On Automated Program Construction and Verification

Rudolf Berghammer[1] and Georg Struth[2]

[1] Institute of Computer Science, Christian-Albrechts-University of Kiel, Germany
rub@informatik.uni-kiel.de
[2] Department of Computer Science, University of Sheffield, UK
g.struth@dcs.shef.ac.uk

Abstract. A new approach for automating the construction and verification of imperative programs is presented. Based on the standard methods of Floyd, Dijkstra, Gries and Hoare, it supports proof and refutation games with automated theorem provers, model search tools and computer algebra systems combined with "hidden" domain-specific algebraic theories that have been designed and optimised for automation. The feasibility of this approach is demonstrated through fully automated correctness proofs of some classical algorithms: Warshall's transitive closure algorithm, reachability algorithms for digraphs, and Szpilrajn's algorithm for linear extensions of partial orders. Sophisticated mathematical methods that have been developed over decades could thus be integrated into push-button engineering technology.

1 Introduction

Programs without bugs is one of the great ideals of computing. It motivated decades of research on program construction and verification. A commonality of most approaches to program correctness is the combination of mathematical models of programs with mechanised program analysis tools. It requires integrating the science of programming into the engineering of programs.

Provably correct programs can be obtained in different ways: *Program construction* or *synthesis* means deriving executable programs from mathematical models or specifications, so that a program is correct if its derivation is sound. *Program verification* means proving that a given program satisfies a set of assertions provided by the programmer, usually by demonstating that whenever it initially meets certain constraints (a precondition) it will satisfy a certain property (a postcondition) upon termination. In the presence of loops, invariants must be maintained during their execution.

Models for programs and their properties have been based on different mathematical formalisms. Relational calculi are among the most ubiquitous ones. They form the basis of established methods like Alloy [16], *B* [1] or *Z* [22]. Tool support has usually been integrated through interactive theorem provers or finitist methods such as model checkers or SAT-solvers. To be useful in practice, modelling languages must be simple but expressive, and tool support should be as invisible and automatic as possible. These requirements are incompatible and need to be balanced to yield methods that are lightweight yet powerful.

Our main contribution is an approach to the construction and verification of imperative programs which aims at this balance in a novel way through computer enhanced mathematics. Its backbone is a combination of off-the-shelf automated theorem proving systems (ATP systems), model generators and computer algebra systems with domain-specific algebras that are designed and optimised for automation. This combination allows automatic program correctness proofs, but it also supports program development at a more fundamental level through the inference of specification statements and algorithmic properties in a game of proof and refutation. While algebraic theories and automation technology can largely be hidden behind an interface, developers can focus on the conceptual level and use simple intuitive relational languages for modelling and reasoning.

A second contribution is the demonstration of the potential and feasibility of this approach through partial correctness proofs of some classical algorithms:

- Warshall's algorithm for the transitive closure of a relation,
- two reachability algorithms for directed graphs,
- Szpilrajn's algorithm for linear extensions of partial orders.

While Warshall's and Szpilrajn's algorithm are constructed from specifications, the reachability algorithms are verified. In our current scenario, assertions must be provided by the developer for verification. Similarly, for program construction, we do not automatically synthesise invariants or abduce preconditions, but we provide automated tool support for developing and formally justifying these tasks. In particular, in both cases, all proof obligations are discharged fully automatically. We use different domain-specific algebras at the dark side of the interface, and also discover a new refinement law for reflexive transitive closures in our case studies. All ATP proofs in this paper can be found at a web site [13].

We build on decades of work on formal methods, logics and algebras for programs, ATP technology, and program construction. At the engineering side, our use of relational calculi is inspired by formal methods like Alloy, B or Z; our approach to program correctness is essentially that of Dijkstra and Gries [11] and closely related to Hoare logic [12]. We use the ATP system Prover9 and the model generator Mace4 [19] in our proof experiments, but most other state-of-the-art ATP systems and model generators would yield a similar performance. For testing and visualising relational programs we use the RelView computer algebra system [5]. At the mathematical side, we use Tarski's relation algebras [24] and various reducts, such as idempotent semirings, Kleene algebras or domain semirings [8,17]. Our case studies build on previous manual calculational correctness proofs in relation algebra [3,6], but most of our new proofs are different, sometimes simpler, and valid in a larger class of models.

The power and main novelty of our approach lies in the balance of all these theories, methods and tools. However, presently, we have not much to show beyond the proof of concept. Further work will be needed to transform our ideas into program construction and verification tools that will be useful for teaching and software development practice.

2 A Simple Relational Modelling Language

For constructing and verifying programs we do not assume much more than an intuitive conceptual and operational understanding of binary relations and sets with their fundamental operations and properties, as needed for basic modelling with Alloy or RelView. To obtain a uniform readable syntax, and to make it easy to replay our automation experiments, we use a notation that can be processed by Prover9 and Mace4 although it deviates from most textbooks.

As usual, a *binary relation* x on a set A is a subset of the Cartesian product $A \times A$, a set of ordered pairs on A. Our approach extends to heterogeneous relations of type $A \times B$, but this would only overload the presentation. As sets, relations form Boolean algebras and inherit the Boolean operations $+$ of union or join, $*$ of intersection or meet and $'$ of complementation; there is an empty relation 0 and a universal relation $U = A \times A$. The *relative product* $x;y$ of two binary relations x and y is formed by the ordered pairs (a, b) with $(a, c) \in x$ and $(c, b) \in y$ for some $c \in A$. The *converse* $x^\wedge$ of a binary relation x is formed by the ordered pairs (b, a) with $(a, b) \in x$. The *identity relation* 1 consists of all ordered pairs (a, a) with $a \in A$.

Apart from these basics we assume that readers are familiar with the algorithmics of finite binary relations as presented in undergraduate textbooks [7]. Briefly, finite binary relations can be represented as directed graphs (digraphs) — ordered pairs corresponding to vertices linked by arrows — and implemented via the algebra of Boolean adjacency matrices. The relational operations are reflected in the matrix algebra: join by matrix sum, relative product by matrix product, conversion by matrix transposition; the identity relation by the diagonal matrix, the empty relation by the zero matrix, and the universal relation by the matrix in which each element is 1. Sets can either be implemented as vectors, as row-constant matrices or as subidentity matrices with ones at most along the diagonal. The multiplication of a matrix with a vector corresponds to computing the preimage (a set) of a set with respect to a relation. The matrix representation provides an important intuition for our case studies; all our programs can be implemented directly as matrix algorithms in the RelView tool.

Only a few additional concepts are needed for our case studies. The *reflexive transitive closure* and the *transitive closure* of a relation x are

$$\mathsf{rtc}(x) = \sum_{i \geq 0} x^i \qquad \text{and} \qquad \mathsf{tc}(x) = \sum_{i \geq 1} x^i,$$

with powers x^i defined inductively. The *domain* $d(x)$ and *range* $r(x)$ of x are defined as the sets

$$d(x) = \{a \in A : \exists b \in A.(a, b) \in x\},$$
$$r(x) = \{b \in A : \exists a \in A.(a, b) \in x\}.$$

Throughout this paper, we will call singleton sets *points* and single ordered pairs *atoms*. The technicalities of reasoning with relations are delegated as far as possible to tools. They are hidden from developers behind an interface.

3 The Dark Side of the Interface

For automated program construction and verification, the relational concepts discussed in the previous section must be implemented in domain-specific theories that are suitable and optimised for automated proof search. Developers are not supposed to see these theories in detail, they are hidden behind an interface. We use variants and reducts of relation algebras (in the sense of Tarski [18,21,24]) as domain-specific theories in our case studies.

A *relation algebra* is a structure $(R, +, *, ', 0, U, ;, 1, ^\wedge)$ that satisfies the following axioms taken from Maddux's textbook [18].

```
x+y=y+x  &  x+(y+z)=(x+y)+z  &  x=(x'+y')'+(x'+y)'.
x;(y;z)=(x;y);z  &  x;1=x  &  (x+y);z=(x;z)+(y;z).
(x^)^=x  &  (x+y)^=x^+y^  &  (x;y)^=y^;x^  &  x^;(x;y)'+y'=y'.
```

These axioms are effectively executable by Prover9 and Mace4 [15]; complete input files can be found in a proof data base [13]. The first line contains Huntington's axioms for Boolean algebras, the second line those for relative products, and the third line those for conversion. The following standard definitions are always included in our input files.

```
x*y=(x'+y')'  &  x<=y <-> x+y=y  &  0=x*x'  &  U=x+x'.
% x!=0 -> U;(x;U)=U.
```

The first equation defines meet via De Morgan's law. The second formula defines the standard order of the algebra. The next two equations define the least and the greatest element of the algebra. Tarski's axiom in the second line is needed for proving one single auxiliary lemma in this paper. Apart from this, it has not been used and is therefore commented out.

Following Ng [20], we axiomatise the reflexive transitive closure $\mathsf{rtc}(x)$ of a binary relation x as a least fixed point:

```
1+x;rtc(x)=rtc(x)  &  z+x;y<=y -> rtc(x);z<=y.
1+rtc(x);x=rtc(x)  &  z+y;x<=y -> z;rtc(x)<=y.
```

The transitive closure $\mathsf{tc}(x)$ of x is defined as `tc(x)=x;rtc(x)`. The expressions `rtc(x)` and `tc(x)` can now be used at the developer's side of the interface for modelling and reasoning about programs, whereas the implementation of these concepts in relation algebra at the dark side of the interface is used for automated reasoning with Prover9 and Mace4, but need not concern the developer.

It is standard to model sets in relation algebras either as *vectors* or as *subidentities* (elements below 1). We present the first alternative for relation algebras and the second one for Kleene algebras below. Both allow us to implement points and atoms. In fact, we only need *weak points*, which are points or zero, and *weak atoms*, which are atoms or zero.

```
inj(x) <-> x;x^<=1.                        % def injection
vec(x) <-> x=x;U.                          % def vector
wpoint(x) <-> vec(x) & inj(x).             % def weak point
watom(x) <-> wpoint(x;U) & wpoint(x^;U).   % def weak atom
```

At the developer's side of the interface, `set(x) <-> vec(x)` can be used for typing sets, and the predicates `wpoint` and `watom` type (weak) points and (weak) atoms. The right-hand sides of these definitions implement these concepts. They can again be hidden. Implementation details are not needed to understand our case studies; a discussion can be found in the literature [21].

Finally, the domain of a relation x is implemented as `d(x)=1*x;U`, and the range `r(x)=d(x^)` as the domain of the converse of x.

The calculus of relations can effectively be automated [15]. But experiments show that ATP systems still have difficulties with proving correctness of complex programs from the axioms of relation algebras and auxiliary concepts alone. Domain-specific and problem-specific theories and assumption sets must be engineered for applications. Adding assumptions requires libraries of verified relational properties [13]. Enhancing proof search requires reducts of relation algebras. Variants of idempotent semirings and Kleene algebras are known to be very suitable in this respect [8,14]. All verified facts about binary relations can, of course, safely be used as independent assumptions with these reducts.

An *idempotent semiring* is a structure $(S, +, ;, 0, 1)$ that satisfies the axioms

```
x+y=y+x  &  x+(y+z)=(x+y)+z  &  x+0=x  &  x+x=x.
x;(y;z)=(x;y);z  &  x;1=x  &  1;x=x  &  x;0=0  &  0;x=0.
x;(y+z)=x;y+x;z  &  (x+y);z=x;z+y;z.
x<=y <-> x+y=y.
```

An idempotent semiring expanded by the reflexive transitive closure operation axiomatised above is a *Kleene algebra*. Conversion can now be axiomatised as

```
(x^)^=x  &  (x+y)^=x^+y^  &  (x;y)^=y^;x^  &  x<=x;(x^;x).
```

The universal relation can be axiomatised as `x<=U`.

Sets can again be modelled as vectors or as subidentities, but, in contrast to relation algebras, the subalgebras of all subidentities in idempotent semirings or Kleene algebras are not necessarily Boolean algebras.

The simplest approach — and best suited for ATP — uses *domain semirings* and *Kleene algebras with domain* [8]. Domain is axiomatised via an *antidomain* operation:

```
a(x);x=0  &  a(x;a(a(y)))=a(x;y)  &  a(a(x))+a(x)=1.
```

Intuitively, the antidomain $a(x)$ of a relation x is the set of all elements which are not in the domain $d(x)$ of x, hence `d(x)=a(a(x))`. Axiomatic details are again not important, but the following two properties are essential for understanding our implementation of sets across the interface: First, the set $d(S)$ of all domain elements of S is precisely the set of all $x \in S$ that satisfy $x = d(x)$. Second $d(S)$ forms a Boolean algebra among the subidentities of S and the meet operation is multiplication. In this setting, we can therefore type `set(x) <-> d(x)=x`.

The notion of range is dual to that of domain. Its axiomatisation only requires swapping the order of multiplication.

In the context of domain semirings, points can be implemented via *rectangular* relations. The corresponding axioms are

```
rctangle(x) <-> x;(U;x)=x.
wpoint(x) <-> set(x) & rctangle(x).
```

Intuitively, a relation is rectangular if it is equal to the cartesian product of its domain and range, and this explains why points are rectangular sets[1]. In some sense, rectangles can be understood as generalised points.

4 Automation Technology Review

In this section we briefly sketch the Dijkstra-Gries approach to program development and discuss the three tools used for its automation behind the interface.

The Dijkstra-Gries Approach. As mentioned in the introduction, program construction means that a program is derived from a specification, and program verification means that a given program is proved to be correct with respect to a given specification. For imperative programs, specifications usually consist of preconditions, postconditions and invariants which model the inductive properties implemented in the body of a loop. The correctness of a simple while loop is implied by the following proof obligations:

1. The invariant is established by the initialisation (before the loop starts)
2. Each execution of the loop's body preserves the invariant, as long as the guard of the loop is true.
3. The invariant establishes the postcondition if the guard of the loop is false.
4. The loop terminates.

We will automatically analyse the first three proof obligations in our case studies, hence prove *partial program correctness*. We do not formally consider termination (*total correctness*) since this requires a different kind of analysis.

For program verification, the precondition, the postcondition and the invariant are added as assertions to the given program code, and the proof obligations can then be generated and analysed automatically. This approach is based on the seminal work of Floyd and Hoare [10,12].

For program construction, the invariant is hypothesised as a modification of the postcondition and the proof obligations are then established step by step together with the synthesis of the program: the guard of its loop, the initialisation of the variables to establish the invariant and the assignments in the loop's body to maintain it. Based on the fundamental principle that "a program and its correctness proof should be developed hand-in-hand with the proof usually leading the way" ([11], p. 164), the approach has been pioneered by Dijkstra [9], and elaborated by Gries [11] and others into a variety of techniques.

[1] The usual definition of rectangles in relation algebras uses only $x;U;x \le x$, and we could automatically verify that this is equivalent to $x;U;x = x$. In Kleene algebra, in contrast, the equational definition is strictly stronger than the original one; they are separated by a three element counterexample found by Mace4.

Tools for Proofs and Refutations. To automate all synthesis and verification proofs, we use the ATP system Prover9 [19]. The main reason is that its input syntax is very readable and that the model generator Mace4 [19] is based on the same syntax. This makes it particularly easy to replay our proof experiments. The interplay of ATP systems and model generators is essential for our games of proof and refutation in program construction.

Prover9 is complete for first-order logic with equality; it allows reasoning with relation algebra and all reducts we consider. It takes a set of assumptions and a proof goal and uses sophisticated proof search and redundancy elimination strategies combined with complex heuristics to deduce the goal from the assumptions. We use the tool entirely as a black box and as push-button technology. Prover9 is only a semi-decision procedure. In theory it can prove all first-oder theorems, but may run forever on non-valid proof goals. In practice, however, it often runs out of steam without proving a theorem. On success, Prover9 outputs a proof (which is usually not revealing for humans). Mace4 searches for finite models. It accepts essentially the same input as Prover9, and tries to construct a model of the assumptions that fails the proof goal, that is, a counterexample.

Prover9 and Mace4 support infix and postfix notation for algebraic operators and precedence declarations. For relation algebra, we use the following code:

```
op(500, infix,   "+").  % join
op(480, infix,   "*").  % meet
op(300, postfix, "’").  % complementation
op(450, infix,   ";").  % composition
op(300, postfix, "^").  % conversion
```

Operations with a lower number bind more strongly than those with a higher number. For Kleene algebra, we use the following declaration:

```
op(500, infix,   "+").
op(490, infix,   ";").
```

Assumptions and goals must be put into the following environment:

```
formulas(assumptions).     ...     end_of_list.
formulas(goal).            ...     end_of_list.
```

Examples for input and output files, and the complete set of proofs for our case studies, can be found in a proof database [13]. We have so far verified more than 500 theorems of relation algebras and Kleene algebras, including more or less all “textbook theorems”.

In our experiments, to demonstrate the robustness of the approach, we always use the weakest possible assumption set, ideally the theory axioms and basic definitions alone, at the expense of long running times. Adding the right lemmas would usually bring proof search down to a few seconds.

Besides Prover9 and Mace4 we also use the RelView system [5]. This is an interactive and graphics-oriented special purpose computer algebra system for the visualisation and manipulation of binary relations, and for relational prototyping and programming. RelView is optimised for very large objects, for instance, membership relations, which is especially important for prototyping. It uses an efficient internal implementation of relations via reduced ordered binary decision diagrams [4]. RelView provides predefined operations and tests for modelling and analysing relations in the lightweight style sketched in Section 2. (We do not use the RelView modelling language since it clashes with Prover9 syntax.) Relational functions and programs can be built from these operations. Relational functions are introduced as usual in mathematics, and relational programs are essentially while-programs based on relational datatypes.

Within our approach, the main applications of RelView are specification testing and support for reasoning with concrete finite binary relations. Specification testing includes mainly the evaluation of relational specifications and the comparison of the results obtained with original specifications in another formalism, or even the intuitive notion of the problem, in order to find weaknesses or inconsistencies. Support for relational reasoning is very important in program construction for finding and analysing loop invariants, that is, whether a candidate for an invariant satisfies or violates a proof obligation in some special cases. RelView allows developers to experiment with programs and assertions and to visualise this information via graphical representations of relations. Particularly useful for program construction and verification is the fact that the tool allows testing validity of arbitrary Boolean combinations of relational inclusions via an `ASSERT` command and that the relations needed can be randomly generated.

5 Synthesis of Warshall's Algorithm

As a first case study, we construct Warshall's classical algorithm [25] for computing the transitive closure of a finite binary relation — a digraph — from its specification. We carefully separate the developer's view from the domain-specific theory — in this case Kleene algebra with domain — and the automation technology at the dark side of the interface. We also aim at illustrating how a proof and refutation game with Prover9, Mace4 and RelView is essential in this construction.

We use the Dijkstra-Gries framework to derive a simple while-program from a given precondition and postcondition, inferring the loop invariant, the guard of the loop and the variable assignments along the way. This development is inspired by a previous manual correctness proof in relation algebra [3].

Initial Specification. Consider the following program construction task:

> *Given a finite binary relation x, find a program with a relational variable y that stores the transitive closure of x after its execution.*

We aim at a while-program of the following schematic form:

```
... y:=x ...
while ... do
  ... y:=? ... od
```

The precondition and postcondition are evident from the above specification:

```
pre(x) <-> x=x.
post(x,y) <-> y=tc(x).
```

The formula $x = x$ expresses for Prover9 that the precondition is always true. This is the case because the input relation x can be arbitrary. The proof obligations from Section 4 guide us through the synthesis of the initialisation, the guard and the body of the loop.

Developing the Invariant. The most important ingredient of our construction is still missing: Warshall's insight that makes the algorithm work.

> *Initially, compute only those paths contributing to the transitive closure of x that traverse no inner vertices. Then iteratively add inner vertices and compute the local transitive closure restricted to each new inner-vertex set incrementally from that of its predecessor set. Terminate when all possible new inner vertices have been added.*

How the incremental computation can be achieved will concern us later, but the invariant of the algorithm should by now be evident:

> *The variable y must maintain the transitive closure of x restricted to each set v of inner vertices that is constructed along the way.*

Formally, for Prover9, we obtain:

```
inv(x,y,v) <-> (set(v) -> y=rtc(x;v);x).
```

Note that $(x;v)^k;x$ yields the first and last points of x-paths with k inner vertices from the set v, and that sets are implemented as subidentities via domain.

Initialisation, and Guard. According to the specification, the set variable v should be initialised as `v:=0` and the loop should terminate when $v = 1$, or, even better, when $v = d(x)$, that is, when all vertices from which x is enabled have been visited. The guard of the loop should therefore be `v!= d(x)` (or `v!= 1`). We can justify these assumptions by verifying the following proof obligations.

Theorem 1. *The invariant is established by the initialisation (if the precondition holds); it establishes the postcondition when the guard of the loop is false.*

Proof. Using Kleene algebra with domain (behind the interface) we proved

```
pre(x) -> inv(x,x,0).
inv(x,y,v) & v=1 -> post(x,y).
inv(x,y,v) & v=d(x) -> post(x,y).
```

Since Prover9 can only handle one non-equational proof goal at a time, two goals always need to be commented out. The assumption file contains the axioms for Kleene algebras with domain, and the definition of set, transitive closure, precondition, postcondition and invariant, as listed above. The proofs of the first and the third goal were instantaneous and very short. The proof of the second goal was slightly harder. It required about 18s and has 129 steps. □

Termination and Development of the Loop. We now consider termination of the algorithm, and synthesise the body of the loop by considering the proof obligation that the invariant be preserved when executing the loop. This means synthesising assignments to the set variable v and the relational variable y:

- The assignment to v is obvious from the above discussion. We add a single new point p to the set v; `v:=v+p`.
- The assignment of y should, if possible, increment y, which stores the transitive closure of x restricted to v, by the transitive closure computed incrementally with respect to y and p. So we postulate `y:=y+f(y,p)`.

Our program should then have the following form:

```
y,v:=x,0
while v!=d(x) do
  p:=point(v')
  y,v:=y+f(y,p),v+p od
```

Here, `point` is a choice function that returns some point from the complement set of v, which is taken at the set level. It satisfies the axiom

```
wpoint(point(x)) & point(x)!=0.
```

but this is not needed in our development apart from termination. At the dark side of the interface, v' is translated to $a(v)$. Now, the assignment `v:=v+p` with v and p disjoint enforces termination of the loop. This is the only part of the proof which we do not automate. It remains to determine `y+f(y,p)`.

We compare the value y before and after the assignment to v. Before the assignment, $y = \mathsf{rtc}(x; v); x$. After the assignment `v:=v+p` we have the value

$$y = \mathsf{rtc}(x; (v + p)); x = \mathsf{rtc}(x; v + x; p).$$

So we could try to refine the reflexive transitive closure of this sum into a sum of reflexive transitive closures.

Consider $\mathsf{rtc}(a + b)$ for arbitrary relations a and b. First, the most straightforward refinement into $\mathsf{rtc}(a) + \mathsf{rtc}(b)$ can be refuted by a six-element counterexample. Hence we need to consider this expression in more detail.

Obviously, $\mathsf{rtc}(a + b)$ represents arbitrary alternating sequences of a and b, hence should be equal to $\mathsf{rtc}(\mathsf{rtc}(a); \mathsf{rtc}(b))$. This has already been verified by ATP [13]. Such sequences could form either one single a-block (possibly empty), or alternating blocks of a and b. But $b = x; p$ is not an arbitrary relation. In the matrix model, the point p projects x onto a matrix in which only one single non-zero row. This can be visualised in experiments with RelView. In other words, b is a rectangle and we conjecture the following more general fact.

Lemma 2. *Let x, y be elements of a relation algebra in which Tarski's axiom holds. If y is rectangular, then $x; y$ is rectangular.*

Proof. Using the axioms of relation algebras, Tarski's axiom and the above definition of rectangles, we proved `rctangle(y) -> rctangle(x;y)` by ATP. □

Interestingly, Lemma 2 does not hold in all Kleene algebras (Mace4 found a three-element counterexample) or for relation algebras without Tarski's axiom. But we can safely add it as an independent assumption.

The fact that $b = x;p$ is a rectangle has some impact on the alternating blocks of a and b. Intuitively, rectangles can be seen as generalised points. Many of their properties can be conjectured by thinking about points in the first place and then proved or refuted by Prover9 and Mace4. First, $b;b = b * b = b$, hence all b-blocks must have length one. Second, $b;\mathsf{rtc}(a);b \le b$, since in the left-hand side of this inequality, all inputs and outputs are projected onto the rectangle b. Hence, if b is rectangular, the developer might conjecture that $\mathsf{rtc}(a+b)$ can be refined to $\mathsf{rtc}(a) + \mathsf{rtc}(a);b;\mathsf{rtc}(a)$. And indeed we can prove the following new refinement law for reflexive transitive closures that is of general interest.

Proposition 3. *Let x, y be elements of a Kleene algebra with greatest element, and let y be rectangular. Then*

$$\mathsf{rtc}(x+y) = \mathsf{rtc}(x) + \mathsf{rtc}(x);y;\mathsf{rtc}(x).$$

Proof. By ATP, using the Kleene algebra axioms and the definitions of universal relation and rectangle. The $\geq$-proof took less than 10s. The $\leq$-proof took about 30s; it has 56 steps. □

Proposition 3 allows us to refine the term $(x;v + x;p)$ in the assignment of y since $x;p$ is rectangular by Lemma 2. We can therefore incrementally compute the transitive closure of x restricted to inner vertices in $v+p$ from that restricted to inner vertices in v by updating the relational variable y to $y + y;p;y$.

Lemma 4. *Let x, v, p be elements of a relation algebra in which Tarski's axiom holds, let p be rectangular, and let $y = \mathsf{rtc}(x;v);x$. Then*

$$\mathsf{rtc}(x;(v+p));x = y + y;p;y.$$

Proof. Using the idempotent semiring axioms, the refinement law from Proposition 3, and the equation from Lemma 2, we could automatically prove

```
rctangle(z) -> rtc(x;(v+z));x=rtc(x;v);x+(rtc(x;v);((x;z);rtc(x;v)));x.
```

in less than 15s. Note that Prover9 would not accept p as an implicitly universally quantified variable. Setting $y = \mathsf{rtc}(x;v);x$ then yields the result. □

We have thus formally justified the assignment `y:=y+y;p;y`, which is another key insight in Warshall's algorithm. We derived it from a general refinement law for reflexive transitive closures. Based on this formal development we can now prove explicitly and in declarative style the remaining proof obligation.

Theorem 5. *Executing the loop preserves the invariant if the guard of the loop is true.*

Proof. We attempted to prove the formula

```
wpoint(w) & inv(x,y,v) & y!=d(x) -> inv(x,y+y;(w;y),v+w).
```

from the axioms of Kleene algebras with domain, the definition of weak point, and the decomposition law from the proof of Lemma 4. However, Mace4 immediately found a three-element counterexample, so we needed to strengthen the assumptions. Adding the independent assumption `x;U=d(x);U`, which we automatically verified in relation algebras, yielded a short proof in less than 15s. □

A proof of Theorem 5 from the axioms of Kleene algebra and $x;U = d(x);U$ within reasonable time bounds did not succeed. Alternatively, we have found a less elegant and generic proof of Theorem 5 based on a decomposition of points instead of rectangles.

Partial Correctness. The result of the construction is summed up in the main theorem of this section.

Theorem 6. *The following variant of Warshall's transitive closure algorithm is (partially) correct:*

```
y,v:=x,0
while v!=d(x) do
  p:=point(v')
  y,v:=y+y;p;y,v+p od
```

The proof of this theorem has been fully automated in every detail, and the algorithm has been shown to be correct by construction. Kleene algebra alone did not suffice for this analysis. We used two additional independent assumptions that hold in relation algebras. One even required Tarski's axiom. Mace4 is instrumental for indicating when such assumptions are needed. Finding the right assumptions of course requires some background knowledge, but could be automated. A tool could blindly try combinations of previously verified relational lemmas from a given library.

To appreciate the complexity of proof search involved, it should be noted that most of the proofs involving transitive closures in this section are essentially inductive. With our algebraic axiomatisation of (reflexive) transitive closures of binary relations, these inductions could be captured calculationally in a first-order setting and therefore be automated.

In conclusion, the construction of Warshall's algorithm required some general familiarity with relations and some operational understanding of reflexive transitive closures. For the operational understanding, testing the loop invariant and developing the refinement law for reflexive transitive closures, experimenting with Mace4 and RelView was very helpful. But for the synthesis of the algorithm, no particular knowledge of the calculus of relation algebras or Kleene algebras was needed. All that could be hidden in the darkness of the interface.

6 Verification of Reachability Algorithms

The task of determining the set of states that are reachable from some given set of states in a digraph can also be reduced to relational reasoning. As a second case study, we show how two matrix-based algorithms can be automatically *verified.* In this scenario, the programmer annotates code with assertions for the precondition, postcondition and loop invariant. The code and the assertions are then tranformed into proof obligations from which program correctness is proved automatically in one full sweep behind the interface.

The RelView system provides a relational modelling language and an imperative programming language in which the programs and all assertions can be implemented and executed. We assume that RelView, Prover9 and Mace4 have been integrated into an imaginary tool that does all the translations between programs and tools, and runs the tools in the background.

A Naive Algorithm. The following relational algorithm computes the set w of states that are reachable in some digraph y from a set v of initial states; a previous manual construction can be found in [3]:

```
{pre(y,v) <-> x=x}
w:=v
while -(y^;w<=w) do
  {inv(y,v,w) <-> v<=w & w<=rtc(y^);v}
  w:=w+y^;w od
{post(y,v,w) <-> w=rtc(y^);v}
```

In this program, y is an adjacency matrix; v and w are vectors or other implementations of sets. But our imaginary verification tool ignores all types and selects Kleene algebra as the domain-specific theory after a syntactic analysis. It also ignores the operation of converse in $y^\wedge$ because y itself does not occur in the code. It therefore uniformly replaces $y^\wedge$ by x. Note that this step has little impact on ATP performance. The tool then passes the following postcondition, guard of the loop and invariant to Prover9; it ignores the trivial precondition:

```
guard(x,v,w) <-> -(x;w<=w).
post(x,v,w) <-> w=rtc(x);v.
inv(x,v,w) <-> v<=w & w<=rtc(x);v.
```

The postcondition says that upon termination the vector w stores all those vertices that are linked by an arrow in the reflexive transitive closure of x to a vertex in v. The idea of the program, to compute intermediate states w iteratively with respect to x such that after each iteration w is a superset of v and a subset of the set of reachable states, is captured by the invariant.

Proving partial correctness means discharging the following proof obligations, which our imaginary verification tool could automatically extract from the code and the assertions.

```
inv(x,v,w) & -guard(x,v,w) -> post(x,v,w).
inv(x,v,v).
inv(x,v,w) & guard(x,v,w) -> inv(x,v,w+x;w).
```

Using the axioms of Kleene algebra, Prover9 could instantaneously verify the first and the second proof obligation; the third one needed about ten seconds. In fact, the third proof obligation did not require the assumption that the guard of the loop is true. Termination has again been neglected. But obviously, the set w is enlarged in each iteration of the loop and finitely bounded by the guard.

A Refined Algorithm. The main drawback of the naive algorithm is that the guard of the loop is recomputed in each turn of the loop. Finite differencing yields a refined algorithm which uses a new vector or set variable u to store the intermediate values of $x; w \cdot w'$, where the complement w' of w is again taken at the set level (For a manual development, see again [3]). In this example, we do not use the precondition, postcondition and invariant in declarative style, but encode assertions directly in the relational modelling language.

```
{pre: true}
w,u:= v,v'*y^;v
while u!=0 do
  {inv: v<=w & w<=rtc(y^);v & u=w'*y^;w}
  w:=w+u
  u:=w'*y^;u od
{post: w=rtc(y^);v}
```

We now assume that our imaginary tool translates these assertions into a Kleene algebra with range (with dual domain axioms). The range operation is used behind the scene for typing sets and for computing sets of successor states: The set of states that are reachable (in one step) from a set v with respect to a relation x is given by the range of v;x. Now $a(x)$ denotes the antirange of x and $r(x)$ the range of x. If x is a set, then $x = r(x)$, $r(x) = a(a(x))$, and $a(r(x)) = a(x)$. Thus the verification tool can use $r(x)$ for "typing" that x is a set and generate the following formulas:

```
r(u)!=0.                                                % guard
r(w)=r(r(v);rtc(x)).                                    % postcond.
r(v)<=r(w) & r(w)<=r(r(v);rtc(x)) & r(u)=a(w);r(r(w);x). % invariant
```

Again, $y^\wedge$ is uniformly replaced by x, the trivial precondition is omitted. The tool would then generate the following proof obligations:

(1) The invariant is established by the initialisation.

```
r(v)<=r(v) & r(v)<=r(r(v);rtc(x)) & a(v);r(r(v);x)=a(v);r(r(v);x).
```

(2) The invariant establishes the postcondition if the guard of the loop is false.

```
r(v)<=r(w) & r(w)<=r(r(v);rtc(x)) & a(w);r(r(w);x)=0
                    -> r(r(v);rtc(x))<=r(w) & r(w)<=r(r(v);rtc(x)).
```

(3) Executing the loop preserves the invariant if the guard of the loop is true.

```
r(v)<=r(w) & r(w)<=r(r(v);rtc(x)) & r(u)=a(w);r(r(w);x)
                    -> r(v)<=r(w)+r(u) & r(w)+r(u)<=r(r(v);rtc(x)).
```

A proof of (1) took about 35s from the axioms of Kleene algebras with range and the definition of invariant. It yielded a proof with 42 steps. For the proof of (2), the postconditions has been split into two inequalities, and the (negated) guard of the loop has been ignored. A proof from the axioms alone failed within reasonable time; the following additional assumptions were required:

```
r(x+y)=r(x)+r(y)  &  r(v)+r(r(w);x)<=r(w) -> r(r(v);rtc(x))<=r(w).
```

Both formulas have already been verified [8]. The first assumption is additivity of range, which seems fundamental enough to be added to any domain-specific theory that uses range. The second additional assumption is based on the observation that the proof of the formula $r(w) \leq r(r(v); \mathsf{rtc}(x))$ from the third assumption, $r(w) \leq r(r(v); x)$, is essentially induction with respect to x. With these additional assumptions, Prover9 took about 150s; the proof has 102 steps. Proofs with fewer axioms failed within reasonable time. Whether a tool could automatically learn such additional assumptions is an interesting research question. The proof of (3) used the axioms of Kleene algebras with range and additivity of range. Prover9 took less than 130s and provided a proof with 149 steps.

The results of this section further demonstrate the flexibility of our approach. Here, two reachability algorithms, to which preconditions, postconditions and invariants have been added as assertions, have been automatically verified. Such correctness proofs could run in the background and complement existing techniques for extended static checking. Again, reasoning about reachability required induction, which could be fully automated in Kleene algebra.

7 Synthesis of Szpilrajn's Algorithm

Our final case study is again on program construction. To further demonstrate the versatility of our approach, we now use relation algebra with sets modelled as vectors as the domain-specific theory behind the interface. We synthesise Szpilrajn's algorithm [23] that computes the linear extension of a given partial order. This synthesis is inspired by a previous manual correctness proof [6].

Initial Specification. Consider the following program construction task:

> *Given a finite partial order x, find a program with variable y that stores the linear extension of x after execution.*

Again, we conjecture that our program is a simple while loop:

```
... y:=x ...
while ... do
  ... y:=? ... od
```

Obviously, the precondition is that the input relation x is a partial order, a reflexive, antisymmetric and transitive relation. In our relational modelling language we can write:

```
1<=x  &  x*x^<=1  &  x;x<=x.
```

But we could also provide more declarative concepts such as `ref(x)`, `antisym(x)` and `trans(x)`. The postcondition is that the relational variable y stores a total order relation that extends x:

```
1<=y & y*y^<=1 & y;y<=y & x<=y & y+y^=U.
```

The fourth inequality states that the partial order relation y extends x; the last one expresses totality of y.

Developing the Invariant. The basic idea of Szpilrajn's algorithm is to build a chain of partial extensions of the partial order x by iteratively adding atoms (single ordered pairs) z that are incomparable by x, and incrementally computing the partial order for these extensions. Algorithmic details will again be part of the development, but we can now state the invariant:

The relation y is a partial order that contains x.

In our relational modelling language, this can be formalised as

```
1<=y & y*y^<=1 & y;y<=y & x<=y
```

Initialisation and Guard. We assume that our algorithm has only one global relational variable, namely y, which is initialised as x. Moreover, the while loop should terminate when no further extension of x can be computed, that is, when $y + y^\wedge = U$. The guard of the loop should therefore be `y+y^!=U`. We can justify these choices by verifying the following proof obligations:

Theorem 7. *The invariant is established by the initialisation (if the precondition holds); it establishes the postcondition when the guard of the loop is false.*

Proof. Using the axioms for relation algebras, these trivial ATP exercises needed no time. The invariant after initialisation *is* the precondition; the conjunction of the invariant and the negated guard of the loop *is* the postcondition. □

Termination and Development of the Loop. Termination of the algorithm is obvious, since only finitely many atoms can be added. As before, this is not further formalised.

To synthesise the loop body, we must determine the assignments to atoms z and the relational variable y. Obviously, z can be an arbitrary atom from the complement of $y + y^\wedge$. The variable y should be incremented by a function in y and z, that is, `y:=y+f(y,z)`. So we postulate that our program be of the following form, where `atom` is a choice function that picks some atom from $(y + y^\wedge)'$:

```
y:=x
while y+y^!=U do
  z:=atom((y+y^)')
  y:=y+f(y,z) od
```

The choice function can be axiomatised by `watom(atom(x)) & atom(x)!=0`, analogously to `point`. It remains to synthesise f and to show that the resulting assignment preserves the invariant whenever the guard of the loop is true. This can again be based on experiments with RelView, Prover9 and Mace4. But, since atoms are rectangles, Proposition 3 can again be used:

$$\mathsf{rtc}(y + z) = \mathsf{rtc}(y) + \mathsf{rtc}(y); z; \mathsf{rtc}(y).$$

Now $\mathsf{rtc}(y) = y$ since y is reflexive and transitive, which can be checked by ATP. Hence one of Szpilrajn's key insights can again be derived from our refinement law: `y:=y+y;z;y`.

Theorem 8. *Executing the loop preserves the invariant (if the guard of the loop is true).*

Proof. We assume that before executing the loop y is a partial order that extends x. We must prove that after execution the same properties hold of the new value of y. We use Prover9 with the axioms of relation algebras and the definition of weak atoms, but do not need the guard condition.

- The new value of y is a reflexive extension of x (this proof required no time):

  ```
  1<=y & y*y^<=1 & y;y<=y & x<=y -> x+1<=y+y;(z;y).
  ```

- The new value of y is transitive.

  ```
  y;y<=y & watom(z) & z<=(y+y')^
           -> (y+y;(z;y));(y+y;(z;y))<=(y+y;(z;y)).
  ```

  We used the axioms for relation algebra without the relation-algebraic definition of $U = x + x'$, but needed four additional (verified) assumptions,

  ```
  x;(y+z)=x;y+x;z  &  x<=y -> z;x<=z;y  &  x<=y -> x;z<=y;z  &  x<=U.
  ```

  and the definition of a weak atom from Section 3. These assumptions are very natural and should perhaps be included in any assumption set for relation algebras. The proof then took about 200s and has 158 steps.
- The new value of y is antisymmetric.

  ```
  y;y<=y & y*y^<=1 & watom(z) & z<=(y+y^)'
                    -> (y+y;(z;y))*(y+y;(z;y))^<=1.
  ```

  We could prove this goal from the axioms of relation algebras, and the definition of weak atom alone. The proof needed about 410s and has 274 steps, which is very long compared to similar experiments. □

The fact that additional assumptions were needed for proving transitivity may seem disappointing, but we can do better: Using idempotent semirings with converse (cf. Section 3) yielded a fully automated proof with 111 steps in less than 10s. Injections, vectors, weak points and weak atoms were used as before.

Of course we cannot use this simpler domain-specific theory for proving antisymmetry since this property cannot be expressed in Kleene algebra. To appreciate the complexity of proof search involved it should be mentioned that the manual relation-algebraic proof of antisymmetry [6] is quite involved: It requires several lemmas and covers almost an entire page. Proving the lemmas themselves is rather tedious and heavily involves the Schröder and Dedekind rules. Since these rules are difficult to prove by ATP, it is rather surprising that our automated correctness proof succeeds, possibly via a different route.

Partial Correctness. The result of this construction can be summed up in the main theorem of this section.

Theorem 9. *The following variant of Szpilrajn's algorithm for computing linear extensions of partial orders is (partially) correct:*

```
y:=x;
while y+y^!=U do
  z:=atom((y+y^)')
  y:=y+y;z;y od
```

Choosing the right domain-specific theories, the entire program construction could be automated from the theory axioms alone, that is, without any additional assumptions. The granularity of proof would again allow a fully automatic post-hoc verification with ATP systems in the background.

8 Discussion

Our main technical contribution is the demonstration that an integration of ATP systems and domain-specific algebras supports automatic correctness proofs for imperative programs through verification or program construction. We believe that these results motivate a new approach, to program construction in particular, which could combine simple high level program development techniques with computer enhanced mathematics based on domain-specific algebras and powerful proof automation. This section sketches some research questions and speculates about tools in which this approach could be implemented.

Theory Engineering. Relational calculi are not only useful for verifying and constructing algorithms for graphs, ordered sets, or other structures like games, Petri nets and lattices, they also form the basis for popular software development methods such as Alloy, B or Z. In preliminary experiments we have shown that the basic calculus of binary relations, as presented in the textbooks by Maddux [18] and Schmidt and Ströhlein [21] or Abrial's B-Book [1], can be automated [13]. Similar experiment in computer enhanced mathematics with reducts of relation algebras, in particular variants of idempotent semirings and Kleene algebras, are equally positive. While fully automated proofs of simpler theorems are possible from the theory axioms alone, more difficult goals require selecting appropriate lemmas (or even deleting "prolific" but unnecessary axioms). This suggests the following research questions: How can we organise and manage theory-specific and problem-specific knowledge to obtain useful assumption sets for ATP systems? How can we learn or abduce specific assumptions that are needed for particular proofs?

Relation algebras and variants of Kleene algebras capture the control flow in imperative programs and provide semantics for various computing applications, but they are less appropriate for modelling data structure or data types, or quantitative aspects of computations. Correctness proofs involving numbers, lists or arrays require different background theories or decision procedures. While their integration into our approach seems straightforward, their combination with relation algebras or Kleene algebras need further investigation.

Program Construction Technology. The most significant future task is to build program construction and verification tools that support the development of programs from specifications.

First, this requires the design of suitable modelling languages similar to those of Alloy or RelView. Second, existing libraries must be linked into a coherent data base. Third, to reason about data structures and data types, our present tool set should be complemented by SMT solvers and other decision procedures. Fourth, to manage the program construction process, automated tools need to be combined with interactive theorem provers which can handle the proof obligations of the Dijkstra-Gries approach or Hoare logic through tactics. These tools can also be used for residual inductive proofs that cannot be automated or for splitting complex proof goals into subgoals. Fifth, mechanisms for selecting appropriate theories behind the interface must be developed. We believe that our approach can largely be based on existing technology that only needs to be balanced in suitable ways. While for relation-based algorithms, development tools could be built around the RelView system, more general tools could support any other relation-based software development method.

The ultimate goal of our approach is to turn program construction into an activity in which the intellectually demanding and creative engineering tasks are separated — at an appropriate level of granularity — from routine calculations, such that developers can focus on the conceptual side of the construction while delegating technicalities to automated tools. The resulting approach would not only support teaching formal program development in more lightweight ways; it might also substantially increase the automation of existing formal methods for software development.

9 Conclusion

We have introduced a new approach to program construction and verification that is based on computer enhanced mathematics through a combination of domain-specific algebras with ATP systems, model generators and computer algebra tools. Using this combination we could prove the correctness of some standard algorithms fully automatically within the Dijkstra-Gries approach. While these results seem an interesting contribution per se, we see them predominantly as first steps within a larger programme aiming at lightweight formal methods with heavyweight automation. Traditional program construction techniques could thereby be lifted to a new level of simplicity and applicability. It seems feasible to realise this programme through a collective activity within the Mathematics of Program Construction community in the near future.

Acknowledgement. We want to thank Walter Guttmann and the anonymous referees for valuable remarks. We are grateful to Demetris Kennes for his help with preparing the proofs at our web site.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (1998)

3. Berghammer, R.: Combining Relational Calculus and the Dijkstra-Gries Method for Deriving Relational Programs. Information Sciences 119, 155–171 (1999)
4. Berghammer, R., Leoniuk, B., Milanese, U.: Implementation of Relation Algebra using Binary Decision Diagrams. In: de Swart, H. (ed.) RelMiCS 2001. LNCS, vol. 2561, pp. 241–257. Springer, Heidelberg (2002)
5. Berghammer, R., Neumann, F.: RelView – an OBDD-based computer algebra system for relations. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2005. LNCS, vol. 3718, pp. 40–51. Springer, Heidelberg (2005)
6. Berghammer, R.: Applying Relation Algebra and RelView to Solve Problems on Orders and Lattices. Acta Informatica 45, 211–236 (2008)
7. Cormen, T.H., Leiserson, C.E., Rivest, D.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
8. Desharnais, J., Struth, G.: Modal Semirings Revisited. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 360–387. Springer, Heidelberg (2008)
9. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
10. Floyd, R.W.: Assigning Meanings to Programs. In: Proc. AMS Symposia on Applied Mathematics, vol. 19, pp. 19–31 (1967)
11. Gries, D.: The Science of Computer Programming. Springer, Heidelberg (1981)
12. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10), 576–580 (1969)
13. Höfner, P., Struth, G.: Algebraic Reasoning with Prover9 (2009), `www.dcs.shef.ac.uk/~georg/ka/`
14. Höfner, P., Struth, G.: Automated Reasoning in Kleene Algebra. In: Pfenning, P. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 279–294. Springer, Heidelberg (2007)
15. Höfner, P., Struth, G.: On Automating the Calculus of Relations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 50–66. Springer, Heidelberg (2008)
16. Jackson, D.: Software Abstractions. The MIT Press, Cambridge (2006)
17. Kozen, D.: Kleene Algebra with Tests. ACM Trans. Program. Lang. Syst. 19(3), 427–443 (1997)
18. Maddux, R.D.: Relation Algebras. Elsevier, Amsterdam (2006)
19. McCune, W.: Prover9 and Mace4 (2007), `www.prover9.org`
20. Ng, J.: Relation Algebras with Transitive Closure. Ph.D. thesis, University of California, Berkeley (1984)
21. Schmidt, G., Ströhlein, T.: Relations and Graphs. Springer, Heidelberg (1993)
22. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, Englewood Cliffs (2006)
23. Szpilrajn, E.: Sur l'extension de l'ordre partiel. Fundamenta Math. 16, 386–389 (1930)
24. Tarski, A.: On the Calculus of Relations. J. Symbolic Logic 6, 73–89 (1941)
25. Warshall, S.: A Theorem on Boolean Matrices. Journal of the ACM 9, 11–12 (1962)