Can Refinement be Automated?

Peter Höfner[1] Georg Struth[2]

Department of Computer Science, University of Sheffield, UK

Abstract

We automatically verify Back's atomicity refinement law and a classical data refinement law for action systems. Our novel approach mechanises a refinement calculus based on Kleene algebras in an off the shelf resolution and paramodulation theorem prover and a counterexample checker with heuristics for hypothesis learning. The proofs are supported by a toolkit of meaningful refinement laws that has also been verified and that, for the first time, allows the refinement of programs and software systems, and the verification of further complex refinement laws, by automated deduction. This suggests that a substantial proportion of refinement could indeed be automated.

Keywords: refinement calculus; Kleene algebras; automated deduction; action systems; proof learning.

1 Introduction

The idea of stepwise refining specifications to programs is certainly intriguing: Provide software developers with a calculus of meaningful refinement laws and they will deliver code that is dependable, modular and optimised. Since the pioneering work of Dijkstra, Hoare and Back, refinement has matured into an established research area and been integrated into popular formal software development methods such as *VDM*, *Z* or *B*. However, the scientific success of refinement seems in contrast with its acceptance in industrial practice. Refinement calculi are complex formalisms that require considerable mathematical knowledge and that their integration into formal methods is usually achieved through interactive theorem provers that can be tedious to handle. A much wider acceptance of refinement could certainly be achieved through *automation*. But is that possible?

The development of refinement calculi over the decades can be characterised as an *algebraic turn* that led to considerable simplifications and abstractions. An important step was Back and von Wright's quantale-based approach that abstractly characterises the two fundamental models of refinement: binary relations and predicate transformers [6]. This approach is, however, essentially higher-order; a serious

[1] Email: p.hoefner@dcs.shef.ac.uk

[2] Email: g.struth@dcs.shef.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

obstacle against automation. More recently, in two seminal papers [35,36], von Wright reconstructed a substantial part of the refinement calculus in a variant of Kleene algebra. These *demonic refinement algebras* are entirely within first-order equational logic which, in principle, for the first time opens the way to automated deduction. But this intriguing potential has so far not been explored.

Automated deduction within formal methods remains, however, a challenge. Driven by the belief that theorem provers cannot handle complex algebraic axiom systems, considerable effort has been put into the development of special purpose calculi, but with rather limited practical impact (cf. [23]). Yet modern technology has changed this situation: the authors have recently demonstrated that off the shelf first-order theorem provers can successfully verify statements of considerable complexity and practical relevance if combined with a suitable algebra [15], e.g., Kleene algebra. This suggests that reasoning with demonic refinement algebras and therefore a substantial part of refinement can be automated as well.

This paper provides some evidence that this is indeed the case. We use McCune's Prover9 in combination with the counterexample search tool Mace4 [19] in a rather naïve approach, i.e, without much tuning. Our main results are as follows.

- We develop a toolkit of meaningful refinement laws based on demonic refinement algebras. It contains laws, e.g., for deconstructing and reconstructing concurrency, for simulation and for loop refinement.
- We use this toolkit for automatically verifying two complex laws for data refinement and atomicity refinement of action systems.
- We propose novel heuristics for hypothesis learning that seem indispensable for succeeding with more complex analysis tasks.

Our proof experiments are largely positive. Many basic refinement laws can be easily proved in a few seconds. In one particular example, our learning approach simplifies a proof from the literature; in another example it yields a more general theorem. Only for proving the atomicity refinement theorem, two steps were needed. But even von Wright's proof by hand of this theorem in demonic refinement algebra is almost two pages long and the proof search involved and the complexity of the axioms used is substantial. For the sake of readability we do not display all input/output files and the complete machine proofs. They can all be found at a web site [1].

So can refinement be automated? We believe that our approach has the potential of closing the prevailing gap between proof checking and model checking in formal methods. Mechanisations of refinement calculi with expressive higher-order interactive theorem provers have a long tradition [34,10], but these approaches do not offer the desirable degree of automation. Also the standard tools for mechanising software development in B or Z are highly interactive [2,3,26]. Model checking approaches to refinement have also been developed since more than one decade (cf. [25,16,14,18,28]). But these approaches are limited in expressiveness and users often pay for the ease of proof through the complexity of encoding. Automated deduction might provide a more suitable balance between expressiveness and automation. In this paper, at least a first step towards automation is certainly taken. And further pursuit of this challenging programme promises significant progress towards more applicable formal methods.

2 Kleene Algebras

The abstract refinement calculi studied in this paper are variants of Kozen's Kleene algebras [17] which are based on pioneering work of Salomaa [27], Conway [11], Park [22] and others. Particular strengths of Kleene algebras are syntactic simplicity, universal applicability, concise elegant equational proof-style and easy possibility of mechanisation. This already led to simplifications and unifications in various computer science and software engineering applications. Many arguments that previously required tedious semi-formal reasoning over pages can usually be reduced to their essence in a few lines of equational calculations.

An *idempotent semiring* is a structure $(S, +, \cdot, 0, 1)$ such that $(S, +, 0)$ is a commutative monoid with idempotent addition, $(S, \cdot, 1)$ is a monoid, multiplication distributes over addition from the left and right, and 0 is a left and right annihilator of multiplication. The axioms are, for all $x, y, z \in S$,

$$x + (y + z) = (x + y) + z, \qquad x + y = y + x, \qquad x + 0 = x, \qquad x + x = x,$$
$$x(yz) = x(yz), \qquad x1 = x = 1x,$$
$$x(y + z) = xy + xz, \qquad (x + y)z = xz + yz,$$
$$x0 = 0 = 0x.$$

Encodings for the automated theorem prover Prover9 are presented in Section 5. As usual in algebra, we stipulate that multiplication binds stronger than addition, and we omit the multiplication symbol. In the context of refinement, elements of S denote actions of a program, multiplication denotes sequential composition, addition denotes nondeterministic choice, 1 denotes the ineffective action and 0 the destructive action.

Two properties of idempotent semirings are important for our purposes.

- Every semiring induces an *opposite semiring* in which the order of multiplication is swapped. If a statement holds in a semiring, its dual holds in its opposite.
- The relation $\leq$ defined by $x \leq y \Leftrightarrow x + y = x$ for all elements x, y is a partial order. It is (up to isomorphism) the only order with least element 0 and for which addition and multiplication are isotone in both arguments (i.e. $x \leq y \Rightarrow x + z \leq y + z$ and likewise for all $x, y, z \in S$).

It follows that every idempotent semiring is also a semilattice $(S, \leq)$ with addition as join. Therefore, for all $x, y, z \in S$,

$$x + y \leq z \Leftrightarrow x \leq z \wedge y \leq z. \tag{1}$$

This law allows a case analysis of sums at left-hand sides of inequalities. It is very helpful for automated deduction since its keeps expressions small. There is no similar law for right-hand sides of inequalities. In the context of refinement, $\geq$ corresponds to the refinement ordering.

To model finite iteration or reflexive transitive closure, a further operation needs to be added. A *Kleene algebra* is a structure $(K, ^*)$ such that K is an idempotent semiring and *star* * is a unary operation axiomatised by the *star unfold axioms*

$$1 + xx^* \leq x^*, \qquad 1 + x^*x \leq x^*$$

and the *star induction axioms*

$$z + xy \leq y \Rightarrow x^*z \leq y, \qquad z + yx \leq y \Rightarrow zx^* \leq y,$$

for all $x, y, z \in S$. This axiomatises finite iterations within first-order logic with Park-style rules as least prefixed points (which are also least fixed points). By the first star unfold axiom, an iteration x^* is either ineffective, whence 1, or it continues after one single x-action. By the first star induction law x^* is the least element with that property. This form of iteration proceeds from left to right through a sequence. The second star unfold and star induction law are duals of the first ones with respect to opposition. They correspond to right-to-left iteration. The star is also isotone with respect to the ordering and the star unfold axioms can be strengthened to equations. These properties will also be used for theorem proving.

Kleene algebras have many interesting models. The most relevant ones for refinement are

- binary relations under union, relational composition and reflexive transitive closure with the empty and the unit relation;
- predicate transformers under union, function composition and iterated function application with the empty function and the identity function;
- sets of program traces under union, trace products (obtained by “glueing” traces together at their respective beginning and end points), reflexive transitive closure, the empty set and the set of all points from which traces can be built.

Paths, e.g., sequences of program states or languages, e.g., sequences of program actions immediately arise as special cases from traces.

Some interesting models are, however, excluded by the Kleene algebra axioms. First, the right zero law $x0 = 0$ seems inadequate for infinite actions x. Intuitively, it seems paradoxical to abort, for instance, the execution of an infinite loop.

Second, the predicate transformer model underlying the refinement calculus of Back and von Wright [6] is excluded as well. Only universally conjunctive or disjunctive predicate transformers satisfy the Kleene algebra axioms [36]. Here, universal conjunctivity (disjunctivity) means distributivity of multiplication over arbitrary infima (suprema). This of course implies distributivity over the empty infimum (supremum), which yields the right zero axiom. *Positively* conjunctive (disjunctive) predicate transformers, however, which only distribute over non-empty infima (suprema), do not satisfy the right zero axiom. But they are the appropriate models for demonically nondeterministic programs under the standard weakest precondition semantics.

Third, in some contexts, the left distributivity axiom $x(y + z) = xy + xz$ is inappropriate, for instance in the context of process algebras [8], tree languages [32] or probabilistic refinement [20]. We will, however, not further pursue this direction in this paper.

3 Demonic Refinement Algebras

To link Kleene algebras with the refinement calculus, von Wright has adapted the set of axioms in two ways: by dropping the right zero axiom and by adding a *strong iteration* operation which encompasses finite and infinite iteration [35]. The resulting structure is particularly suitable for modelling action system refinement [7].

Formally, a *demonic refinement algebra* is a structure $(K, {}^\infty)$ such that K is a Kleene algebra without the right zero axiom and the *strong iteration* ${}^\infty$ is a unary operation axiomatised by the *strong unfold* and the *strong coinduction* axiom

$$x^\infty = 1 + xx^\infty, \qquad\qquad y \le z + xy \Rightarrow y \le x^\infty z,$$

and linked with the star by the *isolation axiom*

$$x^\infty = x^* + x^\infty 0,$$

for all $x, y, z \in K$. Note that von Wright uses the notation of the refinement calculus while we adhere to that of Kleene algebra and language theory. The converse strong unfold law, $1 + x^\infty x = x^\infty$, follows from the axioms. Moreover, strong iteration is isotone with respect to the ordering.

The particular axioms of demonic refinement algebra can easily be motivated from the predicate transformer model with infinite iteration, cf. [36]. It is also immediately obvious that demonic refinement algebras do not capture the relational semantics, since in the presence of the right zero axiom, the isolation axiom collapses strong iteration to finite iteration. However, all theorems of demonic refinement algebra that do not mention strong iterations are also theorems of Kleene algebra.

4 Prover9 and Mace4

We use McCune's Prover9 tool [19] for proving theorems in demonic refinement algebra. Prover9 is a saturation-based theorem prover for first-order equational logic. It implements an ordered resolution and paramodulation calculus and, by its treatment of equality by rewriting rules and Knuth-Bendix completion, it is particularly suitable for algebraic reasoning. Further benefits of Prover9 are its extensive documentation and the fact that it is complemented by the tool Mace4 that searches for counterexamples. The combination of Prover9 and Mace4 is very useful in practice and supports the hypothesis learning heuristics described in Section 6.

Prover9 and Mace4 accept any input in a syntax for first-order equational logic. The input file consists essentially of a list of assumptions (the set of support), e.g., the axioms of demonic refinement algebra, and a goal to be proved. Prover9 then negates the goal, transforms the assumptions and the goal into clausal normal form and uses the inference rules of the ordered resolution and paramodulation calculus to derive a refutation. Mace4, in contrast, enumerates finite models of the assumptions and checks whether they satisfy the conditions expressed by the goal.

The inference procedure of saturation-based theorem proving is discussed in detail in the Handbook on Automated Reasoning [23]. Roughly, it consists of two interleaved modes.

- The deductive mode closes an initial clause set under the inference rules of resolution, factoring and paramodulation. The paramodulation rule essentially implements equational reasoning through the Leibniz principle that allows replacing equals by equals.
- The simplification mode discards clauses from the working clause set in case they are redundant with respect to another clause, e.g., when they are subsumed.

The inference process stops when the closure has been computed (never, perhaps) or when the empty clause `$F` — which denotes inconsistency — has been produced. In the second case, the tool reconstructs a proof by connecting the assumptions with the empty clause and displays this proof as a sequence. Examples are given below. This yields a semi-decision procedure for first-order logic. Whenever the goal is entailed by the assumptions, i.e., whenever the set of support plus the negation of the goal is inconsistent, the empty clause can be produced in finitely many steps. Due to the nature of the inference process, it is obvious that simplification rules are applied eagerly and deduction rules lazily to keep the working set small.

In practice, so-called syntactic orderings and other strategies are used to reduce the nondeterminism of the general inference procedure. The sizes of terms, literals and clauses are computed (recursively) from a given precedence of constant and function symbols and only "very big" terms in equations and literals in clauses are used in inferences. Moreover, "short" formulas are given precedence in inferences. The choice of strategy may have a crucial impact on the success of the proof search within reasonable time and memory constraints. In Prover9, it can therefore be declared as part of the input file. However, default strategies are provided for the less sophisticated user.

5 Automating Demonic Refinement Algebras

In demonic refinement algebras, inequalities and equations can be defined interchangeably. Every equation $x = y$ can be replaced by the two inequalities $x \leq y$ and $y \leq x$, whereas every inequality $x \leq y$ can be replaced by an equation $x+y = y$. Therefore, two different encodings of demonic refinement algebras are possible. An equational encoding for Prover9 and Mace4 is

```
op(500, infix, "+").   %declaration of operator precedences
op(490, infix, ";").   %multiplication
op(480, postfix, "*"). %finite iteration
op(480, postfix, "’"). %strong iteration

formulas(sos). %equational axioms of demonic refinement algebra
  x+y = y+x  &  x+0 = x  &  x+(y+z) = (x+y)+z  &  x+x = x.          %additive monoid
  x;1 = x  &  1;x = x  &  x;(y;z) = (x;y);z.                        %multiplicative monoid
  x;(y+z) = x;y+x;z  &  (x+y);z = x;z+y;z.                          %distributivity laws
  0;x = 0.                                                          %right zero law
  1+x;x* = x*  &  1+x*;x = x*.                                      %Kleene star
  ((x;y+z)+y = y -> x*;z+y = y)  &  ((y;x+z)+y = y -> z;x*+y = y).
  x’ = 1+x;x’  &  (y+(x;y+z) = x;y+z -> y+x’;z = x’;z).             %strong iteration
  x’ = x*+x’;0.
end_of_list.

formulas(sos). %verified laws that are useful in proofs
  (x+y)+z = z <-> (x+z = z & y+z = z).                              %case analysis
  (x+y = y -> x*+y* = y*)  &  (x+y = y -> x’+y’ = y’).              %isotonicity
end_of_list.

formulas(goals).
   %put proof goal here
end_of_list.
```

In the first part of the input, the precedence of operators is fixed. Here, the star and strong iteration bind stronger than multiplication which itself binds stronger than addition. The second part is a set of support list that contains the axioms of demonic refinement algebra. The third part is an additional set of support list that contains some useful laws of demonic refinement algebras that have previously been proved. The dual unfold law for strong iteration, for instance, will often be added to this set, too. The third part of the input lists the goal to be proved.

In a previous paper [15] we have used a similar equational encoding for Kleene algebras. Our experience shows that the approach works well for purely equational reasoning, but the size of terms is unnecessarily increased and application of isotonicity rules may require intricate matchings or unifications that are difficult to resolve for the theorem-prover. An inequational encoding, in contrast, yields shorter expressions and simpler isotonicity reasoning, but most resolution-based theorem provers do not support inequational reasoning, e.g., with focused chaining rules that would offer similar benefits as paramodulation.

The proof experiments from this paper show that an integration of equational and inequational reasoning is an essential step towards proving more complex theorems. To this end we define a special binary predicate `<=` that is assumed to be reflexive and transitive. In order to avoid duplication of proofs we do not add an axiom like $x \leq y \Leftrightarrow x+y = y$. We also do not use the antisymmetry rule. Therefore, strictly speaking, our proof search is incomplete with respect to demonic refinement algebra; but this doesn't show up in applications. The integrated approach still preserves the benefits of equational reasoning when performing paramodulation under inequality symbols, e.g. inferring $a \leq c$ from $a \leq b$ and $b = c$. Also, isotonicity axioms for the Kleene algebra operations must now be added explicitly to the set of axioms. The corresponding set of support is

```
  formulas(sos). %preorder axioms of demonic refinement algebra
    x+y = y+x  &  x+0 = x  &  x+(y+z) = (x+y)+z  &  x+x = x. %idempotent semiring
    x;1 = x & 1;x = x  &  x;(y;z) = (x;y);z.
    0;x = 0.
    x;(y+z) = x;y+x;z  &  (x+y);z = x;z+y;z.
    x <= x.                                                  %preorder
    x<=y & y<=z -> x<=z.
    1+x;x* = x*  &  1+x*;x = x*.                             %star
    (x;y+z<=y -> x*;z<=y)  &  (y;x+z<=y -> z;x*<=y).
    x’ = 1+x;x’  &  (y<=x;y+z -> y<=x’;z)  &  x’ = x*+x’;0.  %strong iteration
    (x<=y -> x+z<=y+z)                                       %isotonicity
    (x<=y -> x;z<=y;z)  &  (x<=y -> z;x<=z;y).
    (x<=y -> x*<=y*)  &  (x<=y -> x’<=y’).
  end_of_list.
```

The nature of the saturation-based proof procedure makes proof search often difficult to control. A restriction of the number of assumptions is usually helpful when a proof from the full set of axioms does not succeed, however, sometimes the presence of a seemingly unnecessary formula can make many formulas in the working set redundant, whence trigger their deletion. Some axioms, like commutativity of addition or the unfold laws, easily make the search space explode; some other axioms, like additional idempotence or the unit or zero laws can act as rewriting rules that may, under certain circumstances, considerably prune the search space. Comparing the number of operations and axioms of demonic refinement algebras with those of the example sets at the Prover9 web site and the literature, the fact that many theorems can nevertheless be proved from the full set of axioms in a naïve way was rather unexpected.

6 Basic Refinement Calculus

In this section we develop a toolkit of meaningful basic refinement laws in the context of demonic refinement algebra. It is common practice to use such laws for proving more complex refinement laws or for developing and analysing concrete refinements of programs and software systems. These more abstract laws can often replace the more low-level induction or coinduction axioms of Kleene algebra or demonic refinement algebra in computations.

We have already developed a toolkit of automatically verified theorems for Kleene algebras and several extensions that use the right zero axiom $x0 = 0$ [15]. Many of these theorems are also valid in the weaker context of demonic refinement algebra. Since the Prover9 output files present all hypotheses used for proving a particular goal, this can easily be checked. In those cases where Prover9 uses the right zero axiom, proofs can be replayed without this axiom or else the goal can be refuted by Prover9. Since this task is simple, but tedious and rather not interesting for this paper, we do not present a deeper discussion.

The main results of this paper deal with the refinement of concurrent systems, e.g. action systems. In this context, the expressions $(x+y)^*$ or $(x+y)^\infty$ denote the repeated concurrent execution of two processes x and y. Concurrency refinement can often analysed in three phases:

(i) the deconstruction of concurrency into interleaving;

(ii) the transformation of interleaving;

(iii) the reconstruction of concurrency.

We will automatically verify a basic refinement calculus that support these tasks. Many particular refinement laws can be found in [35]. All technical details, including the Prover9 and Mace4 input and output files, can be found at a web site [1].

The following auxiliary laws for strong iteration can be proved in a few seconds with Prover9 from the full set of axioms.

$$x \le 1^\infty, \qquad 1^\infty x = 1^\infty, \qquad (x^\infty)^\infty = 1^\infty, \qquad (x^*)^\infty = 1^\infty,$$
$$x^\infty = 1 + x^\infty x, \qquad 0^\infty = 1, \qquad x^\infty x^\infty = x^\infty, \qquad 1 \le x^\infty,$$
$$(x^\infty)^* = x^\infty, \qquad x^* x^\infty = x^\infty, \qquad x^\infty x^* = x^\infty.$$

The first law in the first row says that each demonic refinement algebra has a maximal element, namely 1^∞. The first law of the second row is the dual unfold law for strong iteration previously mentioned. The remaining laws collect further basic properties of strong iteration. Most of them can be used by Prover9 to simplify expressions. We also verified isotonicity of strong iteration, $x \le y \Rightarrow x^\infty \le y^\infty$, (in about three minutes) with the equational set of support before adding it to the inequational one.

We now list the more meaningful refinement laws for demonic refinement algebras verified with Prover9. Although there are usually similar laws for the star and for strong iteration, the proofs are often completely different.

- *Sliding laws.* These allow the sliding of loops over sequences.

$$x(yx)^* = (xy)^*x, \tag{2}$$
$$x(yx)^\infty = (xy)^\infty x, \tag{3}$$
$$(x^*y)^\infty = y^*(x^*y)^\infty. \tag{4}$$

- *Denesting laws.* These allow the reduction and reconstitution of concurrency.

$$(x+y)^* = x^*(yx^*)^*, \tag{5}$$
$$(x+y)^\infty = x^\infty(yx^\infty)^\infty, \tag{6}$$
$$(x+y)^\infty = (x^*y)^\infty x^\infty, \tag{7}$$
$$(x+y)^* = y^*x(x+y)^* + y^*, \tag{8}$$
$$(x+y)^\infty = y^*x(x+y)^\infty + y^\infty. \tag{9}$$

- *Simulation laws.* These are standard, e.g., in data refinement.

$$yx \le xz \Rightarrow y^*x \le xz^*, \tag{10}$$
$$xy \le zx \Rightarrow xy^* \le z^*x, \tag{11}$$
$$xy \le zx \Rightarrow xy^\infty \le z^\infty x, \tag{12}$$
$$yx \le xy \Rightarrow y^*x^* \le x^*y^*, \tag{13}$$
$$yx \le xy \Rightarrow y^\infty x^\infty \le x^\infty y^\infty. \tag{14}$$

  A dual law of (12), $yx \le xz \Rightarrow y^\infty x \le xz^\infty$, does not hold in demonic refinement algebra. Mace4 presents a counterexample with 3 elements.

- *Semicommutation laws.* These combine denesting with simulation.

$$yx \le xy \Rightarrow (x+y)^* \le x^*y^*, \tag{15}$$
$$yx \le xy \Rightarrow (x+y)^\infty \le x^\infty y^\infty. \tag{16}$$

  Since $x^*y^* \le (x+y)^*$ and $x^\infty y^\infty \le (x+y)^\infty$ (this has also been automatically verified), the right-hand sides can even be strengthened to equalities.

- *Disabledness laws.* These expresses that if y is always disabled by x then x before an iteration of y reduces to x.

$$xy = 0 \Rightarrow xy^* = x, \tag{17}$$
$$xy = 0 \Rightarrow xy^\infty = x. \tag{18}$$

Proofs by hand for most of these theorems can be found in [35]. Though short and concise, these proofs are often surprisingly difficult and require some familiarity with Kleene algebras.

Automating the proofs of these refinement laws was often straightforward, but some cases were non-trivial, computational expensive and required experimentation. In some cases, equations had to be split into two inequalities. In some further cases, the use of expensive axioms, as for instance commutativity of addition or unfolds, had to be prohibited. In some cases, we had to use lemmas that were already verified as assumptions. We also experimented with the hint feature of Prover9, which allows one to reuse previous statements or proofs to guide the proof search, but without much success. For the more important formulas of the above list, the requirements are listed in the following Figure 1. The computation times are loose upper bounds for a rather slow machine.

nr	(2)	(3)	(4)	(5)	(6)	(9)	(10)	(11)	(12)	(15)	(16)	(18)
t[s]	60	80	10	3	20	8	50	50	1	140	45	1
rest.	-	+	-	-	+	-	-	-	+	+	+	-
lem.	-	+	-	-	-	-	-	-	-	+	+	-

Figure 1. Requirements for proofs

Learning the appropriate set of restricted axioms and useful lemmas required some experimentation. Here, the interplay of Prover9 and Mace4 proved very helpful. We based our heuristics on the following universal principle of logic:

> *A conclusion cannot be proved from a set of premises if and only if there is some model in which all premises are true and the conclusion is false.*

In practice, when starting with a too small set of hypotheses, Mace4 usually finds a counterexample. When adding more and more hypotheses, Mace4 eventually will not return a counterexample in reasonable time. Then Prover9 should be started. Expensive axioms such as commutativity of addition or the unfold rules should be added as late as possible. If a proof without commutativity is not possible, it often helps to commute terms in the set of support and the goal. Handwritten proofs in variants of Kleene algebras are usually quite short, i.e., less than ten lines. The number of permutations of terms is therefore strongly limited.

These observations motivate some novel heuristics for hypothesis learning that should and could be automated as a pre-processing phase for Prover9. This automation might considerably increase the applicability of Prover9 and simplify proof search.

7 Denesting in Detail

In this section, we further discuss the automated proof of the non-trivial part of the denesting law (6), namely

$$(x+y)^{\infty} \leq x^{\infty}(yx^{\infty})^{\infty}.$$

Experimenting with our hypothesis learning heuristics, Prover9 can find a proof from the restricted axiom set

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

but without any further assumptions. The proof displayed by Prover9 after about 16s is

```
 1   x <= y ; x + z -> x <= y’ ; z                    # label(non_clause)    [ assumption ]
 2   (x + y)’ <= (x’ ; y)’ ; x’                                   # label(goal)    [ goal ]
 3   x + (y + z) = (x + y) + z                                               [ assumption ]
 4   (x + y) + z = x + (y + z)                                             [ copy 3, flip ]
 5   x ; 1 = x                                                               [ assumption ]
 6   x ; (y ; z) = (x ; y) ; z                                               [ assumption ]
 7   (x ; y) ; z = x ; (y ; z)                                             [ copy 6, flip ]
 8   x ; (y + z) = x ; y + x ; z                                             [ assumption ]
 9   x ; y + x ; z = x ; (y + z)                                           [ copy 8, flip ]
10   (x + y) ; z = x ; z + y ; z                                             [ assumption ]
11   x ; y + z ; y = (x + z) ; y                                          [ copy 10, flip ]
12   x <= x                                                                  [ assumption ]
13   x’ = x ; x’ + 1                                                         [ assumption ]
14   x ; x’ + 1 = x’                                                      [ copy 13, flip ]
15   -(x <= y ; x + z) | x <= y’ ; z                                         [ clausify 1 ]
16   -((c3 + c4)’ <= (c3’ ; c4)’ ; c3’)                                          [ deny 2 ]
17   x ; y + x = x ; (y + 1)                                                   [ para 5 9 ]
18   (x + y) ; z + u = x ; z + (y ; z + u)                                    [ para 11 4 ]
19   -((c3 + c4)’ <= c3’ ; (c4 ; (c3 + c4)’ + 1))                [ ur 15 16, rewrite 7 17 ]
20   -((c3 + c4)’ <= c3 ; (c3 + c4)’ + (c4 ; (c3 + c4)’ + 1))                  [ ur 15 19 ]
21   x ; (x + y)’ + (y ; (x + y)’ + 1) = (x + y)’                            [ para 18 14 ]
22   -((c3 + c4)’ <= (c3 + c4)’)                            [ back_rewrite 20, rewrite 21 ]
23   $F                                                                   [ resolve 22 12 ]
```

Of course, this proof is only a tiny part of the proof search, which in this case consists of more than 1800 steps. The entire output file can be found at [1]. The machine proof can easily be retranslated into the usual equational style of reasoning with Kleene algebra. The essential part of the machine proof starts at line 19; the previous part containing mainly rearrangements of axioms.

First, to prove the claim, it suffices by strong coinduction to show that

$$(x+y)^\infty \leq x^\infty y(x+y)^\infty + x^\infty = x^\infty(y(x+y)^\infty + 1).$$

This step corresponds to line 19 of the machine proof. Second, applying strong coinduction again, it suffices to show that

$$(x+y)^\infty \leq x(x+y)^\infty + y(x+y)^\infty + 1.$$

This corresponds to line 20. Third, this inequality holds, since

$$x(x+y)^\infty + y(x+y)^\infty + 1 = (x+y)(x+y)^\infty + 1 = (x+y)^\infty$$

by distributivity and strong unfold. This corresponds to the remaining lines of the machine proof. Interestingly, the equational reconstruction of the proof found by Prover9, although it does not use any additional assumptions, is simpler than the one in [35].

8 Data Refinement

Back and von Wright have presented several laws for data refinement of action systems in the predicate transformer setting [6]. von Wright has already translated one of these laws into demonic refinement algebra. In this section, we provide an automated proof of this law. Here, for the first time, we leave the level of pure demonic refinement algebra and predominantly work at the more abstract level of refinement laws introduced and verified in Section 6. Our hypothesis learning heuristics now becomes particularly important for finding the right set of support for Prover9.

Our analysis of the data refinement law not only illustrates the applicability of our refinement toolkit. Based on previous work on diagrammatic reasoning in Kleene algebra [31,13], it also shows that the usual refinement diagrams can in principle be recovered from the output of Prover9.

In traditional Kleene algebra notation, the data refinement law considered is written as follows.

Theorem 8.1 *Let $b^\infty = b^*$, $za' \leq az$, $zb \leq z$, $s' \leq sz$ and $ze' \leq e$. Then*

$$s'(a' + b)^\infty e' \leq sa^\infty e.$$

The first hypothesis says that b cannot loop infinitely. The second hypothesis expresses that a is data refined by a' with respect to upward simulation z. By the third hypothesis, 1 is data refined by b. The fourth and fifth condition expresses the standard data refinement of initialisations and finalisations.

The proof planning is now greatly simplified in the presence of some experience with data refinement. It can be expected that some forms of denesting and semi-commutation will suffice for deconstructing and reconstructing concurrency. For the transformation of interleaving, simulation laws seem highly relevant, whereas the laws of the additive monoid seem avoidable. The effect of the star and strong iteration axioms is hopefully captured by simulation, so that these laws should only be added if necessary. These considerations provide a basis for hypothesis learning. In fact, the following set of demonic refinement axioms is sufficient for a proof

```
formulas(sos).
  x;(y;z) = (x;y);z.
  x <= x  &  (x<=y & y<=z -> x<=z).
  (x<=y -> x;z<=y;z)  &  (x<=y -> z;x<=z;y)  &  (x<=y -> x*<=y*)  &  (x<=y -> x’<=y’).
end_of_list.
```

and the following three refinement laws are needed.

```
formulas(sos).
  (x+y)’ = y’;(x;y’)’.                              %denest
  (z;y<=x;z -> z;y’<=x’;z)  &  (x;y<=x -> x;y*<=x).  %simulation
end_of_list.
```

From these sets of support, Prover9 returns after 10s with the proof

```
  1   x <= y & y <= z -> x <= z                        # label(non_clause)   [ assumption ]
  2   x <= y -> x ; z <= y ; z                         # label(non_clause)   [ assumption ]
  3   x <= y -> z ; x <= z ; y                         # label(non_clause)   [ assumption ]
  4   x ; y <= z ; x -> x ; y’ <= z’ ; x               # label(non_clause)   [ assumption ]
  5   x ; y <= x -> x ; y * <= x                       # label(non_clause)   [ assumption ]
  6   (all s all ss all e all ee all a all aa all b
        (b’ = b * & x ; aa <= a ; x & x ; b <= x & ss <= s ; x & x ; ee <= e
         ->
        ss ; ((aa + b)’ ; ee) <= s ; (a’ ; e)))                    # label(goal)   [ goal ]
  7   x ; (y ; z) = (x ; y) ; z                                                [ assumption ]
  8   (x ; y) ; z = x ; (y ; z)                                               [ copy 7, flip ]
  9   -(x <= y) | -(y <= z) | x <= z                                           [ clausify 1 ]
 10   -(x <= y) | x ; z <= y ; z                                               [ clausify 2 ]
 11   -(x <= y) | z ; x <= z ; y                                               [ clausify 3 ]
 12   (x + y)’ = y’ ; (x ; y’)’                                                [ assumption ]
 13   x’ ; (y ; x’)’ = (y + x)’                                              [ copy 12, flip ]
 14   -(x ; y <= z ; x) | x ; y’ <= z’ ; x                                     [ clausify 4 ]
 15   -(x ; y <= x) | x ; y * <= x                                             [ clausify 5 ]
 16   c8 * = c8’                                                                   [ deny 6 ]
 17   c1 ; c7 <= c6 ; c1                                                           [ deny 6 ]
 18   c1 ; c8 <= c1                                                                [ deny 6 ]
 19   c3 <= c2 ; c1                                                                [ deny 6 ]
 20   c1 ; c5 <= c4                                                                [ deny 6 ]
 21   -(c3 ; ((c7 + c8)’ ; c5) <= c2 ; (c6’ ; c4))                                 [ deny 6 ]
```

```
22   c1 ; (c7 ; x) <= c6 ; (c1 ; x)                       [ hyper 10 17, rewrite 8 8 ]
23   c1 ; c8' <= c1                                       [ hyper 15 18, rewrite 16 ]
24   c3 ; x <= c2 ; (c1 ; x)                               [ hyper 10 19, rewrite 8 ]
25   x ; (c1 ; c5) <= x ; c4                                          [ hyper 11 20 ]
26   x ; (c1 ; c8') <= x ; c1                                         [ hyper 11 23 ]
27   c1 ; (c8' ; x) <= c1 ; x                              [ hyper 10 23, rewrite 8 ]
28   -(c2 ; (c1 ; ((c7 + c8)' ; c5)) <= c2 ; (c6' ; c4))              [ ur 9 24 21 ]
29   -(c1 ; ((c7 + c8)' ; c5) <= c6' ; c4)                              [ ur 11 28 ]
30   -(c1 ; (c8' ; ((c7 ; c8')' ; c5)) <= c6' ; c4)         [ para 13 29, rewrite 8 ]
31   -(c1 ; (c8' ; ((c7 ; c8')' ; c5)) <= c6' ; (c1 ; c5))            [ ur 9 25 30 ]
32   c1 ; (c7 ; c8') <= c6 ; c1                                     [ hyper 9 22 26 ]
33   -(c1 ; ((c7 ; c8')' ; c5) <= c6' ; (c1 ; c5))                    [ ur 9 27 31 ]
34   c1 ; (c7 ; c8')' <= c6' ; c1                                     [ hyper 14 32 ]
35   c1 ; ((c7 ; c8')' ; x) <= c6' ; (c1 ; x)             [ hyper 10 34, rewrite 8 8 ]
36   $F                                                           [ resolve 35 33 ]
```

Again, a translation of the machine proof into equational format is not too difficult. The initialisation, the finalisation and the refinement of the loop can be separated. The initialisation and finalisation imply that it is sufficient to show that $sz(a'+b)^\infty e' \le sa^\infty ze'$ and therefore

$$z(a'+b)^\infty \le a^\infty z$$

by isotonicity. The left-hand side of this expression can be denested and, using the assumptions $b^\infty = b^*$, $zb \le z$ and the simulation law, be simplified to

$$z(a'+b)^\infty = zb^\infty(a'b^\infty)^\infty = zb^*(a'b^*)^\infty \le z(a'b^*)^\infty.$$

But $z(a'b^*)^\infty \le a^\infty z$ follows from the strong simulation law and $za'b^* \le azb^* \le az$.

This algebraic reasoning can easily be translated into diagrams, as used in term rewriting (cf. [33]) and refinement (cf. [6]). This has extensively been discussed in [13]. In particular, glueing of diagrams along edges corresponds to isotonicity reasoning. Therefore, the following diagrams describe the transition from the succedent of the data refinement theorem to the analysis of the infinite loop.

$$\begin{tikzcd}
 & \cdot \arrow[r, "(a'+b)^\infty"] & \cdot \arrow[dr, "e'"] & \\
\cdot \arrow[ur, "s'"] \arrow[dr, "s"'] & & & \cdot \\
 & \cdot \arrow[r, "a^\infty"'] & \cdot \arrow[ur, "e"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \cdot \arrow[r, "(a'+b)^\infty"] & \cdot \arrow[dr, "e'"] & \\
\cdot \arrow[ur, "s'"] \arrow[dr, "s"'] & & & \cdot \\
 & \cdot \arrow[r, "a^\infty"'] \arrow[uu, "z"] & \cdot \arrow[ur, "e"'] \arrow[uu, "z"] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "(a'+b)^\infty"] & \cdot \\
\cdot \arrow[u, "z"] \arrow[r, "a^\infty"'] & \cdot \arrow[u, "z"']
\end{tikzcd}$$

The next sequence of diagrams describes essentially the reasoning in the above sequence of equations.

$$\begin{tikzcd}
\cdot \arrow[r, "b^*"] & \cdot \arrow[r, "(a'b^*)^\infty"] & \cdot \\
\cdot \arrow[u, "z"] \arrow[rr, "a^\infty"'] & & \cdot \arrow[u, "z"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "b^*"] & \cdot \arrow[r, "(a'b^*)^\infty"] & \cdot \\
\cdot \arrow[u, "z"] \arrow[ur, "z"'] \arrow[rr, "a^\infty"'] & & \cdot \arrow[u, "z"']
\end{tikzcd}$$

The last two diagrams describe the reasoning from the simulation assumption to the loop via the strong simulation law (12).

$$\begin{tikzcd}
\bullet \arrow[r, "(a'b^*)^\infty"] & \bullet \\
\bullet \arrow[u, "z"] \arrow[r, "a^\infty"'] & \bullet \arrow[u, "z"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\bullet \arrow[r, "a'"] & \bullet \arrow[r, "b^*"] & \bullet \\
\bullet \arrow[u, "z"] \arrow[r, "a"'] & \bullet \arrow[u, "z"] \arrow[ur, dash, "z"'] &
\end{tikzcd}$$

The correspondence between algebra and diagrams could be made more precise by putting the refinement laws into diagrammatic form. Demonic refinement algebra would then yield an algebraic semantics for these diagrams. A further elaboration of this correspondence seems very promising for proof visualisation.

It is obvious from the axioms of demonic refinement algebra that downward simulations cannot be modelled in this setting. There are nevertheless two extensions that capture this additional concept. The first one introduces converses x° to each action x. A further discussion can be found in [31]. The second one introduces forward and backward modalities. A discussion of the relation to simulation and converse can be found in [12,21]. An automation of these extensions seems feasible. Modal Kleene algebras have already successfully treated by Prover9 [15].

9 Atomicity Refinement

In 1989 Back proved a rather complex atomicity refinement theorem for action systems by reasoning over sequences of program states [5]. This proof spreads over several pages. It has been replayed later at the level of predicate transformers and, even more concisely, in demonic refinement algebra [35,36]. The proof in demonic refinement algebra still fills almost two pages. Our automated analysis reveals some glitches in this proof. The following theorem presents a cleaned-up version of this law in demonic refinement algebra.

Theorem 9.1 *Let $s \le sq$, $a \le qa$, $qb = 0$, $rb \le br$, $(a+r+b)l \le l(a+r+b)$, $rq \le qr$, $ql \le lq$, $r^* = r^\infty$ and $q \le 1$. Then*

$$s(a+r+b+l)^\infty q \le s(ab^\infty q + r + l)^\infty.$$

A discussion about the intuition behind the assumptions and the theorem can be found in [5]. For the symbol pushing of automated theorem proving this information is not needed. Given the length of von Wright's proof it is no surprise that Prover9 does not succeed in one full sweep. However, an automated proof up to the reconstruction of concurrency is possible. We therefore split Theorem 9.1 into two lemmas.

Lemma 9.2

(i) $s(a+r+b+l)^\infty q \le sl^\infty qr^\infty q(ab^\infty qr^\infty)^\infty$ follows from the conditions of Theorem 9.1 except $q \le 1$.

(ii) Let $q \le 1$. Then $sl^\infty qr^\infty q(ab^\infty qr^\infty)^\infty \le s(ab^\infty q + r + l)^\infty$.

As before, the proofs heavily rely on hypothesis learning. The following set of support can be used for both proofs. The restricted demonic refinement axioms are

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

The basic refinement laws are

```
formulas(sos).
  y;x<=x;y -> (y+x)'=x';y'.                               %semicommutation
  (x';y') <= (y+x)'.                                      %trivial part of semicommutation
  (x+y)' = y';(x;y')'.                                    %denesting
  (x;y)';x = x;(y;x)'.                                    %sliding
  (z;y<=x;z -> z;y'<=x';z)  &  (y;z<=z;x -> y*;z<=z;x*). %simulation
  x;y=0 -> x;y'=x.                                        %disabledness
end_of_list.
```

The goal for Lemma 9.2(i) is

```
formulas(goals).
  all a all b all l all q all r all s(
    a<=q;a  &  r;b<=b;r  &  (a+(r+b));l<=l;(a+(r+b))  &  r;q<=q;r
    &  r*=r'  &  q;l<=l;q  &  q;b=0  &  s<=s;q
    ->
    s;(((a+(r+b))+l)';q) <= (s;(l';q));((r';q);((a;b');(q;r'))')).
end_of_list.
```

We also keep the full set of assumption in the proof of Lemma 9.2(ii), although this is not strictly necessary.

```
formulas(goals).
  all a all b all l all q all r all s(
     a<=q;a  &  r;b<=b;r  &  (a+(r+b));l<=l;(a+(r+b))  &  r;q<=q;r
     &  r*=r'  &  q;l<=l;q  &  q;b=0  &  s<=s;q  &  q<=1
     ->
     (s;(l';q));((r';q);((a;b');(q;r'))') <= s;(((a;b');q+r)+l)').
end_of_list.
```

The machine proof of Lemma 9.2(i) requires about 1200s and 75 steps; it is displayed in Figure 2. Although it is not particularly readable for humans, it can again be translated into equational style.

$$\begin{aligned}
s(a+b+r+l)q &= sl^\infty(a+b+r)^\infty q \\
&= sl^\infty(b+r)^\infty a(b+r)^\infty)^\infty q \\
&= sl^\infty b^\infty r^\infty(ab^\infty r^\infty)^\infty q \\
&\le sl^\infty b^\infty r^\infty(qab^\infty r^\infty)^\infty q \\
&= sl^\infty b^\infty r^\infty q(ab^\infty r^\infty q)^\infty \\
&\le sql^\infty b^\infty r^\infty q(ab^\infty r^\infty q)^\infty \\
&\le sl^\infty qb^\infty r^\infty q(ab^\infty r^\infty q)^\infty \\
&\le sl^\infty qr^\infty q(ab^\infty r^\infty q)^\infty \\
&= sl^\infty qr^\infty q(ab^\infty r^* q)^\infty \\
&\le sl^\infty qr^\infty q(ab^\infty qr^*)^\infty \\
&= sl^\infty qr^\infty q(ab^\infty qr^\infty)^\infty .
\end{aligned}$$

The first step uses the semicommutation law (16). The second step uses the denesting law (6). The third step uses again the semicommutation law (16) . The fourth step uses the assumption $a \le qa$. The fifth step uses the sliding law (3). The sixth step uses the assumption $s \le sq$. The seventh step uses the simulation law (12). The eighth step uses the disabledness law (18) and the assumption $qb = 0$. The ninth step uses the assumption $r^\infty = r^*$. The tenth step uses the star simulation law (10). The eleventh step uses again $r^\infty = r^*$. Given the length of the equational

```
1    x ; 1 = x & 1 ; x = x                                              # label(non_clause)   [ assumption ]
2    x <= y & y <= z -> x <= z                                          # label(non_clause)   [ assumption ]
3    x <= y -> x ; z <= y ; z                                           # label(non_clause)   [ assumption ]
4    x <= y -> z ; x <= z ; y                                           # label(non_clause)   [ assumption ]
5    x <= y -> x’ <= y’                                                 # label(non_clause)   [ assumption ]
6    x ; y <= y ; x -> (x + y)’ = y’ ; x’                               # label(non_clause)   [ assumption ]
7    x ; y <= z ; x -> x ; y’ <= z’ ; x                                 # label(non_clause)   [ assumption ]
8    x ; y <= y ; z -> x * ; y <= y ; z *                               # label(non_clause)   [ assumption ]
9    x ; y = 0 -> x ; y’ = x                                            # label(non_clause)   [ assumption ]
10   (all a all b all l all q all r all s
       (a <= q ; a & r ; b <= b ; r & (a + (r + b)) ; l <= l ; (a + (r + b)) & r ; q <= q ; r
       & r * = r’ & q ; l <= l ; q & q ; b = 0 & s <= s ; q
       ->
       s ; (((a + (r + b)) + l)’ ; q) <= (s ; (l’ ; q)) ; ((r’ ; q) ; ((a ; b’) ; (q ; r’))’)))     # label(goal)   [ goal ]
11   x ; 1 = x                                                                                   [ clausify 1 ]
12   1 ; x = x                                                                                   [ clausify 1 ]
13   x ; (y ; z) = (x ; y) ; z                                                                   [ assumption ]
14   (x ; y) ; z = x ; (y ; z)                                                                 [ copy 13, flip ]e
15   x <= x                                                                                      [ assumption ]
16   -(x <= y) | -(y <= z) | x <= z                                                              [ clausify 2 ]
17   -(x <= y) | x ; z <= y ; z                                                                  [ clausify 3 ]
18   -(x <= y) | z ; x <= z ; y                                                                  [ clausify 4 ]
19   -(x <= y) | x’ <= y’                                                                        [ clausify 5 ]
20   -(x ; y <= y ; x) | y’ ; x’ = (x + y)’                                                      [ clausify 6 ]
21   -(x ; y <= y ; x) | (x + y)’ = y’ ; x’                                                    [ copy 20, flip ]
22   (x + y)’ = y’ ; (x ; y’)’                                                                   [ assumption ]
23   x’ ; (y ; x’)’ = (y + x)’                                                                 [ copy 22, flip ]
24   x’ ; y’ <= (y + x)’                                                                         [ assumption ]
25   (x ; y)’ ; x = x ; (y ; x)’                                                                 [ assumption ]
26   -(x ; y <= z ; x) | x ; y’ <= z’ ; x                                                        [ clausify 7 ]
27   -(x ; y <= y ; z) | x * ; y <= y ; z *                                                      [ clausify 8 ]
28   x ; y != 0 | x ; y’ = x                                                                     [ clausify 9 ]
29   c1 <= c4 ; c1                                                                                  [ deny 10 ]
30   c5 ; c2 <= c2 ; c5                                                                             [ deny 10 ]
31   (c1 + (c5 + c2)) ; c3 <= c3 ; (c1 + (c5 + c2))                                                 [ deny 10 ]
32   c5 ; c4 <= c4 ; c5                                                                             [ deny 10 ]
33   c5 * = c5’                                                                                     [ deny 10 ]
34   c4 ; c3 <= c3 ; c4                                                                             [ deny 10 ]
35   c4 ; c2 = 0                                                                                    [ deny 10 ]
36   c6 <= c6 ; c4                                                                                  [ deny 10 ]
37   -(c6 ; (((c1 + (c5 + c2)) + c3)’ ; c4) <= (c6 ; (c3’ ; c4)) ; ((c5’ ; c4) ; ((c1 ; c2’) ; (c4 ; c5’))’))    [ deny 10 ]
38   -(c6 ; (((c1 + (c5 + c2)) + c3)’ ; c4)
       <= c6 ; (c3’ ; (c4 ; (c5’ ; (c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’)))))               [ copy 37, rewrite 14 14 14 14 ]
39   (x + x)’ = x’ ; x’                                                                          [ hyper 21 15 ]
40   x ; ((y ; z)’ ; y) = x ; (y ; (z ; y)’)                                         [ para 25 14, rewrite 14 ]
41   0’ = 1                                                                         [ hyper 28 12, rewrite 12 ]
42   c1 ; x <= c4 ; (c1 ; x)                                                        [ hyper 17 29, rewrite 14 ]
43   (c5 + c2)’ = c2’ ; c5’                                                                      [ hyper 21 30 ]
44   ((c1 + (c5 + c2)) + c3)’ = c3’ ; (c1 + (c5 + c2))’                                          [ hyper 21 31 ]
45   -(c6 ; (c3’ ; ((c1 + (c5 + c2))’ ; c4))
       <= c6 ; (c3’ ; (c4 ; (c5’ ; (c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’)))))            [ back_rewrite 38, rewrite 44 14 ]
46   c5’ ; c4 <= c4 ; c5’                                                        [ hyper 27 32, rewrite 33 33 ]
47   c4 ; c3’ <= c3’ ; c4                                                                        [ hyper 26 34 ]
48   c4 ; c2’ = c4                                                                               [ hyper 28 35 ]
49   c6 ; x <= c6 ; (c4 ; x)                                                        [ hyper 17 36, rewrite 14 ]
50   x’ ; (x’ ; (y ; (x’ ; x’))’) = (y + (x + x))’                                [ para 39 23, rewrite 39 14 ]
51   x’ <= (0 + x)’                                                                  [ para 41 24, rewrite 11 ]
52   c4 ; (c2’ ; x) = c4 ; x                                                                 [ para 48 14, flip ]
53   x’ <= (0 + (0 + x))’                                                                     [ hyper 16 51 51 ]
54   (c1 ; x)’ <= (c4 ; (c1 ; x))’                                                               [ hyper 19 42 ]
55   x ; (c5’ ; c4) <= x ; (c4 ; c5’)                                                            [ hyper 18 46 ]
56   c4 ; (c3’ ; x) <= c3’ ; (c4 ; x)                                            [ hyper 17 47, rewrite 14 14 ]
57   -(c6 ; (c4 ; (c3’ ; ((c1 + (c5 + c2))’ ; c4)))
       <= c6 ; (c3’ ; (c4 ; (c5’ ; (c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’)))))                         [ ur 16 49 45 ]
58   -(c4 ; (c3’ ; ((c1 + (c5 + c2))’ ; c4)) <= c3’ ; (c4 ; (c5’ ; (c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’))))    [ ur 18 57 ]
59   1 <= (0 + (0 + 0))’                                                                          [ para 41 53 ]
60   x <= (0 + (0 + 0))’ ; x                                                        [ hyper 17 59, rewrite 12 ]
61   (x ; y)’ ; x <= (0 + (0 + 0))’ ; (x ; (y ; x)’)                                              [ para 40 60 ]
62   (c1 ; x)’ ; y <= (c4 ; (c1 ; x))’ ; y                                                       [ hyper 17 54 ]
63   (x + (0 + 0))’ = x’                                    [ para 41 50, rewrite 41 41 41 11 11 12 12, flip ]
64   (x ; y)’ ; x <= x ; (y ; x)’                                             [ back_rewrite 61, rewrite 63 41 12 ]
65   -(c3’ ; (c4 ; ((c1 + (c5 + c2))’ ; c4)) <= c3’ ; (c4 ; (c5’ ; (c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’))))    [ ur 16 56 58 ]
66   -(c4 ; ((c1 + (c5 + c2))’ ; c4) <= c4 ; (c5’ ; (c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’)))                 [ ur 18 65 ]
67   -(c4 ; (c5’ ; ((c1 ; (c2’ ; c5’))’ ; c4))
       <= c4 ; (c5’ ; (c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’)))                  [ para 23 66, rewrite 43 43 14 14 14 52 ]
68   -(c5’ ; ((c1 ; (c2’ ; c5’))’ ; c4) <= c5’ ; (c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’))                      [ ur 18 67 ]
69   -((c1 ; (c2’ ; c5’))’ ; c4 <= c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’)                                      [ ur 18 68 ]
70   -((c4 ; (c1 ; (c2’ ; c5’)))’ ; c4 <= c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’)                            [ ur 16 62 69 ]
71   -(c4 ; (c1 ; (c2’ ; (c5’ ; c4)))’ <= c4 ; (c1 ; (c2’ ; (c4 ; c5’)))’)          [ ur 16 64 70, rewrite 14 14 ]
72   -((c1 ; (c2’ ; (c5’ ; c4)))’ <= (c1 ; (c2’ ; (c4 ; c5’)))’)                                         [ ur 18 71 ]
73   -(c1 ; (c2’ ; (c5’ ; c4)) <= c1 ; (c2’ ; (c4 ; c5’)))                                               [ ur 19 72 ]
74   -(c2’ ; (c5’ ; c4) <= c2’ ; (c4 ; c5’))                                                             [ ur 18 73 ]
75   $F                                                                                             [ resolve 74 55 ]
```

Figure 2. Proof of Lemma 9.2(i)

proof, the success of Prover9 seems quite impressive. In general we find that machine proofs and their equational reconstructions in the usual style of Kleene algebra differ by a factor between 5 and 10. A main reason is that term rearrangements due to associativity or commutativity are usually not displayed in hand-written proofs.

The proof of Lemma 9.2(ii) with Prover9 is much simpler than the previous one. It requires a few milliseconds, has 30 steps and can be found at our web site. Its translation into an equational proof is not particularly interesting. The combination of the two parts of Lemma 9.2 has not been achieved, since the introduction of the additional assumption $q \leq 1$, which is needed in the second proof, leads to an explosion of the search space.

It might be possible to obtain a fully automated proof with a chaining-based prover, which provides a more effective treatment of inequational reasoning. To our knowledge, this is not available in state of the art theorem provers. Again, the proofs can be translated into diagrams, but we do not further pursue this direction.

10 Atomicity Refinement Light

Our attempts to prove the atomicity refinement theorem lead us to consider simplified variants. Setting $r = 0$ and $l = 0$, an automated proof can be obtained in a few seconds. Setting only $l = 0$, Theorem 9.1 simplifies as follows.

Proposition 10.1 *Let $s = sq$, $a = qa$, $qb = 0$, $(a + b)l \leq l(a + b)$, $ql \leq lq$ and $q \leq 1$. Then*

$$s(a + b + l)^{\infty}q = s(ab^{\infty}q + l)^{\infty}.$$

A fully automated proof of this statement is possible. The input file, based on our hypothesis learning heuristics, is

```
formulas(sos).
  x;1 = x  &  1;x = x  &  x;(y;z) = (x;y);z.
  x <= x  &  x<=y & y<=z -> x<=z.
end_of_list.
```

The refinement laws assumed are

```
formulas(sos).
  y;x <= x;y -> (y+x)’=x’;y’.                %semicommutation
  z;(x’;y’)<=z;(y+x)’.                       %inverse semicommutation with context
  (x+y)’= y’;(x;y’)’.                        %denesting
  (x;y)’;x=x;(y;x)’.                         %sliding
  z;y<=x;z -> (v;(z;y’));w<=(v;(x’;z));w.  %simulation with context
  y<=1 -> (x;y);z<= x;z.                     % one with context
  x;y = 0 -> x;y’=x.                         %disabledness
end_of_list.
```

Here, contexts are hard-coded into the rules to avoid the free generation of unnecessary contexts by the isotonicity laws. The proof goal is

```
formulas(goals).
  all s all a all b all l all q
    (s=s;q  &  a=q;a  &  q;b = 0  &  (a+b);l<=l;(a+b)  &  q;l <= l;q  &  q<=1
    ->
    s;(((a+b)+l)’;q)<=s;((a;b’);q+l)’).
end_of_list.
```

The proof takes 1013s and has 46 steps. Since the output of the mechanised proof is again not particularly readable, we will only present its equational translation and refer the interested reader to our web site. The equational proof is

$$\begin{aligned} s(a+b+l)^\infty q &= sl^\infty(a+b)^\infty q \\ &= sl^\infty b^\infty(ab^\infty)^\infty q \\ &= sl^\infty b^\infty(qab^\infty)^\infty q \\ &= sl^\infty b^\infty q(ab^\infty q)^\infty \\ &= sql^\infty b^\infty q(ab^\infty q)^\infty \\ &\leq sl^\infty qb^\infty q(ab^\infty q)^\infty \\ &= sl^\infty qq(ab^\infty q)^\infty \\ &\leq sl^\infty(ab^\infty q)^\infty \\ &= s(ab^\infty q+l)^\infty . \end{aligned}$$

The first step applies the semicommutation law (16). The second step uses the denesting law (6). The third step uses the assumption $a = qa$. The fourth step uses the sliding law (3). The fifth step uses the assumption $sq = s$. The sixth step uses the simulation law (12). The seventh step uses the assumption $qb = 0$ and the disabledness law (18). The eight step uses the assumption $q \leq 1$. The last step uses isotonicity and $x^\infty x^\infty = x^\infty$.

The assumptions $a = qa$ and $sq = s$ could again, as in the case of Theorem 9.1, be weakened to $s \leq sq$ and $a \leq qa$.

11 Discussion

The refinement laws presented support automated reasoning in demonic refinement algebra at the level of Back and von Wright's refinement calculus. Our particular examples consider action system refinement, i.e., the refinement of concurrent and reactive systems. The correspondence between demonic refinement algebras and action systems has been described in detail by von Wright [35,36].

This paper focuses on one particular aspect of refinement, namely the automated verification of *refinement laws*. Our paper shows that this can be dealt with entirely at the abstract algebraic level.

Concrete refinement proofs in software development processes, however, require a different approach. Here, the abstract algebraic level is used similarly to categories in areas like differential geometry (for coordinate-free reasoning). Facts about concrete models (data types, program stores, tests, actions or traces, abstraction relations) can be expressed abstractly as *bridge lemmas* in the language of demonic refinement algebra and then treated by automated deduction. The concrete level can be treated by suitable solvers or theorem provers for set theory, arithmetics, lists, arrays and likewise. This is illustrated by the following two examples:

- We have already described the approach in the context of Hoare logic, where (arithmetical) properties of assignments and stores are casted into bridge lemmas, whereas the inference rules dissolve in Kleene algebra [15].
- For data refinement, simulation properties between abstract and concrete datatypes and abstraction relations that relate the two levels must be determined at the relational and set-theoretic level. Again, these properties then provide bridge

lemmas for refinement laws in demonic refinement algebra, e.g. for simulation or for loop transformation, as treated in this paper.

The development of suitable solvers and special-purpose provers is an interesting research question. The automated derivation of abstraction relations for data refinement has recently been considered [9,24]; saturation-based calculi for sets have also been developed [29,30]. The integration of existing solvers and decision procedures for specific data structures into our framework is certainly possible.

12 Conclusion

We have shown that a substantial part of demonic refinement algebra can be mechanised in an automated theorem prover, that a useful toolkit of refinement laws can be developed and be automatically verified and that some refinement laws of considerable complexity can be automatically verified.

A further significant contribution of this paper certainly consists in the diverse research questions that arise from these results.

From the refinement point of view, it seems very interesting to extend demonic refinement algebras to encompass also downward simulations, which is needed in data refinement. As already mentioned, this could be achieved either by adding an operation of converse or through modal operators. More generally, modal demonic refinement algebras would considerably increase the expressiveness of the approach.

A second task is the integration of other variants of Kleene algebras into the refinement toolkit (cf. Section 2) in order to reason about refinements in process algebras and of probabilistic systems.

A third question concerns the proof presentation and the integration of diagrammatic reasoning, which is very common in the refinement community, but could only be sketched in this paper. While a translation from diagrammatic statements to Kleene algebra and Prover9 input is straightforward, the retranslation of machine proofs into equational and diagrammatic format seems more involved. Such an extended input and output might lead to more userfriendly tools.

From the theorem proving point of view, a first task is the automation of the hypothesis learning heuristics proposed in this paper. The search for feasible hypotheses seems easily parallelisable and indispensable for proving complex theorems.

Beyond the rather naïve approach taken in this paper, it seems also interesting to evaluate the impact of the more sophisticated mechanisms provided by Prover9 for guiding the proof search, i.e., the manipulation of syntactic orderings, the assignment of weights to literals and clauses and the use of hints. It might be possible that more complex proofs can be fully automated and hypothesis learning can be reduced with these mechanisms.

Finally, the implementation of an integrated chaining and paramodulation prover that allows combined efficient reasoning with inequalities and equations (cf. [4]) might drastically improve the proof search. The example of refinement shows that automated reasoning with inequalities is perhaps more difficult, but surely not less interesting than its equational counterpart.

Acknowledgements.
We are most grateful to Mark Schaefer for placing his powerful PC at our disposal for proof search. With our own slow machines we would possibly not have met the deadline. We would also like to thank our referee for interesting pointers to the literature.

References

[1] `http://www.dcs.shef.ac.uk/~georg/ka`.

[2] Atelier B, `http://www.atelierb.societe.com`.

[3] B-Core(UK) Ltd, *The B-Tool*, `http://www.b-core.com/btool.html`.

[4] Bachmair, L. and H. Ganzinger, *Ordered chaining calculi for first-order theories of transitive relations*, J. ACM **45** (1998), pp. 1007–1049.

[5] Back, R.-J., *A method for refining atomicity in parallel algorithms*, in: E. Odijk, M. Rem and J.-C. Syr, editors, *Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science **366** (1989), pp. 199–216.

[6] Back, R.-J. and J. von Wright, "Refinement Calculus: A Systematic Introduction," Graduate Texts in Computer Science, Springer, 1998.

[7] Back, R.-J. and J. von Wright, *Reasoning algebraically about loops*, Acta Informatica **36** (1999), pp. 295–334.

[8] Benson, D. B. and J. Tiuryn, *Fixed points in free process algebras, Part I*, Theoretical Computer Science **63** (1989), pp. 275–294.

[9] Butler, M., *On the use of data refinement in the development of secure communication systems*, Formal Aspects of Computing **14** (2002), pp. 2–34.

[10] Butler, M., J. Grundy, T. Långbacka, R. Rukšėnas and J. von Wright, *The refinement calculator: Proof support for program refinement*, in: L. Groves and S. Reeves, editors, *Formal Methods Pacific'97*, Springer, 1997 pp. 40–61.

[11] Conway, J. H., "Regular Algebra and Finite Machines," Chapman & Hall, 1971.

[12] Desharnais, J., B. Möller and G. Struth, *Kleene algebra with domain*, ACM Trans. Computational Logic **7** (2006), pp. 798–833.

[13] Ebert, M. and G. Struth, *Diagram chase in relational system development*, in: M. Minas, editor, *3rd IEEE Workshop on Visual Languages and Formal Methods*, Elecronic Notes in Theoretical Computer Science **127** (2005), pp. 87–105.

[14] Formal Systems (Europe) Limited, *FDR2*, `http://www.fsel.com`.

[15] Höfner, P. and G. Struth, *Automated reasoning in Kleene algebra*, in: *CADE 2007*, Lecture Notes in Artificial Intelligence **4603** (2007), pp. 279–294.

[16] Jackson, D., *Abstract model checking of infinite specifications*, in: M. Naftalin, T. Denvir and M. Bertran, editors, *FME '94*, Lecture Notes in Computer Science **873** (1994), pp. 519–531.

[17] Kozen, D., *A completeness theorem for Kleene algebras and the algebra of regular events*, Information and Computation **110** (1994), pp. 366–390.

[18] Leuschel, M. and M. Butler, *Automatic refinement checking for b*, in: K.-K. Lau and R. Banach, editors, *ICFEM'05*, Lecture Notes in Computer Science **3785** (2005), pp. 345–359.

[19] McCune, W., *Prover9 and Mace4*, `http://www.cs.unm.edu/~mccune/prover9`.

[20] McIver, A. K., E. Cohen and C. C. Morgan, *Using probabilistic Kleene algebra for protocol verification*, in: R. A. Schmidt, editor, *Relations and Kleene Algebra in Computer Science*, Lecture Notes in Computer Science **4136** (2006), pp. 296–310.

[21] Möller, B. and G. Struth, *Algebras of modal operators and partial correctness*, Theoretical Computer Science **351** (2006), pp. 221–239.

[22] Park, D., *On the semantics of fair parallelism*, in: D. Bjørner, editor, *Abstract Software Specifications*, Lecture Notes in Computer Science **86** (1980), pp. 504–526.

[23] Robinson, J. A. and A. Voronkov, editors, "Handbook of Automated Reasoning (in 2 volumes)," Elsevier and MIT Press, 2001.

[24] Robinson, N. J., *Incremental derivation of abstraction relations for data refinement*, in: J. S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering*, Lecture Notes in Computer Science **2885** (2003), pp. 246–265.

[25] Roscoe, A. W., *Model-checking csp*, in: R. A. W., editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, Prentice Hall, 1994 pp. 353–378.

[26] Saaltink, M., *The z/eves system*, in: J. Bowen, M. Hinchey and D. Till, editors, *ZUM '97*, Lecture Notes in Computer Science **1212** (1997), pp. 72–85.

[27] Salomaa, A., *Two complete axiom systems for the algebra of regular events*, J. ACM **13** (1966), pp. 158–169.

[28] Smith, G. and J. Derrick, *Verifying data refinements using a model checker*, Formal Aspects of Computing **18** (2006), pp. 264–287.

[29] Struth, G., *A calculus for set-based program development*, in: J. S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering*, Lecture Notes in Computer Science **2885** (2003), pp. 541–559.

[30] Struth, G., *Automated element-wise reasoning with sets*, in: J. R. Cuellar and Z. Liu, editors, *Proc. SEFM 2004* (2004), pp. 320–329.

[31] Struth, G., *Abstract abstract reduction*, Journal of Logic and Algebraic Programming **66** (2006), pp. 239–270.

[32] Takai, T. and H. Furusawa, *Monodic tree Kleene algebra*, in: R. A. Schmidt, editor, *Relations and Kleene Algebra in Computer Science*, Lecture Notes in Computer Science **4136** (2006), pp. 402–416.

[33] Terese, editor, "Term Rewriting Systems," Cambridge University Press, 2003.

[34] von Wright, J., *Program refinement by theorem prover*, in: *6th Refinement Workshop* (1994).

[35] von Wright, J., *From Kleene algebra to refinement algebra*, in: E. A. Boiten and B. Möller, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science **2386** (2002), pp. 233–262.

[36] von Wright, J., *Towards a refinement algebra*, Science of Computer Programming **51** (2004), pp. 23–45.