PAAR-2010

Practical Aspects of Automated Reasoning

Boris Konev, Renate Schmidt, and Stephan Schulz (eds.)

2010

Programme Chairs

Boris Konev
Renate Schmidt
Stephan Schulz

Programme Committee

Peter Baumgartner
Roberto Bruttomesso
Koen Claessen
Bernd Fischer
Pascal Fontaine
Jim Grundy
Volker Haarslev
John Harrison
Thomas Hillenbrand
Joe Hurd
Gerwin Klein
Konstantin Korovin
Temur Kutsia
Daniel Le Berre
William McCune
Albert Oliveras
Jens Otten
Adam Pease
Silvio Ranise
Philipp Rümmer
Peter Schneider-Kamp
John Slaney
Alan Smaill
Mark Stickel
Geoff Sutcliffe
Josef Urban
Christoph Weidenbach
Leonardo de Moura
Hans de Nivelle

Workshop Website

`http://www.eprover.org/EVENTS/PAAR-2010/paar-2010.html`

Preface

The second Workshop on Practical Aspects of Automated Reasoning was held on July 14, 2010, in Edinburgh, UK, in association with the Federated Logic Conference (FLoC-2010).

PAAR provides a forum for developers of automated reasoning tools to discuss and compare different implementation techniques, and for users to discuss and communicate their applications and requirements. The workshop brought together different groups to concentrate on practical aspects of the implementation and application of automated reasoning tools. It allowed researchers to present their work in progress, and to discuss new implementation techniques and applications.

Topics included were:

- automated reasoning in classical and non-classical logics, implementation of provers;
- automated reasoning tools for all kinds of practical problems and applications;
- practical experiences, case studies, feasibility studies;
- evaluation of implementation techniques and automated reasoning tools;
- benchmarking approaches;
- non-standard approaches to automated reasoning, non-standard forms of automated reasoning, new applications;
- implementation techniques, optimisation techniques, strategies and heuristics;
- system descriptions and demos.

We were particularly interested in contributions that help the community to understand how to build useful and powerful reasoning systems in practice, and how to apply existing systems to real problems.

Invited Talks

Contributed Papers

Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers

Lawrence C. Paulson
Computer Laboratory
University of Cambridge, U.K.
lp15@cam.ac.uk

Abstract

Sledgehammer is a highly successful subsystem of Isabelle/HOL that calls automatic theorem provers to assist with interactive proof construction. It requires no user configuration: it can be invoked with a single mouse gesture at any point in a proof. It automatically finds relevant lemmas from all those currently available. An unusual aspect of its architecture is its use of unsound translations, coupled with its delivery of results as Isabelle/HOL proof scripts: its output cannot be trusted, but it does not need to be trusted. Sledgehammer works well with Isar structured proofs and allows beginners to prove challenging theorems.

1 Introduction

Interactive theorem provers are widely used by researchers for modelling complex algorithms or systems using logic. They typically support rich formalisms that include recursive definitions of functions and types; it may even be possible to reason about a partial recursive function's domain of definition or to define a relation co-inductively. The greatest weakness of these tools is the actual theorem proving, which is extremely laborious; perhaps they should instead be called specification editors.

For nearly 20 years, researchers have sought to make interactive theorem provers better at proving theorems. Much of this effort has been devoted to providing decision procedures. Certainly decision procedures are essential, especially for arithmetic; without them, obvious identities can take hours to prove, with them, complicated facts can be proved instantly. But most of the time, our problem lies beyond the scope of any standard decision procedure. Automatic tools are needed that can work on any type of problem.

Automatic theorem provers (ATPs) are capable of creating long, incomprehensible chains of deduction. Many researchers have attempted to use them to support interactive theorem proving; particularly pertinent are Ahrendt et al. [1], Bezem et al. [5], Hurd [7] and Siekmann et al. [26]. But the only one to pass the test of time is Sledgehammer [14, 21], which links Isabelle/HOL to the automatic provers E, SPASS and Vampire. Isabelle users invoke Sledgehammer routinely when undertaking difficult proofs. In a recent study involving older Isabelle proof scripts, Böhme and Nipkow demonstrated that Sledgehammer could prove 34% of the nontrivial goals contained in those proofs [6].

Sledgehammer was first released in February 2007 to users daring enough to download an Isabelle nightly build. It was announced in November 2007 as a component of Isabelle2007. This paper outlines the design goal that made Sledgehammer successful (§2). It describes some of the lessons learnt in the past three years and its effect on the way Isabelle/HOL is taught (§3). Avenues for research are also discussed (§4).

2 Design Principles

The single most important design goal was one-click invocation. Some earlier systems required the user to gather up all facts that could be relevant to the problem, and furthermore to reduce it to first-order

form. Problem preparation could easily take hours, with no guarantee that the call to a first order theorem prover would succeed. Such a tool would be of little value to users.

The two aspects of problem preparation (translation into first-order logic; identification of relevant facts) each required a substantial research effort. The numerous choices outlined below were made on the basis of innumerable experiments that consumed many thousands of hours of processor time.

2.1 Translation into First-Order Logic

Most interactive theorem provers support a language much richer than that of first-order logic. Isabelle/HOL [16] supports polymorphic higher-order logic, augmented with axiomatic type classes [32].[1] Many user problems contain no higher-order features, and might be imagined to lie within first-order logic; however, even these problems are full of typing information. Type information can take quadratic space [12] because every term must be labelled with its type, recursively, right down to the variables. Hurd [8] observed that omitting type information greatly improved the success rate of his theorem prover, Metis. This is hardly surprising, since the type information virtually buries the terms themselves. Hurd was able to omit type information because his proofs are reconstructed within the HOL4 system, which rejected any proofs that did not correspond to well-typed higher-order logic deductions.

Sledgehammer was always intended to rely on an analogous process of sound proof reconstruction, and from the outset it was clear that including complete type information would be unworkable. Completely omitting type information, although successful for Hurd, would not have worked for Isabelle because of its heavy use of type classes. Meng and I chose to include enough type information to enforce correct type class reasoning (the type class hierarchy is easily expressed using Horn clauses) but not to specify the type of every term [14, §4]. Colleagues have expressed horror at the very idea of using unsound translations; I have written a lengthy exploration of the salient issues [12, §2.8].

Although resolution theorem proving is based on clause form, most modern ATPs accept problems in first-order format. Sledgehammer nevertheless translates problems into clause form itself, and using a naive application of distributive laws rather than a polynomial time algorithm based on formula renaming [17]. Moreover, the translation to clauses is performed using Isabelle's internal proof engine; this was thought to be essential to allow proof reconstruction within Isabelle. Sledgehammer's naive translation algorithm has caused real difficulties. Its exponential worst-case behaviour can be triggered by real-world examples. To prevent the naive translation from generating prohibitively many clauses, an arbitrary cut-off (currently 60) had to be introduced. Using first-order format could improve the success rate by exploiting the superior translation technology built into modern ATPs.

Higher-order problems posed special difficulties. We never expected first-order theorem provers to be capable of performing deep higher-order reasoning, but merely hoped to automate proofs where the higher-order steps were trivial. We examined several methods of translating higher-order problems into first-order logic, allowing for at least truth values to be used as the values of terms and for curried functions taking varying numbers of arguments [12]. We eventually adopted a translation based on the one that we used for first-order logic, modified to introduce higher-order mechanisms (such as an "apply operator" for function values) only when absolutely necessary. We thereby eliminated our original distinction between first-order and higher-order problems. A higher-order feature within a problem affects the translation locally, giving a smooth transition from purely first-order to heavily higher-order problems.

We also experimented with two methods of eliminating λ-abstractions in terms: by translating them into combinator form or by declaring equivalent functions. We ultimately opted for combinators, using fairly naive translation scheme. (More sophisticated schemes delivered no benefits.) Experience

[1] Note that Isabelle/HOL is the instantiation of Isabelle [20] to higher-order logic. Isabelle is a generic theorem prover, based on a logical framework [19].

suggests, unfortunately, that Sledgehammer is seldom successful on problems containing higher-order elements. Integration with a genuine higher-order automatic theorem prover, such as LEO-II [3], seems necessary. This would pose interesting problems for proof reconstruction: LEO-II's approach is to reduce higher-order problems to first-order ones by repeatedly applying specialised inference rules and then calling first-order ATPs. A LEO-II proof will therefore consist of a string of higher-order steps followed by a first-order proof. The latter part we know how to do; the crucial challenge is to devise a reliable way of emulating the higher-order steps within Isabelle.

Arithmetic remains an issue. A purely arithmetic problem can be solved using decision procedures, but what about problems that combine arithmetic with a significant amount of logic? In principle, Sledgehammer could solve such problems with the help of an ATP that combined arithmetic and logical reasoning, analogous to LEO-II's approach to higher-order logic. Current SMT solvers are probably of little value, because they do not handle quantified formulas well. But progress in that field is extremely rapid, and soon this option could become attractive.

2.2 Relevance Filtering

Our initial goals for Sledgehammer were modest: to improve upon Isabelle's built-in automatic tools, using only the lemma libraries used by those tools. There were two libraries, one consisting of facts useful for forward and backward chaining, the other consisting of rewriting rules for simplification. Each library contained hundreds of lemmas. Meng and I [13] discovered that automatic theorem provers could solve only trivial problems in the presence so many extraneous facts; by developing a lightweight, symbol-based relevance filter, we greatly improved the success rate. Users would still have to identify relevant facts that did not belong to these lemma libraries.

Tobias Nipkow made the crucial suggestion to dispense with the lemma libraries, substituting the full collection of Isabelle theorems (around 7000). This idea offered the enticing prospect that any relevant existing theorem, however obscure, could be located. I thought this goal to be unrealistic; it seemed to have too much in common with McAllester's Ontic system [9]. Ontic was intended to be able to prove mathematical results using known results that it identified automatically, and it seems fair to say that this objective was too ambitious. But Sledgehammer would only be part of the system rather than all-encompassing, and it could take advantage of 20 years of increasing hardware performance.

We were able to scale up the relevance filter to cope with the 20-fold increase in the number of facts to process. However, it relies on ad-hoc heuristics that sometimes deliver poor results. Briefly, it assigns a score to every available theorem based upon how many constants that theorem shares with the conjecture; this process iterates to include theorems relevant to those just accepted, but with a decay factor to ensure termination. The constants are weighted to give unusual ones greater significance. The relevance filter copes best when the statement to be proved contains some unusual constants; if all the constants are common then it is unable to discriminate among the hundreds of facts that are picked up. The relevance filter is also memoryless: it has no information about how many times a particular fact has been used in a proof, and it cannot learn.

It would obviously be preferable for the automatic theorem provers themselves to perform relevance filtering. Or we should use a sophisticated system based on machine learning, such as Josef Urban's MaLARea [28], where successful proofs provide information to guide other proofs. Unfortunately, any such approach will fail given Sledgehammer's use of unsound translations. In unpublished work by Urban, MaLARea easily proved the full Sledgehammer test suite by identifying an inconsistency in the translated lemma library; once MaLARea had found the inconsistency in one proof, it easily found it in all the others. Sledgehammer is successful only because its relevance filter generally selects too few lemmas to produce an inconsistent axiom set, even with the unsound translations.

To accomplish better relevance filtering, we must decide whether to adopt a general first-order ap-

proach or to build a sophisticated relevance filter directly into Isabelle. The former approach could take advantage of the efforts of the entire ATP community, but it would have to be good enough to cope with soundly translated (and presumably enormous) formulas. The latter approach would avoid translation issues, but it would impose the entire effort onto a few Isabelle developers.

2.3 Parallelism

Parallelism was another design objective, both to exploit the abundance of cheap processing power and so that users would not have to wait. Sledgehammer was intended to run in the background; Isabelle would continue to respond to commands, and users could keep working. This idea has turned out to be misconceived: thinking is difficult, and when users hope that a proof might be found for them, they stop and wait for Sledgehammer to report back. We do hope that eventually Sledgehammer will be configured to run spontaneously, without even the need for a mouse click. Then users will simply work and occasionally be delighted to have solutions displayed for them. Such a configuration would require a machine with enough processing power to support several ATP executions without becoming sluggish. An agent-based implementation of similar ideas, using a blackboard architecture, has for some time been part of the Ωmega system [4, 26].

The parallel invocation of different theorem provers is invaluable. Böhme and Nipkow [6] have demonstrated that running three different theorem provers (E [25], SPASS [30] and Vampire [22]) for five seconds solves as many problems as running the best theorem prover (Vampire) for two full minutes. It would be better to utilise even more theorem provers. I have undertaken informal, unpublished experiments involving many other systems.

- Gandalf [27] shows great potential, but unfortunately it does not output useful proofs; one cannot easily identify which axioms have taken part in the proof. A simple source code modification to improve the legibility of proofs would allow Gandalf to make useful contributions. Unfortunately, I was unable to identify the necessary changes. Gandalf has been found to be unsound,[2] but a small percentage of incorrect proofs would be tolerable.

- People sometimes suggest that we include Prover9 [10]. In my experiments, Prover9 performed poorly on the large problems generated by Sledgehammer. It could be effective in conjunction with an advanced and selective relevance filter.

- Another possibility is to run multiple instances of a theorem prover with different heuristics. This is not necessary with Vampire, which attempts a variety of heuristics in separate time slices. It could be particularly effective with the E prover, but designing suitable heuristics requires highly specialised skills.

2.4 Proof Reconstruction

Isabelle subscribes to the LCF philosophy: all proofs ultimately reduce to primitives executed by a logical kernel. Isabelle users would not trust a tool that uncritically accepted proofs from an external source (especially one coded in C++!). But Sledgehammer had an even stronger design objective: to deliver Isabelle proofs in source form. We envisaged that Sledgehammer runs would demand substantial computational resources; if somebody used Sledgehammer many times while constructing a proof, would it be feasible to run that proof again, perhaps to modify it using a laptop while at a conference? To be useful, Sledgehammer would have to return a piece of proof script that could be executed cheaply.

[2] See `http://www.cs.miami.edu/~tptp/TPTP/BustedAsUnsound.html`

2.4.1 Reconstruction of the Resolution Proof

The original plan was to emulate the inference rules of automatic theorem provers directly within Isabelle. We should have known better: Hurd [7] had noticed that the proofs delivered by automatic theorem provers (Gandalf, in his case) were not detailed and explicit enough. We made the same discovery [14] (in the case of SPASS), and despite considerable efforts, were only able to reconstruct a handful of proofs.

We came up with a new plan: to use a general theorem prover, Metis, to reconstruct each proof step. Metis was designed to be interfaced with LCF-style interactive theorem provers, specifically HOL4. Integrating it with Isabelle's proof kernel required significant effort [21]. Metis then became available to Isabelle users, and it turned out to be capable of reconstructing proof steps easily. The output of Sledgehammer was now a list of calls to Metis, each of which proved a clause. This approach was inspired by the Otterfier proof transformation service [33].

Resolution proofs should ideally be translated to natural, intuitive Isabelle proofs. The best-known prior work on translating resolution proofs is TRAMP [11]; its applicability to Sledgehammer is unexplored. Preliminary work has commenced at Munich to see to what extent resolution proofs can be transformed into intelligible proofs.

2.4.2 One-Line Reconstruction, or ATPs as Relevance Filters

Having Metis available made possible an entirely different approach to proof reconstruction: to throw the proof away and allow Metis to find its own proof, using the lemmas that took part in the original resolution proof [21]. With this approach, Sledgehammer became merely a lemma finder, one that used automatic theorem provers merely as relevance filters. But this approach was generally effective, and had the great advantage that each Sledgehammer call now delivered a one-line result, rather than a lengthy and incomprehensible proof script in which all formulas were in clause form. At least one ATP implementer expressed disbelief that his system could be used merely as a relevance filter, but this approach allows any ATP to be used with Sledgehammer provided it returns the list of axioms used in its proof.

Metis sometimes fails to reconstruct the result of a Sledgehammer call. (Böhme and Nipkow [6] present detailed statistics.) Reconstruction necessarily fails if the resolution proof did not correspond to a well-typed Isabelle proof (recall that, normally, Sledgehammer omits most type information when translating Isabelle formulas into first-order logic). This type of failure could be eliminated by using sound translations, but the overall success rate would actually decrease considerably. The seasoned Sledgehammer user eventually learns to recognise unsound proofs (certain lemmas always seem to be mentioned). The number of such proofs could perhaps be reduced by ad hoc measures, such as removing those lemmas from the scope of Sledgehammer, possibly even with the help of machine learning.

Unfortunately, sometimes Metis is simply not powerful enough to prove a theorem that has already been proved by a more powerful system, despite being given a small list of axioms. ATPs frequently use many more axioms than are strictly necessary. The minimization tool developed by Philipp Meyer at TUM takes a set of axioms returned by a given ATP and repeatedly calls the same ATP with subsets of those axioms in order to find a minimal set. Reducing the number of axioms improves Metis's success rate, while also removing superfluous clutter from the proof scripts. ATPs themselves could return proofs using a minimum of axioms, or alternatively, proofs of a minimum length. Vampire's well-known limited resource strategy [23], although designed to cope with limited processor time, could probably be modified to minimise proofs efficiently.

3 Sledgehammer and Teaching

Sledgehammer was not designed specifically as an aid to novices. Experienced users have come to rely on it. But Sledgehammer seems to offer the greatest benefits to the least experienced users. It has certainly transformed the way Isabelle is taught. There are two reasons for this:

- Because it identifies relevant facts, users no longer need to memorise lemma libraries.
- Because it works in harmony with Isar structured proofs, users no longer need to learn many low-level tactics.

Demonstrations of interactive theorem provers necessarily involve deception. The implementers naturally want to show off their system in the best possible light, so they present examples that look more difficult than they really are. Typically they define some recursive functions and prove properties using obvious inductions followed by some sort of auto-tactic. The audience will be duly impressed, and some among them will decide to adopt that tool as the basis for their Ph.D. research. Too late, they encounter the crucial issue:

What do I do when the auto-tactic fails?

Typically, the answer is that one must write incomprehensible scripts that invoke a plethora of obscure commands. These generally include tactics to manipulate the set of assumptions in natural deduction or a sequent calculus. There may be tactics to transform an assumption by applying a rewrite rule or theorem, or to create a case split from an assumption. Some tactics substitute user-supplied terms into theorems.

In Isabelle, the simple combination of structured proofs and Sledgehammer takes the user surprisingly far. This is not the place to give a detailed tutorial on Isar structured proofs [15, 31]. In brief, they support natural deduction through local scopes that can introduce assumptions (using the keyword **assume**) as well as local variables and definitions. Moreover, while traditional tactic scripts contain only commands, a structured proof explicitly states the assumptions and goals. When proving a proposition, you can state intermediate properties that you believe to be helpful. If you understand the problem well enough to propose some intermediate properties, then all you have to do is state a progression of properties in small enough gaps for Sledgehammer to be able to prove each one.

```
proof -
  assume x: "x ∈ lambda_system M f"
  hence "x ⊆ space M"
    by (metis sets_into_space lambda_system_sets)
  hence "space M - (space M - x) = x"
    by (metis double_diff equalityE)
  thus "space M - x ∈ lambda_system M f" using x
    by (force simp add: lambda_system_def)
qed
```

Figure 1: A Structured Proof Completed with the Help of Sledgehammer

Figure 1 presents an example, where two intermediate facts (introduced by the keyword **hence**) assist in the proof of the conclusion (introduced by **thus**). Each of the intermediate facts is proved by a call to Metis that was generated using Sledgehammer. Nor need we restrict ourselves to a linear progression of facts. Because proofs are structured, you can nest the proofs of these lemmas to any depth.

Isar also supports calculational reasoning [2]. A chain of reasoning steps, connected by familiar relations such as $=$, $\leq$ and $<$, can be written with separate proofs for each step of the calculation. Once

again, if you can see the intermediate stages of the transformation, then the proofs of each step can be found easily. Figure 2 presents an example of calculational reasoning, taken from a measure theory development.

```
have "f(u ∩ (x ∩ y)) + f(u - x ∩ y) =
     (f(u ∩ (x ∩ y)) + f(u ∩ y - x)) + f(u - y)"
  by (metis class_semiring.add_a ey)
also have "... =  (f((u ∩ y) ∩ x) + f(u ∩ y - x)) + f(u - y)"
  by (metis Int_commute Int_left_commute)
also have "... = f(u ∩ y) + f(u - y)" using fx Int y u
  by auto
also have "... = f u"
  by (metis fy u)
finally show "f(u ∩ (x ∩ y)) + f(u - x ∩ y) = f u" .
```

Figure 2: A Calculational Proof Completed with the Help of Sledgehammer

Top down proof development is greatly assisted by a trivial Isar feature: the ability to omit proofs. Where a proof is required, the user may simply insert the word **sorry**. Isabelle then regards the theorem as proved.[3] The user can then check that the newly introduced proposition indeed suffices to prove the next proposition in the development. A difficult proof can develop as a series of propositions, each initially "proved" using **sorry** but eventually using either Sledgehammer, another automated method, or a nested proof development of the same form. Progress in such a proof can be measured in terms of the difficulty of the propositions that lack real proofs. Although we can never be certain that a proof development can be completed until the very end, the ability to write **sorry** in place of a proof reduces the risk of discovering that a lemma is useless only after spending weeks proving it.

In January 2010, as part of its new MPhil. programme, the University of Cambridge offered a lecture course on Isabelle [18]. The course materials included almost no information about the low-level tactics that had been the mainstay of Isabelle proofs for nearly 20 years. Only two of the 12 lectures were devoted to Isar structured proofs, and they took a novel approach: rather than proceeding methodically through the Isar fundamentals, the lectures presented the outer skeleton of a proof, with crucial sections replaced by **sorry**. They described the idea of trying to eliminate each **sorry** using either Sledgehammer or some automatic tactic. Practical work submitted by the students later demonstrated that several of them had learned how to write complex, well-structured proofs. I was happy to reassure them that submitting work generated largely by Sledgehammer was by no means cheating!

I have taught Isabelle on a number of occasions, starting in the mid-1990s. Sledgehammer is obviously not the only thing to have changed in that expanse of time. The introduction of Isar, continual improvements to Isabelle's automated reasoners, and 15 years of Moore's Law have transformed the user experience. Interactive theorem proving has never been practical because it required far too much effort, even from highly specialised and experienced experts. For the first time, we can envisage the day when interactive theorem proving becomes straightforward enough to be adopted on a large scale.

4 Conclusions

Sledgehammer has been available for over three years, and in that time it has become an essential part of the Isabelle user's workflow. It is possibly the only interface between an interactive theorem prover

[3] The existence of **sorry** does not compromise Isabelle's soundness, because it is only permitted during interactive sessions. A theory file containing an occurrence of **sorry** may not be imported by another theory.

and automatic ones to achieve such popularity with users. It has transformed the way beginners perceive Isabelle.

Counterexample generators are worth mentioning at this point. Users waste much time attempting to prove statements that are not theorems. Tools that can refute invalid conjectures are every bit as helpful as those that prove theorems. Random testing is an obvious way to do this, but counterexample finding can also make use of automated deduction technology. An early example is refute, which uses a SAT solver to find counterexamples [29]; it is also a component of Isabelle.

Sledgehammer has a number of limitations, most of which open up suggestions for future work. The relevance filter is primitive, but an improved one will have to be part of Sledgehammer itself as long as unsound translations are used; only if a compact but sound translation is invented can we rely on automatic theorem provers doing relevance filtering for themselves. Unsound translations can be used safely because Sledgehammer does not trust the proofs that it receives from ATPs but merely uses them as hints to generate Isabelle proof scripts; proofs that violate Isabelle's typing rules are eliminated at this stage. The success rate for first-order problems might be improved by eliminating Sledgehammer's transformation to clause form, delegating that task to ATPs; the impact of such a change on proof reconstruction might be limited, since that is now done using Metis. Sledgehammer's performance on higher-order problems is unimpressive, and given the inherent difficulty of performing higher-order reasoning using first-order theorem provers, the way forward is to integrate Sledgehammer with an actual higher-order theorem prover, such as LEO-II [3]. Proof reconstruction would benefit from new ideas, especially so that it can deliver natural, intuitive proofs.

Acknowledgements

The Cambridge team that developed Sledgehammer included Jia Meng, Claire Quigley and Kong Woei Susanto. Sledgehammer has since been refined, improved and tested by the Isabelle team at T.U. Munich. Christoph Benzmüller and Jasmin Blanchette commented on this paper.

The research was supported by the Engineering and Physical Sciences Research Council [grant number GR/S57198/01], *Automation for Interactive Proof*.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integrating automated and interactive theorem proving. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction— A Basis for Applications*, volume II. Systems and Implementation Techniques, pages 97–116. Kluwer Academic Publishers, 1998.

[2] Gertrud Bauer and Markus Wenzel. Calculational reasoning revisited (an Isabelle/Isar experience). In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2001*, LNCS 2152, pages 75–90. Springer, 2001. Online at `http://link.springer.de/link/service/series/0558/tocs/t2152.htm`.

[3] Christoph Benzmüller, Lawrence C. Paulson, Frank Theiss, and Arnaud Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning — 4th International Joint Conference, IJCAR 2008*, LNAI 5195. Springer, 2008.

[4] Christoph Benzmüller and Volker Sorge. OANTS – an open approach at combining interactive and automated theorem proving. In Manfred Kerber and Michael Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 81–97. A. K. Peters, 2000.

[5] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automatic proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3-4):253–275, 2002.

[6] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In J. Giesl and R. Hähnle, editors, *Automated Reasoning (IJCAR 2010)*, LNCS 6173. Springer, 2010. In press.

[7] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, LNCS 1690, pages 311–321. Springer, 1999.

[8] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.

[9] David McAllester. Ontic: A knowledge representation system for mathematics. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, LNCS 310, pages 742–743. Springer, 1988.

[10] William McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/mace4/`.

[11] Andreas Meier. TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level (system description). In David McAllester, editor, *Automated Deduction — CADE-17 International Conference*, LNAI 1831, pages 460–464. Springer, 2000.

[12] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, January 2008.

[13] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[14] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.

[15] Tobias Nipkow. A tutorial introduction to structured Isar proofs. `http://isabelle.in.tum.de/dist/Isabelle/doc/isar-overview.pdf`.

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.

[17] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [24], chapter 6, pages 335–367.

[18] Lawrence C. Paulson. Interactive formal verification. `http://www.cl.cam.ac.uk/teaching/0910/L21/`. Lecture course materials.

[19] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[20] Lawrence C. Paulson. Tool support for logics of programs. In Manfred Broy, editor, *Mathematical Methods in Program Development: Summer School Marktoberdorf 1996*, NATO ASI Series F, pages 461–498. Springer, Published 1997.

[21] Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2007*, LNCS 4732, pages 232–245. Springer, 2007.

[22] Alexander Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning — First International Joint Conference, IJCAR 2001*, LNAI 2083, pages 376–380. Springer, 2001.

[23] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.*, 36(1-2):101–115, 2003.

[24] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science, 2001.

[25] Stephan Schulz. System description: E 0.81. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097, pages 223–228. Springer, 2004.

[26] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, Immanuel Normann, and Martin Pollet. Proof development with Ωmega: The irrationality of $\sqrt{2}$. In Fairouz Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, pages 271–314. Kluwer Academic Publishers, 2003.

[27] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[28] Josef Urban. MaLARea: a metasystem for automated reasoning in large theories. In Geoff Sutcliffe, Josef

Urban, and Schulz Schulz, editors, *ESARLT 2007: Empirically Successful Automated Reasoning in Large Theories*, volume 257 of *CEUR Workshop Proceedings*, 2007.

[29] Tjark Weber. Bounded model generation for Isabelle/HOL. *Electron. Notes Theor. Comput. Sci.*, 125(3):103–116, 2005.

[30] Christoph Weidenbach. Combining superposition, sorts and splitting. In Robinson and Voronkov [24], chapter 27, pages 1965–2013.

[31] Makarius Wenzel. Isabelle/Isar — a generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec*. University of Białystok, 2007. Studies in Logic, Grammar, and Rhetoric 10(23).

[32] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: TPHOLs '97*, LNCS 1275, pages 307–322. Springer, 1997.

[33] J. Zimmer, A. Meier, G. Sutcliffe, and Y. Zhang. Integrated proof transformation services. In C. Benzmüller and W. Windsteiger, editors, *Workshop on Computer-Supported Mathematical Theory Development, 2nd International Joint Conference on Automated Reasoning*, Electronic Notes in Theoretical Computer Science, 2004.

Vampire: the New Blood

Andrei Voronkov
School of Computer Science
University of Manchester, U.K.

The first version of the theorem prover Vampire was implemented 17 years ago. Vampire has always been known for its fast implementation, for example it has won 20 world cup winner titles in first-order theorem proving. In this talk we will overview the results of a complete re-design of Vampire made in the last 1.5 years. Vampire has become even faster and includes several new features that makes it an ideal tool for applications: reasoning with theories, generation of colored proofs, interpolant generation, and reasoning with very large theories.

The talk is based on joint work with Krystof Hoder and Laura Kovacs.

Automated Reasoning in the Simulation of Evolvable Systems

Djihed Afifi
University of Manchester
Manchester, U.K.
djihed@cs.man.ac.uk

David E. Rydeheard
University of Manchester
Manchester, U.K.
david@cs.man.ac.uk

Howard Barringer
University of Manchester
Manchester, U.K.
howard@cs.man.ac.uk

Abstract

We present a novel application of automated theorem proving for the logical simulation of *evolvable* systems. Modelled using a logical framework, these systems are built hierarchically from components where each component is specified as a first order theory and may have an associated supervisory component. The supervisory component monitors and possibly changes its associated component. The simulation of this framework makes intensive use of automated theory proving – when running a simulation, almost all computational steps are those of a theorem prover. We present this novel combination of a logical setting involving meta-level logics and large sets of formulae for system description, together with theorem proving requirements which involve often slowly changing specifications with the need for rapid assessment of deducibility and consistency. We illustrate how theorem provers are used using an evolvable extension of the blocks world and present a caching structure to reduce simulation times. We then evaluate the suitability of several theorem provers for this application.

1 Introduction

We present an application of automated theorem proving for the simulation of computational systems. The computational systems we consider are *evolvable*, i.e. may reconfigure their structure and programs at run-time. Examples include business process modelling [9], adaptive query processing over changing databases [7] and data structure repair [4]. In [1], a logical framework for describing such systems is introduced. The underlying logic of this framework allows us to build a simulation engine for executing system specifications. This engine uses automated theorem proving technology to determine the satisfiability of logical formula sets as well as the deducibility of a logical formula.

The architecture of evolvable systems that we employ allows the 'localisation' of monitoring and evolution. Components in a system may have 'supervisors' which are themselves components and which monitor their 'supervisees' and may evolve them if required. Such supervised components may be assembled hierarchically, with supervisors at each level of the hierarchy if necessary (cf. Archware [14]).

In the logical framework [1], the state of a component is a set of ground atomic formulae that describe the current properties of the system. An action operates on the state in a style similar to STRIPS [8], changing the formula set by adding and deleting formulae. Models consist of sets of interpretations of the theories of each component. This revision-based method of description, which is common in planning and other AI applications, should be contrasted with the use pre and post-conditions [10]. This revision-based approach leads to a simple mechanical execution process, which we employ to build a simulator. This mechanisation makes intensive use of automated theorem proving (ATP) technology. Some issues relating to the appropriateness of the technology are:

- Changing verification requirements: ordinary actions affect the state only, while evolutionary actions may involve changing a component theory as well as its state. Thus as a system runs, theorem proving takes place in a highly dynamical setting.
- Determining the applicability of actions: an executed action may have a set of logical formulae as preconditions. A valid application of an action requires that (a) the preconditions are derivable from the system state and the component theory, and (b) the resultant state is consistent.

- Handling meta-logics: the supervisor's state is at a *meta-level* to that of the supervisee allowing the supervisor to hold facts about the supervisee. the meta-level logic of the supervisor's state may hold facts about the supervisee. These facts need to be consistent in the supervisor-supervisee pairing at all times during execution.
- Verifiability for minimum models: The revision-based logic used in system specifications is based on a notion of 'minimum model' [1] and we require that deducibility is relative to this minimality requirement. This requirement is related to the notions of the closed world assumption and circumscription in AI [13].

The logic-based simulation imposes a range of requirements on any theorem prover including the ability to:

- handle large sets of formulae for realistic systems,
- determine satisfiability of a set of formulae and deducibility of formulae from a given set of axioms.
- run unassisted, as opposed to guided, to make it possible to run large simulations that generate a high number of proof calls possible.
- construct models, not just establish satisfiability of a given formula set. Models are used to instantiate free variables in formulae.
- extract 'support sets' of formulae for proofs: States are often large sets of formulae, but those required to establish a property may be a small subset (often various different minimal 'support sets' may exist). Identifying support sets can aid proof caching (see below).

We have experimented with several theorem proving systems for this application:

- Paradox - from Chalmers University, a model finder for first-order logic [2].
- iProver - from The University of Manchester, an instantiation-based prover [11].
- Vampire - from The University of Manchester, a very fast resolution-based first-order theorem prover [15].

One point about the dynamics of theorem proving invocation needs explanation: during simulation, most individual actions cause a small change to the system, with the occasional evolution action that changes and reconfigures a system. On the other hand, actions generate multiple proof obligations that may duplicate prover dispatches. The execution of an action may trigger up to seven proof obligations for some action types. Furthermore, the overhead of discharging proof obligations is significant. The simulation of a simple system eventually spends a substantial amount of time communicating theories and proof results to and from the theorem prover. However, by examining simulations, we note that most changes do not affect state consistency, and in many cases proof obligations are duplicated or are trivially resolved.

In order to reduce the overhead of theorem proving various forms of proof caching have been employed. In general terms, proof caching is expensive. However, our particular application enables relatively simple approaches to speed up proof matching. The result is the elimination of a significant number of proof calls and dramatic speed-ups of simulations. In most cases more than 60% of proof obligations are eliminated with a corresponding increase in performance.

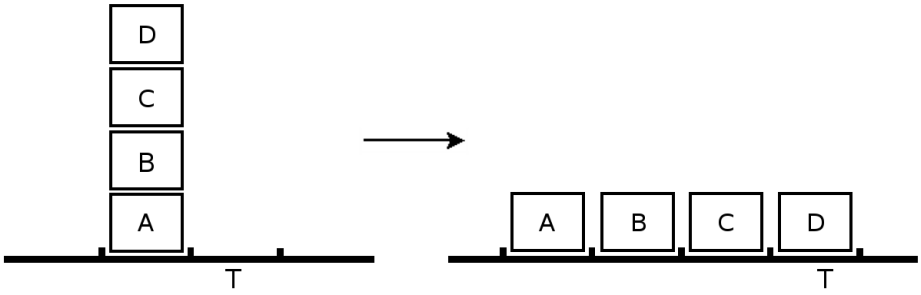


Figure 1: The scenario on the left shows the initial blocks world state, with a table having a capacity of 2 (i.e at most two blocks directly on the table) a tower of blocks. The scenario on the right shows a goal state that could be achieved by demolishing the tower and using a table of capacity 4.

This framework uses a non-standard type of inference relative to a minimum model determined by a set of 'observable formulae'. This is a generalisation of the notion that absence of ground atoms indicates falsehood. The requirement to reason in 'minimum models' [1] is handled in simple cases by a completion process in which we close observable predicates. The general case of reasoning in minimum models combines this completion process with deducibility, but is not required for the system specifications we consider here.

In the next section, using an example, we will illustrate how theorem provers are used during the simulation of an evolvable system, as well as the different between normal executions and evolutionary execution. In section 4 we describe a caching technique that eliminates a substantial number of proof dispatches. Finally, we discuss the results of experiments with several different types of theorem provers in section 4 and draw conclusions in section 5.

2 Logical Simulation of Evolvable Systems

We illustrate how theorem provers are used in our logic-based simulation through the following example of an evolvable blocks world [19]. This consists of a number of blocks which may be on each other or on a table. Actions may move blocks around the world. A theory for a blocks world is defined in Figure 2. This is a simplified theory for illustration purposes. A complete axiomatisation of the blocks world is given in [3] and is used in the simulation. We will show the simulation of a simple blocks world example and then illustrate how a supervisor may be introduced to overcome its limitations.

The blocks world theory *BlocksWorld* defines a finite set of blocks $\{A,B,C,D\}$ and a single table T. The 'BWC' axiom defines constraints on the predicate 'on'. The constraint 'TableSize' is a parametric formula that initially defines tables with a capacity of 2, i.e at most two blocks may be directly on the table. The state of a blocks world component is recorded using ground atoms built from the binary predicate 'on'. We distinguish between 'observation' predicates and 'abstraction' predicates. A state is described using only observation predicates. Only positive atoms of an observation predicate may be present in the state, those omitted are assumed to be false. The schema defines the abstraction predicate 'free' that will be used to deduce the availability of free space on the table. Abstraction predicates are defined using the constraints of the theory and may be deduced during simulation.

The component has the single action definition, 'move', that moves a block x from an object y to an object z. The action is conditional on the set 'pre'. The action is performed on the state in a revision-based manner by deleting the atoms in the set 'del' and adding the atoms in the set 'add'.

The component has an initial state where the blocks form a tower on top of the table as shown in

BlocksWorld

TYPES

$Blocks \stackrel{dfn}{=} \{A,B,C,D\}$

$Tables \stackrel{dfn}{=} \{T\}$

$Objects \stackrel{dfn}{=} Blocks \cup Tables$

OBSERVATION PREDICATES

$on : Blocks \times Objects$

ABSTRACTION PREDICATES

$free : Objects$

CONSTRAINTS

$BWC \stackrel{dfn}{=}$

$\forall b, b_1, b_2 : Blocks, o_1, o_2 : Objects \cdot$

$\neg on(b,b) \ \wedge$

$on(b,o_1) \wedge on(b,o_2) \Rightarrow (o_1 = o_2) \ \wedge$

$on(b_1,b_2) \Rightarrow (\exists o : Objects \cdot on(b_2,o))$

$\ldots$

$TableSize(T,2) \stackrel{dfn}{=}$

$(\exists b_1, b_2 : Blocks \cdot on(b_1,T) \wedge on(b_2,T) \wedge (b_1 \neq b_2)) \Leftrightarrow \neg free(T) \ \wedge$

$\forall b_1, b_2, b_3 : Blocks \cdot on(b_1,T) \wedge on(b_2,T) \wedge on(b_3,T) \Rightarrow ((b_1 = b_2) \vee (b_2 = b_3) \vee (b_1 = b_3))$

$BlockSize(1) \stackrel{dfn}{=}$

$\forall b : Blocks \cdot (\exists b_1 : Blocks, o : Objects \cdot on(b_1,b) \wedge on(b,o)) \Leftrightarrow \neg free(b) \ \wedge$

$\forall b_1, b_2 : Blocks \cdot on(b_1,b) \wedge on(b_2,b) \Rightarrow (b_1 = b_2)$

ACTIONS

$move(x : Blocks, y, z : Objects)$	
pre	$\{on(x,z), free(x), free(y)\}$
add	$\{on(x,y)\}$
del	$\{on(x,z)\}$

INITIAL STATE

$\{on(D,C), on(C,B), on(B,A), on(A,T)\}$

PROGRAM

$move(D,C,T); move(C,B,T); move(B,A,T)$

Figure 2: The Blocks World schema. Some axioms to assert the non-circularity of 'on' have been omitted from this paper for brevity. A full axiomatisation of the blocks world is given in [3].

Figure 1. The component is equipped with a program that demolishes the tower by moving all blocks on the table in turn. The program is a sequence of 'move' actions.

When this specification is executed, the first requirement is that it is *consistent*, meaning that its constraints and state together are consistent.

After checking the component's consistency, the component's program is run. The first action to be executed in the program is $move(D,C,T)$, which moves the the topmost block D in the initial state to the table T. The action move has the precondition set *pre* which checks that both the block and the receiving object are $free$. *Precondition checking* requires a theorem prover to establish deducibility of the preconditions from the state formulae and the component theory. If this is successful, the action revises the state by adding and deleting formulae as defined in the schema. The resultant state in turn needs to be checked for consistency with the component's theory to ensure that no action renders the

component inconsistent.

In order to continue demolishing the tower an attempt is made at the second action in the program, $move(C,B,T)$. This will fail because the precondition cannot be met. The precondition $free(T)$ cannot be deduced as the current blocks world theory is restricted to just towers of blocks on the table as defined in *TableSize*. The only way to change this is to alter the theory. To achieve this, we introduce another system that monitors this system and has the ability to change the blocks world specification. We refer to this new system as a supervisor.

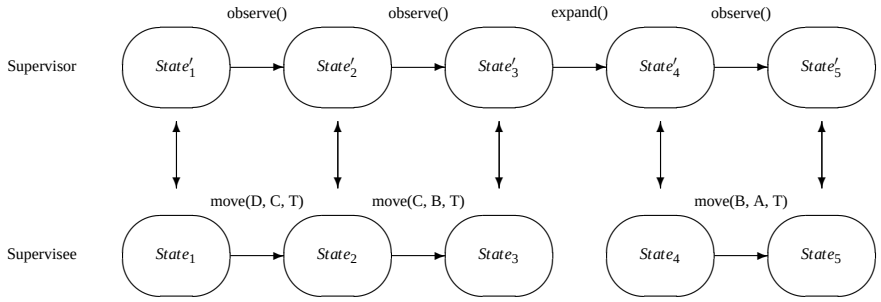


Figure 3: A paired execution trace

BlocksWorldSupervisor in Figure 4 presents such a supervisory system. It records properties of its *BlocksWorld* supervisee using meta-level predicates. The *holds* predicate is used to express properties of the supervisee. If $holds(\phi)$ occurs in the supervisor's state, then ϕ should be provable in the supervisee's state. The supervisor can use this predicate to query and monitor the supervisee. Similarly, the *constraint* predicate reflects the supervisee's constraints. The supervisor monitors the *BlocksWorld* using the action *observe* that tests formulae at the supervisee level. It also has the action *expand* which makes use of the evolution predicate *evolve*. The predicate *evolve* induces change at the supervisee level by revising the set of constraints that it has.

Using the *observe* action, the supervisor queries the state of the supervisee and detects when the table has an insufficient number of slots using the object level formula $free(T)$ as reflected at the supervisor level. This is a paired execution of the two example programs where the execution traces of the supervisor and the supervisee programs run in synchrony and are related by a meta-view relationship as depicted in Figure 3.

The supervisor monitors the supervisee and only intervenes when all all space on the table has been used, i.e when at the supervisee level $\neg free(T)$ holds. In this case, the *expand* action alters the *BlocksWorld* by replacing the *TableSize* constraint with one that allows for one more space. This is done using the *evolve* predicate, which introduces changes given by the supervisor to the supervisee. In this example, the change is the replacement of a constraint. The supervisor also has the ability to alter the state, redefine predicates or actions, or reconfigure the supervisee with a new component structure.

Meta-level conditions reflected by the predicate *holds* are checked by firing proof obligations using the supervisee's theory. The act of changing the supervisee's constraint using the *expand* action is called an *evolution* step. Theory consistency checks are necessary after performing the evolution for both components to disallow evolutions that lead to inconsistent theories.

Finally, component programs may have guarded instructions in which an instruction is executed only if its guard can be proved. We conclude this section by listing the cases where a theorem prover is invoked during the simulation of a component:

1. **Consistency** in the theory and state of each component, checked before and after each action. Models constructed in consistency checking may also be required.

BlocksWorldSupervisor META TO *BlocksWorld*

TYPES
$ConfigName$

FUNCTIONS
$s : ConfigName \rightarrow ConfigName$

OBSERVATION PREDICATES
$current : ConfigName$
$holds$: FORMULA $\times ConfigName$
$constraint$: CONSTRAINTNAME
$evolve$: ATOMS $\times$ ATOMS $\times$ CONSTRAINTNAMES $\times$ CONSTRAINTNAMES $\times ConfigName$

CONSTRAINTS
...

ACTIONS

$observe(P :$ FORMULAE$)$	
pre	$\{current(c)\}$
add	$\{holds(p, s(c)) \mid p \in P\} \cup \{current(s(c))\}$
del	$\{current(c)\}$

$expand(n : Int)$	
pre	$\{current(c), constraint(TableSize(T,m)), m < n\}$
add	$\{current(s(c)), holds(free(T), s(c)),$ $evolve(\{\},\{\},$ $\{TableSize(T,n)\},\{TableSize(T,m)\},$ $\{\},\{\},s(c)),$ $constraint(TableSize(T,n))\}$
del	$\{current(c), constraint(TableSize(T,m))\}$

INITIAL STATE
$\{current(c_0)\}$

Figure 4: The Blocks World Supervisor schema

2. **Precondition testing** by proving action preconditions.

3. **Guarded choice checking** by proving the guards.

4. **Meta-level checking** that the supervisee's reflection in the supervisor is correct, done before and after each evolution.

5. **Post-evolution consistency**, checking that evolutions don't produce inconsistent specifications.

In the above list, items 1 and 5 invoke theorem provers for satisfiability checking, while items 2, 3 and 4 invoke derivability checks.

2.1 System Simulation

In the example, the pairing of *BlocksWorldSupervisor* and *BlocksWorld* specifications will, when executed, ensure the complete demolition of the tower. The supervisor intervenes repeatedly to evolve the

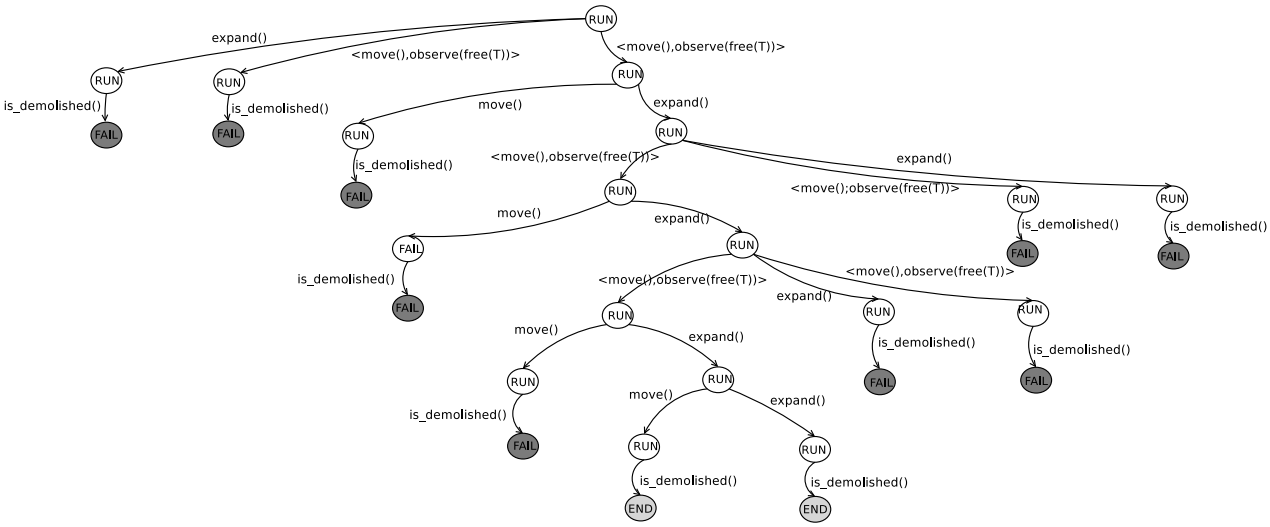


Figure 5: A complete run of the blocks world

supervisee and expand the table whenever it cannot accommodate any more blocks. To achieve this, the supervisor is equipped with the following program:

$$([\ |observe(free(T))\ |(observe(\neg free(T));expand())\ ])*;$$
$$is\_demolished();$$

This program is an iteration of an non-deterministic choice instruction that queries whether the table is free after every supervisee move action. The *expand* action is only performed when the table is observed to be not free. The *is_demolished* action performs a logical check to determine whether the demolition objective was successfully reached.

A complete run of this paired simulation will result in a configuration trace tree as shown in figure 5. Each node in this tree represents a state of the system configuration. This simulation is exhaustive as it looks for all possible runs of the pair of the components (this search can either be depth-first or breadth-first). The dark nodes indicate actions that have failed: this occurs when a supervisee's *move* action is not possible, or when an evolution is necessary. The grey nodes represent successful runs where the run successfully demolished the tower. Following the nodes in the trace tree, these successful runs occur when each block move is immediately preceded by an evolution to expand the table size.

3 Caching and Eliminating Proof Obligations

Simulating specifications can generate a substantial number of proof obligations. A basic non-supervisory action generates two proof obligations for every sub-component that it modifies, but an evolution action generates seven proof obligations for each pair of components that it affects. The simulation of the previous relatively simple blocks world generates 35 deducibility checks and 29 satisfiability checks. Firing an external theorem prover and communicating theories and results uses a substantial amount of time relative to the simulation execution time. Given that in any realistic simulation very large states will result and that the performance of a theorem prover is typically non-linear with the size of the state, it is

$$
\begin{array}{lcl}
\mathit{Cache} & \stackrel{dfn}{=} & \mathit{ComponentCacheKey} \rightarrow \mathit{ComponentCache} \\
\mathit{ComponentCacheKey} & \stackrel{dfn}{=} & \mathit{SchemaID} \times \mathit{ConstraintNames} \\
\mathit{ConstraintNames} & \stackrel{dfn}{=} & \mathit{ConstraintName}^* \\
\mathit{ComponentCache} & \stackrel{dfn}{=} & \mathit{State} \rightarrow \mathit{ProvenFormulae} \times \mathit{UnprovenFormulaeSets} \\
\mathit{State} & \stackrel{dfn}{=} & \mathit{Atom}^* \\
\mathit{ProvenFormulae} & \stackrel{dfn}{=} & \mathit{Formula}^* \\
\mathit{UnprovenFormulaeSet} & \stackrel{dfn}{=} & \mathit{FormulaeSet}^* \\
\mathit{FormulaeSet} & \stackrel{dfn}{=} & \mathit{Formula}^*
\end{array}
$$

Figure 6: Cache Structure

important to reduce the number of dispatched proof obligations. Several techniques can be used to optimise execution and discharge some of the proof obligations without the need to call an external theorem prover.

By examining simulations, we note that the most commonly executed actions are of a basic type, which are occasionally interrupted by the rarer large system reconfigurations that alter component theories or replace components. Most actions do not affect the axioms of a component and therefore do not affect its internal consistency. A substantial number of consistency checks can therefore be eliminated by storing the consistency results of previous prover invocations. A component is given a 'consistent' flag that is set once and is reset when actions changes are deemed to affect this consistency.

The state of a component contains only positive atoms of observation predicates. Therefore, in some cases, when preconditions are themselves ground observable atoms, derivability reduces to testing membership using symbolic equality. The minimum model interpretation means that the absence of an atom of an observation predicates indicates its falsehood. This allows us to reduce preconditions that rely heavily on observable predicates. It is often the case that the precondition set is reduced to either the empty set (i.e preconditions are met) or to $false$, eliminating the need for any external theorem prover.

3.1 Caching of Prover Results

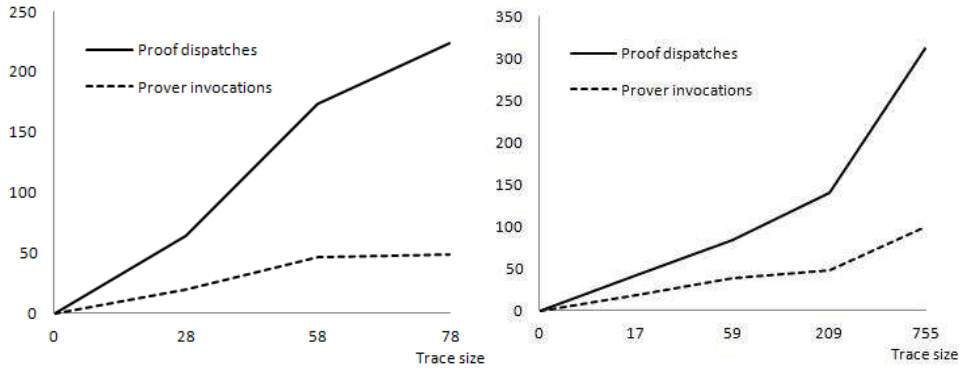
A cache structure suitable for this application is depicted in Figure 6. Properties of our framework make this cache structure relatively efficient.

The cache stores the prover invocation results for each component separately. To perform lookup, a key for the cache consists of the name of the component together with the names of its constraints. Only the signatures of the constraints are stored, e.g. $\mathit{TableSize}(T,2)$. This eliminates the need to store whole formulae and simplifies component cache lookup. An example of a cache key is

$$(\mathit{BlocksWorld}, \{\mathit{TableSize}(T,2), \mathit{BlockSize}(1)\})$$

Each component is then associated with all states that it may have had in the past. For each state that a component may have been in, the list of proved formulae is stored. Formulae lookups are done syntactically, so this is efficient for looking up previously proved ground atoms. The cache also stores the sets of formulae that were previously disproved given a state.

Looking up formulae in this cache is undertaken as follows. Given a proof obligation for the formulae set S, once a cache key has been matched together with a state, S is reduced to the set S' by eliminating formulae that were previously proved. If S' is the empty set then this is considered a cache hit as proved formulae and a prover invocation is not necessary. if S' is not empty, it is compared with every set of the previously unproven formulae. If S' contains any of these sets then this is also a cache hit as unproven



(a) Blocks World Example (b) ATM Example

Figure 7: Prover invocation elimination using the caching technique

formulae. If S' is not matched by any unproven formulae set, an external prover invocation is necessary to determine the deducibility of S'. If the prover returns a proved result then S' is merged with the proved formulae in the cache. if S' is not proved it is added to the set of previously unproven formulae sets.

3.2 Performance

The caching technique eliminates a substantial number of proof obligations. In the blocks world example, 20 out of 35 deduction requests (57%) and 24 out of 29 satisfiability requests (82%) are eliminated. The running time is reduced from 16 seconds to 5 seconds on an AMD Athlon 2000+ processor with 1GB of RAM. Figure 7 shows the number of total proof dispatches and actual external prover invocations for specifications that generate increasing program traces. The ATM example, given in [1], models an evolvable banking and automated teller machines system that enforces several layers of security using evolvable pairs. The trace size is an indication of the number of actions being performed. The figure shows that the longer the simulation the more beneficial caching becomes, with some simulations eliminating over 90% of all proof obligations.

4 Comparing Theorem Provers

The simulator converts theories to classical untyped first order logic in the TPTP3 [18] format so CASC (CADE ATP System Competition) [17] can be used. Three provers were used in this study:

- **Paradox** [2]: a finite model generator that flattens first order logic formulae into proposition clauses, then uses the MiniSat [6] SAT solver to solve the resulting problem.
- **Vampire 10** [15]: a fast resolution-based theorem prover.
- **iProver** [11]: an instantiation based prover that combines first order reasoning with a SAT solver (using MiniSat [6]).

The table in Figure 8 shows the running time in seconds of the simulation using these provers. The measure of the complexity of a simulation is the trace size, which grows approximately linearly with the number of proof obligations being made.

For the blocks world example, Paradox was able to undertake both the satisfiability checks and the proofs. It is fast at finding models for axiom sets and for checking counter satisfiability [16]. Overall,

Trace size	Paradox	iProver	Vampire
28	5	12	15
58	8	18	22
78	11	24	32

Figure 8: Simulation time (in seconds) using different provers. Note that satisfiability checking was done with Paradox when using Vampire.

it was the best performer of the three theorem provers for this application. Its ability to determine the deducibility or otherwise the counter satisfiability of formulae makes it the most suitable choice for this application.

Although Vampire's resolution is fast, it is not appropriate to establish the counter satisfiability of non-theorems, nor could it establish the satisfiability of theories in the above example. This necessitates the use of other theorem provers for these purposes when using Vampire.

iProver is unique because it can do both resolution reasoning and SAT checking, but they are done successively with manual options to turn off either features. Its resolution reasoning is fast at establishing the non-satisfiability of sets of formulae and for deriving theorems, while its SAT solving mode is fast at establishing the satisfiability of sets of axioms and the counter-satisfiability of non-theorems. iProver can spend time unnecessarily using one of its modes for an input that is best suited for the other mode.

5 Conclusion

This paper gave an overview of the practical aspects of using theorem provers for the simulation of the evolvable systems framework presented in [1]. This framework differs from rewriting tools such as Maude [12] in the fact that it allows for a supervisory model that improves usability and the separation of concerns [5]. In the simulation, theorem provers are used to deduce formulae from an axiom set and for establishing the satisfiability of sets of formulae. The simulation generates a large number of proof obligations. However, a caching technique was used to eliminate a substantial number of prover invocations and speed up simulation.

Three theorem provers were used in this study: Paradox [2], iProver [11] and Vampire [15]. Paradox's model finding was best suited for establishing satisfiability of axiom sets as well as the counter satisfiability of non-theorems. Although vampire's resolution is fast, it could not establish the counter-satisfiability of non-theorems. iProver's manually changeable two modes, resolution and SAT solving, could perform both tasks. In the future, running multiple theorem provers in parallel could be performed to exploit the strength of each prover.

References

[1] H. Barringer, D. Gabbay, and D. Rydeheard. Modelling evolvable component systems: Part I: A logical framework. *Logic Jnl IGPL*, 17(6):631–696, 2009.

[2] K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.

[3] S.A. Cook and Y. Liu. A complete axiomatization for blocks world. *Journal of Logic and Computation*, 13(4):581, 2003.

[4] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 176–185, 2005.

[5] E.W. Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[6] N. Eén and N. Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, pages 502–518, 2003.

[7] K. Eurviriyanukul, A. Fernandes, and N. Paton. A foundation for the replacement of pipelined physical join operators in adaptive query processing. *Current Trends in Database Technology–EDBT 2006*, pages 589–600.

[8] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[9] R.M. Greenwood, I. Robertson, B.C. Warboys, and B.S. Yeomans. An evolutionary approach to process system development. In *Proceedings of the International Process Technology Workshop, Villard de Lans (Grenoble)*. Citeseer, 1999.

[10] C. B. Jones. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice-Hall, 1990.

[11] K. Korovin. iProver - an instantiation-based theorem prover for first-order logic (system description). In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.

[12] P. Lincoln M. Clavel, S. Eker and J. Meseguer. Principles of maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[13] J. McCarthy. Circumscription–a form of non-monotonic reasoning. *Artificial intelligence*, 13(1-2):27–39, 1980.

[14] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. ArchWare: Architecting Evolvable Software. In *Proceedings of the 1st European Workshop on Software Architecture (EWSA'04), European Projects in Software Architecture - Invited Paper*, LNCS, pages 257–271, St Andrews, UK, May 2004.

[15] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.

[16] G. Sutcliffe. The 4th IJCAR Automated Theorem Proving Competition. *AI Communications*, 22(1):59–72, 2009.

[17] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[18] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[19] T. Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.

Progress in Automating Higher-Order Ontology Reasoning*

Christoph Benzmüller
Articulate Software, CA, USA
cbenzmueller@articulatesoftware.com

Adam Pease
Articulate Software, CA, USA
apease@articulatesoftware.com

Abstract

We report on the application of higher-order automated theorem proving in ontology reasoning. Concretely, we have integrated the Sigma knowledge engineering environment and the Suggested Upper-Level Ontology (SUMO) with the higher-order theorem prover LEO-II. The basis for this integration is a translation from SUMO's SUO-KIF representations into the new typed higher-order form representation language TPTP THF. We illustrate the benefits of our integration with examples, report on experiments and analyze open challenges.

1 Introduction

In recent years much progress has been made regarding applications of first-order automated theorem provers (FO-ATP) in ontology reasoning and question answering. A prominent example is the application of FO-ATPs to the Suggested Upper-Level Ontology (SUMO) [22]. Related work has also been reported for Cyc [24] and the LogAnswer question answering system [13], and further related references are given in [13]. In all those approaches, translations from the ontology representation languages into proper first-order representations are employed.

Challenges that have been identified in this application context, amongst others, include the large theories challenge and answer extraction problem. The former addresses the problem that the proofs are often shallow while the axiom sets are usually huge and may contain lots of information irrelevant to a given query. The latter deals with the issue of extracting single or multiple answers to queries from prover output.

Another challenge, which is in the focus of this paper, is that knowledge bases such as SUMO contain a small but significant amount of higher-order representations. SUMO, for example, began as just an upper level ontology encoded in first-order logic but subsequently its logic has been expanded to also include higher-order elements.

The approach taken in the above systems to reason with higher-order content is to employ specific translation 'tricks', possibly in combination with or in addition to some pre-processing techniques. An example is the quoting technique for embedded formulas as employed in the Sigma knowledge engineering environment [22]. Unfortunately, however, this solution is strongly limited. The effect is that many desirable inferences are currently not supported, so that many queries cannot be answered. Illustrating examples are presented in this paper and a solution is proposed that employs higher-order automated theorem proving (HO-ATP) for the task. Our solution exploits the new TPTP infrastructure for higher-order automated theorem proving [27] and provides a generic translation from the Standard Upper Ontology Knowledge Interchange Format (SUO-KIF) [20] into the new typed higher-order form (THF) language of the TPTP.

This paper is structured as follows. In Section 2 we briefly sketch the background of our work. Section 3 further motivates it with examples. Our translation from the SUO-KIF to TPTP THF is presented in Section 4. Implementation, system integration and experiments are reported in Section 5. The paper ends with a discussion of open challenges and future work.

*This work was funded by the German Research Foundation (DFG) under grant BE 2501/6-1.

2 Background

SUMO [17] is an open source[1], formal ontology. In addition to the expressive logic it was authored in, it has also been translated into the OWL semantic web language. It has undergone ten years of development, review by a community of hundreds of people, and application in expert reasoning, linguistics and performance testing for theorem provers. SUMO has been subjected to partial formal verification with automated theorem provers. This consisted of asking a theorem prover to prove the negation of each axiom in the knowledge base. While necessarily incomplete, this did focus the attention of the prover with more success than simply asking it to prove "false". With repeated testing on incrementally more generous time allotments, this method caught a number of non-obvious contradictions. It has been one method of many partial methods to ensure quality and consistency.

SUMO covers areas of knowledge such as temporal and spatial representation, units and measures, processes, events, actions, and obligations. SUMO has been extended with a number of domain specific ontologies, which are also public, together they number some 20,000 terms and 70,000 axioms. Domain specific ontologies extend and reuse SUMO, for example, in the areas of finance and investment, country almanac information, terrain modeling, distributed computing, and biological viruses. SUMO has also been mapped by hand [18] to the entire WordNet lexicon of approximately 100,000 noun, verb, adjective and adverb word senses, which not only acts as a check on coverage and completeness, but also provides a basis for application to natural language understanding tasks.

SUMO has natural language generation templates and a multi-lingual lexicon that allows statements in SUMO to be automatically paraphrased in multiple natural languages.

The formal language of SUMO is SUO-KIF, a simplified version of the original KIF [14], with extensions for higher-order logic. Since SUO-KIF syntax is rather self-explaining we avoid a formal introduction here and provide some explanations on the fly. For further details we refer to [20].

Sigma [21] is a browsing and inference system that is both a stand-alone system for ontology development and an embeddable component for reasoning. We have also developed a set of optimizations that improve the performance of reasoning on SUMO, typically by "trading space for time" — pre-computing certain inferences and storing them in the knowledge base [21]. In many cases this results in speedups of several orders of magnitude. While Sigma originally included only the Vampire prover for performing logical inference on SUMO, it now embeds the TPTPWorld environment [29], giving it access to some 40 different systems, including the world's most powerful automated theorem provers and model generators. Sigma now also integrates the SInE reasoner [16], which was the winner of the SUMO division of the CASC international theorem proving competition [23]. Use of the SInE axiom selection system has been shown to provide orders of magnitude improvements in theorem proving performance compared to using top-performing theorem prover, such as E or Vampire, alone. By selecting its best guess at axioms relevant to a particular query, it can dramatically reduce the search space for solving queries on large knowledge bases, such as SUMO, where only a small number of axioms are likely to be relevant to any given query. Sigma handles making statements and posing queries to the different reasoners, optimizing the knowledge sent to them to support efficient inference, and handling their output, formatting answers and proofs in a standard and attractive format. Sigma includes a Java API and XML messaging interface.

HO-ATP is currently experiencing a renaissance that has been fostered by the recent extension of the successful TPTP infrastructure for first-order logic [26] to higher-order logic, called TPTP THF [27, 28]. Available HO-ATPs include LEO-II [10], TPS [2], IsabelleP, IsabelleM/N[2] and Satallax [3]. These systems are available online via the SystemOnTPTP tool [25], they support the TPTP THF infrastructure, and they employ THF0 [11], the simple type theory fragment of the THF language, as input language.

[1] `www.ontologyportal.org`

[2] IsabelleM and IsabelleN are model finders in the Isabelle proof assistant [19] that have been made available in batch mode, while IsabelleP applies a series of Isabelle proof tactics in batch mode.

3 Examples and Challenges

Our goal has been to enable and study applications of HO-ATP for question answering in ontology reasoning, exemplary in SUMO. In this section we present some motivating examples. They illustrate the potential of our approach to reason within temporal and other contexts. We also point to a problem regarding Boolean extensionality and epistemic modalities.

Embedded Formulas Embedded formulas are one prominent source of higher-order aspects in SUMO. This is illustrated by the following example, which has been adapted from [22]. (Premises are marked with **P** and the query is marked with **Q**. In SUMO variables always start with a '?'. Free variables in queries are implicitly existentially quantified and those in premises are implicitly universally quantified.)

Example 1 (During 2009 Mary liked Bill and Sue liked Bill. Who liked Bill in 2009?)**.**
P1: `(holdsDuring (YearFn 2009) (and (likes Mary Bill) (likes Sue Bill)))`
Q: `(holdsDuring (YearFn 2009) (likes ?X Bill))`

The challenge is to reason about the embedded formulas `(and (likes Mary Bill) (likes Sue Bill))` and `(likes ?X Bill)` within the context `(holdsDuring (YearFn 2009) ...)`. In our example, the embedded formula in the query does not match the embedded formula in the premise, however, it is inferable from it. The quoting technique presented in [22], which encodes embedded subformulas as strings, fails for this query. There are possible further 'tricks' though which could eventually be applied. For example, we could split **P1** in a pre-processing step into **P2:** `(holdsDuring (YearFn 2009) (likes Mary Bill))` and **P3:** `(holdsDuring (YearFn 2009) (likes Sue Bill))`. However, such means quickly reach their limits when considering more involved embedded reasoning problems. The following modifications of Example 1 illustrate the challenge.

Example 2 (Example 1 modified; 'and' reformulated)**.**
P4:
```
(holdsDuring (YearFn 2009)
                (not (or (not (likes Mary Bill)) (not (likes Sue Bill)))))
```
Q: `(holdsDuring (YearFn 2009) (likes ?X Bill))`

Example 3 (At all times Mary likes Bill. During 2009 Sue liked whomever Mary liked. Is there a year in which Sue has liked somebody?)**.**
P5: `(holdsDuring ?Y (likes Mary Bill))`
P6: `(holdsDuring (YearFn 2009) (forall (?X) (=> (likes Mary ?X) (likes Sue ?X))))`
Q: `(holdsDuring (YearFn ?Y) (likes Sue ?X))`

In particular, Example 3 illustrates that the reasoning tasks may indeed quickly become non-trivial for approaches based on translations to first-order logic. This example can be further modified as follows. Here we use a propositional variable `?P` in order to encode that what generally holds also holds in all `holdsDuring`-contexts.

Example 4 (What holds that holds at all times. Mary likes Bill. During 2009 Sue liked whomever Mary liked. Is there a year in which Sue has liked somebody?)**.**
P7: `(=> ?P (holdsDuring ?Y ?P))`
P8: `(likes Mary Bill)`
P9: `(holdsDuring (YearFn 2009) (forall (?X) (=> (likes Mary ?X) (likes Sue ?X))))`
Q: `(holdsDuring (YearFn ?Y) (likes Sue ?X))`

We may instead of **P7** express that true things hold at all times in an alternative way, cf. **P7'** below.[3]

[3] Instead of **P7'** we may equally well use e.g. **P7'':** `(holdsDuring ?Y (equal Chris Chris))` or any other embedded tautology.

Example 5 (Example 4 modified)**.**
P7': `(holdsDuring ?Y True)`
P8: `(likes Mary Bill)`
P9: `(holdsDuring (YearFn 2009) (forall (?X) (=> (likes Mary ?X) (likes Sue ?X))))`
Q: `(holdsDuring (YearFn ?Y) (likes Sue ?X))`

Some key steps of the informal argument for the latter query are: Since `True` is always valid and since we assume `(likes Mary Bill)` we know that these two formulas are equivalent. Hence, they are equal. We can thus replace `True` in `(holdsDuring ?Y True)` by `(likes Mary Bill)`. The remaining argument is straightforward.

Set abstraction Another important higher-order construct in SUMO is the set (or class) constructor `KappaFn`. It takes two arguments, a variable and a formula, and returns the set (or class) of things that satisfy the formula. We illustrate the use of `KappaFn` in Example 6.

Example 6 (The number of people John is grandparent of is less than or equal to three. How many grandchildren does John at most have?)**.**
P10: `(<=> (grandchild ?X ?Y) (exists (?Z) (and (parent ?Z ?X) (parent ?Y ?Z))))`
P11: `(<=> (grandparent ?X ?Y) (exists (?Z) (and (parent ?X ?Z) (parent ?Z ?Y))))`
P12: `(lessThanOrEqualTo (CardinalityFn (KappaFn ?X (grandparent John ?X))) 3)`
Q: `(lessThanOrEqualTo (CardinalityFn (KappaFn ?X (grandchild ?X John))) ?Y)`

The query can be proved valid independent of the specific axiomatization of `CardinalityFn`. This is because the embedded set abstractions can be shown equal.

Extensionality In the examples discussed so far we have assumed that the semantics of our logic is classical and that the Boolean and functional extensionality principles are valid. In particular Boolean extensionality, which says that two formulas `P` and `Q` are equal if and only if they are equivalent (or, alternatively, that there are not more than two truth values), is relevant for all of the examples above. Without it we could not even prove the following query since the denotations of the two embedded formulas could be different despite the equivalence of these formulas.

Example 7 (During 2009 Mary liked Bill and Sue liked Bill. Is it the case that in 2009 Sue liked Bill and Mary liked Bill?)**.**
P1: `(holdsDuring (YearFn 2009) (and (likes Mary Bill) (likes Sue Bill)))`
Q: `(holdsDuring (YearFn 2009) (and (likes Sue Bill) (likes Mary Bill)))`

Functional extensionality, which is required in Example 6 in combination with Boolean extensionality, has been discussed as an option for the semantics of KIF in [15]. The validity of Boolean extensionality has never been questioned though in the literature. Weakening it would require a semantics with more than two truth values and this is not considered an option, neither in [15] nor in [20]. For a detailed discussion of functional and Boolean extensionality in classical higher-order logic we refer to [7].

Modalities Challenge Assuming Boolean extensionality in the semantics of SUO-KIF seems perfectly fine for the above examples. We do not want to conceal, though, the following problem related to it. SUMO employs epistemic modalities, such as `believes` and `knows`. When used in combination with Boolean extensionality, however, inferences are enabled that do obviously contradict their intended meaning. We give an example that is very similar to Example 5; the main difference is that the temporal context has been replaced by an epistemic context.

Example 8 (Adapted Example 5 within epistemic context: Everybody knows that Chris is equal to Chris. Mary likes Bill. Chris knows that Sue likes whomever Mary likes. Does Chris know that Sue likes Bill?)**.**
P7": `(knows ?Y (equal Chris Chris))`
P8: `(likes Mary Bill)`
P9': `(knows Chris (forall (?X) (=> (likes Mary ?X) (likes Sue ?X)))`
Q: `(knows Chris (likes Sue Bill))`

Assuming Boolean extensionality the query is valid, even though we have not explicitly stated the fact `(knows Chris (likes Mary Bill))`. Intuitively, however, this assumption seems mandatory for enabling the proof of the query. Hence, we here (re-)discover an issue that some logicians possibly claim as widely known: modalities have to be treated with great care in classical, extensional higher-order logic. Our ongoing work therefore studies how we can suitably adapt the modeling of affected modalities in SUMO in order to appropriately address this issue.

Relation and Function Variables Generating suitable instantiations for relation or function variables is another prominent higher-order challenge. For instance, in the following query the relation `sib`, with `(sib ?X ?Y)` if and only if `(or (sister ?X ?Y) (brother ?X ?Y))))`, is a valid instantiation for the queried variable `?R`. (There are other instantiations possible for ?R in our example though and enumerating them is a challenge for future work.) Our example illustrates that the invention of new concepts like the notion of sibling from simpler notions like `brother` and `sister` is in principle feasible in higher-order logic, though there are clearly practical limitations.

Example 9 (Mary, Sue, Bill and Bob are mutually distinct. Mary is neither a sister of Sue nor of Bill, and Bob is not a brother of Mary. Sue is a sister of Bill and of Bob, and Bob is a brother of Bill. Is there a relation that holds both between Bob and Bill and between Sue and Bob; we exclude the trivial universal relation $\lambda X, Y.\top$)**.**
P13: `(and (not (equal Mary Sue)) (not (equal Mary Bill)) (not (equal Mary Bob)) (not (equal Sue Bill)) (not (equal Sue Bob)) (not (equal Bob Bill)))`
P14: `(and (not (sister Mary Sue)) (not (sister Mary Bill)) (not (brother Bob Mary)))`
P15: `(and (sister Sue Bill) (sister Sue Bob) (brother Bob Bill))`
Q: `(and (?R Bob Bill) (?R Sue Bob) (not (forall (?X ?Y) (?R ?X ?Y))))`

4 THF Translation

The main objective for our translation from SUMO to TPTP THF0 [11] has been to enable inferences as required for query examples as presented above.

THF0 provides a syntax for Church's simple type theory [1], that is, a classical logic built on top of the simply typed λ-calculus. The standard base types in simple type theory are o and ι; the former denotes the set of Booleans and the latter a (non-empty) set of individuals . They are represented in THF0 as `$i` and `$o`. Further base types can be declared as needed. Function types in THF0 are encoded with the >-constructor, e.g. the type of predicates (resp. sets) over type `$i` is denoted as `$i > $o`. THF0 files obey the convention that the types of constant symbols and variable symbols have to be declared before their first use. Type declarations for constant symbols are typically provided in a type signature part at the beginning of each THF0 file while types of variable symbols are provided in their binding positions.

In our translation of SUMO to THF0 we recursively analyze all SUMO terms and subterms with the aim of assigning consistent type information to them. From this process we then extract the assigned

type information for all constant and variable symbols as required in THF0 files. When applying our transformation procedure to **P12**, for example, we generate the following THF0 information:

```
%%% The extracted Signature %%%
thf(grandparent_THFTYPE_IiioI,type,(
    grandparent_THFTYPE_IiioI: $i > $i > $o )).

thf(lCardinalityFn_THFTYPE_IIioIiI,type,(
    lCardinalityFn_THFTYPE_IIioIiI: ( $i > $o ) > $i )).

thf(lJohn_THFTYPE_i,type,(
    lJohn_THFTYPE_i: $i )).

thf(ltet_THFTYPE_IiioI,type,(
    ltet_THFTYPE_IiioI: $i > $i > $o )).

thf(n3_THFTYPE_i,type,(
    n3_THFTYPE_i: $i )).

%%% The translated axioms %%%
thf(ax,axiom,
    ( ltet_THFTYPE_IiioI
    @ ( lCardinalityFn_THFTYPE_IIioIiI
      @ ^ [X: $i] :
          ( grandparent_THFTYPE_IiioI @ lJohn_THFTYPE_i @ X ) )
    @ n3_THFTYPE_i )).
```

This THF0 representation is, for obvious reasons, not intended for human consumption. It serves the sole purpose of communicating the reasoning problem to the higher-order theorem provers. We briefly sketch a few aspects: (i) So far, we use THF0 type `$i` as only base type other than `$o`; for example, SUMO formulas and sentences are mapped to type `$o` while constants such as `lJohn_THFTYPE_i` and `n3_THFTYPE_i`, which are the translations of the SUMO constants `John` and `3`, are currently both declared of type `$i`.[4] Function types, e.g. for `lCardinalityFn_THFTYPE_IIioIiI`, are determined by our translation algorithm. Future work includes the introduction of further base types in combination with a better exploitation of the rich typing information already available in SUMO. (ii) As expected, the simple type computed for `lCardinalityFn_THFTYPE_IIioIiI`[5], the translation of SUMO constant `CardinalityFn`, is `( $i > $o ) > $i`, that is, the arguments for this constant have to be sets of objects of type `$i`. (iii) `KappaFn` is mapped to λ-abstraction.

Assigning types to SUMO terms is in fact not as straightforward as this example might suggest. One major problem is that SUMO supports self-applications as in the following SUMO axiom.

```
(instance instance BinaryRelation)
```

In order to translate such axioms we currently split affected constants like `instance` into separate constants:

[4] TPTP syntax requires all constants in lower case, hence, the leading 'l' and 'n'. Moreover, we also encode the computed type information in the constant name; the reasons for this will become clear below.

[5] The 'I's encode bracketing information.

```
%%% The extracted Signature %%%
thf(lBinaryPredicate_THFTYPE_i,type,(
    lBinaryPredicate_THFTYPE_i: $i )).

thf(instance_THFTYPE_IIiioIioI,type,(
    instance_THFTYPE_IIiioIioI: ( $i > $i > $o ) > $i > $o )).

thf(instance_THFTYPE_IiioI,type,(
    instance_THFTYPE_IiioI: $i > $i > $o )).

%%% The translated axiom(s) %%%
thf(ax,axiom,
    ( instance_THFTYPE_IIiioIioI @ instance_THFTYPE_IiioI
                                 @ lBinaryPredicate_THFTYPE_i )).
```

Obviously, we thereby lose important information, for example, in our examples we now only know that `instance_THFTYPE_IiioI` denotes a binary relation. If we want this information restored for `instance_THFTYPE_IIiioIioI` we can generate a new constant `instance_THFTYPE_IIIiioIioIioI` and a new axiom

```
thf(ax,axiom,((instance_THFTYPE_IIIiioIioIioI @ instance_THFTYPE_IIiioIioI
                                              @ lBinaryPredicate_THFTYPE_i))).
```

Currently such a duplication of axioms is still disabled in our translation. Future work, however, will study the need for such duplications more closely.

Our first project goal has thus been achieved, namely to provide a translation of the entire SUMO into THF0 that can be parsed and type checked by all THF0 reasoners in the TPTP and that, in spite of its need for further improvement, can already serve as a starting point for examples as we have discussed.[6]

5 System Implementation, Integration, and Initial Experiments

The THF translation mechanism has been implemented as part of the Sigma environment. This enabled the reuse of already existing infrastructure, e.g. for manipulating formulas and knowledge bases, as well as the reuse of existing first-order logic TPTP tools in Sigma.

Additionally, an initial integration of the LEO-II system has been created with Sigma. There are three modes in which LEO-II can be applied to queries in this integration. The *local* mode only translates the user assertions and the query, the *global* mode translates the entire SUMO knowledge base and then adds the user assertions and the query, and the *SInE* mode employs Hoder's SInE system to extract a (hopefully) relevant subset of the axioms from the SUMO knowledge base.

We have conducted initial experiments with the LEO-II prover (version v1.1) integrated to Sigma: All examples in this paper can be effectively solved by LEO-II in local mode, except for Example 9: Ex.1 (0.19s), Ex.2 (0.19s), Ex.3 (0.13s), Ex.4 (0.16s), Ex.5 (0.08s), Ex.6 (0.34), Ex.7 (0.18s), Ex.8 (0.04s), Ex.9 (2642.55s) — the timings were obtained on a standard MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor and 2GB of memory. There is actually no general problem with Example 9, only LEO-II performs particularly poorly on it and the reasons for this should be investigated. Tests with other HO-ATPs via the SystemOnTPTP tool confirm that IsabelleP, for example, finds a proof in 10s.

[6] The THF0 translation of SUMO is available at: `http://www.ags.uni-sb.de/~chris/papers/SUMO.thf`.

We have submitted 28 related examples, each in two or three different versions, to the TPTP for inclusion. The different versions are corresponding to the three modes for calling LEO-II in Sigma as discussed before. Recent experiments of Geoff Sutcliffe with his TPTP infrastructure indicates that LEO-II is slightly ahead of the other provers for these example problems. The important news, however, is that the main hypotheses of our work has been confirmed: higher-order automated reasoners have the potential to advance the state-of-art in ontology reasoning and question answering. This has also been confirmed by the detection (and subsequent fixing) of some problematic axioms in SUMO in the course of our experiments. For example, in the following axiom for 'pretending' the last occurrence of `True` has been detected as semantically wrong and was subsequently replaced by `False` ('pretending' is is a social interaction where a cognitive agent or group of cognitive agents attempts to make another cognitive agent or group of cognitive agents believe something that is false):

```
(=> (instance ?PRETEND Pretending)
    (exists (?PERSON ?PROP) (and (hasPurpose ?PRETEND (believes ?PERSON ?PROP))
                                 (truth ?PROP True))))
```

Moreover, not only HO-ATP theorem provers are applicable to support ontology reasoning but also higher-order model finders. The IsabelleN model finder, for example, has revealed several typos in earlier versions of our example problems by constructing countermodels.

On the downside, however, our tests also show that much further work is needed for turning our proof of concept into a practically reliable and robust success story. For example, only very few of our examples can currently be solved in the SInE mode and even less can be solved in the global mode. Hence, the challenges involved in making inference efficient over large theories turns out even worse for the HO-ATPs than it already is for the FO-ATPs. This was to be expected though, in particular, since the theoretical and technical maturity of HO-ATPs is still many years, if not decades, behind those of FO-ATP systems.

6 Discussion

In this paper we have shown that HO-ATP is in principle capable of advancing the state-of-art in ontology reasoning and question answering in expressive frameworks such as SUMO. There are many open issues though that require much further thought and work. We briefly discuss a few.

The large theories challenge requires the development or adaptation of strong relevance filters such as SInE. We are still using an older version of SInE and we speculate that the latest version, in which the maximal number of selected axioms can be predetermined by a parameter, may already significantly improve the performance of the HO-ATPs in SInE mode.

There is also an important meta-reasoning task to be solved for Sigma. Currently, the selection of reasoners and further options is task of the user. In the future, however, we plan to automate this task. We envision distributed, possibly even cooperative, proof attempts by reasoners working for different translation targets like TPTP FOL and TPTP THF. Hence, the intended meta-reasoner needs to support various non-trivial tasks including: (i) selection of appropriate reasoners and translation targets, (ii) relevance filtering, (iii) control of (distributed) prover execution, (iv) extraction of answers from prover results, (v) result verification, (vi) preparation of answers and their presentation to the user.

For several of these tasks existing technology can possibly be adapted. Answer extraction, for example, is already supported in Sigma for all first-order provers which obey the standardized TPTP proof output format. And for supporting distributed or even cooperative reasoning with external systems in Sigma the agent-based OANTS architecture [12] can possibly be adapted.

One of the most interesting and relevant challenges, however, is the modalities challenge. As we have shown, Boolean extensionality and epistemic modalities such 'knows' or 'believes', for example, do not go well together. This observation is relevant beyond the borders of SUMO and it clearly also affects the current first-order translations: if they will eventually be extended so that they can successfully handle Examples 4 and 5, then they will also face the problem of Example 8.

Traditional (propositional) modal logics approaches and reasoners seem hardly applicable for the task since the modalities are usually employed in SUMO in combination with other first-order and higher-order constructs. One of our current research directions therefore aims at exploiting recent results that show how (multi-)modal logics can be elegantly encoded as simple fragments of higher-order logics [9, 8]. The idea is to consider modalities such as 'knows' or 'believes' as abbreviations for lambda-terms (as presented in [9]) denoting the appropriate modal operators. This solution explicitly supports the coexistence of different modalities in combination with other first-order and higher-order constructs. Related case studies on epistemic reasoning (for example, an elegant and efficient solution of the Wise Men Puzzle) with classical, extensional higher-order theorem provers can be found in [6].

We may also consider a modification of the theorem prover LEO-II and its underlying calculus. The idea would be to provide means for annotating function and predicate symbols regarding their pre-determined extensionality properties and to distinguish in the inference process according to these annotations. `holdsDuring`, for example, would be annotated as fully extensional, while `knows` and `believes` would not. Hence, the inference in Example 8 could be blocked in the prover while Example 5 would still go through. A respective research proposal in such a direction can already be found in [5] (which unfortunately was not funded at the time). Such a solution would allow us to make an informed and context-dependent choice regarding the extensionality principles for the semantics of SUO-KIF.

References

[1] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, second edition, 2002.

[2] Peter B. Andrews and Chad E. Brown. TPS: A hybrid automatic-interactive system for developing proofs. *Journal of Applied Logic*, 4(4):367–395, 2006.

[3] Julian Backes and Chad E. Brown. Analytic tableaux for higher-order logic with choice. In J. Giesl and R. Haehnle, editors, *IJCAR 2010 - 5th International Joint Conference on Automated Reasoning*, LNAI, Edinburgh, UK, July 2010. Springer. to appear.

[4] P. Baumgartner, A. Armando, and D. Gilles, editors. *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence. Springer, 2008.

[5] Christoph Benzmüller. ALONZO: Deduktionsagenten höherer Ordnung für Mathematische Assistenzsysteme. Research project proposal to the DFG Aktionsplan Informatik, available at `http://www.ags.uni-sb.de/~chris/papers/R23.pdf`, 2003.

[6] Christoph Benzmüller. *Automating Quantified Multimodal Logics in Simple Type Theory – A Case Study*. SEKI Working-Paper SWP-2009-02 (ISSN 1860-5931)). SEKI Publications, DFKI Bremen GmbH, Germany, 2009. `http://arxiv.org/abs/0905.4369`.

[7] Christoph Benzmüller, Chad Brown, and Michael Kohlhase. Higher-order semantics and extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[8] Christoph Benzmüller and Lawrence C. Paulson. *Quantified Multimodal Logics in Simple Type Theory*. SEKI Report SR–2009–02 (ISSN 1437-4447). SEKI Publications, DFKI Bremen GmbH, Germany, 2009. `http://arxiv.org/abs/0905.2435`.

[9] Christoph Benzmüller and Lawrence C. Paulson. Multimodal and intuitionistic logics in simple type theory. *The Logic Journal of the IGPL*, 2010. In print.

[10] Christoph Benzmüller, Lawrence C. Paulson, Frank Theiss, and Arnaud Fietzke. LEO-II — a cooperative automatic theorem prover for higher-order logic. In Baumgartner et al. [4], pages 162–170.

[11] Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. THF0 — the core TPTP language for classical higher-order logic. In Baumgartner et al. [4], pages 491–506.

[12] Christoph Benzmüller and Volker Sorge. OANTS – an open approach at combining interactive and automated theorem proving. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 81–97. A.K.Peters, 2000.

[13] Ulrich Furbach, Ingo Glöckner, and Björn Pelzer. An application of automated reasoning in natural language question answering. *AI Communications*, 23(2-3):241–265, 2010. PAAR Special Issue.

[14] Michael R. Genesereth. Knowledge interchange format. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning*, pages 238–249. Morgan Kaufmann, 1991.

[15] Patrick Hayes and Christopher Menzel. A semantics for the knowledge interchange format. In *IJCAI 2001 Workshop on the IEEE Standard Upper Ontology*, August 2001.

[16] Krystof Hoder. Automated reasoning in large knowledge bases. Master's thesis, Department of Theoretical Computer Science and Mathematical Logic, Charles University, Prague, 2008.

[17] Ian Niles and Adam Pease. Towards a standard upper ontology. In *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, pages 2–9, New York, NY, USA, 2001. ACM.

[18] Ian Niles and Adam Pease. Linking lexicons and ontologies: Mapping WordNet to the Suggested Upper Merged Ontology. In H. R. Arabnia, editor, *Proceedings of the International Conference on Information and Knowledge Engineering. IKE'03, June 23 - 26, 2003, Las Vegas, Nevada, USA, Volume 2*, pages 412–416. CSREA Press, 2003.

[19] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[20] Adam Pease. Standard Upper Ontology Knowledge Interchange Format. `http://sigmakee.cvs.sourceforge.net/*checkout*/sigmakee/sigma/suo-kif.pdf`.

[21] Adam Pease. The Sigma ontology development environment. In F. Giunchiglia, A. Gomez-Perez, A. Pease, H. Stuckenschmid, Y. Sure, and S. Willmott, editors, *Proceedings of the IJCAI-03 Workshop on Ontologies and Distributed Systems*, volume 71. CEUR Workshop Proceedings, 2003.

[22] Adam Pease and Geoff Sutcliffe. First order reasoning on a large ontology. In G. Sutcliffe, J. Urban, and S. Schulz, editors, *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories, Bremen, Germany, 17th July 2007*, volume 257 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[23] Adam Pease, Geoff Sutcliffe, Nick Siegel, and Steven Trac. Large theory reasoning with SUMO at CASC. *AI Communications*, 23(2-3):137–144, 2010.

[24] Deepak Ramachandran, Pace Reagan, and Keith Goolsbey. First-orderized ResearchCyc: Expressivity and efficiency in a common-sense ontology. In Shvaiko P., editor, *Papers from the AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications, Pittsburgh, Pennsylvania, USA, 2005*. Technical Report WS-05-01 published by The AAAI Press, Menlo Park, California, July 2005.

[25] Geoff Sutcliffe. TPTP, TSTP, CASC, etc. In V. Diekert, M. Volkov, and A. Voronkov, editors, *Proceedings of the 2nd International Computer Science Symposium in Russia*, number 4649 in Lecture Notes in Computer Science, pages 7–23. Springer, 2007.

[26] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[27] Geoff Sutcliffe and Christoph Benzmüller. Automated reasoning in higher-order logic using the TPTP THF infrastructure. *Journal of Formalized Reasoning*, 3(1):1–27, 2010.

[28] Geoff Sutcliffe, Christoph Benzmüller, Chad Brown, and Frank Theiss. Progress in the development of automated theorem proving for higher-order logic. In R. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2009.

[29] Steven Trac, Geoff Sutcliffe, and Adam Pease. Integration of the TPTPWorld into SigmaKEE. In B. Konev, R. Schmidt, and S. Schulz, editors, *PAAR/ESHOL*, volume 373 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

GridTPT: a distributed platform for Theorem Prover Testing*

Thomas Bouton Diego Caminha B. de Oliveira David Déharbe
Pascal Fontaine
david@dimap.ufrn.br
[Thomas.Bouton,Diego.Caminha,Pascal.Fontaine]@loria.fr
Universidade Federal do Rio Grande do Norte, Brazil
LORIA, INRIA & Nancy University, Nancy, France

Abstract

Programming provers is a complex task; completeness or even soundness may often be broken by apparently harmless bugs. A good testing platform can contribute in detecting problems early and helping development. This paper presents GridTPT, the distributed platform for testing the veriT SMT solver. Its features are fairly standard, but it allows to easily distribute the task in a cluster.

We plan to make this platform available as an open source tool for the community of developers of automated theorem provers. This presentation to PAAR'2010 will provide the opportunity to discuss the need for such a tool and the necessary features in a broader context. We would like to extract a requirement specification from this discussion, that would be useful to get dedicated implementation resources for distribution, maintenance and future development of GridTPT.

1 Introduction

The implementation of efficient automated theorem provers requires intricate data structures and algorithms and is therefore error-prone. As a consequence, establishing the functional correctness of those tools includes applying large test suites, in addition to other measures such as third-party certification of intermediate and final results through e.g. proof generation and proof checking. The faster those verification results are available, the sooner mistakes are discovered and can be corrected by the developers. Also, automated theorem proving is intrinsically of a heuristic nature and requires experimenting with many different combinations of parameters. Again, this experimental study needs frequently applying large test suites.

Testing over a large number of benchmarks can easily be done in parallel (at least from a theoretical viewpoint). However, owning and maintaining a large cluster of machines is both time-consuming and financially expensive, and most prover developers do not have the resources to do this work in addition to research and implementation of the prover. Nevertheless, many research and university environments do have large clusters that are not fully used. More and more often these computing facilities are again clustered via a grid infrastructure that provides access to hundreds and even thousands of cores. It is often possible to obtain a low priority (i.e. when not in use for the financing projects) access to those clusters, and this low priority access will most of the time be suitable for the use of prover testing. Once the cluster is found, one needs to develop the software infrastructure for running the tests. Although a set of *ad hoc* scripts would do the basic job, a dedicated platform developed over a long period could provide many useful services.

Our goal is to share with the theorem proving community the software for a distributed testing platform for automated provers that we have built incrementally to support the development of the SMT solver veriT[1]. This software reduced the testing time from a week-end to a few minutes. For instance,

*This work is partly supported by the ANR project DECERT, and the INRIA-CNPq project SMT-SAVeS.

[1]For the development of veriT, we have been kindly granted access to a large grid infrastructure of INRIA known as Grid5000 [6].

the approximately nine thousand formulas in the categories for which veriT is complete are checked in 12 minutes with 80 cores and a 30 seconds time-out. As another example, the whole TPTP (around 14000 files) is run on E [12] in 20 minutes using 160 cores and a 30 seconds time-out. We plan to release this powerful and customizable platform under the open-source BSD license as well as offering maintenance to meet new requirements of external users. For theorem prover developers this would reduce the problem of having a good testing infrastructure to finding the cluster to run our software on.

There already exists platforms for evaluating solvers, and in particular, the platforms for the various annual competitions, for instance CASC [10, 13] and SMT-COMP [1, 2].[2] The main focus for those frameworks is to precisely and fairly measure the running time for the various solvers on the instances chosen for the competition. The purpose of GridTPT is different: it includes comparing versions of the same solver/prover but being precise in measuring running time is not the main objective. Much more importantly, the tool gathers statistics, and provides to the user ways to understand the tendencies and the relations between various quantities.

The platform is now stable and has reached a point where its use in a larger context, for slightly different goals, and in various environments, requires the feedback of the community, which we would like to get at PAAR. Following the presentation of GridTPT, we expect to get, from potential users, additional requirements to enhance and make the platform more attractive. Additionally, after we show the benefit of using the platform, we expect some of the participants will be interested in being users.

2 State of the platform

The testing platform has been used and improved to support the development of the SMT solver veriT [7] for more than a year. The test data used by the platform are the different categories of SMT-LIB [11, 3] benchmarks that are supported by the solver. The best way to run the tests and to access data is through the web interface, but the reports are in plain text, and all the scripts may be run from the command line.

Three types of tests can be performed over a selected set of benchmarks:

- functional test: the satisfiability status (satisfiable, unsatisfiable, or unknown), execution time (or failure, or time out) and other (user defined) statistics are gathered.
- consistency test: for each benchmark, the solver generates verification conditions corresponding to intermediate results. External solvers[3] are applied to recheck these conditions to ensure that not only the final result, but also the reasoning leading to this result is correct.
- memory test: memory leaks are detected using Valgrind (see `http://valgrind.org/`).

The latter two tests only generate a brief report to notify the developers if further debugging is required on particular benchmarks.

A prototype version of the script was sequential. An extensive test over the SMT-lib used to take several days to complete. The present version distributes the work over several multi-core computers, drastically reducing the total execution time to a few minutes. It uses a master/slave architecture, where one node assumes the role of the master, distributes the benchmarks and gathers the results, while the other nodes are slaves and execute the solver. It is fault tolerant: in case the connection to a slave is lost (due to a network failure, node hanging, ...), the full test is not affected. A test can be suspended at any time, and resumed later without significant duplicated work. Finally and most importantly, the

[2] Some organizations even give access to the clusters outside the competitions.

[3] In the case of veriT, we use CVC3 [4] to check intermediate conflict clauses and PicoSAT [5] to check the overall Boolean abstraction (MiniSAT [9] is used internally).

framework has been written to be easily portable: its implementation language is Python with a few OS-specific scripts written in bash.

Competitions usually distribute one process per computer, to eliminate interferences between processes. This is required in order to accurately and fairly measure the running time of the various solvers on the competition benchmarks. Since we prefer to get short testing time, we allow to send one job for every core available on the cluster, even if these are on the same processor. This introduces some slight variations in running times, but this is an acceptable price to pay to divide the overall testing time by a factor of 8 (in our case). Our experiments show that, even so, times are measured quite accurately.

The statistics collected by the tool and their value are not hard coded, but rather gathered from the output of the prover. They should be prefixed with a configurable character string – so that these statistics can be recognized from irrelevant information, such as an execution trace – followed by the name of the statistic and its value. Similarly, error messages need to be prefixed by a definable string. Notice that the statistics should at least provide the result of the prover on the formula. The execution time can be computed by the command *time* (on *nix systems). Figure 1 presents a typical output from veriT. Obviously, it is easy to put the information in the required form without modifying the internals of the prover by simply using a shell script wrapper.

```
verit 200907 - the veriT solver (UFRN/LORIA).
[...]
STAT_DESC: clauses: Number of clauses generated
STAT_DESC: res: 0 (UNSAT), 1 (SAT), -1 (UNKNOWN)
STAT_DESC: nodes: Number of nodes in the input formula as a DAG representation
STAT_DESC: nodes_tree: Number of nodes in the input formula as a tree representation
STAT_DESC: atoms: Number of atoms in the input formula as a tree representation
STAT_DESC: total_time: Total time
STAT: clauses=1486
STAT: res=0
STAT: nodes=799
STAT: nodes_tree=4114
STAT: atoms=1825
STAT: total_time=1.204
[...]
```

Figure 1: A typical output from veriT.

New tests are triggered automatically by cron jobs (if no test exists for the current version in the subversion repository), or manually through the developer-only section on the website of the solver. In the latter case, the tester has the opportunity to choose the solver revision, the solver options, the list of benchmarks on which the solver is to be run, and an optional comment. Reports are automatically generated and can be consulted on line, via the project website. The access to the reports is restricted to developers only. Other available features include the capacity to compare either graphically or textually two functional reports and to extract CSV (comma-separated values format) files for reports or for comparison of reports in order to do more sophisticated treatments using other tools (such as spreadsheets). Some of these features are demonstrated in the next section.

3 Illustration

This section contains illustrative information on the following capacities of the platform: functional report, a textual comparison report, and a graphical comparison report.

3.1 Report example

An extract of a sample test report is given below:

```
-veriT report----------------
Date            : 20090904133605
-informations----------------
Host name       : Grid5000
Number of cores : 80
CPU type        : xeon-harpertown at 2.5GHz
Executable      : ./verit
Build time      : 20090903181241
Options         : --enable-simp --enable-unit-simp --cnf-p-definitional -v
Number of files : 8965
CPU limit       : 30s
-grid statistics-------------
Cumulative time  : 876m (14h)
Total time       : 12m
Theoretical time : 11m
-legend----------------------
total_time: Total time
res: 0 (UNSAT), 1 (SAT), -1 (UNKNOWN)
atoms: Number of atoms in the input formula as a tree representation
nodes_tree: Number of nodes in the input formula as a tree representation
clauses: Number of clauses generated
nodes: Number of nodes in the input formula as a DAG representation
-summary---------------------
Total number of benchmarks : 8965
Number of success : 7638
     between 0 and 5 sec : 6835
     between 5 and 10 sec : 465
     between 10 and 15 sec : 199
     between 15 and 20 sec : 89
     between 20 and 25 sec : 36
     between 25 and 30 sec : 14
Number of "CPU time limit exceeded" : 1327
-data------------------------
Name                                                  total_time res       atoms  nodes_tree clauses  nodes
QF_IDL/Averest/binary_search/BinarySearch_live_bgmc000.smt        0.000   1      207793      208901       0    339
QF_IDL/Averest/binary_search/BinarySearch_live_bgmc002.smt        0.000   1      415523      417695       0    607
QF_IDL/Averest/binary_search/BinarySearch_live_bgmc003.smt        0.000   0      623253      626489       0    875
QF_IDL/Averest/binary_search/BinarySearch_live_blmc000.smt        0.000   1      623314      626594       1    942
QF_IDL/Averest/binary_search/BinarySearch_live_blmc002.smt        0.004   0     1246566     1253082       0   1814
QF_IDL/Averest/binary_search/BinarySearch_safe_bgmc000.smt        0.000   0         406         546       0    203
QF_IDL/Averest/binary_search/BinarySearch_safe_bgmc001.smt        0.000   1          99         175       0    109
QF_IDL/Averest/binary_search/BinarySearch_safe_bgmc002.smt        0.000   0      208393      209661       0    550
QF_IDL/Averest/binary_search/BinarySearch_safe_bgmc003.smt        0.000   1      416380      418776       2    897
QF_IDL/Averest/binary_search/BinarySearch_safe_blmc000.smt        0.000   0      416266      418750       0   1008
QF_IDL/Averest/binary_search/BinarySearch_safe_blmc001.smt        0.004   1      831718      836406       8   1329
QF_IDL/Averest/binary_search/BinarySearch_safe_blmc002.smt        0.004   1     1247479     1254435       3   1950
[...]
```

The presentation of the report may need to be adapted for other solvers. However, as mentioned above, the list of statistics is not hardcoded, and is built during the parsing of the output of the solver, assuming that it follows some formatting instructions.

3.2 Textual comparison

The comparison tool has the following parameters:

- the two functional reports to be compared;
- the categories of benchmarks to compare the reports on;
- the minimum spread, in percent of execution time, for the benchmark to be shown;
- the minimum spread, in absolute execution time, for the benchmark to be shown.

If the two reports are on different sets of benchmarks, only the common subset is shown. The statistics from both reports are shown. Optionally, the comparison tool may hide benchmarks for which running time are not sufficiently different. The spread between the execution times must then be higher than a specified percentage and a minimum value. Indeed, for benchmarks solved very quickly, 0.01 second is twice as fast as 0.02, but the running time difference may still be considered negligible.

Here is a small excerpt of a comparison report:

Name	total time		result	
	20090729191611	20080729142114	20090729191611	20080729142114
QF_UFIDL/pete3/bug_file3.smt	0.800	Failed	1	Failed
QF_UFIDL/pete3/bug_file4.smt	198.404	Failed	1	Failed
QF_UFIDL/pete3/bug_file5.smt	25.286	4.75	1	1
QF_UFIDL/uclid/22s.smt	0.332	0.58	0	0
QF_UFIDL/uclid/43s.smt	6.604	2.28	0	0
QF_UFIDL/uclid/cache.inv10.smt	3.536	5.46	0	0
QF_UFIDL/uclid/cache.inv14.smt	105.999	Failed	0	Failed
QF_UFIDL/uclid/cache.inv8.smt	0.652	1.08	0	0
QF_UFIDL/uclid/elf.rf10.smt	18.149	25.23	0	0
QF_UFIDL/uclid/elf.rf8.smt	0.320	0.62	0	0
QF_UFIDL/uclid/elf.rf9.smt	2.536	3.38	0	0
QF_UFIDL/uclid/ooo.rf10.smt	25.942	Failed	0	Failed
QF_UFIDL/uclid/ooo.rf8.smt	1.208	1.56	0	0
QF_UFIDL/uclid/ooo.tag10.smt	3.736	5.02	0	0
QF_UFIDL/uclid/ooo.tag12.smt	39.114	Failed	0	Failed
QF_UFIDL/uclid/q2.12.smt	15.201	19.58	0	0
QF_UFIDL/uclid/q2.14.smt	77.169	Failed	0	Failed
QF_UFIDL/uclid2/bug1.smt	9.721	13.57	1	1
QF_UFIDL/uclid2/bug2.smt	0.780	1.45	1	1
QF_UFIDL/uclid2/ooo.rf11.smt	158.894	Failed	0	Failed

On the web page, for each benchmark, the color code explicitly highlights improvement or regression.

3.3 Graphical comparison

Usually, on large sets of benchmarks, a graphical comparison helps to highlight the difference in execution time between two revisions of the solver. The web interface also displays an XY logarithmic graph (see figure 2). Again, more in-depth analysis may be done very quickly using the CSV data extraction facility and using a spreadsheet.

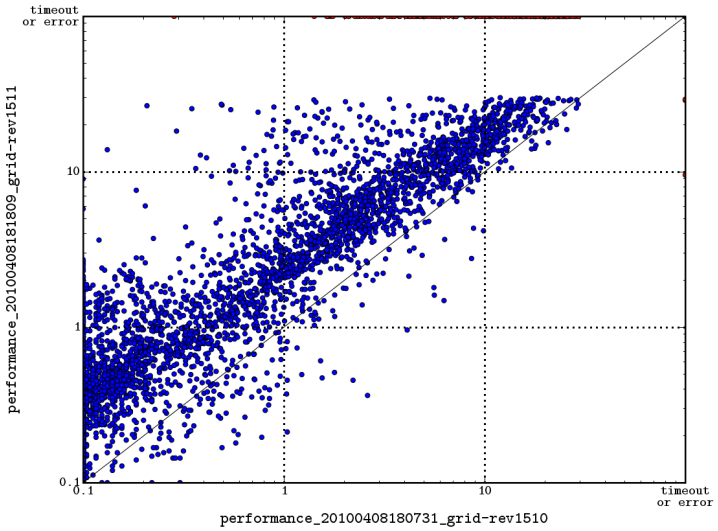


Figure 2: Example of a graphical comparison. A blue dot corresponds to one benchmark.

4 Conclusion and future work

We presented GridTPT, the testing platform used daily in the development of the veriT SMT solver. Since most prover developers have the same kind of needs for such a platform, we feel that this work may benefit other groups in the ATP community. The platform was used internally on several third-party SMT solvers. First positive experiments were also carried on with a first-order theorem prover (namely, the E prover [12]).

The most important difficulty we encountered in integrating the platform into our web server (for easy access by developers outside our institution) is the access policy to the cluster and the electronic security policy of our institution. The cluster is strongly firewalled, whereas the web server is outside the protected area; sending jobs and getting back information from the cluster to the web server requires hacks and ssh bounces. We believe that users elsewhere may encounter similar problems. Unfortunately, this prevents to have a clean package and an easy installation procedure that would work out-of-the-box for all cases. It will be necessary to collect and provide off-the-shelf solutions that will allow to circumvent those problems semi-automatically, when an automatic installation is not suitable. Another issue that we will certainly have to face is the variety of tools that clusters use for reserving resources.

The variation in running times with respect to the computer architecture is not linear, since it depends on the instruction set of the processors, the frequencies, the cache sizes and management policy,... that affect differently the running time depending on the program and even on the input data. Nevertheless, we think that it may be useful to investigate some kind of architecture calibration, i.e. recompute an approximation of the running time on a reference architecture. The motivations for such a calibration are twofold. First it would then be possible to use heterogeneous clusters if precise time measurement is not required. Second, it would also allow developers to compare old results (on out-of-use architectures) with newer ones.

Among the ongoing works, we are currently integrating the fuzzing tools for SMT-lib [8] on the distributed architecture. We also have a prototype of an interface to better visualize the differences in running time, with respect to the kind of benchmarks (categorized by their directory, subdirectory, and name prefix). The new version of the SMT-LIB [3] brings a novelty in allowing scripts in the prover input language; finding the right way to nicely integrate testing for scripts will also be necessary.

Acknowledgments: We would like to thank Stephan Merz for his guidance. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`). We also thank the anonymous reviewers for their comments.

References

[1] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 20–23. Springer, 2005.

[2] C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and results of the 3rd annual satisfiability modulo theories competition (SMT-COMP 2007). *International Journal on Artificial Intelligence Tools*, 17(4):569–606, 2008.

[3] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard : Version 2.0. First official release of Version 2.0 of the SMT-LIB standard., 2010. See also `http://www.smtlib.org/`.

[4] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.

[5] A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[6] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, Nov. 2006. See also `https://www.grid5000.fr`.

[7] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In R. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156, Montreal, Canada, 2009. Springer.

[8] R. Brummayer and A. Biere. Fuzzing and delta-debugging SMT solvers. In *SMT '09: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5, New York, NY, USA, 2009. ACM.

[9] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[10] F. J. Pelletier, G. Sutcliffe, and C. B. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[11] S. Ranise and C. Tinelli. The SMT-LIB standard : Version 1.2, Aug. 2006. See also `http://www.smtlib.org/`.

[12] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[13] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

Automated Higher-order Reasoning about Quantales

Han-Hing Dang, Peter Höfner
Institut für Informatik, Universität Augsburg
D-86135 Augsburg, Germany
{dang,hoefner}@informatik.uni-augsburg.de

Abstract

Originally developed as an algebraic characterisation for quantum mechanics, the algebraic structure of quantales nowadays finds widespread applications ranging from (non-commutative) logics to hybrid systems. We present an approach to bring reasoning about quantales into the realm of (fully) automated theorem proving. This will yield automation in various (new) fields of applications in the future. To achieve this goal and to receive a general approach (independent of any particular theorem prover), we use the TPTP Problem Library for higher-order logic. In particular, we give an encoding of quantales in the typed higher-order form (THF) and present some theorems about quantales which can be proved fully automatically. We further present prospective applications for our approach and discuss practical experiences using THF.

1 Introduction

Automated theorem proving (ATP) has brought automated reasoning into a wide variety of domains. Examples where significant and important success systems has been achieved with ATP, are software verification and mathematics. Full automation (without user interaction) is often only possible by first-order logic. For these tasks ATP systems like Vampire [26] and Prover9 [21] exist and SystemOnTPTP [28] provides a common language and a common interface. First-order ATP systems and the TPTP library have extensively been used by different users in various case studies covering various areas of sciences (e.g., [9, 15, 17, 18, 31]).

Recently, TPTP and SystemOnTPTP were extended to cover not only first-order reasoning but also reasoning within higher-order logic [6, 29]. A general aim is again full automation and no user interaction. As a part of TPTP, higher-order logic can now be expressed by the typed higher-order form (THF) which implements Church's simple theory of types [10]. This theory is based on the simply typed λ-calculus in which functional types are formed from basic types. The decision for simple type theory was made since this theory is already used as a common basis for a lot of higher-order ATP systems [6].

Up to now only a few case studies using THF exist. In this paper we provide a case study and bring reasoning about the algebraic structure of quantales into the realm of fully automated theorem proving. In particular, we encode quantales into the typed higher-order form of the TPTP library and perform a proof experiment using that approach with about 50 theorems. In doing so, we also evaluate all the ATP systems for higher-order reasoning that are integrated in SystemOnTPTP w.r.t. their capabilities for automated reasoning about quantales. In detail, these systems are IsabelleP [24], LEO-II [5], Satallax [2] and TPS [1]. We have chosen the structure of quantales due to several reasons:

- From a mathematical point of view quantales are special cases of the first-order structures of semirings and Kleene algebras. Since the latter structures are particular suitable for automated reasoning [15, 16, 17], the hope is that quantales are also suitable.

- Following [6], THF is particular suitable for set-based encodings. In this paper we will derive an encoding of quantales based on sets. Hence the hope is again that quantales yield good automation results.

- As a third reason why quantales are chosen, we mention the prospective applications. Originally, quantales were introduced to formalise phenomena of quantum mechanics. Later, this algebraic structure found various fields of application. Examples are classical logic like CTL and CTL* [22], non-classical logic like separation logic or non-commutative logic [12, 25, 32] as well as reasoning within hybrid systems [14]. Further examples can be found in [27].

There are two main contributions of the paper: First we develop the above-mentioned case study. As far as we know it is one of the largest proof experiments of THF. As a consequence, we show that fully automated reasoning within higher-order logic is feasible in principle. Unfortunately, none of the ATP systems used can even prove half of the given theorems. At the moment more complex properties cannot be verified with state-of-the-art ATP systems from the axioms. To achieve full automation in various new fields of applications, further development of fully automated ATP systems is needed. However, there was the same situation some years ago when TPTP offered the first problem suites for first-order logic. After TPTP was launched, the evolution of ATP systems were quite impressive, especially for the problems listed in the TPTP library. Since quantales will be part of TPTP v4.1.0 we hope for the same effect and believe that fully automated reasoning within quantales will be much more feasible in a couple of years. Due to this we will present, as the second contribution, a number of possible applications where automation in quantales can be applied if ATP systems perform better.

The paper is organised as follows: In Section 2 we define the algebraic structure of quantales and give all necessary mathematical background. In the following section we sketch the typed higher-order form of TPTP. We begin our case study by encoding quantales within THF in Section 4. After that we try to verify basic properties of quantales. In particular, we present and discuss the results of our proof experiment in Section 5. From this basic toolkit we then give possible fields of applications in Section 6. We conclude the paper by discussing some on-going and future work.

2 Quantales

Quantales, the algebraic basis for our case study, are used in a wide range of sciences like computer science, mathematics, physics or philosophy. Therefore they unify a wide range of applications from a mathematical point of view. More precisely, quantales are partially ordered sets that generalise various lattices of multiplicative ideals from ring theory as well as point free topologies and functional analysis. Later, in Section 6, we will discuss fields of application in much more detail.

Before defining quantales, we recapitulate some lattice theory.

A *complete lattice* $(S,\leq)$ is a partially ordered set in which all subsets have a supremum. The definition implies that all subsets have also an infimum and that there is a least element $0 =_{df} \bigsqcup\emptyset$ and a greatest element $\top =_{df} \bigsqcup S$. The infimum of an arbitrary set $X \subseteq S$ is denoted by $\bigsqcap X$ while the supremum is denoted by $\bigsqcup X$. The binary variants for two elements $x,y \in S$ are written as $x \sqcap y$ and $x \sqcup y$, resp.

A *quantale* (e.g. [27]) is a structure $(S,\leq,\cdot,1)$ where $(S,\leq)$ is a complete lattice and $\cdot$ is an associative, completely disjunctive inner operation on S, i.e., $\cdot$ distributes over arbitrary suprema: For an index set I and arbitrary elements $x,y_i \in S$ we have

$$x \cdot \left(\bigsqcup_{i\in I} y_i\right) = \bigsqcup_{i\in I} (x \cdot y_i) \quad \text{and} \quad \left(\bigsqcup_{i\in I} y_i\right) \cdot x = \bigsqcup_{i\in I} (y_i \cdot x)\ . \tag{$*$}$$

Moreover 1 is required to be the identity of multiplication, i.e., $x \cdot 1 = 1 \cdot x = x$. The notion of a quantale is equivalent to Conway's notion of a *standard Kleene algebra* [11] and forms a special idempotent semiring. As an immediate consequence of the definition, multiplication is strict, i.e., $x \cdot 0 = 0 \cdot x = 0$ for all $x \in S$.

We now have a closer look at the axiomatisation of quantales. It is easy to see that the infinite distributivity laws ($*$) make it nearly impossible to encode quantales within first-order logic. We are only aware of one possibility. However this would require many predicates and types and does not yield good results for automation. We follow the lines of Conway's book [11] and axiomatise quantales by the following set-based formulas.

$$\bigsqcup\emptyset = 0\,, \quad (1) \qquad x\cdot 1 = 1\cdot x = x\,, \quad (4)$$

$$\bigsqcup\{x\} = x\,, \quad (2) \qquad (x\cdot y)\cdot z = x\cdot(y\cdot z)\,, \quad (5)$$

$$\bigsqcup\{\bigsqcup X_i : i\in I\} = \bigsqcup\bigcup_{i\in I} X_i\,, \quad (3) \qquad \bigsqcup X_1\cdot\bigsqcup X_2 = \bigsqcup\{x_1\cdot x_2 : x_1\in X_1, x_2\in X_2\}\,, \quad (6)$$

where X_1, X_2, $X_i \subseteq S$ for all $i \in I \subseteq \mathbf{N}$, x, y, $z \in S$ and $\bigcup_{i\in I}$ denotes set union over an index set I. In this axiomatisation Axiom (1) characterises the special element 0. The Laws (2) and (3) "inductively" define the supremum; Equations (4) and (5) make the lattice to a multiplicative monoid. The last axiom is the counterpart for the infinite distributivity laws ($*$). From these axioms one can easily define the order relation $\leq$ by $x \leq y \Leftrightarrow_{df} x \sqcup y = y$; the infimum operator can be characterised, as usual, by

$$\bigsqcap X = \bigsqcup\{y : \forall x\in X : y\leq x\}\,. \quad (7)$$

We will use the definitions of $\bigsqcup$, $\bigsqcap$ and $\cdot$ to encode quantales in the typed higher-order TPTP library in Section 4. Next to these operators Conway defines an operator * for finite iteration by $X^* = \{X^i : i \in \mathbf{N}\}$, where $X^0 = 1$ and $X^{n+1} = X^n \cdot X$. Since our approach follows Conway's axiomatisation, we would need arithmetic to encode this axiom. However THF does not allow arithmetic at the moment. Hence we do not discuss the star operator in this paper though we are aware of equivalent axioms that only need first-order logics (see [20]).

3 THF and Church's Simple Type Theory

In this section we sketch the higher-order approach in the TPTP problem library. We will only mention those points that are necessary for the encoding of quantales later on. In particular, we explain the meaning of the symbols that may occur in formulas. A detailed description of the higher-order approach can be found in [29].

The typed higher-order form (THF) of the TPTP problem library implements higher-order logic by Church's simple theory of types [10] since this theory is already used as a common basis for a lot of higher-order ATP systems [6]. That theory is based on the simply typed λ-calculus in which functional types are formed from basic types. These consist of types of *individuals* `$i`, of *Boolean values* `$o`; further ones can be built using the *function type constructor* `>`.

In THF all formulas are annotated and have the following form:

```
thf( <formula_name>, <role>, ( <formula> )).
```

`<formula_name>` identifies the formula by a unique name, the attribute `<role>` specifies the role of the formula like axiom, (type) definition, conjecture or theorem. The actual formula is given in the last part of the THF-structure. Detailed examples will be shown in the next section. Symbols that are allowed to occur in the formulas are `!`, `?` and `^`. They stand for $\forall$, $\exists$ and λ, resp. The binary operator `@` denotes function application.

4 Encoding in Higher-order TPTP-Syntax

As it can be seen from Section 2 in Conway's axiomatisation of quantales, a suitable encoding of set theory in higher-order logic is required. We have chosen an encoding that was already proposed by Benzmüller, Rabe and Sutcliffe [6]. This encoding has already been successfully implemented and applied. Sets are being represented by their characteristic functions. In particular, we use for an element x and a set X the following equivalence

$$x \in X \Leftrightarrow \mathsf{X}(x).$$

On the right-hand side X denotes a predicate. By this, set operations such as intersection or union can easily be expressed using the typed λ-calculus. For example, binary union is defined by

$$\lambda\,\mathsf{X},\mathsf{Y},x.\ ((\mathsf{X}x) \vee (\mathsf{Y}x))$$

assuming that the predicates X and Y have type $\alpha \to \$o$ and x has type α.

In the remainder we give an extract of the complete input file to demonstrate the encoding. A full encoding of the Axioms (1)–(6) is given in the Appendix[1] and at a web site [13]. We only consider the supremum definition here since it forms the most interesting part of the encoding.

```
14 thf(sup,type,(
15     sup: ( ( $i > $o ) > $i ) )).
```

The formula defines the type of the supremum operation. It takes a characteristic function of an arbitrary set as an argument (which has the type `( $i > $o )`) and returns its supremum of type `$i`. With this, the encoding of Axioms (1) and (2) is straightforward.

```
16 thf(sup_es,axiom,( (sup @ emptyset) = zero )).
17 thf(sup_singleset,axiom,(
18     ! [X: $i] : ( ( sup @ ( singleton @ X ) ) = X ) )).
```

Clearly, the function `emptyset` maps every element of type `$i` into false. Furthermore in the second formula `! [X: $i]` denotes a $\forall$-quantification over all elements `X` and `( singleton @ X )` a set containing only a single element `X`.

For the axiom $\bigsqcup\{\bigsqcup X_i : i \in I\} = \bigsqcup\bigcup_{i\in I} X_i$, we have to model characteristic functions representing the sets $\{\bigsqcup X_i : i \in I\}$ and $\bigcup_{i\in I} X_i$. This is done by defining functions that take a set of sets as an argument. Using the λ-calculus, the function for $\{\bigsqcup X_i : i \in I\}$ can be rephrased into

$$\lambda\mathsf{F},x.\ \exists\mathsf{Y}.\ (\mathsf{F}\mathsf{Y}) \wedge \left(\left(\bigsqcup\mathsf{Y}\right) = x\right)$$

where $\mathsf{F} = \{X_i : i \in I\}$ and Y denotes a set X_j that contains x. This is directly encoded into THF.

```
20 thf(supset,type,(
21     supset: ( ( ( $i > $o ) > $o ) > $i > $o ) )).
22 thf(supset,definition,
23     ( supset = ( ^ [F: ( $i > $o ) > $o, X: $i ] :
24       ? [Y: $i > $o] : ( ( F @ Y ) & ( ( sup @ Y ) = X ) ) ) )).
```

In a similar way, a function `unionset` for $\bigcup_{i\in I} X_i$ can be given (cf. the Appendix). Using these functions, Axiom (3) can now encoded by

[1] The line numbers we give in this section correspond to the one of the Appendix.

```
thf(sup_set,axiom,(
    ! [X: ( $i > $o ) > $o] : ( ( sup @ ( supset @ X ) ) =
                               ( sup @ ( unionset @ X ) ) ) )).
```

Arbitrary index sets I can now be handled by quantification over sets of sets. This is a big advantage of higher-order encodings. As mentioned before, there are attempts to encode quantales within first-order logic. However all known characterisations are very complex and very difficult to read. Set theory is encoded much more naturally in higher-order logic and therefore the axiomatisation for quantales we have given is quite natural. Moreover, it has been stated that set-theoretic theorems are solved more efficiently in higher-order logic than using first-order encodings [7].

5 Automating Basic Properties

In this section we use the encoding of Section 4 to automatically verify basic properties of quantales. We proved around 50 theorems, in particular theorems that involve infima and suprema over infinite sets. So far we have only proved a basic subset of the theorems we are interested in, since at the moment none of the ATP systems can even prove half of the them. Hence automating proofs of more complex properties in advanced fields of applications (cf. Section 6) are currently not possible. If not only the axioms but some further properties are provided, all listed theorems can be proved automatically.

For our experiment we used Sutcliffe's SystemOnTPTP Tool [28]. In particular we evaluated four higher-order logic ATP systems for finding proofs of theorems in quantales: IsabelleP 2009-1 [24], LEO-II 1.1 [5], Satallax 1.2 [2] and TPS 3.080227G1d [1]. The computers for the evaluation used a 2.8 GHz Intel Pentium 4 CPU, 1 GB of memory, running on a Linux 2.6 operating system. We set a CPU time limit of 300 s, which is known to be sufficient for the ATP systems to prove almost all the theorems they would be able to prove even with a significantly higher limit [30]. The results of this testing are shown in Table 1; a number indicates that a proof is found in that time, and a "–" indicates that the system reaches the time limit or gives up before reaching this limit.

In the remainder of the section we will discuss some of the results in more detail. In particular we report on some of the difficulties and practical aspects we faced when performing the proof experiment of Table 1 fully automatically.

Table 1 includes properties of $\bigsqcup$ and $\bigsqcap$ that denote the role of 0 being the least element w.r.t the natural order $\leq$ of the lattice structure. Moreover we encoded simple isotony properties ((E20)–(E25)), associativity (E18) and distributivity laws ((E35), (E37)–(E39)). At first we fed the ATP systems with simple theorems using the $\bigsqcup$ operation. We realised that in contrast to other ATP systems LEO-II timed out already when showing (E1) which could be immediately inferred by instantiating Axiom (2) setting $x = 0$. However since LEO-II is able to show Property (E6) that can also be used together with (E5) to infer (E1) we think that either the given encoding is not appropriate for LEO-II or its search strategies are not effective enough for our tasks.

Looking at (E14) and (E15) that denote commutativity laws for $\bigsqcup$, Isabelle and TPS seem to have problems. The properties would be derivable from commutativity of set union which both systems could prove immediately.

Another difficulty arose when proving Theorem (E9) which simply denotes a binary variant of Axiom 3 ($\bigsqcup\{\bigsqcup X_i : i \in I\} = \bigsqcup \bigcup_{i \in I} X_i$). The axiom is quantified over sets of sets. None of the ATP systems was able to instantiate the axiom appropriately. To overcome this difficulty, an auxiliary function has been defined for building sets consisting of sets. By two additional assumptions, this function is related to ordinary set union (`unionset`).

	System	Isabelle	LEO-II	Satallax	TPS
(E1)	$\bigsqcup\{0\} = 0$	3.2	–	0.2	10.7
(E2)	$\bigsqcup(\{0\} \cup \{0\}) = 0$	3.1	–	92.1	–
(E3)	$\bigsqcup\{\bigsqcup\{x\}\} = x$	3.1	–	0.2	–
(E4)	$\bigsqcup(\{x\} \cup \emptyset) = x$	3.2	–	–	–
(E5)	$\bigsqcup\emptyset = \bigsqcup\{0\}$	3.1	–	0.2	18.3
(E6)	$\bigsqcup\emptyset = 0$	3.1	0.1	0.2	10.8
(E7)	$\bigsqcup(\{x\} \cup \{0\}) = \bigsqcup(\{\bigsqcup\{x\}\} \cup \{\bigsqcup\emptyset\})$	3.1	–	64.0	–
(E8)	$\bigsqcup(\{\bigsqcup\{x\}\} \cup \{\bigsqcup\{y\}\}) = \bigsqcup(\{x\} \cup \{y\})$	3.2	–	0.1	–
(E9)	$\bigsqcup(\{\bigsqcup X\} \cup \{\bigsqcup Y\}) = \bigsqcup(X \cup Y)$	–	–	–	–
(E10)	$\bigsqcup(\{\bigsqcup\{x\}\} \cup \{\bigsqcup\emptyset\}) = \bigsqcup(\{x\} \cup \emptyset)$	–	–	–	–
(E11)	$\bigsqcup(\{\bigsqcup\{x\}\} \cup \{\bigsqcup\emptyset\}) = x$	–	–	–	–
(E12)	$\bigsqcup(\{x\} \cup \{0\}) = x$	–	–	–	–
(E13)	$0 \leq x$	–	–	–	–
(E14)	$\bigsqcup(\{x\} \cup \{y\}) = \bigsqcup(\{y\} \cup \{x\})$	–	0.1	0.8	–
(E15)	$x \sqcup y = y \sqcup x$	–	0.1	0.8	–
(E16)	$\bigsqcup(\{\bigsqcup\{x\}\} \cup (\{\bigsqcup\{y\}\} \cup \{\bigsqcup\{z\}\})) = \bigsqcup(\{x\} \cup (\{y\} \cup \{z\}))$	–	–	–	–
(E17)	$\bigsqcup((\{\bigsqcup\{x\}\} \cup \{\bigsqcup\{y\}\}) \cup \{\bigsqcup\{z\}\}) = \bigsqcup((\{x\} \cup \{y\}) \cup \{z\})$	–	–	–	–
(E18)	$(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$	–	–	–	–
(E19)	$x \sqcup 0 = x$	–	–	–	–
(E20)	$x \leq x \sqcup y$	–	–	–	–
(E21)	$x \in X \Rightarrow \bigsqcup X = \bigsqcup(X \cup \{x\})$	–	0.1	0.6	–
(E22)	$x \in X \Rightarrow \bigsqcup X = \bigsqcup(\{\bigsqcup X\} \cup \{x\})$	–	–	–	–
(E23)	$x \in X \Rightarrow \bigsqcup X = \bigsqcup X \sqcup x$	–	–	–	–
(E24)	$x \in X \Rightarrow x \leq \bigsqcup X$	–	–	–	–
(E25)	$X \subseteq Y \Rightarrow \bigsqcup X \leq \bigsqcup Y$	–	–	–	–
(E26)	$\{x \cdot y : x \in X, y \in \emptyset\} = \emptyset$	3.2	0.1	0.2	0.3
(E27)	$\{x \cdot y : x \in \emptyset, y \in Y\} = \emptyset$	3.3	0.1	0.2	0.3
(E28)	$\bigsqcup\{z\} \cdot \bigsqcup\emptyset = \bigsqcup\{x \cdot y : x \in \{z\}, y \in \emptyset\}$	3.3	–	0.3	18.8
(E29)	$0 \cdot \{z\} = \bigsqcup\{x \cdot y : x \in \emptyset, y \in \{z\}\}$	3.5	–	0.7	–
(E30)	$x \cdot 0 = 0$	–	–	–	–
(E31)	$0 \cdot x = 0$	–	–	–	–
(E32)	$\{x' \cdot y' : x' \in \{x\}, y' \in \{y,z\}\} = \{x \cdot y, x \cdot z\}$	3.5	0.1	–	0.3
(E33)	$\{x' \cdot y' : x' \in \{x,y\}, y' \in \{z\}\} = \{x \cdot z, y \cdot z\}$	3.4	0.1	–	0.3
(E34)	$x \cdot \bigsqcup(\{y\} \cup \{z\}) = \bigsqcup\{x' \cdot y' : x' \in \{x\}, y' \in \{y,z\}\}$	3.7	–	3.5	–
(E35)	$x \cdot (y \sqcup z) = x \cdot y \sqcup x \cdot z$	–	–	–	–
(E36)	$(\bigsqcup(\{x\} \cup \{y\})) \cdot z = \bigsqcup\{x' \cdot y' : x' \in \{x,y\}, y' \in \{z\}\}$	3.9	–	3.6	–
(E37)	$(x \sqcup y) \cdot z = x \cdot z \sqcup y \cdot z$	–	–	–	–
(E38)	$(x \sqcup y) \sqcap z = x \sqcap z \sqcup y \sqcap z$	–	–	–	–
(E39)	$x \sqcap (y \sqcup z) = x \sqcap y \sqcup x \sqcap z$	–	–	–	–
(E40)	$\bigsqcap\{0\} = 0$	–	–	–	–
(E41)	$\bigsqcap\emptyset = \top$	3.2	–	–	–
(E42)	$\bigsqcap\{x\} = x$	–	–	–	–
(E43)	$\bigsqcap(\{x\} \cap \emptyset) = \top$	3.2	–	–	–
(E44)	$\bigsqcap(\{\bigsqcap\{x\}\} \cup \{\bigsqcap\{y\}\}) = \bigsqcap(\{x\} \cup \{y\})$	–	–	–	–
(E45)	$\bigsqcap(\{\bigsqcap X\} \cup \{\bigsqcap Y\}) = \bigsqcap(X \cup Y)$	–	–	–	–
(E46)	$\bigsqcap(\{\bigsqcap\{x\}\} \cup \{\bigsqcap\emptyset\}) = x$	–	–	–	–
(E47)	$\bigsqcap(\bigsqcap\{x\} \sqcap \bigsqcap\emptyset) = x$	–	–	–	–
(E48)	$\top \cdot \top = \top$	–	–	–	–
(E49)	$x \leq \top$	–	–	–	–
Proved		18	8	16	8

Table 1: Comparison of ATP systems for basic properties of quantals

```
thf(unionset_union,axiom,(
   ! [X: $i > $o, Y: $i > $o] : (
     ( unionset @ ( setofset @ X @ Y ) ) = ( union @ X @ Y ) ) )).
thf(sup_unionset_setofset,axiom,(
   ! [X: $i > $o, Y: $i > $o] : (
     ( sup @ ( unionset @ ( setofset @ X @ Y ) ) ) =
       ( sup @ ( unionset @ ( setofset @ ( singleton @ ( sup @ X ) ) @
       ( singleton @ ( sup @ Y ) ) ) ) ) ) )).
```

By this approach at least Isabelle was able to show this property.

For our experiment, we further encoded simple properties like (finite) associativity of $\bigsqcup$ (E18) or both annihilation laws ((E30), (E31)). None of the four ATP systems were able to show the theorems directly. This behaviour is a bit surprising since a proof by hand for (E30) simply uses Axioms (1), (2) and (6):

$$x \cdot 0 = \bigsqcup\{x\} \cdot \bigsqcup\emptyset = \bigsqcup\{x_1 \cdot x_2 : x_1 \in \{x\}, x_2 \in \emptyset\} = \bigsqcup\emptyset = 0 \,.$$

Therefore we extracted some steps of hand-written proofs and tried to show these separately. Using for example (E16) and (E17) as additional assumptions Isabelle is able to show (E18). With similar tricks we were able to prove all theorems of Table 1 fully automatically. This implies that one should add more properties than the pure axioms as assumptions.

These initial tasks with all the systems allow us to select the most powerful system for future applications, which are IsabelleP and Satallax at the moment. None of the ATP systems we included into our evaluation was even able to show half of the theorems we encoded. This could be due to an inappropriate encoding of our operations or due to inappropriate search strategies of the theorem provers used. For example the definition of `supset` gives the impression that the introduction of existentially quantified set variables Y leads to blind search and consequently bad results by increasing the state space.

6 Prospective Applications

Based on the given encoding we have proved a basic theorem kit for quantales. Together with the axioms the proved properties can be used as assumptions for automated theorem proving. In this section we sketch some of the prospective fields of applications where quantales are used and where our approach could be applied. Due to the complexity of these problems, tackling them seem not to be feasible with of-the-shelf theorem provers at the moment. However a further step in the evolution of fully automated higher-order ATP systems would enable us to perform these tasks.

Mathematics: Applications in mathematics are straightforward. As described in Section 2, quantales generalise various lattices of multiplicative ideals from ring theory as well as point free topologies and functional analysis. All these areas are possible places where THF can now be applied. Before tackling these problems one should start to verify more basic results on quantales and lattices as given in the foundational papers of Mulvey [23] and Rosenthal [27].

Logics: Quantales also occur in various logics. For computer scientist the branching time logic CTL* and its sub-logics CTL and LTL are the most prominent ones. In [22], a correspondence between quantales and these temporal logics is given. However, to realise reasoning in this setting arithmetic is needed since formulas like

$$\bigsqcup_{j\geq 0} \left(x^j \cdot y \sqcap \bigsqcap_{k<j} x^k \cdot z\right)$$

occur. We are not aware of any possibility of encoding properties like this without using arithmetics. For physicists, (non-commutatitive) linear logic is more suitable — its connection to quantales is discussed in [32]. Last but not least we want to mention that quantales are also used for reasoning about dynamic epistemic logic [3, 4]. This logic is used to model multi-agent systems and phenomena in philosophy.

Computer science: Besides reasoning in logic, quantales have further applications. For this paper we only mention hybrid systems — heterogeneous systems characterised by the interaction of discrete and continuous dynamics [14]. Algebraic reasoning with quantales can be used to verify properties about safety and liveness at an abstract level.

Physics: Originally, quantales were derived for modelling phenomena of quantum mechanics [8]. But quantales can also be used to formalise quantum logic — a logic defined for quantum physics [19, 25].

This closes our small list of prospective new applications for quantales where automated reasoning could be applied. However, as mentioned before, tackling these problems is not feasible at the moment since the performance of automated higher-order theorem provers is not yet sufficient.

A further application might be quantum computing since this is based on quantum mechanics. However, at the moment we are not aware of any formal treatment of quantum computing using quantales.

7 Conclusion and Outlook

We presented an approach to bring the algebraic structure of quantales into the realm of automated reasoning. This was done by using the higher-order approach of TPTP. In particular we presented an encoding in the typed higher-order form THF from which it was possible to prove a basic set of theorems about quantales. However, practical experience shows that at the moment only simple theorems can be proved; more complex properties need more assumptions as input or better search strategies for the ATP systems involved. We also presented prospective new applications for automated reasoning.

To perform the proof experiment, we used a set-based axiomatisation of quantales given by Conway. For future work, it would be interesting to investigate more suitable axiomatisations and more efficient encodings for the THF core since difficult theorems still need extra lemmas for full automation. Another research question is of course whether more efficient search strategies w.r.t. reasoning within quantales exist. To support the development of fully automated higher-order ATP systems, quantales will be part of the TPTP library v.4.1.0. This step hopefully helps to improve higher-order ATP systems for reasoning in algebraic structures similar to quantales within the near future.

Acknowledgements: We thank B. Möller, G. Sutcliffe and R. Glück for fruitful discussions and remarks.

References

[1] P. B. Andrews and C. E. Brown. TPS: A hybrid automatic-interactive system for developing proofs. *Journal of Applied Logic*, 4(4):367–395, 2006.

[2] J. Backes and C. Brown. Analytic tableaux for higher-order logic with choice. In J. Giesl and R. Haehnle, editors, *Automated Deduction — CADE-22*, LNAI, 2010. (to appear).

[3] A. Baltag, B. Coecke, and M. Sadrzadeh. Reasoning about dynamic epistemic logic. In W. van der Hoek, editor, *European Workshop on Multi-Agent Systems*, pages 605–614, 2004.

[4] A. Baltag, B. Coecke, and M. Sadrzadeh. Algebra and sequent calculus for epistemic actions. *ENTCS*, 126:27–52, 2005.

[5] C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. The LEO-II project. In *Proceedings of the Fourteenth Workshop on Automated Reasoning, Bridging the Gap between Theory and Practice*, 2007.

[6] C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0 — The Core of the TPTP Language for Higher-order Logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Deduction*, number 5159 in LNAI, pages 491–506. Springer, 2008.

[7] C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Combined reasoning by automated cooperation. *Journal of Applied Logic*, 6(3):318–342, 2008.

[8] G. Birkhoff and J. von Neumann. The logic of quantum mechanics. *Annals of Mathematics*, 37:823–843, 1936. Reprint in [19].

[9] J. Bos. Applied Theorem Proving - Natural Language Testsuite. http://www.coli.uni-sb.de/ bos/atp/, 2000.

[10] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

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

[12] H.-H. Dang, P. Höfner, and B. Möller. Towards algebraic separation logic. In R. Berghammer, A. Jaoua, and B. Möller, editors, *Relations and Kleene Algebra in Comp. Science*, volume 5827 of *LNCS*. Springer, 2009.

[13] P. Höfner. Database for automated proofs of Kleene algebra. `http://www.kleenealgebra.de` (accessed July 5, 2010).

[14] P. Höfner. *Algebraic Calculi for Hybrid Systems*. Books on Demand GmbH, 2009.

[15] P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In F. Pfennig, editor, *Automated Deduction*, volume 4603 of *LNAI*, pages 279–294. Springer, 2007.

[16] P. Höfner and G. Struth. On automating the calculus of relations. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning*, volume 5159 of *LNCS*, pages 50–66. Springer, 2008.

[17] P. Höfner, G. Struth, and G. Sutcliffe. Automated verification of refinement laws. *Annals of Mathematics and Artificial Intelligence, Special Issue on First-order Theorem Proving*, pages 35–62, 2008.

[18] A. Hommersom, P. Lucas, and P. van Bommel. Automated Theorem Proving for Quality-checking Medical Guidelines. In G. Sutcliffe, B. Fischer, and S. Schulz, editors, *Workshop on Empirically Successful Classical Automated Reasoning*, 2005.

[19] C. A. Hooker, editor. *The Logico-algebraic Approach to Quantum Mechanics*. D. Reidel Pub. Co., 1975.

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

[21] W. W. McCune. Prover9 and Mace4. <`http://www.cs.unm.edu/~mccune/prover9`>. (accessed July 5, 2010).

[22] B. Möller, P. Höfner, and G. Struth. Quantales and temporal logics. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, pages 263–277. Springer, 2006.

[23] C. Mulvey. &. *Rendiconti del Circolo Matematico di Palermo*, 12(2):99–104, 1986.

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

[25] J. Paseka and J. Rosicky. Quantales. In B. Coecke, D. Moore, and A. Wilce, editors, *Current Research in Operational Quantum Logic: Algebras, Categories and Languages*, volume 111 of *Fundamental Theories of Physics*, pages 245–262. Kluwer, 2000.

[26] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Communications*, 15(2-3):91–110, 2002.

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

[28] G. Sutcliffe. System description: SystemOnTPTP. In D. McAllester, editor, *Automated Deduction*, volume 1831 of *LNAI*, pages 406–410. Springer, 2000.

[29] G. Sutcliffe and C. Benzmüller. Automated reasoning in higher-order logic using the TPTP THF infrastructure. *Journal of Formalized Reasoning*, 2010. (to appear).

[30] G. Sutcliffe and C. Suttner. Evaluating general purpose automated theorem proving systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.

[31] A. Wojcik. Formal Design Verification of Digital Systems. In *20th Design Automation Conference*, 1983.

[32] D. N. Yetter. Quantales and (noncommutative) linear logic. *Journal of Symbolic Logic*, 55(1):41–64, 1990.

A Complete THF-Encoding of Quantales

```
% --- Empty Set
    thf(emptyset_decl,type,(
      emptyset: $i > $o )).
    thf(emptyset,definition,
      ( emptyset  = ( ^ [X: $i] : $false ) )).
% --- Singleton Set
    thf(singleton_decl,type,(
      singleton: ( $i > $i > $o ) )).
    thf(singleton,definition,
      ( singleton  = ( ^ [X: $i,U: $i] : ( U = X ) ) )).
% --- Supremum
    thf(zero,type,(
      zero: $i )).
    thf(sup,type,(
      sup: ( ( $i > $o ) > $i ) )).
    thf(sup_es,axiom,(
      (sup @ emptyset) = zero )).
    thf(sup_singleset,axiom,(
      ! [X: $i] : ( ( sup @ ( singleton @ X ) ) = X ) )).
    thf(supset,type,(
      supset: ( ( ( $i > $o ) > $o ) > $i > $o ) )).
    thf(supset,definition,
      ( supset = ( ^ [F: ( $i > $o ) > $o, X: $i ] :
        ? [Y: $i > $o] : ( ( F @ Y ) & ( ( sup @ Y ) = X ) ) ) )).
    thf(unionset,type,(
      unionset: ( ( ( $i > $o ) > $o ) > $i > $o ) )).
    thf(unionset,definition,
      ( unionset  = ( ^ [F: ( $i > $o ) > $o, X: $i ] :
        ( ? [Y: $i > $o] : ( ( F @ Y ) & ( Y @ X ) ) ) ) )).
    thf(sup_set,axiom,(
       ! [X: ( $i > $o ) > $o] : ( ( sup @ ( supset @ X ) ) =
                                   ( sup @ ( unionset @ X ) ) ) )).
% --- Multiplication
    thf(multiplication,type,(
      multiplication: $i > $i > $i )).
    thf(crossmult,type,(
      crossmult: ( $i > $o ) > ( $i > $o ) > $i > $o )).
    thf(crossmult_def,definition,(
      crossmult = ( ^ [X: $i > $o,Y: $i > $o, A: $i] : (
        ? [X1: $i, Y1: $i] : ( ( X @ X1 ) & ( Y @ Y1 ) & ( A = ( multiplication @ X1 @ Y1 ) ) )
      ) ) )).
    thf(multiplication_sup,axiom,(
        ! [X: $i > $o, Y: $i > $o] : ( ( multiplication @ ( sup @ X ) @ ( sup @ Y ) )
        = ( sup @ ( crossmult @ X @ Y ) ) ) )).
    thf(one,type,(
      one: $i )).
    thf(multiplication_neutralr,axiom,(
        ! [X: $i] : ( ( multiplication @ X @ one ) = X ) )).
    thf(multiplication_neutrall,axiom,(
        ! [X: $i] : ( ( multiplication @ one @ X ) = X ) )).
```

Fast Decision Procedure for Propositional Dummett Logic Based on a Multiple Premise Tableau Calculus

Guido Fiorino
Dipartimento di Metodi Quantitativi per le Scienze Economiche ed Aziendali,
Università di Milano-Bicocca, Piazza dell'Ateneo Nuovo, 1,
20126 Milano, Italy.
guido.fiorino@unimib.it

Abstract

We present a procedure to decide propositional Dummett logic. This procedure relies on a tableau calculus with a multiple premise rule and optimizations. The resulting implementation outperforms the state of the art graph-based procedure.

1 Introduction

In this note a tableau calculus, some optimizations and an implementation to decide propositional Dummett logic are described.

Dummett logic can be axiomatized by adding to any proof system for propositional intuitionistic logic the axiom scheme $(p \to q) \vee (q \to p)$. It has a well-known semantical characterization by linearly ordered Kripke models, thus Dummett logic is also known as Linear Chain logic. Gödel studied finite approximations of Dummett Logic ([15]), namely the sequence G_n, $n \geq 1$, of logics that are semantically characterized by linearly ordered Kripke models with at most n worlds. For this reason another name for the logic under consideration is Gödel-Dummett Logic. Dummett Logic has been considered by people interested in computer science [4, 5] and many valued logics [7, 8]. In [17] it has been recognized as an important fuzzy logic. We quote [9] for a thorough treatment of the subject.

In the late '90 were presented tableau [1, 12] and sequent calculi [11] for propositional Dummett logic. These calculi are believed to be highly inefficient to the purpose of performing practical theorem proving ([5, 20]), mainly because they contain a multiple premise rule that in the worse case analysis gives rise to tableau proofs having a factorial number of branches with respect to the number of formulas in the premise. To get rid of the multiple premise rule, paper [5] proposes to exploit the following logical equivalences: $A \to (B \vee C) \equiv (A \to B) \vee (A \to C)$, $A \to (B \wedge C) \equiv (A \to B) \wedge (A \to C)$, $(A \vee B) \to C \equiv (A \to C) \wedge (B \to C)$ and $(A \wedge B) \to C \equiv (A \to C) \vee (B \to C)$. In the recent [21] it is proved that the deduction in Dummett logic can be reduced to the construction of a graph and an implementation has been developed.

Recent investigations in propositional intuitionist logics have introduced optimization techniques that dramatically reduce the running time ([2]). Such techniques can be extended to other logics. The aim of this note is to show that the tableau calculi based on a multiple premise rule plus optimization techniques give rise to a fast decision procedure for propositional Dummett logic. Our fast implementation is based on three main ideas that are motivated in this note: (i) a new tableau calculus. Compared with [1, 12], our tableau calculus provides a new treatment of negated formulas and a new multiple premise rule; (ii) the optimization technique Simplification, first described in [22]. This optimization has been employed in [2] to improve automated deduction in propositional intuitionistic logic; (iii) the equivalences for Dummett logic quoted above. The equivalences are employed to reduce the number of branches generated by the multiple premise rule. They are used in the opposite style with respect to [5, 6, 21]. On the ideas described in this note (full details with proofs in [13]) we have developed a prolog implementation that is faster than the state of the art graph-based procedure of [21].

$$\frac{S,\mathbf{T}(A\wedge B)}{S,\mathbf{T}A,\mathbf{T}B}\mathbf{T}\wedge \quad \frac{S,\mathbf{F}(A\wedge B)}{S,\mathbf{F}A|S,\mathbf{F}B}\mathbf{F}\wedge \quad \frac{S,\mathbf{F_c}(A\wedge B)}{S,\mathbf{F_c}A|S,\mathbf{F_c}B}\mathbf{F_c}\wedge \quad \frac{S,\mathbf{T_{cl}}(A\wedge B)}{S,\mathbf{T_{cl}}A,\mathbf{T_{cl}}B}\mathbf{T_{cl}}\wedge$$

$$\frac{S,\mathbf{T}(A\vee B)}{S,\mathbf{T}A|S,\mathbf{T}B}\mathbf{T}\vee \quad \frac{S,\mathbf{F}(A\vee B)}{S,\mathbf{F}A,\mathbf{F}B}\mathbf{F}\vee \quad \frac{S,\mathbf{F_c}(A\vee B)}{S,\mathbf{F_c}A,\mathbf{F_c}B}\mathbf{F_c}\vee \quad \frac{S,\mathbf{T_{cl}}(A\vee B)}{S,\mathbf{T_{cl}}A|S,\mathbf{T_{cl}}B}\mathbf{T_{cl}}\vee$$

$$\frac{S,\mathbf{T}(\neg A)}{S,\mathbf{F_c}A}\mathbf{T}\neg \quad \frac{S,\mathbf{F}(\neg A)}{S,\mathbf{T_{cl}}A}\mathbf{F}\neg \quad \frac{S,\mathbf{F_c}(\neg A)}{S,\mathbf{T_{cl}}A}\mathbf{F_c}\neg \quad \frac{S,\mathbf{T_{cl}}(\neg A)}{S,\mathbf{F_c}A}\mathbf{T_{cl}}\neg$$

$$\frac{S,\mathbf{T}A,\mathbf{T}(A\to B)}{S,\mathbf{T}A,\mathbf{T}B}\mathbf{T}\to \quad \frac{S,\mathbf{F_c}(A\to B)}{S,\mathbf{T_{cl}}A,\mathbf{F_c}B}\mathbf{F_c}\to \quad \frac{S,\mathbf{T_{cl}}(A\to B)}{S,\mathbf{F_c}A|S,\mathbf{T_{cl}}B}\mathbf{T_{cl}}\to$$

$$\frac{S,\mathbf{T}((A\wedge B)\to C)}{S,\mathbf{T}(A\to(B\to C))}\mathbf{T}\to\wedge \quad \frac{S,\mathbf{T}(\neg A\to B)}{S,\mathbf{T_{cl}}A|S,\mathbf{T}B}\mathbf{T}\to\neg \quad \frac{S,\mathbf{T}((A\vee B)\to C)}{S,\mathbf{T}(A\to C),\mathbf{T}(B\to C)}\mathbf{T}\to\vee$$

$$\frac{S,\mathbf{T}((A\to B)\to C)}{S,\mathbf{F}(A\to p),\mathbf{T}(p\to C),\mathbf{T}(B\to p)|S,\mathbf{T}C}\mathbf{T}\to\to \text{ with } p \text{ a new atom}$$

Figure 1: The invertible rules of $\mathbb{D}$.

2 Basic Definitions

We consider the propositional language based on a denumerable set of propositional variables $\mathscr{PV}$, the boolean constants $\top$ and $\bot$ and the logical connectives $\neg,\wedge,\vee,\to$. In the following, formulas (respectively set of formulas and propositional variables) are denoted by letters A, B, $C\ldots$ (respectively S, T, U,... and p, q, r,...) possibly with subscripts or superscripts.

A well-known semantical characterization of Dummett logic (**Dum**) is by *linearly ordered Kripke models*. In this note *model* means a linearly ordered Kripke model, namely a structure $\underline{K}=\langle P,\leq,\rho,\Vdash\rangle$, where $\langle P,\leq,\rho\rangle$ is a linearly ordered set with minimum ρ and $\Vdash$ is the *forcing relation*, a binary relation on $P\times(\mathscr{PV}\cup\{\top,\bot\})$ such that: (i) if $\alpha\Vdash p$ and $\alpha\leq\beta$, then $\beta\Vdash p$; (ii) for every $\alpha\in P$, $\alpha\Vdash\top$ holds and $\alpha\Vdash\bot$ does not hold. Hereafter we denote the members of P with lowercase letters of the Greek alphabet.

The forcing relation is extended in a standard way to arbitrary formulas as follows: (i) $\alpha\Vdash A\wedge B$ iff $\alpha\Vdash A$ and $\alpha\Vdash B$; (ii) $\alpha\Vdash A\vee B$ iff $\alpha\Vdash A$ or $\alpha\Vdash B$; (iii) $\alpha\Vdash A\to B$ iff, for every $\beta\in P$ such that $\alpha\leq\beta$, $\beta\Vdash A$ implies $\beta\Vdash B$; (iv) $\alpha\Vdash\neg A$ iff, for every $\beta\in P$ such that $\alpha\leq\beta$, $\beta\Vdash A$ does not hold.

We write $\alpha\nVdash A$ when $\alpha\Vdash A$ does not hold. It is easy to prove that, for every formula A, the *persistence* property holds: if $\alpha\Vdash A$ and $\alpha\leq\beta$, then $\beta\Vdash A$. A formula A *is valid in a model* $\underline{K}=\langle P,\leq,\rho,\Vdash\rangle$ if and only if $\rho\Vdash A$. It is well-known (see e.g. [10]) that **Dum** coincides with the set of formulas valid in all models.

The rules of our calculus $\mathbb{D}$ for **Dum** are in Figures 1 and 2. The rules of $\mathbb{D}$ work on signed formulas, that is well-formed formulas prefixed with one of the *signs* $\{\mathbf{T},\mathbf{F},\mathbf{F_c},\mathbf{T_{cl}}\}$, and on sets of signed formulas (hereafter we omit the word "signed" in front of "formula" in all the contexts where no confusion arises).

The semantical meaning of the signs is explained by means of the *realizability* relation ($\rhd$) defined as follows. Let $\underline{K}=\langle P,\leq,\rho,\Vdash\rangle$ be a model, let $\alpha\in P$, let H be a signed formula and let S be a set of signed formulas. We say that α *realizes* H, α *realizes* S and $\underline{K}$ *realizes* S, and we write $\alpha\rhd H$, $\alpha\rhd S$ and $\underline{K}\rhd S$, respectively, if the following conditions hold:

1. $\alpha\rhd\mathbf{T}A$ iff $\alpha\Vdash A$;
2. $\alpha\rhd\mathbf{F}A$ iff $\alpha\nVdash A$;
3. $\alpha\rhd\mathbf{F_c}A$ iff $\alpha\Vdash\neg A$;
4. $\alpha\rhd\mathbf{T_{cl}}A$ iff $\alpha\Vdash\neg\neg A$;

$$\frac{S, \mathbf{T_{cl}}A}{S_c, \mathbf{T}A}\mathbf{T_{cl}}\text{-Atom}$$

$$\frac{S, \mathbf{F}(A_1 \to B_1), \ldots, \mathbf{F}(A_n \to B_n)}{S_c, \mathbf{T}A_1, \mathbf{F}B_1, S^1_{\mathbf{F}\to} | S_c, \mathbf{T}A_2, \mathbf{F}B_2, S^2_{\mathbf{F}\to} | \ldots | S_c, \mathbf{T}A_n, \mathbf{F}B_n, S^n_{\mathbf{F}\to}}\mathbf{F}\to$$

$$\begin{array}{lcl} S_c & = & \{\mathbf{T}A | \mathbf{T}A \in S\} \cup \{\mathbf{F_c}A | \mathbf{F_c}A \in S\} \cup \{\mathbf{T_{cl}}A | \mathbf{T_{cl}}A \in S\}; \\ S^1_{\mathbf{F}\to} & = & \{\mathbf{F}(A_2 \to B_2), \ldots, \mathbf{F}(A_n \to B_n)\}, \\ S^i_{\mathbf{F}\to} & = & \{\mathbf{F}((A_1 \wedge B_i) \to B_1), \ldots, \mathbf{F}((A_{i-1} \wedge B_i) \to B_{i-1}), \mathbf{F}(A_{i+1} \to B_{i+1}), \ldots, \mathbf{F}(A_n \to B_n)\}, \\ & & \text{for } i = 2, \ldots, n. \end{array}$$

Figure 2: The non-invertible rules of $\mathbb{D}$.

5. $\alpha \rhd S$ iff α realizes every formula in S;

6. $\underline{K} \rhd S$ iff $\rho \rhd S$.

Since the meaning of $\mathbf{T}$, $\mathbf{F_c}$ and $\mathbf{T_{cl}}$ is related to the forcing of a formula and since, by the persistence property, the forced formulas are preserved upwards, we call *stable* the formulas signed with $\mathbf{T}$, $\mathbf{F_c}$ or $\mathbf{T_{cl}}$. As discussed in the following, stable formulas have a central role in the organization of our deduction strategy. We point out that $\mathbf{F_c}A$ and $\mathbf{T_{cl}}A$ are synonym $\mathbf{T}\neg A$ and $\mathbf{T}\neg\neg A$ respectively. If $\mathbf{F_c}A$ holds in a world of a Kripke model, then A is *certainly not forced in the future*. If $\mathbf{T_{cl}}A$ holds in a world of a Kripke model $\underline{K}$, then A is forced in the maximum of $\underline{K}$, which is a world semantically behaving as a *classical model*. We could rewrite the rules of the calculus by using only the signs $\mathbf{T}$ and $\mathbf{F}$. In such a calculus the $\mathbf{F_c}$-rules are replaced by rules treating negated formulas and the rules for $\mathbf{T_{cl}}$ are replaced by rules treating double negated formulas. We prefer this object language because it makes the rules less cumbersome than the object language with two signs only.

From the meaning of the signs we get the conditions that make a set of formulas inconsistent. A set S is *inconsistent* iff $\{\mathbf{T}A, \mathbf{F}A\} \subseteq S$, $\{\mathbf{T}A, \mathbf{F_c}A\} \subseteq S$, $\{\mathbf{F_c}A, \mathbf{T_{cl}}A\} \subseteq S$, $\mathbf{T}\bot \in S$, $\mathbf{F}\top \in S$, $\mathbf{F_c}\top \in S$ or $\mathbf{T_{cl}}\bot \in S$. It is easy to prove the following result:

Theorem 1. *If a set of formulas S is inconsistent, then for every Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ and for every $\alpha \in P$, $\alpha \not\rhd S$.*

We refer to [16] for a full presentation of tableaux systems. A *closed proof table* is a proof table whose leaves are all inconsistent sets. A closed proof table is a proof of the calculus and a formula A is provable iff there exists a closed proof table for $\{\mathbf{F}A\}$.

The calculus $\mathbb{D}$ has two non-invertible rules, namely $\mathbf{F}\to$ and $\mathbf{T_{cl}}$-Atom. Rule $\mathbf{F}\to$ is inspired to the rule of [1]. Rule $\mathbf{T_{cl}}$-Atom can be explained as follows: let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a model and let $\alpha \in P$ such that $\alpha \Vdash \neg\neg p$. Let ϕ the maximum with respect to $\langle P, \leq, \rho \rangle$. Then $\phi \Vdash p$. We notice that nothing can be concluded about the forcing in α of p, whereas if $\alpha \Vdash \neg\neg A$, with A a non-atomic formula, we have information about the forcing in α of the subformulas of A. This explains the $\mathbf{T_{cl}}$-rules in Figure 1.

3 Correctness

To prove the correctness of $\mathbb{D}$ with respect to Dummett logic we need to prove that, if there exists a closed proof table for $\{\mathbf{F}A\}$, then A is a valid formula in Dummett logic. The main step is to prove that the rules of the calculus preserve realizability:

Proposition 1. *For every rule of $\mathbb{D}$, if a model realizes the premise, then there exists a model realizing at least one of the conclusions.*

Proof: we provide the proof for $\mathbf{F}\rightarrow$, the other cases being trivial. If the premise of $\mathbf{F}\rightarrow$ is realized, then there exist $\alpha_1,\ldots,\alpha_n$ elements of a model $\underline{K}$ such that $\alpha_i \rhd \mathbf{T}A_i, \mathbf{F}B_i$, for $i=1,\ldots,n$. Let $\beta_i = \max\{\alpha | \alpha \rhd \mathbf{T}A_i, \mathbf{F}B_i\}$, for $i=1,\ldots,n$, thus, β_i is the maximal element such that $\beta_i \rhd \mathbf{T}A_i, \mathbf{F}B_i$. The structure $\underline{K}' = \langle \{\beta_1,\ldots,\beta_n\}, \leq, \rho', \Vdash \rangle$, with $\rho' = \min\{\beta_1,\ldots,\beta_n\}$, is a model such that $\rho' \rhd S_c, \mathbf{T}A_j, \mathbf{F}B_j, S_{\mathbf{F}_\rightarrow}$ for some $j \in \{1,\ldots,n\}$. Let $m = \min\{i \in \{1,\ldots,n\} | \beta_i = \rho'\}$. If $m=1$, then the leftmost conclusion of $\mathbf{F}\rightarrow$ is realized. If $m>1$, then $\beta_1,\ldots,\beta_{m-1} > \beta_m$. Let $\beta_k = \min\{\beta_1,\ldots,\beta_{m-1}\}$. It follows that $\beta_k \rhd \mathbf{T}A_k, \mathbf{F}B_k, \mathbf{T}B_m$. Thus $\beta_m \rhd \mathbf{F}((A_i \wedge B_m) \rightarrow B_i)$, for $i=1,\ldots,m-1$ and this implies that the m-th conclusion of the rule is realized. □

The idea behind the rule $\mathbf{F}\rightarrow$ can be explained as follows: If the j-th conclusion of the rule **Dum** of [1] is realizable and no model realizes the first $j-1$ conclusions, then $\alpha_j > \alpha_1,\ldots,\alpha_{j-1}$ holds. This also implies that $\alpha_1,\ldots,\alpha_{j-1} \rhd \mathbf{T}B_j$, hence $\alpha_j \rhd \mathbf{F}((A_k \wedge B_j) \rightarrow B_k)$, for $k=1,\ldots,j-1$ and this proves that the j-th conclusion of $\mathbf{F}\rightarrow$ is realized.

Remark 1. We call *side information* the formula B_j added in the j-th conclusion of $\mathbf{F}\rightarrow$ as conjunct in $\mathbf{F}((A_i \wedge B_j) \rightarrow B_i)$, for $i=1,\ldots,j-1$. The side information arises from the knowledge that the left-hand side conclusions are not realizable. This information is correct but it is not necessary to get the completeness. The notion of side information is introduced to reduce the search space by means of the *simplification technique* which is described in Section 4.

Remark 2. The rule $\mathbf{F}\rightarrow$ can also be given in the following form

$$\frac{S, \mathbf{F}(A_1 \rightarrow B_1), \ldots, \mathbf{F}(A_n \rightarrow B_n)}{S_c, \mathbf{T}A_1, \mathbf{F}B_1, S^1_{\mathbf{F}_\rightarrow} | S_c, \mathbf{T}A_2, \mathbf{F}B_2, S^2_{\mathbf{F}_\rightarrow} | \ldots | S_c, \mathbf{T}A_n, \mathbf{F}B_n, S^n_{\mathbf{F}_\rightarrow}} \mathbf{F}\rightarrow_{new}$$

$$\begin{array}{lcl} S_c & = & \{\mathbf{T}A | \mathbf{T}A \in S\} \cup \{\mathbf{F_c}A | \mathbf{F_c}A \in S\} \cup \{\mathbf{T_{cl}}A | \mathbf{T_{cl}}A \in S\}; \\ S^1_{\mathbf{F}_\rightarrow} & = & \{\mathbf{F}(A_2 \rightarrow B_2), \ldots, \mathbf{F}(A_n \rightarrow B_n)\}, \\ S^i_{\mathbf{F}_\rightarrow} & = & \{\mathbf{T}(p \rightarrow B_i), \mathbf{F}(A_1 \wedge p \rightarrow B_1), \ldots, \mathbf{F}(A_{i-1} \wedge p \rightarrow B_{i-1}), \mathbf{F}(A_{i+1} \rightarrow B_{i+1}), \ldots, \mathbf{F}(A_n \rightarrow B_n)\}, \\ & & \text{with } p \text{ a new propositional variable,} \\ & & \text{for } i = 2,\ldots,n. \end{array}$$

The correctness of $\mathbf{F}\rightarrow$-new can be easily obtained from $\mathbf{F}\rightarrow$ following the proof of correctness given in [12] for the rules $\mathbf{T}\rightarrow\rightarrow$ and $\mathbf{T}\rightarrow\vee$ (it can also be noticed that it is applied the indexing technique consisting in replacing a formula with a new propositional variable). This version highlights that the side information B_i is treated by the rules of the calculus once.

By the above proposition:

Theorem 2 (Soundness of $\mathbb{D}$)**.** *If there exists a proof of a formula A, then A is valid in every model.*

4 Rules to Optimize the Proof Search

The tableau rules in Figures 1 and 2 are the core of $\mathbb{D}$. Now we discuss some rules, introduced to reduce the size of the proofs. We note that $\mathbb{D}$ improves the known multiple premise calculi [1, 12] by two aspects: the rule $\mathbf{F}\rightarrow$ and the rules to treat $\mathbf{T_{cl}}$-formulas.

Simplification Simplification is an effective optimization both in classical and intuitionistic logic. In the framework of tableau systems Simplification has been introduced in [22], where it is applied to classical and modal logics. Recently it has been fitted to intuitionistic logic ([2]). Simplification is based

$$\frac{S, \mathbf{T}A}{S[A/\top], \mathbf{T}A}\,\text{Replace}\,\mathbf{T} \qquad \frac{S, \mathbf{F_c}A}{S[A/\bot], \mathbf{F_c}A}\,\text{Replace}\,\mathbf{F_c} \qquad \frac{S, \mathbf{T_{cl}}A}{S[\neg A/\bot], \mathbf{T_{cl}}A}\,\text{Replace}\,\mathbf{T_{cl}}$$

Figure 3: Replacement rules

on the well-known replacement rules (see e.g. [19]) consisting in replacing a formula with a logically equivalent one. Our adaptation of Simplification to the object language of $\mathbb{D}$ consists in the replacement rules of Figure 3. It is an easy task to check that the replacement rules preserve the realizability and are invertible. The logical constant $\top$ and $\bot$ are replaced by means of the *simplification rules* consisting in the usual boolean simplification rules for conjunction and disjunction, plus the simplification rules for intuitionistic negation and implication.

The semantical meaning of $\top$ and $\bot$ implies that replacements affect neither the correctness nor the completeness, thus these replacements rules can be applied at any step of a tableau proof. Hereafter with Simplification we mean the set of rules described in this section, including the usual boolean simplification rules. We point out that the use of Simplification can reduce considerably the search-space ([22] and [2] give an account for propositional classical and intuitionistic logics, respectively).

Reducing the branching of the multiple premise rule $\mathbf{F}\rightarrow$ In [21] the equivalences

$$A \rightarrow (B \vee C) \equiv (A \rightarrow B) \vee (A \rightarrow C),\ A \rightarrow (B \wedge C) \equiv (A \rightarrow B) \wedge (A \rightarrow C)$$

$$(A \vee B) \rightarrow C \equiv (A \rightarrow C) \wedge (B \rightarrow C),\ \text{and}\ (A \wedge B) \rightarrow C \equiv (A \rightarrow C) \vee (B \rightarrow C)$$

are exploited to get the rules of Figure 4.

$$\frac{S, \mathbf{T}(A \rightarrow (B \vee C))}{S, \mathbf{T}(A \rightarrow B) | S, \mathbf{T}(A \rightarrow C)} \qquad \frac{S, \mathbf{T}((A \vee B) \rightarrow C)}{S, \mathbf{T}(A \rightarrow C), \mathbf{T}(B \rightarrow C)} \qquad \frac{S, \mathbf{T}((A \wedge B) \rightarrow C)}{S, \mathbf{T}(A \rightarrow C) | S, \mathbf{T}(B \rightarrow C)} \qquad \frac{S, \mathbf{T}(A \rightarrow (B \wedge C))}{S, \mathbf{T}(A \rightarrow B), \mathbf{T}(A \rightarrow C)}$$

$$\frac{S, \mathbf{F}(A \rightarrow (B \wedge C))}{S, \mathbf{F}(A \rightarrow B) | S, \mathbf{F}(A \rightarrow C)} \qquad \frac{S, \mathbf{F}(A \rightarrow (B \vee C))}{S, \mathbf{F}(A \rightarrow B), \mathbf{F}(A \rightarrow C)} \qquad \frac{S, \mathbf{F}((A \vee B) \rightarrow C)}{S, \mathbf{F}(A \rightarrow C) | S, \mathbf{F}(B \rightarrow C)} \qquad \frac{S, \mathbf{F}((A \wedge B) \rightarrow C)}{S, \mathbf{F}(A \rightarrow C), \mathbf{F}(B \rightarrow C)}$$

Figure 4: Rules of [6, 21] to treat implicative formulas.

The rules of Figure 4 allow to reduce implicative formulas to implicative atomic formulas, that is implicative formulas whose antecedent and consequent are propositional variables. The problem of deciding a set of formulas containing atomic implicative formulas and atomic formulas only is reducible to the problem of reachability on a graph [21]. To get sets whose implicative formulas are atomic a price has to be paid. First, it is always necessary to treat $\mathbf{T}$-implicative formulas, and in the case of formulas of the kind $\mathbf{T}(p \rightarrow (B \vee C))$ and $\mathbf{T}((A \wedge B) \rightarrow C)$ branches are generated. Multiple premise calculi treat formulas of the kind $\mathbf{T}((A \wedge B) \rightarrow C)$ and $\mathbf{T}(p \rightarrow (B \vee C))$ by means of single-conclusion rules. Second, the $\mathbf{F}$-rules above decompose the $\mathbf{F}\rightarrow$-formulas either by increasing their number in the conclusion or by introducing a new branch. Note also that such rules do not introduce in the conclusions any stable information. As we discussed in the previous section, we consider useful from a practical perspective to discover stable information in order to reduce the search space.

With regard to the $\mathbf{F}\rightarrow$-formulas, the calculi of [6, 21] and $\mathbb{D}$ give rise to decision procedures behaving in the opposite way. The calculi provided in [6, 21] employ the rules in Figure 4 to insert in a

$$\frac{\mathbf{F}(A \to B), \mathbf{F}(A \to C)}{\mathbf{F}(A \to (B \vee C))} \qquad \frac{\mathbf{F}(A \to C), \mathbf{F}(B \to C)}{\mathbf{F}((A \wedge B) \to C)}$$

Figure 5: The factorization rules of $\mathbb{D}$.

set all the possible $\mathbf{F} \to$-formulas. The calculus $\mathbb{D}$ employs the rules in Figure 5 to reduce the number of $\mathbf{F} \to$-formulas in the sets. In the case of $\mathbb{D}$ the reason to reduce this number is related to the presence of the multiple premise rule $\mathbf{F} \to$, whose number of conclusions depends on the $\mathbf{F} \to$-formulas in its premise. We aim to devise a decision procedure using the rules in Figure 5 in order to reduce as much as possible the number of $\mathbf{F} \to$-formulas before to apply the rule $\mathbf{F} \to$.

The rules to handle $\mathbf{T_{cl}}$-formulas The calculus $\mathbb{D}$ has rules for the formulas signed with $\mathbf{T_{cl}}$, this amounts to have ad-hoc rules to treat $\mathbf{T}\neg\neg$-formulas, a kind of formulas for which [1] does not have ad-hoc rules but takes back to the rule $\mathbf{F_c}\neg$.

To avoid backtracking still preserving the completeness, in [1] the application of the rule $\mathbf{F_c}$ has to be deferred until no $\mathbf{F}$-rule is applicable. This means that the calculus defers to analyze the stable formulas of the kind $\mathbf{F_c}(\neg A)$ (in other words the calculus does not analyze the double negated formulas). On the contrary $\mathbb{D}$ analyzes the double negated formulas by means of $\mathbf{T_{cl}}$-rules. The advantage of analyzing formulas of the kind $\mathbf{F_c}(\neg A)$ for every case of A is that the discovery of new stable subformulas of $\mathbf{F_c}(\neg A)$ is not deferred to a stage when no other rule, included the rule $\mathbf{F} \to$, is applicable. The early discovery of stable formulas gives advantages since this information allows to shrink the search space by means of Simplification. Summarizing, the $\mathbf{T_{cl}}$-rules of $\mathbb{D}$ are the rules allowing to analyze $\mathbf{F_c}\neg$-formulas. Among them $\mathbf{T_{cl}}$-Atom is the only non-invertible rule. In Section 5 it is proved that to avoid backtracking it is sufficient to defer the application of $\mathbf{T_{cl}}$-Atom until no other rule is applicable.

We notice that the side information can also be characterized into the logic calculus by introducing appropriate connectives and signs along the ideas presented in Remark 4.

5 A Strategy to Decide Dummett Logic and Its Completeness

The implementation described in the next section uses the rules of the calculus according the following strategy. The non-invertible rules $\mathbf{F} \to$ and $\mathbf{T_{cl}}$-Atom are applied only when no invertible rule of Figures 1-3 is applicable. This strategy is necessary to avoid backtracking in proof search. Rule $\mathbf{F} \to$ is applied when no rule but $\mathbf{T_{cl}}$-Atom is applicable. This guarantees that the application of the non-invertible rule $\mathbf{F} \to$ is invertible, that is, no information necessary to the completeness is lost. Rules in Figure 5 allow to reduce the number of $\mathbf{F} \to$-formulas to be handled by $\mathbf{F} \to$, thus are applied before to apply $\mathbf{F} \to$. Finally rule $\mathbf{T_{cl}}$-Atom is applied. Every application of $\mathbf{T_{cl}}$-Atom is invertible because the $\mathbf{F}$-atomic formulas is the only information which is lost.

The decision procedure can be implemented in polynomial space by means of a depth-first strategy. Because to the side information introduced in the conclusion of $\mathbf{F} \to$, the depth of the deduction is quadratic in the formulas to be proved. As a matter of fact, to insert the side information, in every set S of the conclusion of $\mathbf{F} \to$ are introduced as many $\wedge$ connectives as the $\mathbf{F} \to$-formulas in the premise. This number is bounded by the size n of the formula to be decided. If the data structures of the implementation allow to store the formulas once, that is, independently of the number of their occurrences, then this is the new information that needs to be stored. Along a branch the number of applications of rule $\mathbf{F} \to$ is bounded by n. Since every application of $\mathbf{F} \to$ introduces in a set n new connectives $\wedge$ at most, it follows that to handle the side information, along a branch are introduced n^2 new $\wedge$ connectives at most.

As regards the rules treating the side information, we point out that the sign of a side information is **T**. This implies that all its occurrences in the set are replaced by $\top$. If the data structures represents the different occurrences of a formula once, then the replacement takes constant time. Every application of the simplification rules deletes one connective and takes constant time. The analysis of the other rules is straightforward, thus we conclude that, since along a branch the rule $\mathbf{F}\rightarrow$ introduces n^2 connectives $\wedge$, the depth of the deductions is at most quadratic. We also note that although the depth of a branch is $O(n^2)$, only $O(n)$ rules from Figures 1 and 2 are applied. This implies that as in the calculus of [1], the number of branches of the deductions is factorial in n.

Remark 3. An alternative to our approach is to decide Dummett logic via a decision procedure for propositional Intuitionistic logic. Paper [3] introduces the notion of Generalized Tableaux to decide intermediate logics. A Generalized Tableau is a tableau for propositional Intuitionistic logic plus a rule to be applied once as first rule of the deduction. The aim of this rule is to introduce formulas obtained by instantiating the axiom scheme of the logic under consideration. For the case of Dummett logic, to decide a given formula A, the special rule introduces the set of formulas obtained by instantiating in every possible way the propositional variables the axiom schemata $(p\rightarrow q)\vee(q\rightarrow p)$ with the formulas in $Rsf(A)$=$\{B|B$ is subformula of A and B is a propositional variable or $B\equiv C\rightarrow D$ or $B\equiv\neg C\}$. Since $|Rsf(A)|=O(|A|)$ and there are $|Rsf(A)|$ choices for p and q, it follows that the special rule introduces $O(|A|^2)$ formulas ($|A|$ denotes the cardinality of A). Thus the number of connectives to be handled in the deduction is $O(|A|^3)$. Paper [18] proves that propositional intuitionistic logic is decidable in $O(n\lg n)$-SPACE, hence this technique requires $O(n^3\lg n)$-SPACE and the depth of the deductions is $O(|A|^3)$.

6 The Implementation and the Performances

We devote this section to give an account of our implementation EPDL[1]. The first issue we face is how EPDL handles the side information. We emphasize that the side information B_j has sign **T**, this means that such instances of B_j will occur in the subsequent sets with sign **T**. The formula B_j occurring in $\mathbf{F}((A_i\wedge B_j)\rightarrow B_i)$ conveys the information that when $\mathbf{T}A_i$ and $\mathbf{F}B_i$ are realized, also $\mathbf{T}B_j$ has to be realized. Since the rules of Simplification are used, we want use the stable information $\mathbf{T}B_j$ and possibly the information derivable from $\mathbf{T}B_j$ to reduce the size of the proofs. The side information is not necessary to get the completeness of the calculus, thus there are different ways to handle it, both in the logical calculus and in the implementation. It has to be noticed that if the side information is not treated properly, then there can be disadvantages. For example, in the implementation we can decide not to apply to the side information rules having two conclusions but only the rules having one, so useless branching is not introduced into the proof (thus, if B_j is of the kind $C\vee D$, then the rule $\mathbf{T}\vee$ is not applied; if B_j is of the kind $C\wedge D$, then the rule $\mathbf{T}\wedge$ is applied to deduce the two stable formulas $\mathbf{T}C$ and $\mathbf{T}D$). With respect to the efficiency, there is another remark that deserves attention and we have considered in the implementation but not in the presentation of the calculus. Let us consider the case that starting from the j-th conclusion of $\mathbf{F}\rightarrow$, $j>1$, in a subsequent step the formula $\mathbf{T}A_i$ ($i\in\{1,\ldots,n\}$ and $i\neq j$) is inserted. By Simplification the formula $\mathbf{F}((A_i\wedge B_j)\rightarrow B_i)$ becomes $\mathbf{F}(B_j\rightarrow B_i)$ and a useless branch arises. The branch is useless because to get the completeness only $\mathbf{F}B_i$ is necessary (note that this is the formula we get if we apply the rule **Dum**). In the presentation of the logical calculus we have decided not to be concerned about these aspects related to the efficiency. We have faced them at the development stage. To avoid the disadvantages quoted above the new connective % is introduced. The aim of % is to identify conjunctive formulas whose right operand is a side formula. Since the right operand of % is the side information, the truth value of % depends on its left operand. The rules treating formulas of the

[1]EPDL is downloadable from `http://www.dimequant.unimib.it/~guidofiorino/epdl.jsp`.

kind $A\%B$ behave as the rules for conjunctive formulas, thus $\dfrac{\mathbf{T}(A\%B)}{\mathbf{T}A, \mathbf{T}B}\mathbf{T}\%$ is an additional rule of the implementation. Beside the rule $\mathbf{T}\%$ there are the Simplification rules related to $\%$, all behaving as the Simplification rules for $\wedge$ except for the case $\top\%B$ handled by the rule

$$\frac{S}{S[\top\%B/\top]}\text{Simp}\,\top\%.$$

Rules for the formulas of the kind $\mathscr{S}(\mathscr{A}\%\mathscr{B})$, with $\mathscr{S} \in \{\mathbf{F}, \mathbf{F_c}, \mathbf{T_{cl}}\}$, do not need to be implemented. Indeed, $A\%B$ always occurs in the antecedent of an $\mathbf{F}\rightarrow$-formula and this implies that in the proof table the formula $A\%B$ occurs with sign $\mathbf{T}$.

Remark 4. After the rule $\mathbf{T}\%$ is applied, EPDL treats the side information as a standard formula. The subsequent rules applied to side information can give rise to useless branches. To avoid the generation of branches, another implementation is possible along the following ideas: (i) introduce a new sign $\tilde{\mathbf{T}}$ to mark the side information; (ii) treat $\mathbf{T}(A\%B)$ by the rule $\dfrac{\mathbf{T}(A\%B)}{\mathbf{T}A, \tilde{\mathbf{T}}B}\mathbf{T}\%\text{–}new$; (iii) treat the $\tilde{\mathbf{T}}$-formulas by the rule $\dfrac{\tilde{\mathbf{T}}(A \wedge B)}{\tilde{\mathbf{T}}A, \tilde{\mathbf{T}}B}\mathbf{T}\%$. We recall the side information is not necessary to preserve the completeness. Thus the $\tilde{\mathbf{T}}$-rules for the remaining connectives are not needed; (v) introduce the rule $\dfrac{S, \tilde{\mathbf{T}}A}{S[A/\top], \tilde{\mathbf{T}}A}\text{Replace}\,\tilde{\mathbf{T}}$ to exploit the stable information conveyed by the side information A.

Stable formulas convey information related to the preservation of the forcing relation. In order to exploit Simplification as much as possible, our strategy is to treat as soon as possible all the stable information. In particular, the choice of the rule $\mathbf{F}\wedge$ is delayed until no other rule in Figure 1 is applicable.

In Figure 6 we compare LC-models [2] based on [21] with our implementation EPDL[3]. The formulas considered come from two sources. The first three families are formulas characterizing intermediate logics (see [10, 14])[4]. Nish stands for Nishimura formulas, defined as follows: $\text{Nish}_1 = p$; $\text{Nish}_2 = \neg p$; $\text{Nish}_3 = \neg\neg p$; $\text{Nish}_4 = \neg\neg p \rightarrow p$; $\text{Nish}_k = \text{Nish}_{k-1} \rightarrow (\text{Nish}_{k-3} \vee \text{Nish}_{k-4})$, $(k \geq 5)$. GdeJ refers to Gabbay de-Jongh formulas, semantically characterized by k-ary trees Kripke models and whose definition is the following: $\text{GdeJ}_k = \bigwedge_{i=0}^{k}((p_i \rightarrow \bigvee_{i\neq j} p_j) \rightarrow \bigvee_{i\neq j} p_j) \rightarrow \bigvee_{i=0}^{k} p_i$ $(k \geq 1)$. Finally, the shortening Fin stands for the following sequence of formulas: $\text{Fin}_1 = \neg p_1 \vee \neg\neg p_1$, $\text{Fin}_2 = \neg p_1 \vee (\neg p_1 \rightarrow \neg p_2) \vee (\neg p_1 \rightarrow \neg\neg p_2)$, $\text{Fin}_k = \neg p_1 \vee (\neg p_1 \rightarrow \neg p_2) \vee (\neg p_1 \wedge \neg p_2 \rightarrow \neg p_3) \vee \cdots \vee (\neg p_1 \wedge \cdots \wedge \neg p_{k-1} \rightarrow \neg\neg p_k)$ $(k \geq 3)$, which is valid on Kripke models whose poset has at most k maximal elements.
The other families are formulas of ILTP library [23] (in the tables the names have been shortened to save space and brackets around benchmark names denote unprovable formulas in **Dum**). Considering the timings, EPDL is a clear winner in all the families. It is interesting to analyze the growing ratio within each family. The growing ratio of EPDL is higher than LC-models on the families 201 and 207, lower on the families Nish, GdeJ, 205, 206, 208 and 212, and equal in the remaining families. As a further experiment the two provers have been run on two sets containing 40000 randomly generated formulas. The formulas in the first set were built on 23 connectives and 3 variables, the formulas in the second

[2] LC-models is downloadable from `http://www.loria.fr/~larchey/LC`.

[3] Experiments performed on Intel(R) Xeon(TM) CPU 3.00GHz, RAM 2GB. Timings expressed in seconds. On 206.5 the computation was stopped after some hours because memory occupation was more than 90%. Names of unprovable formulas are in parenthesis.

[4] Other formulas characterizing intermediate logics have been considered in the experiments. Results are disregarded since both the provers decide them in few seconds.

Formula	LC-models	EPDL
Nish.9	9	0.01
Nish.10	49	0.01
Nish.11	192	0.01
Nish.12	895	0.01
GdeJ.9	3	0.07
GdeJ.10	6	0.11
GdeJ.11	11	0.15
GdeJ.12	21	0.19
Fin.97	224	5.02
Fin.98	230	5.16
Fin.99	238	5.32
Fin.100	245	5.61
201.2	43.16	0.21
201.3	362	3.14
201.4	2183	35.96
201.5	12186	378

Formula	LC-models	EPDL
202.2	0.06	0.01
202.3	0.41	0.02
202.4	4.16	0.15
202.5	43.79	1.18
204.17	14.10	0.01
204.18	17.80	0.01
204.19	22.12	0.01
204.20	27.06	0.01
205.3	61.61	0.03
205.4	214	0.08
205.5	762	0.24
205.6	2932	0.68
206.2	0.16	0.01
206.3	28.74	0.01
206.4	1683	0.01
206.5	N.A.	0.01
(207.3)	46	0.31
(207.4)	200	3.71
(207.5)	805	40.23
(207.6)	3280	406

Formula	LC-models	EPDL
(208.1)	0.02	0.01
(208.2)	0.52	0.01
(208.3)	21.46	0.02
(208.4)	442	0.06
(210.17)	16.98	0.01
(210.18)	21.20	0.02
(210.19)	26	0.02
(210.20)	31.48	0.02
(211.17)	381	0.20
(211.18)	458	0.24
(211.19)	522	0.28
(211.20)	589	0.32
(212.1)	0.03	0.01
(212.2)	2.00	0.01
(212.3)	133	0.01
(212.4)	1589	0.01

Figure 6: Time comparison between LC-models and EPDL.

Formula	Basic	+Fact	+Side	EPDL
201.2	0.36	0.22	0.39	0.21
201.3	8.25	2.94	7.45	3.14
201.4	191	39.91	116	35.96
201.5	4830	502	1629	378
203.7	1.94	0.01	0.09	0.00
203.8	17.17	0.01	0.24	0.00
203.9	171.63	0.01	0.59	0.01
203.10	1885	0.00	1.42	0.01

Formula	Basic	+Fact	+Side	EPDL
205.3	0.03	0.03	0.03	0.03
205.4	0.12	0.13	0.08	0.08
205.5	0.76	0.80	0.23	0.24
205.6	5.80	5.92	0.68	0.68
(207.3)	0.54	0.27	0.56	0.31
(207.4)	11.51	3.63	9.70	3.71
(207.5)	266	46.63	139	40.23
(207.6)	6692	573	1827	406

Figure 7: Time comparison between different versions of EPDL.

set consisted of 49 connectives and 5 variables[5]. To decide all the formulas in the first set EPDL took 18.75 seconds and LC-models took 18531. As regard the formulas of the second kind, EPDL took 66 whereas LC-models took 969079 seconds. These results show that EPDL is faster than LC-models and, more importantly, that EPDL scales better since its increasing factor between the two kinds of families formulas is lower than LC-models.

The Figure 7 provides an account both of the formulas on which the optimizations work and a comparison between them. From left to right, "Basic" refers to EPDL lacking of the factorization rules and of the side information in the conclusion of $\mathbf{F}\rightarrow$; "+Fact" stands for Basic extended with the factorization rules; "+Side" denotes to Basic extended with the side information in the conclusion. The comparison clearly evidences that the optimizations are effective both individually and together. In particular, on SYJ201 and SYJ207 families of the ILTP library both the optimizations contribute to improve the performances. As regard SYJ203, the side information allows to reduce the timing. Finally, the factorization of $\mathbf{F}\rightarrow$-formulas improves the performances of EPDL on SYJ205.

Despite EPDL is a decision procedure whose time complexity is exponential, figures emphasize that by adding to the logical rules some optimizations, the result is a decision procedure which is effective on a wide range of formulas and outperforms LC-models which implements a polynomial time decision procedure.

[5]The formula SYJ201+1.001 is built on 23 connectives and 3 variables, whereas SYJ201+1.002 contains 49 connectives and 5 variables.

References

[1] A. Avellone, M. Ferrari, and P. Miglioli. Duplication-free tableau calculi and related cut-free sequent calculi for the interpolable propositional intermediate logics. *Logic Journal of the IGPL*, 7(4):447–480, 1999.

[2] A. Avellone, G. Fiorino, and U. Moscato. Optimization techniques for propositional intuitionistic logic and their implementation. *Theoretical Computer Science*, 409(1):41–58, 2008.

[3] A. Avellone, P. Miglioli, U. Moscato, and M. Ornaghi. Generalized tableau systems for intermediate propositional logics. In D. Galmiche, editor, *Automated Reasoning with Analytic Tableaux and Related Methods: Tableaux '97*, volume 1227 of *LNAI*, pages 43–61. Springer-Verlag, 1997.

[4] A. Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Annals for Mathematics and Artificial Intelligence*, 4:225–248, 1991.

[5] A. Avron. A tableau system for Gödel-Dummett logic based on a hypersequent calculus. In Roy Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1847 of *Lecture Notes in Artificial Intelligence*, pages 98–111. Springer, 2000.

[6] A. Avron and B. Konikowska. Decomposition proof systems for Gödel-Dummett logics. *Studia Logica*, 69(2):197–219, 2001.

[7] M. Baaz. Infinite-valued Gödel logics with 0-1 projections and relativizations. In *Proceedings of Gödel '96 - Kurt Gödel's Legacy*, volume 6 of *Lecture Notes Logic*, pages 23–33, 1996.

[8] M. Baaz and C.G. Fermüller. Analytic calculi for projective logics. In Neil V. Murray, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '99*, volume 1617 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 1999.

[9] M. Baaz, A. Ciabattoni, and C. G. Fermüller. Hypersequent calculi for Gödel logics – a survey. *J. of Logic and Computation*, 13(6):835–861, 2003.

[10] A. Chagrov and M. Zakharyaschev. *Modal Logic*. Oxford University Press, 1997.

[11] R. Dyckhoff. A deterministic terminating sequent calculus for Gödel-Dummett logic. *Logic Journal of the IGPL*, 7(3):319–326, 1999.

[12] G. Fiorino. An $O(n \log n)$-SPACE decision procedure for the propositional Dummett Logic. *Journal of Automated Reasoning*, 27(3):297–311, 2001.

[13] G. Fiorino. Fast decision procedure for propositional dummett logic based on a multiple premise tableau calculus. Technical Report 164, Dipartimento di Metodi Quantitativi per le Scienze Economiche ed Aziendali, Università degli Studi di Milano-Bicocca, 2008.

[14] D.M. Gabbay. *Semantical Investigations in Heyting's Intuitionistic Logic*. Reidel, Dordrecht, 1981.

[15] K. Gödel. On the intuitionistic propositional calculus. In S. Feferman et al, editor, *Collected Works*, volume 1. Oxford University Press, 1986.

[16] R. Hähnle. Tableaux and related methods. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 100–178. Elsevier and MIT Press, 2001.

[17] P. Hajek. *Metamathematics of Fuzzy Logic*. Kluwer, 1998.

[18] J. Hudelmaier. An $O(n \log n)$-SPACE decision procedure for intuitionistic propositional logic. *Journal of Logic and Computation*, 3(1):63–75, 1993.

[19] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, New York, 1952.

[20] D. Larchey-Wendling. Combining proof-search and counter-model construction for deciding Gödel-Dummett logic. In Andrei Voronkov, editor, *CADE*, volume 2392 of *LNCS*, pages 94–110. Springer, 2002.

[21] D. Larchey-Wendling. Graph-based decision for Gödel-Dummett logics. *J. Autom. Reasoning*, 38(1-3):201–225, 2007.

[22] F. Massacci. Simplification: A general constraint propagation technique for propositional and modal tableaux. In Harrie de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, Oosterwijk, The Netherlands*, volume 1397 of *LNCS*, pages 217–232. Springer-Verlag, 1998.

[23] Thomas Raths, Jens Otten, and Christoph Kreitz. The iltp problem library for intuitionistic logic. *J. Autom. Reasoning*, 38(1-3):261–271, 2007.

A Comparison of Solvers for Propositional Dynamic Logic

Ullrich Hustadt
Department of Computer Science, University of Liverpool, Liverpool, UK
U.Hustadt@liverpool.ac.uk
Renate A. Schmidt
School of Computer Science, The University of Manchester, UK
Renate.Schmidt@manchester.ac.uk

Abstract

Calculi for propositional dynamic logics have been investigated since the introduction of this logic in the late seventies. Only in recent years have practical procedures been suggested and implemented. In this paper, we compare three such systems, namely, the Tableau Workbench by Abate, Goré, and Widmann (2009), the pdlProver system by Goré and Widmann (2009), and the MLSOLVER system by Friedmann and Lange (2009).

1 Introduction

Propositional dynamic logic (PDL) is an expressive logic for reasoning about programs and actions [7]. Initially intended for program verification, it has found applications in a wide range of areas including verification of rule-based expert systems, synthesis of composite web services, and the formalisation of multi-agent systems.

In recent years there has been renewed interest in PDL and, in particular, in complexity optimal calculi and implementations of theorem provers for PDL [10, 13, 15]. The aim of this paper is to investigate the effectiveness of the current generation of PDL decision procedures. In particular, we are interested in evaluating two features recently introduced into such systems, namely, caching and on-the-fly eventuality checking. To this end we introduce two classes of benchmark formulae for PDL and test the performance of three implemented PDL decision procedures on them.

In Section 2 we give a brief definition of the syntax and semantics of PDL. In Section 3 we discuss the earliest decision procedures for PDL while in Section 4 we do the same for the most recent efforts to develop efficient calculi and implemented systems. In Section 5 we then describe two classes of benchmark formulae that we have used to compare these systems. Section 6 presents the results of benchmarking the Tableau Workbench, pdlProver, and MLSOLVER on these two classes.

2 Propositional dynamic logic

The language of PDL is defined over a countable set $\mathsf{AP} = \{p, q, \ldots\}$ of *propositional variables* and a countable set $\mathsf{AA} = \{a, b, c, \ldots\}$ of *atomic actions*. The connectives of PDL are the Boolean connectives $\neg$, $\wedge$, $\vee$, the dynamic logic connectives $\vee$, $;$ *, ?, and the modal operators $[\_]$ and $\langle\_\rangle$.

The set F of *formulae* and A of *action formulae* are the smallest sets such that (i) $\mathsf{AA} \subseteq \mathsf{A}$, $\mathsf{AP} \subseteq \mathsf{F}$, (ii) if φ and ψ are formulae in F and α and β are action formulae in A then $\varphi?$, α^*, $\alpha \cup \beta$, $\alpha;\beta$ are action formulae in A and $\neg\varphi$, $\varphi \wedge \psi$, $[\alpha]\varphi$, and $\langle\alpha\rangle\varphi$ are formulae in F. Additional connections including $\top$, $\bot$, $\vee$, and $\rightarrow$ are defined as usual.

The semantics of PDL is based on Kripke structures. A *frame* is a pair (W, R) where W is a non-empty set of *worlds* and R is a function that maps each atomic action a to a binary relation $R(a)$ over W. A *model* (W, R, I) consists of a frame (W, R) together with an *interpretation function* I that maps each

propositional variable p to a set $I(p)$ of worlds. The functions R and I can then be extended to arbitrary action formulae and formulae as follows:

$$I(\neg\varphi) = W - I(\varphi) \qquad I(\varphi \wedge \psi) = I(\varphi) \cap I(\psi)$$
$$I([\alpha]\varphi) = \{w \mid \forall v \in W.(w,v) \in R(\alpha) \rightarrow v \in I(\varphi)\}$$
$$I(\langle\alpha\rangle\varphi) = \{w \mid \exists v \in W.(w,v) \in R(\alpha) \wedge v \in I(\varphi)\}$$
$$R(\varphi?) = \{(w,w) \mid w \in I(\varphi)\} \qquad R(\alpha \cup \beta) = R(\alpha) \cup R(\beta)$$
$$R(\alpha;\beta) = \{(w,v) \mid \exists u \in W.(w,u) \in R(\alpha) \wedge (u,v) \in R(\beta)\}$$
$$R(\alpha^*) = \{(w,v) \mid \exists n \in \mathbb{N} \exists u_0,\ldots,u_n \in W.(u_0 = w \wedge u_n = v \wedge \forall 1 \leq i \leq n-1.(u_i,u_{i+1}) \in R(\alpha)\}$$

Given a model (W,R,I) and a formula φ, we say φ is *true at a world* $w \in W$ iff $w \in I(\varphi)$. A model (W,R,I) *satisfies* a formula φ iff $I(\varphi)$ is non-empty. In this case we also say that φ is *satisfiable in* (W,R,I). A formula φ is *satisfiable* iff there exists a model (W,R,I) satisfying φ.

As is described in more detail in the following two sections, given a formula φ, tableau-based decision procedures for PDL try to build a representation of a model satisfying φ. Such a representation can be viewed as a directed graph whose nodes represent worlds and whose edges represent, and are labelled with, atomic actions linking two worlds. The nodes of the graph are not just labelled with propositional variables, but are also labelled with PDL formulae. The intended meaning is that each of the formulae labelling a node n is true at the world represented by n. If two nodes n and n' are connected via a directed edge from n to n' labelled with an atomic action a, then we say that n' is a-reachable from n. Given the labelling of nodes and edges, we can extend this notion of reachability to arbitrary action formulae.

A particular problem in the construction of a model graph are so-called *eventualities*. Eventualities are formulae of the form $\langle\alpha^*\rangle\varphi$. Suppose a node n in the model graph is labelled with an eventuality $\langle\alpha^*\rangle\varphi$. In order for the graph to represent a model in which $\langle\alpha^*\rangle\varphi$ is true at the world represented by n, we need a node n' in the graph which is α-reachable from n and which is labelled with the formula φ. In the absence of such a node our model will not adhere to the truth conditions for $\langle\alpha^*\rangle\varphi$ as set out by the semantics of PDL. In such a situation, $\langle\alpha^*\rangle\varphi$ is also called an *unfulfilled eventuality*. Detecting unfulfilled eventualities as early as possible in the construction process is a key concern for PDL decision procedures.

3 Early PDL decision procedures

Decision procedures for the satisfiability problem for PDL were first presented by Fischer and Ladner [7] and Pratt [16]. The satisfiability problem for PDL is EXPTIME-complete and already the decision procedure by Pratt [16] was complexity optimal.

Pratt's procedure proceeds in stages. Given a formula, in the first stage a directed graph is constructed with each node being labelled with a set of (labelled) formulae. The construction ensures that there are no two nodes with the same labelling set and that the number of nodes is at most exponential in the size of the given formula. The graph represents a class of potential models of the given formula, but may contain nodes and subgraphs which cannot occur in a model, for example, nodes labelled with inconsistent sets of formulae or subgraphs with unfulfilled eventualities. In subsequent stages these are deleted from the graph. The given formula is satisfiable iff a non-empty graph remains after all necessary deletions have been performed. The construction stage of the procedure can be completed in exponential time in the size of the given formula, each deletion step requires polynomial time in the size of the graph obtained from the construction stage, and there can be at most as many deletion stages as there are nodes in the graph. Overall, this leads to an EXPTIME decision procedure.

Pratt's method has drawbacks which make it impractical for a lot of applications. Most importantly, the initial construction stage can lead to a structure of exponential size even if the satisfiability or unsatisfiability of the given formula only depends on a small subgraph of the whole structure. This means that by the time the procedure enters the second stage, and may detect the satisfiability or unsatisfiability relatively quickly, exponential effort has already been expended on the construction stage.

The tableau calculus for PDL and Converse PDL by De Giacomo and Massacci [6] aims to address this problem. It uses a more traditional approach in which a tableau tree is constructed and explored using depth-first, left-to-right search. Each branch of the tree represents a single candidate model. Propositionally inconsistent sets of formulae are recognised immediately while a check for unfulfilled eventualities is conducted as soon as the construction of a candidate model is completed. This approach leads to a NEXPTIME algorithm. De Giacomo and Massacci claim that storing the whole tableau tree, instead of just a branch, the re-use of tableau nodes across different branches of the tableau, and an "on-the-fly" propagation of information about unsatisfiable sets of formulae leads to an EXPTIME algorithm. In this approach a check for fulfilled and unfulfilled eventualities is still necessary. Important details of this check are however missing in [6].

4 Current PDL calculi and systems

An approach combining features of both Pratt's procedure and De Giacomo and Massacci's tableau calculus is the on-the-fly tableau-based decision procedure by Abate, Goré and Widmann [3]. The procedure constructs a tableau tree where nodes are not only labelled with sets of formulae but also with so-called histories and variables. Histories are used to prevent cyclic applications of the tableau rules. Variables pass information from child nodes to parent nodes, in particular, information about the satisfiability status of a node and information about unfulfilled eventualities. The rules of the calculus specify how the formula sets of child nodes are computed from the formula set of a parent node as well as how the values of variables of a parent node are computed from the values of the corresponding variables in its child nodes. Side conditions on the rules ensure that no infinite branches are constructed thus ensuring termination. Since branches can be at worst exponentially long, a tableau can be of double exponential size. Overall, this results in a 2EXPTIME algorithm. The Tableau Workbench (TWB) [1, 2] includes an implementation of a this algorithm.

In its tableau construction the procedure by Abate, Goré and Widmann is close to that of De Giacomo and Massacci. However, an important difference between the two is the way the check for unfulfilled eventualities is performed. In the tableau calculus of De Giacomo and Massacci, this check can be performed as soon as the construction of one branch of the tableau is completed. The check takes into account information from all the nodes in that branch. If no unfulfilled eventualities are found (and none of the nodes is labelled with an inconsistent set of formulae), then the candidate model associated with the branch is indeed a model for the given PDL formula. However, if the check identifies an unfulfilled eventuality, then the construction moves to an alternative branch of the tableau and another check for unfulfilled eventualities takes place as soon as its construction is completed. Since branches share nodes, this means that nodes will be considered again and again in consecutive checks. In contrast, the tableau calculus of Abate, Goré, and Widmann uses information passed from child nodes to parent nodes through variables in order to compute whether there are unfulfilled eventualities. The advantage is that the computation is only done once for each node. However, the disadvantage is that the computation can only take place when the information required for the computation is available for all child nodes. This also includes the case where the child nodes are generated by application of a β-rule, e.g., a rule performing a case distinction for a disjunctive formula. Consequently, the value of the variable used for the check for unfulfilled eventualities associated with the root node can potentially only be determined

once the whole tableau has been constructed. Thus, while the check for unfulfilled eventualities is not separated into a separate stage of the procedure, the overall behaviour is quite similar to that of Pratt's procedure.

The PDL decision procedure by Goré and Widmann [13] deviates from the classical tableau approach by constructing an and-or graph instead of a tree. Again nodes are labelled by sets of formulae plus additional attributes recording the satisfiability status of a node, information on which eventualities present in the set of formulae associated with the node have been expanded in the node, and which nodes might potentially be used to fulfil each of the eventualities. The construction process ensures that there are no two nodes with the same set of formulae and the same set of eventualities expanded in the node. That is, whenever the application of a tableau rule generates a set of formulae and set of expanded eventualities already present in the graph, the corresponding node is re-used, a technique also called *caching*. As there are at worst an exponential number of distinct sets of formulae and sets of eventualities generated by the tableau rules, the size of the and-or graph is at worst exponential. Just as in the tableau-based decision procedure by Abate, Goré and Widmann [3] the value of the attributes for the satisfiability status of a node and for the information which nodes might potentially be used to fulfil each of the eventualities are computed taking into account information on its successor nodes. The way in which this information is computed appears to differ in that an unsatisfiable status is propagated earlier, but there is no detailed description of the process in [13]. The overall result is an EXPTIME decision procedure. The pdlProver system [12] provides an implementation of that procedure.

LoTREC 2.0 [11, 17] is a generic tableau-based system for building models of formulae in modal and description logics. It includes a module for PDL, however, it cannot be used as a 'black-box' decision procedure like the other systems and is consequently not included in our comparison.

Finally, Friedmann and Lange [10] have proposed a platform for satisfiability checking for various modal fixpoint logics, including PDL. Given a formula their approach generates a parity game as a product of a tableau for the formula and a deterministic automaton recognising 'bad branches' in the tableau. The satisfiability of the formula is then determined by solving the parity game. A generator for these parity games and a solver for them are implemented in the MLSOLVER system [8] and the PGSOLVER [9] system, respectively.

5 Benchmark formulae for PDL

Benchmarking implemented systems for non-classical logics is not easy. The number of non-classical logics far outstrips the number of available implemented decision procedures. While each logic is usually reasonably well-motivated by potential applications, the lack of implemented systems usually means that there is no motivation to formalise a large number of problems in one of these logics. Commonly, all one can find is a small number of illustrative formalisations of problems. In the worst case, all one can find is an axiomatisation of the logic which allows one to use instances of the axioms to be used as test cases for an implemented decision procedure. Neither illustrative formalisations nor instances of axioms typically turn out to be particularly challenging and do not allow us to infer much about the properties of the implemented systems.

As an alternative to using real world problems, Balsiger, Heuerding, and Schwendimann [5] suggested the use of synthetic benchmarks consisting of sets of scalable formulae. The selection of suitable benchmarks was supposed to be guided by the following principles: (i) the benchmark sets should contain provable as well as non-provable formulae; (ii) the benchmark sets should vary in structure; (iii) some of the benchmark sets should be hard enough for future decision procedures; (iv) for each formula the satisfiability status should be known; (v) simple 'tricks' should not help to solve the formulae; (vi) a 'complete test' should be possible in reasonable time; and (vii) it should be possible to concisely sum-

marise the benchmarking results.

In particular, for each of the modal logics K, KT, and S4 they proposed nine sets of scalable satisfiable formulae and nine sets of scalable unsatisfiable formulae. These benchmark sets were used in a comparison of decision procedures for modal logics conducted in conjunction with the TABLEAUX conference in 1998 [4]. Based on the benchmark results obtained by the various systems at the time, it appears that the benchmark sets have shortcomings regarding the three most important of the seven principles, namely, (iii), (v), and (vii). In particular, it turned out that most of the 18 sets of benchmark formulae were easily solvable. The reason seemed to be that these benchmark formulae were amenable to techniques like Boolean constraint propagation, non-chronological backtracking or the use of proof methods not based on tableau calculi, e.g., translation methods and resolution methods. A few benchmark sets were hard for all the systems involved, for example, pigeon hole formulae disguised by adding occurrences of modal operators. Pigeon hole formulae are known to possess only exponential length refutation in most calculi and obtaining shorter proofs requires conceptually different methods, e.g., the use of cutting plane proof methods. Another problem is that while the results of performance tests for the eighteen classes can be easily summarised, there is no sufficiently fine-grained metric, which one could use to say that one system performs better than another. In general, given the number of benchmark sets the most likely situation is that a system performs slightly better on some and slightly worse on others. For example, in 1998 none of the systems participating in the comparison outperformed all others on all benchmarks sets.

A consequence of these problems is that these benchmark formulae do not provide a motivation for developers of modal theorem provers to further improve their systems. If the system is already reasonably well-developed, then it will solve most of the benchmark formulae easily. Those that remain hard seem to require other methods than the automata, tableau, or resolution methods that most modal theorem provers are based on.

In [14], we have proposed an alternative benchmarking approach, called *scientific benchmarking* or *hypothesis-driven benchmarking*. In this approach benchmark problems are chosen to verify a particular hypothesis concerning the decision procedures under consideration.

In the following, we want to test two hypotheses for the PDL solvers TWB, pdlProver and MLSOLVER. The first hypothesis concerns the type of formulae for which the re-use of nodes in a tableau construction is advantageous. This should be the case if the number of distinct nodes in a tableau is rather small, but without caching the tableau would still be rather large. The second hypothesis concerns the drawbacks of the two stage approaches or approaches which can only determine the satisfiability of a formula once a tableau has been fully explored.

To test these hypotheses, we re-use two classes of benchmark formulae originally introduced for propositional linear time temporal logic (PLTL) in [14], but reformulated for PDL. The first class, $\mathscr{C}^1_{\mathrm{PDL}}$, consists of formulae of the form

$$\begin{array}{c}[a^*]\langle a\rangle\top \wedge [a^*]([a]L^1_1 \vee \ldots \vee [a]L^1_k) \wedge \ldots \wedge [a^*]([a]L^\ell_1 \vee \ldots \vee [a]L^\ell_k)\\ \wedge [a^*](\neg p_1 \vee \langle a^*\rangle p_2) \wedge [a^*](\neg p_2 \vee \langle a^*\rangle p_3) \wedge \ldots \wedge [a^*](\neg p_n \vee \langle a^*\rangle p_1),\end{array}$$

while the second class, $\mathscr{C}^2_{\mathrm{PDL}}$, consists of formulae of the form

$$\begin{array}{l}[a^*]\langle a\rangle\top \wedge (r_1 \vee L^1_1 \vee \ldots \vee L^1_k) \wedge \ldots \wedge (r_1 \vee L^\ell_1 \vee \ldots \vee L^\ell_k) \wedge (\neg r_1 \vee q_1)\\ \wedge (\neg r_1 \vee \neg q_n) \wedge [a^*](\neg r_n \vee [a]r_1) \wedge [a^*](\neg r_{n-1} \vee [a]r_n) \wedge \ldots \wedge [a^*](\neg r_1 \vee [a]r_2)\\ \wedge [a^*](\neg r_n \vee [a]\neg q_n) \wedge \ldots \wedge [a^*](\neg r_1 \vee [a]\neg q_n) \wedge [a^*](\neg q_1 \vee \langle a^*\rangle s_2)\\ \wedge [a^*](\neg s_2 \vee q_2 \vee [a]q_n \vee \ldots \vee [a]q_3) \wedge \ldots \wedge [a^*](\neg q_{n-1} \vee \langle a^*\rangle s_n) \wedge [a^*](\neg s_n \vee q_n).\end{array}$$

For benchmarking purposes, the $L^i_1, \ldots, L^i_k$ are propositional literals generated by choosing k distinct variables randomly from a set $\{p_1, \ldots, p_n\}$ of n propositional variables and by determining the polarity

of each literal with probability p. The remainder of each formula only depends on the parameter n. To use these formulae for benchmarking purposes we fix the parameters k, n and p. Then, for each of the values of ℓ between 1 and $8n$ we have generated a test set of 100 formulae, which are tested for satisfiability using the various systems under consideration. Similar to random kSAT formulae, formulae in $\mathscr{C}^1_{\mathrm{PDL}}$ and $\mathscr{C}^2_{\mathrm{PDL}}$ are likely to be satisfiable if the number ℓ is small and likely to be unsatisfiable if ℓ is large.

Most of the observations made in [14] about the corresponding PLTL formulae carry over to their PDL counterparts. For example, if a formula in $\mathscr{C}^1_{\mathrm{PDL}}$ is satisfiable, then it is satisfiable in a model with just n worlds. If a formula in $\mathscr{C}^2_{\mathrm{PDL}}$ is satisfiable, then it is satisfiable in a model with just one world and r_1 has to be false at that world.

Given these model-theoretic insights about the formulae, their satisfiability is relatively easy to check, in particular, they are as easy to solve as propositional kSAT formulae over n propositional variables. But the classes are constructed in such a way that PDL decision procedures, which have to rely on proof-theoretic means, find them challenging.

In the case of $\mathscr{C}^1_{\mathrm{PDL}}$, each formula φ_1 in it imposes a uniform set of constraints on all worlds of a model which gives little guidance in the search for a satisfying model. Furthermore, if the propositional formula $(L^1_1 \vee \ldots \vee L^1_k) \wedge \ldots \wedge (L^\ell_1 \vee \ldots \vee L^\ell_k)$ is satisfiable, then potentially every sequence of satisfying truth assignments for this formula could be a model $\mathscr{M}_1$ of φ_1. Only when we check whether all eventualities $\langle a^* \rangle p_i$ are satisfied within $\mathscr{M}_1$ will we know that our search for a model has been successful. We thus expect that naive tableau-based systems and systems, which like Pratt's method only perform an eventuality check after some exhaustive search for candidate models, will perform poorly. On the other hand, decision procedures which use caching should be able to take advantage of the small number of distinct truth assignments that exist for $p_1, \ldots, p_n$.

The class $\mathscr{C}^2_{\mathrm{PDL}}$ is meant to illustrate how quickly a tableau-based system can find a model for a formula provided it makes the right choices for disjunctive formulae and how efficiently it can recover from making the wrong choices. Decision procedures which use a two stage approach or which can only determine the satisfiability of a formula once a tableau has been fully explored will always consider the part of the tableau on which the propositional variable r_1 is true. However, constructing this part of the tableau is computationally costly and fruitless as no model can be constructed in which r_1 is true. In contrast, a decision procedure which can test candidate models one by one, and happens to first consider models in which r_1 is false, will quickly find a model for satisfiable formulae in this class. For unsatisfiable formulae we do not expect to see a significant difference between the two types of decision procedures as both would need to consider the two cases of r_1 being true and r_1 being false.

The class can also be used to illustrate problems with transferring variable selection heuristics used in SAT solvers to more complex logics. Commonly used heuristics select the variable with the highest number of occurrences first. In $\mathscr{C}^2_{\mathrm{PDL}}$ this is the variable r_1. If in addition, the first truth assignment used is the one which maximises the number of clauses that are satisfied, then r_1 will be made true first. The fallacy here is to focus solely on a Boolean abstraction of a modal formula. This ignores that in modal logics not all indecomposable subformulae are 'equal'.

6 Benchmarking results

We conducted the benchmarks with the Tableau Workbench, pdlProver and MLSOLVER on the two classes $\mathscr{C}^1_{\mathrm{PDL}}$ and $\mathscr{C}^2_{\mathrm{PDL}}$. The benchmarks were performed on PCs with Intel Core 2 Duo E6400 CPU @ 2.13GHz with 3GB main memory using Fedora 11. For each individual satisfiability test a time-limit of 1000 CPU seconds was used.

In all experiments, for both classes, the parameters k, n and p were fixed to 3, 5, and 0.5, respectively.

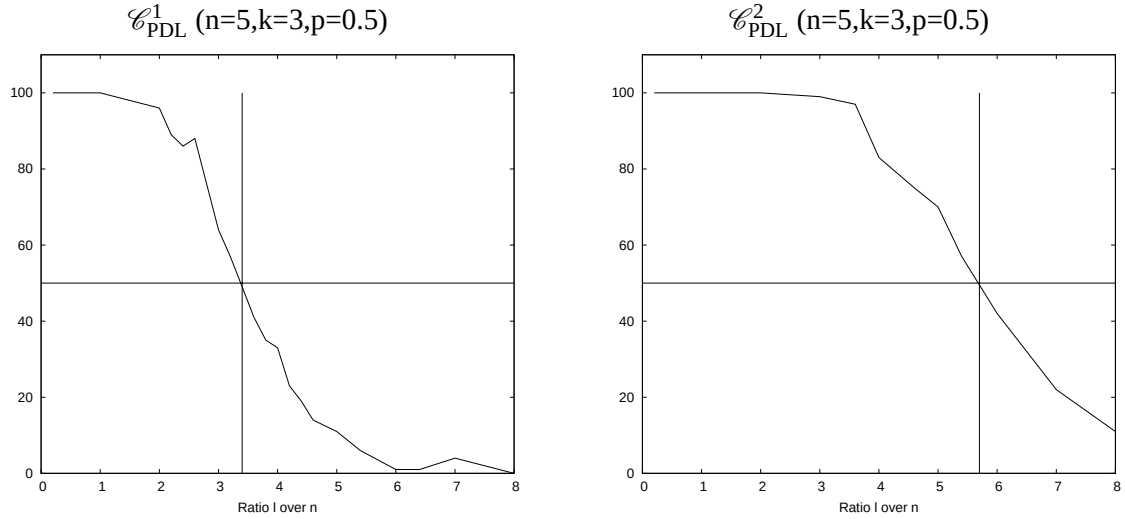


Figure 1: Satisfiability of formulae in $\mathscr{C}^1_{\mathrm{PDL}}$ and $\mathscr{C}^2_{\mathrm{PDL}}$

Remember that the satisfiability problem of propositional 2SAT formula is solvable in polynomial time. So, for $k = 2$, the satisfiability problem of $\mathscr{C}^1_{\mathrm{PDL}}$ and $\mathscr{C}^2_{\mathrm{PDL}}$ is also solvable in polynomial time and $k = 3$ is the minimal value for k that ensures that the satisfiability problem of $\mathscr{C}^1_{\mathrm{PDL}}$ and $\mathscr{C}^2_{\mathrm{PDL}}$ is NP-complete. The particular choice of p means that the randomly generated literals in our formulae have an equal probability of being positive or negative. Regarding the parameter n, the number of propositional variables we can use in our formulae, note that for $n = 3$ there is only one way of choosing $k = 3$ distinct propositional variables. For $n = 5$ there are ten different ways of choosing three distinct propositional variables, which in turn allows us to build a sufficiently large number of distinct formulae for our experiments.

Figure 1 shows the percentage of satisfiable formulae in $\mathscr{C}^1_{\mathrm{PDL}}$ and $\mathscr{C}^2_{\mathrm{PDL}}$ for these parameter values. For $\mathscr{C}^1_{\mathrm{PDL}}$ we see that for ratios ℓ/n smaller than 2 almost all formulae are satisfiable while for ratios ℓ/n greater than 5 almost all formulae are unsatisfiable. For ratios ℓ/n between 2 and 5 we see a phase transition in the satisfiability of formulae. For a ratio ℓ/n equal to 3.4 half the formulae are satisfiable. For $\mathscr{C}^2_{\mathrm{PDL}}$ we see that for ratios ℓ/n smaller than 3.5 almost all formulae are satisfiable and only for ℓ/n greater than 8.0 almost all formulae are unsatisfiable. Here, for the ratio ℓ/n equal to 5.7 half of the

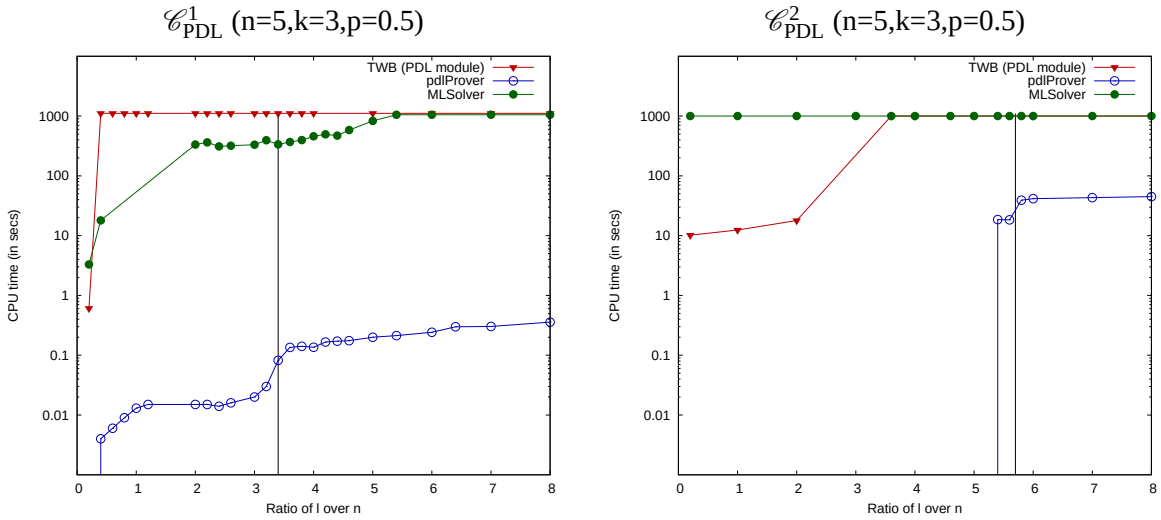


Figure 2: Performance of the decision procedures

formulae are satisfiable.

Figure 2 shows the median CPU time graphs for all three procedures on $\mathscr{C}^1_{\text{PDL}}$ and $\mathscr{C}^2_{\text{PDL}}$. In each graph the vertical line indicates the ratio ℓ/n at which test sets contain 50% satisfiable and 50% unsatisfiable formulae.

As can be seen in Figure 2, $\mathscr{C}^1_{\text{PDL}}$ separates pdlProver, the only system which uses caching, from the other two. As suggested, caching allows a prover to take advantage of the uniformity of the constraints imposed on the worlds of a model by formulae in $\mathscr{C}^1_{\text{PDL}}$. Thus, the good performance of pdlProver on this class was predictable. The absence of similar optimisations in the PDL module of the Tableau Workbench and in MLSOLVER are the most likely explanation for their poor performance. However, even then one might have expected both systems to be able to solve formulae in $\mathscr{C}^1_{\text{PDL}}$ with $\ell/n > 6$, which are almost all unsatisfiable and have a very constrained and limited search space for models.

For $\mathscr{C}^2_{\text{PDL}}$ the ideal system has negligible median runtime for $\ell/n < 5.7$, as up to this point the majority of formulae is satisfiable and a model for a satisfiable formula can easily be found. Only pdlProver could be 'guided' to behave in the expected way (by inputting formulae in the 'right' form, that is, exactly the form given on page 5; changing the order of conjuncts or the order of disjuncts within each conjunction seems to lead to worse results) and to make the right choices in the model construction up to $\ell/n \leq 5.4$ that is almost 'optimal'. In contrast, the Tableau Workbench and MLSOLVER fail to show a similar behaviour. For MLSOLVER we also observe a marked difference between $\mathscr{C}^1_{\text{PDL}}$ and $\mathscr{C}^2_{\text{PDL}}$. While for $\mathscr{C}^1_{\text{PDL}}$ MLSOLVER was able to solve the majority of formulae for each ratio ℓ/n, on $\mathscr{C}^2_{\text{PDL}}$ the opposite is true and it solved not a single formula in this class. On both classes the behaviour of the Tableau Workbench and MLSOLVER is as expected.

Figure 3 shows the CPU time percentile graphs for the three systems on $\mathscr{C}^1_{\text{PDL}}$ and $\mathscr{C}^2_{\text{PDL}}$. The graphs provide additional insight into their behaviour. The x-axis indicates the ratio ℓ/n as in previous figures. The y-axis indicates the percentile, from 10th percentile up to the 100th percentile. The 50th percentile corresponds to the median shown in Figure 2. The z-axis indicates the CPU time. In particular, for MLSOLVER and pdlProver the graphs confirm our expectations. As a two stage procedure, the performance of MLSOLVER does not greatly depend on whether a formula is satisfiable or unsatisfiable. Similarly, for pdlProver on $\mathscr{C}^1_{\text{PDL}}$, there is little variation in the performance of the system. However, caching allows pdlProver to perform much better than MLSOLVER. In contrast, on $\mathscr{C}^2_{\text{PDL}}$ the performance of pdlProver is closely related to whether a formula is satisfiable or not. We clearly see that in Figure 3 that as the percentage of unsatisfiable formulae increases so does the percentage of formulae for which pdlProver needs non-negligible time (more than 40 CPU seconds) to solve them.

Overall, pdlProver shows the best performance on these two classes of PDL formulae. The experiments illustrate the importance of caching and of detecting satisfiability as early as possible. In addition, the experiments show that the two classes of benchmark formulae originally devised for PLTL are also useful for 'black-box' performance evaluations of PDL solvers.

7 Conclusion

In this paper we presented benchmarking results for three implemented system for the satisfiability problem in propositional dynamic logic following the hypothesis-driven benchmarking methodology.

The benchmarks presented were intended to test two hypotheses for PDL solvers, namely, (i) that caching is important to control the search space of a system, and (ii) that the possibility of early detection of satisfiability is an essential feature of an efficient PDL solver. The benchmark results seem to support the validity of both hypotheses.

An additional aim of the hypothesis-driven benchmarking methodology is to highlight strengths and weaknesses of particular methods or systems and the benchmark results clearly do so as well.

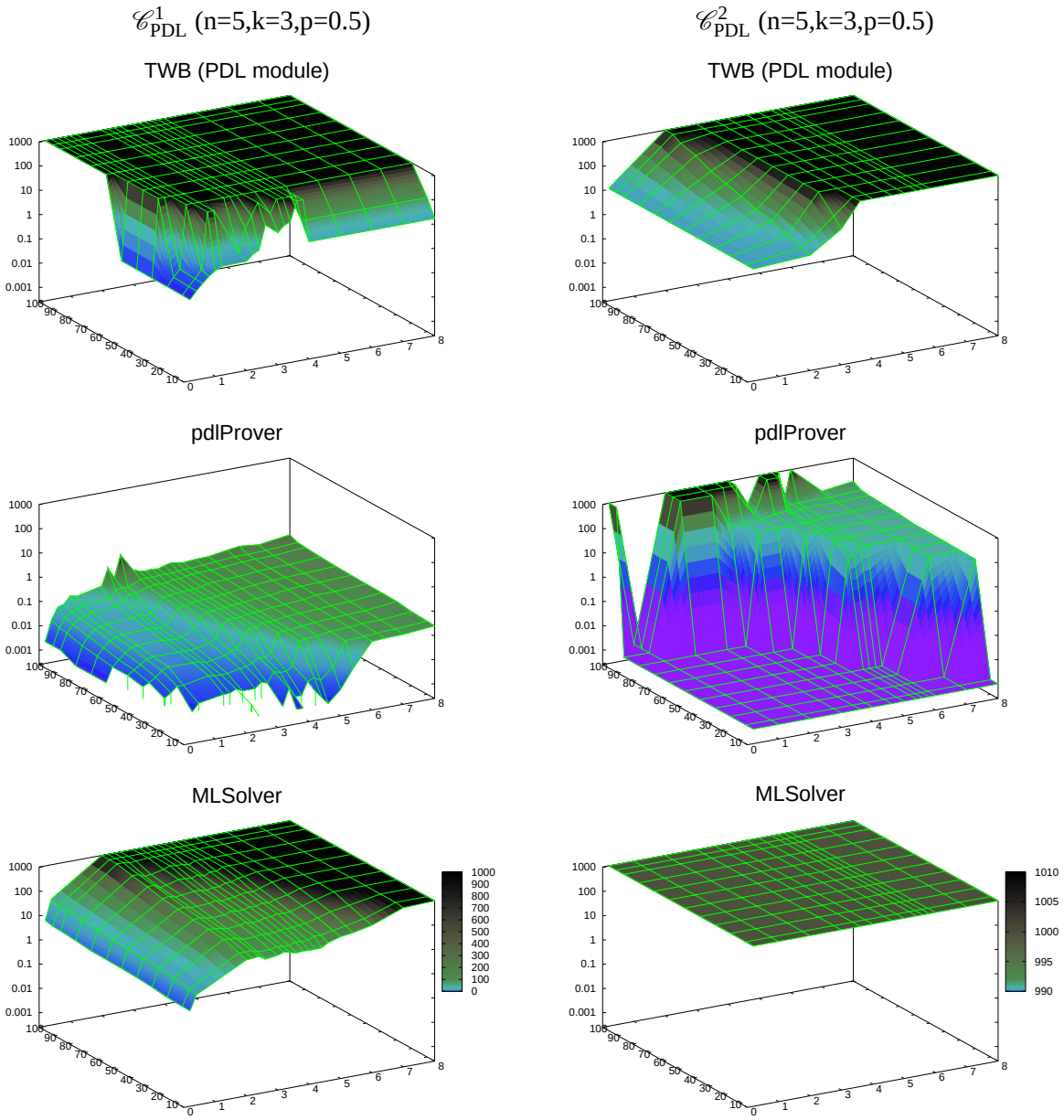


Figure 3: CPU time percentile graphs

Finally, the benchmarking approach is intended to motivate implementers to improve their systems. By using formulae for benchmarking whose satisfiability or unsatisfiability is far easier to detect than the worst-case complexity of the satisfiability problem for PDL suggests, there is little excuse for a system to perform badly on these.

References

[1] P. Abate and R. Goré. The Tableau Workbench (TWB). `http://twb.rsise.anu.edu.au/`.

[2] P. Abate and R. Goré. The Tableau Workbench. *Electron. Notes Theor. Comput. Sci.*, 231:55–67, 2009.

[3] P. Abate, R. Goré, and F. Widmann. An on-the-fly tableau-based decision procedure for PDL-satisfiability. *Electr. Notes Theor. Comput. Sci.*, 231:191–209, 2009.

[4] P. Balsiger and A. Heuerding. Comparison of theorem provers for modal logics: Introduction and summary. In Harrie de Swart, editor, *Automated reasoning with analytic tableaux and related methods: international conference (TABLEAUX '98)*, volume 1397 of *LNAI*, pages 25–26. Springer, 1998.

[5] P. Balsiger, A. Heuerding, and S. Schwendimann. A benchmark method for the propositional modal logics k, kt, s4. *J. Autom. Reasoning*, 24(3):297–317, 2000.

[6] G. De Giacomo and F. Massacci. Combining deduction and model checking into tableaux and algorithms for converse-PDL. *Info. and Comp.*, 162:117–137, 2000.

[7] M. J. Fischer and R. Ladner. Propositional dynamic logic of regular programs. *J. Comp. and System Sci.*, 18:194–211, 1979.

[8] O. Friedmann and M. Lange. MLSOLVER. `http://www2.tcs.ifi.lmu.de/mlsolver/`.

[9] O. Friedmann and M. Lange. PGSOLVER. `http://www2.tcs.ifi.lmu.de/pgsolver/`.

[10] O. Friedmann and M. Lange. A solver for modal fixpoint logics. In *Prelim. Proc. M4M-6*, pages 176–187. Roskilde University, Denmark, 2009.

[11] O. Gasquet, A. Herzig, D. Longin, and M. Sahade. LoTREC: Logical tableaux research engineering companion. In *Proc. TABLEAUX'05*, volume 3702 of *LNAI*, pages 318–322. Springer, 2005.

[12] R. Goré and F. Widmann. pdlProver. `http://users.cecs.anu.edu.au/~rpg/PDLProvers/`.

[13] R. Goré and F. Widmann. An optimal on-the-fly tableau-based decision procedure for PDL-satisfiability. In *Proc. CADE-22*, volume 5663 of *LNCS*, pages 437–452, 2009.

[14] U. Hustadt and R. A. Schmidt. Scientific benchmarking with temporal logic decision procedures. In *Proc. KR2002*, pages 533–544. Morgan Kaufmann, 2002.

[15] L. A. Nguyen and A. Szalas. Optimal tableau decision procedures for pdl. *CoRR*, abs/0904.0721, 2009.

[16] V. R. Pratt. A near-optiomal method for reasoning about actions. *J. Comp. and System Sci.*, 20:231–254, 1980.

[17] B. Said. LoTREC generic tableau prover. `http://www.irit.fr/Lotrec/`.

Trie Based Subsumption and Improving the *pi*-Trie Algorithm*

Andrew Matusiewicz
Institute of Informatics, Logics, and Security Studies
Department of Computer Science
University at Albany
Albany, NY 12222
email: a_matusiewicz@cs.albany.edu
Neil V. Murray
Institute of Informatics, Logics, and Security Studies
Department of Computer Science
University at Albany
Albany, NY 12222
email: nvm@cs.albany.edu
Erik Rosenthal
Institute of Informatics, Logics, and Security Studies
Department of Computer Science
University at Albany
Albany, NY 12222
erikr@cs.albany.edu

Abstract

An algorithm that stores the prime implicates of a propositional logical formula in a trie was developed in [10]. In this paper, an improved version of that *pi*-trie algorithm is presented. It achieves its speedup primarily by significantly decreasing subsumption testing. Preliminary experiments indicate the new algorithm to be substantially faster and the trie based subsumption tests to be considerably more efficient than the clause by clause approach originally employed.

1 Introduction

Prime implicants were introduced by Quine [13] as a means of simplifying propositional logical formulas in disjunctive normal form (DNF); prime implicates play the dual role in conjunctive normal form (CNF). Implicants and implicates have many applications, including abduction and program synthesis of safety-critical software [7]. All prime implicate algorithms of which the authors are aware make extensive use of clause set subsumption; improvements in both the *pi*-trie algorithm and its core subsumption operations are therefore relevant to all such applications.

Numerous algorithms have been developed to compute prime implicates — see, for example, [1, 2, 3, 5, 6, 8, 9, 12, 14, 18, 19]. Most use clause sets or truth tables as input, but rather few allow arbitrary formulas, such as the *pi*-trie algorithm introduced in [10]. This recursive algorithm stores the prime implicates in a trie — i.e., a labeled tree — and has a number of interesting properties, including the property that, at every stage of the recursion, once the subtrie rooted at a node is built, some superset of each branch in the subtrie is a prime implicate of the original formula. This property along with the way the recursion assembles branches admits variations of the algorithm that compute only restricted sets of prime implicates, such as all positive or not containing specific variables. These variations significantly prune the search space during the computation, and experiments indicate that significant speedups are obtained. In this paper, the algorithm is enhanced while these properties are retained.

*This research was supported in part by the National Science Foundation under grants IIS-0712849 and IIS-0712752.

The primary improvement developed here is the elimination of unnecessary subsumption tests. This is accomplished by performing subsumption checks between tries whose branches represent clause sets. Experiments indicate the trie-based subsumption tests to be far superior to the clause by clause approach originally employed. This in turn yields a substantially faster *pi*-trie algorithm.

Basic terminology and the fundamentals of *pi*-tries are summarized in Section 2. The analysis that leads to the new *pi*-trie algorithm is developed in Section 3, and the trie-based set operations and experiments with them are described in Section 4. Finally, the new *pi*-trie algorithm and the results of experiments that compare the new algorithm with the original are presented in Section 5.

2 Preliminaries

The terminology used in this paper for logical formulas is standard: An *atom* is a propositional variable, a *literal* is an atom or the negation of an atom, and a *clause* is a disjunction of literals.[1] Clauses are often referred to as sets of literals. An *implicate* of a formula is a clause entailed by the formula, and a non-tautological clause is a *prime implicate* if no proper subset is an implicate. The set of prime implicates of a formula $\mathscr{F}$ is denoted $\mathscr{P}(\mathscr{F})$. Note that a tautology has no prime implicates, while, since a contradiction implies any clause, the empty clause is the only prime implicate of a contradiction.

2.1 Background

The trie is a well-known data structure introduced by Fredkin in 1960 [4]; a variation was introduced by Morrison in 1968 [11]. It is a tree in which each branch represents the sequence of symbols labeling the nodes[2] on that branch, in descending order. Tries have been used in a variety of settings, including representation of logical formulas — see, for example, [15]. The nodes along each branch represent the literals of a clause, and the conjunction of all such clauses is a CNF equivalent of the formula. If there is no possibility of confusion, the term *branch* will often be used for the clause it represents. Further, it will be assumed that a variable ordering has been selected, and that nodes along each branch are labeled consistently with that ordering. A trie that stores all prime implicates of a formula is called a *prime implicate trie*, or simply a *pi*-trie.

It is convenient to employ a ternary representation of *pi*-tries, with the root labeled 0 and the *i*th variable appearing only at the *i*th level. If $v_1, v_2, \ldots, v_n$ are the variables, then the children of a node at level i are labeled v_{i+1}, $\neg v_{i+1}$, and 0, left to right. With this convention, any subtrie (including the entire trie) is easily expressed as a four-tuple consisting of its root and the three subtries. For example, the trie $\mathscr{T}$ can be written $\langle r, \mathscr{T}^+, \mathscr{T}^-, \mathscr{T}^0 \rangle$, where r is the label of the root of $\mathscr{T}$, and $\mathscr{T}^+$, $\mathscr{T}^-$, and $\mathscr{T}^0$ are the three (possibly empty) subtries. The ternary representation will generally be assumed in this paper.

The reader is assumed to be familiar with resolution and subsumption [16]; the observations and Lemma 1 are well known or obvious and are stated without proof.

Observations.

1. Each implicate of a logical formula is subsumed by at least one prime implicate.
2. $\mathscr{P}(\mathscr{F})$ is subsumption free.

[1] The term *clause* is also used for a conjunction of literals, especially with *disjunctive normal form*.

[2] Many variations have been proposed in which arcs rather than nodes are labeled, and the labels are sometimes strings rather than single symbols.

3. Resolution is consequence complete (modulo subsumption) for propositional CNF formulas. Thus, if C is an implicate of $\mathscr{F}$, there is a clause D that subsumes C and can be derived from $\mathscr{F}$ by resolution. In particular, every prime implicate of $\mathscr{F}$ can be derived by resolution.

4. A formula is equivalent to the conjunction of its prime implicates.

Let *resolution-subsumption* be the operation on clause sets defined by a single resolution step followed by removal of all subsumed clauses. Define a clause set $\mathscr{S}$ to be *prime* if $\mathscr{S} = \mathscr{P}(\mathscr{F})$ for some logical formula $\mathscr{F}$; equivalently, $\mathscr{S} = \mathscr{P}(\mathscr{S})$.

Lemma 1. A clause set $\mathscr{S}$ is a fixed point of resolution-subsumption iff $\mathscr{S}$ is prime. □

3 Prime Implicates under Truth-Functional Substitution

The *pi*-trie algorithm performs the recursion by substituting truth constants for variables to reduce the number of variables; this section contains an analysis of the relationship among $\mathscr{P}(\mathscr{F})$, $\mathscr{P}(\mathscr{F}[\alpha/v])$, and $(\mathscr{P}(\mathscr{F}))[\alpha/v]$, where $\alpha = 0, 1$, and v is a variable occurring in $\mathscr{P}(\mathscr{F})$. Partition $\mathscr{P}(\mathscr{F})$ into clause sets $\mathscr{S}^-$, $\mathscr{S}^+$, and $\mathscr{R}$, where $\mathscr{S}^-$ has the clauses containing $\neg v$, $\mathscr{S}^+$ has the clauses containing v, and the clauses of $\mathscr{R}$ contain neither. Observe that $\mathscr{P}(\mathscr{F})[1/v] = (N[1/v] \cup P[1/v] \cup \mathscr{R}[1/v])$. Then $\mathscr{S}^+[1/v] = \emptyset$ and $\mathscr{R}[1/v] = \mathscr{R}$. Also, $\mathscr{Q} = N[1/v]$ can be obtained by removing all occurrences of $\neg v$ from the clauses of $\mathscr{S}^-$. Let $\tilde{\mathscr{R}}$ be the the clauses in $\mathscr{R}$ not subsumed by any clause in $\mathscr{Q}$. The next lemma uses this notation.

Lemma 2. If $\mathscr{F}$ is any logical formula, then $\mathscr{P}(\mathscr{F}[1/v]) = \mathscr{Q} \cup \tilde{\mathscr{R}}$.

Proof. Note first that, since $\mathscr{S}^+[1/v] = \emptyset$, $\mathscr{P}(\mathscr{F}[1/v])$ is logically equivalent to $\mathscr{Q} \cup \tilde{\mathscr{R}}$. Thus, by Lemma 1, it suffices to show the latter to be a fixed point under resolution-subsumption. Since $\tilde{\mathscr{R}} \subseteq \mathscr{R} \subseteq \mathscr{P}(\mathscr{F})$, no clause in $\tilde{\mathscr{R}}$ can subsume any other clause in $\tilde{\mathscr{R}}$. The same is true for $\mathscr{Q}$ since it is true for $\mathscr{S}^-$, and the clauses of $\mathscr{Q}$ are obtained from the clauses of $\mathscr{S}^-$ by removing $\neg v$.[3] To complete the proof, it suffices to show that resolution can produce only a subsumed clause. There are two cases to consider: resolving two clauses from $\tilde{\mathscr{R}}$ and resolving with at least one clause from $\mathscr{Q}$.

Case 1. Let C be the resolvent of two clauses in $\tilde{\mathscr{R}}$. Then C does not contain v and is subsumed by a clause $\tilde{C}$ in $\mathscr{P}(\mathscr{F})$. The clause $\tilde{C}$ is in $\mathscr{R}$ and thus is either in $\tilde{\mathscr{R}}$ or is subsumed by a clause in $\mathscr{Q}$.

Case 2. Let C be the resolvent of two clauses with at least one from $\mathscr{Q}$. Then $C \cup \{\neg v\}$ is the resolvent of the corresponding clauses from $\mathscr{P}(\mathscr{F})$, so there is a clause $\tilde{C} \in \mathscr{P}(\mathscr{F})$ that subsumes $C \cup \{\neg v\}$. If $\neg v \in \tilde{C}$, then $\tilde{C} \in N$, so $\tilde{C} - \{\neg v\}$ is in $\mathscr{Q}$ and subsumes C. Otherwise, $\tilde{C} \in \mathscr{R}$, and the analysis of Case 1 applies. □

There is an entirely similar result when 0 is substituted for v:

Corollary. Partition $\mathscr{P}(\mathscr{F})$ into three sets: $\mathscr{S}^-$, the clauses containing $\neg v$, $\mathscr{S}^+$, the clauses containing v, and $\mathscr{R}$, the clauses containing neither. Let $\mathscr{Q} = \mathscr{S}^+[0/v] = \{C - \{v\} \mid C \in \mathscr{S}^+\}$, and let $\tilde{\mathscr{R}}$ be the clauses in $\mathscr{R}$ not subsumed by any clause in $\mathscr{Q}$. Then $\mathscr{P}(\mathscr{F}[0/v]) = \mathscr{Q} \cup \tilde{\mathscr{R}}$.

For the remainder of the paper, when it is clear that truth-functional substitution is for variable v, $\mathscr{F}[0/v]$ and $\mathscr{F}[1/v]$ will be denoted by $\mathscr{F}_0$ and $\mathscr{F}_1$, respectively.

Lemma 2 and its corollary say that, with respect to variable v, $\mathscr{P}(\mathscr{F})$ can be transformed into $\mathscr{P}(\mathscr{F}_\alpha)$ in polynomial time; moreover it places a limitation on the required checks for subsumption.

[3] It is possible that removing $\neg v$ could create a subsumption relationship to clauses in $\mathscr{R}$, but such clauses are removed from $\mathscr{R}$ to form $\tilde{\mathscr{R}}$.

Specifically, one must only check whether clauses in $\mathscr{Q}$ subsume clauses in $\mathscr{P}(\mathscr{F})$ that contain neither v nor $\neg v$, which takes time proportional to the product of the clause set sizes. The goal, however, is to transform $\mathscr{P}(\mathscr{F}_0)$ and $\mathscr{P}(\mathscr{F}_1)$ into $\mathscr{P}(\mathscr{F})$.

To that end, note first that $\mathscr{F} \equiv (v \vee \mathscr{F}_0) \wedge (\neg v \vee \mathscr{F}_1)$. So $\mathscr{P}(\mathscr{F})$ is logically equivalent to $(v \vee \mathscr{P}(\mathscr{F}_0)) \wedge (\neg v \vee \mathscr{P}(\mathscr{F}_1))$. Denote these conjuncts by J_0 and J_1, respectively; they can be regarded as clause sets by distributing v ($\neg v$) over the clauses of $\mathscr{P}(\mathscr{F}_0)$ ($\mathscr{P}(\mathscr{F}_1)$). Observe that J_0 and J_1 are (separately) resolution-subsumption fixed points because by definition so are $\mathscr{P}(\mathscr{F}_0)$ and $\mathscr{P}(\mathscr{F}_1)$. Subsumption cannot hold between a clause in J_0 and one in J_1 because the former contains v and the latter, $\neg v$. So if $J_0 \cup J_1$ must be altered to produce a resolution-subsumption fixed point, the changes result (directly or indirectly) from resolutions having one parent from each. These can be restricted to resolving on v and $\neg v$ because any other produces a tautology. Note that each such resolvent is the union of a clause from $\mathscr{P}(\mathscr{F}_0)$ and one from $\mathscr{P}(\mathscr{F}_1)$.

It turns out to be sufficient to consider only resolvents formed by one such resolution. This is a consequence of the following theorem, which is a restated version of Theorem 1 from [10].

Theorem 1. Let $\mathscr{F}$ be a logical formula and let v be a variable in $\mathscr{F}$. Suppose E is a prime implicate of $\mathscr{F}$ not containing v. Then $E \subseteq (C \cup D)$, where $C \in \mathscr{P}(\mathscr{F}_0)$ and $D \in \mathscr{P}(\mathscr{F}_1)$. □

Theorem 1 and the discussion leading up to it suggest how $\mathscr{P}(\mathscr{F})$ can be computed from $\mathscr{P}(\mathscr{F}_0)$ and $\mathscr{P}(\mathscr{F}_1)$. It will be useful to denote $\mathscr{P}(\mathscr{F}_0)$ and $\mathscr{P}(\mathscr{F}_1)$ by $\mathscr{P}_0$ and $\mathscr{P}_1$, respectively, and to partition each into two subsets. Let $\mathscr{P}_0^{\supseteq}$ be those clauses in $\mathscr{P}_0$ that are subsumed by some clause in $\mathscr{P}_1$. Let $\mathscr{P}_0^{\bowtie}$ be the remaining clauses in $\mathscr{P}_0$. Similarly define $\mathscr{P}_1^{\supseteq}$, and $\mathscr{P}_1^{\bowtie}$.

Theorem 2. Let J_0, J_1, $\mathscr{P}_0$, $\mathscr{P}_1$, $\mathscr{P}_0^{\supseteq}$, $\mathscr{P}_0^{\bowtie}$, $\mathscr{P}_1^{\supseteq}$, and $\mathscr{P}_1^{\bowtie}$ be defined as above. Then

$$\mathscr{P}(\mathscr{F}) = (v \vee \mathscr{P}_0^{\bowtie}) \cup (\neg v \vee \mathscr{P}_1^{\bowtie}) \cup (\mathscr{P}_0^{\supseteq} \cup \mathscr{P}_1^{\supseteq}) \cup \mathscr{U}$$

where $\mathscr{U}$ is the maximal subsumption-free subset of $\{C \cup D \mid C \in \mathscr{P}_0^{\bowtie}, D \in \mathscr{P}_1^{\bowtie}\}$ in which no clause is subsumed by a clause in $\mathscr{P}_0^{\supseteq}$ or in $\mathscr{P}_1^{\supseteq}$.

Proof. By considering each type of resolvent of a clause in J_0 with one in J_1 with respect to its addition to $J_0 \cup J_1$, the composition of $\mathscr{P}(\mathscr{F})$ can be verified. So assume $\{v\} \cup C$ is resolved with $\{\neg v\} \cup D$ on variable v, where $C \in \mathscr{P}_0$ and $D \in \mathscr{P}_1$. Four types of resolutions are possible, characterized by the blocks in which C and D reside. In three cases, $C \in \mathscr{P}_0^{\supseteq}$ or $D \in \mathscr{P}_1^{\supseteq}$.

Suppose first that $C \in \mathscr{P}_0^{\supseteq}$. Then there is a clause $D' \in \mathscr{P}_1$ that subsumes C. So the resolvent of $\{v\} \cup C$ with $\{\neg v\} \cup D'$ is C. All other resolutions involving $\{v\} \cup C$ result in a superset of C; a subset of C cannot be produced because $\mathscr{P}_0$ and $\mathscr{P}_1$ are prime implicate sets. So C is in $\mathscr{P}(\mathscr{F})$ and $\{v\} \cup C$ is not. This accounts for clauses in $(v \vee \mathscr{P}_0^{\bowtie})$ and in $\mathscr{P}_0^{\supseteq}$.

Similarly, the resolvents of $\{\neg v\} \cup D$ account for clauses in $(\neg v \vee \mathscr{P}_1^{\bowtie})$ and in $\mathscr{P}_1^{\supseteq}$. To summarize, all clauses in $\mathscr{P}_0^{\supseteq}$ and in $\mathscr{P}_1^{\supseteq}$ are in $\mathscr{P}(\mathscr{F})$, and the corresponding clauses of $v \vee \mathscr{P}_0^{\supseteq}$ and $\neg v \vee \mathscr{P}_1^{\supseteq}$ are not.

Now suppose $C \in \mathscr{P}_0^{\bowtie}$ and $D \in \mathscr{P}_1^{\bowtie}$. The resolvent $C \cup D$ may subsume or be subsumed by others of this type. It can also be subsumed by, but cannot subsume,[4] a clause from $\mathscr{P}_0^{\supseteq}$ or from $\mathscr{P}_1^{\supseteq}$. By removing such subsumed clauses from all clauses of this type, the set $\mathscr{U}$ results. □

4 Operations on Clause Sets

In [10], a branch by branch analysis leads to the PIT routine of the *pi*-trie algorithm introduced there. Theorem 2 in this paper is fundamentally the same. Each leads naturally to a method with which $\mathscr{P}(\mathscr{F})$

[4] Easily shown by contradiction from the properties of $\mathscr{P}_0$ and $\mathscr{P}_1$.

can be constructed from $\mathscr{P}_0$ and $\mathscr{P}_1$. However, Theorem 2 provides a set oriented characterization based on resolution and subsumption. The resulting development is arguably more intuitive. More importantly, the set oriented view has led to the more efficient version of the algorithm reported here.

One improvement results from identifying $\mathscr{P}_0^{\supseteq}$ and $\mathscr{P}_1^{\supseteq}$ before considering any clauses as possible members of $\mathscr{U}$. This contrasts with the PIT routine of [10] in which branch by branch subsumption checks are based on prime marks. An unmarked branch can be combined with another to form a possible member of $\mathscr{U}$, only to be eventually discovered to represent a clause in (say) $\mathscr{P}_0^{\supseteq}$; unnecessary subsumption checks result. A second improvement also results that is surprisingly effective. By handling $\mathscr{P}_0^{\supseteq}$ and $\mathscr{P}_0^{\bowtie}$ as separate sets, prime marks are unnecessary. Since the trie representation is being used, prime marks reside at leaf nodes. Checking for their presence requires traversing the branch, and this is almost as expensive as the subsumption check itself. (Using clause lists would allow first, rather than last, literals of a clause to be marked. But then the space economy of the trie would be lost.)

It turns out that a third improvement appears to be the most significant. Clause set operations can be realized recursively on entire sets, represented as tries.[5] Experiments show that the trie-based operations outperform branch by branch operations, and that the advantage increases with the size of the trie.

We define the following operators on clause sets F and G.

$Subsumed(F,G) = \{C \in G \mid C \text{ is subsumed by some } C' \in F\}$

$Unions(F,G) = \{C \cup D \mid C \in F,\ D \in G,\ C \cup D \text{ is not tautological }\}$

These definitions are purely set-theoretic. The pseudocode below realizes the operations assuming that clause sets are represented as ternary tries. Recall that the trie $\mathscr{T}$ can be written $\langle r, \mathscr{T}^+, \mathscr{T}^-, \mathscr{T}^0 \rangle$, where r is the root label of $\mathscr{T}$, and $\mathscr{T}^+$, $\mathscr{T}^-$, and $\mathscr{T}^0$ are the three subtries. Tries with three empty children are called *leaves*.

Algorithm 1: *Subsumed*(T_1,T_2)

input : Two clausal tries T_1 and T_2
output: T, a trie containing all the clauses in T_2 subsumed by some clause in T_1
if $T_1 = null$ *or* $T_2 = null$ **then**
  $T \leftarrow null$;
else if *leaf*(T_1) **then**
  $T \leftarrow T_2$;
else
  $T \leftarrow$ new Leaf;
  $T^+ \leftarrow Subsumed(T_1^+, T_2^+) \cup Subsumed(T_1^0, T_2^+)$;
  $T^- \leftarrow Subsumed(T_1^-, T_2^-) \cup Subsumed(T_1^0, T_2^-)$;
  $T^0 \leftarrow Subsumed(T_1^0, T_2^0)$;
  if *leaf*(T) **then**
    $T \leftarrow null$;
return T;

For convenience and readability, ordinary set union ($\cup$) and subtraction ($-$) have been employed in the pseudocode. Union can be implemented recursively for the trie representation. But the resulting performance is improved only slightly over a straightforward iteration on clauses. Subtraction is also straightforward but is always employed with the result of a subsumption test. In practice, it is easiest to extract the subsumed branches as a side effect during the subsumption test. Experiments involving both *pi*-tries and subsumption testing in isolation are reported in Section 5.

[5] Tries have been employed for (even first order) subsumption [17], but on a clause to trie basis, rather than the trie to trie basis developed here.

Algorithm 2: *Unions*(T_1,T_2)

input : Two clausal tries T_1 and T_2
output: T, a trie of the pairwise unions of the clauses in T_1 and T_2
if $T_1 = null$ *or* $T_2 = null$ **then**
  $T \leftarrow null$;
else if *leaf*(T_1) **then**
  $T \leftarrow T_2$;
else if *leaf*(T_2) **then**
  $T \leftarrow T_1$;
else
  $T \leftarrow$ new Leaf;
  $T^+ \leftarrow Unions(T_1^+, T_2^+) \cup Unions(T_1^0, T_2^+) \cup Unions(T_1^+, T_2^0)$;
  $T^- \leftarrow Unions(T_1^-, T_2^-) \cup Unions(T_1^0, T_2^-) \cup Unions(T_1^-, T_2^0)$;
  $T^0 \leftarrow Unions(T_1^0, T_2^0)$;
  if *leaf*(T) **then**
    $T \leftarrow null$;
return T;

5 The *pi*-trie Algorithm with set-wise operations

Theorem 2 leads to an alternate, simpler characterization of the *pi*-trie algorithm. We can view the algorithm in the standard divide-and-conquer framework, where each problem $\mathscr{F}$ is divided into subproblems $\mathscr{F}_0, \mathscr{F}_1$ by substitution on the appropriate variable (see the pseudocode for prime). The base case of this is where substitution yields a constant, which gives us $\mathscr{P}(0) = \{\{\}\}$ or $\mathscr{P}(1) = \{\}$.

Algorithm 3: prime($\mathscr{F}$,V)

input : A boolean formula $\mathscr{F}$ and a list of its variables $V = \langle v_1, \ldots, v_k \rangle$
output: The clause set $\mathscr{P}(\mathscr{F})$ — the prime implicates of $\mathscr{F}$
if $\mathscr{F} = 1$ **then**
  return $\emptyset$; `// Tautologies have no prime implicates.`
else if $\mathscr{F} = 0$ **then**
  return $\{\{\}\}$; `//` $\mathscr{P}(0)$ `is the set of just the empty clause.`
else
  $\mathscr{F}_0 \leftarrow \mathscr{F}[0/v_1]$;
  $\mathscr{F}_1 \leftarrow \mathscr{F}[1/v_1]$;
  $V' \leftarrow \langle v_2, \ldots, v_k \rangle$;
  return PIT(prime($\mathscr{F}_0, V'$) , prime($\mathscr{F}_1, V'$) , v_1);

The rest of the algorithm consists of combining $\mathscr{P}_0$ and $\mathscr{P}_1$ to form $\mathscr{P}(\mathscr{F})$. This is done both here and in [10] by a routine called PIT. But here, it is based on the *Subsumed* and *Unions* operators. The *SubsumedStrict* operator produces only clauses with proper subsets as subsuming clauses. It is similar in principle to *Subsumed*, but requires additional flags for bookkeeping.

Figure 1 compares the *pi*-trie algorithm from [10] to the updated version using the recursive *Subsumed* and *Unions* operators.[6] The input for both algorithms is 15-variable 3-CNF with varying numbers

[6]It is surprisingly difficult to find publicly available prime implicate generators. Substantial email inquiries based on publications produced only the system of Zhuang, Pagnucco and Meyer [20] that implements belief revision using prime

Algorithm 4: PIT(F_0,F_1,v)

input : Clause sets $\mathscr{P}_0 = \mathscr{P}(\mathscr{F}_0)$ and $\mathscr{P}_1 = \mathscr{P}(\mathscr{F}_1)$, variable v
output: The clause set $F = \mathscr{P}(\mathscr{F})$
$\mathscr{P}_0^{\supseteq} \leftarrow Subsumed(\mathscr{P}_1, \mathscr{P}_0)$; // Initialize $\mathscr{P}_0^{\supseteq}$
$\mathscr{P}_1^{\supseteq} \leftarrow Subsumed(\mathscr{P}_0, \mathscr{P}_1)$; // Initialize $\mathscr{P}_0^{\supseteq}$
$\mathscr{P}_0^{\bowtie} \leftarrow \mathscr{P}_0 - \mathscr{P}_0^{\supseteq}$;
$\mathscr{P}_1^{\bowtie} \leftarrow \mathscr{P}_1 - \mathscr{P}_1^{\supseteq}$;
$\mathscr{U} \leftarrow Unions(\mathscr{P}_0^{\bowtie}, \mathscr{P}_1^{\bowtie})$;
$\mathscr{U} \leftarrow \mathscr{U} - SubsumedStrict(\mathscr{U}, \mathscr{U})$;
$\mathscr{U} \leftarrow \mathscr{U} - Subsumed(\mathscr{P}_0^{\supseteq}, \mathscr{U})$;
$\mathscr{U} \leftarrow \mathscr{U} - Subsumed(\mathscr{P}_1^{\supseteq}, \mathscr{U})$;
return $F = v \vee \mathscr{P}_0^{\bowtie} \cup \neg v \vee \mathscr{P}_1^{\bowtie} \cup \mathscr{U}$;

of clauses, and the runtimes are averaged over 20 trials. The great discrepancy between runtimes requires that they be presented in log scale; it is explained in part by Figure 2, which compares the runtime of *Subsumed* to Algorithm 5, a naïve subsumption algorithm. The performance of the two systems converges as the number of clauses increases. With more clauses, formulas are unsatisfiable with probability approaching 1. As a result, the base cases of the prime algorithm are encountered early, and subsumption in the PIT routine plays a less dominant role, diminishing the advantage of the new algorithm.

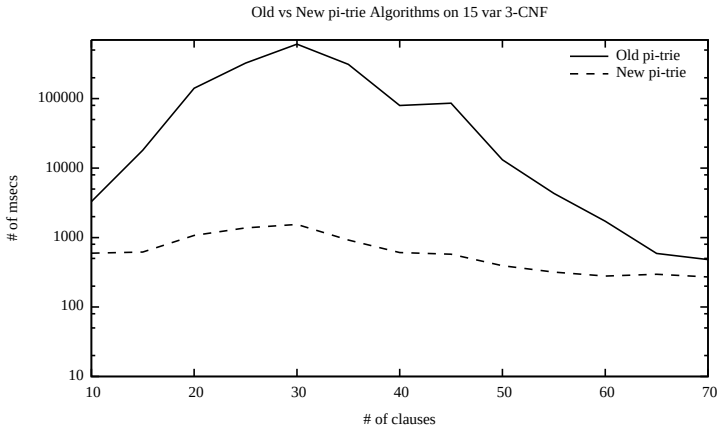


Figure 1: Old vs New *pi*-trie algorithm

implicates. That system was much less efficient than a simple prototype implemented by the first author, which in turn was much less efficient than the original *pi*-trie algorithm, available at http://www.cs.albany.edu/ritries. (A public release of the new algorithm is under development.)

Algorithm 5: NaiveSubsumed(F,G)

input : Clause sets F and G
output: The clauses in G subsumed by some clause in F
$H \leftarrow \emptyset$;
for $C \in F$ **do**
  for $D \in G$ **do**
    if $C \subseteq D$ **then** $H \leftarrow H \cup \{D\}$;
return H;

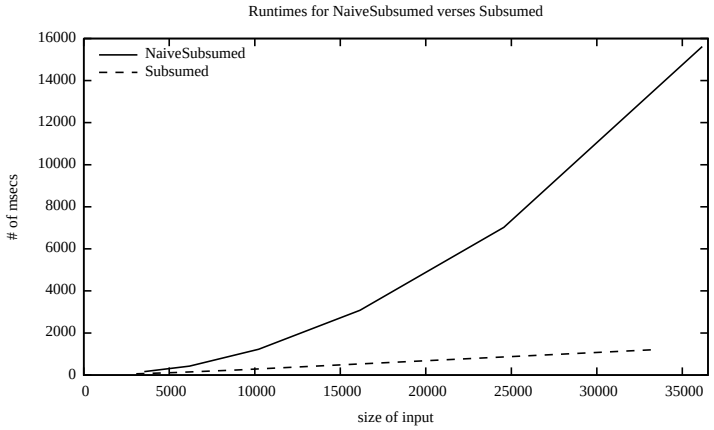


Figure 2: *Subsumed* vs *NaiveSubsumed*

The input for Figure 2 is a pair of n-variable CNF formulas where $n \in \{10,\ldots,15\}$ with results averaged over 20 trials for each n. Each formula with n variables has $\lfloor\binom{n}{3}/4\rfloor$ clauses of length 3, $\lfloor\binom{n}{4}/2\rfloor$ clauses of length 4, and $\binom{n}{5}$ clauses of length 5. This corresponds to $\frac{1}{32}$ of the $2^k\binom{n}{k}$ possible clauses of length k for $k = 3,4,5$.

The two clause sets are compiled into two tries for the application of *Subsumed* and into two lists for the application of *NaiveSubsumed*. For *Subsumed*, the runtimes for each n are graphed against the sum of the nodes in both input tries. *NaiveSubsumed* is graphed against the number of literal instances in both input formulas. This takes into accout the fact that in general tries are a more compact representation of a clause set than a list, so it is inaccurate to graph both runtimes against a single parameter.

It can be seen that the ratio of the runtimes changes as the input size increases – this suggests that the runtimes of *NaiveSubsumed* and *Subsumed* differ asymptotically. Additional evidence is supplied by the following lemma:

Lemma 3. *Subsumed*, when applied to two full ternary tries of depth h and combined size $n = 2(\frac{3^{h+1}-1}{2})$, runs in time $O(n^{\frac{\log 5}{\log 3}}) \approx O(n^{1.465})$.

Proof. At each level, *Subsumed* recurses on five pairs of children, giving the recurrence relation $Z(h) = 5Z(h-1)$ with $Z(0) = 1$ and thus $Z(h) = 5^h$ for the runtime of *Subsumed* with respect to height. Expressing height in terms of size, from $n = 2 \cdot \frac{3^{h+1}-1}{2} = 3^{h+1} - 1$ we get $h = \log_3 \frac{n+1}{3}$. This allows us to obtain $T(n) = Z(\log_3 \frac{n+1}{3})$ as an expression for runtime with regard to size. Thus we have

$$T(n) = 5^{\log_3 \frac{n+1}{3}} = 5^{\frac{\log_5 \frac{n+1}{3}}{\log_5 3}} = \frac{n+1}{3}^{\frac{1}{\log_5 3}}$$

Therefore $T(n) = O(n^{\frac{1}{\log_5 3}}) = O(n^{\frac{\log 5}{\log 3}}) \approx O(n^{1.465})$. □

This is less than *NaiveSubsumed*'s obvious runtime of $O(n^2)$ but still more than linear. Lemma 3 is interesting but the general upper bound may be quite different.

References

[1] Guilherme Bittencourt. Combining syntax and semantics through prime form representation. *Journal of Logic and Computation*, 18:13–33, 2008.

[2] O. Coudert and J. Madre. Implicit and incremental computation of primes and essential implicant primes of boolean functions. In *29th ACM/IEEE Design Automation Conference*, pages 36–39, 1992.

[3] J. de Kleer. An improved incremental algorithm for computing prime implicants. In *Proc. AAAI-92*, pages 780–785, San Jose, CA, 1992.

[4] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[5] Peter Jackson. Computing prime implicants incrementally. In *Proc. 11^{th} International Conference on Automated Deduction, Saratoga Springs, NY, June, 1992*, pages 253–267, 1992. In *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Vol. 607.

[6] Peter Jackson and J. Pais. Computing prime implicants. In *Proc. 10^{th} International Conference on Automated Deductions, Kaiserslautern, Germany, July, 1990*, volume 449, pages 543–557, 1990. In Lecture Notes in Artificial Intelligence, Springer-Verlag, Vol. 449.

[7] B.A. Jose, S.K. Shukla, H.D. Patel, and J.P. Talpin. On the deterministic multi-threaded software synthesis from polychronous specifications. In *Formal Models and Methods in Co-Design (MEMOCODE'08), Anaheim, California, June 2008*, 2008.

[8] A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9:185–206, 1990.

[9] V.M. Manquinho, P.F. Flores, J.P.M. Silva, and A.L. Oliveira. Prime implicant computation using satisfiability algorithms. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence, Newport Beach, U.S.A., November, 1997*, pages 232–239, 1997.

[10] A. Matusiewicz, N.V. Murray, and E. Rosenthal. Prime implicate tries. In *Proceedings of the International Conference TABLEAUX 2009 - Analytic Tableaux and Related Methods, Oslo, Norway, July 2009*, pages 250–264, 2009. In Lecture Notes in Artificial Intelligence, Springer-Verlag. Vol. 5607.

[11] D.R. Morrison. Patricia — practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[12] T. Ngair. A new algorithm for incremental prime implicate generation. In *Proc. IJCAI-93, Chambery, France, (1993)*, 1993.

[13] W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[14] Anavai Ramesh, George Becker, and Neil V. Murray. Cnf and dnf considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning*, 18(3):337–356, 1997.

[15] Ray Reiter and Johan de Kleer. Foundations of assumption-based truth maintenance systems: preliminary report. In *Proc. 6th National Conference on Artificial Intelligence, Seattle, WA, (July 12-17, 1987)*, pages 183–188, 1987.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[17] Stephan Schulz. Simple and efficient clause subsumption with feature vector indexing. In *Proceedings of the IJCAR 2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland, July 2004*.

[18] J. R. Slagle, C. L. Chang, and R. C. T. Lee. A new algorithm for generating prime implicants. *IEEE transactions on Computers*, C-19(4):304–310, 1970.

[19] T. Strzemecki. Polynomial-time algorithm for generation of prime implicants. *Journal of Complexity*, 8:37–63, 1992.

[20] Zhi Qiang Zhuang, Maurice Pagnucco, and Thomas Meyer. Implementing iterated belief change via prime implicates. In *Australian Conference on Artificial Intelligence*, pages 507–518, 2007.

Automation for Geometry in Isabelle/HOL

Laura I. Meikle
School of Informatics,
University of Edinburgh,
Edinburgh, U.K.
lauram@dai.ed.ac.uk

Jacques D. Fleuriot
School of Informatics,
University of Edinburgh,
Edinburgh, U.K.
jdf@inf.ed.ac.uk

Abstract

We present an implementation of a theory of two-dimensional geometry based on the signed area of triangles, together with a collection of automatic and interactive techniques we have developed to assist in our formal verification of geometric algorithms.

1 Introduction

Despite the successes of current state-of-the-art, fully automatic theorem provers, their results are far from replicating the expertise of human mathematicians. The search space for complex problems tends to explode very quickly, rendering them impractical for many verification tasks. As a result, much work in mechanised reasoning has focussed on the usability of systems for interactive theorem proving, where a human user guides the proof attempt. The end product of this process is a body of formalised mathematics which has the correctness guarantees that machines can provide but which is much more sophisticated than current systems could produce automatically.

Unfortunately, this approach requires a great deal of effort from the human user; not only do they require a good insight into the direction a proof should take, but they are often burdened with the tedium of proving enormous amounts of low level detail, which can detract from the main proof endeavour. For these reasons, there is a growing interest in semi-automated theorem proving, where a system incorporates automation techniques within an interactive theorem proving environment. These systems provide the best of both worlds: a user can observe an automated proof attempt, using its results when applicable, or instead manually guide the system when the automation flounders or when they have a good sense of how to proceed.

One such system is the generic theorem prover Isabelle [11], which has an extensive library of theories and some automatic proof methods which combine simplification and classical reasoning. Our interests have concentrated on using Isabelle for geometry theorem proving, and more recently for proving the correctness of geometric algorithms using the notion of signed areas. This interim paper describes parts of our Isabelle theory and the framework that we have developed for proving geometric algorithms, with a particular focus on the automation, systems integrations and related techniques that we have built in the course of this work.

2 A Theory of Planar Geometry Using Signed Area

The algorithms that we have been formally verifying are taken from the field of computational geometry. Specifically, we have been reasoning about convex hulls and Delaunay triangulations. Our formalisations have focussed on two-dimensional problems as these provide a clear appreciation of the difficulties that run at the core of many of the algorithms and we believe the lessons learnt from these proofs will scale easily when dealing with real-world problems that may involve higher dimensions. The algorithms share the common feature that they compute the positions of a set of points relative to each other. This led us to conclude that the concept of signed area would be a suitable representation.

We began by defining the type point as any pair of real numbers:

```
typedef point = "{p::(real*real). True}"
```

In Isabelle, this command produces three constants behind the scenes:

```
point :: real*real
Rep_point :: point ⇒ real*real
Abs_point :: real*real ⇒ point
```

Abs_point and Rep_point are the derived coercion functions that enable one to move from the newly defined point type to its underlying representation and back. Thus Rep_point, for instance, enables reasoning about points to be converted into reasoning about coordinates and hence polynomials.

As our verifications relied upon reasoning about the relative positions of points, we needed to formlise this notion. For this we used the *signed area* of a triangle; with the convention being that if the points are ordered anti-clockwise, the area is positive, and if the points are ordered clockwise, the area is negative. In our theory this was formalised as[1]:

```
constdefs signedArea :: "[point, point, point] ⇒ real"
      "signedArea a b c ≡ (xCoord b - xCoord a)*(yCoord c - yCoord a)
                        - (yCoord b - yCoord a)*(xCoord c - xCoord a)"
```

where the predicates xCoord and yCoord were formally represented by:

```
constdefs xCoord :: "point ⇒ real"
         "xCoord P ≡ fst(Rep_point P)"

constdefs yCoord :: "point ⇒ real"
         "yCoord P ≡ snd(Rep_point P)"
```

Using this definition it was then easy to formally represent the orientation of points; we say that three points a, b and c make a *left turn* if they make an anti-clockwise cycle:

```
constdefs leftTurn :: "[point, point, point] ⇒ bool"
      "leftTurn a b c ≡ 0 < signedArea a b c"
```

We deviate from many geometry theories by including so-called *degenerate* cases where the points may be collinear, or equivalently, the area of the triangle they define is zero:

```
constdefs collinear :: "[point, point, point] ⇒ bool"
         "collinear a b c ≡ signedArea a b c = 0"
```

A consequence of permitting collinearity is that an *ordering* on points along a line must be established. We achieved this by defining the concept of between. For collinear points a, b and c, we represent and define b lying between a and c as follows:

```
constdefs isBetween :: "[point, point, point] ⇒ bool"
         "b isBetween a c ≡ collinear a b c ∧ (∃d. signedArea a c d ≠ 0) ∧
                            (∀ d. signedArea a c d ≠ 0 ⟶
                                0 < signedArea a b d / signedArea a c d < 1 )"
```

[1] As the magnitude of the area was irrelevant in our proofs we simplified the usual definition by omitting the factor of $\frac{1}{2}$.

We have chosen this definition in preference to more succinct variations because the existence clause facilitates instantiating witnesses. Our theory also includes the obvious lemmas concerning these predicates, omitted here for brevity (although some relevant ones are introduced in in the subsequent section).

3 Extending Isabelle's Simplifier and Classical Reasoner

To a human mathematician the statement that three points are collinear is interpreted in only one way. However, to a computer the terms `collinear` *a b c* and `collinear` *b a c* are symbolically evaluated and interpreted differently. One of the most tedious parts of our earliest proofs was dealing with this very issue; in order for a lemma to be applied or a goal to be discharged it was often necessary to compare and adjust the ordering of points manually. Thus, for our theory of planar geometry, the first and most obvious automation was the establishment of rewrite rules to express the geometric terms in a canonical form.

Isabelle makes it easy for a user to encode this type of automation by enabling them to extend its simplifier and classical reasoner on demand. In the following subsections we present the rules which have been added to automate much of the geometric reasoning often required in verification tasks.

3.1 Simplification Rules

The standard simplification tactic—`simp`—is one of the single most powerful automation tools in Isabelle. Developers can annotate proved lemmas, *e.g.* with the token "`[simp]`", to indicate that the standard simplifier should attempt to use that lemma as a rule. This allows expressions to be reduced to canonical forms, in many cases, and even in some cases entire decision procedures can be encoded and goals proven automatically.

The following set of simplification rules allows our geometric predicates to be written in a canonical form automatically, removing much tedium (and in some cases proving goals automatically):

```
signedAreaRotate [simp]: "signedArea b c a = signedArea a b c"
signedAreaRotate2 [simp]: "signedArea b a c = signedArea a c b"
collRotate [simp]: "collinear c a b = collinear a b c"
collSwap [simp]: "collinear a c b = collinear a b c"
swapBetween [simp]: "a isBetween c b = a isBetween b c"
leftTurnRotate [simp]: "leftTurn b c a = leftTurn a b c"
leftTurnRotate2 [simp]: "leftTurn b a c = leftTurn a c b"
```

Each rule is expressed as an equality (with a proof, omitted here in the interest of space), with the convention that the simplifier replaces the left-hand side of the equality with the right-hand side of the equality whenever possible. The rules above are known as permutative rewrite rules as each side of the equation is the same up to renaming of variables. It is worth noting that such rules can be problematic because once they apply, they can create infinite loops. However, Isabelle's simplifier is aware of this danger and treats permutative rules by means of a special strategy, called ordered rewriting: a permutative rewrite rule is only applied if the term becomes smaller with respect to a fixed lexicographical ordering on terms. Recognising this special status automatically is a very useful feature of Isabelle.

In addition to the above rules, our theory adds several other proved rules to Isabelle's simplifier, not for the purpose of reducing to a canonical form, but in order to supply trivial facts automatically where needed:

```
areaDoublePoint [simp]: "signedArea a a b = 0"
```

```
areaDoublePoint2 [simp]: "signedArea a b b = 0"
twoPointsColl [simp]: "collinear a b b"
twoPointsColl2 [simp]: "collinear a a b"
notBetweenSelf [simp]: "¬ a isBetween a b"
notBetweenSelf2 [simp]: "¬ b isBetween a b"
notLeftTurn [simp]: "(¬ leftTurn a c b) =
                     (leftTurn a b c ∨ collinear a b c)"
```

Despite these simp rules discharging many of our subgoals automatically, there exists a large collection of trivial subgoals that require manual proof. One such example is showing collinearity when we know a betweenness relation holds, *i.e*

```
isBetweenImpliesCollinear : "a isBetween b c ⟹ collinear a b c"
isBetweenImpliesCollinear2 : "b isBetween a c ⟹ collinear a b c"
```

Of course, these facts are trivially proven by expanding the definition of between but it is cumbersome for the user to always perform this step. Applying a general tactic is preferential, so we first tried adding these rules to Isabelle's simp set, assuming they would work as conditional rewrites. However, these rules together can cause Isabelle to enter an endless loop. We have found it difficult to understand why this looping occurs in certain situations, even after inspecting the trace output. This emphasises that care must be taken when extending the simplifier. The Isabelle manual advises that users should include only canonical simplifications, *i.e.*, only rules which are universally desirable, and while this is sensible in practice, it means that much useful control knowledge cannot be expressed as simplification rules.

For our purposes though, another means of easily automating the "conditional rewrites" in Isabelle does exist. One can extend the "classical reasoner" rather than the simplifier. This approach shall be looked at next.

3.2 Conditional Rewrite Rules

In Isabelle, classical reasoning is different from simplification. While the latter is deterministic, classical reasoning uses search and backtracking in order to prove a goal outright using a Natural Deduction style of reasoning [11]. We can add rules to Isabelle's classical reasoner by marking them as introduction, elimination or destruction rules. This gives a powerful automation framework alongside the default simplifier. Regretfully the Isabelle tutorial is somewhat vague on their use–distinguishing between them as follows: "Introduction rules allow us to infer new information ... Elimination rules allow us to deduce consequences". To add to confusion Isabelle distinguishes between two types of elimination rules: if information is lost then the rule is called a destruction rule (although in the usual natural deduction sense it would just be called an elimination rule). We have found that the following rules are useful introduction rules for our problems:

```
notCollThenDiffPoints [intro]: "¬collinear a b c ⟹ a≠b ∧ a≠c ∧ b≠c"
isBetweenImpliesCollinear [intro]: "a isBetween b c ⟹ collinear a b c"
isBetweenImpliesCollinear2 [intro]: "b isBetween a c ⟹ collinear a b c"
isBetweenImpliesCollinear3 [intro]: "c isBetween a b ⟹ collinear a b c"
isBetweenPointsDistinct [intro]: "a isBetween b c ⟹ a≠b ∧ a≠c ∧ b≠c"
leftTurnDiffPoints [intro]: "leftTurn a b c ⟹ a≠b ∧ a≠c ∧ b≠c"
onePointIsBetween [intro]: "collinear a b c ⟹
                    a isBetween b c ∨ b isBetween a c ∨ c isBetween a b"
```

Another type of automation we wanted to implement was the identification of contradicting assumptions

and subsequent discharge of these subgoals. This is a common problem in geometry theorem proving as case splits are often needed to identify the positioning of points relative to each other. This method of proving will generally introduce some cases which cannot exist. It is up to the human user to identify these cases and discharge them. This is not an easy task when there is an enormous list of assumptions; manually discovering which assumptions contradict and then finding the correct lemma to apply is difficult, not to mention mundane. To automatically find certain contradictions, we added the following destruction rules to the classical reasoner:

```
areaContra [dest]: "[| signedArea a c b < 0; signedArea a b c < 0 |] ⟹ False"
areaContra2 [dest]: "[| 0 < signedArea a c b; 0 < signedArea a b c |] ⟹ False"
notBetweenSamePoint [dest]: "a isBetween b b ⟹ False"
notBetween [dest]: "[| A isBetween B C; B isBetween A C |] ⟹ False"
notBetween2 [dest]: "[| A isBetween B C; C isBetween A B |] ⟹ False"
notBetween3 [dest]: "[| B isBetween A C; C isBetween A B |] ⟹ False"
conflictingLeftTurns [dest]: "[| leftTurn a b c; leftTurn a c b |] ⟹ False"
conflictingLeftTurns2 [dest]: "[| leftTurn a b c; a isBetween b c |] ⟹ False"
conflictingLeftTurns3 [dest]: "[| leftTurn a b c; collinear a b c |] ⟹ False"
```

3.3 Limitations of Adding Automatic Rules

The simplification, introduction and elimination rules above remove a large amount of the low-level manipulation that was otherwise necessary when working in our theory. Despite this automation being easy to implement in Isabelle, there are two specific limitations we wish to point out.

The first of these is that the behaviour of the simplifier is rather opaque. Currently it provides a resulting proof state (or failure notification), and it can supply an extremely verbose trace of its activity. It does not provide a concise statement of which substitutions led to the resulting proof state. And—in the all-too-frequent situation where the simplification set includes potentially looping rules—it is very hard to determine which simplification rules are causing non-termination.

Secondly, if a user wants to add more sophisticated automation they have to create their own tactics. This is not an easy task, for it requires one to be familiar with the underlying ML code for Isabelle. This codebase is large and fairly complicated, and it is not nearly as well documented as the more user-friendly end-user mode “Isar”.

4 Integrating with QEPCAD

The simplification rules in the previous section can automatically handle some of the simplest problems we encountered in our verification, but they do not automatically solve any of what we consider our non-trivial lemmas. They assist by removing a great deal of the tedious manipulations—which is what they were intended for. For deeper problems, we have turned to some of the existing techniques for automatic geometry theorem proving (GTP).

Extensive research into mechanical GTP has been carried out over the past 40 years, and as a result it is a mature field. It has split into two main paradigms: the algebraic methods and the synthetic (or coordinate free) techniques. The latter have the advantage that they can often produce short, readable and intuitive proofs which are usually based on geometrically meaningful notions such as full-angles between lines [4] or signed areas of triangles [3]. This signed area method may seem like an ideal candidate for automatically discharging many of our proof obligations, and has recently been implemented in Coq [8]. Unfortunately the basic method does not apply to inequalities, making it inapplicable to our lemmas about betweenness, and the variants which do handle inequalities resort to algebraic methods

such as cylindrical algebraic decomposition. This has led us to adopt the powerful and complete algebraic approach of Hong and Collins, based on partial cylindrical algebraic decomposition [5]. This provides a decision procedure for quantifier elimination over real closed fields. Another benefit afforded by this approach is that many other formal reasoning tasks will benefit from this automation in Isabelle, for example the verification of control and hybrid systems.

Rather than implement this decision procedure ourselves, we have decided to integrate Isabelle with the external tool QEPCAD-B which implements cylindrical algebraic decomposition efficiently in C [2] As we believe other other external tools can also provide great assistance in the proof development process, we have sought to build a general framework that can simplify and automate the ways that existing best-of-breed tools can be accessed. Of course, where it is feasible in terms of development time and execution time, decision procedures embedded within Isabelle are clearly preferable because of the soundness guarantees of both the methods and the integration. However our focus has been primarily on integrating with external tools due to the wide variety and coverage of such tools. We note that in some cases these tools may generate information which can guide or generate pure-Isabelle proofs which do not rely on the external systems; our integration with QEPCAD can for example produce witnesses and counterexamples in some situtions In some instances though, we have been willing to use the results of these external tools without formal proof (i.e. the external tools act as oracles). We have done the latter for pragmatic reasons, with a clear indication of what external results are being used so that a reader can verify them to their own satisfaction (by hand or using their preferred tools), and with the anticipation that fully automated, formally correct methods will ultimately be available for proving these statements.

Our integration with QEPCAD has been done in a framework we have developed called the Prover's Palette (Figure 1). This framework is built upon the recently developed Eclipse Proof General [1], and takes advantage of its broker middleware. The integration and the framework are described in detail elsewhere [9]; however there is a large amount of automation in both the integration framework and the QEPCAD integration built on this framework, much of it developed more recently. This section focuses not on the integration framework so much as on the automation and other techniques which are applicable more widely, to theories of planar geometry or to others developing automation tools.

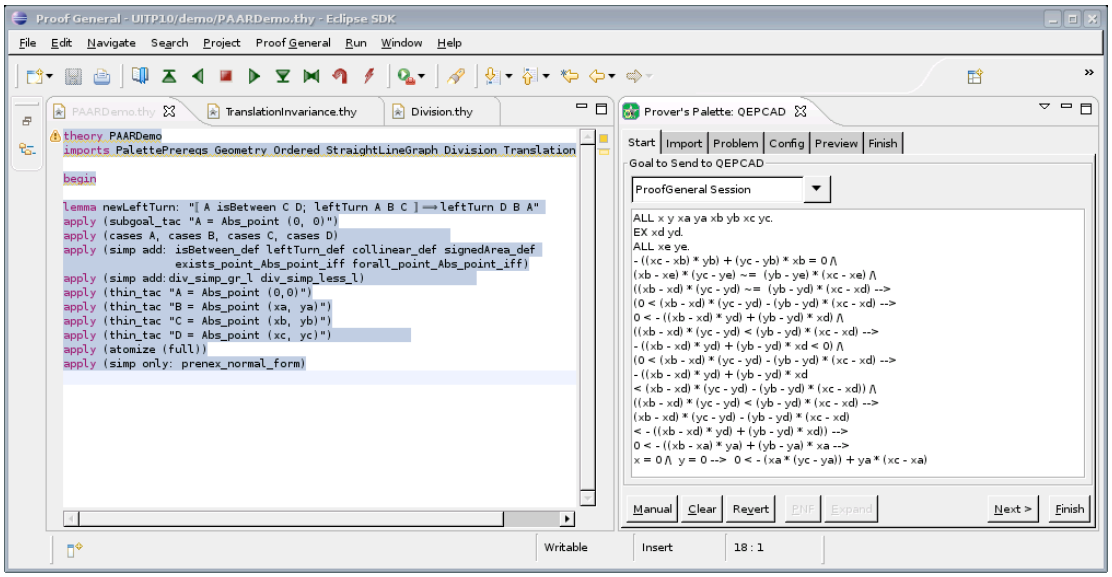


Figure 1: The Prover's Palette with Isabelle and QEPCAD

4.1 Automatic Pop-up for Useful Results

One feature of the integration framework is the ability to exploit concurrency. An external system can run automatically in the background, appearing only when it is able to solve a subgoal. Since the initial development, this has turned out to be exceedingly useful, not for the cases we expected, but when we have made a mistake in our proof: several times, we found the QEPCAD integration popping up to tell us that our current goal is false. This was not due to extreme negligence on our part (at least not the majority of times!), but due to the complexity of verifying algorithms, especially when they contain loops. We chose to formally represent our algorithms using Isabelle's development of Floyd-Hoare logic as this allowed the formal specifications of the geometric algorithms to closely resemble their implementations [7]. As a result of using this logic, our verifications relied upon us discovering the correct loop invariants (the facts which hold true on each iteration of the loops) This discovery process was often one of iterative refinement. Thus, by alerting us when a statement was flawed, the external tool accelerated the feedback cycle. This vastly reduced the time which could have been wasted attempting to prove verification conditions with incomplete loop invariants.

We believe this automation technique, performing some computation in the background, will be one of the most important mechanisms by which interactive theorem provers become widely usable. Isabelle now includes a feature which can automatically generate counter-examples for some problems. QEPCAD complements this capability by using different methods to identify false subgoals. The ability to combine multiple automation techniques in a single proving environment, where the automation is effectively invisible except when useful, has been a major assistance to our efforts.

4.2 Expansion and Prenex Normal Form

One requirement of QEPCAD is that the subgoal be expressed in prenex normal form (PNF) without reference to predicates defined elsewhere in the Isabelle theory. To minimise the effort required by the end-user, the Prover's Palette framework offers support for detecting whether these conditions (or others) are met and for automatically generating the appropriate commands for converting a subgoal to the desired form.

When some of a subgoal is detected as incompatible, the QEPCAD integration automatically disables *those parts* of the subgoal, but the integration GUI allows the user to use the remainder of the subgoal with QEPCAD. This is in keeping with the Prover's Palette philosophy that the an external tool integration should be helpful but not overly restrictive. In addition, however, the integration presents the user with a button which will automatically generate, insert and apply the commands to convert the subgoal to the required form where appropriate.

Specifically, if there are definitions that QEPCAD will not understand (*e.g.* `collinear`), the QEPCAD integration GUI gives the user the option to automatically expand them (*e.g.* `apply (simp only: collinear_def)`): the user clicks one button and the subgoal is transformed.

Additionally, where a problem is not in PNF, the integration can insert the correct Isabelle commands into the theory file to convert the problem into PNF. The Prover's Palette can also allow a user to specify that these conversions always be made automatically.

While none of these items is mathematically difficult, they are extremely tedious and this tedium limits the utility of the dependent technique. This burden is removed from the user through simple automation which does the necessary pre-processing,

4.3 Removing Division

Another obstruction to the use of QEPCAD is its inability to reason about division. All statements containing division must be rewritten in terms of multiplication. To simplify this process, we have

produced the following set of rewrite rules:

```
div_simp_eq_l:  (a/b = (c::real)) =
       ((b≠0 ⟶ a = c·b) ∧ (b=0 ⟶ c=0))
div_simp_eq_r:  ((c::real) = a/b) =
       ((b≠0 ⟶ c·b = a) ∧ (b=0 ⟶ c=0))
div_simp_neq_l:  (a/b ≠ (c::real)) =
       ((b≠0 ⟶ a ≠ c·b) ∧ (b=0 ⟶ c≠0))
div_simp_neq_r:  ((c::real) ≠ a/b) =
       ((b≠0 ⟶ c·b ≠ a) ∧ (b=0 ⟶ c≠0))
div_simp_less_l:  (a/b < (c::real)) =
       ((b>0 ⟶ a < c·b) ∧ (b<0 ⟶ a > c·b) ∧ (b=0 ⟶ 0<c))
div_simp_less_r:  ((c::real) < a/b) =
       ((b>0 ⟶ c·b < a) ∧ (b<0 ⟶ c·b > a) ∧ (b=0 ⟶ c<0))
div_simp_gr_l:  (a/b > (c::real)) =
       ((b>0 ⟶ a > c·b) ∧ (b<0 ⟶ a < c·b) ∧ (b=0 ⟶ 0>c))
div_simp_gr_r:  ((c::real) > a/b) =
       ((b>0 ⟶ c·b > a) ∧ (b<0 ⟶ c·b < a) ∧ (b=0 ⟶ c>0))
```

For the statements encountered in our proofs, these rules are sufficient to remove the division and allow the goals to be sent to QEPCAD. However, they will not work for more intricate statements involving division, such as those containing a sum of fractions. Also, it is worth noting that in Isabelle, dividing by zero equals zero; this is a design choice to ensure that division is total and we have fewer conditional rewrite rules.

5 Translation Invariance

While QEPCAD is theoretically a complete decision procedure, some of the problems we sent to it exceeded reasonable time- and/or space-complexity: either it ran out of memory or hadn't terminated after 12 hours. One of these problems is:

```
lemma segExtensionStillIntersects:  "[|X isBetween A B;
       straightEdgesIntersect e {X,B} |] ==>
         straightEdgesIntersect e {A, B}"
```

where `straightEdgesIntersect` is defined as:

```
types edge = "point set"

constdefs straightEdgesIntersect ::  "[edge, edge] => bool"
      "straightEdgesIntersect ea eb ≡
         EX a1 a2 b1 b2.  ea={a1,a2} ∧ eb={b1,b2} ∧
           ( ( {a1,a2} = {b1,b2} )
             ∨ ( (b1 isBetween a1 a2) ∨ (b2 isBetween a1 a2) ∨
               (a1 isBetween b1 b2) ∨ (a2 isBetween b1 b2) )
             ∨ ( leftTurn a1 a2 b1 ∧ leftTurn a2 a1 b2 ∧
               leftTurn b1 b2 a2 ∧ leftTurn b2 b1 a1 ) )"
```

With geometric intuition, it is easy to convince oneself that this lemma is translation invariant: it is true if and only if the problem is slid in the plane such that one point is the origin. In the QEPCAD integration

GUI, we can manually change one pair of (x,y) co-ordinates to be $(0,0)$. Sending the revised problem in 8 variables, instead of 10, yields a result from QEPCAD in 4 seconds!

This problem is not unique, and translation invariance is a common property used to justify proving geometric theorems where "without loss of generality" (WLOG) one point is the origin. Unfortunately Isabelle does not have a WLOG tactic. A recent development within HOL Light, however, has seen the introduction of a WLOG tactic [6]. This tactic reasons about many situations in mathematical written proofs where the WLOG is commonly found, including geometry. We have since extended our Isabelle theory of geometry to simplify the reduction whereby one point is taken as the origin:

```
origin == Abs_point ( 0,0 )
negative A == Abs_point ( -(xCoord A),-(yCoord A) )
translatedBy A Δ == Abs_point ( (xCoord A + xCoord Δ),
       (yCoord A + yCoord Δ) )
```

We prove that the origin is equivalent to a point negated by itself:

```
originTranslated:  origin = translatedBy A (negative A)
```

And then subsequently prove:

```
signedAreaTranslates:  signedArea A B C = signedArea
       (translatedBy A Δ) (translatedBy B Δ) (translatedBy C Δ)
leftTurnTranslates:  leftTurn A B C = leftTurn (translatedBy A Δ)
      (translatedBy B Δ) (translatedBy C Δ)
isBetweenTranslates:  A isBetween B C = (translatedBy A Δ)
      isBetween (translatedBy B Δ) (translatedBy C Δ)
```

With these lemmas it becomes straightforward to show that propositions in our theory involving a point (x,y), are equivalent to the same proposition translated by $(-x,-y)$; simplification rules then yield the proposition in terms of the origin and one fewer point.

The process is not fully automated, but it is very easy using the Prover's Palette for a user to test whether translation is worth doing, and then fairly quick to perform the translation fully formally. We note that scaling and rotation could be applied in similar ways to remove two further variables, although this has not yet been implemented in our theory.

6 Conclusion

We have, of course, benefitted from an enormous amount of work in automation and semi-automation which it would be impossible to describe exhaustively. What we have tried to do in the course of this paper is describe some of the proof techniques and automation which we have implemented, and which we have found useful, in hopes that these ideas and their implementations may prove of benefit to others. Further details on the Prover's Palette and our empirical results can be found in [10].

Our experience with theorem proving has led us to the conclusion that there is no one "magic bullet" which will make formal verification suddenly easy. Equally, however, our observations and experiences (and our results with verifying geometric algorithms, to be published later this year) leave us convinced that this will be achieved—through the gradual accumulation of theory libraries, automation techniques and proof techniques—shared among the community and subsequently improved upon. We welcome feedback on our approaches described here.

Acknowledgements. We would like to thank the reviewers for their useful comments. This work was funded by the EPSRC grant EP/E005713/1.

References

[1] Aspinall D., C. Lüth, and D. Winterstein, *A Framework for Interactive Proof*, Towards Mechanized Mathematical Assistants, Springer LNAI 4573 (2007), 161-175.

[2] Brown C. W., *QEPCAD B: a program for computing with semi-algebraic sets using CADs*, SIGSAM Bulletin, **37** (2003), 97-108.

[3] Chou S. C., X. S. Gao, and J. Z. Zhang. *Automated generation of readable proofs with geometric invariants, I. multiple and shortest proof generation*, Journal of Automated Reasoning, 17:325-3477, 1996.

[4] Chou S. C., X. S. Gao, and J. Z. Zhang. *Automated generation of readable proofs with geometric invariants, II. theorem proving with full-angles*, Journal of Automated Reasoning, 17:349-370, 1996.

[5] Collins G. E., and H. Hong, *Partial Cylindrical Algebraic Decomposition for Quantifier Elimination*, Journal of Symbolic Computation **12** (1991), 299-328.

[6] Harrison J., *Without Loss of Generality*, TPHOLs, LNCS, v. 5674, pp 43-59, 2009.

[7] Hoare C. A. R., *An axiomatic basis for computer programming*, Communications of the ACM, v.12 n. 10, pp 576-580, 1969.

[8] Janicic P., J. Narboux, and P. Quaresma, *The Area Method : a Recapitulation*, submitted to JAR, 2009.

[9] Meikle L. I., and J. D. Fleuriot, *Combining Isabelle and QEPCAD-B in the Prover's Palette*, AISC/MKM/-Calculemus (2008), 315-330.

[10] Meikle L. I., *The Formal Verification of Geometric Algorithms*, to appear as PhD Thesis, University of Edinburgh.

[11] Nipkow T., Paulson L. C., and Wenzel M. *Isabelle HOL: A Proof Assistant for Higher-Order Logic*, The tutorial can be found at: `www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/tutorial.pdf`

Using the TPTP Language for Representing Derivations in Tableau and Connection Calculi

Jens Otten
Institut für Informatik
University of Potsdam, Germany
jeotten@cs.uni-potsdam.de

Geoff Sutcliffe
Department of Computer Science
University of Miami, USA
geoff@cs.miami.edu

Abstract

The TPTP language, developed within the framework of the TPTP library, allows the representation of problems and solutions in first-order and higher-order logic. Whereas the writing of solutions in resolution calculi is well documented and used, an appropriate representation of solutions in tableau or connection calculi using the TPTP syntax has not yet been specified. This paper describes how the TPTP language can be used to represent derivations and solutions in standard tableau, sequent and connection calculi for classical first-order logic.

1 Introduction

The TPTP language specifies syntax and semantics for expressing problems in first-order and higher-order logic. It is used not only within the TPTP library [23], but also within similar problem libraries, e.g., the ILTP library [15]. The TPTP syntax for representing problems is used by a variety of automated theorem proving (ATP) systems based on different proof calculi. The TPTP language also allows representation of solutions, e.g., derivations and models, produced by ATP systems. The writing of derivations in resolution calculi is well documented and specified [25]. At the last CADE system competition, CASC-22 [24], three of the five ATP systems that output proofs in the core FOF division use the TPTP syntax. All three of those systems produce proofs that are based on resolution calculi.

Even though the TPTP syntax is flexible, the presentation of derivations in, e.g., tableau, sequent or connection calculi is not straightforward. Derivations in these calculi differ significantly from derivations in the resolution calculus. Whereas the leaves of a proof in the tableau calculus consists of the axioms of the *calculus*, the leaves of a derivation in the resolution calculus consists of the formulae of the given problem; the axiom of the (formal) resolution calculus is the empty clause [18], which occurs only at the root of a refutation.

This paper describes how the TPTP language can be used to represent derivations and proofs in standard tableau and connection calculi. As the sequent calculus is closely related to the tableau calculus, this can easily be adapted to present derivations in the sequent calculus as well. This is a proposed format, not yet formally established as a TPTP standard; community feedback with suggestions for improvement are welcome. The goal is to produce a format that is compatible with the existing format for representing derivations (reviewed in Section 2.2), so that existing TPTP infrastructure for proof processing, e.g., the GDV proof verifier [21] and the IDV proof visualizer [26], can be used with little or no modification.

2 The TPTP Language

The TPTP language is suitable for representing problems as well as derivations in first-order and higher-order logic. The following description presents its main concepts. A detailed definition is part of the TPTP library [23]; see also [25].

```
%------------------------------------------------------------------------------
% File     : SYN054+1 : TPTP v4.0.1. Released v2.0.0.
% Domain   : Syntactic
% Problem  : Pelletier Problem 24
% Status   : Theorem
% Rating   : 0.00 v2.1.0
%------------------------------------------------------------------------------
fof(pel24_1,axiom, ( ~ ( ? [X] : ( big_s(X) & big_q(X) ) ) )).
fof(pel24_2,axiom, ( ! [X] : ( big_p(X) => ( big_q(X) | big_r(X) ) ) )).
fof(pel24_3,axiom, ( ~ ( ? [X] : big_p(X) ) => ? [Y] : big_q(Y) )).
fof(pel24_4,axiom, ( ! [X] : ( ( big_q(X) | big_r(X) ) => big_s(X) ) )).
fof(pel24,conjecture, ( ? [X] : ( big_p(X) & big_r(X) ) )).
%------------------------------------------------------------------------------
```

Figure 1: The presentation of the TPTP problem `SYN054+1`

2.1 Representing Problems

The top level building blocks for problems using the TPTP syntax are annotated formulae, of the following form:

language(name,role,formula,source,useful_info).

The *language* is one of `thf`, `fof`, or `cnf`, for formulae in typed higher-order, first-order, and clause normal form. Each annotated formula has a unique *name*. The *role* is, e.g., `axiom` or `conjecture`. The *source* describes where the formula came from, e.g., an input file, and *useful_info* is a list of user information. The last two fields are optional.

Example 1. *Pelletier's problem 24 [14] consists of the following subformulae.*

$$\begin{array}{cll} \neg(\exists x(Sx \wedge Qx)) & \textit{Axiom 1} & (1) \\ \forall(Px \Rightarrow (Qx \vee Rx)) & \textit{Axiom 2} & (2) \\ \neg(\exists x Px) \Rightarrow \exists y Qy & \textit{Axiom 3} & (3) \\ \forall x((Qx \vee Rx) \Rightarrow Sx) & \textit{Axiom 4} & (4) \\ \exists x(Px \wedge Rx) & \textit{Conjecture} & (5) \end{array}$$

It stands for the first-order formula $(\textit{Axiom 1} \wedge \textit{Axiom 2} \wedge \textit{Axiom 3} \wedge \textit{Axiom 4}) \Rightarrow \textit{Conjecture}$*. This problem is in the TPTP library under the name* `SYN054+1`*. Its representation using the TPTP syntax is given in Figure 1 (with an abbreviated version of the full TPTP header).*

2.2 Representing Derivations

A derivation (in the resolution calculus) is a directed acyclic graph whose leaf nodes are formulae from the problem, and whose interior nodes are formulae inferred from parent formulae. A refutation is a derivation that has the root node *false*, representing the empty clause. A derivation written in the TPTP language is a list of annotated formulae, as for problems. For derivations the *source* has one of the forms

`file(`*file_name,file_info*`)`

`inference(`*inference_name,inference_info,parents*`)`

The former is used for formulae taken from the problem file. The latter is used for inferred formulae, in which *inference_name* is the name of the inference rule applied by the ATP system, *inference_info* is a

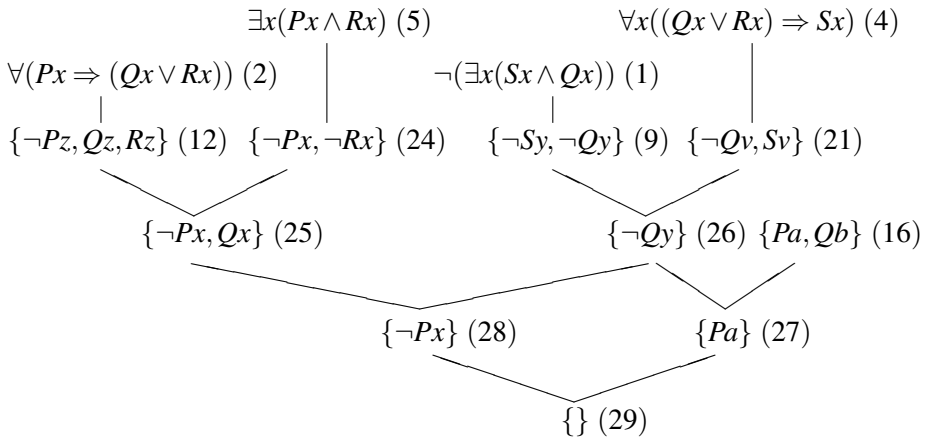


Figure 2: A derivation for SYN054+1 in the resolution calculus

[]ed list of additional information about the inference, and *parents* is a list of the (logical) parents' node names in the derivation. The *inference_info* normally includes a `status()` term that record the semantic relationship between the parents and the inferred formula as an SZS ontology value [22]. Variable bindings applied to a logical parent are captured in `bind`/2 terms following the parent's name.

```
%--------------------------------------------------------------------------------------------
fof(1, axiom,~(?[X1]:(big_s(X1)&big_q(X1))),file('SYN054+1.p',pel24_1)).
fof(2, axiom,![X1]:(big_p(X1)=>(big_q(X1)|big_r(X1))),file('SYN054+1.p',pel24_2)).
fof(3, axiom,(~(?[X1]:big_p(X1))=>?[X2]:big_q(X2)),file('SYN054+1.p',pel24_3)).
fof(4, axiom,![X1]:((big_q(X1)|big_r(X1))=>big_s(X1)),file('SYN054+1.p',pel24_4)).
fof(5, conjecture,?[X1]:(big_p(X1)&big_r(X1)),file('SYN054+1.p',pel24)).
fof(6, negated_conjecture,~(?[X1]:(big_p(X1)&big_r(X1))),inference(assume_negation,[],[5])).
fof(7, plain,![X1]:(~(big_s(X1))|~(big_q(X1))),inference(fof_nnf,[],[1])).
fof(8, plain,![X2]:(~(big_s(X2))|~(big_q(X2))),inference(variable_rename,[],[7])).
cnf(9, plain,(~big_q(X1)|~big_s(X1)),inference(split_conjunct,[],[8])).
fof(10,plain,![X1]:(~(big_p(X1))|(big_q(X1)|big_r(X1))),inference(fof_nnf,[],[2])).
fof(11,plain,![X2]:(~(big_p(X2))|(big_q(X2)|big_r(X2))),inference(variable_rename,[],[10])).
cnf(12,plain,(big_r(X1)|big_q(X1)|~big_p(X1)),inference(split_conjunct,[],[11])).
fof(13,plain,(?[X1]:big_p(X1)|?[X2]:big_q(X2)),inference(fof_nnf,[],[3])).
fof(14,plain,(?[X3]:big_p(X3)|?[X4]:big_q(X4)),inference(variable_rename,[],[13])).
fof(15,plain,(big_p(esk1_0)|big_q(esk2_0)),inference(skolemize,[],[14])).
cnf(16,plain,(big_q(esk2_0)|big_p(esk1_0)),inference(split_conjunct,[],[15])).
fof(17,plain,![X1]:((~(big_q(X1))&~(big_r(X1)))|big_s(X1)),inference(fof_nnf,[],[4])).
fof(18,plain,
    ![X2]:((~(big_q(X2))&~(big_r(X2)))|big_s(X2)),inference(variable_rename,[],[17])).
fof(19,plain,
    ![X2]:((~(big_q(X2))|big_s(X2))&(~(big_r(X2))|big_s(X2))),inference(distribute,[],[18])).
cnf(21,plain,(big_s(X1)|~big_q(X1)),inference(split_conjunct,[],[19])).
fof(22,negated_conjecture,![X1]:(~(big_p(X1))|~(big_r(X1))),inference(fof_nnf,[],[6])).
fof(23,negated_conjecture,
    ![X2]:(~(big_p(X2))|~(big_r(X2))),inference(variable_rename,[],[22])).
cnf(24,negated_conjecture,(~big_r(X1)|~big_p(X1)),inference(split_conjunct,[],[23])).

cnf(25,plain,(big_q(X1)|~big_p(X1)),inference(csr,[],[12,24])).
cnf(26,plain,(~big_q(X1)),inference(csr,[],[9,21])).
cnf(27,plain,(big_p(esk1_0)),inference(sr,[],[16,26])).
cnf(28,plain,(~big_p(X1)),inference(sr,[],[25,26])).
cnf(29,plain,($false),inference(sr,[],[27,28])).
%--------------------------------------------------------------------------------------------
```

Figure 3: A derivation for SYN054+1 in the resolution calculus using the TPTP syntax

Example 2. *Figure 2 shows a conversion of some of the axioms and the negated conjecture of problem* `SYN054+1` *from Example 1 to clause normal form, and a subsequent refutation of the clause normal form in the resolution calculus [16]. The leaf nodes are the formulae of the problem. As the root node is the empty clause, the derivation is a proof for problem* `SYN054+1`*. The representation of this derivation using the TPTP syntax is given in Figure 3. It is a slightly simplified version of the original proof output by the EP system [17]. The five inferences of the resolution proof are represented by the nodes 25 to 29. The nodes of the proof in Figure 2 are annotated by the corresponding EP node numbers.*

3 Representing Derivations in the Tableau Calculus

Tableau calculi are well-known proof search calculi for classical and non-classical logics [4, 6]. The axiom and 12 rules of a standard tableau calculus for classical logic are given in Table 1 [20]. It uses *signed formulae* of the form A^T or A^F, in which A is a first-order formula and T/F is its sign (or polarity). The signed formula A^F can be interpreted as $A \Rightarrow \mathit{false}$. The usage of signed formulae allows an elegant and uniform representation of the rules of the tableau calculus. The α-rules add formulae to a branch of a derivation, and the β-rules split a branch of the derivation into two branches. When eliminating a universal quantifier using the γ-rule, all free occurrences of the variable x in A are replaced by the term t. In the δ-rule the variable x is replaced by a Skolem term that consists of a unique Skolem function symbol sk_i and all variables $x_1, \ldots, x_n$ that occur free in A. A formula A is valid if, and only if, there is a derivation of A^F in the tableau calculus.

Table 1: The axiom and the rules of the tableau calculus

Axiom	$\dfrac{A^T \quad A^F}{\bot}$		
α-rules	$\wedge^T \ \dfrac{(A \wedge B)^T}{\begin{array}{c}A^T\\B^T\end{array}}$	$\vee^F \ \dfrac{(A \vee B)^F}{\begin{array}{c}A^F\\B^F\end{array}}$	$\Rightarrow^F \ \dfrac{(A \Rightarrow B)^F}{A^T \wedge B^F}$
	$\neg^T \ \dfrac{(\neg A)^T}{A^F}$	$\neg^F \ \dfrac{(\neg A)^F}{A^T}$	
β-rules	$\wedge^F \ \dfrac{(A \wedge B)^F}{A^F \quad B^F}$	$\vee^T \ \dfrac{(A \vee B)^T}{A^T \quad B^T}$	$\Rightarrow^T \ \dfrac{(A \Rightarrow B)^T}{A^F \quad B^T}$
γ-rules	$\forall^T \ \dfrac{(\forall x A)^T}{A^T[x\backslash t]}$	$\exists^F \ \dfrac{(\exists x A)^F}{A^F[x\backslash t]}$	
δ-rules	$\forall^F \ \dfrac{(\forall x A)^F}{A^F[x\backslash sk_i(x_1,\ldots,x_n)]}$	$\exists^T \ \dfrac{(\exists x A)^T}{A^T[x\backslash sk_i(x_1,\ldots,x_n)]}$	

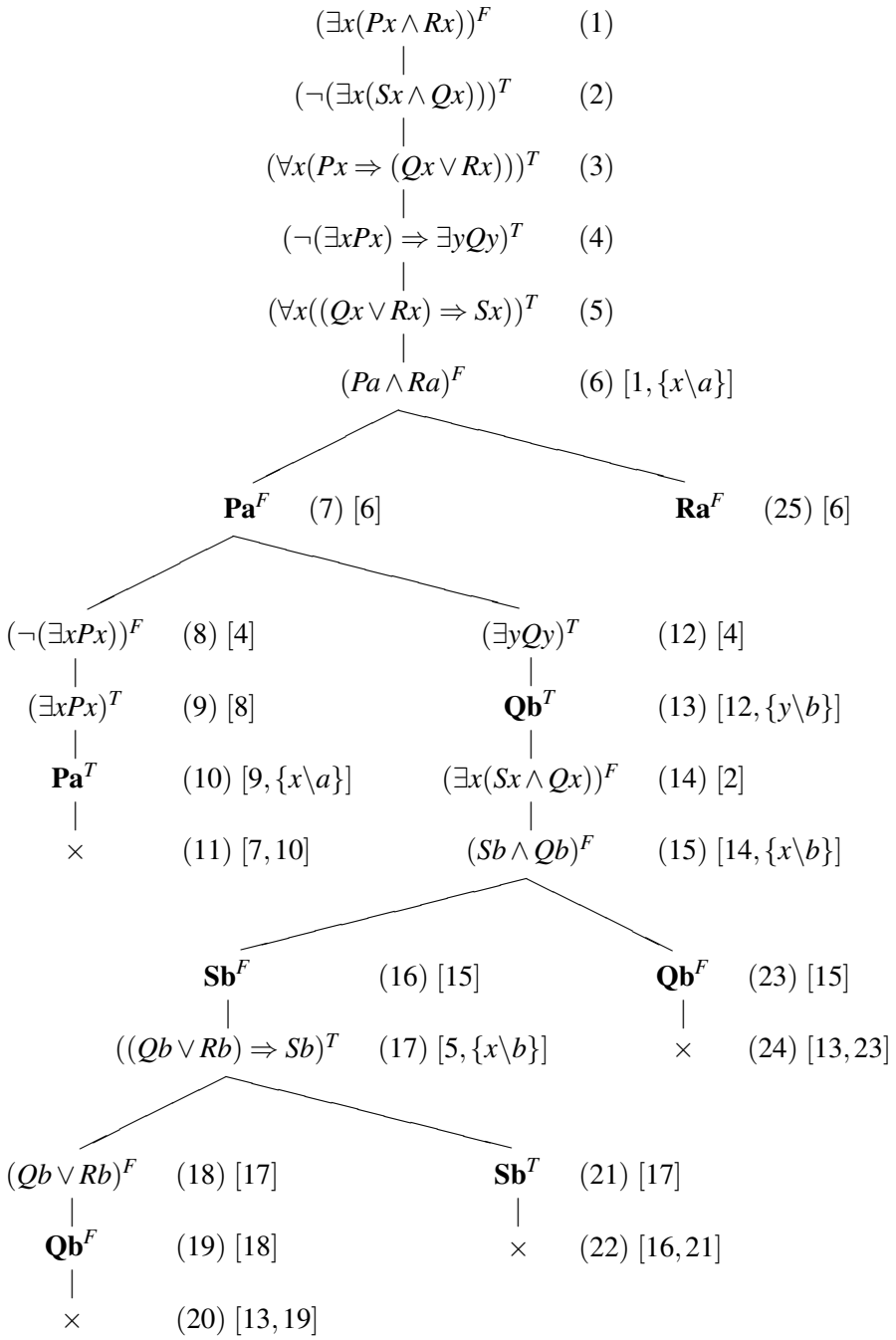


Figure 4: A derivation for SYN054+1 in the tableau calculus

Example 3. *A derivation of problem* SYN054+1 *from Example 1 in the tableau calculus is shown in Figure 4. It follows the common representation of standard tableau calculi for classical logic [6]. Each node is annotated in ( )s by its number and in [ ]s by the number of the node whose formula is used as the premise of the inference rule, its* logical parent*. Additionally, a substitution is given when the* γ*- or* δ*-rule is applied. The constants a and b are Skolem terms. Branches that are closed by an axiom are marked with* $\times$*. The derivation in Figure 4 is not a proof, because the rightmost branch (node 25) is not closed by an axiom.*

```
%----------------------------------------------------------------------------------------------------
fof(0, conjecture,?[X]:(big_p(X)&big_r(X)),file('SYN054+1.p',pel24)).
fof(1, negated_conjecture,(~ ?[X]:(big_p(X)&big_r(X)))=>$false,inference(neg_conj,[pparent([0])],[0])).
fof(2, axiom,~ ?[X]:(big_s(X)&big_q(X)),file('SYN054+1.p',pel24_1)).
fof(3, axiom,![X]:(big_p(X)=>(big_q(X)|big_r(X))),file('SYN054+1.p',pel24_2)).
fof(4, axiom,~ ?[X]:big_p(X)=>?[Y]:big_q(Y),file('SYN054+1.p',pel24_3)).
fof(5, axiom,![X]:((big_q(X)|big_r(X))=>big_s(X)),file('SYN054+1.p',pel24_4)).

fof(6, plain,(big_p(X)&big_r(X))=>$false,
       inference(exists_F,[status(thm),pparent([5])],[1:[bind(X,$fot(a))]])).

fof(7, plain,~big_p(a),inference(and_F,[and_F(split,[position(l)]),pparent([6])],[6])).
fof(8, plain,(~ ?[X]:big_p(X))=>$false,
       inference(implies_T,[implies_T(split,[position(ll)]),pparent([7])],[4])).
fof(9, plain,?[X]:big_p(X),inference(neg_F,[status(thm),pparent([8])],[8])).
fof(10,plain,big_p(a),inference(exists_T,[status(thm),pparent([9])],[9])).
fof(11,plain,$false,inference(axiom,[status(thm),pparent([10])],[7,10])).

fof(12,plain,?[Y]:big_q(Y),inference(implies_T,[implies_T(split,[position(lr)]),pparent([7])],[4])).
fof(13,plain,big_q(b),inference(exists_T,[status(thm),pparent([12])],[12:[bind(Y,$fot(b))]])).
fof(14,plain,(?[X]:(big_s(X)&big_q(X)))=>$false,inference(neg_T,[status(thm),pparent([13])],[2])).
fof(15,plain,(big_s(b)&big_q(b))=>$false,
       inference(exists_F,[status(thm),pparent([14])],[14:[bind(X,$fot(b))]])).

fof(16,plain,~big_s(b),inference(and_F,[and_F(split,[position(lrl)]),pparent([15])],[15])).
fof(17,plain,(big_q(b)|big_r(b))=>big_s(b),
       inference(forall_T,[status(thm),pparent([16])],[5:[bind(X,$fot(b))]])).
fof(18,plain,(big_q(b)|big_r(b))=>$false,
       inference(implies_T,[implies_T(split,[position(lrll)]),pparent([17])],[17]).
fof(19,plain,~big_q(b),inference(or_F,[status(thm),pparent([18])],[18])).
fof(20,plain,$false,inference(axiom,[status(thm),pparent([19])],[13,19])).
fof(21,plain,big_s(b),inference(implies_T,[implies_T(split,[position(lrlr)]),pparent([17])],[17])).
fof(22,plain,$false,inference(axiom,[status(thm),pparent([21])],[16,21])).

fof(23,plain,~big_q(b),inference(and_F,[and_F(split,[position(lrr)]),pparent([15])],[15])).
fof(24,plain,$false,inference(axiom,[status(thm),pparent([23])],[13,23])).
fof(25,plain,~big_r(a),inference(and_F,[and_F(split,[position(r)]),pparent([6])],[6])).
%----------------------------------------------------------------------------------------------------
```

Figure 5: A derivation for SYN054+1 in the tableau calculus using the TPTP syntax

Even though a derivation in the tableau calculus is still an acyclic directed graph, its structure is different from the structure of a derivation in the resolution calculus. Hence the proof presentation using the TPTP language needs to be adapted. For a tableau, in addition to listing the logical parents of each formula in the parents list, the *physical* parent of each node is recorded in a `pparent()` term in the inference information list. The branching of the tableau is recorded in the same way as splitting inferences are recorded in CNF refutations [25, 21]. The *inference_name* is `axiom` or the name of the applied inference rule, i.e., `and_T`, `or_F`, `implies_F`, `neg_T`, `neg_F`, `and_F`, `or_T`, `implies_T`, `forall_T`, `exists_F`, `forall_F`, or `exists_T`. A formula of the form A^F is represented in the TPTP language by the formula `A=>$false`, if A is a non-atomic formula; otherwise, if A is an atomic formula, it is represented by `~`A. A formula of the form A^T is represented by A.

Example 4. *The derivation of Figure 4 is shown using the TPTP syntax in Figure 5. The non-negated original conjecture is added as node 0. Observe the fact that the physical parent might differ from the logical parent. For example, the physical parent of node 8 is node 7, its formula is* $(\neg(\exists x P x))^F$*, which is obtained by an* implies_T *inference, whose premise is the formula of node 4.*

The proof representation is independent from the specific proof *search* (algorithm). Hence, the proposed format can also be used to represent derivations obtained by, e.g., free-variable tableaux or a proof search using iterative deepening.

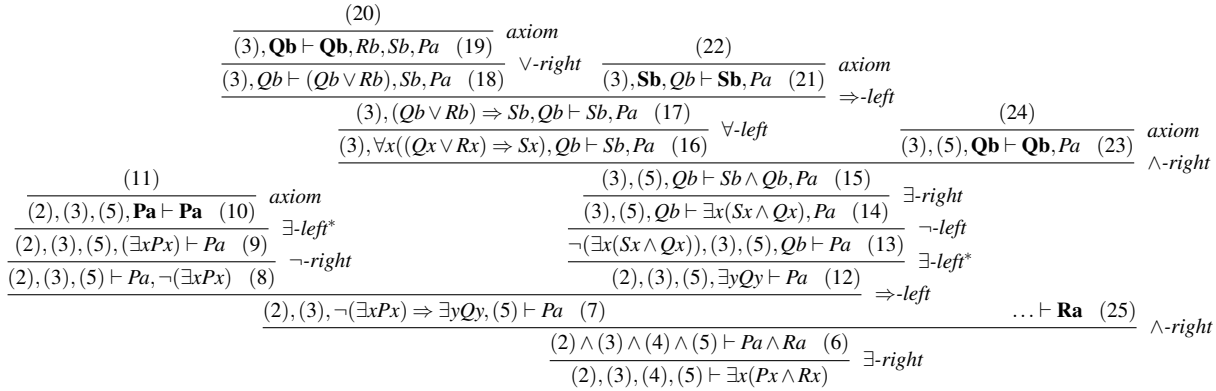


Figure 6: A derivation for SYN054+1 in the sequent calculus

Representing Derivations in the Sequent Calculus. The tableau calculus is closely related to the sequent calculus [5]. Therefore, the TPTP format for tableau derivations can also be used for representing derivations in standard sequent calculi [20]. Formulae of the form A^T occur (only) on the left side of the sequents (the *antecedent*), formulae of the form A^F occur (only) on the right side (the *succedent*). Each inference rule $rule^T$ and $rule^F$ in the tableau calculus corresponds to exactly one rule *rule-left* and *rule-right* in the sequent calculus, respectively.

Example 5. *Figure 6 shows the the sequent derivation that corresponds to the tableau derivation given in Figure 4. However, as the Eigenvariable condition needs to be respected (for the ∃-left* rule), it might be necessary to reorder some inference rules in order to obtain a correct sequent proof.*

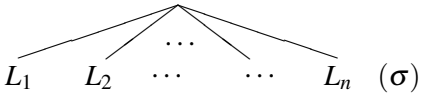
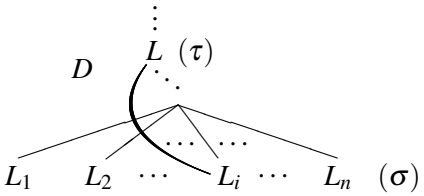
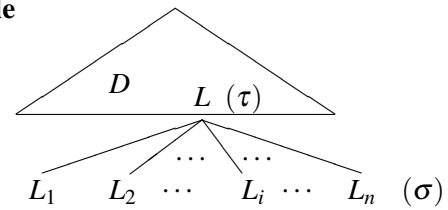
4 Representing Derivations in the Connection Calculus

Connection calculi, e.g. the connection method [2], the connection tableau calculus [8] and the model elimination calculus [9], are established proof search calculi. In principle, derivations in the clausal connection calculus can be seen as derivations in the tableau calculus with a connectedness condition [6]. But to this end many additional inferences need to be inserted. Hence it is advantageous to have a different representation of derivations, in which each inference in the connection calculus is related to exactly one inference in the representation (using the TPTP language).

The main concept of connection calculi is the guidance of the proof search by connections. A *connection* is a set of literals with opposite polarity but identical atomic formulae, i.e., $\{L_1, L_2\}$ is a connection if, and only if, $L_1 = \neg L_2$ or $\neg L_1 = L_2$. The connection calculus has three main inference rules: *start*, *reduction*, and *extension* rule. These rules are depicted in Table 2. For details see [2, 8, 11]. A formula F in disjunctive (conjunctive) clause normal form is valid (unsatisfiable) if, and only if, there is a derivation in the (clausal) connection calculus such that every literal is an element of at least one connection.

Example 6. *A derivation of problem* SYN054+1 *from Example 1 in the clausal connection calculus is shown in Figure 7 and Figure 8. Again, each inference is annotated by its number, the clause number used in the inference, and a substitution. The inferences with numbers 5 and 10 are applications of reduction rules. The branch containing the circled literal, i.e., inference number 7, is closed by an application of the lemma rule (see [11] for details). The derivation is a proof as every literal is element of a connection, hence all branches are closed by at least one connection. The major left branch in Figure 7 is a compact representation of the derivation shown in Figure 4, containing only the bold literals of Figure 4.*

Table 2: The rules of the (clausal) connection calculus

Start rule		is a derivation for a clause $C = \{L_1, \ldots, L_n\}$ and a (term) substitution σ.
Reduction rule		is a derivation, if D (without the thick line) is a derivation, L_i is a (leaf) literal not element of a connection, L is a literal on the path from L_i to the root, and $\{\tau(L), \sigma(L_i)\}$ is a connection.
Extension rule		is a derivation for a clause $C = \{L_1, \ldots, L_n\}$ and a substitution σ, if D is a derivation, L is a (leaf) literal not element of a connection, and $\{\tau(L), \sigma(L_i)\}$ is a connection for some i.

A derivation in the clausal connection calculus using the TPTP language is a list of clausal annotated formulae, as described in Section 2.2. Similar to a tableau, the *parents* is an ordered list in which the first element is the name of the physical parent of the node, and the following element is the name of the logical parent, i.e., the clause of the inference. Again, variable bindings are captured in `bind`/2 terms. Additionally, the number of the selected literal of the physical parent is captured in a `cnf_selected`/1 term. Optionally, such a term can be assigned to the logical parent as well, which would make it easier to identify the connection. The *inference_name* is the name of the applied inference rule, i.e., either `start`, `reduction`, `extension`, or `lemma`.

Example 7. *The derivation of Figure 7 is shown using this TPTP syntax in Figure 9. The output is produced by the most recent version of the* leanCoP *system [12, 10].*

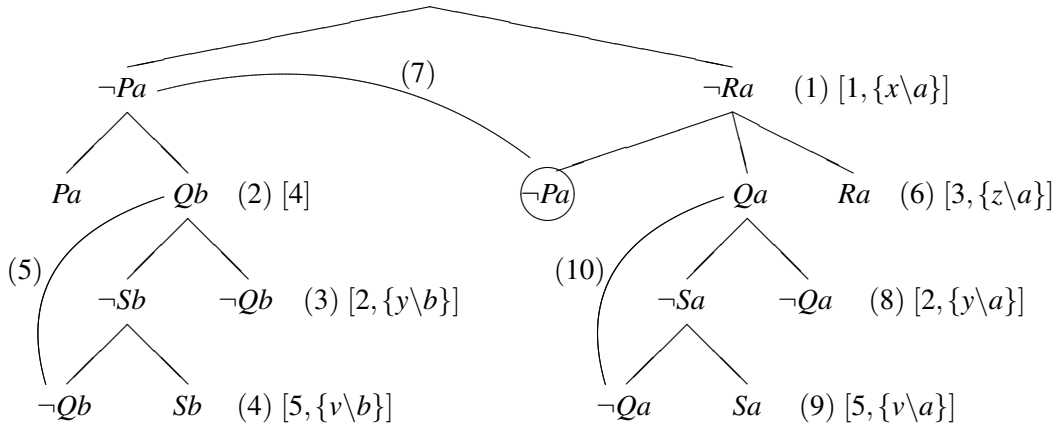


Figure 7: A derivation for `SYN054+1` in the connection calculus (tableau representation)

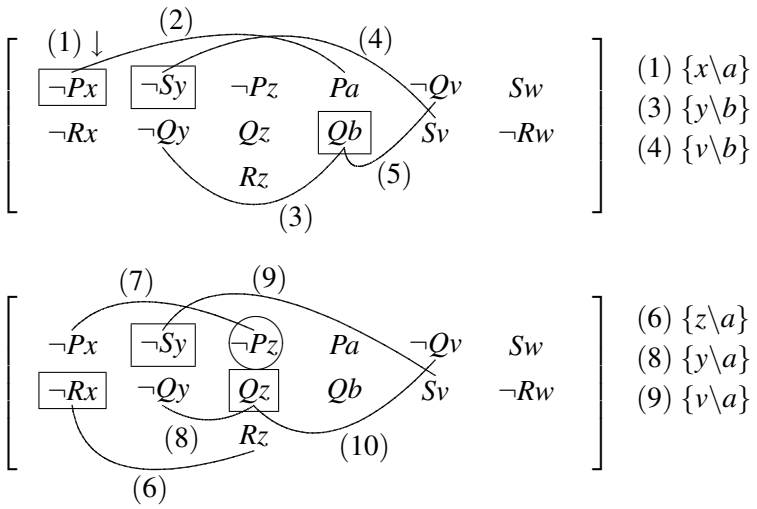


Figure 8: A derivation for SYN054+1 in the connection calculus (matrix representation)

```
%--------------------------------------------------------------------------------------------------
fof(pel24_1,axiom,~ ?[X]:(big_s(X)&big_q(X)),file('SYN054+1.p',pel24_1)).
fof(pel24_2,axiom,![X]:(big_p(X)=>(big_q(X)|big_r(X))),file('SYN054+1.p',pel24_2)).
fof(pel24_3,axiom,~ ?[X]:big_p(X)=>?[Y]:big_q(Y),file('SYN054+1.p',pel24_3)).
fof(pel24_4,axiom,![X]:((big_q(X)|big_r(X))=>big_s(X)),file('SYN054+1.p',pel24_4)).
fof(pel24,conjecture,?[X]:(big_p(X)&big_r(X)),file('SYN054+1.p',pel24)).

fof(f0,negated_conjecture,~ ?[X]:(big_p(X)&big_r(X)),
       inference(negate_conjecture,[status(cth)],[pel24])).
cnf(c1,plain,(~big_p(X)|~big_r(X)),inference(clausify,[status(esa)],[f0])).
cnf(c2,plain,(~big_s(Y)|~big_q(Y)),inference(clausify,[status(esa)],[pel24_1])).
cnf(c3,plain,(~big_p(Z)|big_q(Z)|big_r(Z)),inference(clausify,[status(esa)],[pel24_2])).
cnf(c4,plain,(big_p(a)|big_q(b)),inference(clausify,[status(esa)],[pel24_3])).
cnf(c5,plain,(~big_q(V)|big_s(V)),inference(clausify,[status(esa)],[pel24_4])).

cnf(1,plain,(~big_p(a)|~big_r(a)),
      inference(start,[status(thm)],[c1:[bind(X,$fot(a))]])).
cnf(2,plain,(big_p(a)|big_q(b)),
      inference(extension,[status(thm),pparent([1:[cnf_select([1])]])],[c4])).
cnf(3,plain,(~big_s(b)|~big_q(b)),
      inference(extension,[status(thm),pparent([2:[cnf_select([2])]])],[c2:[bind(Y,$fot(b))]])).
cnf(4,plain,(~big_q(b)|big_s(b)),
      inference(extension,[status(thm),pparent([3:[cnf_select([1])]])],[c5:[bind(V,$fot(b))]])).
cnf(5,plain,$false,
      inference(reduction,[pparent([4:[cnf_select([1])]])],[2])).
cnf(6,plain,(~big_p(a)|big_q(a)|big_r(a)),
      inference(extension,[status(thm).pparent([1:[cnf_select([2])]])],[c3:[bind(Z,$fot(a))]])).
cnf(7,plain,$false,
      inference(lemma,[pparent([6:[cnf_select([1])]])],[1])).
cnf(8,plain,(~big_s(a)|~big_q(a)),
      inference(extension,[status(thm),pparent([6:[cnf_select([2])]])],[c2:[bind(Y,$fot(a))]])).
cnf(9,plain,(~big_q(a)|big_s(a)),
      inference(extension,[status(thm),pparent([8:[cnf_select([1])]])],[c5:[bind(V,$fot(a))]])).
cnf(10,plain,$false,
      inference(reduction,[pparent([9:[cnf_select([1])]])],[6])).
%--------------------------------------------------------------------------------------------------
```

Figure 9: A derivation for SYN054+1 in the connection calculus using the TPTP syntax

5 Conclusion

A proposal for representing standard tableau, sequent and connection calculi in the TPTP language has been presented. Even though derivations in these calculi differ significantly from those in the resolution calculus, the existing TPTP syntax is flexible enough to represent derivations in these calculi as well. A common standard for presentation of derivations and proofs will increase the interoperability between ATP systems, ATP tools, and application software (see, e.g., [19]).

Future work includes the development of tools to translate connection proofs into sequent proofs, which are often used in interactive proof editors, such as Coq [1], NuPRL [3] or PVS [13].

A possible extension of the current work includes the use of the TPTP language to represent derivations in tableau and connection calculi for non-classical logics, e.g., intuitionistic and modal logics [7, 27]. These calculi often use additional annotations, e.g., a prefix that is assigned to each formula. The TPTP syntax might need to be carefully extended in order to allow the presentation of derivations in these calculi as well.

References

[1] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[2] W. Bibel. *Automated Theorem Proving*. Vieweg and Sohn, 1987.

[3] R. Constable, S. Allen, H. Bromly, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[4] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.

[5] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 36:176–210, 405–431, 1935.

[6] R. Hähnle. Tableaux and Related Methods. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 100–178. Elsevier Science, 2001.

[7] C. Kreitz and J. Otten. Connection-based Theorem Proving in Classical and Non-classical Logics. *Journal of Universal Computer Science*, 5:88–112, 1999.

[8] R. Letz and G. Stenz. Model Elimination and Connection Tableau Procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 2015–2114. Elsevier Science, 2001.

[9] D.W. Loveland. Mechanical Theorem Proving by Model Elimination. *Journal of the ACM*, 15(2):236–251, 1968.

[10] J. Otten. leanCoP 2.0 and ileancop 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic. In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 283–291, 2008.

[11] J. Otten. Restricting Backtracking in Connection Calculi. *AI Communications*, 23(2-3):159–182, 2010.

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

[13] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 411–414. Springer-Verlag, 1996.

[14] F.J. Pelletier. Seventy-five Problems for Testing Automatic Theorem Provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.

[15] T. Raths, J. Otten, and C. Kreitz. The ILTP Problem Library for Intuitionistic Logic - Release v1.1. *Journal of Automated Reasoning*, 38(1-2):261–271, 2007.

[16] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

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

[18] J. Schumann. *Automated Theorem Proving in Software Engineering*. Springer-Verlag, 2002.

[19] J. Siekmann, C. Benzmüller, and S. Autexier. Computer Supported Mathematics with OMEGA. *Journal of Applied Logic*, 4(4):533–559, 2006.

[20] R.M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968.

[21] G. Sutcliffe. Semantic Derivation Verification. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.

[22] G. Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 38–49, 2008.

[23] 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.

[24] G. Sutcliffe. The CADE-22 Automated Theorem Proving System Competition - CASC-22. *AI Communications*, 23(1):47–60, 2010.

[25] G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 67–81, 2006.

[26] S. Trac, Y. Puzis, and G. Sutcliffe. An Interactive Derivation Viewer. In S. Autexier and C. Benzmüller, editors, *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers, 3rd International Joint Conference on Automated Reasoning*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 109–123, 2006.

[27] A. Waaler. Connections in Nonclassical Logics. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1487–1578. Elsevier Science, 2001.