Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a preprint version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/91550

Please be advised that this information was generated on 2018-07-08 and may be subject to change.

Peter Höfner
Annabelle McIver
Georg Struth (eds.)

1st Workshop

on
Automated Theory Engineering

Proceedings

Preface

This volume contains the proceedings of the first Workshop on Automated Theory Engineering which was held in Wrocław, Poland, on July 31, 2011 as a satellite workshop of the 23rd International Conference on Automated Deduction (CADE).

Theory engineering, a term coined by Tony Hoare, means the development and mechanisation of mathematical axioms, definitions, theorems and inference procedures as needed to cover the essential concepts and analysis tasks of an application domain. It is essential for the qualitative and quantitative modelling and analysis of computing systems. The aim of theory engineering is to present users with domain specific modelling languages, and to devolve the technical intricacies of analysis tasks as far as possible to tools that provide heavyweight automation.

Theory engineering is relevant to the design of systems, programs, APIs, protocols, algorithms, design patterns, specification languages, programming languages and beyond. It involves technologies, such as interactive and automated theorem proving systems, satisfiability and satisfiability modulo theories solvers, model checkers and decision procedures.

The aim of this workshop was to bring together tool and theory developers with industrial engineers to exchange experiences and ideas that stimulate further tool developments and theory designs.

The diversity of topics relevant to theory engineering is reflected by the contributions to this volume. Each paper was refereed by at least three reviewers on its originality, technical soundness, quality of presentation and relevance to the workshop. The programme included two invited lectures by experts in the area: "An Overview of Methods for Large-Theory Automated Theorem Proving" by Josef Urban (Radboud University, The Netherlands), and "Do Formal Methodists have Bell-Shaped Heads?" by Timothy G. Griffin (University of Cambridge, UK).

We would like to thank our colleagues without whose help and support the workshop would not have been possible. First, Aaron Stump, the CADE workshop and tutorial chair and the local organisers Hans de Nivelle, Katarzyna Wodzyńska and Tomasz Wierzbicki for all their help. Second, the authors who supported this workshop by submitting papers and the two invited speakers for their contributions. Third, the members of the Programme Committee for carefully reviewing and selecting the papers. Finally, it is our pleasure to acknowledge the generous financial support by NICTA and the Department of Computer Science of the University of Sheffield.

Sheffield and Sydney, July 2011

Peter Höfner
Annabelle McIver
Georg Struth

Organisation

Programme Committee

Michael Butler	University of Southampton, UK
Ewen Denney	NASA, US
Peter Höfner	NICTA, Australia
Joe Hurd	Galois, Inc., US
Rajeev Joshi	NASA (JPL), US
Annabelle McIver	Macquarie University/NICTA, Australia
Stephan Merz	INRIA, France
Marius Portmann	University of Queensland/NICTA, Australia
Georg Struth	University of Sheffield, UK
Geoff Sutcliffe	University of Miami, US

Workshop Organisers

Peter Höfner	NICTA, Australia
Annabelle McIver	Macquarie University/NICTA, Australia
Georg Struth	University of Sheffield, UK

Sponsors

NICTA, Australia
Department of Computer Science, The University of Sheffield, UK

Table of Contents

Do Formal Methodists have Bell-Shaped Heads?

(Invited Paper)

Timothy G. Griffin

Computer Laboratory
University of Cambridge
Timothy.Griffin@cl.cam.ac.uk

Abstract

Data networking has not been kind to formal methods. The Internet's birth gave rise to an intense culture war between the bell-heads and the net-heads, which the net-heads have largely won. In this area formal methodists have long been seen as the humourless enforcers of the defeated bell-heads. The result: formal methods are not a part of the mainstream of data networking and are largely relegated to the the thankless task of reverse engineering (security-related protocols are perhaps the rare exception). If we want to move beyond this situation we must build tools that enhance the ability to engage in the Internet culture — tools that encourage community-based development of open-source systems and embrace the open-ended exploration of design spaces that are only partially understood.

1 State the Problem Before Describing the Solution?

The title of this section, sans question mark, is the same as a one-page put-down of the culture of computer networking written by Leslie Lamport in 1978 [7]. The idea is simple and seductive. Start with a specification of the problem that is independent of any solution! Then describe your solution and show that it solves the problem. What could be more simple? Motherhood and apple pie when it comes to technologies like fly-by-wire, software on deep space probes, control systems for nuclear power plants, and so on.

But did this make sense in 1978 for the early pioneers of the Internet? Does it make sense today now that the Internet has grown from lab experiment to ubiquitous infrastructure? I don't think so. The reasons are obvious. The traditional (formal) approach to protocol development is to start with a specification [5]. But we can only specify a problem when we understand what we are doing. Part of the thrill of systems-related networking in the past decades has been in exploring virgin territory opened up by Moore's law and related exponential curves in bandwidth and memory. In addition, the exploration has not been conducted in a top-down manner, but rather in a grass-roots bottom-up way. And the active community has never agreed on exactly where it was going then or now.

When I was hired at Bell Laboratories in 1991 research in Internet-related technologies was essentially banned. The networking professionals (bell-heads) knew that circuits where the only realistic technology and that the "junk" from EE and operating systems (net-heads) would never work. Bell Labs didn't change this policy until about 1997, by which time it was too late (see Day's interesting book [4]). Sadly, the situation may be a bit worse in Europe than elsewhere. Many EU countries set up national telecom institutes during the 1970's, which had the unfortunate consequence of isolating communications engineers from computer scientists. Elsewhere communications has been absorbed into computer science!

In networking, formal methods has historically been associated with the losing side of these culture wars.

2   Satisfied with Reverse-Engineering?

If Internet protocols are not developed from formal specifications it may still be worthwhile to *reverse engineer* existing protocols. I've done a lot of reverse engineering with routing protocols. I think it can be very fruitful, especially if we can come away with general principles that have applicability beyond protocol-specific details. This last point is very important — your results must be interesting to a larger community because you can't expect the networking community to care. By the time you have figured something out, they will have moved on to a new set of problems.

3   Change of Thinking?

Given the cultural constraints, perhaps reverse engineering is the best we can do. Attempts have been made to lower the cost of formal methods, see the work on *lightweight formal methods* [1, 6]. I think that it is a very worthwhile effort, but one that is useful in *all* areas where formal methods are applied. Is there an approach that fully embraces the community-based open-ended nature of Internet-related systems work?

I'll suggest one answer — domain-specific languages (DSLs). It seems to me that DSLs allow us to raise the level of abstraction in which problems are solved. I will discuss only three examples — the Statecall Policy Language (SPL) [8], Ynot [3], and Idris [2].

I've had fun discussing this topic with Anil Madhavapeddy, Jon Crowcroft, and Boon Thau Loo. I hope this short talk will stimulate further discussion at the ATE workshop.

References

[1] S. Agerholm and P. G. Larsen. A lightweight approach to formal methods. *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*, 1998.

[2] E. Brady. Idris - systems programming meets full dependent types. In *PLPV 2011*, 2011.

[3] A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceedings of ICFP'09*, 2009.

[4] J. Day. *Patterns in Network Architectures : A return to fundamentals.* Prentice Hall, 2008.

[5] M. G. Gouda. *Elements of Network Protocol Design.* Wiley, 1998.

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

[7] L. Lamport. State the problem before describing the solution. *ACM SIGSOFT Software Engineering Notes*, 3(1):26, 1978.

[8] A. Madhavapeddy. Combining static model checking with dynamic enforcement using the statecall policy language. In *International Conference on Formal Engineering Methods (ICFEM)*, 2009.

An Overview of Methods for Large-Theory Automated Theorem Proving

(Invited Paper)

Josef Urban

Radboud University Nijmegen

`josef.urban@gmail.com`

Abstract

This is an attempt at a brief initial overview of the state of the art in the young field of first-order automated reasoning in large theories (ARLT). It is necessarily biased by the author's imperfect knowledge, and hopefully will serve as material provoking further corrections and completions.

1 Why Large Theories?

Why should we want to (automatically) reason in large theories and develop them instead of small theories? Here are several answers:

- Mathematicians work in large theories. They know a lot of concepts, facts, examples and counter-examples, proofs, heuristics, and theory-development methods.

- Other scientists (and humans in general) work with large theories. Consider physics, chemistry, biology, law, politics, large software libraries, Wikipedia, etc. Our current knowledge about the world is large.

- In the last years, more and more knowledge is becoming available formally by all kinds of human efforts (interactive theorem proving, common-sense reasoning, knowledge bases for various sciences, Semantic Web, etc.). This is an opportunity for automated reasoning to help with the sciences and tasks mentioned above.

- Existing resolution/superposition automated reasoning systems often derive large numbers of facts, even from small initial number of premises. Managing such large numbers can profit from specialized large-theory techniques.

Automated reasoning in large theories is today often about increasing the comfort of users of automated reasoning methods: It is typically possible to *manually* select premises from which some conjecture should follow. Often this is even a significant part of one's formal reasoning wisdom. But ultimately, *manual* is the opposite of *automated*.

1.1 Large Formal Theories Are Not Our Enemy

However, *automated selection of relevant facts* is only the very first step that recently made existing ATP methods usable and useful in large theories. This premise-selection view treats large theories to a large extent only as an ATP person's *enemy*: We need to select the few right facts from the large pile of less relevant facts before we get down to the "real science" of "doing ATP".

This is in the author's opinion a very limited view of the large-theory field. The bigger reason for making large complex theories and knowledge bases available to the automated reasoning world is that they can contain a large amount of domain-specific problem-solving knowledge,

and likely (in less explicit form), also a large amount of general problem-solving knowledge that the automated reasoning field should reveal and integrate into its pool of methods.

For this, however, another limited view needs to be overcome: Large theories (and theories in general) are not just random collections of usable facts. Mathematical theories in particular have been developed by smart people over centuries, and quite likely such theories are the best, deeply computer-understandable corpus of *abstract human thinking* that we currently have. It seems negligent to ignore the internal theory structure, and the problem-solving and theory-engineering knowledge developed by mathematicians so far. Especially when we know that first-order ATP is an undecidable problem, and that the current ATP methods are on average far behind what trained mathematicians can do.

Thus, large complex formal theories and knowledge bases are not an enemy, but an opportunity. Not just an opportunity to reason with the knowledge of many already established facts, but also an opportunity to analyze and learn how smart people reason and prove difficult theorems, develop their conceptual space, and how they find surprising connections and solutions. In short, large formal theories are a great new playground for developing general AI. But because general AI (and theorem-proving oriented AI in particular) has been in the second half of the 20th century labeled as unproductive, general AI research in this field should go hand-in-hand with practical applications and usability testing. So far, this has fortunately often been the case in this young field.

2   Corpora

Several large formal knowledge bases have become recently available to experiments with first-order automated reasoning tools. To name the major ones (in alphabetic order):

- The CYC (OpenCyc, ResearchCyc) common-sense knowledge base [16]

- The Isabelle/HOL mathematical library [10]

- The Mizar/MML mathematical library [25]

- The SUMO (and related ontologies) common-sense knowledge base [13]

It is likely that more will follow (or already are available). For example, the HOL Light/Flyspeck [4, 5] large mathematical library should benefit from similar first-order translation techniques as the Isabelle/HOL library. More common-sense knowledge bases like YAGO [21] might be produced by semi-automated methods, and bridges to all kinds of specialized scientific databases are being build, spearheaded by systems like Biodeducta [19]. The LogAnswer project [3] has already started to reason over the first-order export of the *full texts* of German Wikipedia.

The corpora differ in their purpose/origin, size, complexity, consistency, completeness, and the extent to which they cover various large-theory aspects. The common-sense ontologies contain a lot of classification/hierarchical knowledge, resulting typically in simple Horn clauses, and also a lot of concept definitions with relatively few facts proved about them. Storing and maintaining proofs has so far been a secondary aspect. Their primary emphasis was not (so far) on building up libraries of more and more advanced proved theorems about the world, but rather on covering as many concepts as possible by suitable definitions.

On the other hand, the mathematical theories have a much larger number of nontrivial mathematical theorems in them, and their formal content typically follows some established

4

informal theory developments based on well-known and fixed mathematical foundations. There is more concept/fact re-use in mathematics, and nontrivial proofs of many facts exist and (at least in theory) can be made available in common formats and for large-theory techniques based on inspection of previous proofs and theory developments.

3   Automated Methods for Reasoning in Large Theories

The existing large-theory reasoning methods can be divided into several groups, using various criteria. One criterion is the method used for knowledge selection. The methods developed so far include syntactic heuristics, heuristics using semantic information, methods that look at previous solutions, and combinations thereof. Systems and methods that make use mainly of syntactic criteria for premise selection include:

- The SInE (SUMO Inference Engine) algorithm by Kryštof Hoder [6], and its E implementation by Stephan Schulz.[1] The basic idea is to use global frequencies of symbols to define their global *generality*, and build a relation linking each symbol S with all formulas F in which S is has the lowest global generality among the symbols of F. In common-sense ontologies, such formulas typically *define* the symbols linked to them, which is the reason for calling this relation a *D-relation*. Premise selection for a conjecture is then done by recursively following the D-relation, starting with the conjecture's symbols. Various parameters can be used, e.g., limiting the recursion depth significantly helps for the Mizar library [26], and preliminary experiments show that also for the Isabelle/HOL library.

- The default premise selection heuristic used by the Isabelle/Sledgehammer export [11] seems to be quite similar to SInE, however it works internally in Isabelle, and uses additional mechanisms like blacklisting. D-relation is not used there, the formulas are linked to all symbols they contain.

- The *Conjecture Symbol Weight* clause selection heuristics in E prover [18] give lower weights to symbols contained in the conjecture, thus preferring during the inference steps the clauses that have common symbols with the conjecture. This is remotely similar to general *goal-oriented* ATP techniques, as for example the Set of Support (SoS) strategy in resolution/superposition provers,[2]. Note that also the majority of tableau calculi are in practice goal-oriented, and the leanCoP [12] prover in particular performs surprisingly well on the MPTP Challenge large-theory benchmark.

A method which is purely signature-based, however the word *semantics* appears in it, is *latent semantics*. Latent semantics is a machine learning method that has been successfully used for example in the Netflix Challenge, and in web search. Its principle is to automatically derive "semantic" equivalence classes of words (like *car, vehicle, automobile*) from their co-occurrences in documents, and to use such equivalence classes (also called *synsets* in the WordNet ontology) instead of the original words for searching and related tasks. This technique has been so far used in:

- Paul Cairns' Alcor system [1] for searching and advice over the Mizar library.

- Yuri Puzis' initial relevance ordering of premises used in the SRASS ATP metasystem [22].

---

[1] http://www.mpi-inf.mpg.de/departments/rg1/conferences/deduction10/slides/stephan-schulz.pdf
[2] In particular, SPASS [30] has been used successfully on the Isabelle data.

Semantics (in the original logical sense) has been used for a relatively long time in various ways for guiding the ATP inference processes. An older system that is worth mentioning with respect to the current efforts is John Slaney's SCOTT system [20] constraining Otter inferences by validity in models. A similar idea has been recently revived by Jiří Vyskočil at the Prague ATP seminar: His observation was that mathematicians have very fast conjecture-rejection methods based on a (relatively small) pool of (often imprecise) models in their heads, similar to some fast heuristic software testing methods. This motivated Petr Pudlák's semantic axiom selection system for large theories [15], implemented later also by Geoff Sutcliffe in SRASS. The basic idea is to use finite model finders like MACE [9] and Paradox [2] to find counter-models of the conjecture, and gradually select axioms that exclude such counter-models. The models can differentiate between a formula and its negation, which is typically beyond the heuristic symbolic means. This idea has been also used later in the MaLARea system [27], however in the context of many problems solved simultaneously and many models kept in the pool, and using the models found also as classification features for machine learning.

MaLARea is also an example of a system that uses learning from previous proofs for guiding premise-selection for new conjectures. The idea of this approach is to define suitable features characterizing conjectures (symbolic, semantic, structural, etc.), and to use machine learning methods on available proofs to learn the function that associates the conjecture features with the relevant premises. A sophisticated learning approach has been suggested and implemented in E prover by Stephan Schulz for his PhD work [17], which unfortunately preceded the appearance of large theories by several years.[3] In this approach, proofs are abstracted into proof traces, consisting of clause patterns in which symbol names are abstracted into higher-order variables. Such proof traces from many proofs are collected into a common knowledge base, which is loaded when a new problem is solved, and used for guiding clause selection. This is probably quite similar to the *hints* technique in Prover9 [8], which however seems to be used more in a single-problem proof-shortening scenario.

Note that such techniques already move the large-theory techniques towards smart general-purpose ATP techniques for proof guidance. A recent attempt in this direction is the MaLeCoP system [28]. There, the clause relevance is learned from all closed tableau branches, and the tableau extension steps are guided by a trained machine learner that takes as input features a suitable encoding of the literals on the current tableau branch. In some sense this tries to transfer the promising premise selection techniques deeper into the core of ATP systems. Unlike the above mentioned technique used in E prover, the advising is however left to external systems, which communicate with the prover over a sufficiently fast link.

4   More Systems and Metasystems

Not all systems do premise selection, however they may be still worth of mentioning.

One way how to reason with full large theories is to significantly limit the reasoning power. At the extreme, such methods become the many search methods available for the corpora mentioned above. A somewhat more involved memorization/reasoning technique is *subsumption* implemented in various ATP systems. A type-aware extension of subsumption is implemented for the Mizar library in the MoMM system [24]. Extending such limited systems further in a controlled and restricted way might be quite rewarding.

---

[3]The author and Stephan Schulz have shortly tried to revive this old E code and test it on the MPTP Challenge benchmark in 2007, however without any significant results. So this advanced code is still waiting to be properly revived and tested.

Another interesting large-theory techniques is lemmatization and concept creation. An example lemmatization system has been implemented by Petr Pudlák in his PhD thesis [14]: The system uses lemmas found in successful proofs to enrich the whole theory, find new proofs, and shorten existing ones. Concept creation is a long-time AI research, going back to Lenat's seminal work on AM [7]. Recently, concept creation has been tried to shorten long, automatically produced proofs in [29]. Refactoring of proofs into human-digestible form seems to be a very interesting task that we are facing more and more as the automated methods are getting more and more usable. As computers are getting better in solving hard and large problems, we should also make them better in explaining their solutions to us.

References

[1] P. A. Cairns. Informalising formal mathematics: Searching the Mizar library with latent semantics. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *MKM*, volume 3119 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2004.

[2] K. Claessen and N. Sorensson. New Techniques that Improve MACE-style Finite Model Finding. In P. Baumgartner and C. Fermueller, editors, *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[3] U. Furbach, I. Glöckner, and B. Pelzer. An application of automated reasoning in natural language question answering. *AI Commun.*, 23(2-3):241–265, 2010.

[4] T. C. Hales. Introduction to the flyspeck project. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[5] T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.

[6] K. Hoder and A. Voronkov. Sine qua non for large theory reasoning. In *CADE 11*, 2011. To appear.

[7] D. Lenat. *An Artificial Intelligence Approach to Discovery in Mathematics*. PhD thesis, Stanford University, Stanford, USA, 1976.

[8] W.W. McCune. Prover9. http://www.mcs.anl.gov/ mccune/prover9/.

[9] W.W. McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA, 2003.

[10] J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *J. Automated Reasoning*, 40(1):35–60, 2008.

[11] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.

[12] J. Otten and W. Bibel. leanCoP: Lean Connection-Based Theorem Proving. *Journal of Symbolic Computation*, 36(1-2):139–161, 2003.

[13] A. Pease and G. Sutcliffe. First order reasoning on a large ontology. In G. Sutcliffe et al. [23].

[14] P. Pudlak. Search for Faster and Shorter Proofs using Machine Generated lemmas. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 34–52, 2006.

[15] P. Pudlak. Semantic selection of premises for automated theorem proving. In G. Sutcliffe et al. [23].

[16] D. Ramachandran, Reagan P., and K. Goolsbey. First-orderized ResearchCyc: Expressiveness and Efficiency in a Common Sense Knowledge Base. In P. Shvaik , editor, *Proceedings of the Workshop on Contexts and Ontologies: Theory, Practice and Applications*, 2005.

[17] S. Schulz. *Learning Search Control Knowledge for Equational Deduction*. PhD thesis, Technische Universität München, Munich, Germany, 2000.

[18] S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.

[19] J. Shrager, R. Waldinger, M. Stickel, and J. P. Massar. Deductive biocomputing. *PLoS ONE*, 2(4):e339, Apr 2007.

[20] J. K. Slaney, E. Lusk, and W. W. McCune. SCOTT: Semantically Constrained Otter, System Description. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, number 814 in Lecture Notes in Artificial Intelligence, pages 764–768. Springer-Verlag, 1994.

[21] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A large ontology from Wikipedia and WordNet. *J. Web Semantics*, 6(3):203–217, 2008.

[22] G. Sutcliffe and Y. Puzis. SRASS — A semantic relevance axiom selection system. In F. Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 2007.

[23] G. Sutcliffe, J. Urban, and S. Schulz, editors. *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories*, volume 257 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[24] J. Urban. MoMM - fast interreduction and retrieval in large libraries of formalized mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.

[25] J. Urban. Mptp 0.2: Design, implementation, and initial experiments. *J. Automated Reasoning*, 37(1-2):21–43, 2006.

[26] J. Urban, K. Hoder, and A. Voronkov. Evaluation of automated theorem proving on the Mizar mathematical library. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *ICMS*, volume 6327 of *Lecture Notes in Computer Science*, pages 155–166. Springer, 2010.

[27] J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskocil. MaLARea SG1–machine learner for automated reasoning with semantic guidance. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2008.

[28] J. Urban, Jirí Vyskocil, and Petr Stepánek. MaLeCoP: Machine learning connection prover. In K. Brünnler and G. Metcalfe, editors, *TABLEAUX*, volume 6793 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2011.

[29] J. Vyskocil, D. Stanovský, and J. Urban. Automated proof compression by invention of new definitions. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 447–462. Springer, 2010.

[30] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In R. A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.

Inconsistencies in the Process Specification Language (PSL)

Michael Beeson

San José State University
San Jose, CA
ProfBeeson@gmail.com

Jay Halcomb

H&S Information Systems
Guerneville, CA
jhalcomb@hsinfosystems.com

Wolfgang Mayer

University of South Australia
Adelaide, Australia
wolfgang.mayer@unisa.edu.au

Abstract

The Process Specification Language (PSL) [6] is a first-order logical theory designed to describe manufacturing or business processes and formalize reasoning about them. It has been developed by several authors over a period of years, yet it is inconsistent with the simplest axioms that preclude non-trivial models. We demonstrate several inconsistencies using an automated theorem prover and attempt to repair the inconsistencies. We conclude that even with our amendments, PSL with its infinite models remains inadequate to represent complex industrial processes. We propose an alternative axiomatization that admits finite models, which may be better suited to automated theorem provers and model finders than the current version of PSL.

1 Introduction

The *Process Specification Language* (PSL) [6] is an ontology designed to formalize reasoning about processes in first-order logic. The ontology has been developed by several authors over a decade or more and has become an ISO standard [1]. PSL is a modular theory where individual modules formalize specific aspects of processes and their activities. The full PSL consists of about a thousand axioms in 75 subtheories that build upon the central *PSL Outer Core* subtheory that includes about ninety axioms in seven modules.[1] Although PSL itself is a domain-independent ontology, it can be tailored to a specific application by adding domain-specific axioms. Statements constructed from predicates and terms defined in the Outer Core constrain the possible models of the theory and therefore can be used to characterize the valid processes and their execution sequences.

One of the underlying cornerstones of PSL is the idea that applications can be made interoperable by exchanging sentences formalized in PSL in order to share information about process specifications and concrete process executions. Formal theorem proving technology can then be employed to reason about processes and validate concrete executions in order to answer *Competency Questions* [9] of interest to the application domain. However, this requires a consistent theory.

The semantics of PSL are formulated using the notions and vocabulary of trees. These are Activity Trees, representing activities generically, and Occurrence Trees representing particular activity occurrences temporally, with activity trees intended to be subtrees of Occurrence Trees. However, as it stands no subset of PSL can consistently demonstrate the existence of any nontrivial Occurrence Trees. What is missing is a proof of consistency of PSL with simple axioms asserting the existence of a few activity occurrences. Without an intuitively comprehensible model, it will be difficult indeed to use PSL for actual process specifications.

Since its inception, PSL has undergone several revisions. As of April, 2011, the most recent version posted on the NIST website [6] is the May 2008 revision. We will refer to that version as PSL 2008. We also worked with a revision we call PSL 2010, supplied to us by Conrad Bock of NIST [7]. Unless stated otherwise, we use the term "PSL" to refer to PSL 2010.

In this paper we show that the version of PSL 2008 and its revision PSL 2010 contain flaws that lead to inconsistency even for the simplest process instances. We identify the responsible axioms and propose further revisions that resolve the inconsistencies. Although the resulting ontology can consistently

---

[1]The modules are: PSL Core, Subactivity Theory, Occurrence Tree Theory, Discrete State Theory, Atomic Activity Theory, Complex Activity Theory, and Complex Activity Occurrence Theory.

(a) Process Template   (b) PSL Occurrence Tree   (c) PSL Activity Trees
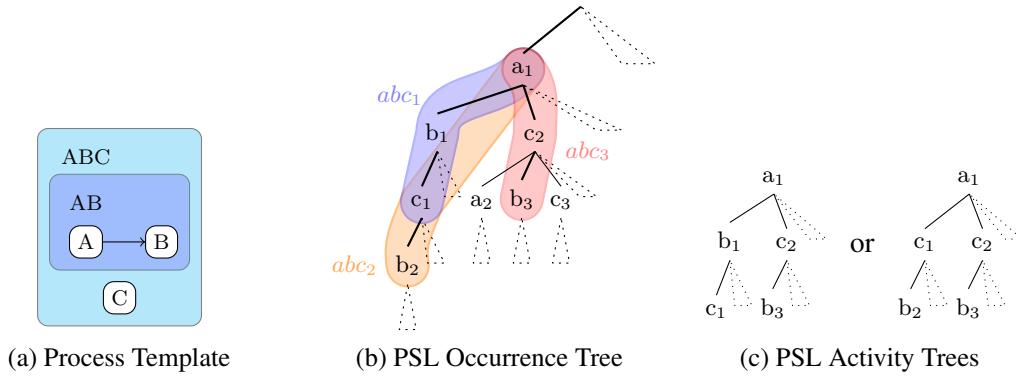
Figure 1: Example Process and PSL Trees

represent non-trivial processes, significant challenges remain to be addressed. In particular, it is difficult to obtain possible concrete (finite) models from the theory, whose axioms admit only infinite non-trivial models. We propose an alternative axiomatization that is based on Backofen et al.'s first-order theory of finite trees [2] that may be better suited to automated model construction.

First, we summarize the fundamental modeling primitives of PSL. Second, we describe inconsistencies we found in the course of our experiments and propose revisions to the axioms we believe are responsible for the inconsistency. Third, we outline where PSL may need to be strengthened in order to eliminate unwanted models. Fourth, we summarize our observations gathered from our experiments. Fifth, we introduce our Finite Tree PSL axiomatization before concluding the paper.

2   The PSL Outer Core Ontology

The PSL Outer Core formalizes basic properties of processes and the sequences of activities that may appear in a process execution, along with related objects and time points. The Outer Core is based on the following core concepts: *Activities* specify the actions and events that can appear in a process. Activities are partitioned into atomic activities and complex activities that consist of a number of sub-activities. An *Occurrence* of an activity represents an execution of the activity. Occurrences and activities are related via the binary *occurrence_of* predicate. PSL does not specify any concrete activities and occurrences; these must be introduced when tailoring the ontology to a specific application.

The *Occurrence Tree* contains all (infinite) sequences of occurrences of atomic activities starting from an initial state. This model is inspired by the Situation Calculus [10], where different branches in the situation tree represent alternative "worlds" where different sequences of actions apply. For each atomic activity X, the *successor* function maps each occurrence to its unique successor occurrence of X. Nodes in the tree are associated with timestamps and propositions that describe the process state before and after the occurrence. In addition, the occurrences in the branch of the tree must conform to the *earlier* predicate that embeds the sequence of occurrences in each branch into a totally ordered sequence of time points. One may be tempted to use *successor* or *earlier* to prune the Occurrence Tree; however, this may lead to inconsistency with the axioms of PSL. Instead, occurrences in branches that are admissible for a particular application domain are marked using the *legal* predicate.

Figure 1a shows a visual representation of a process specification where a composite activity *ABC* consists of activities *AB* and *C*, and activity *AB* consists of subactivities *A* and *B*. In any execution of *AB*, an occurrence of *A* must be followed by an occurrence of *B*. Within any occurrence of *ABC*, occurrences of *AB* and *C* may occur in any order. The relationship between complex activities and their subactivities is captured in the *subactivity* relation in PSL. Let activities *A*, *B*, and *C* be atomic, primitive activities that

have no subactivities. A PSL formalization in Prover9 syntax is available at [`ABC.in`].

The corresponding PSL Occurrence Tree is shown in Figure 1b. We use x_i to denote an occurrence of activity X, where i is an index that distinguishes different occurrences of an activity. Assume further that the activity occurrences $a_1, b_1, b_2, b_3, c_1, c_2$ are the only legal occurrences. Each node in the tree represents an occurrence of an atomic activity, and edges connect the occurrences to their direct successors. Bold edges represent *legal* branches, thin edges illegal branches, and dotted triangular edges denote subtrees that have been omitted for brevity (the tree has infinite depth and arbitrary branching factor). The tree contains two legal sequences of atomic activities: (a_1, b_1, c_1, b_2) and (a_1, c_2, b_3).

Occurrences of complex activities are defined by their atomic activities and are not part of the Occurrence Tree. Three complex activity occurrences $abc_{1,2,3}$ of activity ABC are drawn as hyperedges in Figure 1b. The complex occurrences of *AB* have been omitted. Occurrences of unrelated activities may appear in between activities of a complex occurrence. For example, b_1 is between a_1 and c_1 yet it does not belong to abc_2. The composition relationship between occurrences of complex activities and occurrences of their subactivities is formalized in the *subactivity_occurrence*, *root_occ* and *leaf_occ* predicates in PSL. For example, the complex activity abc_1 in Figure 1b is represented by the facts *subactivity_occurrence*(x, abc_1) for $x \in \{a_1, b_1, c_1, abc_1\}$, *root_occ*$(a_1, abc_1)$ and *leaf_occ*(c_1, abc_1).

Activity Trees for a complex activity specify the possible orderings of atomic activity occurrences within a complex activity. Each Activity Tree is characterized by the *min_precedes* partial order between its occurrences and its *root* and *leaf* occurrences (all axiomatized as predicates). Each complex activity occurrence corresponds to a branch in an activity tree, which orders its subactivity occurrences. By imposing constraints on the activity trees, the models of PSL can be restricted to the complex occurrences that are legal in a particular application domain. For example, the branches in the left activity tree in Figure 1c are represented by *root*$(a_1, ABC) \land min\_precedes(a_1, b_1, ABC) \land min\_precedes(b_1, c_1, ABC) \land$ *leaf*$(c_1, ABC) \land min\_precedes(a_1, c_2, ABC) \land min\_precedes(c_2, b_3, ABC) \land leaf(b_3, ABC)$.

A complex activity may have multiple activity trees—one for each occurrence in the Occurrence Tree that initiates a complex occurrence. Figure 1c shows two possible Activity Trees for our example. The tree on the left has branches for abc_1 and abc_3, whereas the right tree has abc_2 and abc_3. Note that the axioms of PSL prohibit us from constructing a single tree with branches for all three complex occurrences, as *leaf*(c_1, ABC) and *min_precedes*(c_1, b_2, ABC) are mutually incompatible.

3   Inconsistencies in the PSL Outer Core

In order for PSL to be used effectively, it must be consistent in a suitable sense. This "suitable sense" is not just the logical consistency of the pure theory. Because there are no axioms asserting the existence of any objects, activities, or activity occurrences, the entire theory is satisfied by the one-element model containing a single time point [`trivial.model`]. Although PSL 2010 without individual constants is consistent, this observation does not preclude the possibility that the theory becomes inconsistent as soon as one adds constants for specific activities, objects, and activity occurrences. If we add some constants (or, equivalently, add additional existential axioms for PSL predicates), then if all is well, we should expect the theory with those constants to have a "ground model", in which the elements of the model correspond to objects "constructed from the constants by the axioms", i.e. given by the values of terms built up from the constants and Skolem functions.

The third author first reported [11] that this was not the case for an earlier version of PSL 2008. There were several errors in the axioms that resulted in inconsistency as soon as there are two activity occurrences! Specifically, introducing three constants and the axiom *min_precedes*(b, c, a), he found contradictions using Prover9. He traced these problems ("after a tedious debugging process") to three axioms, and he suggested changes to these axioms, which would prevent the specific inconsistencies he

```
(forall (?a1 ?a2)
    (if (and (subactivity ?a1 ?a2)        (forall (?s1 ?s2)
             (not (atomic ?a2))                (if (earlier ?s1 ?s2)
             (not (= ?a1 ?a2)))                    (exists (?a ?s3)
        (not (exists (?s)                             (and (arboreal ?s3)
            (and (activity_occurrence ?s)                  (generator ?a)
                 (subtree ?s ?a2 ?a1)))))))             (= ?s2 (successor ?a ?s3)))))))
```

Figure 2: Mayer's revisions to Axiom 11 of the   Figure 3: Revised *successor* Axiom 11.
Complex Activity Theory.

discovered. Although some of the problematic axioms have been revised in PSL 2010, not all sources of inconsistency have been eliminated.

The first two authors conducted further experiments regarding the consistency of the PSL Outer Core. Unless stated otherwise, we used a simplified version of PSL 2010, where the time-point axioms and the *successor* axioms that require infinite models were removed, and Axiom 11 of the Complex Activity Tree was revised as shown in Figure 2. The input and output files are available at `http://dl.dropbox.com/u/20534396/papers/ATE2011-PSL/index.html` in Prover9, Mace4 or Paradox3 format. The resulting subtheory of PSL admits the construction of a finite non-trivial ground model. The theory and a Mace4 model file can be found at [`PSL_work.in`] and [`PSL_work.model`]. However, even with those revisions, we will show that PSL is still inconsistent when a few existential axioms or individual constants are added to the theory.

We also found PSL 2010 to be inconsistent with some very simple assumptions about two activity occurrences. Specifically, we assumed a scenario where the occurrence *a* is immediately followed by an occurrence *b* of activity *B* within the occurrence *ab* of a complex activity *AB* [`PSL.contra1.in`]:

$$activity(B) \land subactivity(B,AB) \land$$
$$activity\_occurrence(b) \land occurrence\_of(b,B) \land activity\_occurrence(a) \land occurrence\_of(a,A) \land$$
$$root\_occ(a,ab) \land occurrence\_of(ab,AB) \land earlier(a,b) \land next\_subocc(a,b,AB)$$

In the proof of inconsistency [`PSL.contra1.proof`], Axiom 11 of the Complex Activity Theory indeed fails because of the superfluous *not* and the missing hypothesis as described below. However, we have difficulties in understanding the intended meanings of both Axiom 8, which formalizes properties of root occurrences of atomic activities, and Axiom 11, and hence we cannot say definitely whether the proposed changes discussed below will resolve the problem in general.

We propose revisions to the theory that will eliminate the specific inconsistencies that we have seen. The revised PSL Outer Core theory in Prover9 format is available at [`PSL_revised.in`]. We have shown that this theory has non-trivial finite models if the axioms requiring infinite models are omitted [`PSL_revised_finite.in`][`PSL_revised_finite.model`]. However, we cannot claim that these revisions will eliminate all inconsistencies that may arise when posing different ground assumptions.

3.1   Complex Activity Theory Axiom 11

Prior to our modifications, Axiom 11 of the Theory of Complex Activities, which relates the activity tree of a complex occurrence to the subtrees corresponding to its subactivities, still has an incorrect negation and needed an extra hypothesis. Figure 2 shows the axiom and the proposed revisions. The formulas in Figure 2 and subsequent figures are written in KIF notation, which uses question marks to indicate variables and LISP-style parentheses. For example, $\exists x P(x)$ would be written as (exists (?x) (P ?x)).

Intuitively, the inconsistency arises because *subactivity* is reflexive (and can be used with atomic activities), but *subtree* is not (and its second argument must be a complex activity).

```
(forall (?a1 ?a2)
    (iff (subtree ?s1 ?a1 ?a2)
        (and (root ?s1 ?a1)
            (exists (?s2)
                (and (root ?s2 ?a2)
                    (or (= ?s1 ?s2)  (min_precedes ?s1 ?s2 ?a1))
                    (forall (?s3)
                        (if (min_precedes ?s2 ?s3 ?a2)
                            (min_precedes ?s2 ?s3 ?a1)))))))))
```

Figure 4: Revised *subtree* definition.

3.2   Complex Activity Theory Definition 4: Subtree

The contradiction found in the previous section can in part be attributed to the definition of $subtree(o,a_1,a_2)$. Contrary to most other predicates named *sub__* in PSL, *subtree* is irreflexive in the sense that $(\forall o,a)\neg subtree(o,a,a)$. This mix of reflexive and irreflexive predicates can easily lead to confusion, and missing hypotheses, as demonstrated earlier.

With the current definition one cannot establish that a concrete subtree actually exists in a given ground model. For example, if one asserts that *A* is a subactivity of *AB*, which is in turn a subactivity of *ABC*, and that occurrence a_1 of *A* is a root of *AB* and *ABC*, then $subtree(a_1,ABC,AB)$ is not entailed by PSL because $(\forall x,a)\neg min\_precedes(x,x,a)$ is a theorem of PSL.

Furthermore, the axiom is not strong enough to ensure that the subtree is indeed completely embedded into the activity tree of the superactivity. PSL has a model where both trees overlap at the root *r* of the subtree, yet not all occurrences in the subtree are also in the supertree: $(\exists x,a_1,a_2,r,y)subtree(x,a_1,a_2) \land min\_precedes(x,r,a_1) \land min\_precedes(r,y,a_2) \land \neg min\_precedes(x,y,a_1)$ is satisfiable [subtree_overlap.tptp][subtree_overlap.model]. This is at best counterintuitive.

We propose to revise this axiom and base the subtree property on the relationship between the *min_precedes* ordering in the sub- and supertree. Our revisions ensure that the activity tree of the superactivity embeds the tree of the subactivity. Figure 4 shows the new axiom. Our definition is reflexive such that $(\exists x,a)subtree(x,a,a)$ is satisfiable in PSL. The new *subtree* definition allows us to omit the extra hypotheses introduced earlier from Figure 2; the negation must still be removed.

However, even with these corrections, inspection of models of simple occurrence trees such as the overlap model above reveals that unintended elements are introduced into the models by PSL's *initial* predicate, which represents the first occurrence of an activity. This points to another critical incompleteness in PSL. *Generator* activities are those which have initial occurrences in the occurrence tree: $(\forall a)generator(a) \rightarrow (\exists s)initial(s) \land occurrence\_of(s,a)$. Introduction of these initial occurrences is not explicitly constrained in PSL by any definition, and is only loosely constrained by *earlier* relations and by the undefined PSL *successor* function. Moreover, the *earlier* relation, which orders primitive and complex occurrences in branches of the occurrence tree, seems unable to consistently express the notions of initial and final activity occurrences during finite temporal intervals, or of possibly unique occurrences, like the assassination of Abraham Lincoln. We return to this in discussing Finite Tree PSL.

3.3   Complex Activity Theory Definition 5: Sibling

PSL informally defines two subactivity occurrences s_1 and s_2 of a complex activity occurrence *A* to be *siblings* iff s_1 and s_2 either have the same immediate predecessor in the activity tree for *A*, or if both s_1 and s_2 are roots of activity trees for *A* and both are immediate successors of the same occurrence in the Occurrence Tree. For example b_1 and c_2 are siblings within *A* in Figure 1b. Intuitively one would expect that $sibling(s_1,s_2,A)$ implies that $s_1 \neq s_2$. However, the axioms allow that $sibling(s_1,s_1,A)$ holds.

The subexpression formalizing the *successor* constraint is incorrect: it admits models where s_1 and

```
(forall (?s1 ?s2 ?a)
  (iff (sibling ?s1 ?s2 ?a)
       (and (not (= ?s1 ?s2))
         (or (exists (?s3)
               (and (next_subocc ?s3 ?s1 ?a)
                    (next_subocc ?s3 ?s2 ?a)))
           (and
             (root ?s1 ?a)
             (root ?s2 a)
             (or (and (initial ?s1) (initial ?s2))
                 (exists (?s4 ?a1 ?a2)
                   (and
                     (= ?s1 (successor ?a1 ?s4))
                     (= ?s2 (successor ?a2 ?s4))
                     (arboreal ?s4)
                     (occurrence_of ?s1 ?a1)
                     (occurrence_of ?s2 ?a2)))))))))
```

Figure 5: Revised *sibling* definition.

```
(forall (?s1 ?s2 ?a)
  (iff (iso_occ ?s1 ?s2 ?a)
    (exists (?a1 ?a2 ?a3)
      (and (atomic ?a1) (atomic ?a2)
        (atomic ?a3)
        (subactivity ?a3 ?a)
        (occurrence_of ?s1 (conc ?a1 ?a3))
        (occurrence_of ?s2 (conc ?a2 ?a3))
        (all (?a4)
          (if
            (and
              (subactivity ?a4 (conc ?a1 ?a3))
              (subactivity ?a4 (conc ?a2 ?a3))
              (subactivity ?a4 ?a))
            (or (= ?a3 ?a4)
                (not (subactivity ?a3 ?a4)))
      )))))))
```

Figure 6: Revised *iso_occ* definition.

s_2 are equal to *successor*(x_i, y) where x_i and y are arbitrary terms and not necessarily activities and an activity occurrence, respectively. This stems from the axiomatization of the Occurrence Tree, where the arguments of the *successor* function are constrained only if the entire term *successor*(x, y) is known to be an occurrence of activity x. Therefore, by choosing x to be a term that does not represent an activity, the existential clause can always be satisfied. Figure 5 presents our attempt to resolve both problems.

3.4　Complex Activity Occurrence Theory Definition 1: iso_occ

Two occurrences s_1 and s_2 are said to be "occurrence isomorphic" with respect to a complex activity A if both occurrences contain a common subactivity of A. The axiom has recently been revised to restrict the common subactivity to be "maximal" (such that no common superactivity exists), yet the formalization implies a contradiction with the remaining axioms of PSL if the existence of two occurrence isomorphic activities is asserted [iso_occ.contra.in][iso_occ.contra.proof]. The deficiency in the formalization stems from overlooking that *subactivity* is reflexive. Figure 6 shows the repaired axiom. We are unsure whether the predicate is intended to be reflexive. If the relation should be irreflexive, an additional clause demanding that $s_1 \neq s_2$ must be added to the definiens. We have not verified if this extra assumption can be consistently made.

3.5　Complex Activity Occurrence Theory Definition 5: same_grove

Two occurrences of a complex activity are said to be in the "same grove" if they are in alternative branches in the Occurrence Tree. More formally, occurrences o_1 and o_2 of activity A satisfy *same_grove*(o_1, o_2, A) iff their root occurrences are siblings in the Occurrence Tree. The formal axiom does not make use of the *sibling* predicate defined earlier but duplicates part of it, including the problematic existential quantification discussed earlier. The axiom implies that either all complex activities share a common root occurrence, or no root of a complex occurrence may be a successor in the activity tree [same_grove.in][same_grove.cont.proof]. We propose to correct this problem by using the *sibling* predicate instead (see Figure 7). Curiously, this definition is almost identical to that in an earlier version of PSL presented by Bock and Grüninger [5].

4　Incompleteness of PSL Outer Core

Although PSL includes axioms that are too strong and results in inconsistency, certain axioms needed to exclude undesirable models are absent or not strong enough. In the following we identify some of these axioms. Because one cannot efficiently enumerate the possible models for given assumptions, we are far from claiming that our analysis has successfully identified all problematic axioms.

```
(forall (?o1 ?o2)                        (forall (?a ?b0 ?b1)
  (iff (same_grove ?o1 ?o2)                (if (and (atomic ?a)
       (exists (?a ?s1 ?s2)                         (atomic ?b0)
         (and (occurrence_of ?o1 ?a)                (atomic ?b1)
              (occurrence_of ?o2 ?a)               (subactivity ?a (conc ?b0 ?b1))
              (root_occ ?s1 ?o1)                   (not (primitive ?a)))
              (root_occ ?s2 ?o2)            (exists (?a0 ?a1)
              (or (= ?s1 ?s2)                     (and (subactivity ?a0 ?a ?b0)
                 (sibling ?s1 ?s2 ?a))))))          (subactivity ?a1 ?a ?b1)
                                                    (= ?a (conc ?a0 ?a1))))))))
```

Figure 7: Revised *same_grove* definition.   Figure 8: Revised distributivity axiom.

4.1 Atomic Activity Theory Axiom 8: Distributivity

As pointed out by Mayer [11], distributivity of the lattice of atomic activities is formalized incorrectly, in that the axiom is actually a tautology. Therefore, it will eliminate models that do not satisfy this property. The correct formalization of distributivity is given in Figure 8.

4.2 Occurrence Tree Axiom 11: successor

Axiom 11 of the Occurrence Tree Theory should express that every non-initial activity occurrence is the successor of another activity occurrence. Unfortunately, the formalization suffers from the same problem as described in Section 3.3, in that the arguments of the *successor* function are not sufficiently constrained. Our solution is to amend the axiom as shown in Figure 3. The definition of the *poss* predicate that represents the legal successor activities in the occurrence tree requires similar amendments.

5 Observations from Experiments

5.1 Scalability

The axiomatization of PSL is not particularly amenable to applying automated theorem proving technology. We have already pointed out that constructing models is hindered by having only infinite models. Although removing some of the axioms yields a theory that has finite models, findings such models remains difficult. Even with state of the art model finders like Mace4 and Paradox3, we could not consistently find models of 13 or more elements (within generous limits of 4GB memory and 12h CPU time). Furthermore, the resulting models may not satisfy all axioms of full PSL (for example, certain tree properties) and must be inspected and/or post-processed to ensure the results are meaningful. Finding proofs of contradiction is equally challenging, due to the large number of clauses generated from Skolem functions. We expect that considerable tuning and adaptation of PSL will be necessary if the theory is to be used for reasoning about non-trivial processes.

5.2 Compositionality

The treatment of composite activities, such as concurrent aggregation and complex activities, is not transparent in PSL and must be anticipated by the modeler.

Occurrences of concurrent activities must be considered explicitly when defining complex activities. As both the Activity Tree and the Occurrence Tree specify an ordering between *atomic* activities, and PSL distinguishes *conc* activities from *primitive* (i.e., non-concurrent) activities, concurrent occurrences must be allowed explicitly in the specification of complex activities. For example, asserting that $\neg((\exists s_1, s_2) occurrence\_of(s_1, A) \land occurrence\_of(s_2, B) \land min\_precedes(s_2, s_1, NotBA))$ is insufficient to ensure that no occurrence of B is before an occurrence of A within complex activity

NotBA. Instead, one must write $\neg((\exists s_1, s_2, x, y) atomic(x) \wedge atomic(y) \wedge occurrence\_of(s_1, conc(A, x)) \wedge occurrence\_of(s_2, conc(B, y)) \wedge min\_precedes(s_2, s_1, NotBA))$.

Similarly, one must anticipate which activities will eventually be expressed as complex activities. For example, PSL precludes us from writing

$$(\forall x, y, z) occurrence\_of(x, A) \wedge occurrence\_of(y, B) \wedge occurrence\_of(z, AB)$$
$$\wedge subactivity\_occurrence(x, z) \wedge subactivity\_occurrence(y, z) \rightarrow min\_precedes(x, y, AB)$$

to express that occurrences of *A* must precede occurrences of *B* within each occurrence of *AB* if *A* or *B* are complex activities. Instead, we must amend the antecedent with $leaf\_occ(u, x) \wedge root\_occ(v, y)$ and substitute *u* for *x* and *v* for *y* in the consequent. Furthermore, mixing complex activities with concurrent ones is not admissible in PSL. Writing such explicit specifications is error prone, and the resulting axioms may be difficult to read.

5.3   Complex Activity Occurrences and Fluents

PSL is tailored to expressing constraints on trees in order to define the intended process models for a particular application domain. Although the axioms allow one to impose constraints on complex activities to eliminate impossible occurrences, the axioms are not complete enough to prove that a complex activity actually occurred given a sequence of atomic activity occurrences that conform to the complex activity. This is for a number of reasons. First, PSL does not include *closure axioms* that state that no other activities and occurrences than those in the activity trees occur in a complex activity [5]. Second, the axioms do not entail that a complex activity occurred even if all the occurrences in a branch of an activity tree occurred. These must be added when the ontology is tailored to an application. We have not verified that this can be done consistently.

Moreover, the modeling of objects and state in PSL is incomplete. The first and second authors observed that the axioms pertaining to fluents and their propagation in the Occurrence Tree are not sufficient to handle composite expressions like conjunction and implication. It seems that a clear distinction between fluents and other objects in combination with a powerful *reflection principle* is needed to tackle these challenges. Detailed treatment of such extensions is beyond the scope of this paper.

5.4   Concurrency Model

PSL uses atomic activities formed from the *conc* constructor to represent the aggregate of multiple activities occurring concurrently. For example, $conc(a_1, a_2)$ represents an activity where subactivities a_1 and a_2 are executed concurrently. Occurrences of such aggregate activities can be part of the Occurrence Tree. This model can express some form of concurrency, however, it is limited in that the concurrent activities must have the same duration (because only the occurrence of the aggregate activity in the Occurrence Tree is associated with begin and end timestamps). Activities that run concurrently with a sequence of other activity occurrences cannot be expressed easily, and multiple concurrent instances of the same activity cannot be expressed [5].

PSL Outer Core lacks axioms that relate the properties of the individual activities to that of the concurrent aggregate activity. Suitable axioms must be added when activities are defined, and possible interferences between activities must be anticipated.

6   Finite tree and minimal PSL

The PSL documentation does not directly address questions arising from the differences between modeling finite and infinite occurrence trees, which is a distinction that cannot be captured in FOL; the compactness

theorem fails for the class of finite models of trees, and the problem of FOL validity in the class of all finite models is undecidable [12]. In fact, the class of valid FOL sentences over finite models is not recursively enumerable but is co-recursively enumerable. Therefore, any useful separation of infinite from finite trees will be impossible in the current PSL FOL theory. However, it is unclear what that augmentation might be. Finite model theory is an active field with many open questions, including some bearing directly upon the infamous P≠NP problem. Regular tree languages definable in *FO* and in *FOmod* (a language with counting quantifiers) are discussed by Benedikt and Segoufin [4]. Gradel et al. [8] discuss the more general problem of embedding finite models into "representations" of infinite databases. History and classical results about finite and infinite models are given in Baldwin [3].

After being informed of contradictions in PSL found the authors, Conrad Bock and Michael Grüninger defined a "minimal PSL", which they hoped would be free of contradictions. This is a collection of PSL axioms taken from the PSL Outer Core subtheories (specifically, from pslcore.th, complex.th, occtree.th and actocc.th), and some "lemmas" which are theorems of PSL. All axioms that would require infinite models are omitted. This subset can be found in [`psl-1016-618.in`]. We could verify using Mace4 that this subset does have finite non-trivial models where some activity occurrences exist. Unfortunately, this minimal version of PSL lacks substantive axioms bearing upon tree constructions. We came to the conclusion that, while the Occurrence Tree is fundamental to PSL's ontology, PSL itself does not contain enough axioms to reason about finite trees well (let alone to reason about infinite occurrence trees).

One possible way to improve PSL is to modify it so that occurrence and activity trees will be finite, and to supplement its other theories with appropriate axioms defined from the new tree primitive. If we modify PSL so that it has finite models, then occurrence and activity trees will be finite trees, which are more tractable than the infinite occurrence trees of PSL. A good theory of finite trees, which has the needed definitions, is presented in Backofen et al. [2]. This theory of finite trees has attractive properties: it is decidable and it permits the use of inductive schema.

We have translated the tree axioms of [2] into the language of PSL, calling it *FTPSL* and using *earlier* for the tree ordering. It was necessary to reverse the order of some of the FTPSL predicates as axiomatized by Backofen et al. in order to comply with the directionality of PSL axioms regarding the notion of *subactivity_occurrence*. Besides *earlier*, the primitives of FTPSL are the binary predicates *subactivity_occurrence*, *proper_subactivity_occurrence* and *immediate_subactivity_occurrence*. The translated FTPSL axioms are available at FTPSLRev1.in, and a model is at FTPSLRev1.model.

We are able to demonstrate, using Mace4, that FTPSL plus simple ground axioms constructs an appropriate occurrence tree for the seven occurrences of example MGM7, has finite models and is therefore consistent [FTPSLRev1_MGM7.in][FTPSLRev1_MGM7.model]. Adding the transcribed finite tree axioms to Bock and Grüninger's minimal PSL, we were able to find a finite model for that theory and establish consistency [FTPSLRev1OCCACTV1_PSL1016Lemmas.in] [FTPSLRev1OCCACTV1_PSL1016Lemmas.model]. However, inspection of the model reveals that activities and occurrences are not always desirably coordinated due to a lack of proper typing in the minimal PSL. We have not attempted to correct this, but by using FTPSL as a basis for defining other PSL notions, and by extending the theory to include trees for activities as well as occurrences, it may be possible to define the compositionality of successor axioms for each activity and for their occurrences (by defining, for each act, $successor(act, occ) = occ_1$). We have not yet investigated whether this approach can be extended to encompass an adequate treatment of fluents and the *holds* predicate as well.

7 Conclusions

We set out to test the hypotheses that (i) PSL is adequate for industrial process definition, and (ii) existing theorem provers are adequate to support reasoning about process descriptions in PSL. We found that PSL

is not adequate for industrial process description, for the following reasons:

- PSL contained, and after correction still contains, inconsistencies when simple ground axioms are added. This precludes the use of model finders and theorem provers.
- PSL lacks a good theory of finite trees, although trees are fundamental to PSL.
- PSL lacks a reflection principle permitting ordinary logical reasoning which coordinates activities and their occurrences, and hence does not allow one to infer what properties will hold after complex activities occur.
- PSL is not formulated in a way "friendly" to automated deduction; specifically with respect to its preference for the use of predicates instead of functions, insufficient "typing" axioms and no clear distinction between fluents and physical objects.

Our experiments with automated theorem provers helped us to discover these defects of PSL, and showed that the combination of PSL and existing theorem-proving technology is not adequate for industrial-strength process engineering. Certainly we saw many cases in which theorem provers could not reach the desired conclusion; some of those cases were due in our opinion to the inadequacies of PSL. Clearly, the problems which may lie with theorem proving technology regarding PSL (particularly scaling) cannot be assessed without first having a consistent process specification language, and in particular, a substantive theory of finite trees which represents compositionality of complex occurrences.

In our opinion, the reason for the enumerated troubles is that PSL does not have a clear "intended model". However, an effective revision to the theory must *start* with a clear informal description of such a model, and then include axioms that are true in that model and characterize it, so that the ground model of the theory plus simple axioms describing an activity should be unique, except possibly for the order of activity occurrences when those do not matter. Unfortunately, these goals cannot be accomplished by simple "surgical modifications" or additions to PSL, because the present version of PSL is incoherent, as we have demonstrated by finding inconsistencies for which there is no obvious repair. Our revisions in this paper "put some duct tape on PSL", but it is still broken.

Further issues remain to be addressed before any PSL or any finite version thereof can be considered ready for industrial use. Foremost, the consistency of the theory must be established for the possible ground models of interest. While we have shown that our small ground examples have models, formal analysis and verification must be carried out to lift this result to all "admissible" ground extensions of the theory. Further documentation is necessary to make the theory more accessible. We have spent many weeks trying to understand the intended meanings of the axioms, yet considerable uncertainties remain. A thorough discussion and examples showing the best practice use of the ontology would certainly help. Furthermore, a collection of Competency Questions [9] and other sentences that are supposed to be consistent with or entailed by the theory, as well as ones that should *not* be consistent or provable, would help in validating any changes made in the future. Currently, tailoring the theory to a particular application domain is a predominantly manual activity. Automated support in writing and validating additional axioms would certainly facilitate the adoption of PSL. Similarly, tailoring the ontology to suit a specific theorem proving technology would immensely benefit from automated supporting tools to analyze performance bottlenecks, rewrite axioms, and verify consistency and completeness of the resulting theory.

References

[1] TC 184/SC 4. *Process Specification Language*. ISO, Geneva, Switzerland, 2004.

[2] R. Backofen, J. Rogers, and K. Vijay-Shankar. A first-order axiomatization of the theory of finite trees. Technical Report IRCS-95-02, Institute for Research in Cognitive Science, University of Pennsylvania, 1995.

[3] J. Baldwin. Finite and infinite model theory — a historical perspective. *Logic Journal of IGPL*, 8(5):605–628, 2000.

[4] M. Benedikt, and L. Segoufin. Regular tree languages definable in FO and in FO mod. *ACM Transactions on Computational Logic (TOCL)*, 11(1):4, 2009.

[5] C. Bock, and M. Grüninger. PSL: A semantic domain for flow models. *Software and Systems Modeling*, 4(2):209–231, 2005.

[6] C. Bock, and M. Grüninger. PSL. `http://www.mel.nist.gov/psl/download.html`.

[7] C. Bock, and M. Grüninger. PSL outer core 2010. `http://sonic.net/~halcomb/papers/psl_outer_core.clf`. Translated to Prover9 automatically by R. Schulz: `http://sonic.net/~halcomb/papers/psl_outer_core-2010-axms-defs.in`.

[8] E. Gradel, P.G. Kolaitis, L. Libkin, M. Marx, J. Spencer, M.Y. Vardi, Y. Venema, and S. Weinstein. Finite Model Theory and Its Applications (Texts in Theoretical Computer Science. An EATCS Series), 2005.

[9] M. Katsumi, and M. Grüninger. Theorem proving in the ontology lifecycle. In *Knowledge Engineering and Ontology Design*, Valencia, Spain, 2010.

[10] H. J. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell.*, 2:159–178, 1998.

[11] W. Mayer. On the consistency of the PSL outer core ontology. Technical Report ACRC-KSE-080201, ACRC, University of South Australia, Mawson Lakes, Adelaide, Australia, Feb 2008. Circulated by email.

[12] B. Trahtenbrot. The impossibility of an algorithm for the decision problem for finite domains. In *Doklady Academii Nauk SSSR*, volume 70, pages 569–572, 1950.

Simplifying Pointer Kleene Algebra

Han-Hing Dang,[*]   Bernhard Möller

Institut für Informatik, Universität Augsburg
D-86135 Augsburg, Germany
{dang,moeller}@informatik.uni-augsburg.de

Abstract

Pointer Kleene algebra has proved to be a useful abstraction for reasoning about reachability properties and correctly deriving pointer algorithms. Unfortunately it comes with a complex set of operations and defining (in)equations which exacerbates its practicability with automated theorem proving systems but also its use by theory developers. Therefore we provide an easier access to this approach by simpler axioms and laws which also are more amenable to automatic theorem proving systems.

1   Introduction

Nowadays many first-order automated theorem proving systems are powerful enough and hence applicable to a large variety of verification tasks. Algebraic approaches especially have already been successfully treated by automatic reasoning in different application ranges [7, 8, 9, 16]. This work concentrates on an algebraic calculus for the derivation of abstract properties of pointer structures, namely pointer Kleene algebra [5]. This approach has proved to be an appropriate abstraction for reasoning about pointer structures and the deduction of various pointer algorithms [12, 13, 14]. Moreover the algebra enables by its (in)equation-based axioms the use of automated theorem provers for deriving and proving assertions about pointer structures, e.g., concerning reachability, allocation and destructive updates. When analysing reachability, projections are used to constrain the links of interest. For instance, in a doubly linked list the set of links in forward and backward direction each must be cycle-free, whereas, of course, the overall link set is cyclic. Unfortunately the existing pointer algebra turns out to be difficult to handle and to hinder automation. Therefore we provide a simplified version of pointer Kleene algebra which, additionally, is more suitable for automation.

The paper is organised as follows: In Section 2, we define a concrete graph-based model of pointer structures and illustrate the operations of pointer algebra in Section 3. Moreover some basics are given on which these specific operations build. Section 4 then gives more suitable axioms for automated deduction systems which are also easier to understand than the original theory. We conclude with an outlook on further applications of this approach in Section 5.

2   A Matrix Model of Pointer Kleene Algebra

We start by giving a concrete model of pointer Kleene algebra that shows how to treat various typical concepts of pointer structures algebraically. This model is based on matrices and represents the graph structure of storage with pointer links between records or objects. In [4] the model is also called a *fuzzy model* because it builds on some notions from the theory of fuzzy sets.

We assume a finite set L of edge labels of a graph with the known operations $\cup, \cap$. Moreover, let V be a finite set of vertices. Then the carrier set $\mathcal{P}(L)^{V \times V}$ of the model consists of matrices

---

with vertices in V as indices and subsets of L as entries. An entry $A(x,y) = M \subseteq L$ in a matrix A means that in the represented graph there are $|M|$ edges from vertex x to vertex y, labelled with the elements of M. The complete algebraic structure is $(\mathcal{P}(L)^{V \times V}, \cup, ; , 0, 1, \top)$ with greatest element $\top$ and operators, for arbitrary matrices $A, B \in \mathcal{P}(L)^{V \times V}$ and $x, y \in V$,

$$
\begin{aligned}
(A \cup B)(x,y) &=_{df} & A(x,y) \cup B(x,y) \ , \\
(A; B)(x,y) &=_{df} & \textstyle\bigcup_{z \in V}(A(x,z) \cap B(z,y)) \ , \\
\top(x,y) &=_{df} & L \ , \\
0(x,y) &=_{df} & \emptyset \ , \\
1(x,y) &=_{df} & \begin{cases} L & \text{if } x = y \ , \\ \emptyset & \text{otherwise} \ . \end{cases}
\end{aligned}
$$

As an example consider a tree structure with $L =_{df} \{\mathsf{left}, \mathsf{right}, \mathsf{val}\}$. Clearly, the labels left and right represent links to the left and right son (if present) of a node in a tree, respectively. The destination of the label val is interpreted as the value of such a node. We will use "label" and "link" as synonyms in the following. Figure 1 depicts a sample matrix and its corresponding labelled directed graph, i.e., a tree with $V =_{df} \{v_i, c_i : 1 \leq i \leq 3\}$.
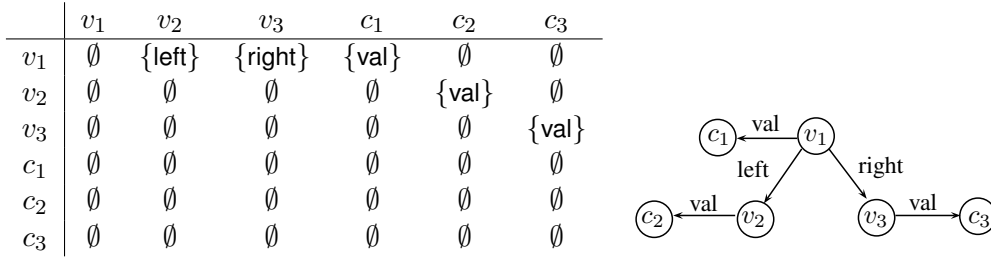

	v_1	v_2	v_3	c_1	c_2	c_3
v_1	$\emptyset$	$\{\mathsf{left}\}$	$\{\mathsf{right}\}$	$\{\mathsf{val}\}$	$\emptyset$	$\emptyset$
v_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{\mathsf{val}\}$	$\emptyset$
v_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{\mathsf{val}\}$
c_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
c_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
c_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 1: Example matrix and the corresponding labelled graph

Multiplication with the $\top$ matrix turns out to be quite useful for a number of issues. Left-multiplying a matrix A with $\top$ produces a matrix where each column contains the union of the labels in the corresponding column of A. Dually, right-multiplying a matrix A with $\top$ produces a matrix where each row contains the union of the labels in the corresponding row of A. Finally, multiplying A with $\top$ from both sides gives a constant matrix where each entry consists of the union of all labels occurring in A.

$$
\top \cdot \begin{pmatrix} L_{11} & \cdots & L_{1n} \\ \vdots & \cdots & \vdots \\ L_{n1} & \cdots & L_{nn} \end{pmatrix} = \begin{pmatrix} L_1 & \cdots & L_n \\ \vdots & \cdots & \vdots \\ L_1 & \cdots & L_n \end{pmatrix} \qquad \begin{pmatrix} L_{11} & \cdots & L_{1n} \\ \vdots & \cdots & \vdots \\ L_{n1} & \cdots & L_{nn} \end{pmatrix} \cdot \top = \begin{pmatrix} M_1 & \cdots & M_1 \\ \vdots & \cdots & \vdots \\ M_n & \cdots & M_n \end{pmatrix} \quad (1)
$$

where $L_i = \bigcup_{1 \leq j \leq n} L_{ij}$ and $M_j = \bigcup_{1 \leq i \leq n} L_{ij}$. This entails

$$
\top \cdot \begin{pmatrix} L_{11} & \cdots & L_{1n} \\ \vdots & \cdots & \vdots \\ L_{n1} & \cdots & L_{nn} \end{pmatrix} \cdot \top = \begin{pmatrix} M & \cdots & M \\ \vdots & \cdots & \vdots \\ M & \cdots & M \end{pmatrix} \quad (2)
$$

where $M = \bigcup_{1 \leq j \leq n} \bigcup_{1 \leq i \leq n} L_{ij}$. Such matrices can be used to represent sets of labels in the model.

3 Basics and the Original Approach

Abstracting from the matrix model of pointer Kleene algebra we now define some fundamental notions and explain special operations of the algebra. The basic algebraic structure of pointer Kleene algebra in the sense of [4] is a *quantale* [1, 15]. This is, first, an *idempotent semiring*, i.e., a structure $(S, +, \cdot, 0, 1)$, where $+$ abstracts $\cup$ and $\cdot$ abstracts $;$, and the following is required:

- $(S, +, 0)$ forms an idempotent commutative monoid and $(S, \cdot, 1)$ a monoid. This means, for arbitrary $x, y, z \in S$,

$$x + (y + z) = (x + y) + z \ , \quad x + y = y + x \ , \quad x + 0 = 0 \ , \quad x + x = x \ ,$$
$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \ , \qquad x \cdot 1 = x \ , \qquad 1 \cdot x = x \ .$$

- Moreover, $\cdot$ has to distribute over $+$ and 0 has to be a multiplicative annihilator, i.e., for arbitrary $x, y, z \in S$,

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z) \ , \quad (x + y) \cdot z = (x \cdot z) + (y \cdot z) \ ,$$
$$x \cdot 0 = 0 \ , \qquad\qquad\qquad 0 \cdot x = 0 \ .$$

The natural order $\leq$ on an idempotent semiring is defined by $x \leq y \Leftrightarrow_{df} x + y = y$. It is easily verified that the matrix model indeed forms an idempotent semiring in which the natural order coincides with pointwise inclusion of matrices.

Second, to form a quantale, $(S, \leq)$ has to be a complete lattice and multiplication $\cdot$ has to distribute over arbitrary joins. By this, $+$ coincides with the binary supremum operator $\sqcup$. The binary infimum operator is denoted by $\sqcap$; in the model it coincides with the pointwise $\cap$ of matrices. The greatest element of the quantale is denoted by $\top$.

This structure is now enhanced to a *Kleene algebra* [10] by an iteration operator *, axiomatised by the following unfold and induction laws:

$$1 + x \cdot x^* = x^* \ , \quad x \cdot y + z \leq y \Rightarrow x^* \cdot z \leq y \ ,$$
$$1 + x^* \cdot x = x^* \ , \quad y \cdot x + z \leq y \Rightarrow z \cdot x^* \leq y \ .$$

Besides this, the algebra considers special subidentities $p \leq 1$, called *tests*. Multiplication by a test therefore means restriction. Tests can also be seen as an algebraic counterpart of predicates and thus have to form a Boolean subalgebra. The defining property is therefore that a test must have a complement relative to 1, i.e., an element $\neg p$ that satisfies $p + \neg p = 1$ and $p \cdot \neg p = 0 = \neg p \cdot p$. In the matrix model tests are matrices that have non-empty entries at most on their main diagonal; multiplication of a matrix M with a test p means pointwise intersection of the rows or columns of M with the corresponding diagonal entry in p.

3.1 The Original Theory

An essential ingredient of pointer Kleene algebra is an operation that projects all entries of a given matrix to links of a subset $L' \subseteq L$. This is modelled by *scalars*[1] [4], which are tests $\alpha, \beta, \ldots$ that additionally commute with $\top$, i.e., $\alpha \cdot \top = \top \cdot \alpha$. Analogously to (1) and (2) these are diagonal matrices which are constant on the main diagonal. We will use the notation $L(\alpha)$ to denote the unique set of labels that a scalar α comes with. By this, a scalar α in the model corresponds to the matrix

$$\alpha(x, y) = \begin{cases} L(\alpha) & \text{if } x = y \ , \\ \emptyset & \text{otherwise} \ . \end{cases}$$

---

[1]The origin of this term lies in fuzzy relation theory and has similar behaviour as scalars in vector spaces.

It is immediate from the axioms that 0 and 1 are scalars.

Describing projection needs additional concepts. First, for arbitrary element x and scalar α,

$$\alpha\backslash x = x + \neg\alpha \cdot \top \ . \tag{3}$$

In the matrix model this adds the complement of $L(\alpha)$ to every entry in x. More abstractly, the operation is a special instance of the *left residual* defined by the Galois connection $x \leq y\backslash z \Leftrightarrow_{df} x \cdot y \leq z$; but this is of no further importance here.

The next concepts employed are the *completion* operator $\_^\uparrow$ and its dual $\_^\downarrow$, also known from the algebraic theory of fuzzy sets [18]. For arbitrary $x, y \in S$ they are axiomatised as follows.

$$
\begin{aligned}
x^\uparrow \leq y \Leftrightarrow x \leq y^\downarrow \ , && (x \cdot y^\downarrow)^\uparrow = x^\uparrow \cdot y^\downarrow \ , \\
\alpha \text{ is a scalar and } \alpha \neq 0 \text{ implies } \alpha^\uparrow = 1 \ , && (x^\downarrow \cdot y)^\uparrow = x^\downarrow \cdot y^\uparrow \ .
\end{aligned}
\tag{4}
$$

In the matrix model they can be described for a matrix M as follows

$$
M^\uparrow(x,y) = \begin{cases} L & \text{if } M(x,y) \neq \emptyset \ , \\ \emptyset & \text{otherwise} \ , \end{cases} \qquad
M^\downarrow(x,y) = \begin{cases} L & \text{if } M(x,y) = L \ , \\ \emptyset & \text{otherwise} \ . \end{cases}
$$

In both cases each node x is either totally linked or not linked at all with another node y. Such matrices containing only the entries $\emptyset$ and L are also called *crisp* [4]. They behave analogous to Boolean matrices where $\emptyset$ plays the role of 0 and L the one of 1. In the abstract algebra crisp elements are characterised by the equation $x^\uparrow = x$. In particular, $M^\uparrow$ maps a matrix M to the least crisp matrix containing it, while $M^\downarrow$ maps M to the greatest crisp matrix it contains.

Based on these operations and the particular elements of the algebra, *projections* $P_\alpha(\_)$ to label sets $L(\alpha)$ represented by a scalar α can be abstractly defined for arbitrary $x \in S$ by

$$P_\alpha(x) =_{df} \alpha \cdot (\alpha\backslash x)^\downarrow \ . \tag{5}$$

In the matrix model, projections w.r.t. scalars α are used to restrict each entry of a matrix exactly to $L(\alpha)$. As an example consider the resulting matrix of the following projection with $L(\alpha) = \{\mathsf{left}, \mathsf{right}\}$.

$$
P_{\{\mathsf{left},\mathsf{right}\}}\left(\begin{array}{cccc}
\emptyset & \{\mathsf{right}\} & \{\mathsf{left}\} & \{\mathsf{val}\} \\
\emptyset & \emptyset & \emptyset & \{\mathsf{left},\mathsf{right}\} \\
\{\mathsf{val}\} & \{\mathsf{left},\mathsf{right}\} & \emptyset & \emptyset \\
\emptyset & \{\mathsf{val}\} & \emptyset & \emptyset
\end{array}\right) = \left(\begin{array}{cccc}
\emptyset & \emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \{\mathsf{left},\mathsf{right}\} \\
\emptyset & \{\mathsf{left},\mathsf{right}\} & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \emptyset
\end{array}\right) .
$$

To see that this is achieved by Equation (5) consider first the term $\alpha\backslash x$ in the matrix model. This can be rewritten using Equation (3) into the term $x + \neg\alpha \cdot \top$. Hence residuals add to each entry of x all labels not in $L(\alpha)$. The operation $\_^\downarrow$ is then used to keep only those entries in $\alpha\backslash x$ that contain the full set of labels L while all other label sets will be mapped to $\emptyset$. Finally the multiplication with the scalar α again reduces all remaining L entries to $L(\alpha)$.

Finally we turn to the most important operator of pointer Kleene algebra. It calculates all reachable nodes from a given set of nodes. The definition uses domain and codomain operations $\ulcorner\_$ and $\_\urcorner$. They are characterised by the following equations, for arbitrary element x and test p,

$$x \leq \ulcorner x \cdot x \ , \qquad \ulcorner(p \cdot x) \leq p \ , \qquad x \leq x \cdot x\urcorner \ , \qquad (x \cdot p)\urcorner \leq p \ .$$

We note that a third pair of axioms, the modality laws $\ulcorner(x \cdot y) = \ulcorner(x \cdot \ulcorner y)$ and $(x \cdot y)\urcorner = (x\urcorner \cdot y)\urcorner$, is omitted here, because they are implied by another special property that holds for the matrix

model and also the algebra of binary relations. For domain, it reads $x \cdot \top = \ulcorner x \cdot \top$ and is called *subordination of domain*, since the direction $x \cdot \top \leq \ulcorner x \cdot \top$ follows from the above axioms, but the reverse one does not. This is equivalent [4] to $\ulcorner x = x \cdot \top \sqcap 1$, which we postulate as an additional axiom. This and the dual one for codomain then entail the above-mentioned modality laws.

Both operations are mappings from general elements to tests, i.e., the resulting matrices in the model are diagonal matrices. More concretely, the domain operation extracts for every vertex the set of labels on its outgoing edges while the codomain operator returns the labels on the incoming edges. As a simple example for domain consider

$$\ulcorner \begin{pmatrix} \emptyset & \{\mathsf{right}\} & \{\mathsf{left}\} \\ \emptyset & \emptyset & \{\mathsf{left, right}\} \\ \{\mathsf{val}\} & \{\mathsf{left, right}\} & \emptyset \end{pmatrix} = \begin{pmatrix} \{\mathsf{left, right}\} & \emptyset & \emptyset \\ \emptyset & \{\mathsf{left, right}\} & \emptyset \\ \emptyset & \emptyset & \{\mathsf{left, right, val}\} \end{pmatrix}.$$

In [4] reachability is now defined by a mapping that takes two arguments: One argument represents the set of starting nodes or addresses from which the reachable vertices are computed. The other argument represents the graph structure in which the reachability analysis takes place. Address sets are represented in this approach by crisp tests: an address belongs to the set represented by p iff the corresponding entry in the main diagonal of p is L. Now assume that m represents an address set. Then $reach(m, x)$ is defined using the iteration operator * by

$$reach(m, x) = (m \cdot (x^\uparrow)^*)^\urcorner .\tag{6}$$

The mapping *reach* calculates a test representing the set of vertices that are reachable from m using arbitrarily labelled graph links of x. This "forgetting" of the concrete label sets is modelled by completing the graph using the completion operator $\_^\uparrow$. Sample properties one wants to prove about *reach* are

$$reach(m + n, x) = reach(m, x) + reach(n, x) , \qquad reach(reach(m, x), x) = reach(m, x) .$$

To restrict reachability analysis to a subset of labels, projections are combined with *reach* into the mapping $reach(m, P_\alpha(x))$ for a scalar α. By this, non-α links are deleted from x.

Finally, an important operation in connection with pointer structures is *overriding* one structure x by another one y, denoted $y|x =_{df} y + \neg \ulcorner y \cdot x$. Here entries of x for which also y provides a definition are replaced by these and only the part of x not affected by y is preserved. This operator enjoys a rich set of derived algebraic properties. For instance, it is interesting to see in which way an overwriting affect reachability. One sample law for this is

$$\ulcorner y^\uparrow \sqcap reach(m, x) = 0 \;\Rightarrow\; reach(m, y|x) = reach(m, x) ,$$

i.e., when the domain of the overriding structure is not reachable from the overridden one it does not affect reachability.

3.2   A Discussion on Automation

One sees that it is very onerous to define the domain specific operations of pointer Kleene algebra from the basic operations. For example projections already include special subidentities of the algebra, residuals and completion and its dual, where each operator itself comes with lots of (in)equations defining behaviour. Furthermore by Equation (6) *reach* also uses crisp subidentities and moreover includes the $\_^\uparrow$, iteration * and the codomain operation. The inclusion of that many axioms often irritates the proof systems and additionally increases the search space.

A naive encoding of these operations in the first-order automated theorem proving system[2] Prover9 [11] already revealed that only a small set of basic properties for projections and reachability calculations could be derived automatically. We used this theorem prover since it performs best for automated reasoning tasks within the presented basic algebraic structures [2]. Moreover, it comes with the useful counterexample search engine Mace4 and outputs semi-readable proofs, which often provide helpful hints. Of course, any other first-order ATP system could also be used with the abstract algebraic approach, e.g., through the TPTP problem library [17].

The resulting ineffective automation could be due to the indirect axiomatisation of $\_^{\downarrow}$ through $\_^{\uparrow}$ by a Galois connection (cf. Definition (4)). In particular, deriving theorems for $\_^{\downarrow}$ will often require to show results for $\_^{\uparrow}$ and vice versa. Furthermore, encoding subtypes as tests and scalars by predicates may be another hindrance to a simple treatment of the axioms by ATP systems. Such an encoding is also inappropriate for ATP systems like Waldmeister [6] which works completely equation-based.

Moreover, using the Galois characterisation of residuals instead of the explicit characterisation in Equation (3) seems to additionally exacerbate the proof search. However, including that characterisation does not seem to simplify the proof search significantly. Therefore we also got a similar result with the application of ATP system when reasoning about restricted reachability.

Finally the given axiomatisations of the specific operators for this particular domain are also difficult to grasp and handle for theory developers due to their complexity. Therefore, in the next section we provide a simpler approach to pointer Kleene algebra which is easier to understand and more amenable to ATP systems.

4 A Simpler Theory for Pointer Kleene Algebra

The preceding section has shown that the original pointer algebra uses quite a number of concepts and ingredients. The present section is intended to show that one can do with a smaller toolbox which also is more amenable to automation. As a first simplification we drop the assumption that address sets need to be interpreted by crisp elements. Plain test elements also suffice to represent source nodes for $reach(p, x)$ since x is by definition already completed.

4.1 Projection

We continue with the notion of projecting a graph to a subgraph that is restricted to a set of labels. For this we first want to find representatives for sets of labels in our algebra.

It is clear that constant matrices and sets of labels are in one-to-one correspondence. By intersecting a constant matrix with the identity matrix one obtains a test which in the main diagonal contains the represented set M of labels, see (1). Multiplying another matrix A with this test from either side intersects all entries in A with M and hence projects A to the label set M. The considered test can also be got by taking the domain of the resulting matrix. Note that neither scalars nor residuals nor the operator $\_^{\downarrow}$ are involved here.

The difference of the just described way to project matrices and projections $P_\alpha(x)$ can be made clear in the example given after Equation (5). By our approach only the {val} entries of the original matrix will be deleted while single {left} and {right} entries remain. It is reasonable also to consider such entries in reachability calculations.

---

[2]We abbreviate the term "automated theorem proving system" to ATP system in the following.

4.2 Domain and Codomain

As a further simplification, plain test elements can be represented in the algebra by domain or codomain elements. In particular, such elements form Boolean subalgebras, resp. We give their axiomatisation through $a(x)$ and $ac(x)$ which are the complements of the domain and codomain of x, resp. From these domain and codomain can be retrieved as $\ulcorner x = a(a(x))$ and $\overline{x\urcorner} = ac(ac(x))$. The axioms read as follows:

$$a(x) \cdot x = 0 \;, \qquad\qquad x \cdot ac(x) = 0 \;,$$
$$a(a(x)) + a(x) = 1 \;, \qquad\qquad ac(ac(x)) + ac(x) = 1 \;,$$
$$a(x \cdot y) \leq a(x \cdot a(a(y))) \;, \qquad ac(x \cdot y) \leq ac(ac(ac(x)) \cdot y) \;.$$

The idea with this approach is to avoid explicit subsorting, i.e., introducing the set of tests as a sort of its own, say by using predicates that assert that certain elements are tests, and to characterise the tests implicitly as the images of the antidomain/anticodomain operators. The axioms entail that those images coincide and form a Boolean algebra with $+$ and $\cdot$ as join and meet, resp.

The given characterisation seems to be still difficult to handle for ATP systems in that form. Often a lot of standard Boolean algebra properties have to be derived first. Therefore we propose an equivalent but more "efficient" axiomatisation at the end of the next section.

4.3 Completion

Next, we turn to the completion operator $\_^{\uparrow}$ which is useful in analysing link-independent reachability in graphs. Rather than axiomatising it indirectly through a Galois connection we characterise it jointly with its complement $\_^{\overline{\uparrow}}$ similarly to domain and antidomain in Section 4.2. In particular, we axiomatise $x^{\overline{\uparrow}}$ as a left annihilator of x w.r.t. $\sqcap$, paralleling the statement that $a(x)$ is a left annihilator for x w.r.t. composition $\cdot$.

The axioms show some similarity to the domain/antidomain ones, but also substantial differences on which we will comment below. However, we still have, analogously to $\ulcorner x = a(a(x))$, that $x^{\uparrow} = (x^{\overline{\uparrow}})^{\overline{\uparrow}}$; for better readability we use this as an abbreviation in the axioms:

$$1^{\uparrow} \leq 1 \;, \qquad x^{\overline{\uparrow}} \sqcap x \leq 0 \;, \qquad \top \leq 0^{\overline{\uparrow}} \;,$$
$$x^{\overline{\uparrow}} \sqcap y^{\overline{\uparrow}} = (x+y)^{\overline{\uparrow}} \;, \qquad x^{\overline{\uparrow}} + y^{\overline{\uparrow}} = (x^{\uparrow} \sqcap y^{\uparrow})^{\overline{\uparrow}} \;,$$
$$(x \cdot y^{\uparrow})^{\uparrow} = x^{\uparrow} \cdot y^{\uparrow} \;, \qquad (x^{\uparrow} \cdot y)^{\uparrow} = x^{\uparrow} \cdot y^{\uparrow} \;.$$

Notice that the inequations can be strengthened to equations. The most striking difference to antidomain are the De-Morgan-like axioms and the axioms concerning multiplication. We cannot use the axiom $x \cdot y^{\uparrow} \leq (x \cdot y^{\uparrow})^{\uparrow}$ instead because it is not valid in the matrix model. Therefore the De Morgan laws do not follow but rather have to be put as additional axioms. They clearly state that the image of $\_^{\uparrow}$ is closed under $\sqcap$ and $+$.

We list a number of useful consequences of the axioms; they are all very quickly shown by Prover9. A sample input file can be found in the appendix.

$$x \leq x^{\uparrow} \;, \qquad\qquad \top^{\uparrow} = \top \;, \qquad\qquad 0^{\uparrow} = 0 \;,$$
$$(x^{\overline{\uparrow}})^{\uparrow} = x^{\overline{\uparrow}} \;, \qquad\qquad \top^{\overline{\uparrow}} = 0 \;, \qquad\qquad 0^{\overline{\uparrow}} = \top \;,$$
$$(x^{\overline{\uparrow}} + y^{\overline{\uparrow}})^{\overline{\uparrow}} = x^{\uparrow} \sqcap y^{\uparrow} \;, \qquad (x^{\overline{\uparrow}} \sqcap y^{\overline{\uparrow}})^{\overline{\uparrow}} = x^{\uparrow} + y^{\uparrow} \;,$$
$$(x^{\uparrow} + y^{\uparrow})^{\overline{\uparrow}} = x^{\overline{\uparrow}} \sqcap y^{\overline{\uparrow}} \;, \qquad (x^{\uparrow} \sqcap y^{\uparrow})^{\overline{\uparrow}} = x^{\overline{\uparrow}} + y^{\overline{\uparrow}} \;,$$
$$x \leq y \Rightarrow y^{\overline{\uparrow}} \leq x^{\overline{\uparrow}} \;, \qquad x \leq y \Rightarrow x^{\uparrow} \leq y^{\uparrow} \;,$$
$$x^{\uparrow} = 0 \Leftrightarrow x = 0 \;, \qquad x^{\overline{\uparrow}} = \top \Leftrightarrow x = 0 \;, \qquad x^{\uparrow} = 0 \Leftrightarrow x = 0 \;,$$
$$(x^{\uparrow})^{\uparrow} = x^{\uparrow} \;, \qquad (x^{\uparrow} \cdot y^{\uparrow})^{\uparrow} = x^{\uparrow} \cdot y^{\uparrow} \;, \qquad (x+y)^{\uparrow} = x^{\uparrow} + y^{\uparrow} \;.$$

In particular, the image of $\_^{\overline{\uparrow}}$ and hence of $\_^{\uparrow}$, i.e., the set of crisp elements, forms a Boolean algebra. Moreover, $\_^{\uparrow}$ is a closure operator. By general results therefore $\_^{\uparrow}$ preserves all existing suprema and, in a complete lattice, has an upper adjoint, which of course is $\_^{\downarrow}$. To axiomatise $\_^{\downarrow}$ we can use the Galois adjunction $x^{\uparrow} \leq y \Leftrightarrow x \leq y^{\downarrow}$. This entails standard laws as e.g.

$$(x \cdot y^{\downarrow})^{\uparrow} = x^{\uparrow} \cdot y^{\downarrow} \ , \quad (x^{\downarrow} \cdot y)^{\uparrow} = x^{\uparrow} \cdot y^{\downarrow} \ , \quad (x^{\downarrow})^{\uparrow} \leq x \ , \quad x \leq (x^{\uparrow})^{\downarrow} \ .$$

With the help of $\_^{\downarrow}$ one can show that the image of $\_^{\overline{\uparrow}}$ is also closed under the iteration *. A last speciality concerns the domain/codomain operation. By the subordination axiom for domain it can be verified that $\ulcorner(x^{\uparrow}) = (\ulcorner x)^{\uparrow}$.

Altogether we retrieve Ehm's result that, in a semiring with domain, the image of $\_^{\overline{\uparrow}}$, forms again a Kleene algebra with domain. Extending that, our axiomatisation yields that this algebra is even Boolean.

Inspired by the above axiomatisation, we now present a new axiomatisation of antidomain and anticodomain that explicitly states De-Morgan-like dualities to facilitate reasoning.

$$\begin{aligned}
a(x) \cdot x = 0 \ , \quad a(0) = 1 \ , && x \cdot ac(x) = 0 \ , \quad ac(0) = 1 \ , \\
a(x) \cdot a(y) = a(x + y) \ , && ac(x) \cdot ac(y) = ac(x + y) \ , \\
a(x) + a(y) = a(a(a(x)) \cdot a(a(y))) \ , && ac(x) + ac(y) = ac(ac(ac(x)) \cdot ac(ac(y))) \ , \\
a(x \cdot y) \leq a(x \cdot a(a(y))) \ , && ac(x \cdot y) \leq ac(ac(ac(x)) \cdot y) \ .
\end{aligned}$$

Using Prover9 we have shown that this axiomatisation is equivalent to the standard axiomatisation of antidomain/anticodomain. And indeed, the automatic proofs of the Boolean algebra properties of tests as well as of the mentioned properties of *reach* are much faster with it.

5   Outlook

This work provides a more suitable axiomatisation of special operations used in pointer Kleene algebra. These axioms are more applicable for ATP systems since less operations and less subtypes of the algebra has to be considered. Moreover the (in)equations of our approach enables a simpler encoding of the algebra for ATP systems due to explicit subsorting. This will also partially include the usage of ATP systems as Waldmeister for such particular problem domains.

It can be seen that the axiomatisations of antidomain (or anticodomain) and anticompletion are very similar. This motivates to further abstract from the concrete involved operations and to define a restriction algebra that replays general derivations.

Moreover, this approach will be used to characterise sharing and sharing patterns in pointer structures within an algebraic approach to separation logic [3]. This will include the ability to reason algebraically about reachability in abstractly defined data structures.

References

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

[2] H.-H. Dang and P. Höfner. First-order theorem prover evaluation w.r.t. relation- and Kleene algebra. In R. Berghammer, B. Möller, and G. Struth, editors, *Relations and Kleene Algebra in Computer Science — PhD Programme at RelMiCS 10/AKA 05*, number 2008-04 in Technical Report, pages 48–52. Institut für Informatik, Universität Augsburg, 2008.

[3] H.-H. Dang, P. Höfner, and B. Möller. Algebraic separation logic. *Journal of Logic and Algebraic Programming*, 80(6):221–247, 2011.

[4] T. Ehm. *The Kleene Algebra of Nested Pointer Structures: Theory and Applications*. PhD thesis, Fakultät für Angewandte Informatik, Universität Augsburg, 2003.

[5] T. Ehm. Pointer Kleene algebra. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 99–111. Springer, 2004.

[6] T. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. WALDMEISTER - High-Performance Equational Deduction. *Journal of Automated Reasoning*, 18:265–270, 1997.

[7] P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In F. Pfenning, editor, *Automated Deduction — CADE-21*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 279–294. Springer, 2007.

[8] P. Höfner and G. Struth. On automating the calculus of relations. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning (IJCAR 2008)*, volume 5159 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2008.

[9] P. Höfner, G. Struth, and G. Sutcliffe. Automated verification of refinement laws. *Annals of Mathematics and Artificial Intelligence*, 55:35–62, February 2009.

[10] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[11] W. McCune. Prover9 and Mace4.
<http://www.cs.unm.edu/∼mccune/prover9>. (accessed July 26, 2011).

[12] B. Möller. Some applications of pointer algebra. In M. Broy, editor, *Programming and Mathematical Method*, number 88 in NATO ASI Series, Series F: Computer and Systems Sciences, pages 123–155. Springer, 1992.

[13] B. Möller. Calculating with pointer structures. In *Proceedings of the IFIP TC2/WG 2.1 International Workshop on Algorithmic Languages and Calculi*, pages 24–48. Chapman & Hall, 1997.

[14] B. Möller. Calculating with acyclic and cyclic lists. *Information Sciences*, 119(3-4):135–154, 1999.

[15] K. I. Rosenthal. *Quantales and their Applications*, volume 234 of *Pitman Research Notes in Mathematics Series*. Longman Scientific & Technical, 1990.

[16] G. Struth. Reasoning automatically about termination and refinement. In S. Ranise, editor, *6th International Workshop on First-Order Theorem Proving*, volume Technical Report ULCS-07-018, Department of Computer Science, pages 36–51. University of Liverpool, 2007.

[17] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[18] M. Winter. A new algebraic approach to L-fuzzy relations convenient to study crispness. *Information Sciences*, 139(3-4):233–252, 2001.

A   Prover9 Encoding of Pointer Kleene Algebra

```
1  op(450, infix, "+").    % Addition
2  op(440, infix, ";").    % Multiplication
3  op(420, infix, "^").    % Meet
4  op(400, postfix, "*").  % Iteration
5  op(410, prefix, "@").   % Antidomain
6  op(410, prefix, "!").   % Anticodomain
7  op(410, prefix, "?").   % Anticompletion

8  % --- Additive commutative and idempotent monoid
9  x+(y+z) = (x+y)+z.
10 x+y = y+x.
11 x+0 = x.
12 x+x = x.
13 % --- Order
14 x <= y <-> x+y = y.
15 % --- Definition of top
16 x <= T.
```

```
17 % --- Multiplicative monoid
18 x;(y;z) = (x;y);z.
19 1;x = x.
20 x;1 = x.

21 % --- Distributivity laws
22 x;(y+z) = x;y + x;z.
23 (x+y);z = x;z + y;z.

24 % ---Annihilation
25 0;x = 0.
26 x;0 = 0.

27 % --- Definition of meet
28 (x<=y & x<=z) <-> x <= y^z.
29 x^(y+z) = x^y + x^z.

30 % --- Definition of domain
31 @0 = 1.
32 @x;x = 0.
33 @x;@y = @(x+y).
34 @x+@y = @(@@x;@@y).
35 @(x;y) = @(x;@@y).
36 @@x = (x;T) ^ 1. % --- Subordination of domain

37 % --- Definition of anticodomain
38 !0 = 1.
39 x;!x = 0.
40 !x;!y = !(x+y).
41 !x+!y = !(!!x;!!y).
42 !(x;y) = !(!!x;y).
43 !!x = (T;x) ^ 1. % --- Subordination of codomain

44 % --- Definition of completion
45 ??1 = 1.
46 ?0 = T.
47 ?x^x = 0.
48 ?x ^ ?y = ?(x + y).
49 ?x + ?y = ?(??x ^ ??y).
50 ??(x ; ??y) = ??x ; ??y.
51 ??(??x ; y) = ??x ; ??y.

52 % --- Iteration - Unfold laws
53 1 + x ; x* = x*.
54 1 + x* ; x = x*.
55 % --- Iteration - Induction laws
56 x;y + z <= y -> x* ; z <= y.
57 y;x + z <= y -> z ; x* <= y.

58 % --- Reachability
59 reach(x,y) = !!(@@x;(??y)*).
60 % --- Projection
61 P(x,y) = @@((T;x);T);y.
```

A Repository for Tarski-Kleene Algebras

Walter Guttmann

Universität Ulm, Germany

walter.guttmann@uni-ulm.de

Georg Struth

University of Sheffield, UK

g.struth@dcs.shef.ac.uk

Tjark Weber

University of Cambridge, UK

tw333@cl.cam.ac.uk

Abstract

We have implemented a repository of algebraic structures and theorems in the theorem proving environment Isabelle/HOL. It covers variants of Kleene algebras and relation algebras with many of their models. Most theorems have been obtained by automated theorem proving within Isabelle. Main purposes of the repository are the engineering of algebraic theories for computing systems and their application in formal program development. This paper describes the present state of the repository, illustrates its potential by a theory engineering and a program verification example, and discusses the most important directions for future work.

1 Introduction

Algebra has long been used for modelling and reasoning about computing systems. Examples are idempotent semirings in combinatorial optimisation, algorithm design and concurrency theory, lattices in domain theory, categories and allegories in functional programming, relations in program semantics, and fixpoint calculi in model checking.

Algebra supports abstraction by focusing on some crucial system features while disregarding others. It offers uniformity since diverse system models and semantics can often be captured by one single structure or minor variations. Its metatheory—universal algebra—allows structuring or combining algebras, decomposing systems and investigating their computational complexity. Last, but not least, algebraic reasoning is usually equational, hence ideally suited for automation.

Algebra has a long history in formal methods, too, in particular in software development, where programs or protocols are formally constructed from specifications. Back and von Wright's refinement calculus, for instance, is to a large extent algebraic. Jackson's Alloy method uses a relational modelling language. Abrial's B-Book contains long catalogues of algebraic laws for reasoning about programs. Hardly any formal method, however, relies on algebra alone. While 'point-free' algebraic techniques can, for instance, be very suitable for modelling a system's control flow, they need to be complemented by 'point-wise' logical techniques for the data flow. Similarly, abstract algebraic reasoning about a system often needs to be complemented by concrete properties of a particular model.

In program verification, data-flow reasoning often seems to dominate: with Hoare logic for while-programs, for instance, dispensing with the control structure is essentially automatic (one inference rule per program construct). Relating pre- and postconditions for the program store with respect to assignments is usually more involved. In program development or construction, the situation is different. Here, algebra can help to reduce non-determinism or preserve safety and termination conditions.

To support such applications in formal methods we have implemented a large repository for algebraic theories in Isabelle/HOL.[1] It currently contains more than 1000 lemmas and theorems. The algebras considered so far are variants of Kleene algebras [8, 17], as they arise in applications to processes, probabilistic systems, program refinement, relational program semantics and automata theory, their modal extensions [9], and reducts and expansions of Tarski's

---

[1]The repository is available at http://www.dcs.shef.ac.uk/~georg/isa/.

relation algebras [26]. Tony Hoare has proposed the name *Tarski-Kleene algebras* for this family of structures. Isabelle/HOL [23] is a theorem proving environment based on higher-order logic. It has recently been complemented by tactics for invoking external automated theorem proving systems (ATP systems) and satisfiability modulo theories solvers (SMT solvers) [6], and by counterexample generators [5].

Hierarchies of algebras can be developed in a modular way in Isabelle by using its axiomatic classes and locales; theorems can be inherited across hierarchies. Abstract algebraic structures can be linked formally with concrete models, for instance relation algebras with binary relations. This yields a seamless transition between point-free algebraic and point-wise logical reasoning. Calculational algebraic proofs can to a large extent be automated by ATP and SMT. The user can control the level of granularity of proofs and use Isabelle's proof scripting language Isar to present statements and proofs in a readable, publishable style. Proof search in Isabelle is greatly enhanced by a relevance filter, which selects hypotheses from a large set of internal libraries.

For a tutorial overview of the repository, including some simple theory formalisation and proof examples, see [10]. Some advanced modelling examples in (modal) Kleene algebras and an abstract formalisation of Hoare logic can be found in [12]. Complementing these two articles, this paper provides a more detailed description of the current structure and content of the repository. It also shows two new examples of automated theory engineering and program verification with Isabelle. Further, it discusses the role of the repository as the backbone of prospective proof environments which can be linked with existing formal methods, and various directions for future work. The repository is open to contributions of the formal methods community.

2   Automated Algebraic Theory Engineering in Isabelle/HOL

Isabelle/HOL [23] is one of the most widely used interactive theorem proving systems. Since its origins as a metalogical framework, there has been a strong focus on proof automation and applications in program analysis and verification. A recent breakthrough has been the integration of external ATP systems and SMT solvers via the Sledgehammer tactic [6]. State-of-the-art provers such as Vampire, E, SPASS and Z3 are called with a number of hypotheses selected by an internal relevance filter. In contrast to alternative tools such as PVS, these provers are not used as oracles. The internally verified ATP system Metis [16] or alternative methods are used to formally reconstruct proofs within Isabelle. This is desirable, since ATP systems and SMT solvers are complex software systems that depend on sophisticated heuristics. Compared to these, Isabelle's proof engine is very simple, easy to verify, and has stood the test of time. In addition, several counterexample generators have been added to Isabelle.

With this new integration at hand, users can benefit from the best of two worlds: the expressivity and versatility of interactive theorem provers, and the computational power of ATP systems, SMT solvers and counterexample generators.

It was already known that Tarski-Kleene algebras lend themselves very well to automated theorem proving, see [15] and references therein. But an implementation of our repository within Isabelle yields additional benefits:

Theory hierarchies: Isabelle's axiomatic classes and locales allow us to design and implement theory hierarchies for Tarski-Kleene algebras in modular ways, building on existing libraries for orders, semilattices and Boolean algebras. For instance, we have formally captured in Isabelle that relation algebras are expansions of Boolean algebras. Models of axiomatic structures can be obtained by instantiation. For example, we have proved that binary relations and formal languages are models of Kleene algebras.

Cross-theory reasoning: Theorems are automatically inherited across the hierarchy. All theorems about orders, for instance, are available automatically for semilattices and Boolean algebras; all theorems about relation algebras hold in the model of binary relations; all theorems of Kleene algebras hold in relation algebras expanded by an operation of reflexive-transitive closure. One particular algebraic axiomatisation can have a variety of computationally interesting models. Theorems proved at the abstract algebraic level are automatically available in all models: for instance, our theorems about Kleene algebras hold for binary relations and formal languages. In each particular model they can be augmented by domain-specific facts that will usually be proved by means of logic and set theory. In the relational semantics of imperative programs, for instance, abstract point-free algebraic facts can be combined with concrete point-wise reasoning about the store of a program and its updates.

Proof management: Isabelle ensures that only verified facts can be used as hypotheses in proofs. Moreover, with the Isar scripting language, the user owns the means of production: proofs can be either fully automated or refined into steps of arbitrary granularity. The proof of an equation $s = t$ in Boolean algebra, for instance, can be broken down into proofs of $s \leq t$ and $t \leq s$. State-of-the-art ATP systems and SMT solvers are powerful enough to prove calculational algebraic statements at textbook-level granularity. In calculational applications, Isabelle becomes almost entirely a proof manager for the external ATP systems.

Hypothesis learning: Isabelle provides a relevance filter that searches its internal libraries and selects hypotheses for individual proof goals. For small theory scopes this is surprisingly effective. In our case studies, proofs of moderate complexity could usually be fully automated by calling Sledgehammer. Structures such as modal Kleene algebras, where large numbers of lemmas are in the scope, seem to bring the relevance filter to its limits.

Theorems for free: Isabelle's higher-order features support more sophisticated forms of proof management which are based on symmetries, dualities and similar properties. In semi-groups with forward and backward modalities, for instance, there is a symmetry between these two kinds of operations that is expressed by swapping the order of multiplication. In Boolean algebra, there is a duality between the underlying join and meet semilattices. In relation algebra, theorems such as the modular laws can be obtained by instantiating more general theorems about Boolean algebras with operators. Operations that are shown to be adjoints of a Galois connection satisfy certain additivity, isotonicity or cancellation properties. All of these properties can be expressed and exploited in Isabelle, and are heavily used in engineering our repository.

Due to these features, the combination of algebra with automated proof search lends itself to the development of lightweight algebraic formal methods with heavyweight automation. Whereas previously a variety of different Isabelle tactics had to be mastered by users in order to make proofs succeed, the ATP/SMT integration largely simplifies this task to proof planning plus push-button proof search. The complementation of automated proof search with counterexample generators such as Nitpick and Quickcheck [5] allows a style of proof and refutation that is particularly beneficial for engineering new theories and debugging specifications.

3   Implementing Tarski-Kleene Algebras in Isabelle

'Tarski-Kleene algebras' loosely characterise a family of algebras based around Kleene and relation algebras. Kleene algebras were originally proposed by Kleene as algebras of regular expressions; more recently variants of Kleene algebras have been used for modelling program refinement [27] and probabilistic protocols [20]. Extensions cover infinite systems [7], modal reasoning similar to propositional dynamic logic [9], Hoare logic [21] and true concurrency [14].

Relation algebras have initially been conceived by Tarski as algebras of binary relations [26]. There is a longstanding history of using such structures for program semantics [4, 19] or as a basis for data refinement [24]. In the area of formal methods, relation algebras have been used for developing algorithms for graphs, orders or lattices from logical specifications [2, 3].

Kleene algebras and relation algebras share many properties, but there are also significant differences. Kleene algebras provide precisely the regular operations of addition (or union or join), multiplication and Kleene star, which in the context of programming can be interpreted as non-deterministic choice, sequential composition and finite iteration. Relation algebras lack the star operation, but provide operations for meet, complementation and converse besides addition and multiplication. Relation algebras have been designed with one particular model in mind, whereas Kleene algebras owe their fundamental status to the fact that they capture several important models of computation.

Our hierarchy connects Kleene algebras and relation algebras by expanding the latter with an operation of reflexive-transitive closure, as proposed by Ng and Tarski [22]. Then every expanded relation algebra is a Kleene algebra and the theorems for Kleene algebras are available in this setting. Similarly, we expand relation algebras by operations of domain and range, which project on the first and second coordinate of a binary relation, and link these operations with the modalities of modal Kleene algebras. In this context, every relation algebra thus expanded is a modal Kleene algebra and all theorems are again inherited.

In the context of program development, most of the theory hierarchy should be hidden behind an interface, providing developers with a simple relational specification language à la Alloy and access to Sledgehammer and Nitpick. From that side of the interface, the distinction between reasoning in relation algebra or Kleene algebra would vanish.

We now describe the theory hierarchy in more detail.

Dioids: Our hierarchy is based on classes for semilattices and variants of semirings. Near semirings (a generalisation of near rings) consist of an additive and a multiplicative semigroup that interact via a single distribution law; we also require that addition is commutative. Near dioids are obtained by making addition idempotent; this gives a semilattice structure with a canonical order (the refinement order in many models). By distributivity, multiplication is isotone in one argument. Adding isotonicity in the other argument gives predioids; requiring both distribution laws yields dioids (and semirings by omitting idempotency). Further variants are obtained by including a multiplicative or an additive unit. The latter is typically a left annihilator of multiplication, but in several models it is not a right annihilator; we account for this similarly to the omission of one of the distribution laws.

Kleene Algebras: All of the dioid variants are expanded by axioms for the Kleene star; they too come in left- and right-sided forms. These weaker Kleene algebras are important in applications involving demonic refinement algebra [27] or probabilistic Kleene algebra [20]. Interestingly, all the known equations of Kleene algebra could already be proved in the variant which omits the right induction axiom.

Omega Algebras: Supplementing the Kleene star operation for finite iteration, omega algebra introduces the omega operation for infinite iterations. This is useful, for instance, to model reactive, not necessarily terminating systems. Among the applications of this theory we provide, for example, highly automated proofs of loop refinement laws and termination theorems.

Domain Semirings: Semirings and dioids are expanded by a domain operation, which abstractly represents the set of states in which a computation is enabled. In particular, the image of the domain operation corresponds to the state space of a program; depending on the axioms it is a distributive lattice or a Boolean algebra [9]. Domain elements can serve as tests, for example, in preconditions and conditional statements.

Range Semirings: The range operation is obtained from domain by swapping the order of multiplication. The entire theory is obtained fully automatically by dualising domain semirings, using Isabelle's locale mechanism.

Modal Kleene Algebras: Domain and range give rise to forward and backward diamond and box operators. They abstractly represent states from which a computation may or must reach certain target states; in particular the forward box corresponds to the weakest liberal precondition. With the Kleene star operation we obtain Kleene algebra with tests [18] and, for applications in formal methods, a semantics for simple while-programs, and algebraic variants of propositional Hoare logic and propositional dynamic logic. Axiomatic algebraic approaches to temporal logics such as LTL and CTL can easily be developed from that basis.

Demonic Refinement Algebra: The axioms in our hierarchy cover related theories, such as von Wright's demonic refinement algebra [27] and the imperative fragment of the Unifying Theories of Programming [13]. So far we only have basic theorems for these; particular models and advanced results need to be added.

Concurrent Kleene Algebra: The development is discussed in more detail in Section 4.

Propositional Hoare Logic: A more basic setting (than modal Kleene algebra) suffices to encode this logic. Based on a Boolean algebra representing the state space, we directly axiomatise preconditions and while-programs; soundness and completeness of the Hoare rules are then derived automatically. This makes the calculus available for a wider range of models.

Boolean Algebra: Based on Huntington's minimalist axioms we have implemented Boolean algebra. This is useful because only few axioms have to be checked for instantiation, but it also yields an interesting test bed for ATP performance due to the difficulty of deriving the usual laws. We use the higher-order features of Isabelle to provide Boolean algebras with operators.

Relation Algebra: Expanding Boolean algebras with operations for composition and converse yields relation algebras. In particular, they are dioids, whence we automatically inherit the dioid theorems. We have added most of the ingredients for relational program development: subidentities and vectors for modelling sets, points, a calculus of functions, and domain and range operations. We have formally shown that relation algebras are domain and range semirings. Finally, we have expanded relation algebra by an operation of reflexive-transitive closure and shown that the resulting structure is a Kleene algebra.

We have formalised the most important models of these structures, for instance, binary relations, languages and program traces for Kleene algebras, omega algebras and Kleene algebras with domain. The formal relationship between the abstract algebras and the concrete models is established by using Isabelle's locale mechanism. An example is discussed in more detail in the next section.

4   Engineering Concurrent Semirings

This section illustrates theory engineering in the context of concurrent semirings and their models. Concurrent semirings have been proposed—under a different name—two decades ago as algebraic axiomatisations of series-parallel posets [11]. They have recently been studied as models for true concurrency based on a simple aggregation and independence model that is inspired by concurrent separation logic [14]. Here, we sketch the implementation of the abstract theory hierarchy from semigroups to concurrent semirings, and of their set-theoretic models based on notions of aggregation and independence. Due to lack of space we cannot show the Isabelle development; the complete code can be found in our repository.

We have first implemented the following algebraic hierarchy using Isabelle's axiomatic classes. An *ordered semigroup* is a structure $(S, \cdot, \leq)$ such that $(S, \cdot)$ is a semigroup, $(S, \leq)$ is a poset and $\cdot$ is isotone with respect to the order: $x \leq y$ implies $z \cdot x \leq z \cdot y$ and $x \cdot z \leq y \cdot z$. An *ordered monoid* $(S, \cdot, 1, \leq)$ is an ordered semigroup expansion such that $(S, \cdot, 1)$ is a monoid. In our setting, $\cdot$ can be interpreted as a form of sequential or serial composition of actions in S.

To model true concurrency we introduce a second operation $\otimes$ of concurrent or parallel composition. In contrast to process algebras such as CCS it is not necessarily related to interleaving. An *ordered bisemigroup* is a structure $(S, \cdot, \otimes, \leq)$ such that $(S, \cdot, \leq)$ is an ordered semigroup and $(S, \otimes, \leq)$ is an ordered commutative semigroup. In particular, $\otimes$ is also isotone. An *ordered bimonoid* $(S, \cdot, \otimes, 1, e, \leq)$ is the obvious expansion, where 1 is the unit of $\cdot$ and e that of $\otimes$.

Next we relate sequential and parallel composition. A *concurrent semigroup* is an ordered bisemigroup that satisfies the *multiplication inclusion law* $x \cdot y \leq x \otimes y$, the *small exchange laws* $(x \otimes y) \cdot z \leq x \otimes (y \cdot z)$ and $x \cdot (y \otimes z) \leq (x \cdot y) \otimes z$, and the *exchange law* $(w \otimes x) \cdot (y \otimes z) \leq (w \cdot y) \otimes (x \cdot z)$. A *concurrent monoid* is an ordered bimonoid that satisfies $1 = e$ and the exchange law. It can be shown that regular languages with shuffle and series-parallel posets satisfy the above laws (for example, $(\{a\} \otimes \{a\}) \cdot (\{b\} \otimes \{b\})$ contains less words than $(\{a\} \cdot \{b\}) \otimes (\{a\} \cdot \{b\})$), but our main justification comes from the model below. We have proved by ATP that the multiplication inclusion law and the small exchange laws are derivable in concurrent monoids, and, using Isabelle's counterexample generators, that none of the specific concurrent semigroup axioms hold already in ordered bisemigroups.

At the final stage of the abstract hierarchy we have implemented concurrent semirings. Formally, a *concurrent semiring* is a structure $(S, +, \cdot, \otimes, 0, 1)$ such that both $(S, +, \cdot, 0, 1)$ and $(S, +, \otimes, 0, 1)$ are idempotent semirings $(x + x = x)$, and $(S, \cdot, \otimes, 1, \leq)$ is a concurrent monoid, where $x \leq y$ if and only if $x + y = y$.

At the concrete set-theoretic level, we have implemented independence algebras over aggregation algebras. An *aggregation semigroup* is a semigroup $(A, \oplus)$ and an *aggregation monoid* a monoid $(A, \oplus, u)$. An *independence relation* is a bilinear binary relation R on an aggregation algebra: $R \ (x \oplus y) \ z \Leftrightarrow R \ x \ z \wedge R \ y \ z$ and $R \ x \ (y \oplus z) \Leftrightarrow R \ x \ y \wedge R \ x \ z$. In the monoidal case, R is also *bistrict*: $R \ u \ x$ and $R \ x \ u$. The idea is that $x \oplus y$ represents a system that consists of two parts x and y; u is the empty system. The linearity laws say that a compound system is independent from another system if and only if its parts are. The strictness laws say that the empty system is independent from any system. We use two independence relations R and S for sequential and concurrent composition and require that $S \ x \ y \Leftrightarrow S \ y \ x$ and $R \subseteq S$.

We have verified a number of properties by ATP that are useful for proving instances of the concurrent semirings and monoid axioms. The following law, for example, is used in the proof of the exchange law: $R \ (w \oplus x) \ (y \oplus z) \wedge S \ w \ x \wedge S \ y \ z \Rightarrow R \ w \ y \wedge R \ x \ z \wedge S \ (w \oplus y) \ (x \oplus z)$.

The last step for building models is to define operations on the powerset of the carrier of an aggregation algebra A. This is similar to lifting word to language products. We define the *complex product* $\circ_R : 2^A \times 2^A \to 2^A$ as $X \circ_R Y = \{x \oplus y \mid x \in X \wedge x \in Y \wedge R \ x \ y\}$. Since ATP systems are rather erratic on set expressions, we prove the property $z \in X \circ_R Y \Leftrightarrow \exists x, y : z = x \oplus y \wedge x \in X \wedge y \in Y \wedge R \ x \ y$ (and similarly for $\circ_S$). It can be used in combination with Isabelle's built-in laws for set extensionality and set inclusion to simplify to first-order expressions that ATP systems can handle.

Theorem proving at this level usually requires the application of Isabelle's simplifier with the mentioned rules, before calling Sledgehammer. We could then easily verify that the independence algebras under consideration form concurrent semigroups or concurrent monoids. Finally, with + interpreted as set union, we verified that independence algebras form concurrent semirings.

5   Verification of a Naive Reachability Algorithm

As a second example, we show the verification of a naive reachability algorithm [2] using the algebraic structures and lemmas available in our repository. This example is peculiar in that relations are not only used at the control flow level, but also, and primarily, as data structures that capture the digraphs or transition systems on which reachability is considered.

The algorithm is implemented in a simple imperative language with assignment, sequential composition and while-loops:

$x := V$; **while** $\neg (x \cdot Y \leq x)$ **do** $x := x + x \cdot Y$ **od**

The algorithm operates on a single variable x. First, x is initialised to V, a vector that represents initial states. Y is an adjacency matrix. The elements x, V and Y can be modelled by binary relations; we represent them more abstractly as elements of a Kleene algebra. Upon termination, x contains the relation $V \cdot Y^*$, that is, those states reachable from V via the reflexive-transitive closure of Y. Partial correctness is thus expressed by the following Hoare triple.

Theorem 1: vars x

   $\{\ True\ \} \ x := V$; **while** $\neg (x \cdot Y \leq x)$ **inv** $\{\ V \leq x \land x \leq V \cdot Y^* \ \}$ **do** $x := x + x \cdot Y$ **od** $\{\ x = V \cdot Y^* \ \}$

Here, we have additionally annotated the while-loop with its invariant, which captures the idea of the program: to compute intermediate relations x iteratively such that after each iteration, x is a superset of V and a subset of $V \cdot Y^*$. To prove this theorem, we rely on built-in automation in Isabelle/HOL that uses the invariant together with Hoare rules for assignment, sequential composition and while-loops to eliminate the algorithm's control structure [25]:

proof (vcg_simp)

After this simplification we are left with three automatically generated verification conditions. The first states that the precondition implies the loop invariant:

   show $V \leq V \cdot Y^*$

We invoke Sledgehammer with this subgoal. It calls 5 external ATP systems, all of which find a proof within a few seconds. They return the set of lemmas from our Kleene algebra repository used in the proof. For example, the prover E automatically generates the following command:

      by (metis mult_isol mult_oner star_ref)

This invokes Isabelle's built-in automation for first-order logic, Metis, to reconstruct the proof using three basic lemmas. Metis immediately succeeds and the first subgoal is thus proved. The second condition states that the invariant is preserved under execution of the loop's body:

   next fix x **show** $V \leq x \land x \leq V \cdot Y^* \land \neg (x \cdot Y \leq x) \Rightarrow V \leq x + x \cdot Y \land x + x \cdot Y \leq V \cdot Y^*$

We invoke Sledgehammer again. This time, only Vampire finds a proof within a few minutes. It uses 9 lemmas and neither Metis nor SMT are able to reconstruct a proof within Isabelle. Instead we proceed by proving one part of the condition from a reduced set of assumptions:

   next fix x **show** $x \leq V \cdot Y^* \Rightarrow x + x \cdot Y \leq V \cdot Y^*$
      by (metis add_lub leq_def mult_assoc mult_isol star_1r subdistr)

In a few seconds, Vampire returns with 6 lemmas, and Metis is able to reconstruct a proof. The second condition is completed by invoking Sledgehammer again. All provers and Metis succeed:

   thus $V \leq x \land x \leq V \cdot Y^* \land \neg (x \cdot Y \leq x) \Rightarrow V \leq x + x \cdot Y \land x + x \cdot Y \leq V \cdot Y^*$
      by (metis add_ub order_trans)

The final condition states that the loop invariant after termination implies the postcondition. Called from Sledgehammer, a proof is automatically produced by SPASS within a few seconds:

> **next fix** x **show** $V \leq x \wedge x \leq V \cdot Y^* \wedge x \cdot Y \leq x \Rightarrow x = V \cdot Y^*$
> **by** (metis add_lub le_neq_trans less_le_not_le star_inductr) **qed**

This completes the proof of Theorem 1. It is fully automatic except for the second verification condition, where Isabelle's proof reconstruction does not keep up with Vampire. This issue would vanish if the external prover returned a detailed proof that could be checked in Isabelle.

We formulated the reachability algorithm in terms of Kleene algebra operations. The proof of Theorem 1 only used axioms and lemmas of Kleene algebra. In our repository, we have shown that binary relations are a model of Kleene algebras. Isabelle/HOL, therefore, automatically generates an instance of Theorem 1 where all Kleene algebra operations have been replaced by the corresponding operations in the relational model: $\cdot$ by relational composition, $+$ by set union, * by the reflexive-transitive closure operation, and $\leq$ by set inclusion.

6 Future Directions

Our repository already contains a significant part of the calculus of variants of Kleene algebras and relation algebras. Extensions for domain-specific applications can be obtained with minor effort. In the context of program development, a large number of laws for dealing with the control structure of programs, as needed by Kleene algebra with tests, relational program semantics, Hoare logic, propositional dynamic logic or the w(l)p calculus, are present. Links with the data flow layer, for instance via the assignment rule of Hoare logic (as described in Section 5), are currently under construction. These will help transforming our repository into a program development and verification environment that could be adapted to support various existing formal methods and perhaps introduce a higher level of simplicity and automation.

We currently envisage the following main directions for future research and development.

Hypothesis learning: While Isabelle's relevance filter works impressively well on smaller theory scopes, learning hypotheses in large theories remains difficult. Our repository is very interesting in that respect since it yields a large benchmark suite of similar algebras, in which a similar kind of reasoning is required. It seems particularly useful to complement syntactic techniques, for instance, whether some term in a lemma matches some term in a proof goal, by domain-specific semantic techniques. For instance, a standard trick in ordered structures such as dioids or Boolean algebras is splitting the unit: $x = x \cdot 1 = x \cdot (x + x') = x \cdot x + x \cdot x' = x \cdot x$ proves idempotency of meet in Boolean algebra. How can such tricks be learned?

Solvers and decision procedures: Some fragments of Tarski-Kleene algebras are known to be decidable, for instance the equational theories of Kleene algebras, Kleene algebras with tests, (continuous) probabilistic Kleene algebras and concurrent semirings, but only the decision procedure for Kleene algebra is available in Isabelle. For many other fragments, decidability is not known. Sometimes, Horn formulas with antecedents of a particular shape can be reduced to equations. None of these hypothesis elimination algorithms are available in Isabelle, and for most variants of Tarski-Kleene algebras they have not been investigated. Integrating such algorithms could dramatically increase proof automation. In this context, decision procedures are typically based on automata, trees or graphs. Thus their output cannot be directly verified by Isabelle. Such procedures would have to be used as oracles or would have to be verified within Isabelle. Most of the proofs in the repository would only require small data structures in the decision procedures, hence even naive implementations would make a difference.

Integration of point-free and point-wise reasoning: The examples in Sections 4 and 5 suggest that these two styles can effectively be combined in our framework. In simple applications, entire verification tasks could probably be blasted away by SMT solvers. More generally, however, updates on program states must be modelled in a concrete semantics (for example, binary relations or predicate transformers) or the abstract algebraic layer must be augmented by rules for assignments and substitutions, as for instance in the B method. The development of specific lemmas and tactics that link the two layers is crucial for applications.

Design of simple modelling languages: The taxonomy of algebraic variants, their axioms and lemmas should, to a large extent, be hidden from the users. Instead simple modelling languages should be developed, for example, relational ones similar to that of Alloy. The underlying provers and counterexample generators could be used by developers to guide their semiformal understanding of a system's properties to be analysed.

Interfaces to formal methods: Due to their versatility, the structures and properties implemented in the repository are relevant to many applications. A prime example is relational program development for which a variety of tools with their own languages and idiosyncrasies exist. Many of these methods could be supported by creating interfaces to our repository.

7   Conclusion

We presented ongoing work on a repository for Tarski-Kleene algebras in Isabelle/HOL which is intended to provide automated proof support for program development methods. The development of the repository and its applicative potential depend strongly on the recent integration of ATP systems, SMT solvers and counterexample generators into Isabelle. Using this technology, new algebraic theories could be engineered quickly and easily, and a high degree of automation should be achievable in practical applications.

While the previous section contains a detailed discussion of ongoing and future work on this project, we conclude the paper with some remarks on automated theorem proving technology.

First of all, order-based reasoning is as important for program development as equational reasoning, for instance, in the context of refinement or when modelling simulation relations. Also reasoning in Tarski-Kleene algebras is, to a large extent, order based. Shifting between the two styles is possible, in principle, since $x = y$ if and only if $x \leq y$ and $y \leq x$, and $x \leq y$ if and only if $x + y = y$, but whereas splitting an equation into inequalities often simplifies proofs, the replacement of inequalities by equations blows up the size of terms and makes proof search more difficult. Ordered chaining calculi [1] have been developed to complement the superposition calculi used in many ATP systems in order to enhance order-based reasoning. But this technology has not been implemented in state-of-the-art tools.

Second, Isabelle's current integration uses only a handful of ATP systems and SMT solvers. Prover9, which on algebraic proof examples often shows the best overall performance [15], is not among them. Standardisation projects for ATP inputs (TPTP) and, in particular, proof output (TSTP) are important here. Via these interfaces, a large class of ATP systems could be accessed via Sutcliffe's System on TPTP. For SMT solvers, similar standards (SMT-LIB) exist.

For programming applications, sorts or types are very important. They are currently supported by only a few ATP systems. Although they can be encoded explicitly as constraints or guards to the algebraic specification, this can drastically slow down the proof search.

References

[1] L. Bachmair and H. Ganzinger. Ordered chaining calculi for first-order theories of transitive relations. *Journal of the ACM*, 45(6):1007–1049, 1998.

[2] R. Berghammer. Combining relational calculus and the Dijkstra–Gries method for deriving relational programs. *Information Sciences*, 119(3–4):155–171, 1999.

[3] R. Berghammer. Applying relation algebra and Rel View to solve problems on orders and lattices. *Acta Informatica*, 45(3):211–236, 2008.

[4] R. Berghammer and H. Zierer. Relational algebraic semantics of deterministic and nondeterministic programs. *Theoretical Computer Science*, 43:123–147, 1986.

[5] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In J. R. Cuellar and Z. Liu, editors, *SEFM 2004*, pages 230–239. IEEE Computer Society, 2004.

[6] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In *Automated Deduction: CADE-23*, 2011. To appear.

[7] E. Cohen. Separation and reduction. In R. Backhouse and J. N. Oliveira, editors, *MPC 2000*, volume 1837 of *LNCS*, pages 45–59. Springer, 2000.

[8] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.

[9] J. Desharnais and G. Struth. Internal axioms for domain semirings. *Science of Computer Prog.*, 76(3):181–203, 2011.

[10] S. Foster, G. Struth, and T. Weber. Automated engineering of relational and algebraic methods in Isabelle/HOL. In H. de Swart, editor, *RAMiCS*, volume 6663 of *LNCS*, pages 52–67. Springer, 2011.

[11] J. L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61(2–3):199–224, 1988.

[12] W. Guttmann, G. Struth, and T. Weber. Automating algebraic methods in Isabelle. In *Formal Methods and Software Engineering: ICFEM*, 2011. To appear.

[13] C. A. R. Hoare and J. He. *Unifying theories of programming*. Prentice Hall Europe, 1998.

[14] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene Algebra and its foundations. *Journal of Logic and Algebraic Programming*, 80(6):266–296, 2011.

[15] P. Höfner, G. Struth, and G. Sutcliffe. Automated verification of refinement laws. *Annals of Mathematics and Artificial Intelligence*, 55(1–2):35–62, 2009.

[16] J. Hurd. System description: The Metis proof tactic. In C. Benzmüller, J. Harrison, and C. Schürmann, editors, *ESHOL 2005*, pages 103–104, 2005.

[17] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[18] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19(3):427–443, 1997.

[19] R. D. Maddux. Relation-algebraic semantics. *Theoretical Computer Science*, 160(1–2):1–85, 1996.

[20] A. K. McIver and T. Weber. Towards automated proof support for probabilistic distributed systems. In G. Sutcliffe and A. Voronkov, editors, *LPAR*, volume 3835 of *LNCS*, pages 534–548. Springer, 2005.

[21] B. Möller and G. Struth. Algebras of modal operators and partial correctness. *Theoretical Computer Science*, 351(2):221–239, 2006.

[22] K. C. Ng. *Relation Algebras with Transitive Closure*. PhD thesis, Univ. of California, Berkeley, 1984.

[23] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[24] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[25] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, TU München, 2006.

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

[27] J. von Wright. Towards a refinement algebra. *Science of Computer Prog.*, 51(1–2):23–45, 2004.

Designing Domain Specific Languages for Verification: First Steps

Phillip James,   Markus Roggenbach

Swansea University, UK

`cspj@swansea.ac.uk`, `m.roggenbach@swan.ac.uk`

Abstract

This paper introduces a first approach at developing a design methodology for creating domain specific languages focused towards modelling and verification. The work presented is ongoing. The overall aim of the work is to show how capturing domain specific knowledge, and then tailoring proof goals around this domain specific knowledge, can improve automatic verification results, whilst also providing a graphical domain specific language.

1   Introduction

For many years, the application of verification processes such as model checking and interactive theorem proving to varying industrial case studies has been successfully illustrated, e. g. see [22, 19, 9, 10]. Even though these approaches have been successful, the use of formal methods within industry is often still limited. Without experts in the field of formal verification, the verification process is often complicated and hence simply takes too long for the everyday domain engineer to learn effectively. Domain specific languages [7] aim to abstract away such technical details from the domain engineer, allowing them to create programs or specifications without having to be an expert programmer or specifier.

Along with these problems, several research projects within the railway domain have shown that automatic verification can fail when domain models do not contain enough "domain knowledge" [6, 10, 11]. For example, in [10, 11], model checking approaches were applied to verify railway interlockings. The results of the verification were only partially successful, as many of the counter examples produced by the model checking process were later ruled to be impossible by domain experts. The problems encountered were due to underspecified programs created by the domain engineers.

This work, in co-operation with Invensys Rail, aims to show that following a particular design methodology for creating domain specific languages allows the creation of a graphical domain specific language that not only makes the task of automatic verification possible, but also less complex. Such gains are possible via carefully designing a domain specific language to ensure it captures domain knowledge relevant to the class of properties which one would like to verify.

2   The Railway Domain and DSLs

To illustrate our approach we use the railway domain. Here we review existing work in the area of verification within the railway domain and the area of domain specific language design.

2.1   Modelling and Verification in the Railway Domain

A prominent example of where formal methods have been applied is the railway domain. Approaches that have been taken include algebraic specification, e.g. [4], process algebraic modelling and verification, e.g. [22, 18], and model oriented specification, where, for example the B method has been used in order to verify part of the Paris Metro railway [5] in terms of both safety and

liveness properties. These approaches show the successful application of formal methods to the railway domain, but fail to comment on the applicability of such processes by domain engineers.

This work is inspired by the work of Bjørner [4]. To this end, we follow the natural language specification of the railway domain given by Bjørner [4]. Bjørner has also given a formal version of this natural language specification using the RSL specification language [20]. In contrast, we focus on using CASL, the Common Algebraic Specification Language [16] as it provides us with more features than RSL, including established tool support in the form of the Heterogeneous Toolset (Hets) [15]. The Hets environment not only provides both interactive and automatic theorem proving support, but also allows translation between different logics through institution maps [14]. Such a translation is shown to be useful in Section 3.

2.2 Domain Specific Language Design

The creation of domain specific languages is often aided by the use of a development framework. There are several examples of such tools including ASF+SDF [21] a meta-environment based on a combination of the algebraic specification formalism ASF and the syntax defining language SDF. ASF+SDF allows creation of domain specific languages and tools such as parsers, compilers and static analysers for the created domain specific language. Extending ASF+SDF, there is Rascal [12], which is currently under development at CWI. Finally, MetaEdit+ [2] is an industrial tool allowing the creation of visual domain specific languages. Interestingly, MetaEdit+ has been used to create a domain specific modelling for railway layouts, see [2].

With respect to our approach for creating domain specific languages, we make use of the *Graphical Modelling Framework*, GMF [8]. GMF is an Eclipse plugin that provides the infrastructure to create, from a UML like model, a Java based graphical editor. This editor can then easily be extended with Java code allowing it to output CASL specifications. The simplicity of this creation process fits well with our design methodology outlined in Section 3.

3 Towards a Design Methodology

In this section, we outline a first proposal for a design methodology for creating domain specific languages for verification. Figure 1 illustrates the proposed design and verification process.

Capturing knowledge: The first area that is illustrated in the left of Figure 1 is the capture of domain knowledge. A natural language specification can be formalised using the OWL Ontology language [3]. OWL has been designed to formalise knowledge about a given domain and thus provides a range of constructs to allow the capture of domain knowledge. It allows specification of *concepts* within a given domain via *classes*, specification of *attributes* of the concepts via *data properties* and specification of *relations* between concepts via *object properties*. It also allows axioms to be stated over such properties. These constructs are very similar to those within UML or any object orientated language. OWL has a well defined formal semantics [17] meaning that every OWL specification has a precise and unambiguous meaning. As we wish to use only automatic tool support, we make use of a decidable fragment of full OWL known as OWL-DL [3]

Creating the DSL: Given an OWL specification, an automatic translation to a GMF (UML like) meta-model is possible.[1] This meta-model along with a set of graphical elements can be used within the GMF process to create a graphical editing tool for the domain. The production

---

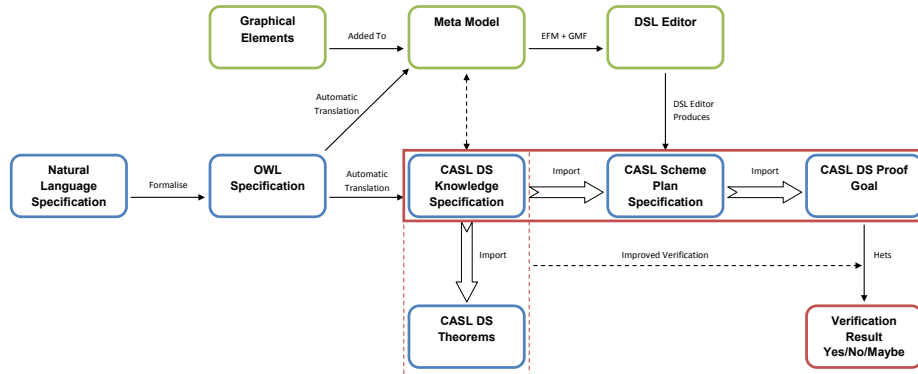[1]We are currently implementing this XML based translation.

Figure 1: A first design methodology for creation of DSLs focusing on verification.

of this graphical editing tool is outlined in the top branch of Figure 1. Here we provide an overview of the steps involved in the GMF process and for more details we refer the reader to [8]. The first step within the GMF creation process is to select the concepts of the domain which should become graphical constructs within the language. These graphical constructs can be split into two main classes essentially representing nodes and edges within the final graphical editor. The next step is to associate with each chosen construct for the language, a graphical image to represent it. Finally, the attributes (or properties for a given concept) which should be attached to each graphical element can be selected. Once these steps have been completed, the GMF tool will automatically produce a Java based graphical editor. This editor consists of a drawing canvas and a palette. Graphical elements from the palette can be dragged and positioned onto the drawing canvas. Along with these features, the Java code base for the editor is readily extensible and we use this fact to extend the editor to produce CASL specifications. Namely, we add a small amount of code for each construct that simple produces a CASL specification for that construct when it is added to the drawing canvas. Obtaining such a CASL specification for each construct is discuss below.

Semantics: To provide a semantics for the graphical editing tool we propose the use of CASL [16]. The main motivation for the use of CASL is thanks to the tool support that is available in the form of the Hets environment [15]. Hets not only provides syntax checking and static analysis of CASL specifications, but also an interface to various interactive and automated theorem provers. The central path of Figure 1 illustrates the addition of CASL to the graphical editing tool as a semantic base. Within Hets, an automatic semantic preserving translation from OWL into CASL has been implemented [13]. The motivation for using OWL and translating to CASL, rather than directly using CASL, is that OWL provides constructs suited towards capturing domain knowledge in a UML style which can be easily adopted by most domain engineers. Using the resulting CASL domain knowledge specification, the graphical editing tool can be extended to produce CASL specifications for domain models created using the editor.

Verification: Finally, verification of the CASL specifications produced by the graphical editing tool is possible using the Hets framework. At this point, Figure 1 highlights the advantage of adding domain knowledge to the verification process. That is, there are two possibilities for verification. The first – illustrated by the solid lined box – is simply verifying the given problem without any domain specific theorems on the domain knowledge level. The second – illustrated by the dotted lined box – includes domain specific theorems that have been proven on the

domain knowledge level. These theorems provide a potential gain for automated verification: (1) They have the potential to remove false counter examples like those experienced in [10]; (2) They allow general domain specific theorems to be added, these in turn improve the speed of the proof process for particular domain specific proof goals.

4   A First Example: Domain Knowledge Helps

To illustrate the advantages that can be gained through adding domain knowledge to the verification process, we study a common installation within the railway domain. Figure 2 shows part of a standard "double lead" junction.
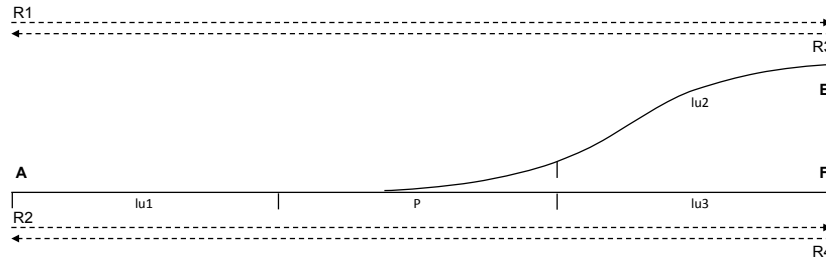

Figure 2: A typical railway junction.

Trains can travel from location A to locations E or F, or from locations E or F to location A. The path along which a train will travel is determined by the position of point P. Logically, the junction is segmented into routes. Here there are four possible routes. Route $R1$ can be set, i.e. trains can travel from A to E when the point is in "reverse" position and there are no trains occupying the point and track segments $lu1$ and $lu2$. Route $R2$ can be set, i.e. trains can travel from A to F when the point is set in "normal" position and there are no trains occupying the point and track segments $lu1$ and $lu3$. In a similar manner, routes $R3$ and $R4$ can be set to allow trains to travel from E to A and F to A respectively.

```
spec Junction [op p: Switch; op lu1: Linear; ... ] =
    %% axioms for connecting components such as points and tracks
    ...
    forall t: Time . point_EnabledReverseAndLu2At n if p stateAt n = unocc /\
        p positionAt n = reverse /\
        (exists t :Time . n < t /\ lu2 stateAt t = unocc);
    ...
    forall n: Time . route1_enabledAt n  if lu1 stateAt n = unocc /\
        (exists t : Time . n < t /\ point_EnabledReverseAndLu2At t);
    ...
then %implies
    ...
    forall n : Time . exists t: Time . n < t /\ route1_enabledAt t   %(Thm1)%
    forall n : Time . exists t: Time . n < t /\ route2_enabledAt t   %(Thm2)%
    forall n : Time . exists t: Time . n < t /\ route3_enabledAt t   %(Thm3)%
    forall n : Time . exists t: Time . n < t /\ route4_enabledAt t   %(Thm4)%
end
```

Figure 3: A parametrised specification of a junction in CASL.

As such a junction is a common installation within the railway domain, it would naturally form a concept or class within an OWL specification for the railway domain, e.g see [1]. Within CASL, it makes sense to capture a junction with parametrisation. Part of the parametrised CASL specification for the junction is given in Figure 3.

The junction specification illustrates the use of domain specific theorems. These theorems capture domain knowledge about the junction. *Thm1* expresses that there always exists a time in the future where route $R1$ is enabled, and similarly *Thm2*, *Thm3* and *Thm4* expresses this for routes $R2$, $R3$ and $R4$ respectively. Here, due to space constraints, we omit the behaviour of trains and points and assume they behave as expected. These theorems are provable using the Hets toolset in a few seconds.[2] Via instantiation of the junction specification, we can now specify the example train station in Figure 4. This station consists of six junctions in total.
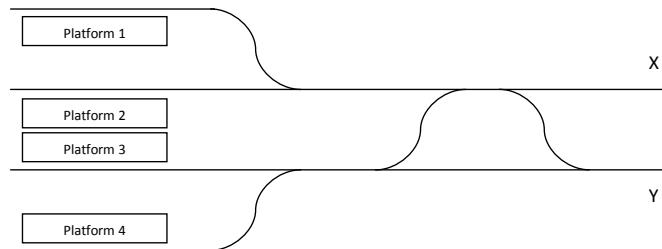

Figure 4: A track plan for an example station.

Over this new track plan for a station, we would like to reason about the enabling of routes allowing trains to enter or leave the station. For example we may wish to know that there always exists a time in the future when a train can leave *Platform 1* and travel to X. This condition is dependent on the setting of several routes across junctions. This property can be expressed as:

$$\forall \ n : Time \bullet \exists \ t1,\ t2,\ t3 : Time \bullet n < t1 \ \wedge \ t1 < t2 \ \wedge \ t2 < t3 \ \wedge$$
$$route3\_1\_enabledAt \ t1 \ \wedge \ route4\_3\_enabledAt \ t2 \ \wedge \ route2\_5\_enabledAt \ t3$$

Referring to Figure 1, if we try to verify such a condition without adding domain specific theorems, i.e. *Thm1* through to *Thm4*, to the verification process, then verification with Hets is not possible.[3] When adding the domain specific theorems into the process, the verification is possible within ten seconds. This illustrates that exploiting domain specific knowledge of particular domain constructs can aid the verification process considerably.

5   Summary and Future Work

In this paper, we have briefly introduced a first attempt at a design methodology for creating domain specific languages focused towards verification. We have also illustrated how the capture and exploitation of domain specific knowledge obtained via this design methodology can provide gains within the automatic verification process. As future work, we wish to explore further examples of how domain specific knowledge can be advantageous. The result will be a classification of types of knowledge and the benefits they can bring to the verification process. We also wish to explore providing useful feedback to domain engineers when a prove attempt is not successful. That is, we wish to explore the production of counter-examples on the level of the graphical editing tool.

---

[2]Verification times are only rough guidelines and not exact scientific benchmarks.

[3]Within fifteen minutes.

References

[1] Invensys Rail Data Model – Version 1A, 2010.

[2] MetaEdit+, Webpage, last accessed April 2011. `http://www.metacase.com/`.

[3] G. Antoniou and F. Harmelen. Web ontology language: Owl. *Handbook on ontologies*, 2009.

[4] D. Bjørner. *Domain Engineering – Technology Management, Research and Engineering*. JAIST Press, 2009.

[5] J. Boulanger and M. Gallardo. Validation and verification of METEOR safety software. In J. Allen, R. J. Hill, C. A. Brebbia, G. Sciutto, and S. Sone, editors, *Computers in Railways VII*, volume 7. WIT Press, 2000.

[6] W. Fokkink and P. Hollingshead. Verification of interlockings: from control tables to ladder logic diagrams. In J. Groote, S. Luttik, and J. V. Wamel, editors, *FMICS'98, Formal Methods for Industrial Critical Systems*. CWI, 1998.

[7] M. Fowler and R. Parsons. *Domain Specific Languages*. Addison-Wesley, 2010.

[8] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

[9] G. Holland, T. Kahsai, M. Roggenbach, and B.-H. Schlingloff. Towards formal testing of jet engine rolls-royce BR725. In L. Czaja and M. Szczuka, editors, *Proc. 18th Int. Conf on Concurrency, Specification and Programming, Krakow, Poland*, 2009.

[10] P. James. SAT-based Model Checking and its applications to Train Control Software. Master's thesis, Swansea University, 2010.

[11] P. James and M. Roggenbach. SAT-based Model Checking of Train Control Systems. Technical report, CALCO-jnr'09, University of Udine, n.5-2010, 2009.

[12] P. Klint, T. Van Der Storm, and J. Vinju. EASY Meta-programming with Rascal. *Generative and Transformational Techniques in Software Engineering III*, 2011.

[13] O. Kutz, D. Lücke, T. Mossakowski, and I. Normann. The OWL in the CASL - Designing Ontologies Across Logics. In C. Dolbear, A. Ruttenberg, and U. Sattler, editors, *OWLED*. CEUR-WS.org, 2008.

[14] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286(2), 2002.

[15] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, Hets. *Tools and Algorithms for the Construction and Analysis of Systems*, 4424, 2007.

[16] P. D. Mosses, editor. *CASL Reference Manual*, volume 2960. Springer, 2004.

[17] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. Owl web ontology language semantics and abstract syntax. Technical report, W3C, 2004.

[18] J. Peleska, D. Große, A. E. Haxthausen, and R. Drechsler. Automated verification for train control systems. In E. Schnieder and G. Tarnai, editors, *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems*. Technical University of Braunschweig, 2004.

[19] A. Simpson. A formal specification of an automatic train protection system. In G. Goos, J. Hartmanis, and J. V. Leeuwen, editors, *FME '94: Proceedings of the Second International Symposium of Formal Methods Europe on Industrial Benefit of Formal Methods*. Springer, 1994.

[20] The RAISE Language Group. *The RAISE specification language*. Prentice Hall, 1993.

[21] M. Van Den Brand, A. Van Deursen, J. Heering, H. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, and J. Scheerder. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. *LNCS*, 2027, 2001.

[22] K. Winter. Model checking railway interlocking systems. *Australian Computer Science Communications*, 24(1), 2002.

A Domain-Specific Language for the Specification of Path Algebras

Vilius Naudžiūnas

Computer Laboratory
University of Cambridge
Vilius.Naudziunas@cl.cam.ac.uk

Timothy G. Griffin

Computer Laboratory
University of Cambridge
Timothy.Griffin@cl.cam.ac.uk

Abstract

Path algebras are used to describe path problems in directed graphs. Constructing a new path algebra involves defining a carrier set, several operations, and proving that many algebraic properties hold. We describe work-in-progress on the development of a domain-specific language for specifying path algebras where implementations and proofs are *automatically* constructed in a bottom-up fashion. Our initial motivation came from the development of Internet routing protocols, but we believe that the approach could have much wider applications. We have implemented the language using the Coq theorem prover.

1 Introduction

Finding shortest paths in a graph is one of the fundamental problems in computer science. Algorithms for solving shortest path problems, such as Dijkstra's or Bellman-Ford, are widely used. Generalizations of shortest paths to semirings and related structures has been an active area of research for over forty years (see [2] for a survey). Recently, this approach has been further extended with exotic algebras that lack distributivity but can still be used to find *locally optimal* paths [7, 8]. We refer to such algebras, with or without distributivity, as *path algebras.*

Constructing a new path algebra involves defining a carrier set, several operations, and proving that many algebraic properties hold. With complex constructions such proofs can be quite challenging. In one example that we look at closely in Section 3, J. Monnot and O. Spanjaar [6] define a *bottleneck semiring* for solving certain combinatorial problems. The proofs that show the constructed algebra is in fact a semiring are non-trivial.

This paper describes a domain-specific language for the specification of path algebras. The goal is automating the implementation and verification of specified algebras. The language is designed so that the specification writer simply supplies a high-level expression comprised of names of *built-in* algebras and algebraic *constructors* (direct product, lexicographic product, etc.). The specified algebra's implementation and proofs (or refutations) for all covered properties are then derived in a bottom-up syntax-directed manner. Details of our approach are described in Section 2.

We have implemented[1] our language using the Coq theorem prover [1]. However, in our approach all interactive theorem proving is performed at language *design time*. Specification writers use a tool implemented in OCaml code that has been automatically extracted [5] from our Coq implementation. That is, specification writers are not required to do any interactive theorem proving.

Our initial motivation came from the development of Internet routing protocols, where the intended specification writers are network operators or protocols designers who are not well versed in the art of proving theorems [4]. However, we feel that this approach may have applications in other areas such as Operations Research. In particular, our implementation allows algebra designers to quickly explore a large space of specifications. By way of illustration, in

---

[1]The implementation is available at http://www.cl.cam.ac.uk/~tgg22/metarouting.

Section 3 we specify the bottleneck semiring mentioned above, and show how required properties for a semiring can be automatically derived.

This is very much a work-in-progress, and we discuss some of our ongoing efforts in Section 4.

2   Language Engineering

Our framework consists of a collection $\mathcal{L}$ of constructions of algebras (such as semirings), and a fixed finite set of properties $\mathbb{P}$ for these algebraic structures. The goal is for every algebra defined by composing constructions in $\mathcal{L}$ to decide for every property in $\mathbb{P}$ if it is true or false. We call such $\mathcal{L}$ to be *closed* w.r.t. $\mathbb{P}$. To achieve this, for every construction `c` in $\mathcal{L}$ and every $p \in \mathbb{P}$ we aim to have a rule of the following shape[2]

$$p(\mathtt{c}(a_1, \ldots, a_n)) \Leftrightarrow \beta_{c,p}(a_1, \ldots, a_n) \tag{1}$$

where $\beta_{c,p}$ stands for some boolean expression over properties in $\mathbb{P}$ of algebras $a_1, \ldots, a_n$. Let us call such rules — *iff-rules*. Notice that, if c takes no arguments, the right hand side is either true or false. Now given an algebra defined by constructions, we can use iff-rules to infer properties in bottom-up way.

Insisting on iff-rules may require adding new properties to $\mathbb{P}$. Consider the selectivity property $(\forall xy.\ x \oplus y = x \lor x \oplus y = y)$ for the direct product of semigroups $(A, \oplus_A)$ and $(B, \oplus_B)$. Instantiating the selectivity property with the direct product construction gives us

$$\forall x_1 y_1 \in A.\ \forall x_2 y_2 \in B.\ (x_1 \oplus_A y_1 = x_1 \land x_2 \oplus_B y_2 = x_2) \lor (x_1 \oplus_A y_1 = y_1 \land x_2 \oplus_B y_2 = y_2) \tag{2}$$

To get the iff-rule, we need to simplify (2), so that it becomes a boolean expression of properties of only A or only B. Consequently, we get

$$
\begin{aligned}
&((\forall x_1 y_1 \in A.\ x_1 \oplus_A y_1 = x_1) \land (\forall x_2 y_2 \in B.\ x_2 \oplus_B y_2 = x_2)) \\
&\lor ((\forall x_1 y_1 \in A.\ x_1 \oplus_A y_1 = y_1) \land (\forall x_2 y_2 \in B.\ x_2 \oplus_B y_2 = y_2)) \\
&\lor ((\forall x_1 y_1 \in A.\ x_1 \oplus_A y_1 = x_1 \lor x_1 \oplus_A y_1 = y_1) \land (\forall x_2 y_2 \in B.\ x_2 = y_2)) \\
&\lor ((\forall x_2 y_2 \in B.\ x_2 \oplus_B y_2 = x_2 \lor x_2 \oplus_B y_2 = y_2) \land (\forall x_1 y_1 \in A.\ x_1 = y_1))
\end{aligned}
\tag{3}
$$

If we do not have properties $\forall xy.\ x = y$, $\forall xy.\ x \oplus y = x$, and $\forall xy.\ x \oplus y = y$ in $\mathbb{P}$, we need to add them to get the iff-rule. As system develops we need to add more and more auxiliary properties. Which properties the final system will contain becomes an empirical observation, which is hard to predict beforehand. Tables in Appendix B list all iff-rules we currently know. The formula in (3) corresponds to the iff-rule in Table 3:

$$
\begin{aligned}
\mathrm{SL}(\mathtt{sProduct}(S, S')) \Leftrightarrow\ &(\mathrm{L}(S) \land \mathrm{L}(S')) \lor (\mathrm{R}(S) \land \mathrm{R}(S')) \lor \\
&(\mathrm{SL}(S) \land \mathrm{SG}(S')) \lor (\mathrm{SL}(S') \land \mathrm{SG}(S))
\end{aligned}
$$

We consider five different signatures shown in Fig. 1. All signatures have non-empty carrier. Additionally, they have axioms that $\oplus$ and $\otimes$ are associative binary operators, and $\leq$ is a preorder (reflexive and transitive) relation. Fig. 3 gives definitions of constructions for sets, semigroups and preorders. Some constructions take arguments of signatures from Fig. 1 together with additional axioms (we call them *preconditions*), e.g. a commutative and idempotent semigroup.

---

[2] We use `this` font for constructions of algebras.

ID	Name	Signature	Axioms
D	Sets	(S)	NE
S	Semigroups	$(S, \oplus)$	NE, ASSOC
P	Preorders	$(S, \leq)$	NE, REFL, TRANS
O	Order semigroups	$(S, \leq, \oplus)$	NE, REFL, TRANS, ASSOC
B	Bisemigroups	$(S, \oplus, \otimes)$	NE, $\text{ASSOC}_\oplus$, $\text{ASSOC}_\otimes$

Figure 1: A table of signatures with axioms. Notice that we have forgetful projections between signatures. Bisemigroups have two projection to semigroups as we can drop either $\oplus$ or $\otimes$.

ID	Name	Formula
NE	Non-empty	$\exists x.True$
ASSOC	Associative	$\forall xyz.x \oplus (y \oplus z) = (x \oplus y) \oplus z$
REFL	Reflexive	$\forall x.x \leq x$
TRANS	Transitive	$\forall xyz.x \leq y \Rightarrow y \leq z \Rightarrow x \leq z$

Figure 2: Definitions of axioms used in signatures.

Note that a construction not only has to define the appropriate operations and relations, but also to make sure that these operations satisfy required axioms.

To define bisemigroups and order semigroups it is enough to define their projections (Fig. 4). The last two constructions in Fig. 4 add special elements. `bAddOne`(B) and `bAddZero`(B) are equivalent to bisemigroups $(B \uplus \{\overline{1}\}, \oplus_B, \otimes_B)$ and $(B \uplus \{\overline{0}\}, \oplus_B, \otimes_B)$ respectively, where $\overline{1}$ is annihilator for $\oplus_B$ and identity for $\otimes_B$, and $\overline{0}$ is identity for $\oplus_B$ and annihilator for $\otimes_B$.

2.1 Witnesses

Another goal we have is to give witnesses to properties in $\mathbb{P}$ with existential quantifiers. We can split iff-rule (1) into two implication.

$$p(\mathsf{c}(a_1, \ldots, a_n)) \Leftarrow \beta_{c,p}(a_1, \ldots, a_n) \tag{4}$$

$$\neg p(\mathsf{c}(a_1, \ldots, a_n)) \Leftarrow \neg\beta_{c,p}(a_1, \ldots, a_n) \tag{5}$$

By $\neg$ we mean that the negation in front of p and $\beta_{c,p}$ is pushed through quantifiers to relation symbols. One of the implication is responsible for proving a property with existential quantifier. Say it is (5). If it is proved constructively, the proof says how to construct a witness for existential quantifier in $\neg p$ from witnesses of existential quantifiers in $\neg\beta_{c,p}$. Consequently, we can construct witnesses in bottom-up way as well as infer properties.

Some iff-rules like $\text{LD}(\texttt{bAddOne}(B)) \Leftrightarrow \text{LD}(B) \wedge \text{IDM}(B_\oplus) \wedge (\text{RI}(B) \vee \neg\text{IDM}(B_\oplus))$ may look unnecessarily complex, as classically it can be simplified to $\text{LD}(\texttt{bAddOne}(B)) \Leftrightarrow \text{LD}(B) \wedge \text{RI}(B)$. Consider the negative form (5) of this rule

$$\neg\text{LD}(\texttt{bAddOne}(B)) \Leftarrow \neg\text{LD}(B) \vee \neg\text{IDM}(B_\oplus) \vee (\neg\text{RI}(B) \wedge \neg\text{IDM}(B_\oplus))$$

Remember that the negation is pushed inside, e.g. $\neg\text{LD}$ is $\exists xyz.z \otimes (x \oplus y) \neq (z \otimes x) \oplus (z \otimes y)$. To prove the implication, we construct three different counterexamples corresponding to three cases separated by disjunction. The rule explicitly says that we need to make a case split if

<u>dUnit</u> is a singleton set $\mathbb{I}$.

<u>dNat</u> is a set $\mathbb{N}$ of natural numbers.

<u>dProduct</u> takes two sets A and B and constructs their direct product $A \times B$.

<u>dUnion</u> takes two sets A and B and constructs their disjoint union $A \uplus B$.

<u>dFSets</u> takes a sets A and constructs a set of all finite subsets of A, denoted by $\mathcal{P}(A)$.

<u>dFMinSets</u> takes a preorder $(A, \leq)$ and constructs a set of all minimal finite subsets of A, denoted by $\mathcal{P}_{\leq}(A)$. A subset X is minimal if $X = \min_{\leq}(X)$ where $\min_{\leq}(X) = \{x \in X \mid \forall y \in X . y \not< x\}$.

<u>sUnit</u> is a semigroup $(\mathbb{I}, K)$ where K is the constant binary operation.

<u>sNatPlus</u> is a semigroup $(\mathbb{N}, +)$.

<u>sNatMin</u> is a semigroup $(\mathbb{N}, \min)$.

<u>sNatMax</u> is a semigroup $(\mathbb{N}, \max)$.

<u>sProduct</u> takes two semigroups $(A, \oplus)$, $(B, \oplus')$ and constructs a semigroup $(A \times B, \oplus_{\times})$ where $(a_1, b_1) \oplus_{\times} (a_2, b_2) = (a_1 \oplus a_2, b_1 \oplus' b_2)$.

<u>sLeftSum</u> takes two semigroups $(A, \oplus)$, $(B, \oplus')$ and constructs a semigroup $(A \uplus B, \oplus_{A \uplus B})$ where

$$x \oplus_{A \uplus B} y = \begin{cases} x \oplus y & \text{if } x, y \in A \\ x & \text{if } x \in A, y \in B \\ y & \text{if } x \in B, y \in A \\ x \oplus' y & \text{if } x, y \in B \end{cases}$$

<u>sRightSum</u> takes two semigroups $(A, \oplus)$, $(B, \oplus')$ and constructs a semigroup $(B \uplus A, \oplus_{A \uplus B})$.

<u>sLex</u> takes two semigroups $(A, \oplus)$ and $(B, \oplus')$, s.t. $\oplus$ is commutative and idempotent, and $\oplus'$ has an identity elements $\overline{0}_B$. The resulting semigroup is

$(A \times B, \vec{\oplus})$ where $(x_1, x_2) \vec{\oplus} (y_1, y_2) =$

$$\begin{cases} (x_1, x_2 \oplus' y_2) & \text{if } x_1 = y_1 \\ (x_1, x_2) & \text{if } x_1 = (x_1 \oplus y_1) \neq y_1 \\ (y_1, y_2) & \text{if } x_1 \neq (x_1 \oplus y_1) = y_1 \\ (x_1 \oplus y_1, \overline{0}_B) & \text{if } x_1 \neq (x_1 \oplus y_1) \neq y_1 \end{cases}$$

$\vec{\oplus}$ first chooses according to $\oplus$, only if $x_1 = y_1$ it chooses according to $\oplus'$. The fourth case is used when x_1 and y_1 are incomparable.

<u>sSelLex</u> is similar to <u>sLex</u>. It does not require $\oplus'$ to have the identity, but instead the $\oplus$ has to be selective. Consequently, the fourth case in $\vec{\oplus}$ can never happen.

<u>sFSetsUnion</u> takes a set A and constructs a semigroup $(\mathcal{P}(A), \cup)$.

<u>sFSetsOp</u> takes a semigroup $(A, \oplus)$ and constructs a semigroup $(\mathcal{P}(A), \hat{\oplus})$ where

$$X \hat{\oplus} Y = \{x \oplus y \mid x \in X, y \in Y\}$$

<u>sFMinSetsUnion</u> takes a preorder $(A, \leq)$, s.t. $\leq$ is antisymmetric, and constructs a semigroup $(\mathcal{P}_{\leq}(A), \cup_{\leq})$ where $\cup_{\leq} = \min_{\leq} \circ \cup$.

<u>sFMinSetsOp</u> takes an order semigroup $(A, \leq, \oplus)$, s.t. $\leq$ is antisymmetric and $\oplus$ is monotone, and constructs a semigroup $(\mathcal{P}_{\leq}(A), \hat{\oplus}_{\leq})$ where $\hat{\oplus}_{\leq} = \min_{\leq} \circ \hat{\oplus}$.

<u>pLeftNaturalOrder</u> takes a semigroup $(A, \oplus)$, s.t. $\oplus$ is commutative and idempotent, and constructs a preorder $(A, \leq_L)$ where $x \leq_L y \Leftrightarrow x \oplus y = x$.

<u>pRightNaturalOrder</u> takes a semigroup $(A, \oplus)$, s.t. $\oplus$ is commutative and idempotent, and constructs a preorder $(A, \leq_R)$ where $x \leq_R y \Leftrightarrow x \oplus y = y$.

<u>pDual</u> takes a preorder $(A, \leq)$ and constructs a preorder $(A, \geq)$.

Figure 3: Constructions of sets, semigroups and preorders.

$IDM(B_{\oplus})$ holds or not in order to construct the counterexample. Classically the case split is trivial, but constructively we cannot do it for arbitrary B.

Splitting iff-rules into (4) and (5) helps us in using the framework that is under development where we do not have iff-rules. If $\beta_{c,p}$ in (4) is different from $\beta_{c,p}$ in (5), we can still generate proofs and compute witnesses, but we may run into situations where we cannot decide if some property is true or false.

2.2 Key Properties

All properties in $\mathbb{P}$ are defined in Appendix A. In this section we explain reasons why some properties were initially added to $\mathbb{P}$. Let us denote the initial set of properties by $\mathbb{P}_0$.

One of the aims of the system is to build semirings. Bisemigroups only guarantee associa-

Constructor in $\mathcal{L}$	$\leq$ projection	$\oplus$ projection
Order semigroups		
oLeftNaturalOrder(S)	pLeftNaturalOrder(S)	S
oRightNaturalOrder(S)	pRightNaturalOrder(S)	S
Constructor in $\mathcal{L}$	$\oplus$ projection	$\otimes$ projection
Bisemigroups		
bUnit	sUnit	sUnit
bNatMinPlus	sNatMin	sNatPlus
bNatMaxMin	sNatMax	sNatMin
bProduct(B, B')	sProduct$(B_\oplus, B'_\oplus)$	sProduct$(B_\otimes, B'_\otimes)$
bLex(B, B')	sLex$(B_\oplus, B'_\oplus)$	sProduct$(B_\otimes, B'_\otimes)$
bSelLex(B, B')	sSelLex$(B_\oplus, B'_\oplus)$	sProduct$(B_\otimes, B'_\otimes)$
bFSetsOp(S)	sFSetsUnion(S_{Set})	sFSetsOp(S)
bFMinSets(O)	sFMinSetsUnion$(O_\leq)$	sFMinSetsOp(O)
bFMinSetsOpUnion(O)	sFMinSetsOp(O)	sFMinSetsUnion$(O_\leq)$
bAddOne(B)	sLeftSum$(\text{sUnit}, B_\oplus)$	sRightSum$(B_\otimes, \text{sUnit})$
bAddZero(B)	sLeftSum$(B_\oplus, \text{sUnit})$	sRightSum$(\text{sUnit}, B_\otimes)$

Figure 4: Constructions for order semigroups and bisemigroups. S ranges over semigroups, B, B' — over bisemigroups, O — over order semigroups. We use indexes $(Set, \oplus, \otimes, \leq)$ to denote projected algebras. In addition to axioms inherited from projections bSelLex requires $B'_\oplus$ to be commutative, and bAddOne requires $B_\oplus$ to be commutative.

tivity. Hence, we need other semiring axioms to know if a bisemigroup is actually a semiring.

Studies of BGP [3] protocol show that path algebras are interesting even when operations $\oplus$ and $\otimes$ do not form a semiring. In such case we are not looking for optimal paths, but for locally optimal paths, where each node has the best path depending on what their neighbours have chosen. An interesting property in this scenario is the increasing property $(\forall xy. x \oplus (y \otimes x) = x)$, which as shown by T.G. Griffin and J.L. Sobrinho [9] is needed for Dijkstra's algorithm to find locally optimal solutions.

Another key property is: the identity for $\oplus$ is also the annihilator for $\otimes$. It implies 0-stability property $(\forall x. \overline{1} \oplus x = \overline{1})$ used by M. Gondran and M. Minoux [2]. 0-stability guarantees the convergence of matrix multiplication shortest path algorithm in n steps, where n is the number of nodes in the graph.

Also properties like idempotency, selectivity or antisymmetry are in $\mathbb{P}_0$, because they are required for some constructions as preconditions.

2.3   Iff-Rules in Context

Some properties defined in Appendix A are only meaningful in the context where some other properties (say $p_1, \ldots, p_n$) are satisfied. We denote this by $p(p_1, \ldots, p_n)$. If a property is defined in a context or a construction has preconditions, the proofs of the iff-rule take the context and preconditions as assumption. For example, to proof the iff-rule for minimal set construction of bisemigroups and right increasing property we need to show

$$\text{CM}_\oplus(\text{bFMinSets}(O)) \Rightarrow \text{IDM}_\oplus(\text{bFMinSets}(O)) \Rightarrow$$
$$\text{LM}(O) \Rightarrow \text{RM}(O) \Rightarrow \text{ASM}_\leq(O) \Rightarrow (\text{RI}(\text{bFMinSets}(O)) \Leftrightarrow \text{LND}(O))$$

2.4 Putting it all together

We defined various constructions of algebras and showed how iff-rules can be used to prove properties. To make an automatic tool out of iff-rules we need to reflect constructions in the collection $\mathcal{L}$ into a mutually inductive language (with a syntactic type for each signature). For each construction we have a corresponding syntactic constructor. To map back from terms in syntax to algebras, we have a semantics function that is mutually inductively defined on the syntax structure, e.g. for bisemigroups the semantics function has type

$$\text{BS} \to (S, \oplus, \otimes, \vec{\pi}, \vec{\rho}) + \text{error}$$

where BS is the syntactic category for bisemigroup specifications, $(S, \oplus, \otimes)$ is a bisemigroup, $\vec{\pi}$ contains proofs that this is actual a bisemigroup, i.e. S is not empty and $\oplus, \otimes$ are associative, $\vec{\rho}$ for each property in $\mathbb{P}$ contains a proof or a refutation. The semantics function fails and returns an error if some preconditions of constructors specified in the input do not hold.

We have implemented all definitions of constructions and proved the iff-rules in Coq. We also defined syntax and the semantic function in Coq. Using code extraction mechanism [5], we generate an OCaml implementation of the semantics function that can be invoked without invoking the Coq theorem prover. This gives us a tool for quickly defining algebras and getting their properties together with witnesses. Also since the semantics function provides implementations for the $\oplus$ and $\otimes$ operations, we can construct a concrete labelled graph and run a generalised shortest path algorithm.

3 Examples

3.1 Lexicographic Product

To show how the language can be used, let us consider a graph where each edge has a distance and a bandwidth. Say we want to find best paths according to these two metrics. Bisemigroups `bNatMinPlus` and `bNatMaxMin` can be used to represent distance and bandwidth respectively. We can make one metric more significant by ordering pairs of metrics lexicographicly. Here are the specifications of algebras in our syntax:

$$\text{bAddOne}(\text{bAddZero}(\text{bSelLex}(\text{bNatMinPlus}, \text{bNatMaxMin}))) \tag{6}$$

$$\text{bAddOne}(\text{bAddZero}(\text{bSelLex}(\text{bNatMaxMin}, \text{bNatMinPlus}))) \tag{7}$$

However, the choice of which metric is more significant gives us algebras with significantly different properties. The bisemigroup specified by (6) is a semiring, but the one specified by (7) is not distributive. We illustrate how we can check these properties using iff-rules in Appendix B by proving left distributivity of the first bisemigroup.

$$\text{LD}(\text{bAddOne}(\text{bAddZero}(\text{bSelLex}(\text{bNatMinPlus}, \text{bNatMaxMin}))))$$
$$\Leftrightarrow \text{LD}(\text{bAddZero}(\text{bSelLex}(\text{bNatMinPlus}, \text{bNatMaxMin}))) \land$$
$$\text{IDM}_{\oplus}(\text{bAddZero}(\text{bSelLex}(\text{bNatMinPlus}, \text{bNatMaxMin}))) \land$$
$$(\text{RI}(\text{bAddZero}(\text{bSelLex}(\text{bNatMinPlus}, \text{bNatMaxMin}))) \lor$$
$$\neg \text{IDM}_{\oplus}(\text{bAddZero}(\text{bSelLex}(\text{bNatMinPlus}, \text{bNatMaxMin}))))$$
$$\Leftrightarrow \text{LD}(\text{bSelLex}(\text{bNatMinPlus}, \text{bNatMaxMin})) \land$$
$$\text{IDM}(\text{sLeftSum}(\text{sUnit}, \text{sSelLex}(\text{sNatMin}, \text{sNatMax}))) \land$$

$$(\text{RI}(\text{bSelLex}(\text{bNatMinPlus}, \text{bNatMaxMin})) \vee$$
$$\neg \text{IDM}(\text{sLeftSum}(\text{sUnit}, \text{sSelLex}(\text{sNatMin}, \text{sNatMax}))))$$
$$\Leftrightarrow (\text{LD}(\text{bNatMinPlus}) \wedge \text{LD}(\text{bNatMaxMin}) \wedge (\text{LC}(\text{sNatPlus}) \vee \text{LCD}(\text{sNatMin}))) \wedge$$
$$(\text{IDM}(\text{sUnit}) \wedge \text{IDM}(\text{sSelLex}(\text{sNatMin}, \text{sNatMax}))) \wedge$$
$$((\text{RSI}(\text{bNatMinPlus}) \vee (\text{RI}(\text{bNatMinPlus}) \wedge \text{RI}(\text{bNatMaxMin}))) \vee$$
$$\neg (\text{IDM}(\text{sUnit}) \wedge \text{IDM}(\text{sSelLex}(\text{sNatMin}, \text{sNatMax}))))$$
$$\Leftrightarrow (\text{LD}(\text{bNatMinPlus}) \wedge \text{LD}(\text{bNatMaxMin}) \wedge (\text{LC}(\text{sNatPlus}) \vee \text{LCD}(\text{sNatMin}))) \wedge$$
$$(\text{IDM}(\text{sUnit}) \wedge \text{IDM}(\text{sNatMax})) \wedge$$
$$((\text{RSI}(\text{bNatMinPlus}) \vee (\text{RI}(\text{bNatMinPlus}) \wedge \text{RI}(\text{bNatMaxMin}))) \vee$$
$$\neg (\text{IDM}(\text{sUnit}) \wedge \text{IDM}(\text{sNatMax})))$$
$$\Leftrightarrow (True \wedge True \wedge (True \vee False)) \wedge (True \wedge True) \wedge$$
$$((False \vee (True \wedge True)) \vee \neg(True \wedge True))$$
$$\Leftrightarrow True$$

Such derivations are done mechanically by our tool. In a similar way we can derive $False$ for left distributivity of the second bisemigroup. The tool also generates a counterexample for left distributivity:

$$(0,1) \otimes_\times ((1,1) \vec{\oplus} (0,0)) = (0,2) \neq (0,1) = ((0,1) \otimes_\times (1,1)) \vec{\oplus} ((0,1) \otimes_\times (0,0))$$

3.2   The Bottleneck Semiring

As a second example, consider the semiring defined by J. Monnot and O. Spanjaar [6] for finding best paths according to their bottleneck. Say we have a graph where edges are labelled by two independent metrics (two natural numbers). Values assigned to edges are partially ordered by comparing metrics pointwise. The weight of a path consists of a set of worst edges in the path. A path x is more preferred to another path y if for each edge in x there is a worse edge in y. We aim to have a semiring that calculates a set of such best paths between every pair of nodes in the graph.

In [6] the semiring is explicitly defined together with non-trivial proofs of semiring axioms. By reverse engineering definitions of the semiring operations, we can specify it in our language in the following way. We can represent pairs of metrics by `E = sProduct(sNatMin, sNatMin)`. Weights of paths are represented by `P = sFMinSetsUnion(pRightNaturalOrder(E))`. Finally, the bisemigroup that can be used to compute sets of best paths is

$$\text{B} = \text{bFMinSets}(\text{oRightNaturalOrder}(\text{P})) \tag{8}$$

Most importantly, in order to come up with the specification, we do not need to look to the proofs given in [6] — these can be generated automatically using the iff-rules as in the previous example.

If we take acyclic graphs as in [6], we can compute shortest paths according to the semiring using Bellman-Ford algorithm (Fig. 5). However, the semiring is not selective and it cannot be used with Dijkstra's algorithm. From the tool we get witnesses $(\{\{(1,0)\}\}$ and $\{\{(0,1)\}\})$ explaining where selectivity fails.

$$
\begin{array}{lll lll}
0-1 & \{\{(1,3)\}\} & 1-4 & \{\{(3,1)\}\} & 3-4 & \{\{(1,1)\}\} \\
0-2 & \{\{(1,1)\}\} & 1-5 & \{\{(3,1);(2,2)\}\} & 3-5 & \{\{(2,2)\}\} \\
0-3 & \{\{(1,3),(3,1)\},\{(2,3)\}\} & 1-6 & \{\{(3,1)\}\} & 3-6 & \{\{(1,1)\}\} \\
0-4 & \{\{(1,3),(3,1)\},\{(2,3)\}\} & 2-3 & \{\{(2,3)\}\} & 4-6 & \{\{(1,1)\}\} \\
0-5 & \{\{(1,3),(3,1),(2,2)\},\{2,3\}\} & 2-4 & \{\{(2,3)\}\} & 5-6 & \{\{(3,2)\}\} \\
0-6 & \{\{(1,3),(3,1)\},\{(2,3)\}\} & 2-5 & \{\{(2,3)\}\} & & \\
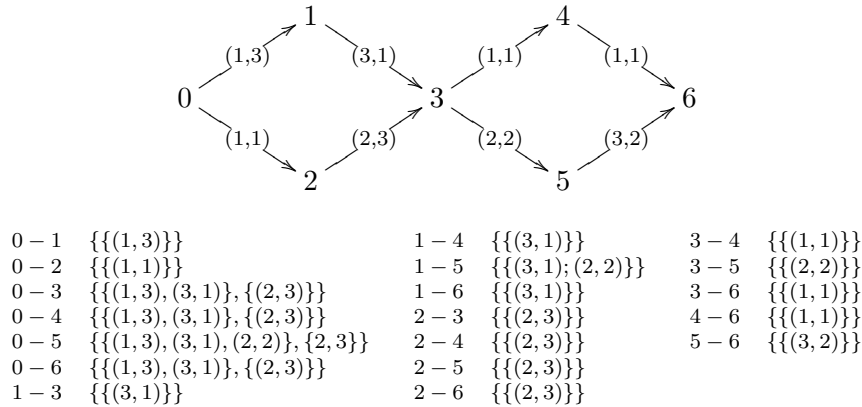1-3 & \{\{(3,1)\}\} & 2-6 & \{\{(2,3)\}\} & &
\end{array}
$$

Figure 5: At the top is the example graph taken from [6]. At the bottom are the results computed by the tool using the bisemigroup defined in (8). For each pair of nodes we have a set of best paths. Each path is valued by a set of worst edges on that path.

4   Discussion, Future Work

Our approach to language design requires many iff-rules and it would be hard to ensure correctness without using a formal theorem prover. We have chosen the Coq theorem prover as it seems to meet our requirements quite well. Dependent types allow defining signatures for algebras as dependent records. Constructive proofs in Coq fit our need to associate witnesses to proofs of existential quantifiers. We haven't yet used some recent Coq features, such as type classes [10], which may simplify some of the infrastructure for our proofs.

A large part of our current effort is directed at "closing" the iff-rules listed in Appendix B. There is of course a conflict between the goal of closure and the goal of increased expressive power, and so various trade-offs have to be assessed at language design time.

Not all natural path problems can be expressed using bisemigroups. Many Internet routing protocols are best modelled by attaching functions to arcs in a graph. We are currently extending our system to encompass such algebraic structures.

References

[1] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer-Verlag, 2004.

[2] M. Gondran and M. Minoux. *Graphs, Dioids and Semirings: New Models and Algorithms.* Springer, 2008.

[3] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking (TON)*, 10(2):232–243, 2002.

[4] T. G. Griffin and J. L. Sobrinho. Metarouting. *SIGCOMM*, 35:1–12, August 2005.

[5] P. Letouzey. A new extraction for coq. *Types for proofs and programs*, pages 617–617, 2003.

[6] J. Monnot and O. Spanjaard. Bottleneck shortest paths on a partially ordered scale. *4OR: A Quarterly Journal of Operations Research*, 1(3):225–241, 2003.

[7] J. Sobrinho and T. G. Griffin. Routing in equilibrium. In *19th International Symposium on Mathematical Theory of Networks and Systems (MTNS 2010)*, 2010.

[8] J. L. Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking*, 13(5):1160–1173, October 2005.

[9] J. L. Sobrinho and T. G. Griffin. Routing in equilibrium. *Mathematical Theory of Networks and System*, 2010.

[10] B. Spitters and E Van Der Weegen. Developing the algebraic hierarchy with type classes in coq. *ITP 2010. International Conference on Interactive Theorem Proving*, 2010.

A Property Set $\mathbb{P}$

We use shorthands: $(x\#y) \Leftrightarrow (x \not\leq y \wedge y \not\leq x)$ and $(x \lesseqgtr y) \Leftrightarrow \neg(x\#y)$. For bisemigroups $\leq$ denotes the left natural order.

$\mathbb{P}_0$	ID(Context)	Description	Formula
	DecSetoid		
	SG	Singleton	$\exists c. \forall x. x = c$
	TE	Has exactly two elements	$\exists ab. \forall x. a \neq b \wedge (x = a \vee x = b)$
	FT	Finite	$\exists l : list\ S. \forall x. x \in l$
	Semigroup		
*	HI	Has identity	$\exists i. \forall x.(i \oplus x = x) \wedge (x \oplus i = x)$
*	HA	Has annihilator	$\exists w. \forall x.(w \oplus x = w) \wedge (x \oplus w = w)$
*	SL	Selective	$\forall xy.(x \oplus y = x) \vee (x \oplus y = y)$
*	CM	Commutative	$\forall ab. a \oplus b = b \oplus a$
*	IDM	Idempotent	$\forall x. x \oplus x = x$
	L	Always returns the left argument	$\forall ab. a \oplus b = a$
	R	Always returns the right argument	$\forall ab. a \oplus b = b$
	LCD	Left condensed	$\forall abc. a \oplus b = a \oplus c$
	RCD	Right condensed	$\forall abc. b \oplus a = c \oplus a$
	LC	Left cancelative	$\forall xyz. z \oplus x = z \oplus y \Rightarrow x = y$
	RC	Right cancelative	$\forall xyz. x \oplus z = y \oplus z \Rightarrow x = y$
	AL	Anti-left	$\forall xy. x \oplus y \neq x$
	AR	Anti-right	$\forall xy. x \oplus y \neq y$
	TG(CM, IDM)		$\forall xyz. x \oplus y \oplus z = x \oplus z \vee x \oplus y \oplus z = y \oplus z$
	Preorder		
*	TT	Total	$\forall xy. x \leq y \vee y \leq x$
*	ASM	Antisymmetric	$\forall xy. x \leq y \wedge y \leq x \Rightarrow x = y$
	OrderSemigroup		
*	LM	Left monotonic	$\forall axy. x \leq y \Rightarrow (a \oplus x \leq a \oplus y)$
*	RM	Right monotonic	$\forall xya. x \leq y \Rightarrow x \oplus a \leq y \oplus a$
	LND	Left non-decreasing	$\forall xy. x \leq x \oplus y$
	RND	Right non-decreasing	$\forall xy. x \leq y \oplus x$
	SND(IDM, ASM)	Selective non-decreasing	$\forall xy. x \leq x \oplus y \vee y \leq x \oplus y$
	IAUS(LM, RM, ASM, SL)		$\forall xyz. x\#y \Rightarrow x\oplus y = y \Rightarrow y\oplus x = y \Rightarrow z\#y \Rightarrow x = z$
	IAF(LM, RM, ASM, SL)		$\forall xyz. x\#y \Rightarrow x\oplus y = y \Rightarrow y\oplus x = y \Rightarrow x\oplus z = z \Rightarrow z\oplus x = z \Rightarrow x \neq z \Rightarrow (y\oplus z = z \wedge z\oplus y = z) \vee z \leq y$
	RT	Right total	$\forall abc.(b \oplus a) \leq (c \oplus a) \vee (c \oplus a) \leq (b \oplus a)$
	LT	Left total	$\forall abc.(a \oplus b) \leq (a \oplus c) \vee (a \oplus c) \leq (a \oplus b)$
	RMCC(RT)		$\forall xyzw.(x\oplus z) < (y\oplus z) \Rightarrow (y\oplus w) < (x\oplus w) \Rightarrow (x \oplus z) \leq (y \oplus w) \vee (y \oplus w) \leq (x \oplus z)$
	LMCC(LT)		$\forall xyzw.(z\oplus x) < (z\oplus y) \Rightarrow (w\oplus y) < (w\oplus x) \Rightarrow (z \oplus x) \leq (w \oplus y) \vee (w \oplus y) \leq (z \oplus x)$
	Bisemigroup		
*	LD	Left distributive	$\forall abc. c \otimes (a \oplus b) = (c \otimes a) \oplus (c \otimes b)$
*	RD	Right distributive	$\forall abc.(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$
*	PITA(HI$_\oplus$, HA$_\otimes$)	Plus identity is times annihilator	$\alpha_\oplus = \omega_\otimes$
*	PATI(HA$_\oplus$, HI$_\otimes$)	Plus annihilator is times identity	$\omega_\oplus = \alpha_\otimes$
*	RSI(CM$_\oplus$, IDM$_\oplus$)	Right strict increasing	$\forall xy. x < x \otimes y$
*	LSI(CM$_\oplus$, IDM$_\oplus$)	Left strict increasing	$\forall xy. x < y \otimes x$
	RI(CM$_\oplus$, IDM$_\oplus$)	Right increasing	$\forall xy. x \oplus (x \otimes y) = x$
	LI(CM$_\oplus$, IDM$_\oplus$)	Left increasing	$\forall xy. x \oplus (y \otimes x) = x$
	RSS(CM$_\oplus$, IDM$_\oplus$)		$\forall abc.(a < b \Leftrightarrow a \otimes c < b \otimes c)$
	LSS(CM$_\oplus$, IDM$_\oplus$)		$\forall abc.(a < b \Leftrightarrow c \otimes a < c \otimes b)$
	RCEC(CM$_\oplus$, IDM$_\oplus$)		$\forall xyz. x \otimes z = y \otimes z \Rightarrow (x \lesseqgtr y)$
	LCEC(CM$_\oplus$, IDM$_\oplus$)		$\forall xyz. z \otimes x = z \otimes y \Rightarrow (x \lesseqgtr y)$

$\mathbb{P}_0$	ID(Context)	Description	Formula
	RCC(CM$_\oplus$, IDM$_\oplus$)		$\forall xyz.x \otimes z \# y \otimes z \Rightarrow (x \lesseqgtr y)$
	LCC(CM$_\oplus$, IDM$_\oplus$)		$\forall xyz.z \otimes x \# z \otimes y \Rightarrow (x \lesseqgtr y)$
	LDT(CM$_\oplus$, IDM$_\oplus$)	Left discrete	$\forall xyz.\neg(z \otimes x < z \otimes y)$
	RDT(CM$_\oplus$, IDM$_\oplus$)	Right discrete	$\forall xyz.\neg(x \otimes z < y \otimes z)$
	LCP(CM$_\oplus$, IDM$_\oplus$)	Left comparable	$\forall xyz.z \otimes x \lesseqgtr z \otimes y$
	RCP(CM$_\oplus$, IDM$_\oplus$)	Right comparable	$\forall xyz.x \otimes z \lesseqgtr y \otimes z$
	RTID(HI$_\oplus$)		$\forall xyz.(x \otimes z) \oplus (y \otimes z) = \alpha_\oplus \otimes z$
	LTID(HI$_\oplus$)		$\forall xyz.(z \otimes x) \oplus (z \otimes y) = z \otimes \alpha_\oplus$
	PITLA(HI$_\oplus$)	Plus identity is times left annihilator	$\forall x.\alpha_\oplus \otimes x = \alpha_\oplus$
	RITRA(HI$_\oplus$)	Plus identity is times right annihilator	$\forall x.x \otimes \alpha_\oplus = \alpha_\oplus$

B   Iff-Rules

	dProduct(D,D')	dUnion(D,D')	dFSets(D)	dFMinSets(P)
Prec.				
SG	SG(D) $\wedge$ SG(D')	False	False	False
TE	(SG(D) $\wedge$ TE(D')) $\vee$ (SG(D') $\wedge$ TE(D))	SG(D) $\wedge$ SG(D')	SG(D)	SG(P)
FT	FT(D) $\wedge$ FT(D')	FT(D) $\wedge$ FT(D')	FT(D)	FT(P)

Table 2: DecSetoid rules

	sProduct(S,S')	sLex(S,S')	sSelLex(S,S')	sLeftSum(S,S')
Prec.		CM(S), IDM(S), HI(S')	CM(S), SL(S)	
HI	HI(S) $\wedge$ HI(S')	HI(S)	HI(S) $\wedge$ HI(S')	HI(S')
HA	HA(S) $\wedge$ HA(S')	HA(S) $\wedge$ HA(S')	HA(S) $\wedge$ HA(S')	HA(S)
SL	(L(S) $\wedge$ L(S')) $\vee$ (R(S) $\wedge$ R(S')) $\vee$ (SL(S) $\wedge$ SG(S')) $\vee$ (SL(S') $\wedge$ SG(S))	SL(S) $\wedge$ SL(S')	SL(S')	SL(S) $\wedge$ SL(S')
CM	CM(S) $\wedge$ CM(S')	CM(S')	CM(S')	CM(S) $\wedge$ CM(S')
IDM	IDM(S) $\wedge$ IDM(S')	IDM(S')	IDM(S')	IDM(S) $\wedge$ IDM(S')
L	L(S) $\wedge$ L(S')	SG(S) $\wedge$ L(S')	SG(S) $\wedge$ L(S')	False
R	R(S) $\wedge$ R(S')	SG(S) $\wedge$ R(S')	SG(S) $\wedge$ R(S')	False
LCD	LCD(S) $\wedge$ LCD(S')	SG(S) $\wedge$ SG(S')	SG(S) $\wedge$ LCD(S')	False
RCD	RCD(S) $\wedge$ RCD(S')	SG(S) $\wedge$ SG(S')	SG(S) $\wedge$ RCD(S')	False
LC	LC(S) $\wedge$ LC(S')	SG(S) $\wedge$ LC(S')	(SG(S) $\vee$ ($\neg$SG(S') $\wedge$ SG(S'))) $\wedge$ LC(S')	LC(S) $\wedge$ AL(S) $\wedge$ SG(S')
RC	RC(S) $\wedge$ RC(S')	SG(S) $\wedge$ RC(S')	(SG(S) $\vee$ ($\neg$SG(S') $\wedge$ SG(S'))) $\wedge$ RC(S')	RC(S) $\wedge$ AR(S) $\wedge$ SG(S')
AL	AL(S) $\vee$ AL(S')	SG(S) $\wedge$ AL(S')	SG(S) $\wedge$ AL(S')	False
AR	AR(S) $\vee$ AR(S')	SG(S) $\wedge$ AR(S')	SG(S) $\wedge$ AR(S')	False
TG	(TG(S) $\wedge$ SG(S')) $\vee$ (TG(S') $\wedge$ SG(S))	*work-in-progress*	*work-in-progress*	SL(S) $\wedge$ TG(S')

Table 3: Semigroup rules

	sFSetsUnion(D)	sFSetsOp(S)	sFMinSetsUnion(P)	sFMinSetsOp(O)
Prec.			ASM(P)	ASM(O)
HI	True	HI(S)	True	*pos* : HI(O)
HA	FT(D)	True	*pos* : FT(P)	True
SL	SG(D)	L(S) $\vee$ R(S) $\vee$ (TE(S) $\wedge$ IDM(S))	TT(P)	SL(O) $\wedge$ ($\neg$SL(O) $\vee$ (IAUS(O) $\wedge$ IAF(O)))
CM	True	CM(S)	True	CM(O)
IDM	True	SL(S)	True	IDM(O) $\wedge$ ($\neg$IDM(O) $\vee$ SND(O))
L	False	False	False	False
R	False	False	False	False
LCD	False	False	False	False
RCD	False	False	False	False
LC	False	False	False	False
RC	False	False	False	False
AL	False	False	False	False
AR	False	False	False	False

	sFSetsUnion(D)	sFSetsOp(S)	sFMinSetsUnion(P)	sFMinSetsOp(O)
TG	SG(D)	*work-in-progress*	*work-in-progress*	*work-in-progress*

Table 4: Semigroup rules. Rules prefixed with *pos* : mean that we know only positive implication (4)

	pLeftNaturalOrder(S)	pRightNaturalOrder(S)	pDual(P)
Prec.	CM(S), IDM(S)	CM(S), IDM(S)	
TT	SL(S)	SL(S)	TT(P)
ASM	True	True	ASM(P)

Table 5: Preorder rules

	oLeftNaturalOrder(S)	oRightNaturalOrder(S)
Prec.	CM(S), IDM(S)	CM(S), IDM(S)
LM	True	True
RM	True	True
LND	SG(S)	True
RND	SG(S)	True
SND	SL(S)	True
IAUS	True	True
IAF	True	True
RT	TG(S)	TG(S)
LT	TG(S)	TG(S)
RMCC	True	SL(S)
LMCC	True	SL(S)

Table 6: OrderSemigroup rules

	bProduct(B,B')	bLex(B,B')	bSelLex(B,B')	bFSetsOp(S)
Prec.		$CM(B_\oplus)$, $IDM(B_\oplus)$, $HI(B'_\oplus)$	$CM(B_\oplus)$, $SL(B_\oplus)$, $CM(B'_\oplus)$	
LD	LD(B) ∧ LD(B')	LD(B) ∧ LD(B') ∧ (LSS(B) ∨ $LCD(B'_\otimes)$) ∧ (LCEC(B) ∨ LTID(B')) ∧ (LCC(B) ∨ RITRA(B'))	LD(B) ∧ LD(B') ∧ ($LC(B_\otimes)$ ∨ $LCD(B'_\otimes)$)	True
RD	RD(B) ∧ RD(B')	RD(B) ∧ RD(B') ∧ (RSS(B) ∨ $RCD(B'_\otimes)$) ∧ (RCEC(B) ∨ RTID(B')) ∧ (RCC(B) ∨ PITLA(B'))	RD(B) ∧ RD(B') ∧ ($RC(B_\otimes)$ ∨ $RCD(B'_\otimes)$)	True
PITA	PITA(B) ∧ PITA(B')	PITA(B) ∧ PITA(B')	PITA(B) ∧ PITA(B')	True
PATI	PATI(B) ∧ PATI(B')	PATI(B) ∧ PATI(B')	PATI(B) ∧ PATI(B')	SG(S)
RSS	RSS(B) ∧ RSS(B') ∧ (RDT(B) ∨ RCEC(B')) ∧ (RDT(B') ∨ RCEC(B))	RSS(B) ∧ RSS(B') ∧ (RCEC(B) ∨ RDT(B'))	RSS(B) ∧ RSS(B')	False
LSS	LSS(B) ∧ LSS(B') ∧ (LDT(B) ∨ LCEC(B')) ∧ (LDT(B') ∨ LCEC(B))	LSS(B) ∧ LSS(B') ∧ (LCEC(B) ∨ LDT(B'))	LSS(B) ∧ LSS(B')	False
RCEC	RCEC(B) ∧ RCEC(B') ∧ ($RC(B_\otimes)$ ∨ $RC(B'_\otimes)$)	RCEC(B) ∧ RCEC(B')	RCEC(B) ∧ RCEC(B')	SG(S)
LCEC	LCEC(B) ∧ LCEC(B') ∧ ($LC(B_\otimes)$ ∨ $LC(B'_\otimes)$)	LCEC(B) ∧ LCEC(B')	LCEC(B) ∧ LCEC(B')	SG(S)
RCC	*work-in-progress*	(RCEC(B) ∨ RCP(B')) ∧ RCC(B) ∧ RCC(B')	RCC(B) ∧ RCC(B')	RCD(S)
LCC	*work-in-progress*	(LCEC(B) ∨ LCP(B')) ∧ LCC(B) ∧ LCC(B')	LCC(B) ∧ LCC(B')	LCD(S)
LDT	LDT(B) ∧ LDT(B')	LDT(B) ∧ LDT(B')	LDT(B) ∧ LDT(B')	False
RDT	RDT(B) ∧ RDT(B')	RDT(B) ∧ RDT(B')	RDT(B) ∧ RDT(B')	False
LCP	LCP(B) ∧ LCP(B') ∧ (LDT(B) ∨ LDT(B'))	LCP(B) ∧ LCP(B')	LCP(B) ∧ LCP(B')	LCD(S)
RCP	RCP(B) ∧ RCP(B') ∧ (RDT(B) ∨ RDT(B'))	RCP(B) ∧ RCP(B')	RCP(B) ∧ RCP(B')	RCD(S)
RI	RI(B) ∧ RI(B')	RSI(B) ∨ (RI(B) ∧ RI(B'))	RSI(B) ∨ (RI(B) ∧ RI(B'))	L(S)
LI	LI(B) ∧ LI(B')	LSI(B) ∨ (LI(B) ∧ LI(B'))	LSI(B) ∨ (LI(B) ∧ LI(B'))	R(S)
RSI	(RI(B) ∧ RSI(B')) ∨ (RSI(B) ∧ RI(B'))	RSI(B) ∨ (RI(B) ∧ RSI(B'))	RSI(B) ∨ (RI(B) ∧ RSI(B'))	False

	bProduct(B,B')	bLex(B,B')	bSelLex(B,B')	bFSetsOp(S)
LSI	(LI(B) ∧ LSI(B')) ∨ (LSI(B) ∧ LI(B'))	LSI(B) ∨ (LI(B) ∧ LSI(B'))	LSI(B) ∨ (LI(B) ∧ LSI(B'))	False
RTID	RTID(B) ∧ RTID(B')	RTID(B) ∧ RTID(B') ∧ (RDT(B) ∨ RCD(B'$_\otimes$)) ∧ (RCP(B) ∨ PITLA(B'))	RTID(B) ∧ RTID(B') ∧ (RDT(B) ∨ RCD(B'$_\otimes$))	False
LTID	LTID(B) ∧ LTID(B')	LTID(B) ∧ LTID(B') ∧ (LDT(B) ∨ LCD(B'$_\otimes$)) ∧ (LCP(B) ∨ RITRA(B'))	LTID(B) ∧ LTID(B') ∧ (LDT(B) ∨ LCD(B'$_\otimes$))	False
PITLA	PITLA(B) ∧ PITLA(B')	PITLA(B) ∧ PITLA(B')	PITLA(B) ∧ PITLA(B')	True
RITRA	RITRA(B) ∧ RITRA(B')	RITRA(B) ∧ RITRA(B')	RITRA(B) ∧ RITRA(B')	True

Table 7: Bisemigroup rules

	bFMinSets(O)	bFMinSetsOpUnion(O)	bAddOne(B)	bAddZero(B)
Prec.	LM(O), RM(O), AMS(O)	LM(O), RM(O), AMS(O)	CM(B$_\oplus$)	
LD	True	IDM(O) ∧ LND(O) ∧ RND(O)	LD(B) ∧ IDM(B$_\oplus$) ∧ (RI(B) ∨ ¬IDM(B$_\oplus$))	LD(B)
RD	True	IDM(O) ∧ LND(O) ∧ RND(O)	RD(B) ∧ IDM(B$_\oplus$) ∧ (LI(B) ∨ ¬IDM(B$_\oplus$))	RD(B)
PITA	True	*work-in-progress*	PITA(B)	True
PATI	*work-in-progress*	True	True	PATI(B)
RSS	False	False	RSS(B) ∧ LSI(B)	False
LSS	False	False	LSS(B) ∧ RSI(B)	False
RCEC	TT(O)	SL(O) ∧ (¬SL(O) ∨ (IAUS(O) ∧ IAF(O)))	RCEC(B)	SL(B$_\oplus$)
LCEC	TT(O)	SL(O) ∧ (¬SL(O) ∨ (IAUS(O) ∧ IAF(O)))	LCEC(B)	SL(B$_\oplus$)
RCC	RT(O) ∧ (¬RT(O) ∨ RMCC(O))	SL(O) ∧ (¬SL(O) ∨ (IAUS(O) ∧ IAF(O)))	SL(B$_\oplus$)	RCC(B)
LCC	LT(O) ∧ (¬LT(O) ∨ LMCC(O))	SL(O) ∧ (¬SL(O) ∨ (IAUS(O) ∧ IAF(O)))	SL(B$_\oplus$)	LCC(B)
LDT	False	False	False	False
RDT	False	False	False	False
LCP	LT(O) ∧ (¬LT(O) ∨ LMCC(O))	SL(O) ∧ (¬SL(O) ∨ (IAUS(O) ∧ IAF(O)))	SL(B$_\oplus$)	LCP(B)
RCP	RT(O) ∧ (¬RT(O) ∨ RMCC(O))	SL(O) ∧ (¬SL(O) ∨ (IAUS(O) ∧ IAF(O)))	SL(B$_\oplus$)	RCP(B)
RI	LND(O)	RND(O)	RI(B)	RI(B)
LI	RND(O)	RND(O)	LI(B)	LI(B)
RSI	False	False	False	False
LSI	False	False	False	False
RTID	False	False	False	False
LTID	False	False	False	False
PITLA	True	*work-in-progress*	PITLA(B)	True
RITRA	True	*work-in-progress*	RITRA(B)	True

Table 8: Bisemigroup rules

Author Index