we treat imperative programs with relations as data type. Concretely this means that the constants, operations and predicates of relation algebra, as introduced in Section 2.1, are available. Furthermore, we consider while-programs of the following specific form only:

$$\begin{array}{l} \boldsymbol{x} := \boldsymbol{I}(\boldsymbol{\alpha}); \\ \textbf{while } B(\boldsymbol{\alpha}, \boldsymbol{x}) \textbf{ do} \\ \quad \boldsymbol{x} := \boldsymbol{E}(\boldsymbol{\alpha}, \boldsymbol{x}) \textbf{ od} \end{array} \tag{W}$$

This specific form is only chosen for simplifying program verification. There are no problems on the conceptional side to handle more complicated programs, like those of [3]. Whether the presented approach scales to larger programs, i.e., whether automated theorem provers are able to automatically verify larger relational programs, is part of future work. However, at this place it should be mentioned that relational programs are often small. This is due to the fact that relation-algebraic expressions frequently allow concise descriptions of computations which in conventional programming languages usually are expressed by, for example, (nested) loops.

In the while-program (W) $\boldsymbol{x}$ denotes a non-empty list $x_1, \dots, x_n$ of variables for relations. Furthermore, $\boldsymbol{\alpha}$ denotes a list of input relations and by $\boldsymbol{I}(\boldsymbol{\alpha})$ and $I_1(\boldsymbol{\alpha}), \dots, I_n(\boldsymbol{\alpha})$ a list of relation-algebraic expressions over the input relations. So, the collateral assignment $\boldsymbol{x} := \boldsymbol{I}(\boldsymbol{\alpha})$ describes the initialisation of the variables. $\boldsymbol{E}(\boldsymbol{\alpha}, \boldsymbol{x})$ denotes a list $E_1(\boldsymbol{\alpha}, \boldsymbol{x}), \ldots, E_n(\boldsymbol{\alpha}, \boldsymbol{x})$ of relation-algebraic expressions, but now over the input relations and the variables. Finally, $B(\boldsymbol{\alpha}, \boldsymbol{x})$ denotes a quantifier-free formula built over the vocabulary of relation algebra, the input relations, and the variables, usually an inclusion or an equation. It is called the loop-condition. As long as it evaluates to true, the loop-body $\boldsymbol{x} := \boldsymbol{E}(\boldsymbol{\alpha}, \boldsymbol{x})$, again a collateral assignment, is executed.

A problem specification consists of a pre-condition $\mathit{Pre}(\boldsymbol{\alpha})$ and a post-condition $\mathit{Post}(\boldsymbol{\alpha}, \boldsymbol{x})$. The pre-condition describes the input restrictions and the post-condition describes the result(s) which should be computed. In our case both conditions are formulated within the language of relation algebra, frequently as conjunctions of relation-algebraic inclusions and equations. A given algorithm (a while-program) is *partially correct* if it satisfies the post-condition after termination, in case that the pre-condition holds. It is *totally correct* if it is partially correct and also guarantees termination, provided the pre-condition holds.

To prove that a program of the presented form (W) is totally correct w.r.t. a given problem specification, we use the inductive assertion method (see e.g., [10,11,13]). This method consists of three major steps: (a) the identification of a *loop-invariant* $\mathit{Inv}(\boldsymbol{\alpha}, \boldsymbol{x})$, (b) the verification of three *proof obligations*, viz. that the loop-invariant is established by the initialisation, maintained by the loop-body, and that the loop-invariant together with the negated loop-condition implies the post-condition, and (c) the *termination* of the program.

Since we are looking at relational while-programs, the loop-invariant $\mathit{Inv}(\boldsymbol{\alpha}, \boldsymbol{x})$ is also formulated within the language of relation algebra. The three proof obligations of (b) then may be formalised by three implications over $\mathit{Pre}(\boldsymbol{\alpha})$, $\mathit{Post}(\boldsymbol{\alpha}, \boldsymbol{x})$, $\mathit{Inv}(\boldsymbol{\alpha})$ and $B(\boldsymbol{\alpha}, \boldsymbol{x})$. The first one,

$$\mathit{Pre}(\boldsymbol{\alpha}) \Rightarrow \mathit{Inv}(\boldsymbol{\alpha}, \boldsymbol{I}(\boldsymbol{\alpha})) \tag{PO1}$$

says that, if the pre-condition holds, then the loop-invariant has to be established by the initialisation of the variables. After it has been shown that the loop-invariant is established, it needs to be maintained during all runs through the loop. This is formally expressed by the implication

$$\mathit{Inv}(\boldsymbol{\alpha}, \boldsymbol{x}) \wedge B(\boldsymbol{\alpha}, \boldsymbol{x}) \Rightarrow \mathit{Inv}(\boldsymbol{\alpha}, \boldsymbol{E}(\boldsymbol{\alpha}, \boldsymbol{x}))\ . \tag{PO2}$$

The implication that formalises the third proof obligation is

$$\mathit{Inv}(\boldsymbol{\alpha}, \boldsymbol{x}) \wedge \neg B(\boldsymbol{\alpha}, \boldsymbol{x}) \Rightarrow \mathit{Post}(\boldsymbol{\alpha}, \boldsymbol{x})\ . \tag{PO3}$$

It expresses that if the while-loop terminates, i.e., $B(\boldsymbol{\alpha}, \boldsymbol{x})$ does not hold any longer, then the loop-invariant has to imply the post-condition. Since we are interested in total correctness, we also want to prove (correct) termination, i.e.

$$\mathit{Pre}(\boldsymbol{\alpha}) \Rightarrow \text{the program yields a defined vaule.} \tag{T}$$

Usually, (correct) termination of (W) means that its while-loop terminates after a finite number of iterations. However, our instantiations of (W) use a specific partial operation *point* on relations, as we will see in later sections. Therefore, a proof of (T) requires, besides the termination of the while-loop, the verification that each application of *point* yields a defined value.

Unfortunately, it is well known that termination is undecidable. However, in specific cases the termination of while-loops can be proven by measure functions. A measure function δ maps program states into a Noetherian pre-order such that, with the above notations, $\mathit{Pre}(\boldsymbol{\alpha})$ and $B(\boldsymbol{\alpha}, \boldsymbol{x})$ imply $\delta(\boldsymbol{E}(\boldsymbol{\alpha}, \boldsymbol{x})) < \delta(\boldsymbol{x})$. By this, every execution of the loop-body strictly decreases the measure and, hence, termination of the while-loop is guaranteed. In this paper, we will not only show how proofs of partial correctness can be automatised, we will also show that, under some circumstances, total correctness proofs can be supported.

3.2 Reflexive-transitive Closure

The first algorithm we verify with the help of Prover9 is an algorithm for computing the reflexive-transitive closure R^* for a given relation R. It is obtained by transforming the functional program of [6] into the following while-program (P1). In the program (P1) a (partial) operation *point* is assumed to be at hand that selects a point from a non-empty vector. The operation *point* is deterministic. In RelView the deterministic selection of a point via the pre-defined operation *point* is done using the internal enumeration of the universe $\mathcal{X}$.

```
$C, v := \mathsf{I}, \mathsf{O}$;
while $v \neq R;\mathsf{L}$ do
    let $p = \mathit{point}(R;\mathsf{L} \cap \overline{v})$;                (P1)
    $C, v := C \cup C;p;p^{\mathsf{T}};R;C, v \cup p$ od
```

The program (P1) uses two variables: C for computing the result and v, a vector, for looping through all points of the range $R;\mathsf{L}$ of R. To enhance readability, it uses a let-clause.[1]. The selected point $\mathit{point}(R;\mathsf{L} \cap \overline{v})$ is denominated with the letter p for its threefold use in the subsequent assignment. If, in case of set-theoretic relations, v models the subset V of the universe $\mathcal{X}$, then the chosen point p models an element x of the set $\mathcal{X} \setminus V$ that possesses at least one successor w.r.t. R, and the subrelation $p;p^{\mathsf{T}};R$ of R consists precisely of those pairs (y, z) of R, for which $y = x$ holds. Although the program (P1) is deterministic, in principle it does not matter which element x is chosen, as long as it was not handled before and has at least one successor. For the verification we only need the following properties (3) specifying p as a point contained in the vector $R;\mathsf{L} \cap \overline{v}$.

$$p;\mathsf{L} = p \qquad \mathsf{L};p = \mathsf{L} \qquad p;p^{\mathsf{T}} \subseteq \mathsf{I} \qquad p \subseteq R;\mathsf{L} \cap \overline{v} \ . \tag{3}$$

There is no requirement on the input relation R. So, the pre-condition $\mathit{Pre}(R)$ equals **true**. The post-condition $\mathit{Post}(R, C)$ depends on the input R and the

[1] We consider the let-clause as syntactical suger only, since the replacement of each occurrence of p in the body of the while-loop of (P1) by $\mathit{point}(R;\mathsf{L} \cap \overline{v})$ and the removal of the let-clause transforms (P1) into the schematic form (W).

Table 1. Auxiliary Facts for Verification

Formula	**Running Time**
$p;\mathsf{L} = p \wedge \mathsf{L};p = \mathsf{L} \wedge p;p^{\mathsf{T}} \subseteq \mathsf{I} \Rightarrow R \cap p = p;p^{\mathsf{T}};R$	73 s
$p;\mathsf{L} = p \wedge \mathsf{L};p = \mathsf{L} \wedge p;p^{\mathsf{T}} \subseteq \mathsf{I} \Rightarrow (R \cap p);\mathsf{L};(R \cap p) \subseteq R \cap p;\mathsf{L}$	248 s
$S;\mathsf{L};S \subseteq S \Rightarrow (R \cup S)^* = R^* \cup R^*;S;R^*$	184 s

(output) variable C and is $C = R^*$, since we want to compute the reflexive-transitive closure of R. Transferring an idea of [6] to the imperative paradigm, we obtain the conjunction of the two equations of (4) as loop-invariant $\mathit{Inv}(R, C, v)$.

$$C = (R \cap v)^* \qquad\qquad v = v;\mathsf{L} \tag{4}$$

The first equation is best described in the Boolean matrix model of relations. It says that C equals the reflexive-transitive closure of the relation (matrix), that is obtained from R by replacing those rows by zero-rows (all entries are zero) where v consists of zeros only.

As discussed in the previous section, it suffices to verify the proof obligations (PO1) to (PO3) to show the partial correctness of the program (P1). Prover9 shows the corresponding instantiation of (PO1) in no time (0 s). Proving the corresponding instantiation of (PO3) is as simple and does not cost time either. In contrast to these cases proving the corresponding instantiation of (PO2), i.e., the maintenance of the loop-invariant, is more complicated. Here, the main goal is to show that under the assumptions of (3) and $v \neq R;\mathsf{L}$ it holds

$$C = (R \cap v)^* \;\Rightarrow\; C \cup C;p;p^{\mathsf{T}};R;C = (R \cap (v \cup p))^* \ . \tag{5}$$

Unfortunately, Prover9 is not able to prove the implication (5) from scratch within 1000 s. It does not have sufficient knowledge about the Kleene star. The theorem prover needs additional properties of this operation as input. Adding auxiliary laws, such as star-monotonicity does not help. One needs further specific knowledge about the Kleene star in relation algebra. In [6] the laws listed in Table 1 are used to prove the correctness of the functional program. If these three laws are added, then Prover9 proves (5) and the entire instantiation of (PO2) within 1 s. Luckily, the additional laws can all be proven fully automatically. The running times are presented in Table 1.

The proof of (5) is by far not trivial (even with the additional properties), but definitely shows some limitation of our approach. It cannot be expected that all proofs can be automated. In fact, it is well known that theorem proving in the area of relation algebra is undecidable. However, Prover9 (or any other automated theorem prover) can assist to get rid of proofs of low or medium complexity. The user can then concentrate on the more complicated proofs, such as the maintenance of the loop-invariant of the algorithm under consideration. As we will show in the next section, sometimes even all proofs can be automated. A longer discussion about lessons learned is given in Section 4.

So far we have established partial correctness only. However, we can even show total correctness with the help of Prover9. To prove total correctness, we have to show that the while-loop of the program (P1) terminates and each of its *point*-calls is defined. We use the values of the variable v as measure function and use Prover9 to verify the inclusion

$$v \subseteq R;\mathsf{L} \tag{6}$$

as well as the universally quantified implication

$$\forall p : \; p;\mathsf{L} = p \wedge \mathsf{L};p = \mathsf{L} \wedge p;p^{\mathsf{T}} \subseteq \mathsf{I} \wedge p \subseteq R;\mathsf{L} \cap \overline{v} \Rightarrow v \subset v \cup p \; . \tag{7}$$

With the help of the inclusion (6) and contraposition it is easy to prove that $R;\mathsf{L} \cap \overline{v} \neq \mathsf{O}$ if $v \neq R;\mathsf{L}$: From $R;\mathsf{L} \cap \overline{v} = \mathsf{O}$ we get $R;\mathsf{L} \subseteq v \subseteq R;\mathsf{L}$ and this yields $v = R;\mathsf{L}$. As a consequence, each call $\mathit{point}(R;\mathsf{L} \cap \overline{v})$ in the program (P1) is defined. The formula (7) states that under the assumptions of (3) the vector v grows strictly. If the universe $\mathcal{X}$ is finite, then (6) and (7) together imply that the while-loop terminates, since v is strictly enlarged by every execution of its body but it cannot exceed $R;\mathsf{L}$. The two properties (6) and (7) constitute again a loop-invariant and, using Prover9, it can successfully be treated in the same fashion as the previous loop-invariant (4). We summarise the results in Table 2.

3.3 Topological Sorting of Cycle-free Relations

A *topological sorting* of a cycle-free relation R is a linear order relation that contains R. The relational program we consider in this section stems from [4] and is the relational version of Kahn's well-known algorithm for computing topological sortings (see [18]). It uses two variables, S for computing the result and v as auxiliary vector variable for the while-loop, and looks as follows:

```
$S, v := \mathsf{I}, \mathsf{O}$;
while $v \neq \mathsf{L}$ do
    let $p = \mathit{point}(\overline{v} \cap \overline{(R^{\mathsf{T}} \cap \overline{\mathsf{I}});\overline{v}})$;        (P2)
    $S, v := S \cup v;p^{\mathsf{T}}, v \cup p$ od
```

Similar to program (P1), program (P2) also uses a let-clause to improve readability. It introduces p as a name for the point chosen from the vector $\overline{v} \cap \overline{(R^{\mathsf{T}} \cap \overline{\mathsf{I}});\overline{v}}$ via the operation *point*. If R and v are set-theoretic relations and the

Table 2. Running Times for Termination Proofs

Formula	Running Time
$\mathsf{O} \subseteq R;\mathsf{L}$	0 s
(3) $\wedge\, v \subseteq R;\mathsf{L} \Rightarrow v \cup p \subseteq R;\mathsf{L}$	1 s
(3) $\wedge\, v \neq R;\mathsf{L} \Rightarrow v \subseteq v \cup p$ [2]	0 s
(3) $\wedge\, v \neq R;\mathsf{L} \Rightarrow v \neq v \cup p$	0 s

[2] In this example Prover9 finds a proof, but outputs "SEARCH FAILED" followed by "Exiting with 1 proof". A close inspection of the proof logs shows that such situations occur if negative clauses are included in the goals. Then the output is misleading, since in such cases Prover9 *did* find a proof, but thought it had to keep searching.

Table 3. Invariants for Topological Sorting

Name	Invariant	Name	Invariant
$\mathit{Inv}_0(v)$	$v;\mathsf{L} \subseteq v$	$\mathit{Inv}_4(S)$	$\mathsf{I} \subseteq S$
$\mathit{Inv}_1(S,v)$	$S;v \subseteq v$	$\mathit{Inv}_5(S)$	$S \cap S^\mathsf{T} \subseteq S$
$\mathit{Inv}_2(S,v)$	$S \cup S^\mathsf{T} = v;v^\mathsf{T} \cup \mathsf{I}$	$\mathit{Inv}_6(S)$	$S;S \subseteq S$
$\mathit{Inv}_3(R,S,v)$	$R \cap v;v^\mathsf{T} \subseteq S$	$\mathit{Inv}_7(R,v)$	$R;v \subseteq v$

vector v models the subset V of the universe $\mathcal{X}$, then the vector $\overline{v} \cap \overline{(R^\mathsf{T} \cap \overline{\mathsf{I}});\overline{v}}$ models the set of minimal elements of the set $\mathcal{X} \setminus V$. So, via the variable v the program (P2) constructs a chain

$$\emptyset \subset \{x_1\} \subset \{x_1,x_2\} \subset \{x_1,x_2,x_3\} \subset \ldots \subset \{x_1,x_2,\ldots,x_n\} = \mathcal{X} \tag{8}$$

of subsets of $\mathcal{X}$, where for all $i \in \{0,\ldots,n-1\}$ the set $\{x_1,\ldots,x_{i+1}\}$ is obtained from the set $\{x_1,\ldots,x_i\}$ by adding a minimal element of $\mathcal{X} \setminus \{x_1,\ldots,x_i\}$. As before this chain can be used to prove termination later on, if $\mathcal{X}$ is finite. Simultaneously to the chain (8) the program (P2) creates another chain

$$\mathsf{I} = S_0 \subset S_1 \subset \ldots \subset S_n \tag{9}$$

of relations, using the variable S. For all $i \in \{0,\ldots,n\}$, the relation S_i is a topological sorting of the input R if both are restricted to $\{x_1,\ldots,x_i\}$. Because of the initialisation of S, outside of this set S_i consists of loops (x,x) only.

Before we treat the automated verification of the above program, we have to be more precise about the choice of p. If p is a point satisfying $p \subseteq \overline{(R^\mathsf{T} \cap \overline{\mathsf{I}});\overline{v}}$, then $R;p \subseteq v \cup p$ follows by the use of the Schröder equivalences (2) (see [4]). As a consequence, we assume the following properties for p:

$$p;\mathsf{L} = p \qquad \mathsf{L};p = \mathsf{L} \qquad p;p^\mathsf{T} \subseteq \mathsf{I} \qquad p \subseteq \overline{v} \qquad R;p \subseteq v \cup p \tag{10}$$

A topological sorting requires a cycle-free relation as input. Since cycle-freeness of R relation-algebraically can be specified as $R;R^* \subseteq \overline{\mathsf{I}}$, we take this formula as pre-condition $\mathit{Pre}(R)$. In case of a finite universe $\mathcal{X}$ we then get that the relation $R^\mathsf{T} \cap \overline{\mathsf{I}}$ is *progressively finite* in the sense of [22]. Hence, $\overline{v} \subseteq (R^\mathsf{T} \cap \overline{\mathsf{I}});\overline{v}$ implies $\overline{v} = \mathsf{O}$. By contraposition we obtain that $\overline{v} \neq \mathsf{O}$ implies $\overline{v} \not\subseteq (R^\mathsf{T} \cap \overline{\mathsf{I}});\overline{v}$ and this is equivalent to the fact that $v \neq \mathsf{L}$ implies $\overline{v} \cap \overline{(R^\mathsf{T} \cap \overline{\mathsf{I}});\overline{v}} \neq \mathsf{O}$. So, in the finite case or, more generally, the Noetherian case (since Noetherian relations are precisely those the transposes of which are progressively finite), there exists a p that satisfies the properties of (10), i.e., all calls of *point* in the program (P2) are defined.

The conjunction of the formulae $R \subseteq S$, $\mathsf{I} \subseteq S$, $S;S \subseteq S$, $S \cap S^\mathsf{T} \subseteq \mathsf{I}$, and $S \cup S^\mathsf{T} = \mathsf{L}$ forms the post-condition $\mathit{Post}(R,S)$ and the loop-invariant $\mathit{Inv}(R,S,v)$ consists of a conjunction of eight formulae, which are shown in Table 3.

The formula $\mathit{Inv}_0(v)$ specifies v as a vector and $\mathit{Inv}_2(S,v)$ to $\mathit{Inv}_6(S)$ constitute the relation-algebraic formalisation of the above described relationship between the sets of the chain (8) and the relations of the chain (9). The remaining two formulae $\mathit{Inv}_1(S,v)$ and $\mathit{Inv}_7(R,v)$ specify that the set, modelled by v, is predecessor-closed w.r.t. S and R, respectively.

As before, we use Prover9 to verify all proof obligations. This time there are no problems at all, and all verification tasks could be fully automated without interactions. The establishment of the loop-invariant as well as the verification of the post condition (proof obligations (PO1) and (PO3)) takes no time; the running times of the maintenance of the invariance are shown in Table 4. This finishes the proof of partial correctness.

We can again use Prover9 to verify total correctness. In fact the proofs are nearly identical to the ones for the program (P1).

3.4 Equivalence of Logical and Relation-algebraic Specifications

In the previous two sections we have automatically proven the total correctness of two relational while-programs. One reason why we could use automated theorem provers is that we are able to write program specifications and loop-invariants as relation-algebraic formulae. However, often specifications and program properties are *not* given in a relation-algebraic manner, but in predicate logic. For example, the post-condition of the program (P2), which characterises a topological sorting S of R, in first-order logic is the conjunction of the following formulae:

$$\begin{array}{l} \forall x,y : (x,y) \in R \Rightarrow (x,y) \in S \\ \forall x : (x,x) \in S \\ \forall x,y,z : (x,y) \in S \wedge (y,z) \in S \Rightarrow (x,z) \in S \\ \forall x,y : (x,y) \in S \wedge (y,x) \in S \Rightarrow x = y \\ \forall x,y : (x,y) \in S \vee (y,x) \in S \ . \end{array} \tag{11}$$

These formulae are standard predicate logic in combination with set theory. For example, the latter three characterise S as transitive, antisymmetric, and total (sometimes also called complete); hence as a linear order. The formulae of (11) are, in the same order, equivalent to $R \subseteq S$, $\mathsf{I} \subseteq S$, $S;S \subseteq S$, $S \cap S^\mathsf{T} \subseteq \mathsf{I}$, and $S \cup S^\mathsf{T} = \mathsf{L}$, respectively.

In this section we show that Prover9 can also be used to verify such equivalences. By this we close the gap between specifications written in predicate logic and specifications written in relation algebra, as we used them earlier. To do so, we have to define some fragments of set theory in Prover9. We define a new predicate `in(`x,y,R`)`, where R is a relation and x, y range over the universe of R. Semantically, we want to have that `in(x,y,R)` iff $(x,y) \in R$. Hence `in(`x,y,R`)` models the membership property. The predicate `in` needs additional axioms for

Table 4. Running Times for Proof Obligation (PO2)

Formula	Running Time
$v \neq \mathsf{L} \wedge (10) \wedge \mathit{Inv}(R,S,v) \Rightarrow \mathit{Inv}_0(v \cup p)$	0 s
$v \neq \mathsf{L} \wedge (10) \wedge \mathit{Inv}(R,S,v) \Rightarrow \mathit{Inv}_1(S \cup v;p^\mathsf{T}, v \cup p)$	22 s
$v \neq \mathsf{L} \wedge (10) \wedge \mathit{Inv}(R,S,v) \Rightarrow \mathit{Inv}_2(S \cup v;p^\mathsf{T}, v \cup p)$	3 s
$v \neq \mathsf{L} \wedge (10) \wedge \mathit{Inv}(R,S,v) \Rightarrow \mathit{Inv}_3(R, S \cup v;p^\mathsf{T}, v \cup p)$	1 s
$v \neq \mathsf{L} \wedge (10) \wedge \mathit{Inv}(R,S,v) \Rightarrow \mathit{Inv}_4(S \cup v;p^\mathsf{T})$	0 s
$v \neq \mathsf{L} \wedge (10) \wedge \mathit{Inv}(R,S,v) \Rightarrow \mathit{Inv}_5(S \cup v;p^\mathsf{T})$	43 s
$v \neq \mathsf{L} \wedge (10) \wedge \mathit{Inv}(R,S,v) \Rightarrow \mathit{Inv}_6(S \cup v;p^\mathsf{T})$	24 s
$v \neq \mathsf{L} \wedge (10) \wedge \mathit{Inv}(R,S,v) \Rightarrow \mathit{Inv}_7(R, v \cup p)$	0 s

the relation-algebraic operations $\cup$, $\cap$, $\overline{}$, $;$, $^\mathsf{T}$ and the constants L, O, and I; all being straight forward. For example, union and transposition are defined as

```
all R all S (in(x,y,(R \/ S)) <-> in(x,y,R) | in(x,y,S)).
all R       (in(x,y,R^)       <-> in(y,x,R)).
```

The universal relation L can be defined as `in(x,y,L)` and similar the other two constants O and I can be defined. This combines the algebraic and the logical point of view on relations. In the same manner we can specify inclusion and equality on relations:

```
all R all S (R <= S <-> (all x all y (in(x,y,R)  -> in(x,y,S)))).
all R all S (R == S <-> (all x all y (in(x,y,R) <-> in(x,y,S)))).
```

With an input file containing all facts about the predicate `in` – the full input file can be found again in the appendix – we have verified that the five logical formulae in (11) in fact are equivalent to their relation-algebraic counterparts. Unfortunately, our experiments show that Prover9 does not always find a proof or needs long running times. This is due to two reasons: (a) proving equivalences is often hard, not only for theorem provers, but also for human beings, and (b) Prover9 does not have further knowledge about the operators (as above). Hence, Prover9 needs to derive all facts needed, but it might also derive useless facts, such as "towers" of transpositions. By the latter we mean that Prover9 searches the search space and derives formulae such as

$$(x,y) \in R \;\Leftrightarrow\; (x,y) \in \; R^{\mathsf{T}^\mathsf{T}} \Leftrightarrow\; (x,y) \in \; R^{\mathsf{T}^{\mathsf{T}^{\mathsf{T}^\mathsf{T}}}} \;\Leftrightarrow\; \ldots$$

Splitting equivalences into two implications is an easy solution for problem (a). Moreover, this strategy can easily be automated by a preparation step while generating the input file for Prover9. Often this improves the running times of Prover9 drastically. For problem (b) there are two different approaches. The first one requires the addition of auxiliary lemmas, as we did in Section 3.2. However, for the proofs presented in this section there is a more generic way. When aiming at the proof for the formula

$$(\forall x, y : (x,y) \in R \Rightarrow (x,y) \in S) \;\Leftrightarrow\; R \subseteq S,$$

it is unlikely that Prover9 requires facts about the operations $\cup$, $\cap$, $\overline{}$, or $^\mathsf{T}$. Hence the corresponding axioms can be dropped. A quick check with Mace4 can show whether one of the skipped equivalences is needed – this is not the case for our experiments. Using the latter strategy, all but one of the above mentioned five equivalences can be proven in nearly no time. Only one equivalence needs to be split into implications. The results are summarised in Table 5.

4 Lessons Learned

In the present paper, we aimed at proof automation and proof assistance for the assertion-based verification of simple relational while-programs. Overall the experiments performed have been successful and our experience was often positive.

Table 5. Running Times for the Verification of the Formulae of (11)

Formula			Running Time
$(\forall x, y : (x, y) \in R \Rightarrow (x, y) \in S)$	$\Leftrightarrow$	$R \subseteq S$	0 s
$(\forall x : (x, x) \in S)$	$\Leftrightarrow$	$I \subseteq S$	0 s
$(\forall x, y, z : (x, y) \in S \wedge (y, z) \in S \Rightarrow (x, z) \in S)$	$\Leftrightarrow$	$S;S \subseteq S$	0 s
$(\forall x, y : (x, y) \in S \wedge (y, x) \in S \Rightarrow x = y)$	$\Rightarrow$	$S \cap S^{\mathsf{T}} \subseteq \mathsf{I}$	1 s
$(\forall x, y : (x, y) \in S \wedge (y, x) \in S \Rightarrow x = y)$	$\Leftarrow$	$S \cap S^{\mathsf{T}} \subseteq \mathsf{I}$	0 s
$(\forall x, y : (x, y) \in S \vee (y, x) \in S)$	$\Leftrightarrow$	$S \cup S^{\mathsf{T}} = \mathsf{L}$	2.5 s

However, there are some lessons to be learned when following this approach. The most important ones are discussed in this section.

All automated theorem proving systems depend on the axioms, given as input. If there are too few, many auxiliary facts need to be derived on the fly; if there are too many, the search space explodes and the system probably will not terminate. When starting our experiments, we used a minimal set of axioms only. We noticed that this set was far too small. So, we added a couple of further well-known laws, such as monotonicity and (sub-)distributivity laws – all these facts can be proven automatically by Prover9; see [16]. It turned out that we found a good set of axioms. With this extended set, our second example could be verified fully automatically and the first one only failed for one goal, which could be proven after we added the three laws of Table 1.

Although Prover9 helps a lot, it cannot be expected that a proof for every (true) fact can be found. It is well known that full automatisation is undecidable for relation algebra. Moreover, many researcher often spent years to find single proofs of difficult theorems – how could an automatic tool like Prover9 do it within a couple of minutes? However, theorem provers can help in verifying proofs of low or medium complexity. Those proofs often occur if induction on the structure of certain objects is used. As a consequence, a researcher can leave the easy theorems to the tool and can concentrate on "hard" tasks and the basic strategies for their proofs.

When experimenting, often hypotheses appear which are supposed to be true, but in fact are false. If, as in our case, counterexample generators (here Mace4) work on the same input files, they can be used to falsify hypotheses. We sometimes believed that a loop-invariant or another property is true, but in fact a certain formula was missing – this saves time of the researcher.

If a property is defined by a list of formulae, such as in our examples to be a point, then the definition of a corresponding predicate makes things much more readable. The same holds for the definition of auxiliary operations via certain properties. During our investigations we noticed that Prover9 unfolds such definitions rather late to keep the sets of formulae it has to treat small. Unfortunately, this strategy may lead to very large running times and a proof even may fail since certain rules cannot be applied. In such situations an unfolding of the definitions by the user (or by a preprocessing tool) led to success.

Since Prover9 does not support types, during all our experiments we have been responsible for the correct typing. We usually work within heterogeneous relation algebra in the sense of [22,23]. Therefore, typing was no problem and the rare typing errors immediately have been discovered and corrected with the help of Mace4 or RelView experiments.

5 Conclusion and Outlook

In this paper, we have shown that program verification can sometimes be achieved by the use of automated theorem provers. In particular, we have followed an approach that automatically verifies imperative programs for relation-based discrete structures by combining relation algebra, the assertion-based technique and the automated theorem prover Prover9. By this, we have been able to prove the correctness of a relational program for determining the reflexive-transitive closure and of a relational topological sorting program. We have also treated the automatic verification of the equivalence of the common logical and the relation-algebraic specifications of the properties used in our example (and elsewhere in a similar context).

So far we have only considered relations as data structures. These data structures can be used for algorithms working on many discrete structures, for instance those mentioned in the introduction. However, relation algebra is limited and cannot, for example, reason about words, regular expressions, paths in graphs, and weighted graphs. Reasoning on these structures is often done by calculations on (variants of) Kleene algebra. Since Kleene algebra is also suited for automated theorem provers (see e.g., [15]), we plan to extend our class of algorithms to Kleene-algebraic data structures.

Presently, we manually generate the loop-invariants. In doing so, the main formulae (e.g. $\mathit{Inv}_2(S, v)$ to $\mathit{Inv}_6(S)$ in case of topological sorting) constitute formalisations of the ideas behind the algorithms and are frequently obtained via suitable generalisations of the post-conditions. Based on them, the auxiliary formulae (the remaining ones in case of topological sorting) are usually discovered when trying to verify that the main formulae are maintained by the loop-bodies. For the latter, the tools Mace4 (for generating counterexamples) and RelView (for program evaluation, animation, and visualisation of relations) proved to be very useful. Under this point of view, our approach consists in the computer-supported application of the fundamental principle that "a program and its correctness proof should be developed hand-in-hand with the proof usually leading the way" (cf. [13], p. 164).

If program verification is done using the level of informality common to usual human-produced mathematical proofs, then the facts specified by the above mentioned auxiliary formulae may be overlooked and this may lead to subtle errors. We believe that our approach, as all computer-aided formal methods of programming, leads to results with a much greater mathematical certainty. Hence it increases the confidence.

Although the generation of loop-invariants is in general hard (or even infeasible), techniques for automatically testing and generating loop-invariants and intermediate assertions have been developed since the middle of the 1970s. They are tailored to specific applications and assume a specific structure of the programs and the used assertions. The applied techniques frequently stem from program analysis and computer algebra; see e.g., [21,19]. Apart from automatically testing loop-invariants via Mace4 and RelView, presently we do not use such ideas. However, the automated testing and generation of loop-invariants and intermediate assertions in case of relational programs is part of future work. We hope that algebraic expressions support such tasks, in particular in cases where algebra leads to nice properties and clear structures.

As we have mentioned in Section 4, we usually work within heterogeneous relation algebra where each relation has a distinct type. For reasons of efficiency, in such a setting a vector usually has a type $X \leftrightarrow \mathbf{1}$, with $\mathbf{1}$ as a specific singleton set, i.e., corresponds to a Boolean column vector. To get along with such situations and to benefit from the advantages of types, w.r.t. the preventation and detection of errors, the extension of our approach to heterogeneous relation algebra is planned for the future, too.

Acknowledgement. We thank the anonymous referees for their valuable comments that helped to improve the paper. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. R. Berghammer: Combining relational calculus and the Dijkstra-Gries method for deriving relational programs. Information Sciences 119, 155-171 (1999).
2. R. Berghammer, T. Hoffmann: Deriving relational programs for computing kernels by reconstructing a proof of Richardson's theorem. Science of Computer Programming 38, 1-25 (2000).
3. R. Berghammer, T. Hoffmann: Relational depth-first-search with applications. Information Sciences 139, 167-186 (2001).
4. R. Berghammer: Applying relation algebra and RelView to solve problems on orders and lattices. Acta Informatica 45, 211-236 (2008).
5. R. Berghammer, G. Struth: On automated program construction and verification. In: C. Bolduc, J. Desharnais, B. Ktari (eds.): Mathematics of Program Construction. LNCS 6120, Springer, 22-41 (2010).
6. R. Berghammer, S. Fischer: Simple rectangle-based functional programs for computing reflexive-transitive closures. In: W. Kahl, T.G. Griffin (eds.): Relational and Algebraic Methods in Computer Science. LNCS 7560, Springer, 114-129 (2012).
7. W. Bibel, P. Schmitt: Automated deduction: A basis for applications. Applied Logic Series, Kluwer (1998).
8. L.H. Chin, A. Tarski: Distributive and modular laws in the arithmetic of relation algebras. Univ. of California Publ. Math. (new series) 1, 341-384 (1951).
9. H.H. Dang, P. Höfner: First-order theorem prover evaluation w.r.t. relation- and Kleene algebra. In: R. Berghammer, B. Möller, G. Struth (eds.): Relations and Kleene Algebra in Computer Science – Ph.D. Programme at RelMiCS 10/AKA 05. Technical Report 2008-04, Institut für Informatik, Universität Augsburg, 48-52, (2008).

10. E.W. Dijkstra: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18, 453-457 (1975).
11. E.W. Dijkstra: A discipline of programming. Prentice-Hall (1976).
12. S. Foster, G. Struth, T. Weber: Automated engineering of relational and algebraic methods in Isabelle/HOL (invited Tutorial). In: H. de Swart (ed.): Relational and Algebraic Methods in Computer Science. LNCS 6663, Springer, 52-67 (2011).
13. D. Gries: The science of computer programming. Springer (1981).
14. C. Hattensperger, R. Berghammer, G. Schmidt: RALF – A relation-algebraic formula manipulation system and proof checker. In: M. Nivat C. Rattray, T. Rus, G. Scollo (eds.): Algebraic Methodology and Software Technology. Workshops in Computing, Springer, 407-408 (1993).
15. P. Höfner, G. Struth: Automated reasoning in Kleene Algebra. In: F. Pfenning (ed.): Automated Deduction – CADE21, LNAI 4603, Springer, 279-294 (2007).
16. P. Höfner, G. Struth: On automating the calculus of relations. In: A. Armando, P. Baumgartner, G. Dowek (eds.): Automated Reasoning. LNAI 5195, Springer, 50-66 (2008).
17. W. Kahl: Calculational relation-algebraic proofs in Isabelle/Isar. In: R. Berghammer, B. Möller, G. Struth (eds.): Relational and Kleene-Algebraic Methods in Computer Science. LNCS 3051, Springer, 179-190 (2004).
18. A.B. Kahn: Topological sorting of large networks. Communications of the ACM 5, 558-562 (1962).
19. L. Kovacs: Invariant generation for P-solvable loops with assignments. In: E.A. Hirsch, A.A. Razborov, A.L. Semenov, A. Slissenko (eds.): Computer Science – Theory and Practice.. LNCS 5010, Springer, 349-359 (2008).
20. W. MacCaull, E. Orłowska: Correspondence results for relational proof systems with application to the Lambek calculus. Studia Logica 71(3), 389-414 (2002).
21. M. Müller-Olm, H. Seidl: Computing polynomial program invariants. Information Processing Letters 91(5), 233-244 (2004).
22. G. Schmidt, T. Ströhlein: Relations and graphs, Discrete mathematics for computer scientists. EATCS Monographs on Theoretical Computer Science, Springer (1993).
23. G. Schmidt: Relational mathematics. Encyclopedia of Mathematics and its Applications, vol. 132, Cambridge University Press (2010).
24. J. Schumann: Automated theorem proving in software engineering. Springer (2001).
25. C. Sinz: System description: ARA – An automated theorem prover for relation algebras. In: D. McAllester (ed.): Automated Deduction – CADE 2000, LNAI 1831, Springer, 177-182 (2000).
26. A. Tarski: On the calculus of relations. Journal of Symbolic Logic 6(3), 73-89 (1941).
27. A. Tarski, S. Givant: A formalization of set theory without variables. AMS Colloquium Publications Vol. 41, (1987).
28. D. von Oheimb, T.F. Gritzner: RALL: Machine-supported proofs for relation algebra. In: W. McCune (ed.): Automated Deduction – CADE 1997, LNCS 1249, Springer, 380-394 (1997).
29. C. Weidenbach et al.: System description: SPASS version 3.0. In: F. Pfenning (ed.): Automated Deduction – CADE 2007, LNAI 4603, Springer, 514-520 (2007).
30. RelView homepage: `http://www.informatik.uni-kiel.de/~progsys/relview/` (accessed 30 April 2013).
31. W.W. McCune: Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9` (accessed 30 April 2013).

A Prover9 Templates

This appendix contains two templates to be used with Prover9. The first one specifies relation algebra.

```
% LANGUAGE SPECIFICATION
 op(500, infix,   "\/" ).        % union
 op(490, infix,   "/\" ).        % intersection
 op(700, infix,   "<=").         % inclusion
 op(480, postfix, "*" ).         % composition (not Kleene star)
 op(300, postfix, "’").          % complementation
 op(300, postfix, "^").          % transposition
% AXIOMS
 formulas(sos).
   % axioms of Boolean algebra %
     %commutativity
       x \/ y = y \/ x.
       x /\ y = y /\ x.
     %associativity
       x \/ (y \/ z) = (x \/ y) \/ z.
       x /\ (y /\ z) = (x /\ y) /\ z.
     %absorpotion
       x \/  (y /\ x) = x.
       x /\  (y \/ x) = x.
     % ordering
       x <= y  <->  x \/ y = y.
       x <= y  <->  x /\ y = x.
     %distributivity
       x /\ (y \/ z) = (x /\ y) \/ (x /\ z).
       x \/ (y /\ z) = (x \/ y) /\ (x \/ z).
     %constants
       L = x \/ x’.
       0 = x /\ x’.
   % composition %
     x * (y * z) = (x * y) * z.
     x * I = x.
     I * x = x.
   % Schroeder/Dedekind %
     x* y /\ z <= (x /\ z* y^) * (y /\ x^* z).
     x* y <= z <-> x^ * z’<= y’.
     x* y <= z <-> z’ * y^ <= x’.
   % standard axioms for finite iteration (Kleene star) %
     %unfold laws
       I \/ x * rtc(x) = rtc(x).
       I \/ rtc(x) * x = rtc(x).
     %induction
       x * y \/ z <= y  ->  rtc(x) * z <= y.
       y * x \/ z <= y  ->  z * rtc(x) <= y.
  end_of_list.
 % CONJECTURE
  formulas(goals).
    %lemma to be proved
  end_of_list.
```

Although Prover9 accepts capital letters as variable symbols, such as `Q`, `R`, and `S`, this template uses the small letters `x`, `y`, and `z` for variables. The reason for this renaming is that the latter variable names are automatically qualified by Prover9, i.e., they can be used without using the keyword `all`.

The second template establishes the relation between local and relation-algebraic specifications (see Section 3.4).

```
% LANGUAGE SPECIFICATION     %--as above--%
% AXIOMS
  formulas(sos).
    %  in()-predicate
      %operations
        all R all S (in(x,y,R \/ S) <-> (in(x,y,R) | in(x,y,S))).
        all R all S (in(x,y,R /\ S) <-> (in(x,y,R) & in(x,y,S))).
        all R all S (in(x,y,R * S)  <-> exists z (in(x,z,R) & in(z,y,S))).
        all R       (in(x,y,R')     <-> -(in(x,y,R))).
        all R       (in(x,y,R^)     <-> in(y,x,R)).
      %constants
        in(x,y,I) <-> x=y.
        -(in(x,y,O)).
        in(x,y,L).
      %inclusion and equality
        all R all S (R == S <-> (R <= S & S <= R)).
        all R all S (R <= S <-> (all x all y (in(x,y,R) -> in(x,y,S)))).
  end_of_list.
  % CONJECTURE     %--as above--%
```