The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006

IJCAR Workshop

ESCoR:
Empirically Successful Computerized Reasoning

August 21st, 2006

Proceedings

Editors:

G. Sutcliffe, R. Schmidt, S. Schulz

Maintaining the ACL2 Theorem Proving System

Matt Kaufmann and J Strother Moore
University of Texas
{kaufmann,moore}@cs.utexas.edu

Abstract

This talk will provide a view into the task of improving the ACL2 theorem prover to meet users' needs.

1 Introduction

The goal of this talk is to provide a sense of some possible challenges to making a theorem proving system useful in practice. Specifically, we will draw from our experiences maintaining the ACL2 theorem proving system [KMM00b, KM04a]. Although ACL2 incorporates years of ongoing research in automated reasoning, the focus of this talk is on the engineering required to make the system useful.

Imagine you're at lunch, where two guys are blabbing to you about their work. Fortunately, this blabbing might be of some interest, since they are talking about how they make their theorem proving system easier to use. They definitely want you to ask questions and share your own related experiences, though you may find it hard to interrupt them after they get started.

But even before we eat, they insist on providing some general background on their system so that when they get going, at least they might make some sense. By the time dessert arrives, these two guys will be eager for questions and comments, if for no other reason than that they can eat!

Thus, this talk consists of three parts.

First, *Before We Eat*: While we're walking to lunch you'll get some general background on ACL2. Then while we're waiting for a table, we'll see a small example that gives a sense of how ACL2 is used. After we're seated, we'll take a quick look at a list of features that we'll come across while we eat.

Next, for the *Main Course*, we will focus on some selected items taken from recent ACL2 release notes (minimally edited in a few cases). These selected items should give a sense of what can be involved in maintaining an empirically successful automated reasoning system.

Finally, we will have a give-and-take during *Dessert*. With any luck, the ensuing questions and comments will be stimulating and informative for all of us.

We will use footnotes for material that we will make a point of skipping during the talk, due to time constraints.[1]

[1] A related talk in 1991 [Kau91] gives a large list of aspects of mechanized reasoning systems and

2 Before We Eat — Some general background on ACL2

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." The ACL2 system is the latest in the Boyer-Moore family of provers, and is joint work of Matt Kaufmann and J Moore, with substantial early contributions from Bob Boyer and ongoing contributions from many. The paper [KM04b] provides a reasonably self-contained introduction to ACL2, including relevant background and how to use the system. Quoting from that paper:

> "ACL2" is the name of a functional programming language (based on Common Lisp), a first-order mathematical logic, and a mechanical theorem prover. ACL2, which is sometimes called an "industrial strength version of the Boyer-Moore system," is the product of Kaufmann and Moore, with many early design contributions by Boyer. It has been used for a variety of important formal methods projects of industrial and commercial interest, including....

A long but incomplete list of applications is then given, including various algorithms, software, and hardware designs.

ACL2 is freely available, distributed under the GPL [gpl]. It can be obtained from the ACL2 home page [KM04a], which also has links to many papers that describe ACL2 applications, as well other useful links (mailing lists, tours, demos, documentation, workshops, and installation instructions).

Here is some relevant data.

- Some milestones:
  - 1973: The Boyer/Moore "Edinburgh Pure Lisp Theorem Prover"
  - 1979: Boyer and Moore, A Computational Logic [BM79]
  - 1986: Kaufmann joins Boyer/Moore project
  - 1988: Boyer and Moore, A Computational Logic Handbook (1997, 2nd ed. [BM97])
  - 1989: Boyer and Moore begin ACL2
  - 1992: Final release of Boyer-Moore "Nqthm" prover
  - 1993: Kaufmann formally added as a co-author of ACL2
  - 2000: Kaufmann, Manolios, and Moore write a book on ACL2, Computer-Aided Reasoning: An Approach [KMM00b], and edit proceedings of the first ACL2 workshop, Computer-Aided Reasoning: ACL2 Case Studies [KMM00a]

comments on them. The focus here is narrower: What sorts of things must be done to make such a system useful? Our focus is actually still narrower, as for example the following are critical but not addressed directly here: fundamental reasoning algorithms, execution efficiency, logical foundations, system architecture, and trust. Rather, we present here selections from release notes that give a flavor of the maintenance required on a particular mature system, ACL2, in order to make it a system that (some) people want to use. That is, the point here is not to give an overview of what ACL2 provides, but to focus on maintenance.

  - 2006: Boyer, Kaufmann, and Moore win ACM Software System Award for Boyer-Moore family of provers

- Version 3.0 source files size: 8.3M of Common Lisp (including source code, documentation strings, and comments)
- There are 229 release note items strictly after Version 2.8 (March, 2004) as follows (but we'll look at just a few of these):
  - 63 in Version 2.9, October, 2004
  - 19 in Version 2.9.1, December, 2004
  - 30 in Version 2.9.2, April, 2005
  - 31 in Version 2.9.3, August, 2005
  - 53 in Version 2.9.4, February, 2006
  - 33 in Version 3.0, June, 2006

An interesting aspect of ACL2 is that it is written in its own formal language (with the exception of a small amount of Common Lisp code mainly of a bootstrapping nature). This has forced us to make ACL2's formal programming language, which is a carefully-crafted extension of an applicative subset of Common Lisp, a language that is both efficient and convenient to use. As a result, ACL2 users often employ ACL2's programming environment to write tools.

2.1 The user's view of ACL2: A small example

The following example will provide a sense of ACL2. The first thing to notice is that the syntax is Lisp's prefix syntax, so for example we write `(+ 3 4)` and `(len x)` rather than more traditional notation such as `3+4` and `len(x)`, respectively. The syntax is case-insensitive.

Suppose that we want to prove that the length does not change when we reverse a list. Lists and some list-processing functions, including `reverse` and `len` for reverse and length of a list, are built into ACL2. So let us submit a theorem named `len-reverse`, stating that for any list `x`, the length of the reverse of `x` is equal to the length of `x`:[2]

```
ACL2 !>(defthm len-reverse
         (implies (true-listp x)
                  (equal (len (reverse x)) (len x))))

ACL2 Warning [Non-rec] in ( DEFTHM LEN-REVERSE ...):  A :REWRITE rule
generated from LEN-REVERSE will be triggered only by terms containing
the non-recursive function symbol REVERSE.  Unless this function is
disabled, this rule is unlikely ever to be used.
```

[2]The warning below is not too important here. Think of it as a reminder that after the proof is complete, we should *disable* the definition of `reverse` so that the equality can be used as a left-to-right rewrite rule by ACL2's inside-out rewriter. If `reverse` were left enabled, the rewriter would first replace a term of the form `(len (reverse x))` by expanding the definition of `reverse`, after which that term would no longer match `(len (reverse x))`.

```
This simplifies, using the :definition REVERSE, to

Goal'
(IMPLIES (TRUE-LISTP X)
         (EQUAL (LEN (REVAPPEND X NIL))
                (LEN X))).

Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes are
suggested by this conjecture.  Subsumption reduces that number to two.
These merge into one derived induction scheme.
```

The proof ultimately fails. But since `Goal'` above did not simplify, we will take a look at it in a moment. As an aside, notice that "`using the :definition REVERSE`" the prover has replaced the original call of `reverse` with a call of `revappend`, which makes sense if we use a "print event" utility:

```
ACL2 !>:pe reverse
 V     -477  (DEFUN REVERSE (X)
                    "Documentation available via :doc"
                    (DECLARE (XARGS :GUARD (OR (TRUE-LISTP X) (STRINGP X))))
                    (COND ((STRINGP X)
                           (COERCE (REVAPPEND (COERCE X 'LIST) NIL)
                                   'STRING))
                          (T (REVAPPEND X NIL))))
```

Of course, if `reverse` hadn't been built in, we could have defined it exactly as shown above, using the `defun` command.

Looking again at `Goal'`, we realize that a suitable rewrite rule could simplify the term `(LEN (REVAPPEND X NIL))`. First let us look at the definition of `revappend`, this time using a "print formula" query.

```
ACL2 !>:pf revappend
(EQUAL (REVAPPEND X Y)
       (IF (CONSP X)
           (REVAPPEND (CDR X) (CONS (CAR X) Y))
           Y))
ACL2 !>
```

Thus, if we don't know it already, we now see that `(revappend x y)` pushes successive elements of `x` onto `y`. The following lemma, aptly named `len-revappend`, says that the length of a `revappend` call is the sum of the lengths of the arguments. ACL2 proves this by induction automatically (in less than 1/100 second).[3]

[3]The reader may notice that the lemma `len-revappend` is about `(revappend x y)` rather than the original term, `(revappend x nil)`. We have generalized by hand to produce a lemma whose proof seems amenable to induction. In this talk we do not consider research in performing such generalizations automatically; in ACL2 as it is today, the user is responsible for such generalization, though occasionally ACL2 makes useful generalizations on the fly.

```
(defthm len-revappend
  (equal (len (revappend x y))
         (+ (len x) (len y))))
```

Theorems are stored by default as (conditional) rewrite rules; so now, any instance of `(len (revappend x y))` encountered during a proof will be replaced by the corresponding instance of `(+ (len x) (len y))`. Thus, our original theorem, `len-reverse`, is now proved automatically and immediately.

To a first approximation, it's fair to say that ACL2 users work as illustrated above. That is, they prove collections of rewrite rules, discovering missing rules by looking at output from the prover. Our intention is that users can make good progress with the system without having to understand automated reasoning structures and concepts, as might be necessary in order to program tactics in tactic-based provers or set parameters in resolution-based provers.

2.2 Summary of some useful ACL2 features

As we wait for our food, we'll take a very brief look at a smattering of ACL2 features that we'll see while devouring the Main Course. Occasionally these features interact in unexpected ways, leading to maintenance tasks. Many improvements in these and other features are the direct result of user requests, which are very important to the evolution of the system.

There is no intention here to be complete. ACL2 is a large system not to be explored thoroughly over the course of a meal! Our goal is just to give a sense of the kind of support provided for one empirically successful automated reasoning system.[4]

- *Release Notes* consist of brief notes alerting the experienced user to important differences between one release and the next. They make frequent citations into the documentation (below) and are not intended to be self-contained or to be read by the new user.

- *Documentation* consists of over 1000 topics organized hierarchically. The documentation source consists of strings in the ACL2 source code that are liberally sprinkled with hyperlink annotations, which is processed to create HTML, Emacs Info, and (generally not used) printed views. The documentation is extensive; for example, the HTML version of the Version 3.0 documentation is about 3.3 megabytes. Documentation for a new release is intended to be complete and accurate for that release; for example, the HTML has grown about 300 kilobytes since Version 2.9, released less than two years ago. The documentation sometimes takes about as much time to write as the code to implement a feature or change, but our impression from users is that it's worth the effort.

- *Error messages* and *warnings* provide critical feedback, often pointing to documentation topics. We put a lot of care into these!

[4] In order to save time, during the talk we'll run through these very quickly, just to give a sense of what is coming.

- *Macros* make it easy to extend the syntax, but they have limitations (addressed by a new feature, `make-event`).
- A *book* is a collection of legal *embedded event forms* (*events*), in particular definitions (`defun` events) and theorems (`defthm` events), that have been *admitted*: syntax has been checked, theorems have been proved, and termination has been proved for recursive definitions.
  - *Certification* of a book creates a *certificate* witnessing the successful processing of the book.
  - The command `(include-book "foo")` will load events from `foo.lisp` into the current session.
  - However, `local` events, e.g., `(local (defun foo ...))`, are not exported by `include-book`. A logical story [KM01] involving conservativity justifies the dropping of `local` events.
  - About 850 books in about 70 directories, mostly contributed by users rather than the developers, are distributed with ACL2, with over 700 more in over 200 directories available from supporting materials for the first five ACL2 workshops (not including the one this year, 2006). Thus there are over 1500 books in our regression suite. We rely heavily on that test suite to test purported improvements to the prover's heuristics.
- Like books, `encapsulate` provides a modular structuring mechanism. `Encapsulate` events can be used to provide partial definitions for functions: that is, functions are total but may have incomplete axiomatizations.
- The `defevaluator` macro generates events that define an *evaluator*, against which one can prove *meta-rules* [BM81, HKK+05] that, in essence, augment the simplifier with formally verified user-defined functions.
- *Proof control* includes `in-theory` events and *hints*, which *disable* (turn off) or *enable* (turn on) specified rules. Supported are not only `in-theory` hints but others, for example directing induction, function expansion, or the use of previously-proved theorems. These can be attached to specific named goals or can be generated by code ("computed hints"), which can be specified globally ("default hints").
- *Database control* includes undo and undo-the-undo (`oops`) commands.
- An interactive *proof-checker* loop is a goal manager that has the feel of tactic-based prover interfaces, allowing a range of commands, from individual rewrites to calls of the full prover.
- *Proof debug* is supported by the above-mentioned proof-checker and also by a utility for inspecting apparent rewriter loops, a `break-rewrite` debugger for the rewriter, and *proof-tree* displays for navigating proof output.
- A *top-level read-eval-print loop* allows for interactive testing of one's functions. Such testing is typically relatively slow unless one issues a compilation command.

- Efficient *execution* is supported for ground terms not only in the top-level loop, but also during proofs. Efficient execution also relies on single-threaded objects [BM02], or *stobjs*, including the ACL2 `state` objects.
- *Guards* provide a powerful, flexible analogue of types, and help support efficient execution by way of a connection to the underlying Common Lisp. The `mbe` ("must be equal") feature allows one to attach efficient code to logically elegant functions [GKM$^+$].
- Lisp *packages* provide namespaces.
- While the main *proof technique* is conditional rewriting, there are certainly others (for example, integrated decision procedures for ground equality and linear arithmetic). And, rewriting is actually congruence-based, i.e., can be used to replace a term with one that is suitably equivalent even if not actually equal.
- A *functional instantiation* utility [BGKM91, KM01] allows deriving a theorem $\varphi(g)$ from a corresponding theorem $\varphi(f)$ provided the function g satisfies all constraints on the function f.

3 Main Course — A selection of recent enhancements to ACL2

We now present a selection of items from recent ACL2 release notes, annotated with explanations and discussion about implications for system maintenance. We introduce each item very briefly, then display the Emacs Info version of the relevant release note, and finally explain the issues if necessary.

3.1 Subgoal counting

This item illustrates the effort we put into prover output. Here, "`:functional-instance`" refers to ACL2's functional instantiation utility, mentioned above; but the main point here is about output format, not functional instantiation.

```
. . . . . .
  Fixed a bug that was causing proof output on behalf of
  :functional-instance to be confusing, because it failed to mention that
  the number of constraints may be different from the number of subgoals
  generated.  Thanks to Robert Krug for pointing out this confusing
  output.  The fix also causes the reporting of rules used when silently
  simplifying the constraints to create the subgoals.
. . . . . .
```

Here is output from a proof attempt using ACL2 Version 2.9.3 that illustrates the problem. Notice that "six constraints" doesn't match up with the subgoal numbering, which counts down from 5 to 1. (We count down to give the user a real-time sense of how much work remains as the output scrolls by.) The old output was confusing, and thus potentially undermined the user's confidence in his understanding of what ACL2 is doing and in his belief in ACL2's correctness.

```
We now augment the goal above by adding the hypothesis indicated by
the :USE hint.  This produces a propositional tautology.  The hypothesis
can be derived from AC-FN-LIST-REV via functional instantiation, provided
we can establish the six constraints generated.

Subgoal 5
(EQUAL (TIMES-LIST X)
       (IF (ATOM X)
           1 (* (CAR X) (TIMES-LIST (CDR X))))).

But simplification reduces this to T, using the :definitions ATOM and
TIMES-LIST and primitive type reasoning.

Subgoal 4
....
```

Here is the corresponding output (suitably elided) from ACL2 3.0.

```
We now augment .... provided
we can establish the six constraints generated.  By the simple :rewrite
rules ASSOCIATIVITY-OF-* and UNICITY-OF-1 we reduce the six constraints
to five subgoals.
```

[. . . and so on, as before]

3.2 A rough edge in theory control

ACL2 uses evaluation as part of its proof strategy, but it allows the user to disable evaluation of calls of a function `f` by disabling the so-called *executable-counterpart* rule for `f`. For a particular type of conditional rule, a *forward-chaining* rule, evaluation of ground hypotheses had taken place without regard to which executable-counterpart rules are disabled, thus severely impacting efficiency in at least one user's experience.

```
.....................................................................
  Fixed a long-standing bug in forward-chaining, where variable-free
  hypotheses were being evaluated even if the executable-counterparts of
  their function symbols had been disabled.  Thanks to Eric Smith for
  bringing this bug to our attention by sending a simple example that
  exhibited the problem.
.....................................................................
```

3.3 Prover heuristic tweaks

Sometimes we find improvements to ACL2's prover heuristics. All three items below describe changes that were carefully made in response to user feedback, and tested with our regression suite to gain confidence that our heuristic changes would not severely impact users. These changes are only necessary because ACL2 attempts to provide significant automation.

```
.....................................................................
  We fixed an infinite loop that could occur during destructor elimination
```

```
  (see *Note ELIM::).  Thanks to Sol Swords for bringing this to our
  attention and sending a nice example, and to Doug Harper for sending a
  second example that we also found useful.
.....................................................................
  The simplifier has been changed slightly in order to avoid using
  forward-chaining facts derived from a literal (essentially, a top-level
  hypothesis or conclusion) that has been rewritten.  As a practical
  matter, this may mean that the user should not expect forward-chaining
  to take place on a term that can be rewritten for any reason (generally
  function expansion or application of rewrite rules).  Formerly, the
  restriction was less severe: forward-chaining facts from a hypothesis
  could be used as long as the hypothesis was not rewritten to t.  Thanks
  to Art Flatau for providing an example that led us to make this change;
  see the comments in source function rewrite-clause for details.
.....................................................................
  We modified the rewriter to avoid certain infinite loops caused by an
  interaction of the opening of recursive functions with equality
  reasoning.  (This change is documented in detail in the source code, in
  particular functions rewrite-fncall and fnstack-term-member.)  Thanks to
  Fares Fraij for sending us an example that led us to make this change.
.....................................................................
```

There are over 36,000 lines of comments in the source code, some of which survived multiple translations from the earliest version of the Boyer-Moore system. The comments are largely intended to be a record, for the implementors, of why things are the way they are. This is important in a software project of 35 years duration. Sometimes the comments show how we used to do something and why and when we changed it. The comments also sometimes contain interesting examples and counterexamples illustrating supposed properties of the code. Despite the original intention of the implementors to use comments as a way of recording the design decisions and history, many ACL2 users read the source code. Since ACL2 is written in ACL2, this is straightforward and sort of represents a second, more detailed, level of documentation.

3.4 A library improvement using `MBE`

The following release note item illustrates one maintenance aspect: we update the distributed books (libraries of definitions and proved theorems), often in consultation with users.

```
.....................................................................
  Several interesting new definitions and lemmas have been added to the
  rtl library developed at AMD, and incorporated into books/rtl/rel4/lib/.
  Other book changes include a change to lemma truncate-rem-elim in
  books/ihs/quotient-remainder-lemmas.lisp, as suggested by Jared Davis.
.....................................................................
```

But buried in this item is a change that we find particularly interesting.

We mentioned *guards* earlier as a flexible analogue of types, and we mentioned `mbe` as a way to attach executable counterparts efficiently.

At AMD, we found a need for more efficient execution of bit-vector operations. Through Version 2.9.1, the *rtl library*, `books/rtl/rel4/lib/`, contained the following definition of the bit-slice operation that returns bits `i` down to `j` of a natural number `x`. (Here, `defund` is a define-then-disable command, implemented in response to a user's request.)

```
(defund bits (x i j)
  (declare (xargs :guard (rationalp x)))
  (if (or (not (integerp i))
          (not (integerp j)))
      0
    (fl (/ (mod x (expt 2 (1+ i))) (expt 2 j)))))
```

However, we found this definition in terms of floor, modulo, and exponentiation operations painfully slow to execute. We really wanted a definition that uses bitwise-and and shift operations instead:

```
(defund bits (x i j)
  (if (< i j)
      0
    (logand (ash x (- j)) (1- (ash 1 (1+ (- i j)))))))
```

Fortunately, we were able to change the definition of `bits` for purposes of execution without changing its logical definition, which saved us from having to rework our proofs of any lemmas! The `mbe` ("must be equal") call below says to use the form after `:logic` as the body, with a proof obligation that the `:guard` (that `x`, `i`, and `j` are natural numbers) implies the equality of the `:logic` and `:exec` forms. The `:guard` must also imply certain formulas generated for the calls in the `:exec` form; for example the call `(ash x (- j))` carries a guard-related obligation that `x` and `(- j)` be integers, which is trivial from the guard assumptions that `x` and `j` are natural numbers. Then calls of `bits` on natural numbers will be executed directly in Common Lisp using the `:exec` form as the definition.

```
(defund bits (x i j)
  (declare (xargs :guard (and (natp x)
                              (natp i)
                              (natp j))))
  (mbe :logic (if (or (not (integerp i))
                      (not (integerp j)))
                  0
                (fl (/ (mod x (expt 2 (1+ i))) (expt 2 j))))
       :exec  (if (< i j)
                  0
                (logand (ash x (- j)) (1- (ash 1 (1+ (- i j))))))))
```

3.5 Some convenience features

The following three items all make life easier for the user, as we explain below each one.

```
.....................................................................
  Improved cw-gstack to allow a :frames argument to specify a range of one
  or more frames to be printed.  See *Note CW-GSTACK::.
.....................................................................
```

ACL2 makes very few restrictions on how users introduce rewrite rules to program the rewriter. This freedom, however, makes it possible to introduce infinite loops. When that occurs, ACL2 aborts cleanly (a major advance starting with Version 2.8 — previously it sometimes seg faulted!) and suggests use of the tool `cw-gstack`, which shows the rewrite stack. Unfortunately, the entire rewrite stack is large, so there was interest in being able to limit the number of frames printed.

```
.....................................................................
  A new event, set-enforce-redundancy, enforces a restriction that all
  defthms, defuns, and most other events are redundant.  See *Note
  SET-ENFORCE-REDUNDANCY::.
.....................................................................
```

AMD's rtl library (mentioned above) employed a methodology in which the proof work was restricted to books in an auxiliary directory. It seemed desirable to enforce this methodology, so that the main directory was kept clean and the auxiliary directory could be modified as desired. Here is how that works.

Suppose we have a file `top.lisp` that we want to certify as a book.

```
(local (include-book "work/book-1"))
(defthm result-1 ...)
...
```

Here, imagine that `result-1` is proved in file `work/book-1.lisp`. The `local` annotation guarantees that additional theorems proved in `work/book-1.lisp` will ultimately disappear, except for `result-1`, which (as seen above) we have made explicit. When we are certifying the present book, we expect that `result-1` will be *redundant* because it already appears in `work/book-1.lisp`. But suppose we accidentally delete `result-1` in `work/book-1.lisp`. ACL2 would then try to prove `result-1`, but we may prefer that ACL2 instead fail immediately with a clear complaint that it didn't find that `result-1` has already been proved.

The item above provides a solution. We simply start `top.lisp` with the form `(set-enforce-redundancy t)`.

One thing we've found is that nothing is ever simple! So for example, certain kinds of events called `deflabel` events are not allowed to be redundant. So even with `set-enforce-redundancy`, we need to allow non-redundant `deflabel` events.

```
.....................................................................
  The function disabledp can now be given a macro name that has a
  corresponding function; see *Note MACRO-ALIASES-TABLE::.  Also,
  disabledp now has a guard of t but causes a hard error on an
  inappropriate argument.
.....................................................................
```

For example, in ACL2 `append` is a macro, because functions must take a fixed number of arguments but we want to be able to apply `append` to an arbitrary number of arguments. We can see how this works by using ACL2's `:trans1` command to perform a single-step macroexpansion.

```
ACL2 !>:trans1 (append x y z)
 (BINARY-APPEND X (BINARY-APPEND Y Z))
ACL2 !>
```

ACL2, however, is kind enough to print terms using `append` rather than using the corresponding function, `binary-append`. Thus, novice users might not even realize that `append` is not a function.

ACL2 has a notion of *table* events that allows maintenance of information of interest, and once such table associates `append` with `binary-append`. The user then may refer to `append` in contexts where a function symbol is expected, for example when disabling a definition, for example:

```
(in-theory (disable append))

.....................................................................
  The macro comp is now an event, so it may be placed in books.
.....................................................................
```

The above item simply allows a compilation directive to be placed in books. It's a simple thing to provide and we wish we had done it sooner in order to save users some annoyance!

3.6 Common Lisp compatibility: Packages

Namespace control is provided by Common Lisp *packages*. Each *symbol* is in essence a pair of strings: a package name and a symbol name. But getting this exactly right is quite tricky. A rather elaborate fix was made in Version 2.8, not shown here, to deal with an unsoundness that could result from a subtle use, carefully employing `local`, of two different packages with the same name. (See ACL2's documentation topic "hidden-death-package" if you want to learn more about this issue. And it points to a very elaborate comment in the source code that gives even more of an idea of how nasty this issue really is.)

Below are three package issues solved more recently than that one, and not nearly as complex. They show how we sometimes need to work hard to ensure compatibility with the host Common Lisp.

```
.....................................................................
  We fixed a soundness hole due to the fact that the "LISP" package does
  not exist in OpenMCL.  We now explicitly disallow this package name as
  an argument to defpkg.  Thanks to Bob Boyer and Warren Hunt for bringing
  an issue to our attention that led to this fix.
.....................................................................
  ACL2 now requires all package names to consist of standard characters
  (see *Note STANDARD-CHAR-P::, none of which is lower case.  The reason
```

```
  is that we have seen at least one lisp implementation that does not
  handle lower case package names correctly.  Consider for example the
  following raw lisp log (some newlines omitted).

       >(make-package "foo")
       #<"foo" package>
       >(package-name (symbol-package 'FOO::A))
       "foo"
       >(package-name (symbol-package '|FOO|::A))
       "foo"
       >
.....................................................................
  (GCL only) A bug in symbol-package-name has been fixed that could be
  exploited to prove nil, and hence is a soundness bug.  Thanks to Dave
  Greve for sending us an example of a problem with defcong (see below)
  that led us to this discovery.
.....................................................................
```

3.7 Portability, and help from others

ACL2 can be built on most (all?) stable Common Lisp implementations, including GCL, OpenMCL, Allegro CL, SBCL, CMUCL, CLISP, and Lispworks. The most recent addition is SBCL. There are at least two reasons for porting to all of these Lisps, in spite of a certain amount of low-level Lisp-specific code we need to write and maintain. One is that we sometimes find bugs in our code that are in some sense "forgiven" by most, but not all, Lisps. The other is that we want users to be able to build on whatever Lisp platform they happen to have. Perhaps a third reason is to support each Lisp's development by providing a non-trivial test suite.

```
.....................................................................
  Added SBCL support.  Thanks to Juho Snellman for significant assistance
  with the port.  Thanks to Bob Boyer for suggesting the use of feature
  :acl2-mv-as-values with SBCL, which can allow thread-level parallelism
  in the underlying lisp; we have done so when feature :sb-thread is
  present.
.....................................................................
```

3.8 User-level debug support

ACL2 has a *break-rewrite* utility that allows the user to put a breakpoint upon the application of a specified rewrite rule, optionally under specified conditions. The situation becomes complicated when there are so-called *free variables in hypotheses*. For example, consider the conditional rewrite rule saying that if predicate `p2` holds of `x` and `y`, and predicate `p3` holds of `y`, then predicate `p1` holds of `x`:

```
(implies (and (p2 x y)
              (p3 y))
         (equal (p1 x) t))
```

Now suppose the rewriter encounters the term `(p1 (foo a))`. So, `x` is bound to `(foo a)` when we apply the above rule. But how can we rewrite the first hypothesis (to *true*) if we do not have a binding for the *free variable* `y`?

In this case, ACL2 simply looks in its current context for some term α for which `(p1 (foo a)` α`)` is known to be true. When it finds such an α then it binds `y` to α and goes on to the next hypothesis. So it will now be "thinking about" `(p3` α`)`. If the rewriter cannot prove this is true, it will backtrack and look for another value of `y` in place of α for the first hypothesis.

The above information can be critical to a user who is trying to understand why a rule is failing to be applied, especially when there is a complex set of available rules operating on the hypotheses. The following item describes an improvement that provides convenient display of such information.

```
.....................................................................
  Improved reporting by the break-rewrite utility upon failure to relieve
  hypotheses in the presence of free variables, so that information is
  shown about the attempting bindings.  See *Note
  FREE-VARIABLES-EXAMPLES-REWRITE::.  Thanks to Eric Smith for requesting
  this improvement.  Also improved the break-rewrite loop so that terms,
  in particular from unifying substitutions, are printed without hiding
  subterms by default.  The user can control such hiding ("evisceration");
  see *Note SET-BRR-TERM-EVISC-TUPLE::.
.....................................................................
```

The ACL2 documentation topic "`free-variables-examples-rewrite`" describes how all this works. We'll just show a piece of that documentation here in order to give a visual cue of what we provide.

```
(1 Breaking (:REWRITE LEMMA-1) on (PROP U0):
1 ACL2 >:eval

1x (:REWRITE LEMMA-1) failed because :HYP 1 contains free variables.
The following display summarizes the attempts to relieve hypotheses
by binding free variables; see :DOC free-variables and see :DOC set-
brr-term-evisc-tuple.

    [1] X : X1
Failed because :HYP 3 contains free variables Y and Z, for which no
suitable bindings were found.
    [1] X : X2
Failed because :HYP 2 rewrote to (BAD X2).
    [1] X : X3
        [3] Z : Z1
            Y : Y1
Failed because :HYP 6 rewrote to (FOO X3 Y1).
        [3] Z : Z1
            Y : Y3
Failed because :HYP 6 rewrote to (FOO X3 Y3).

1 ACL2 >
```

3.9 Some other release note items of interest

- Several bugs have been fixed that are related to `local`. It seems somewhat difficult to anticipate all interactions of other aspects of the system and logic with `local`.

- Two very different kinds of hints for `defthm` events are generally incompatible: `:hints` to direct the automatic prover, and `:instructions` to direct the replay of commands saved during a session with the *proof-checker*, an interactive goal-directed proof management tool. We quite sensibly caused an error if both `:hints` and `:instructions` were present for the same `defthm` event. But we added a notion of default hints without noticing that we needed to allow them with `:instructions`, in which case the default hints should apply to any individual instruction that calls the full prover. (This has been fixed.)

- Users can undo events and they can even undo the undo. But some heavy users are hitting memory limitations, so we now provide the option of trading the "undo the undo" capability with the reclamation of space.

- A feature new to Version 3.0, of excitement to some experienced ACL2 users, is a capability, `make-event`, that is similar to macros but which is sensitive to the environment (e.g., the ACL2 `state` object). The main idea is that expansions that might otherwise depend on the environment, which is illegal for macros,[5] are saved in the book's certificate. But there were lots of complications to solve (for example, what if the `make-event` is submitted interactively before certification is begun).

- Users can specify a limit on backchaining through rewrite rules, and they can specify syntactic checks to control the application of a rewrite rule [HKK+05]. But until a user requested it, these features were not available with conditional meta-rules.

- ACL2 supports rewriting with congruences, where the original and rewritten term are equivalent but not necessarily equal. ACL2 also caches rewrite results, for efficiency. There are occasions when the cached result is from an equality rewrite, but we need to rewrite with an equivalence, which could produce a stronger result. If we always ignore the cache in such cases, efficiency becomes a problem. But after receiving a user request, we instituted a compromise where we give special handling in some cases when the equivalence relation is Boolean equivalence. More recently [KM06], we have provided the user a means to handle this situation for other equivalence relations, together with warnings that bring this situation to the user's attention.

[5] It would take us too far afield to explain in detail why it is illegal for macros to depend on the current state. But it's not hard to imagine that otherwise, a macro might expand to give one definition of a function as a book is certified, but a different definition of the same function when the book is later included. Besides, ACL2 compiles its books, and the Common Lisp specification disallows dependence of macros on the state.

4 Dessert

I intend to leave time for audience members to share related observations from their own experiences, and to ask further questions.

Acknowledgments

We thank Robert Krug and Sandip Ray for useful comments on a draft of this paper. This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591.

References

[BGKM91] R.S. Boyer, D.M. Goldschlag, M. Kaufmann, and J S. Moore. Functional instantiation in first-order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science.* Academic Press, London, 1981.

[BM97] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition.* Academic Press, New York, 1997.

[BM02] R. S. Boyer and J S. Moore. Single-threaded objects in ACL2. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *PADL 2002, LNCS 2257*, pages 9–27, 2002.

[GKM+] D. A. Greve, M. Kaufmann, P. Manolios, J S. Moore, S. Ray, J. L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient execution in an automated reasoning environment. Submitted.

[gpl] `http://www.gnu.org/copyleft/gpl.html`.

[HKK+05] W. A. Hunt, Jr, M. Kaufmann, R. B. Krug, J S. Moore, and E. W. Smith. Meta reasoning in ACL2. In Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*. Springer, August 2005.

[Kau91] M. Kaufmann. An informal discussion of issues in mechanically-assisted reasoning. In M. Archer, J. J. Joyce, K. N. Levitt, and P. H. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 318–337, Los Alamitos, CA, 1991. IEEE Computer Society Press.

[KM01] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[KM04a] M. Kaufmann and J S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, 2004.

[KM04b] M. Kaufmann and J S. Moore. How to prove theorems formally. In *http://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms/*. Department of Computer Sciences, University of Texas at Austin, 2004.

[KM06] M. Kaufmann and J S. Moore. Double rewriting for equivalential reasoning in ACL2. In *Proceedings of ACL2 Workshop 2006*, August 2006.

[KMM00a] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

SPASS+T

Virgile Prevosto[1]* and Uwe Waldmann[2]

[1]CEA – LIST, Centre de Saclay
Software Reliability Laboratory (LSL)
91191 Gif-sur-Yvette Cedex, France
virgile.prevosto@m4x.org

[2]Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
uwe@mpi-inf.mpg.de

Abstract

SPASS+T is an extension of the superposition-based theorem prover SPASS that allows us to enlarge the reasoning capabilities of SPASS using an arbitrary SMT procedure for arithmetic and free function symbols as a black-box. We discuss the architecture of SPASS+T and the capabilities, limitations, and applications of such a combination.

1 Introduction

SPASS (Weidenbach et al. [10]) is a saturation-based theorem prover for first-order logic based on the superposition calculus. Such provers are notoriously bad at dealing with integer or real arithmetic – encoding numbers in binary or unary is not really a viable solution in most application contexts. We have extended SPASS in such a way that we can link it to a rather arbitrary SMT (*Satisfiability Modulo Theories*) procedure for arithmetic and free function symbols, for instance our own MODUPROVE system (based on the DPLL(T) framework of Ganzinger et al. [5]) or the CVC Lite prover by Barrett and Berezin [2]. Briefly, the resulting system, called SPASS+T, works as follows: SPASS uses its deduction rules to generate formulas as usual. In addition, it passes to the SMT procedure all the formulas that can be handled by the procedure, i. e., all ground formulas. As soon as one of the two systems encounters a contradiction, the problem is solved.

The scenario described so far is a special case of hierarchic theorem proving: a combination of a *base prover* that deals with some subclass of formulas, say, formulas in linear arithmetic, and an *extension prover* that deals with formulas over the complete signature

*This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors. Project website: http://www.verisoft.de.

and passes all formulas to the base prover that the latter can handle. The correctness of such a hierarchic combination of deductive systems is obvious. In theory, conditions for completeness have been given by Bachmair, Ganzinger, Sofronie-Stokkermans, and Waldmann [1, 6]. For instance, one can get a complete hierarchic combination of provers if base and non-base vocabulary are separated using abstraction, the term ordering is chosen in such a way that base terms/atoms are smaller than non-base terms/atoms, the base prover can deal with arbitrary formulas over the base vocabulary, the extension is sufficiently complete,[1] and the base theory is compact [1].

In practice, however, sufficient completeness need not hold (and cannot be checked automatically), abstraction enlarges the search space, SMT procedures for some useful theories can only deal with ground formulas (this is for instance the case for Nelson-Oppen combinations of arithmetic and free function symbols), and even compactness may be an issue. As an example, consider the two clauses $\forall x, y.\, (f(x) \neq y + y)$ and $\forall x, y.\, (f(x) \neq y + y + 1)$. If the base theory is Presburger arithmetic (linear integer arithmetic), then the conjunction of these two clauses (expressing the fact that $f(x)$ is neither even nor odd) is inconsistent. Still any finite set of ground instances of these two formulas is consistent. Even unification modulo the base theory would not help.

An integration of first-order theorem provers and SMT procedures can therefore only be a pragmatic one: Completeness can be achieved for special classes of inputs, but not in general. If we work with non-ground problem description but ground goal formulas, then there is some hope that we can ultimately produce sufficiently many ground formulas so that the SMT procedure can find the contradiction. If we want to solve non-ground problems, in particular if we want to find solutions for variables, then the purely hierarchic approach must be supplemented by some knowledge about arithmetic that is built-in directly into SPASS.

2 SPASS

SPASS (Weidenbach et al. [10]) is known to be one of the most advanced implementations of the superposition calculus. The superposition calculus is a refutationally complete procedure for arbitrary first-order clauses with equality, that is, it provides a semi-decision procedure for the unsatisfiability of sets of clauses. Theorem proving methods such as resolution or superposition aim at deducing a contradiction from a set of formulas by recursively inferring new formulas from given ones. New formulas are derived according to a set of inference rules. In the case of superposition, these rules are restricted versions of paramodulation, resolution, and factoring, parameterized by a reduction ordering $\succ$ that is total on ground expressions (that is, ground atoms and ground terms) and by a *selection function* S, which assigns to each clause a (possibly empty) multiset of (occurrences of) *negative* literals. The literals in $S(C)$ are called *selected*. Selected literals, besides being negative, can be arbitrarily chosen. Ordering and selection function impose various restrictions on the possible inferences, which are crucial for the efficiency of theorem provers like SPASS. Let us consider one of the inference rules of the superposition calculus as an example:

[1] Intuitively, sufficient completeness means that every ground term of a base sort is provably equal to a ground term consisting only of base symbols.

Negative superposition $\dfrac{D' \vee t = t' \qquad C' \vee \neg\, s[u] = s'}{(D' \vee C' \vee \neg\, s[t'] = s')\sigma}$

if (i) the literal $t = t'$ is strictly maximal in the first premise, (ii) no literal is selected in the first premise, (iii) either the literal $\neg\, s[u] = s'$ is selected in the second premise or it is maximal and no literal is selected, (iv) $t \not\preceq t'$, (v) $s[u] \not\preceq s'$, (vi) u is not a variable, and (vii) σ is a most general unifier of t and u.

This inference rule combines the unification of t and the subterm u of s with subsequent replacement of $u\sigma$ by $t'\sigma$. When we speak of a *superposition inference* we mean an arbitrary rule of the calculus.

Note that a clause with selected literals cannot serve as the left premise of a superposition inference, and that maximal terms are superposed either on maximal or selected literals, and a smaller term is never replaced by a larger one. Moreover, only the maximal sides of equations are replaced. The pattern of interplay between ordering restrictions and the selection function is the same for all inference rules of the calculus.

The local restrictions of the superposition inference rules are supplemented by a global redundancy criterion that allows us to discard formulas that are provably unnecessary for deriving a contradiction.

3 System Architecture

SPASS+T consists of three modules: the theorem prover SPASS, a (rather arbitrary) SMT procedure for integer or real arithmetic and free function symbols, and a module SMT-Control connecting both. The proof systems are only loosely coupled; they wait for each other only if they have completed their own task.

SPASS sends three kinds of messages to SMT-Control:

- all the ground formulas present in the input or derived during the saturation;
- *proof_found* (signaling that SPASS has derived the empty clause);
- *end_of_file* (signaling that SPASS has saturated its input).

SMT-Control collects the formulas sent by SPASS and repeatedly sends portions of the set of formulas to the SMT procedure,[2] until one of the following events occurs:

- SPASS sends *proof_found*;
- the SMT procedure detects unsatisfiability;
- the SMT procedure detects satisfiability after SPASS has sent *end_of_file* and the SMT procedure has seen all the output of SPASS.

The result is "proof found" in the first two cases, "no proof found" in the third one. Lacking completeness, "no proof found" does in general not imply satisfiability.

[2] Under the assumption that the SMT procedure works incrementally. Alternatively the SMT procedure can be restarted with increasing subsets of the set of formulas.

4 The Simple Case

There are applications where we can guarantee that every base theory formula generated by SPASS is ground. In this case, the very simple setup described above is already sufficient. An example is pointer data structure verification à la McPeak and Necula [7]. In this scenario, we consider recursive data structures involving pointers to records or to *nil*. Record fields can either be scalar values or pointers. In the axioms used to describe the behaviour of a data structure, all variables range over pointer values (i. e., there is no quantification over scalar values). Record fields are encoded as (partial) functions, so $x.data$, i. e., the *data* field of the record that x points to, is written as $data(x)$. McPeak and Necula require that every occurrence of such a function must be guarded by a condition that ensures that the argument is non-nil. For instance, the formula stating that the *prev* pointer is the inverse of the *next* pointer in a doubly linked list looks like

$$\forall x.\, (x \neq nil \wedge next(x) \neq nil \rightarrow prev(next(x)) = x),$$

and the formula stating that the *data* field of any record is positive looks like

$$\forall x.\, (x \neq nil \rightarrow data(x) > 0).$$

To avoid positive disjunctions, we replace the guard formula $x \neq nil$ by $isrecord(x)$, obtaining

$$\forall x.\, (isrecord(x) \wedge isrecord(next(x)) \rightarrow prev(next(x)) = x),$$
$$\forall x.\, (isrecord(x) \rightarrow data(x) > 0).$$

Since the SMT procedure accepts not only the symbols of the arithmetical theory, such as $+, -, \cdot, <$, or $\leq$, but also free function or predicate symbols, the distinction between base and non-base symbols is somewhat arbitrary – if it is useful, we may treat both theory symbols and a subset of the free symbols as base symbols. Let us therefore consider the guard predicate *isrecord* as the only non-base symbol in the signature. If we *select* the occurrences of $isrecord(t)$ in the antecedent, we ensure that the only inferences that are possible with such clauses are (repeated) resolution steps with positive occurrences of $isrecord(s)$. Since $isrecord(s)$ occurs positively only in goal formulas (or formulas recursively derived from goal formulas), and since the term s is always ground in these formulas, the base formulas resulting from resolution are ground and can be passed to the SMT procedure. In effect, SPASS+T mimics the derivation steps of McPeak and Necula, and the completeness result of McPeak and Necula carries over to SPASS+T.

5 Theory Instantiation

If non-ground theory literals are not guarded, then it can easily happen that the usual inference rules of SPASS do not produce those ground instances that the SMT procedure would need to find a contradiction. Consider the following example from Boyer and Moore [3]: Let *min* and *max* be non-theory function symbols denoting the minimum and the maximum element of a list of numbers. Suppose that we want to refute $\neg\, (l < max(a) + k)$ using the assumptions $l \leq min(a)$ and $0 < k$ and the universally quantified

lemma $\forall x.\,(min(x) \leq max(x))$. Analogously to Boyer and Moore, we need an additional inference rule that computes the required ground instance of this lemma. For efficiency reasons, we restrict ourselves to instantiations where a term headed by a non-theory symbol occurs in at least one other ground clause – if a non-theory term occurs in only one clause, this clause is very unlikely to be useful to the SMT procedure, at least in linear arithmetic. We obtain the following inference rule:

Theory instantiation $\dfrac{C[s] \qquad L[t] \vee D}{(L[t] \vee D)\sigma}$

> if $L[t] \vee D$ is not ground, t is headed by a non-theory function symbol and occurs in a maximal literal $L[t]$ immediately below a theory symbol, all function or predicate symbols occurring above t in $L[t]$ are theory symbols or equality, $C[s]$ is ground, σ is an mgu of s and t, and $(L[t] \vee D)\sigma$ is ground.

In the example above, the *theory instantiation* inference

$$\frac{l \leq min(a) \qquad min(x) \leq max(x)}{min(a) \leq max(a)}$$

yields the ground clause that the SMT procedure needs to derive a contradiction. There is one problem with this rule, though: The generated formula $(L[t] \vee D)\sigma$ is subsumed by the second premise $L[t] \vee D$; therefore SPASS will delete it again. We export it to the SMT procedure before the redundancy check, but we must also ensure that $(L[t] \vee D)\sigma$ can again be used as a first premise in a *theory instantiation* inference. To this end, we introduce a special new predicate symbol *ground* and split the inference rule into two parts:

Theory instantiation I $\dfrac{C[s]}{ground(s)}$

> if $C[s]$ is ground and s is headed by a non-theory function symbol.

Theory instantiation II $\dfrac{ground(s) \qquad L[t] \vee D}{ground(u_1) \ldots ground(u_n) \quad \mathbf{export}(D')}$

> if $L[t] \vee D$ is not ground, t is headed by a non-theory function symbol and occurs in a maximal literal $L[t]$ immediately below a theory symbol, all function or predicate symbols occurring above t in $L[t]$ are theory symbols or equality, σ is an mgu of s and t, $D' = (L[t] \vee D)\sigma$ is ground, and $u_1, \ldots, u_n$ are the ground terms occurring in D' that are different from s and headed by a non-theory function symbol.

In the example above, the *theory instantiation I* inference

$$\frac{l \leq min(a)}{ground(min(a))}$$

and the *theory instantiation II* inference

$$\frac{ground(min(a)) \qquad min(x) \leq max(x)}{ground(max(a)) \qquad \mathbf{export}(min(a) \leq max(a))}$$

cause $min(a) \leq max(a)$ to be exported to the SMT procedure and ensure that the clause $ground(max(a))$ is available for further *theory instantiation II* inferences within SPASS.

If we select guard literals whenever possible, and if we choose the atom ordering so that non-theory predicates are larger than theory predicates, then the *theory instantiation* rule is used only as a last resort: It is applied only if there are no non-theory guard literals that might produce ground instances in a more restricted manner.

6 Arithmetic Simplification

There are several reasons to supplement the external SMT procedure in SPASS+T by simplification techniques that are built-in directly into SPASS+T. First of all, such simplification techniques offer the chance to find solutions for variables, say, to replace a clause $\forall x.\,(\neg\, x + 2 = 5 \vee p(x))$ by $\forall x.\,(\neg\, x = 3 \vee p(x))$ and then by $p(3)$. Second, they allow us to reduce different numeric subexpressions to the same number, so that the search space of SPASS+T is not cluttered by different, but numerically equivalent clauses, such as $p(2+1)$, $p(1+2)$, $p(1+(1+1))$, etc. Third, by applying arithmetic simplification in advance, an equation like $\forall x.\,(x + 0 = x)$ can be used to simplify a formula such as $p(((a+2)-1)-1)$ to $p(a)$.

In SPASS+T, we use a combination of additional input axioms encoding some fragment of arithmetic and specialized simplification rules dealing with the numeric part. The latter include rules for evaluation of constant numeric subexpressions, such as

$$\frac{C[c_1 + c_2]}{C[c_0]}$$

and

$$\frac{C[(t + c_1) + c_2]}{C[t + c_0]}$$

where c_1 and c_2 are numeric constants and $c_0 = c_1 + c_2$, and rules for elementary (in-)equation solving, for instance

$$\frac{C \vee [\neg]\, t + c_1 \sim c_2}{C \vee [\neg]\, t \sim c_0}$$

where $c_0 = c_2 - c_1$ and $\sim$ is equality or an ordering relation.

All the arithmetic simplification rules we have implemented have the property that the resulting formula is smaller than the original one in any Knuth-Bendix ordering, provided that all constants have the same weight. Still it should be clear that there is some risk of losing proofs by applying arithmetic simplification. As a trivial example consider the two clauses $\forall x.\,(p(3 \cdot x + 4))$ and $\neg\, p(3 \cdot 5 + 4)$. These clauses are obviously contradictory, but lacking theory unification, there is no way to derive a contradiction as soon as the second clause has been simplified to $\neg\, p(19)$.

In contrast to formulas that describe the relationships between concrete numerical constants, equations such as $\forall x.\,(x + 0 = x)$, $\forall x.\,(x - x = 0)$, or $\forall x, y.\,((x - y) + y = x)$ are preferably added to the input as ordinary axioms. In this way they are not only available for simplification, but also for superposition or resolution inferences. In our experiments, this capability turned out to be extremely useful for refuting non-ground queries.

The *integer ordering expansion* rule, which is essential for some kinds of inductive proofs, is in some way a hybrid between the two cases above.

Integer ordering expansion $$\frac{C \vee s \leq t}{C \vee s = t \vee s \leq t - 1 \qquad C \vee s = t \vee s + 1 \leq t}$$

It is an inference rule, rather than a simplification rule, and it corresponds to resolution inferences with the axioms $\forall x, y.\, (\neg\, x \leq y \vee x = y \vee x \leq y - 1)$ and $\forall x, y.\, (\neg\, x \leq y \vee x = y \vee x + 1 \leq y)$. These are *unordered* resolution inferences, however, so SPASS would not perform them. The *integer ordering expansion* rule fills this gap, but it has to be restricted, for instance by limiting applications to clauses with only one positive literal; otherwise it turns out to be too productive.

7 Experiments

We have tested SPASS+T both on a list of sample conjectures [8] (theorems and non-theorems) from the TPTP library [9] and on our own list of benchmark problems, mostly combining arithmetic and data structures. We used SPASS+T with CVC Lite 2.5 as external decision procedure for the union of the theory of uninterpreted functions and integer arithmetic. SPASS+T options were chosen identically for all benchmark problems; in particular, arithmetic simplification, theory instantiation, and (a very restricted form of) integer ordering expansion were switched on, and precedences were chosen uniformly in such a way that non-theory predicates (notably the containment relation for collections) became larger than theory predicates.[3] Moreover, the following set of auxiliary axioms was supplied:

INT01 $\forall X, Y.\, ((X - Y) + Y = X)$.
INT02 $\forall X.\, (X + 0 = X)$.
INT03 $\forall X.\, (0 + X = X)$.
INT04 $\forall X.\, (X < X + 1)$.
INT05 $\forall X, Y.\, (X < Y \rightarrow X \leq Y)$.
INT06 $\forall X.\, (X \leq X)$.
INT07 $\forall X, Y.\, (X < Y \leftrightarrow Y > X)$.
INT08 $\forall X, Y.\, (X \leq Y \leftrightarrow Y \geq X)$.
INT09 $\forall X, Y.\, (\neg\, (X < Y \wedge X \geq Y))$.
INT10 $\forall X, Y.\, (\neg\, (X \leq Y \wedge X > Y))$.
INT11 $\forall X, Y.\, (X \leq Y \wedge Y \leq X \rightarrow X = Y)$.
INT12 $\forall X, Y.\, (X - Y = X + (-Y))$.
INT13 $\forall X.\, (-(-X) = X)$.
INT14 $\forall X.\, (X \cdot 0 = 0)$.
INT15 $\forall X.\, (0 \cdot X = 0)$.
INT16 $\forall X.\, (X \cdot 1 = X)$.
INT17 $\forall X.\, (1 \cdot X = X)$.
INT18 $\forall X.\, (X/1 = X)$.
INT19 $\forall X.\, (X \bmod 1 = 0)$.

[3] Using the `set_DomPred` option of SPASS.

The experiments were carried out on an office PC with a 2.40 GHz Intel Pentium 4 CPU and 512 MB RAM running Debian Linux. All times reported are wall-clock time in seconds.

7.1 TPTP Integer Arithmetic Problems

The list of integer arithmetic problems in the TPTP library contains 147 theorems and 36 non-theorems [8]. Most of the problems are rather easy. The results are summarized in the following table:

Category	Theorems	Non-theorems
Number of problems:	147	36
Proof found:	140	0
Termination in less than 1 second:	136	0
Termination in 1 to 8 seconds:	4	0
No proof found:	7	36
Termination in less than 1 second:	6	31
Termination in 1 to 8 seconds:	1	3
Non-termination:	0	2

7.2 Further Problems

Integer arithmetic. In addition to the TPTP list, we have developed a collection of 75 theorems mostly combining arithmetic and data structures to test SPASS+T. In the following list, the second column gives the runtime of SPASS+T (wall clock time in seconds) and the result, where "+" means "Proof found", "−" means "No proof found", and "∞" means that the time limit of 600 seconds was exceeded.

(1) 1.82 − $\forall X, Z.\,(\exists Y.\,(X + Y = Z))$

(2) 0.16 + $\forall Y, Z.\,(\exists X.\,(X + Y = Z))$

(3) 0.68 − $\forall X, Z.\,(\exists Y.\,(X - Y = Z))$

(4) 0.16 + $\forall Y, Z.\,(\exists X.\,(X - Y = Z))$

(5) 0.33 − $\exists X.\,(X + X < -10)$

(6) 0.19 + $\exists X.\,(X \cdot 3 + (-5) < -12)$

(7) 0.23 + $\exists X, Y.\,(3 \cdot X + 5 \cdot Y = 24)$

(8) 0.22 + $\exists X, Y.\,(3 \cdot X + 5 \cdot Y = 23)$

(9) 0.38 − $\exists X, Y.\,(3 \cdot X + 5 \cdot Y = 22)$

(10) 0.69 + $\forall X, Y.\,(\neg\, 4 \cdot X + 6 \cdot Y = 21)$

(11) 1.01 + $\forall X, Y, Z.\,(2 \cdot X + Y + Z = 10 \wedge X + 2 \cdot Y + Z = 10 \rightarrow \neg\, Z = 0)$

(12) 1.65 + $\forall X, Y, Z.\, (X < 5 \wedge Y < 3 \wedge X + 2 \cdot Y > 7 \rightarrow Y = 2)$

(13) 4.77 + $\forall X.\, (\exists Y.\, (Y < X \wedge \neg\, \exists Z.\, (Y < Z \wedge Z < X)))$

(14) 0.21 + $\neg\, \exists X.\, (0 < X \wedge \forall Y.\, (Y < X \rightarrow Y + 1 < X))$

Integer arithmetic plus free function or predicate symbols.

(15) 0.18 + $\forall X, Y.\, (X + Y = f(X) \wedge Y - f(X) = 0 \rightarrow Y = f(0))$

(16) 0.91 + $\forall X.\, (f(X) > X) \rightarrow f(f(f(6))) \geq 9$

(17) 3.75 + $\forall X.\, (f(X) > X) \rightarrow \forall Y, Z.\, (f(f(Y) + Z) \geq Y + Z + 2)$

(18) 1.23 + $\forall X, Y.\, (X < Y \rightarrow f(X) < f(Y)) \rightarrow \forall Y.\, (f(f(Y) + 2) > f(f(Y)))$

(19) 1.40 – $\forall X, Y.\, (X < Y \rightarrow f(X) < f(Y)) \rightarrow \forall Y.\, (f(f(Y) + 2) > f(f(Y)) + 1)$

(20) 0.18 + $\forall X.\, (f(X) > 1) \rightarrow 7 - 2 \cdot f(3) < 4$

(21) 1.01 + $\forall X.\, (f(X) > X) \rightarrow \forall X.\, (X - f(X) < 0)$

(22) 1.17 – $\forall X, Y.\, (g(X, Y) = g(X, Y + 2)) \wedge g(3, 3) = g(3, 4) \rightarrow g(3, 2) = g(3, 5)$

(23) 0.20 + $p(14 \cdot 3 + 8) \rightarrow p(50)$

(24) 0.24 + $\forall X.\, (p(X + 3)) \rightarrow p(5)$

(25) 0.46 – $\forall X.\, (p(2 \cdot X)) \rightarrow p(10)$

(26) 0.59 – $p(0) \wedge \forall X.\, (p(X) \rightarrow p(X + 1)) \wedge \forall X.\, (p(X) \rightarrow p(X - 1)) \rightarrow \forall X.\, (p(X))$

Integer arithmetic and arrays. The formulas

ARR01 $\forall A, I, X.\, (\mathit{read}(\mathit{write}(A, I, X), I) = X).$
ARR02 $\forall A, I, J, X.\, (I = J \vee \mathit{read}(\mathit{write}(A, I, X), J) = \mathit{read}(A, J)).$

were used to axiomatize arrays for the following problems:

(27) 0.29 + $\forall A, A'.\, (A = \mathit{write}(\mathit{write}(\mathit{write}(\mathit{write}(A', 3, 33), 4, 444), 5, 55), 4, 44) \rightarrow \mathit{read}(A, 3) = 33)$

(28) 0.61 + $\forall A, A', I.\, (A = \mathit{write}(\mathit{write}(\mathit{write}(\mathit{write}(A', 3, 33), 4, 44), 5, 55), I, 66) \rightarrow \mathit{read}(A, 4) = 44 \vee \mathit{read}(A, 4) = 66)$

(29) 1.43 + $\forall A, A', I.(A = \mathit{write}(\mathit{write}(\mathit{write}(\mathit{write}(A', 3, 33), 4, 444), 5, 55), 4, 44) \wedge 3 \leq I \wedge I \leq 4 \rightarrow 33 \leq \mathit{read}(A, I) \wedge \mathit{read}(A, I) \leq 44)$

(30) 0.25 + $\forall A, A'.\, (A = \mathit{write}(\mathit{write}(\mathit{write}(\mathit{write}(A', 3, 33), 4, 444), 5, 55), 4, 44) \rightarrow \exists I.\, (\mathit{read}(A, I) = 33))$

(31) 0.15 + $\forall A, A'.\,(A = write(write(write(write(A', 3, 33), 4, 444), 5, 55), 4, 44) \rightarrow \exists I.\,(read(A, I) < 40 \wedge 30 < read(A, I)))$

(32) 0.39 + $\forall A, A'.\,(\forall I.\,(read(A', I) > I) \wedge A = write(write(A', 3, 5), 7, 9) \rightarrow \forall I.\,(read(A, I) > I))$

(33) 3.78 + $\forall A, A', J.\,(\forall I.\,(read(A', I) > I) \wedge A = write(A', J, J + 3) \rightarrow \forall I.\,(read(A, I) > I))$

(34) 0.37 + $\forall A, A'.\,(A = write(write(write(write(A', 3, 33), 4, 444), 5, 55), 4, 44) \wedge \forall I.\,(read(A', I) < 100) \rightarrow \forall I.\,(read(A, I) < 100))$

(35) 0.30 + $\forall A, A', J, K.\,(\forall I.\,(J \leq I \wedge I \leq K \rightarrow read(A', I) > 0) \wedge A = write(A', K + 1, 3) \rightarrow \forall I.\,(J \leq I \wedge I \leq K + 1 \rightarrow read(A, I) > 0))$

(36) 5.65 + $\forall A, A', J, K.\,(\forall I.\,(J \leq I \wedge I \leq K \rightarrow read(A', I) > 0) \wedge A = write(A', J + 2, read(A', J + 1) + 1) \rightarrow \forall I.\,(J \leq I \wedge I \leq K \rightarrow read(A, I) > 0))$

(37) 3.52 - $\forall A, J, K.\,(\forall I.\,(J \leq I \wedge I \leq K \rightarrow read(A, I) > 0) \rightarrow \forall I.\,(J + 3 \leq I \wedge I \leq K \rightarrow read(A, I) > 0))$

(38) 31.75 + $\forall A.\,(\forall I.\,(1 \leq I \wedge I \leq 10 \rightarrow read(A, I) > I) \wedge \forall I.\,(11 \leq I \wedge I \leq 20 \rightarrow read(A, I) > 20 - I) \rightarrow \forall I.\,(6 \leq I \wedge I \leq 15 \rightarrow read(A, I) > 5))$

(39) 0.23 + $\forall A.\,(\forall I.\,(20 \leq I \wedge I \leq 30 \rightarrow read(A, I) = I + 25) \rightarrow \exists I.\,(read(A, I) = 50))$

(40) 0.56 - $\forall A.\,(\forall I.\,(20 \leq I \wedge I \leq 30 \rightarrow read(A, I) = 2 \cdot I + 3) \rightarrow \exists I.\,(read(A, I) = 53))$

(41) 0.27 + $\forall A, A', I, J, K.\,(\neg\, read(A', I) = read(A', J) \wedge A = write(write(write(A', J, 0), K, read(A', K) + 1), I, 0) \rightarrow \exists L.\,(\neg\, read(A, L) = read(A', L)))$

(42) 9.90 + $\forall A, A', I, J, K.\,(A = write(write(write(A', I, 3), I + 2, 2), I + 4, 1) \wedge I \leq J \wedge J \leq I + 3 \rightarrow \exists K.\,(J \leq K \wedge K \leq J + 3 \wedge read(A, K) \leq 3))$

Integer arithmetic and collections. The formulas

COL01 $\forall I.\,(\neg\, I \in \emptyset)$.
COL02 $\forall I, S.\,(I \in add(I, S))$.
COL03 $\forall I, S.\,(\neg\, I \in remove(I, S))$.
COL04 $\forall I, S, J.\,(I \in S \vee I = J \leftrightarrow I \in add(J, S))$.
COL05 $\forall I, S, J.\,(I \in S \wedge \neg\, I = J \leftrightarrow I \in remove(J, S))$.

were used to axiomatize collections of integers for the following problems:

(43) 0.31 + $\neg\, 4 \in add(1, add(3, add(5, \emptyset)))$

(44) 0.56 + $\forall S, I, J.\,(S = add(5, add(3, add(1, \emptyset))) \wedge I \in S \wedge J \in S \wedge \neg\, I = J \rightarrow I + J < 9)$

(45) 0.23 + $\forall S, S'.\,(\forall I.\,(I \in S' \rightarrow I > 0) \wedge S = add(2, remove(7, S')) \rightarrow \forall I.\,(I \in S \rightarrow I > 0))$

(46) 0.27 + $\forall S, S'.\,(\forall I.\,(I \in S' \rightarrow I > 0) \wedge S = remove(4, remove(1, remove(2, S'))) \rightarrow \forall I.\,(I \in S \rightarrow I > 2))$

(47) 0.64 + $\forall S.\,(\forall I.\,(I \in S \rightarrow I \geq 0) \wedge \neg\, 0 \in S \wedge \neg\, 1 \in S \rightarrow \forall I.\,(I \in S \rightarrow I \geq 2))$

(48) 0.23 + $\forall S.\,(\forall I.\,(I \in S \rightarrow I \geq 0) \wedge \forall I.\,(I \in S \leftrightarrow I \in remove(0, remove(1, S))) \rightarrow \forall I.\,(I \in S \rightarrow I \geq 2))$

(49) 1.92 + $\forall S, S', J.\,(\forall I.\,(I \in S' \rightarrow I > 0) \wedge J \in S' \wedge S = add(J + 2, remove(J, S')) \rightarrow \forall I.\,(I \in S \rightarrow I > 0))$

(50) 2.29 + $\forall S, S', J, K.\,(\forall I.\,(I \in S' \rightarrow I > 0) \wedge J \in S' \wedge K \geq 0 \wedge S = add(J + K, remove(J, S')) \rightarrow \forall I.\,(I \in S \rightarrow I > 0))$

(51) 1.37 + $\forall S, S'.\,(\forall I.\,(I \in S' \rightarrow I > 0) \wedge \forall I.\,(I \in S \rightarrow \exists J.\,(J \in S' \wedge I > J)) \rightarrow \forall I.\,(I \in S \rightarrow I > 1))$

(52) 1.95 + $\forall S, S'.\,(\forall I.\,(I \in S' \rightarrow I > 0) \wedge \forall I.\,(I \in S \rightarrow \exists J.\,(J \in S' \wedge 2 \cdot I - 5 \cdot J > 0)) \rightarrow \forall I.\,(I \in S \rightarrow I > 2))$

(53) 1.90 + $\forall S, S'.\,(\forall I.\,(I \in S' \rightarrow I > 0) \wedge \forall I.\,(I \in S \rightarrow \exists J, K.\,(J \in S' \wedge K \in S' \wedge I = J + K)) \rightarrow \forall I.\,(I \in S \rightarrow I > 1))$

(54) 0.36 + $\forall S.\,(S = add(10, add(30, add(50, \emptyset)))$
$\rightarrow \exists I.\,(20 \leq I \wedge I \leq 40 \wedge I \in S))$

Integer arithmetic and collections with counting. The formulas

CCO01 $\forall I.\,(\neg\, I \in \emptyset)$.
CCO02 $\forall I, S.\,(I \in add(I, S))$.
CCO03 $\forall I, S.\,(\neg\, I \in remove(I, S))$.
CCO04 $\forall I, S, J.\,(I \in S \vee I = J \leftrightarrow I \in add(J, S))$.
CCO05 $\forall I, S, J.\,(I \in S \wedge \neg\, I = J \leftrightarrow I \in remove(J, S))$.
CCO06 $\forall S.\,(count(S) \geq 0)$.
CCO07 $\forall S.\,(S = \emptyset \leftrightarrow count(S) = 0)$.
CCO08 $\forall I, S.\,(\neg\, I \in S \leftrightarrow count(add(I, S)) = count(S) + 1)$.
CCO09 $\forall I, S.\,(I \in S \leftrightarrow count(add(I, S)) = count(S))$.
CCO10 $\forall I, S.\,(I \in S \leftrightarrow count(remove(I, S)) = count(S) - 1)$.
CCO11 $\forall I, S.\,(\neg\, I \in S \leftrightarrow count(remove(I, S)) = count(S))$.
CCO12 $\forall I, S.\,(I \in S \rightarrow S = add(I, remove(I, S)))$.

were used to axiomatize collections of integers with a counting operation for the following problems:

(55) 22.09 + $\forall S.\,(count(remove(5, S)) \geq 7$
$\rightarrow count(remove(4, S)) \geq 6)$

(56) 3.71 + $\forall S.\,(count(add(5, S)) = count(add(3, S))$
$\rightarrow count(remove(5, S)) = count(remove(3, S)))$

(57) 2.20 + $\forall S, I.\,(count(S) + 1 \geq count(add(I, S)))$

(58) 11.95 + $\forall S, I, K.\,(K > 0 \rightarrow count(S) + K \geq count(add(I, S)))$

(59) 19.49 + $\forall S, I, K.\,(K > 0 \rightarrow count(S) + K \geq count(add(I, add(I, S))))$

(60) 0.42 + $\forall S, I.\,(count(remove(2, S)) = 0$
$\wedge\, count(remove(3, S)) = 0$
$\rightarrow count(remove(I, S)) = 0)$

(61) 0.32 + $\forall S.\,(2 \in S \wedge count(S) = 1 \rightarrow \neg\, 5 \in S)$

(62) 23.22 + $\forall S.\,(2 \in S \wedge 3 \in S \wedge count(S) = 2 \rightarrow \neg\, 5 \in S)$

(63) 28.16 + $\forall S, I.\,(2 \in S \wedge 3 \in S \wedge count(S) = 2 \wedge I > 3 \rightarrow \neg\, I \in S)$

(64) 118.41 + $\forall S, I, J.\,(I < 3 \wedge 6 < J$
$\wedge\, I \in S \wedge J \in S$
$\wedge\, count(S) = 2$
$\rightarrow \neg\, 5 \in S)$

(65) 0.34 + $\exists S, I.\,(count(S) + 1 > count(add(I, S)))$

(66) 309.91 + $\exists S.\,(count(S) = 3)$

(67) 269.01 + $\forall S, I, J, K.\,(I > J \wedge J > K \wedge I \in S \wedge J \in S \wedge K \in S \rightarrow count(S) > 2)$

(68) 0.53 + $\forall S, J.\,(\forall I.\,(I \in S \rightarrow J > I) \rightarrow count(add(J, S)) > count(S))$

(69) 4.04 + $count(add(1, add(3, \emptyset))) = 2$

(70) 1.02 + $count(add(5, remove(3, add(3, \emptyset)))) = 1$

(71) ∞ - $count(add(1, add(5, remove(3, add(2, \emptyset))))) = 3$

(72) 3.49 + $\forall S, I.\,(count(remove(I, add(I, S))) = count(remove(I, S)))$

(73) ∞ - $\forall S, I.\,(count(add(0, remove(I, add(I, S)))) = count(add(0, remove(I, S))))$

Integer arithmetic and pointer data types. The formulas

PNT01 $\forall X.\,(\neg\, isrecord(X) \rightarrow length(X) = 0)$
PNT02 $\forall X.\,(isrecord(X) \rightarrow length(X) \geq 1)$
PNT03 $\forall X.\,(isrecord(X) \rightarrow length(X) = length(next(X)) + 1)$
PNT04 $\forall X.\,(\neg\, isrecord(X) \rightarrow \neg\, isrecord(split1(X)))$
PNT05 $\forall X.\,(isrecord(X) \rightarrow isrecord(split1(X)))$
PNT06 $\forall X.\,(isrecord(X) \rightarrow data(split1(X)) = data(X))$
PNT07 $\forall X.\,(isrecord(X) \wedge \neg\, isrecord(next(X)) \rightarrow \neg\, isrecord(next(split1(X))))$
PNT08 $\forall X.\,(isrecord(X) \wedge isrecord(next(X)) \rightarrow next(split1(X)) = split1(next(next(X))))$
PNT09 $\forall X.\,(\neg\, isrecord(X) \rightarrow \neg\, isrecord(split2(X)))$
PNT10 $\forall X.\,(\neg\, isrecord(next(X)) \rightarrow \neg\, isrecord(split2(X)))$
PNT11 $\forall X.\,(isrecord(X) \wedge isrecord(next(X)) \rightarrow isrecord(split2(X)))$
PNT12 $\forall X.\,(isrecord(X) \wedge isrecord(next(X)) \rightarrow data(split2(X)) = data(next(X)))$
PNT13 $\forall X.\,(isrecord(X) \wedge isrecord(next(X)) \rightarrow next(split2(X)) = split2(next(next(X))))$

were used to axiomatize singly linked lists with pointers and two recursive functions *split1* and *split2* (computing the sublists of odd-numbered and even-numbered elements of a list) for the following problems:

(74) 20.02 + $\forall X, Y.\,(isrecord(X) \wedge isrecord(next(X)) \wedge Y = next(next(X)) \wedge (2 \cdot length(split2(Y)) = length(Y) - 1 \vee 2 \cdot length(split2(Y)) = length(Y)) \rightarrow (2 \cdot length(split2(X)) = length(X) - 1 \vee 2 \cdot length(split2(X)) = length(X)))$

(75) 17.18 + $\forall X, Y.\, (\mathit{isrecord}(X) \wedge \mathit{isrecord}(\mathit{next}(X)) \wedge Y = \mathit{next}(\mathit{next}(X))$
$\wedge\ \mathit{length}(\mathit{split1}(Y)) + \mathit{length}(\mathit{split2}(Y)) = \mathit{length}(Y)$
$\rightarrow\ \mathit{length}(\mathit{split1}(X)) + \mathit{length}(\mathit{split2}(X)) = \mathit{length}(X))$

7.3 Discussion

Summarizing the previous section, we see that SPASS+T solves 63 out of the 75 problems in our list. In order to check to what extent the mechanisms implemented in SPASS+T contribute to these proofs, we have gradually disabled various features and re-run the tests. The following table shows the number of proofs found for several combinations of features:

SMT procedure	Theory instantiation	Arithmetic simplification	Auxiliary axioms	Problems solved (out of 75)
+	+	+	+	63
+	+	−	+	52
+	−	+	+	56
−	−	+	+	39
−	−	−	+	12
+	+	+	−	53

How does SPASS+T compare to a system like Simplify (Detlefs, Nelson, and Saxe [4])? The automatic theorem prover Simplify was developed as the proof engine of ESC/Java and ESC/Modula-3. It handles universal quantifiers in the input by computing (hopefully) relevant instances in a similar way as the *theory instantiation* rule of SPASS+T and analyzing the resulting formulas using SAT checking and a Nelson-Oppen combination of decision procedures. Simplify (version 1.5.4) solves 40 out of the 75 problems in our list. The difference to SPASS+T is partially due to the fact that Simplify solves only four out of 23 problems that involve existential quantifiers, namely (41) and (51)–(53). Somewhat surprisingly, however, Simplify fails also for (55), (56), (59), and (68)–(74). On the other hand, due to the highly optimized implementation and the reduced search space, all problems that Simplify does solve are solved in less than one second.

It is perhaps illustrative to have a closer look at those problems that SPASS+T failed to prove. SPASS+T has some support for solving non-ground equations, but it makes no attempt to be complete in this domain, and in particular it does not use any kind of theory unification. This accounts for most of the missed proofs in the TPTP list and in theorems (1)–(14), as well as for (22), (25), and (40). It may look strange that SPASS+T succeeds for (2) and (4), but fails for (1) and (3), but the explanation is easy: (2) and (4) are proved by superposition with INT01. We could add the symmetric version of INT01, so that (1) and (3) become provable as well, but having both INT01 and its symmetric version in the clause set destroys termination and seems to create more problems than it solves. Similarly, (7) and (8) can be handled by superposition with INT14 or INT16 plus elementary equation solving, whereas (9) would require a fully-fledged Diophantine equation solver.

Theorem (19) is rather difficult for an automated system because for proving it one has to "invent" a term that does not occur in the problem, namely $f(f(Y) + 1)$, and

then use transitivity. Moreover, as one might expect, SPASS+T has no chance to prove the induction theorem (26). On the other hand, an extension of the *theory instantiation* rule would enable SPASS+T to prove the pigeonhole-like formula $\forall X, Y.\ (f(X) = f(Y) \rightarrow X = Y) \wedge 6 < f(3) \wedge f(3) < 9 \wedge 6 < f(4) \wedge f(4) < 9 \rightarrow f(5) < 6 \vee 9 < f(5)$ from the TPTP list. Two changes are required: The inference rule *theory instantiation II* has to take an arbitrary number of premises

$$\frac{ground(s_1) \ldots ground(s_k) \qquad L[t_1, \ldots, t_k] \vee D}{ground(u_1) \ldots ground(u_n) \quad \mathbf{export}(D')}$$

moreover it has to be applicable even if the terms t_i occur below a negated equation symbol. So far, we have not implemented this modification, since the extended rule would be extremely prolific.

Even though the CCO axioms have a simple operational reading, it seems to be difficult for automated provers to detect this fact, as one can see from the poor results for straightforward computation tasks like (71) or (73). Simplify even exceeds the 600 second time limit for the simpler variants (69), (70), and (72).

There is one odd case left: Surprisingly, SPASS+T succeeds to prove formula (37), if *theory instantiation* or *integer ordering expansion* are switched off, but fails to find the proof if both inference rules are switched on. It turns out that, in the latter case, SPASS+T triggers a bug in CVC Lite.

7.4 Download

A current snapshot of SPASS+T for Linux including the benchmark problems can be downloaded from `http://www.mpi-inf.mpg.de/~uwe/software/`.

8 Outlook

The development of SPASS+T is ongoing work. More effort has to be spent on fine-tuning the theory instantiation rule and the arithmetic simplification rules. A more challenging extension is the implementation of subsumption checks for clauses with theory literals, which might turn SPASS+T into a decision procedure for some classes of formulas. We are also looking into using SPASS+T with non-numeric base theories, such as arrays, lists, or bitvectors.

Acknowledgments: We are grateful to Christoph Weidenbach and Thomas Hillenbrand for helpful discussions.

References

[1] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing*, 5(3/4):193–212, 1994.

[2] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the Cooperating Validity Checker. In Rajeev Alur and Doron A. Peled, editors, *Computer*

Aided Verification, 16th International Conference, CAV 2004, LNCS 3114, pages 515–518, Boston, MA, USA, 2004. Springer-Verlag.

[3] Robert S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In Jean E. Hayes, Donald Michie, and Judith Richards, editors, *Machine Intelligence 11: Logic and the acquisition of knowledge*, chapter 5, pages 83–124. Oxford University Press, 1988.

[4] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[5] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004*, LNCS 3114, pages 175–188, Boston, MA, USA, 2004. Springer-Verlag.

[6] Harald Ganzinger, Viorica Sofronie-Stokkermans, and Uwe Waldmann. Modular proof systems for partial functions with weak equality. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning: Second International Joint Conference, IJCAR 2004*, LNAI 3097, pages 168–182, Cork, Ireland, 2004. Springer-Verlag. Corrected version at `http://www.mpi-sb.mpg.de/~uwe/paper/PartialFun-bibl.html`.

[7] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005*, LNCS 3576, pages 476–490, Edinburgh, Scotland, UK, 2005. Springer-Verlag.

[8] Stephan Schulz and Geoff Sutcliffe. `http://www.cs.miami.edu/~tptp/TPTP/Proposals/IntegerArithmetic.p`, March 15, 2006.

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

[10] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobalt, and Dalibor Topić. SPASS version 2.0. In Andrei Voronkov, editor, *Automated Deduction – CADE-18, 18th International Conference on Automated Deduction*, LNAI 2392, pages 275–279, Copenhagen, Denmark, 2002. Springer-Verlag.

Search for faster and shorter proofs using machine generated lemmas

Petr Pudlák
Charles University in Prague
pudlak@artax.karlin.mff.cuni.cz

Abstract

When we have a set of conjectures formulated in a common language and proved from a common set of axioms using an automated theorem prover, it is often possible to automatically construct lemmas that can be used to prove the conjectures in a shorter time and/or with shorter proofs.

We have implemented a system that repeatedly tries to improve the set of assumptions for proofs of given conjectures using lemmas that it extracts from the proofs constructed by an automated theorem prover. In many cases it can significantly reduce the total time or the overall sum of the lengths of the proofs of the conjectures.

We present several examples of such sets of conjectures and show the improvements gained by the system.

1 Motivation

Imagine a professor of mathematics who gives the same lectures every year. She works in a standard theory and she has a fixed set of theorems that she wants to present and prove every year during the course. And because she wants to save her and her students' time, she would like to have as efficient proofs as possible. She would like to find such lemmas that would shorten the total time she has to spend for presenting the proofs.

Our aim will be to search for lemmas in such environment and try to use them to make the proofs of conjectures more efficient.

Finding useful lemmas has of course much more serious applications, namely reducing the amount of resources necessary to prove a particular set of conjectures or to prove conjectures that could not be proved without such lemmas.

Currently the system only restructures and compacts the proofs and doesn't handle conjectures that the prover was not able to prove at the beginning. It can only improve existing proofs. In future we hope to be able to also search for proofs of the conjectures that couldn't be proved from the initial set of assumptions.

2 Previous research

The idea of an automated discovery of lemmas or theorems is not a new one. There were many different approaches to solve this task.

Owen L. Astrachan and Mark E. Stickel [AS92] used the idea of reusing lemmas to speed up a model elimination theorem prover.

Art Quaife [Qua92] used Otter [McC94] to prove many fundamental mathematical theorems. He included the theorems he had already proved as assumptions for the more complicated ones. The sequence in which the theorems were proved was determined by Quaife, based on his mathematical knowledge.

Marc Fuchs, Dirk Fuchs and Matthias Fuchs sought for lemmas using genetic programming to improve tableau-based proof search [FFF99].

The HR system [Col02a], named after mathematicians Hardy and Ramanujan, uses a model generator to construct models based on a set of axioms, attempts to formulate conjectures and then prove them using an automated theorem prover. A brief description can be also found in [Col02b].

Larry Wos and Gail W. Pieper describe the technique of lemma adjunction in [WP03] and also discuss many different approaches for evaluating lemmas.

Article [SGC03] summarizes many different criteria for constructing and identifying quality lemmas. Humans generally use the *inductive approach* – from many similar problems they try to induce a more general conjecture. This approach requires good knowledge of the particular field of mathematic and also good mathematical intuition. Another approach described is *generative*, when more sophisticated techniques (for example syntactic manipulation) are used to construct new conjectures. An example of such a system is the HR [Col02b] program. The *manipulative approach* tries to construct interesting conjectures from already existing theorems. Finally, the *deductive approach* tries to automatically construct many logical consequences from a set of axioms using an automated theorem prover and then filter them and pick those that are interesting for the researcher.

The authors of [SGC03] also categorize different possible filters for interestingness. The filters include non-obviousness, novelty, suprisingness, intensity and usefulness.

Our approach is somewhat different from those mentioned above. It lies in between the manipulative and the deductive approach. We observe proofs of all the given problems and identify lemmas that are common to many of the proofs. The filter we develop and use for selecting good lemmas falls into the *usefulness* category - our measure is, how much each lemma could contribute to the proofs of other conjectures. Unlike other measures, this one can easily be evaluated by comparing different proofs conducted by the prover.

The lemmas that we produce are therefore interesting from the point of view of a machine. Hence, this can give us an interesting comparison between human and machine opinions on the usefulness of a lemma.

The ideas in the article [SGC03] and personal communication with many other researchers inspired our work which we present in this paper together with the empirical results we have obtained.

3 Overview of the system

In sequel we assume that a consistent theory is given with some (possibly infinite) set of axioms. All the formulas we work with are formulated in the language of the theory and the conjectures that are to be proved and the lemmas that are constructed are proved using axioms selected from the axioms of the theory.

The systems starts by proving the conjectures one by one. The proofs of the conjectures that were successfully proved are then analyzed by the system. Formulas that appear multiple times in the proofs are then used as additional assumptions when looking for less costly proofs of the conjectures. The system tries to optimize the set of such formulas. We will call such formulas *lemmas*.

3.1 Basic notions

Let us first define some basic notions we will use throughout the text.

Notation 1 (Axiom, conjecture, lemma, proof) *By an* axiom *we understand either an axiom of the underlying theory or a well known theorem of the theory that we use as an assumption.*

A conjecture *is a formula given on the input that is to be proved by the system. The system tries to find the most efficient proof of the conjecture using different* sets of assumptions*. Each such a set can contain some of the axioms, other conjectures or lemmas.*

A lemma *is a formula constructed by the system that is proved in a similar fashion as the conjectures are, and is used to improve the proofs of the conjectures or the proofs of other lemmas.*

A proof *is an output of a successful run of the prover. As we use a single theorem prover with the same settings on every run, the proof only depends on the conjecture being proved and on the set of assumptions being used.*

The aim of the system is to find such lemmas and such sets of assumptions for the conjectures and the lemmas that either the total time required to prove the conjectures and the lemmas or the overall size or length of the proofs is minimized.

3.2 System input

Initially, the system is given a set **C** of conjectures and for each conjecture C from **C** the system is given an initial set of assumptions $\mathbf{A}_C$. The set $\mathbf{A}_C$ consists of a subset of axioms of the underlying theory and (possibly) of some other conjectures from **C**.

The idea is that the proofs of conjectures in **C** need not use the set of all the axioms of the theory (which might be infinite) and that some other conjectures from **C** might be useful. The user may have his own idea, which conjectures to add as assumptions to $\mathbf{A}_C$.

In the process of computation the system tries to optimize the set of lemmas and for each lemma the set of its assumptions.

3.3 System output

At the end the system outputs the conjectures given at the input along with the lemmas that participate in the fastest/shortest set of proofs. For each conjecture and each lemma it outputs the optimal set of assumptions it has found, in the sense described later.

4 An example

In our example we use a commonly used encoding to encode an axiomatization of propositional logic into terms, thus forcing the prover to use only the specified axioms and rules for inferences. Then we use the system to prove some basic propositional theorems.

Note that the automated theorem provers were already used to look for interesting axiomatics of propositional logic, for example [MVF+02]. However, our aim is not to look for a new or otherwise interesting axiomatic, we use the system to optimize the proofs of several theorems.

We will denote the code of a formula by an over-line. For example for a propositional formula φ we denote the term that codes it by $\overline{\varphi}$. Negation is coded by a unary function n and implication is coded by a binary function i. The fact that φ is a theorem $\vdash \varphi$ is coded by a unary predicate t. The following table summarizes the codes:

$$\begin{array}{ll} \overline{\neg\varphi} & n(\overline{\varphi}) \\ \overline{\varphi \to \psi} & i(\overline{\varphi},\overline{\psi}) \\ \overline{\vdash \varphi} & t(\overline{\varphi}) \end{array}$$

We use "$\to$" just for implication in propositional logic we are coding. We use "$\Rightarrow$" for implication in predicate logic in which the conjectures are presented and in which the prover actually works. Thus the statement "if φ is provable then $\psi \to \chi$ is provable" would be coded as $t(A) \Rightarrow t(i(B,C))$.

In order to increase the difficulty of the task we've used Meredith's single axiom for propositional logic. It has only a single axiom schema and a single rule.

The axiom schema is defined for any formulas φ, ψ, χ, ξ and η, but the theorem prover computes with their codes, which are represented by the variables A, B, C, D and E in the language of the theorem prover. As these variables are universally quantified, the prover can replace each variable by any coded formula. The same applies to the coded modus ponens rule.

The coded representations of the schema and the rule are:

$$\begin{array}{rl} \text{Meredith:} & \vdash ((((\varphi \to \psi) \to (\neg\chi \to \neg\xi)) \to \chi) \to \eta) \to ((\eta \to \varphi) \to (\xi \to \varphi)) \\ & t(i(i(i(i(i(A,B),i(n(C),n(D))),C),E),i(i(E,A),i(D,A)))) \\ \text{MP:} & \varphi, \varphi \to \psi \vdash \psi \\ & (t(A) \wedge t(i(A,B))) \Rightarrow t(B) \end{array}$$

This set of axioms was used as the initial set of assumptions for all the conjectures. No other dependencies between the conjectures were specified. The system was let to discover all the lemmas by itself from the proofs of the conjectures.

Table 1 shows the conjectures whose proofs the system was improving. The conjectures were proved using E prover[Sch02]. The cost of the proofs was measured in the number of processed clauses. This measure closely corresponds to the time the prover spends on a problem, but it is not dependent on the hardware the prover runs on. In the second column the conjectures are presented in their coded form. The third column shows the number of processed clauses the prover spent on each of the conjectures if they were proved only using the original assumptions. The fourth column shows the cost of the proofs after the system had finished the improvements using lemmas generated from the proofs. Note that to obtain the total cost of the final proofs we also have to include the cost of the proofs of the generated lemmas. The full listing of the generated lemmas is included in Table 10. As we can see, the total cost of proofs needed to prove the conjectures in this case was reduced to 3%.

5 Description of the system

Let us first discuss how the lemmas are generated. The modification of the set of assumptions of the conjectures by using the lemmas will be described in the next part.

conjecture	coding into terms	original cost	final cost
$\neg(\varphi \to \neg\psi) \vdash \varphi$	$t(n(i(A,n(B)))) \Rightarrow t(A)$	938	62
$\neg(\varphi \to \neg\psi) \vdash \psi$	$t(n(i(A,n(B)))) \Rightarrow t(B)$	1673	17
$\varphi \vdash \neg\varphi \to \psi$	$t(A) \Rightarrow t(i(n(A),B))$	31	4
$\psi \vdash \neg\varphi \to \psi$	$t(B) \Rightarrow t(i(n(A),B))$	13	3
$\neg(\varphi \to \varphi) \vdash \psi$	$t(n(i(A,A))) \Rightarrow t(B)$	41	8
$\vdash \varphi \to (\psi \to \varphi)$	$t(i(A,i(B,A)))$	17	4
$\varphi \to \neg(\psi \to \neg\chi), \varphi \vdash \chi$	$(t(i(A,n(i(B,n(C)))))) \wedge t(A)) \Rightarrow t(C)$	287	20
$\vdash \varphi \to \varphi$	$t(i(A,A))$	15	2
$\vdash \neg\varphi \to (\varphi \to \psi)$	$t(i(n(A),i(A,B))))$	5731	4
$\vdash \neg\neg\varphi \to \varphi$	$t(i(n(n(A)),A))$	3297	7
$\vdash \varphi \to \neg\neg\varphi$	$t(i(A,n(n(A))))$	3682	4
total cost of the conjectures:		15725	135
cost for proving the lemmas:			375
total cost:		15725	510

Table 1: Formulas proved from the Meredith's axiomatization.

5.1 Generation and evaluation of lemmas

Notation 2 (Subsumption) *By $c \sqsubseteq d$ we denote that the clause c that subsumes the clause d.*

Let a set of proofs[1] **P** be given. We collect all the clauses that appear in the proofs and that were derived only from the assumptions into a single set

$$\mathbf{L} = \bigcup_{P \in \mathbf{P}} \{c \mid c \text{ is a clause in } P \text{ derived only from assumptions and without Skolem symbols}\}$$

We do not include clauses that were derived from the negated conjecture, because they are not true formulas in our theory. Also we remove those clauses that contain Skolem symbols created by the prover, because these symbols have different meaning in different runs of the prover. (In future we might implement the reverse skolemization algorithm [CP80, CP93] to deal with such formulas.)

Thus, the clauses in **L** don't contain any skolem symbols, but they have no special form, they can contain arbitrary number of positive and/or negative literals.

Note that although it would be an obvious thing to do, we don't use the conjectures as lemmas. The reason is that the conjectures are not necessarily clauses, therefore we can't process them the same way as the clauses produced by the prover. As it turns out, many of the conjectures are then anyway discovered as lemmas by the system. However, this issue deserves a better solution in the future.

From the set **L** we construct a minimal $\mathbf{L}'$ set with respect to subsumption such that $\mathbf{L}'$ has the following properties:

1. $\mathbf{L}' \subseteq \mathbf{L}$, therefore $\mathbf{L}'$ contains only true formulas.

[1] Recall that for us a proof is a successful run of the prover that proves a particular conjecture from a given set of assumptions.

2. for every $d \in \mathbf{L}$ there is $c \in \mathbf{L}'$ such that $c \sqsubseteq d$.

3. for any pair $c \in \mathbf{L}'$, $d \in \mathbf{L}'$ we know that c doesn't subsume d: $c \not\sqsubseteq d$.

Therefore, $\mathbf{L}'$ contains just the most general variants of the lemmas appearing in $\mathbf{L}$.

The clauses from $\mathbf{L}'$ are then used as lemmas to modify the set of assumptions of the conjectures.

Now let us have a clause c that is a lemma, $c = L_1, \ldots, L_k, \neg L_{k+1}, \ldots, \neg L_n$ where L_i are atomic formulas. Let $|L_i|$ be the number of function symbols appearing in L_i. Let P by a proof whose set of assumptions we want to improve by c. We would like to have an estimate that would tell us, if it is likely that c will contribute to the proof P. A natural idea suggests itself that the more often an atomic formula L_i occurs in the proof P the more likely the lemma will contribute and also that the longer L_i is (measured in the number of symbols) the more likely it will shorten/speed up the proof. Therefore our estimation formula is defined by

$$\text{weight}(c,P) = \sum_{i=1}^{n} |L_i| \cdot (\textit{the number of occurrences of } L_i \textit{ in } P) \tag{1}$$

There are many other possible estimation methods. One may, for example, take into account the number of formulas in the proof which the lemma subsumes, look for similar terms in the lemma and in the proof, etc.

5.2 Evaluation of the proofs

In order to evaluate the applicability of lemmas, we need to have a criterion for the cost of a proof:

Notation 3 (Measure of a proof) *A* proof measure *is a function that maps proofs into non-negative real numbers.*

We use the following proof measures:

the number of processed clauses reported by the prover; this measure is closely related to the time the prover spends while searching for the proof of the conjecture, but is independent of the hardware of the computer the prover runs on;

the length of the proof is the number of formulas appearing in the proof that is constructed by the prover;

the size of the proof is the total number of occurrences of function symbols (not predicate symbols) appearing in the proof that is constructed by the prover.

The length and the size of a proof are also independent on the hardware of the computer being used.

Notation 4 *If we are proving a conjecture C from assumptions $A_1, \ldots, A_n$, we denote the measure of the proof by*

$$||A_1, \ldots, A_n \rhd C||$$

If the prover is not able to conduct the proof we set

$$||A_1, \ldots, A_n \rhd C|| = +\infty$$

Remark 1 (The proof size/length) *If we could prove all the given conjectures together, the size/length of the resulting hypothetical proof would be of course less than the sum of the sizes/lengths of the individual proofs of the conjectures. But in most cases the prover is not able to prove all the conjectures together, therefore the system proves the conjectures one by one. The system then tries to compact the proofs of the conjectures to make them closer to the size of the hypothetical proof.*

The number of generated lemmas is usually very large, so only some of them will be included in the output. Such lemmas are marked as *accepted*. Each lemma is initially unaccepted. When the system figures out that the lemma is worth including in the output, it marks it as accepted.

For each conjecture or lemma C the system maintains a list $\mathbf{S}_C$ of sets of assumptions that were used to produce different proofs of the conjecture.

Each such a set $S \in \mathbf{S}_C$ is marked as *accepted* iff all the lemmas it contains are accepted. The initial set of assumptions of each conjecture contains no lemmas, hence it is always accepted. Now, we may define:

Notation 5 (Best accepted set of assumptions) *The best accepted set of assumptions*

$$S_C^{best} \in \mathbf{S}_C$$

of a conjecture C is an accepted set of assumptions that produces the best proof of C with respect to the proof measure:

$$\forall S:\ ((S \in \mathbf{S}_C) \wedge (\textit{all the lemmas in } S \textit{ are accepted})) \Rightarrow ||S \rhd C|| \geq ||S_C^{best} \rhd C|| \qquad (2)$$

If there are several sets of assumptions that match the criteria for the best accepted set (they produce proofs of the same measure), we arbitrarily choose one among them.

Let us call the proof of C produced from S_C^{best} the best accepted proof of C.

Remark 2 (System output) *The output of the system consists of all the input conjectures and all the accepted lemmas at the time the system finishes execution. For each of these conjectures or lemmas the best accepted set of assumptions is presented.*

5.3 Modifying the set of assumptions to get better ones

The outline of the work of the system is as follows:

1. The system first tries to prove all the given conjectures one by one. Let $\mathbf{C}$ be the set of those conjectures that were successfully proved. Let $\mathbf{L}$ be the set of constructed lemmas and let $\mathbf{L}^a \subseteq \mathbf{L}$ be the set of lemmas that are accepted. Initially, these sets are empty.

2. The system takes the best accepted proofs of all the conjectures and accepted lemmas. From those proofs the system constructs new lemmas as described in section 5.1. It sets

$$\mathbf{L} = \mathbf{L} \cup \{\text{the newly constructed lemmas}\}$$

3. The system produces pairs consisting of a best assumption set of a conjecture and of a lemma that will be used to improve the set of assumptions of the best proof of the conjecture. It takes all the new lemmas together with the lemmas it already has constructed before and combines them with all the best accepted sets of assumptions of all the conjectures and accepted lemmas. This way it produces every possible pair of

$$\mathbf{L} \times \{S_C^{\text{best}} \,|\, C \in \mathbf{C} \cup \mathbf{L}^a\}$$

However some pairs of a lemma l and the best set of assumptions S_C^{best} of a conjecture C have to be excluded, namely

- if already $l \in S_C^{\text{best}}$, or
- if the lemma l is the conjecture C itself, or
- if the conjecture C is directly or indirectly used to prove l; this means that either C is one of the assumptions in the best accepted set of l, or it is one of the assumptions in the best set of some conjecture that is an assumption of l, and so on.

Otherwise it could happen for example that there would be two equivalent lemmas, one proving another with a one-step proof.

4. These pairs are sorted according to the estimate described in section 5.1.

5. The system subsequently takes these pairs (l, S_C^{best}) starting with the one with the highest weight:

   (a) It tries to conduct a new proof of C using $S_C^{\text{best}} \cup \{l\}$. The gain of this single improvement is

   $$g = ||S_C^{\text{best}} \rhd C|| - ||S_C^{\text{best}}, l \rhd C||$$

   (b) If g is positive, it means that the lemma l has brought an improvement. The lemma l is then checked, if its total gain to all the conjectures whose set of assumptions were improved by l is larger than the cost of the proof of l. If so, the system sets

   $$\mathbf{L}^a = \mathbf{L}^a \cup \{l\}$$

   which means that l is marked as accepted.

   If l is accepted, we want to incorporate the lemmas that originate from the proof of l as well as the lemmas that originate from the proofs of the conjectures that use l as an assumption. Hence, in such a case the process is restarted from the beginning and the system goes again to point 2.

   (c) Until there are any unprocessed pairs (l, S_C^{best}) the system takes the next pair and goes again to 5a.

6. When all the pairs are exhausted and no improvement was made the system outputs the conjectures $\mathbf{C}$, the accepted lemmas $\mathbf{L}^a$ and their best accepted sets of assumptions S_C^{best}, $C \in \mathbf{C} \cup \mathbf{L}^a$, as described above, and halts.

It may happen that as the result of adding l, some of the assumptions in S_C^{best} are not needed any more and do not appear in the proof. In such a case the system omits them and tries to prove the conjecture only using this reduced set of assumptions. If the proof attempt succeeds, it is evaluated the same way as described above. This practice helps to reduce the number of assumptions appearing in the proofs. Without it, the prover would soon become overwhelmed by the number of assumptions and no further improvement would be possible. In the current version of the system this check is not iterated, so the proof conducted from the reduced set of assumptions is not checked again for redundant assumptions.

6 Experimental results

The system is still under development, therefore we don't have an in-depth statistics of its behavior. However, we have performed tests on different sets of conjectures from different sources and we present a selection of them.

All experiments were run on a Linux machine with Intel® Pentium® 4 CPU 3.4GHz with 2GB of RAM. The conjectures were proved using E prover with resource limits set to 80 seconds, 512MB RAM and 100000 processed clauses.

We have developed an independent server component that lies between the actual system and the automated prover. The component caches all conducted proofs on a hard-disk and if a client requests the server to perform a proof that has already been processed, the server returns the cached version. This greatly speeds up the development of the system, particularly if we rerun the system many times with different settings or with slight modifications on the same set of problems.

For some cases we include a full list of the conjectures and the lemmas, for others we only summarize the results.

6.1 TPTP problems – set theory

We used several selected TPTP [SS98] problems concerning set theory. These problems originate from [Qua92]. For presentation in this paper we have converted the machine syntax into a more human-readable form. The meaning of the symbols that we use is summarized in Table 2. The conjectures that were proved are shown in Table 3. The table also shows the results obtained when running the system with the proofs being measured by the number of processed clauses. The second column labeled "level" is an inductively defined property defined for both the conjectures and the lemmas. The axioms have level 0. If a conjecture or a lemma is proved just from the axioms, it has level 1. Each lemma or conjecture has its level set to the maximum level of its assumptions plus 1. This can give us an approximation on how complicated the conjecture or the lemma is. The third column labeled $||\cdot||_i$ shows the initial cost of the proof of the conjecture while the fourth column labeled $||\cdot||_f$ shows the final cost of the proof of the conjecture when the system terminates. The fifth column shows the actual formula converted into a human-readable form. The last column shows which lemmas were finally used to prove the conjecture.

Some of the conjectures in the listings may have levels greater than 1 although they don't use any lemmas or the level of a particular conjecture is higher than one plus the level of the lemmas that are used to prove it. The reason is that the user has specified that some other

$A \subseteq B$	subclass
$\{A, B\}$	unordered pair
$\langle A, B \rangle$	ordered pair
$\{A\}$	singleton
$A_{[1]}$	first member of an ordered pair A
$A_{[2]}$	second member of an ordered pair A
$A \in B$	membership
$\overline{A}$	class complement
$A \times B$	cross product
$\emptyset$	null class
$A \cap B$	intersection
$\mathcal{U}$	universal class
member_of(A)	choice function (from the axiom of choice)

Table 2: Meaning of the symbols used in the TPTP set theory sample

conjectures should be used as assumptions to prove the conjecture and they increase the level of the conjecture.

We can see that often the conjectures that were harder to prove with respect to the proof measure have a higher level. This means that they were finally proved using several layers of lemmas.

Several conjectures that were too complicated are omitted for brevity.

The lemmas that the system generated with the processed clauses count proof measure are in Table 4. The lemmas are clauses, but for the presentation we have converted them into a more readable form using implication notation. As the lemmas have no initial set of axioms given by the user, they also have no initial cost, so only their final cost is shown.

They are sorted by their total accumulated improvement with respect to the proof measure, the more useful lemmas are at the beginning of the table. This is however only an informative ordering, as it depends very much on the order in which the lemmas were tried. For example, consider some two almost same lemmas l_1 and l_2. Whichever is chosen second will bring no improvement as the assumption sets were already improved by the one chosen first.

The lemmas are more or less complicated formulas that the system found useful for proving the conjectures. This can give us an interesting comparison, as the input conjectures were selected by a human, whereas the lemmas were constructed by a machine. Tables 5 and 6 show the results on the same set of conjectures when different proof measures were selected. Many of the lemmas appear in all three tables, although in different positions. Such lemmas seem to be essential for the automated prover when working with this particular theory.

Note that some of the lemmas the system has found are just conjectures reformulated as clauses. In such a case the system usually proves such conjecture using the lemma in a single step and then further improves the proof of the lemma.

Table 7 shows the summary of achieved results. For each proof measure the initial and the final cost of the proofs is shown.

no.	level	$\|\|\cdot\|\|_i$	$\|\|\cdot\|\|_f$	formula	lemmas
C1	1	1	1	$\forall X : X = X$	
C2	1	5	2	$\forall X : (X \subseteq X)$	
C3	1	2	2	$\exists X : \forall Z : \neg(Z \in X)$	
C4	2	6	2	$\forall X : (\emptyset \subseteq X)$	L18
C5	2	16	9	$(\emptyset \in \mathcal{U})$	L5
C6	2	4	3	$\forall X,Y : ((Y \in \{X\}) \Rightarrow Y = X)$	L14
C7	1	3	3	$\forall Z : (Z = \emptyset \vee \exists Y : (Y \in Z))$	
C8	1	2	2	$\forall X : (\{X\} \in \mathcal{U})$	
C9	2	11	4	$\forall X : ((X \subseteq \emptyset) \Rightarrow X = \emptyset)$	L18 L11
C10	2	8	2	$\forall X,Y : (\{X\} \in \langle X,Y\rangle)$	
C11	3	19	4	$\forall X,Y : ((X \in Y) \Rightarrow (\{X\} \subseteq Y))$	L1
C12	1	9	9	$\forall X,Y : \neg(Y \in (\overline{X} \cap X))$	
C13	1	2	2	$\forall X,Y : (\langle X,Y\rangle \in \mathcal{U})$	
C14	1	11	11	$\forall X,Y,Z : (((X \subseteq Y) \wedge (Y \subseteq Z)) \Rightarrow (X \subseteq Z))$	
C15	2	7	3	$\forall X : ((X \in \mathcal{U}) \Rightarrow (X \in \{X\}))$	L12
C16	1	22	21	$\forall X,Y : (\{X,X\} \subseteq \{X,Y\})$	
C17	2	9	4	$\forall X : ((X \in \mathcal{U}) \Rightarrow \{X\} \neq \emptyset)$	L12
C18	2	21	3	$\forall X : (\neg(X \in \mathcal{U}) \Rightarrow \{X\} = \emptyset)$	L8
C19	1	3	3	$\forall X : (\{member\_of(X)\} = X \Rightarrow (X \in \mathcal{U}))$	
C20	2	7	2	$\forall X,Y : (\{X,\{Y\}\} \in \langle X,Y\rangle)$	
C21	1	10	10	$\forall X,Y : ((\{member\_of(X)\} = X \wedge (Y \in X)) \Rightarrow member\_of(X) = Y)$	
C22	6	87	25	$\forall X,Y : ((X \subseteq \{Y\}) \Rightarrow (X = \emptyset \vee \{Y\} = X))$	L16
C23	1	19	10	$\forall X,Y : ((\{X\} = \{Y\} \wedge (Y \in \mathcal{U})) \Rightarrow X = Y)$	
C24	3	15	10	$\forall X,Y : ((\{X\} = \{Y\} \wedge (X \in \mathcal{U})) \Rightarrow X = Y)$	L12 L17
C25	2	12	4	$\forall X,Y : ((X \in \mathcal{U}) \Rightarrow \{X,Y\} \neq \emptyset)$	L9
C26	2	9	4	$\forall X,Y : ((Y \in \mathcal{U}) \Rightarrow \{X,Y\} \neq \emptyset)$	L12
C27	1	3	3	$\forall X : (\langle X_{[1]},X_{[2]}\rangle = X \Rightarrow (X \in \mathcal{U}))$	
C28	6	610	7	$\forall X,Y,Z : (((X \in Z) \wedge (Y \in Z)) \Rightarrow (\{X,Y\} \subseteq Z))$	L1 L2
C30	8	706	213	$\forall W,X,Y,Z : ((\langle W,X\rangle = \langle Y,Z\rangle \wedge (X \in \mathcal{U})) \Rightarrow X = Z)$	L3 L10 L5 L17 L9
C31	2	20	12	$\forall W,X,Y,Z : ((\langle W,X\rangle = \langle Y,Z\rangle \wedge (W \in \mathcal{U})) \Rightarrow W = Y)$	L14 L9
C32	2	32	4	$\forall X,Y : ((\neg(X \in \mathcal{U}) \wedge \neg(Y \in \mathcal{U})) \Rightarrow \{X,Y\} = \emptyset)$	L8
C33	2	16	11	$\forall X,Y,Z : (((Y \in \mathcal{U}) \wedge (Z \in \mathcal{U}) \wedge \{X,Y\} = \{X,Z\}) \Rightarrow Y = Z)$	L12 L14
C34	2	16	11	$\forall X,Y,Z : (((X \in \mathcal{U}) \wedge (Y \in \mathcal{U}) \wedge \{X,Z\} = \{Y,Z\}) \Rightarrow X = Y)$	L14 L9
C35	3	21	4	$\forall X,Y : (\{\{X\},\{X,\emptyset\}\} = \langle X,Y\rangle \vee (Y \in \mathcal{U}))$	L10

Table 3: Selected conjectures from the TPTP set theory sample with the proof cost measured by the number of clauses processed by the prover.

no.	level	$\|\|\cdot\|\|$	formula	lemmas
L1	4	32	$(A \in C) \wedge (B \in C) \Rightarrow (\{A,B\} \subseteq C)$	L7
L2	5	4	$(A \in C) \Rightarrow (\{A,B\} \subseteq C) \vee (B \in \overline{C})$	
L3	7	5	$(B \in D) \wedge (A \in C) \Rightarrow (\{\{A,A\},\{A,\{B,B\}\}\} \in (C \times D))$	L6
L4	3	6	$(A \in C) \wedge (A \in B) \Rightarrow (A \in (B \cap C))$	L7
L5	1	8	$(A \in B) \Rightarrow (A \in \mathcal{U})$	
L6	6	7	$(B \in D) \wedge (A \in C) \Rightarrow (\langle A,B\rangle \in (C \times D))$	L7
L7	2	8	$(B \in C) \Rightarrow (A \in \mathcal{U}) \vee (\{B,A\} \subseteq C)$	L13 L5
L8	1	32	$\emptyset = \{A,B\} \vee (A \in \mathcal{U}) \vee (B \in \mathcal{U})$	
L9	1	8	$(A \in \mathcal{U}) \Rightarrow (A \in \{A,B\})$	
L10	2	3	$\emptyset = \{A,A\} \vee (A \in \mathcal{U})$	L8
L11	1	6	$(B \subseteq A) \wedge (A \subseteq B) \Rightarrow A = B$	
L12	1	7	$(A \in \mathcal{U}) \Rightarrow (A \in \{B,A\})$	
L13	1	31	$(A \in C) \Rightarrow (\{A,B\} \subseteq C) \vee (B \in \{A,B\})$	
L14	1	5	$(A \in \{C,B\}) \Rightarrow A = B \vee A = C$	
L15	1	2	$(A \subseteq \mathcal{U})$	
L16	5	4	$(A \subseteq \{B,C\}) \wedge (C \in A) \wedge (B \in A) \Rightarrow A = \{B,C\}$	
L17	2	4	$(A \in \mathcal{U}) \wedge A = B \Rightarrow (A \in \{B,C\})$	L9
L18	1	6	$(\emptyset \subseteq A)$	

Table 4: Lemmas found for the TPTP set theory sample with the proof cost measured by the number of clauses processed by the prover.

no.	level	$\|\|\cdot\|\|$	formula	lemmas
L1	1	37	$(A \in \{C,B\}) \Rightarrow A = B \vee A = C$	
L2	2	64	$\emptyset = \{A,B\} \vee (A \in \mathcal{U}) \vee (B \in \mathcal{U})$	L1
L3	2	43	$(A \in C) \wedge (B \in C) \Rightarrow (\{A,B\} \subseteq C)$	L1
L4	1	24	$(A \in \mathcal{U}) \Rightarrow (A \in \{B,A\})$	
L5	1	24	$(A \in \mathcal{U}) \Rightarrow (A \in \{A,B\})$	
L6	3	28	$\emptyset = \{A,A\} \vee (A \in \{A,A\})$	L2 L4
L7	3	44	$(A \in \mathcal{U}) \Rightarrow (A \in \overline{B}) \vee (A \in B)$	L2
L8	1	34	$(A \in (B \cap C)) \wedge (A \in \overline{C}) \Rightarrow \mathit{false}$	
L9	1	23	$\{A\} = \{A,A\}$	
L10	1	21	$(A \in B) \Rightarrow (A \in \mathcal{U})$	
L11	1	11	$(A \in \emptyset) \Rightarrow \mathit{false}$	
L12	1	18	$(\{A,B\} \in \mathcal{U})$	
L13	2	75	$\langle A,B\rangle = \{\{A\},\{A,\{B\}\}\}$	L9
L14	1	31	$((A \cap B) \subseteq B)$	
L15	2	42	$(A \in \mathcal{U}) \vee (\{A,A\} \subseteq B)$	L1 L10

Table 5: Lemmas found for the TPTP set theory sample with the proof cost measured by the proof size.

no.	level	$\|\|\cdot\|\|$	formula	lemmas
L1	2	19	$\emptyset = \{A,B\} \vee (A \in \mathcal{U}) \vee (B \in \mathcal{U})$	L2
L2	1	15	$(A \in \{C,B\}) \Rightarrow A = B \vee A = C$	
L3	3	16	$(A \in \mathcal{U}) \Rightarrow (A \in \overline{B}) \vee (A \in B)$	L1
L4	4	20	$(A \in B) \Rightarrow (\{A,A\} \subseteq B)$	L2
L5	1	8	$\{A\} = \{A,A\}$	
L6	1	11	$(A \in \mathcal{U}) \Rightarrow (A \in \{A,B\})$	
L7	1	11	$(A \in \mathcal{U}) \Rightarrow (A \in \{B,A\})$	
L8	1	10	$(A \in \emptyset) \Rightarrow \mathit{false}$	
L9	1	16	$(B \subseteq A) \wedge (A \subseteq B) \Rightarrow A = B$	
L10	2	10	$\langle A,B\rangle = \{\{A\},\{A,\{B\}\}\}$	L5
L11	1	9	$(\{A,B\} \in \mathcal{U})$	
L12	1	15	$(A \in C) \wedge (A \in B) \Rightarrow (A \in (B \cap C))$	
L13	1	16	$(A \in B) \Rightarrow (A \in \mathcal{U})$	

Table 6: Lemmas found for the TPTP set theory sample with the proof cost measured by the proof length.

proof measure	initial cost	final cost
number of processed clauses	3991	1921 (48%)
proof size	3487	2794 (80%)
proof length	1086	878 (81%)

Table 7: Results for the TPTP set theory sample.

6.2 Mizar problems – Boolean properties of sets

These theorems address basic boolean properties of sets in the Mizar database for mathematics. The conjectures were converted from the Mizar language by Josef Urban [Urb04, Urb03] into a form suitable for automated theorem provers.

The system was rather effective for reducing the number of processed clauses required to prove these set of conjectures. Table 8 shows the summary of achieved results. For each proof measure the initial and the final cost of the proofs is shown.

proof measure	initial cost	final cost
number of processed clauses	62706	1852 (3.0%)
proof size	10680	9351 (88%)
proof length	3687	3046 (83%)

Table 8: Results for the Mizar boolean properties of sets.

As the list of the conjectures and the lemmas in this case is rather long, we did not include it in this paper.

6.3 Meredith's axiomatization of propositional logic

In this section we give detailed results for the example in section 4. The formulas are presented using standard logic symbols.

Table 9 shows the conjectures along with the results obtained when running the system with the proofs being measured by the number of processed clauses and Table 10 shows the lemmas that were found.

no.	level	$\|\|\cdot\|\|_i$	$\|\|\cdot\|\|_f$	formula	lemmas
C1	8	15	2	$\vdash (A \to A)$	L27
C2	11	3682	2	$\vdash (A \to \neg\neg A)$	
C3	7	3297	2	$\vdash (\neg\neg A \to A)$	
C4	2	17	2	$\vdash (A \to (B \to A))$	
C5	9	5731	2	$\vdash (\neg A \to (A \to B))$	
C6	8	41	5	$\neg(A \to A) \vdash B$	L27
C7	5	31	3	$A \vdash (\neg A \to B)$	
C8	3	13	3	$B \vdash (\neg A \to B)$	L18
C9	11	1673	16	$\neg(A \to \neg B) \vdash B$	L22 L8 L4
C10	7	938	62	$\neg(A \to \neg B) \vdash A$	L17 L9 L16 L30 L18 L4
C11	12	287	20	$(P \to \neg(Q \to \neg R)), P \vdash R$	L21 L30 L8 L4

Table 9: Conjectures from the Meredith's axiomatization example with the proof cost measured by the number of clauses processed by the prover.

Because we code propositional formulas into terms of predicate logic, we can express much more than just that a propositional formula is a theorem. We can also express meta-theorems that speak about provability of different formulas and what are the relations between them. For example, recall how that modus ponens rule was coded as $(t(A) \wedge t(i(A,B))) \Rightarrow t(B)$. The system derived many lemmas of similar nature, thus discovering many admissible rules. This fact becomes much more interesting in the case of modal logic, described in the next section, where the deduction theorem does not hold, hence the admissible rules have much greater importance.

no.	level	$\|\|\cdot\|\|$	formula	lemmas
L1	6	7	$C \vdash (A \to (\neg\neg B \to B))$	L14 L4
L2	6	24	$D \vdash (A \to (B \to (\neg\neg C \to C)))$	L12 L4 L3
L3	3	8	$(((D \to B) \to (E \to B)) \to (B \to C)) \vdash (A \to (B \to C))$	L23 L4
L4	1	4	$A, (A \to B) \vdash B$	
L5	3	5	$A \vdash ((A \to B) \to (C \to B))$	L18 L12
L6	8	6	$((((B \to C) \to (\neg D \to \neg A)) \to D) \to B) \vdash (A \to B)$	L27 L12
L7	7	4	$\vdash ((((\neg A \to \neg B) \to A) \to C) \to (B \to C))$	L12
L8	8	6	$C \vdash (A \to ((\neg B \to \neg A) \to B))$	
L9	5	28	$((C \to E) \to (\neg B \to \neg D)) \vdash (((A \to B) \to C) \to (D \to C))$	L18 L4 L19
L10	8	7	$(((\neg C \to \neg A) \to C) \to B) \vdash (A \to B)$	L7 L4
L11	8	4	$\vdash (((A \to B) \to \neg(\neg B \to \neg C)) \to (C \to \neg(\neg B \to \neg C)))$	L7 L12
L12	1	6	$((((B \to D) \to (\neg E \to \neg C)) \to E) \to A) \vdash ((A \to B) \to (C \to B))$	
L13	6	13	$(((B \to C) \to D) \to (\neg C \to \neg A)) \vdash (A \to (B \to C))$	L9 L12 L4
L14	5	4	$\vdash (A \to (B \to (\neg\neg C \to C)))$	L2
L15	10	4	$B \vdash (A \to \neg\neg A)$	
L16	2	7	$D, C \vdash (A \to (B \to C))$	L18
L17	5	7	$((C \to C) \to B) \vdash (A \to B)$	L31
L18	2	6	$B \vdash (A \to B)$	
L19	4	7	$C, A \vdash (\neg A \to B)$	
L20	6	6	$\vdash (((A \to (B \to B)) \to C) \to (D \to C))$	L17 L31 L12
L21	11	4	$B \vdash (A \to \neg\neg A)$	L15
L22	6	14	$((C \to D) \to B), (\neg D \to \neg A) \vdash (A \to B)$	L9 L18 L4
L23	2	4	$\vdash ((((A \to B) \to (C \to B)) \to (B \to D)) \to (E \to (B \to D)))$	L12
L24	6	4	$\vdash (((A \to \neg\neg B) \to C) \to (B \to C))$	L12
L25	3	4	$\vdash (((A \to (\neg B \to C)) \to D) \to (B \to D))$	L23 L12
L26	3	11	$(D \to C), D \vdash (A \to (B \to C))$	L18 L4
L27	7	4	$\vdash (A \to A)$	L32
L28	1	6	$C \vdash (A \to (B \to A))$	
L29	7	3	$C \vdash (A \to (B \to B))$	
L30	6	5	$(\neg B \to \neg D) \vdash (((A \to B) \to C) \to (D \to C))$	L9 L18
L31	4	4	$\vdash (((A \to A) \to B) \to (C \to B))$	L25 L12
L32	6	4	$\vdash (A \to (B \to (C \to C)))$	L17 L31
L33	9	2	$\vdash (((A \to B) \to \neg\neg C) \to (C \to \neg\neg C))$	L9 L13 L11

Table 10: Lemmas found for the Meredith's axiomatization with the proof cost measured by the number of clauses processed by the prover.

proof measure	initial cost	final cost
number of processed clauses	15725	510 (3.2%)
proof size	1299	1024 (79%)
proof length	314	277 (88%)

Table 11: Results for the Meredith's axiomatization of propositional logic.

Table 11 shows the summary of achieved results. Again, for each proof measure the initial and the final cost of the proofs is shown.

6.4 S5 modal logic

S5 modal logic uses meta-theorems (mentioned above), since the theorem of deduction doesn't hold in S5. From this point of view S5 is an interesting example.

This set of conjectures is similar to the previous example. We have used [Hal05] to construct an axiomatization for S5 modal logic with the three Hilbert's axioms for propositional logic, axioms K, T and 5 and modus ponens and necessitation rule. We have used the same formula coding with an additional unary function symbol $l(\ldots)$ for the modal operator $\Box$.

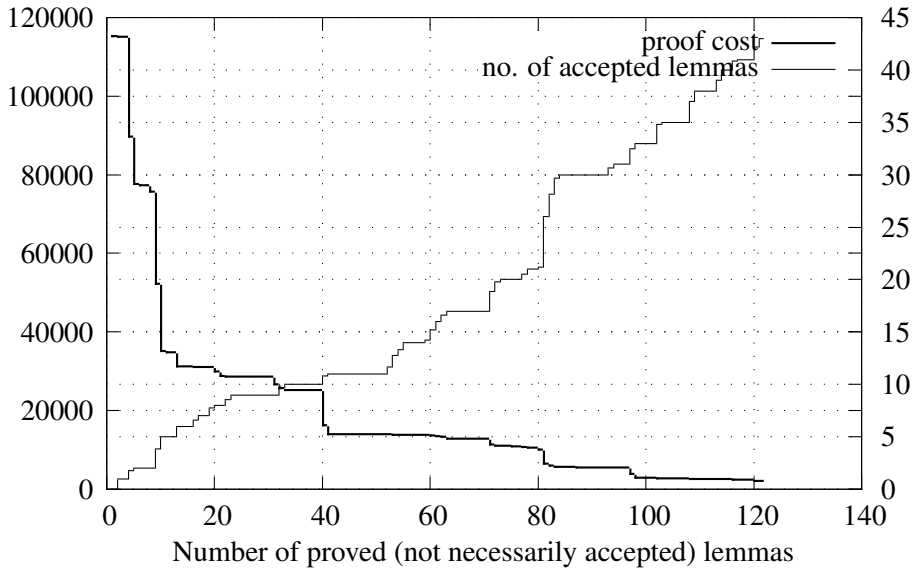

Figure 1: Run of the system on the S5 axiomatization with with the proof cost measured by the number of clauses processed by the prover.

For this example we also show a graph that illustrates performance of the system, see Figure 1. The x axis shows time points distinguished by the total number of lemmas (both accepted and unaccepted) the system has used at least in one proof. The thick line shows how the total cost of the proofs evolved and corresponds to the tick marks on the left. The thin line shows the number of lemmas that were marked as accepted and corresponds to the tick marks on the right. As we can see, the cost of the proofs was reduced to about 1/3 with the first 10 lemmas. The cost of the proofs then gradually decreased and the lemmas that were accepted often brought only a slight gain. There were two more significant improvements at the points 40, 81 and 97, when interesting lemmas were discovered and sudden advancements were made.

Most of the the lemmas that were discovered say that a particular proposition is a theorem of S5. But the system also discovered several lemmas that describe admissible rules of S5. For example the lemma L40 in Table 15 states that from $\Box B$ and $B \rightarrow A$ we can derive $\Box A$.

Table 16 shows the summary of achieved results. For each proof measure the initial and the final cost of the proofs is shown.

no.	level	$\|\|\cdot\|\|_i$	$\|\|\cdot\|\|_f$	formula	lemmas
C1	6	11	2	$\vdash (A \to A)$	L19
C2	1	2	2	$\vdash (\Box P \to P)$	
C3	9	1086	2	$\vdash (A \to \neg\neg A)$	
C4	2	5	3	$\vdash A \vee \neg\Box A$	L43
C5	1	3	3	$\vdash \neg A \vee \Box A$	
C6	9	1089	2	$\vdash (\neg\neg A \to A)$	
C7	7	3586	2	$\vdash ((\neg A \to A) \to A)$	
C8	7	44	2	$\vdash (\neg A \to (A \to B))$	L25
C9	8	680	15	$\vdash \Box(A \to \neg\Box\neg A)$	L37 L2 L36 L33
C10	5	15332	72	$\vdash (P \to \Box\neg\Box\neg P)$	L17 L2 L1
C11	6	43	5	$\neg(A \to A) \vdash B$	L19
C12	8	58	3	$A \vdash (\neg A \to B)$	
C13	2	5	3	$B \vdash (\neg A \to B)$	L10
C14	6	90	10	$\neg(A \to \neg B) \vdash B$	
C15	7	1293	6	$\neg(A \to \neg B) \vdash A$	L20 L23
C16	8	30413	2	$\vdash ((A \to B) \to (\neg B \to \neg A))$	
C17	9	23254	3	$\vdash (A \to (\neg B \to \neg(A \to B)))$	L19
C18	10	9580	4	$A, B \vdash \neg(A \to \neg B)$	L14
C19	1	3	3	$\vdash \Box(\neg\Box\neg P \to \Box\neg\Box\neg P)$	
C20	7	5471	171	$\neg(B \to \neg A) \vdash \neg(A \to \neg B)$	L28 L2 L13 L22 L1 L42 L9 L24
C21	1	8	6	$\Box(A \to B), \Box(B \to A) \vdash \Box(\Box A \to \Box B)$	
C22	10	23271	26	$\neg(A \to \neg\neg(B \to \neg C)) \vdash \neg(\neg(A \to \neg B) \to \neg C)$	L20 L23 L14

Table 12: Conjectures in the S5 axiomatization, prover processed clauses count measure

no.	level	$\|\|\cdot\|\|$	formula	lemmas
L1	2	59	$(A \to C), (A \to (C \to B)) \vdash (A \to B)$	L4
L2	3	40	$(A \to C), (C \to B) \vdash (A \to B)$	L1
L3	3	40	$C, (A \to (C \to B)) \vdash (A \to B)$	L1
L4	1	39	$\Box\Box B, (B \to A) \vdash \Box A$	
L5	5	25	$\vdash (\neg\neg A \to A)$	L3
L6	6	44	$(\neg B \to A) \vdash (\neg A \to B)$	
L7	4	59	$(A \to (\neg C \to \neg B)) \vdash (A \to (B \to C))$	L2
L8	1	33	$(\neg A \to \neg B), B \vdash A$	

Table 13: Lemmas found for the S5 axiomatization with the proof cost measured by the proof size.

no.	level	$\|\|\cdot\|\|$	formula	lemmas
L1	1	13	$(A \to C), (A \to (C \to B)) \vdash (A \to B)$	
L2	2	12	$(A \to C), (C \to B) \vdash (A \to B)$	
L3	2	13	$C, (A \to (C \to B)) \vdash (A \to B)$	L1
L4	4	10	$(\neg B \to A) \vdash (\neg A \to B)$	L7
L5	3	8	$\vdash (\neg\neg A \to A)$	
L6	1	13	$\Box\Box B, (B \to A) \vdash \Box A$	
L7	1	10	$(\neg B \to \neg A) \vdash (A \to B)$	

Table 14: Lemmas found for the S5 axiomatization with the proof cost measured by the proof length.

no.	level	$\|\|\cdot\|\|$	formula	lemmas
L1	3	8	$(A \to C), (A \to (C \to B)) \vdash (A \to B)$	L32
L2	4	11	$(A \to C), (C \to B) \vdash (A \to B)$	
L3	1	9	$(B \to (C \to D)) \vdash (A \to ((B \to C) \to (B \to D)))$	
L4	6	8	$C, (C \to B) \vdash (A \to B)$	L24
L5	2	11	$\Box\Box B, (B \to A) \vdash \Box\Box A$	L43
L6	8	5	$A \vdash (\neg\neg(A \to B) \to B)$	L13
L7	7	247	$\vdash (\neg A \to ((B \to A) \to \neg B))$	L8 L25 L1 L15
L8	5	30	$((C \to A) \to ((C \to B) \to D)) \vdash ((A \to B) \to ((C \to A) \to D))$	L3 L31 L1
L9	5	11	$(A \to (B \to D)), (B \to (D \to C)) \vdash (A \to (B \to C))$	L2 L32
L10	1	6	$B \vdash (A \to B)$	
L11	3	21	$((C \to A) \to B) \vdash (A \to B)$	L10 L32 L15
L12	6	10	$D, (D \to C) \vdash (A \to (B \to C))$	L10
L13	4	8	$(A \to (C \to B)), C \vdash (A \to B)$	L10 L1
L14	9	6	$B, A \vdash \neg(A \to \neg B)$	L20 L6
L15	1	4	$B, (B \to A) \vdash A$	
L16	1	5	$A, B \vdash \Box\Box A$	
L17	4	5	$(B \to (A \to C)) \vdash (A \to (B \to C))$	L11 L32
L18	6	5	$((A \to B) \to A) \vdash ((A \to B) \to B)$	L19 L1
L19	5	4	$\vdash (A \to A)$	
L20	2	7	$(\neg A \to \neg B), B \vdash A$	L37 L15
L21	6	7	$((C \to C) \to B) \vdash (A \to B)$	L19
L22	5	9	$(A \to (C \to B)), ((C \to B) \to C) \vdash (A \to B)$	L2 L1
L23	6	11	$\neg(A \to C) \vdash (\neg A \to B)$	L29 L24
L24	5	10	$(A \to C), \neg C \vdash (A \to B)$	L28 L2
L25	6	4	$\vdash (\neg A \to (A \to B))$	L29
L26	6	32	$(C \to (A \to D)) \vdash (A \to (B \to (C \to D)))$	L10 L9
L27	2	7	$\Box B, (B \to A) \vdash A$	L43
L28	1	4	$\neg A \vdash (A \to B)$	
L29	5	6	$(A \to (\neg C \to \neg B)) \vdash (A \to (B \to C))$	L2
L30	6	5	$(\neg A \to (B \to \neg C)) \vdash ((\neg A \to B) \to (C \to A))$	L29 L32
L31	4	4	$((A \to (B \to C)) \to (((A \to B) \to (A \to C)) \to D)) \vdash ((A \to (B \to C)) \to D)$	L1
L32	2	7	$(A \to (B \to C)) \vdash ((A \to B) \to (A \to C))$	L15
L33	7	4	$\vdash (\neg\neg A \to (B \to A))$	L29 L25
L34	8	4	$\vdash (A \to (\neg\neg B \to B))$	L17 L33
L35	1	9	$(\neg C \to \neg B) \vdash (A \to (B \to C))$	
L36	4	4	$(A \to ((C \to A) \to B)) \vdash (A \to B)$	L1
L37	1	6	$(\neg B \to \neg A) \vdash (A \to B)$	
L38	8	6	$(A \to (\neg B \to (C \to B))) \vdash (A \to (\neg B \to \neg C))$	L7 L9
L39	7	6	$(\neg A \to A) \vdash (\neg A \to B)$	L25 L1
L40	2	9	$\Box B, (B \to A) \vdash \Box A$	L43
L41	8	6	$\vdash (A \to (B \to \neg\neg B))$	L17 L29 L33
L42	6	11	$(A \to D), (D \to C) \vdash (A \to (B \to C))$	L2 L24
L43	1	5	$\Box A \vdash A$	

Table 15: Lemmas found for the S5 axiomatization with the proof cost measured by the number of clauses processed by the prover.

proof measure	initial cost	final cost
number of processed clauses	115327	2091 (1.8%)
proof size	632	379 (60%)
proof length	642	389 (61%)

Table 16: Results for S5 modal logic.

The system again performed very well in the case when the measure was the number of processed clauses. This time, the total cost of the proofs was reduced to less than 2%.

7 Future work

As the system is particularly efficient in speeding up the prover, we believe that it could be modified to search for lemmas that would make it possible to prove conjectures that the prover alone wasn't able to prove. This will require a change of strategy, because currently the system looks primarily for lemmas that improve already existing proofs of the conjectures and therefore are not general enough to prove some new unknown conjecture.

We would also like to investigate the nature of the lemmas that help to improve particular proof measures in order to develop a better strategy for their evaluation.

Finally, we plan to perform a in-depth testing of the system on various sets of conjectures from different sources.

8 Conclusion

Given a related set of conjectures, it is possible to automatically construct lemmas that can significantly reduce the cost of the proofs of the conjectures. The results are summarized in Table 17 for convenience.

	processed clauses		proof size		proof length	
set of conjectures	$\|\|\cdot\|\|_i$	$\|\|\cdot\|\|_f$	$\|\|\cdot\|\|_i$	$\|\|\cdot\|\|_f$	$\|\|\cdot\|\|_i$	$\|\|\cdot\|\|_f$
TPTP set theory	3991	1921 (48%)	3487	2794 (80%)	1086	878 (81%)
Mizar set properties	62706	1852 (3.0%)	10680	9351 (88%)	3687	3046 (83%)
Meredith's axiomatization	15725	510 (3.2%)	1299	1024 (79%)	314	277 (88%)
S5 modal logic	115327	2091 (1.8%)	632	379 (60%)	642	389 (61%)

Table 17: Summary of the results of the system on the presented sets of conjectures.

The system that we have developed performs well on different sets of conjectures, particularly if the cost of the proofs of the conjectures is measured in the number of clauses processed by the prover. The system can also improve the size and/or the length of the proofs, although it is not as effective in these cases.

References

[AS92] Owen L. Astrachan and Mark E. Stickel. Caching and lemmaizing in model elimination theorem provers. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 1992.

[Col02a] Simon Colton. *Automated Theory Formation in Pure Mathematics*. Distinguished Dissertations. Springer, 2002.

[Col02b] Simon Colton. The HR program for theorem generation. In Andrei Voronkov, editor, *CADE*, volume 2392 of *Lecture Notes in Computer Science*, pages 285–289. Springer, 2002.

[CP80] P. T. Cox and T. Pietrzykowski. *A complete, nonredundant algorithm for reversed skolemization*, volume 87 of *Lecture Notes in Computer Science*. Springer, May 1980.

[CP93] Ritu Chadha and David Plaisted. *Finding logical consequences using unskolemization*, volume 689 of *Lecture Notes in Computer Science*. Springer, May 1993.

[FFF99] Marc Fuchs, Dirk Fuchs, and Matthias Fuchs. Generating lemmas for tableau-based proof search using genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1027–1032, Orlando, Florida, USA, July 1999. Morgan Kaufmann.

[Hal05] John Halleck. Logic systems, 2005. http://www.cc.utah.edu/~nahaj/logic/structures/.

[McC94] W. W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, Illinois, 1994.

[MVF+02] William McCune, Robert Veroff, Branden Fitelson, Kenneth Harris, Andrew Feist, and Larry Wos. Short single axioms for boolean algebra. *J. Autom. Reasoning*, 29(1):1–16, 2002.

[Qua92] Art Quaife. *Automated Development of Fundamental Mathematical Theories*. Kluwer Academic Publishers, 1992.

[Sch02] S. Schulz. E – A brainiac theorem prover. *Journal of AI Communications*, 15(2-3):111–126, 2002.

[SGC03] G. Sutcliffe, Y. Gao, and S. Colton. A Grand Challenge of Theorem Discovery. In J. Gow, T. Walsh, S. Colton, and V. Sorge, editors, *Proceedings of the Workshop on Challenges and Novel Applications for Automated Reasoning, 19th International Conference on Automated Reasoning*, pages 1–11, 2003.

[SS98] G. Sutcliffe and C. B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[Urb03] Josef Urban. Translating Mizar for first order theorem provers. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 203–215. Springer, 2003.

[Urb04] Josef Urban. MPTP - motivation, implementation, first experiments. *Journal of Automated Reasoning*, 33(3-4):319–339, 2004.

[WP03] Larry Wos and Gail W. Pieper. *Automated Reasoning and the Discovery of Missing and Elegant Proofs*. Rinton Press, 2003.

Lightweight Relevance Filtering for Machine-Generated Resolution Problems

Jia Meng[1], Lawrence C. Paulson[2]
[1]National ICT, Australia
jiameng@nicta.com.au
[2]Computer Laboratory, University of Cambridge, U.K.
LP15@cam.ac.uk

Abstract

Irrelevant clauses in resolution problems increase the search space, making it hard to find proofs in a reasonable time. Simple relevance filtering methods, based on counting function symbols in clauses, improve the success rate for a variety of automatic theorem provers and with various initial settings. We have designed these techniques as part of a project to link automatic theorem provers to the interactive theorem prover Isabelle. They should be applicable to other situations where the resolution problems are produced mechanically and where completeness is less important than achieving a high success rate with limited processor time.

1 Introduction

Resolution-based automatic theorem provers (ATPs) are *complete* for first-order logic: given unlimited resources, they will find a proof if one exists. Unfortunately, we seldom have unlimited resources. Removing unnecessary axioms from the problem reduces the search space and thus the effort needed to obtain proofs. Identifying unnecessary axioms while retaining completeness appears to be as difficult as proving the theorem in the first place. In general, it is hard to see how we can know that an axiom is redundant except by finding a proof without using it. Syntactic criteria may be helpful when the amount of redundancy is extreme, especially if we can relax the requirement for completeness.

The relevance problem dates back to the earliest days of resolution. As first defined by Robinson [Rob65], a literal is *pure* if it is not unifiable with a complementary literal in any other clause. Clauses containing pure literals can be removed without affecting consistency. This process is a form of relevance test, since pure literals often indicate that the axioms describe predicates not mentioned in the (negated) conjecture. It can be effective, but it is not a full solution.

If a resolution theorem prover is invoked by another reasoning tool, then the problems it receives will have been produced mechanically. Machine-generated problems may contain thousands of clauses, each containing large terms. Many ATPs are not designed to cope with such problems. Traditionally, the ATP user prepares a mathematical problem with the greatest of care, and is willing to spend weeks running proof attempts, adjusting weights and refining settings until the problem is solved. Machine-generated

problems are not merely huge but may be presented to the automatic prover by the dozen, with its heuristics set to their defaults and with a small time limit.

Our integration of Isabelle with ATPs [MQP06] generates small or large problems, depending on whether or not rewrite rules are included. A small problem occupies 200 kilobytes and comprises over 1300 clauses; a large problem occupies about 450 kilobytes and comprises 2500 clauses, approximately. Even our small problems look rather large to a resolution prover. We have run extensive tests with a set of 285 such problems (153 small, 132 large). Vampire does well on our problem set, if it is given 300 seconds per problem: it can prove 86 percent of them. Unfortunately, given 30 seconds per problem, which is more realistic for user interaction, Vampire's success rate drops to 53 percent. (See Fig. 7 in Sect. 5.) Can a cheap, simple relevance filter improve the success rate for short runtimes? Completeness does not matter to us: we are happy to forgo solutions to some problems provided we obtain a high success rate.

In the course of our investigations, we found that many obvious ideas were incorrect. For example, ATPs generate hundreds of thousands of clauses during their operation, so an extra fifty clauses at the start should not do any harm. However, they do.

Paper outline. We begin with background material on Isabelle, ATPs, and our link-up between the two, mentioning related work on other such link-ups (Sect. 2). We describe our initial attempts to improve the success rate of our link-up (Sect. 3), and then describe a family of related relevance filters (Sect. 4). We proceed to our experimental results, illustrating the benefits of filtering through a series of graphs (Sect. 5). Finally, we present brief conclusions (Sect. 6).

2 Background

Resolution theorem provers work by deriving a contradiction from a supplied set of *clauses* [BG01]. Each clause is a disjunction of *literals* (atomic formulae and their negations) and the set of clauses is interpreted as a conjunction. Clause form can be difficult to read, and the proofs that are found tend to be unintuitive, but there is no denying that these provers are powerful. In the sequel we refer to them as automatic theorem provers or ATPs. (This term includes clausal tableau provers, but not SAT solvers, decision procedures etc.) Our experiments mainly use E versions 0.9 and 0.91dev001 [Sch04], SPASS V2.2 [Wei01] and Vampire 8 [RV01a].[1]

Interactive theorem provers allow proofs to be constructed by natural chains of reasoning, generally in a rich formalism such as higher-order logic, but their automation tends to be limited to rewriting and arithmetic. Quantifier reasoning tends to be weak: many interactive systems cannot even prove a simple theorem like $\exists x\, \forall y\, P(x, y) \rightarrow \forall y\, \exists x\, P(x, y)$ automatically. Developers of interactive tools would naturally like to harness the power of ATPs, while preserving the intuitive reasoning style that their users expect.

We are adding automated support to the interactive prover Isabelle. There are many differences between our project and other related work [BHdN02, SBF$^+$03]. The chief difference is that Isabelle's built-in automated support gives the ATPs tough competi-

[1] We downloaded Vampire from the CASC-20 website, http://www.cs.miami.edu/ tptp/CASC/20/; it calls itself version 7.45.

tion. By typing *auto*, the Isabelle user causes approximately 2000 lemmas to be used as rewriting rules and for forward and backward chaining. A related tool, *blast*, performs deep searches in a fashion inspired by tableau provers. Even ten years ago, using early predecessors of these tools, Isabelle users could automatically prove theorems like this set equality [Pau97]:

$$\bigcup_{i\in I}(A_i \cup B_i) = \Big(\bigcup_{i\in I} A_i\Big) \cup \Big(\bigcup_{i\in I} B_i\Big)$$

Set theory problems, and the combinatory logic examples presented in that paper, remain difficult for automatic theorem provers. When we first got our link-up working, we were disappointed to find that ATPs were seldom better than *auto*. We have devoted much effort to improving its success rate. We found that bugs in our link-up were partly to blame for the poor results. Much of our effort went to improving the problem presentation; for example, we found a more compact representation of types. We devoted some time to identifying prover settings to help ATPs cope with huge problems. Above all, we have struggled to find ways to filter out irrelevant axioms.

Of previous work, the most pertinent is the integration between the Karlsruhe Interactive Verifier (KIV) and the tableau prover 3TAP, by Ahrendt and others [ABH+98]. Reif and Schellhorn [RS98] present a component of that integration: an algorithm for removing redundant axioms. It relies on analysing the structure of the theory in which the conjecture is posed. Specifically, their method is based on four criteria for reduction, which they call the minimality, structure, specification and recursion criteria. We did not feel that this method would work with Isabelle theories, which tend to be large: a typical theory introduces many functions and definitions, and proves many theorems. For instance, the theory of the natural numbers proves nearly 200 theorems. Also, a typical Isabelle problem lies at the top of a huge theory hierarchy that includes the full Isabelle/HOL development, any included library theories, and theories for the user's problem domain. We decided to try other methods, which are described in the sequel.

The Isabelle-ATP linkup generates problems that contain conjecture clauses along with clauses from four other sources:

- *Classical* clauses arise from the theorems Isabelle uses for forward and backward chaining.
- *Simplification* clauses arise from the theorems Isabelle uses as rewrite rules. They contain equalities.
- *Arity* and *class inclusion* clauses do not correspond to Isabelle theorems. Instead, they express aspects of Isabelle's type system. Isabelle's *axiomatic type classes* are sets of types that meet a given specification. For instance, the type `nat` of natural numbers is a member of `order`, the class of partial orderings; we express its membership as the unit clause $order(nat)$. An arity relates the type classes of the arguments and results of type constructors. For example, an arity clause

  $$\forall\, \tau[type(\tau) \rightarrow order(list(\tau))]$$

  says if the argument of `list` is a member of class `type`, then the resulting type of lists belong to class `order`. For more information, we refer readers to our previous papers [MP04, MQP06].

Although the arity and class inclusion clauses typically number over one thousand, they probably pose no difficulties for modern ATPs. They are Horn clauses that contain only monadic (unary) predicates, which are not related by any equations. Most arity clauses have one literal, while class inclusion clauses consist of two literals. Pure literal elimination probably suffices to remove redundant arity and class inclusion clauses, so ATPs may delete most of them immediately.

3 Initial Experiments

We have tuned the Isabelle-ATP link-up with the help of a set of problems in clause form. We obtained these by modifying our link-up to save the clause form problems it was producing. They simulate attempted calls to our system at various points in several Isabelle proofs. The original Isabelle proofs for some of these problems require multiple steps, explicit quantifier instantiations, or other detailed guidance. The set now numbers 285 problems. Many have trivial proofs and the only difficulty lies in the problem size. Other problems take a few minutes to prove, and a few remain unsolved.

Our first task was to verify that the problems were solvable. If a problem could not be proved by any ATP, we could remove irrelevant clauses manually, using our knowledge of the problem domain, in the hope of rendering it provable. This process was too laborious to carry out for every failing problem. Over time, with the help of the filtering techniques described below, we were able to obtain proofs for all but five of our problems. We also uncovered problems that were incorrectly posed, lacking crucial lemmas, and found several bugs. These ranged from the trivial (failing to notice that the original problem already contained the empty clause) to the subtle (Skolemization failing to take account of polymorphism).

The laborious hand-minimization can be automated. A simple idea is to note which axioms take part in any successful proofs—call them *referenced* axioms—and to remove all other axioms from the unsolved problems. We have automated this idea for the provers Vampire and E. Both clearly identify references to axiom clauses, which they designate by positive integers. Simple Perl scripts read the entire clause set into an array; referenced axioms are found by subscripting and written to a new file. We thus obtain a reduced version of the problem, containing only the clauses referenced. Repeating this process over a directory of problems yields a new directory containing reduced versions of each solved problem. If both Vampire and E prove a theorem, then the smaller file is chosen. We then concatenate the solutions, removing conjecture clauses. The result is a file containing all referenced axioms. Another Perl script intersects this file with each member of the problem set, yielding a reduced problem set where each problem contains only referenced axioms.

Auto-reduction by using only referenced axioms has an obvious drawback: some unsolved problems are likely to need some axioms that have not been referenced before. Even so, this idea improved our success rate from about 60 percent to 80 percent. It is not clear how to incorporate this idea into an interactive prover, since then its success on certain problems would depend upon the previous history of proof attempts, making the system's behaviour hard to predict. Auto-reduction's immediate benefit is that it yields evidence that the original problems have proofs: a reduced problem is easily checked to

be a subset of the original problem, and with few exceptions it is easy to prove. Fewer suspect problems require hand examination.

Using only referenced axioms does not guarantee that problems will be small. At present, there are 405 referenced clauses. Most of these are specific to various problem domains; approximately 150 of these correspond to standard Isabelle/HOL theorems, and are common to all problems. A proof about protocol verification could have another 90 clauses, for a problem size of about 240. Two points are noteworthy:

1. The problems are smaller if not small, and
2. referenced clauses may be somehow more suitable for automated proof than other clauses.

This second point suggests the concept of *pathological* clauses. Is it possible that there are a few particularly bad clauses whose removal would improve the success rate?

It is difficult to test the hypothesis that the success rate is lowered by a few pathological clauses. There is no reason to believe that the same clauses will turn out to be pathological for all ATPs. Identifying pathological clauses seems to require much guessing and manual inspection. Surely a pathological clause would contain highly general literals such as $X = Y$, $X < Y$, $X \in Y$, or their negations.

We eventually blacklisted 148 theorems from the standard Isabelle/HOL library. A theorem could be blacklisted for various reasons, such as having too big a clause form, being too similar to other theorems, or dealing with too obscure a property. This effort yielded only a small improvement to the success rate, probably because Isabelle's sets of classical and simplification rules already exclude obviously prolific facts such as transitivity. The main benefit of this exercise was our discovery that the generated problems included large numbers of functional reflexivity axioms: that is, axioms such as $X = Y \longrightarrow f(X) = f(Y)$. They are redundant in the presence of paramodulation; since we only use ATPs that use that inference rule for equality reasoning, we now omit such clauses in order to save ATPs the effort of discarding them. This aspect of our project was in part a response to SPASS developer Thomas Hillenbrand's insistence—in an e-mail dated 23 July 2005—on "engineering your clause base once and forever".

4 Signature-Based Relevance Filters

Automatic relevance filtering is clearly more attractive than any method requiring manual inspection of clauses. We decided not to adopt Reif and Schellhorn's approach [RS98], which required analysis of the theory in which each problem was set, and instead to define relevance with respect to the provided conjecture clauses. The simplest way of doing this is to enable the Set of Support option, if it is available. Wos's SOS heuristic [WRC65], which dates from 1965, ensures that all inference rule applications involve at least one clause derived from a conjecture clause. It prevents inferences among the axioms and makes the search goal-directed. It is incomplete in the presence of the ordering heuristics used by modern ATPs, but SPASS still offers SOS and it greatly improves the success rate, as the graphs presented below will demonstrate.

We tried many simple relevance filters based on the conjecture clauses. We already knew that simple methods could be effective: in an earlier experiment, we modified the

link-up to block all axiom clauses except those from a few key theories. That improved the success rate enough to yield proofs for eight hitherto unsolved problems. Clearly if such a crude filter could be beneficial, then something based on the conjecture clauses could be better still. Having automatically-reduced versions of most of our problems allows us to test the relevance filter without actually running proofs: yet more Perl scripts compare the new problems with the reduced ones, reporting any missing axioms.

The abstraction-based relevancy testing approach of Fuchs and Fuchs [FF99] is specifically designed for model elimination (or connection tableau) provers. It is not clear how to modify this approach for use with saturation provers, which are the type we use almost exclusively. Their approach has some curious features. Though it is based upon very general notions, the specific abstraction they implement is a *symbol abstraction*, which involves "identifying some predicate or function symbols" and forming equivalence classes of clauses. We confess that we were not able to derive any ideas from this highly mathematical paper.

4.1 Plaisted and Yahya's Strategy

Plaisted and Yahya's relevance restriction strategy [PY03] introduces the concept of *relevance distance* between two clauses, reflecting how closely two clauses are related. Simply put, the idea is to start with the conjecture clauses and to identify a set R_1 of clauses that contain complementary literals to at least one of the conjecture clauses. Each clause in R_1 has distance 1 to the conjecture clauses. The next round of iteration produces another set R_2 of clauses, where each of its clauses resolves with one or more clauses in R_1; thus clauses in R_2 have distance 2 to the conjecture clauses. The iteration repeats until all clauses that have distances less than or equal to some upper limit are included. This is an all-or-nothing approach: a clause is either included if it can resolve with some already-included clause, or, not included at all.

We found this method easy to implement (in Prolog), but unfortunately too many clauses are included after two or three iterations. This method does not take into account the ordering restrictions that ATPs would respect, thereby including clauses on the basis of literals that would not be selected for resolution. Also, this strategy does not handle equality.

Plaisted and Yahya's strategy suggests a simple approach based on signatures. Starting with the conjecture clauses, repeatedly add all other clauses that share any function symbols with them. This method handles equality, but it again includes too many clauses. Therefore, we have refined Plaisted and Yahya's strategy and designed several new algorithms that work well (Sect. 5).

4.2 A Passmark-Based Algorithm

Our filtering strategies abandon the all-or-nothing approach. Instead, we use a measure of relevance, and a clause is added to the pool of relevant clauses provided it is "sufficiently close" to an existing relevant clause. If a clause mentions n functions, of which m are relevant, then the clause receives a score (relevance mark) of m/n. The clause is rejected unless its score exceeds a given *pass mark*, a real number between 0 and 1. If a clause is accepted, all of its functions become relevant. Iterate this process until no

new clauses are accepted. To prevent too many clauses from being accepted, somehow the test must become stricter with each iteration.

In the first filtering strategy, we attach a relevance mark to each clause and this mark may be increased during the filtering process. The pseudo-code for our algorithm is shown in Figure 1.

The pseudo-code should be self-explanatory. We only give a few more comments below.

- When the function `relevant_clauses` is first called, the working relevant clauses set `W` contains the goal clauses, while `T` contains all the axiom clauses and `U` is empty.
- In function `find_clause_mark`, `|R|` is the number of elements in the set `R`.
- Isabelle allows overloading of functions, so that for example `<=` can denote the usual ordering on integers as well as subset inclusion. Therefore function `find_clause_mark` regards two functions as matching only if their types match as well.
- The multiplication by `P_M` in function `find_clause_mark` makes the relevance test increasingly strict as the distance from the conjecture clauses increases, which keeps the process focussed around the conjecture clauses and prevents too many clauses from being taken as relevant.

4.3 Using the Set of Relevant Functions

We have refined the strategy above, removing the requirement that a clause be close to one single relevant clause. It instead accumulates a pool of relevant functions, which is used when calculating scores. This strategy is slightly simpler to implement, because scores no longer have to be stored, and it potentially handles situations where a clause is related to another via multiple equalities. To make the relevance test stricter on successive iterations, we increase the pass mark after each successive iteration by the formula $p' = p + (1 - p)/c$, where c is an arbitrary convergence parameter. If $c = 2$, for example, then each iteration halves the difference $1 - p$. The algorithm appears in Figure 2.

Since the value of c is used to modify that of p, the optimal values of these parameters need to be found simultaneously. We ran extensive empirical tests. It became clear that large values of c performed poorly, so we concentrated on 1.6, 2.4 and 3.2 with a range of pass marks. We obtained the best results with $p = 0.6$ and $c = 2.4$. These values give a strict test that rapidly gets stricter, indicating that our problems require a drastic reduction in size.

To illustrate these points, Figure 3 presents two graphs. They plot success rates and problem sizes as the pass mark increases from 0.0 (all clauses accepted) to 0.9 (few clauses accepted). Success rates are for Vampire in its default mode, allowing 40 seconds per problem. Problem sizes refer to the average number of clauses per problem, ignoring conjecture clauses and the clauses that formalize Isabelle's type system. Vampire's success rate peaks sharply at 0.6, by which time the average problem size has decreased from 909 to 142 clauses. Since the aim of these experiments was to determine the best

```
function relevant_clauses:
input:  W, working relevant clauses set
        T, working irrelevant clauses set
        P, pass mark
        U, used relevant clauses set
        (Each clause is attached with a relevance mark)
output: relevant clauses to be input to ATPs
begin
while W not empty do {
   for each clause-relevance-mark pair (C,M) in T do
     { update_clause_mark (W, (C,M)) }
   #partition (C,M) pairs in T into two sets
   Rel set contains (C,M) if P ≤ M
   Irrel := T - Rel;
   U := W ∪ U;
   W := Rel;
   T := Irrel;
}
return U;
end


function update_clause_mark:
input:  W, relevant clauses set
        (C,M), a clause-mark pair
effect: updates the relevance mark of C
begin
for each clause-mark (P,P_M) in W do {
  PC := functions_of P;
  CF := functions_of C;
  R := CF ∩ PC;         #where names and types agree
  IR := CF − R;         #remaining functions of clause C
  M := max(M, P_M * |R| / (|R| + |IR|));
}
end
```

Figure 1: A Passmark-Based Filtering Strategy

```
function relevant_clauses:
input:  RF, set of relevant functions
        T, working irrelevant clauses set
        A, relevant clauses accumulator
        P, pass mark
output: relevant clauses to be input to ATPs
begin
repeat {
  for each clause C_i in T do
    { M_i := clause_mark (RF,C_i) }
  Rel := set of all clauses C_i such that P ≤ M_i
  T := T - Rel;
  A := A ∪ Rel;
  P := P + (1 - P) / c;
  RF := (functions_of Rel) ∪ RF;
} until Rel is empty
return A;
end

function clause_mark:
input:  RF, a set of relevant functions
        C, a clause
output: the relevance mark of C
begin
CF := functions_of C;
R := CF ∩ RF;          #where names and types agree
IR := CF − R;          #remaining functions of clause C
return |R| / (|R| + |IR|);
end
```

Figure 2: An Improved Filtering Strategy Using a Set of Relevant Functions

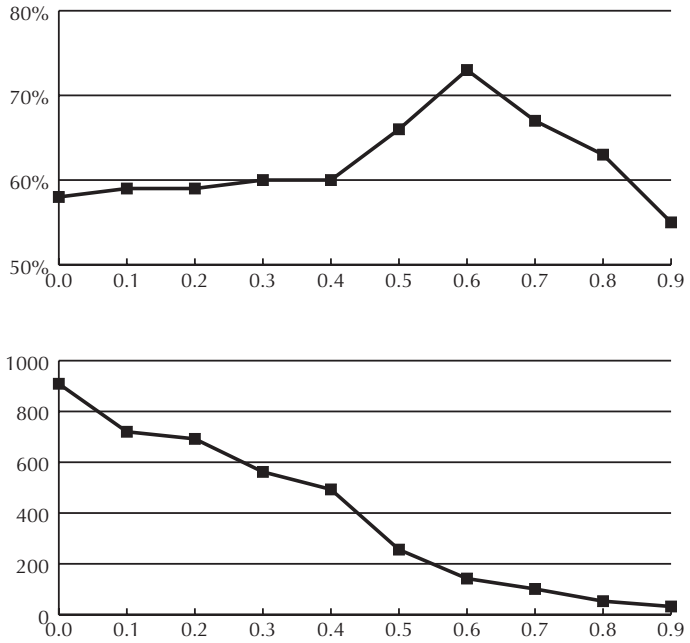

Figure 3: Success Rates and Problem Sizes Against Pass Marks

values for p and c, the choice of ATP probably makes little difference. In particular, if the filter has removed essential axioms, then no ATP will find a proof; this is why the success rate drops when $p > 0.6$. We repeat that these graphs are for illustration only; our parameter settings are based on tests that took hundreds of hours of processor time.

4.4 Taking Rarity into Account

Another refinement takes account of the relative frequencies of the functions in the clause set. Some functions are common while others are rare. An occurrence of a rare function would seem to be a strong indicator of relevance, while its very rarity would ensure that not too many new clauses are included. In the relevance quotient m/n, we boost m to take rarity into account. while leaving n to denote the total number of functions. Taking rarity into account in n would prevent the inclusion of clauses that involve another rare function, such as a Skolem function (each Skolem function is rare, since it only occurs in a few clauses).

This strategy requires a suitable frequency function, which calculates the weight of a symbol according to the number of its occurrences in the entire set of axiom clauses. It is similar to our strategy mentioned in Sect.4.3, except for the function `clause_mark` shown in Figure 4.

A few explanations may be helpful here.

- The relevance mark of a clause `C` calculated by `clause_mark` is not the percentage of relevant functions in `C` any more. Instead, we use `func_weight` function to compute the sum of relevant functions' marks weighted by their frequencies.

- A suitable frequency function (`frequency_function`) is needed. After much test-

```
function clause_mark:
input:  RF, a set of relevant functions
        C, a clause
        ftab, a table of the number of occurrences of each function in the clause set
output: the relevance mark of C
begin
CF := functions_of C;
R := CF ∩ RF;          #where names and types agree
IR := CF − R;          #remaining functions of C
M := 0;
for each function F in R do { M := M + func_weight (ftab, F) }
return (M / (M + |IR|));
end

function func_weight:
input:  ftab, a table of the number of occurrences of each function in the cause set
        F, a function symbol
output: the frequency weighted mark for F
begin
freq := number_of_occurrences (ftab, F);
return (frequency_function freq);
end
```

Figure 4: A Filtering Strategy for Rarely-Occurring Functions

ing we found that it should decrease gently as the frequency increases. If a function occurs n times in the problem, then we increase its contribution from 1 to $1 + 2/\log(n + 1)$. An even gentler decrease, such as $1 + 1.4/\log(\log(n + 2))$, can work well, but something like $1 + 1/\sqrt{n}$ penalizes frequently-occurring functions too much. Hence, our definition of `frequency_function` is

```
frequency_function n = 1 + 2/log(n+1)
```

4.5 Other Refinements

Hoping that unit clauses did not excessively increase resolution's search space, we experimented with adding all "sufficiently simple" unit clauses at the end of the procedure. A unit clause was simple provided it was not an equation; an equation was simple provided its left- or right-hand side was ground (that is, variable-free). We discovered that over 100 unit clauses were often being added, and that they could indeed increase the search space. By improving the relevance filter in other respects, we found that we could do without a special treatment of unit clauses. A number of attempts to favour shorter clauses failed to produce any increase in the success rate.

Definition expansion is another refinement. If a function f is relevant, and a unit clause such as $f(X) = t$ is available, then it can be regarded as relevant. To avoid including "definitions" like $0 = N \times 0$, we check that the variables of the right-hand side are a subset of those of the left-hand side. Definition expansion is beneficial, but its effect is small.

As of this writing, our system still contains a manually produced blacklist of 117 (down from 148) HOL theorems. We also have a whitelist of theorems whose inclusion is forced; it contains one single theorem (concerning the subset relation), which we found that the filter was frequently rejecting. We can probably reduce the blacklist drastically by checking which of those theorems would survive relevance filtering. However, having a high success rate is more important to us than having an elegant implementation.

5 Empirical Results

Extensive testing helped us determine which methods worked best and to find the best settings of the various parameters. The graphs compare the success rates of filtered problems against raw ones. The runtime per problem increases from 10 to 300 seconds. Success rates are plotted on a scale ranging from 40 to 90 percent. We tested the three provers we have found to be best—E, SPASS and Vampire—in their default mode and with alternative settings. E version 0.9 "Soom" is surpassed by version 0.91dev001 (Fig. 5), a development version of "Kanyam" that includes heuristics designed for our problem set. (Version 0.9 surpasses the official Kanyam release, E version 0.91, on our problems.) SPASS (Fig. 6) seems to perform better if SOS is enabled and splitting is disabled.[2] Vampire does extremely well in its CASC mode (Fig. 7).

We ran these tests on a bank of Dual AMD Opteron processors running at 2400MHz. The Condor system[3] managed our batch jobs.

[2] The precise option string is `-Splits=0 -FullRed=0 -SOS=1`.

[3] `http://www.cs.wisc.edu/condor/`

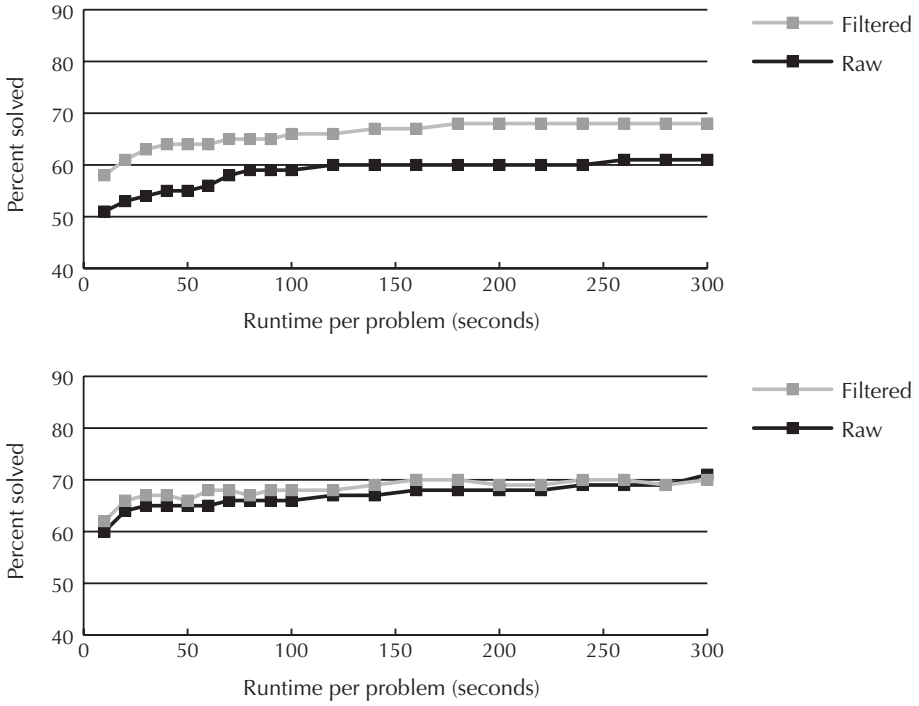

Figure 5: E, Versions 0.9 and 0.91dev001

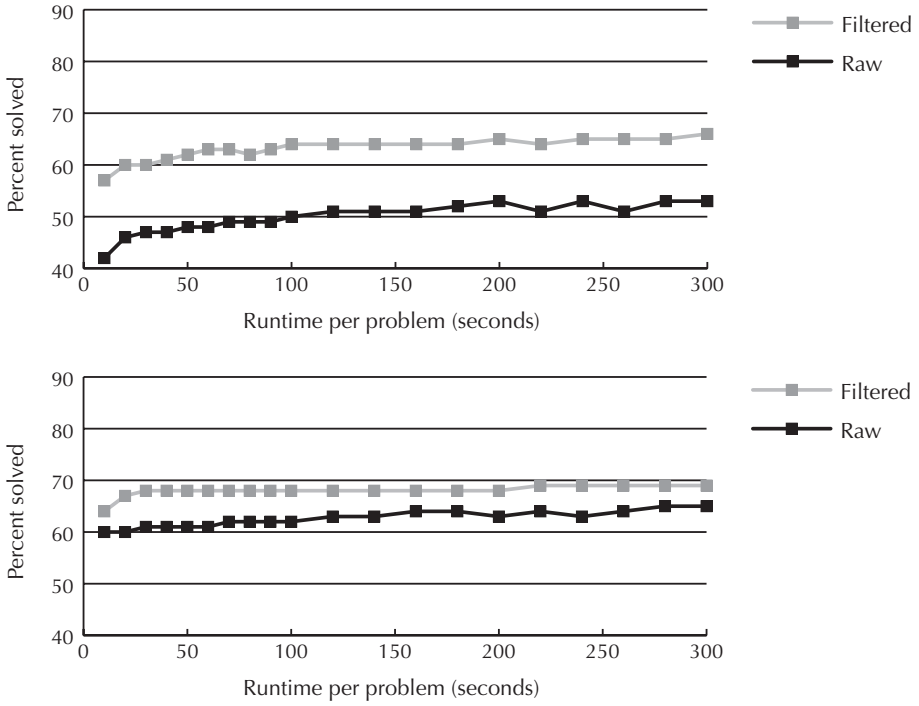

Figure 6: SPASS, Default Settings and with SOS

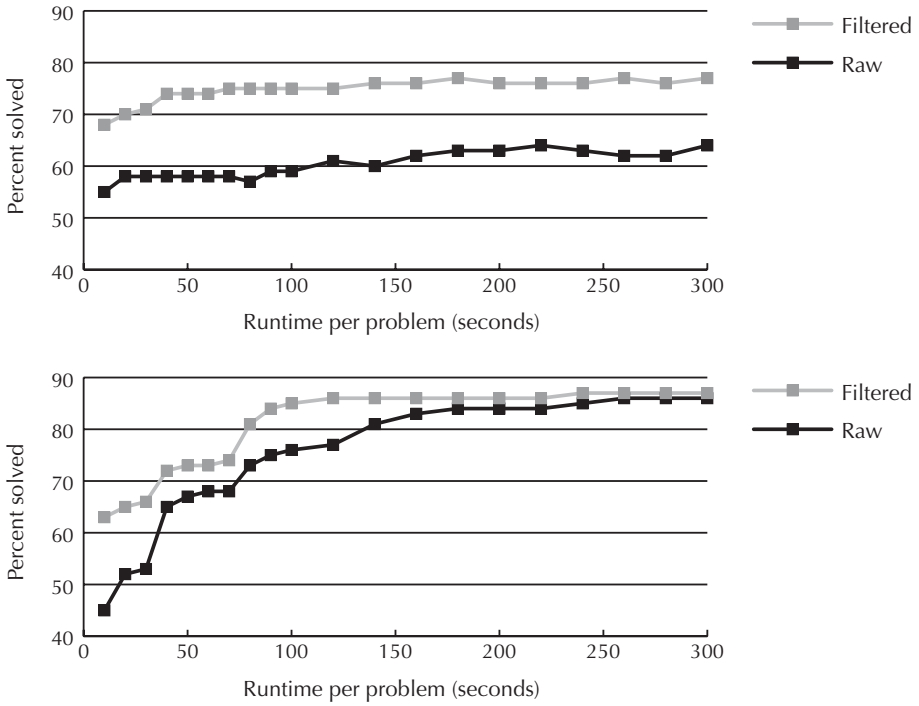

Figure 7: Vampire, Default Settings and in CASC Mode

The graphs offer compelling evidence that relevance filtering is beneficial in our application. The success rate for the filtered problems exceeds that for the raw ones in virtually every case. This improvement is particularly striking given that eight percent of the filtered problems appear to be missing essential clauses, compared with their automatically-reduced counterparts. Perhaps gains are made elsewhere, or these eight percent are too difficult to prove anyway. A further surprise is that, with the exception of Vampire running in CASC mode, increasing processor time to 300 seconds does not give an advantage to the raw problems. The success rate for raw problems should eventually exceed that of the filtered ones, but the crossover point appears to be rather distant. Exceptions are E version 0.91dev001, where crossover appears to be taking place at 280 seconds, and Vampire in CASC mode, where by 300 seconds the two lines are close.

6 Conclusions

We wish to refute large, machine-generated sets of clauses. Experiments with the notion of "referenced axioms" demonstrate that reducing the problem size greatly improves the success rate. However, this technique introduces a dependence on past proof attempts, so we have sought methods of reducing the problem size through a simple analysis of the problem alone.

We have presented simple ideas for relevance filtering along with empirical evidence to demonstrate that they improve the success rate in a great variety of situations. The simplicity of our methods is in stark contrast to the tremendous sophistication of automated theorem provers. It is surprising that such simple methods can yield

benefits. We believe that the secret is our willingness to sacrifice completeness in order to improve the overall success rate. Our method may be useful in other applications where processor time is limited and completeness is not essential. It is signature based, so it works for any problem for which the conjecture clauses have been identified. Our version of the filters operates on Isabelle theorems and assumes Isabelle's type system, but versions for standard first-order logic should be easy to devise.

Our filtering gave the least benefit with the specially-modified version of the E prover. Developer Stephan Schulz, in an e-mail dated 14 April 2006, had an explanation:

> E has a number of goal-directed search heuristics. The new version always selects a fairly extreme goal-directed one for your problems ... [which] will give a 10 times lower weight to symbols from a conjecture than to other symbols (all else being equal). I suspect that this more or less simulates your relevance filtering.

Such a setting could probably be added to other ATPs, and in a fashion that preserves completeness.

The weighting mechanisms of other ATPs may be able to simulate our filter. However, finding good configurations requires expert knowledge of each ATP being used. We were never able to improve upon the default configuration of E, despite its good documentation and helpful developer. (Many ATPs do not have these benefits.) In contrast, our filter provides a uniform solution for all ATPs.

We do not feel that our methods can be significantly improved; their technological basis is too simple. More sophisticated techniques, perhaps based on machine learning, may yield more effective relevance filtering.

As an offshoot of the work reported above, we have submitted 565 problems to the TPTP library. These are 285 raw problems plus 280 solutions: versions that have been automatically reduced as described in Sect. 3 above.

Acknowledgements

The research was funded by the EPSRC grant GR/S57198/01 *Automation for Interactive Proof* and by the L4.verified project of National ICT Australia. Stephan Schulz provided a new version of the E prover, optimized for our problem set.

References

[ABH+98] 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.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Robinson and Voronkov [RV01b], chapter 2, pages 19–99.

[BHdN02] 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.

[BR04] David Basin and Michaël Rusinowitch, editors. *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097. Springer, 2004.

[FF99] Marc Fuchs and Dirk Fuchs. Abstraction-based relevancy testing for model elimination. In Harald Ganzinger, editor, *Automated Deduction — CADE-16 International Conference*, LNAI 1632, pages 344–358. Springer, 1999.

[MP04] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In Basin and Rusinowitch [BR04], pages 372–384.

[MQP06] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 2006. in press.

[Pau97] Lawrence C. Paulson. Generic automatic proof tools. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 3. MIT Press, 1997.

[PY03] David A. Plaisted and Adnan Yahya. A relevance restriction strategy for automated deduction. *Artificial Intelligence*, 144(1-2):59–93, March 2003.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[RS98] Wolfgang Reif and Gerhard Schellhorn. Theorem proving in large theories. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction— A Basis for Applications*, volume III. Applications, pages 225–240. Kluwer Academic Publishers, 1998.

[RV01a] 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.

[RV01b] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science, 2001.

[SBF+03] 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.

[Sch04] Stephan Schulz. System description: E 0.81. In Basin and Rusinowitch [BR04], pages 223–228.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In Robinson and Voronkov [RV01b], chapter 27, pages 1965–2013.

[WRC65] Lawrence Wos, George A. Robinson, and Daniel F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM*, 12(4):536–541, 1965.

Translating Higher-Order Problems to First-Order Clauses

Jia Meng[1], Lawrence C. Paulson[2]
[1]National ICT, Australia
jiameng@nicta.com.au
[2]Computer Laboratory, University of Cambridge, U.K.
LP15@cam.ac.uk

Abstract

Proofs involving large specifications are typically carried out through interactive provers that use higher-order logic. A promising approach to improve the automation of interactive provers is by integrating them with automatic provers, which are usually based on first-order logic. Consequently, it is necessary to translate higher-order logic formulae to first-order form. This translation should ideally be both sound and practical. We have implemented three higher-order to first-order translations, with particular emphasis on the translation of types. Omitting some type information improves the success rate, but can be unsound, so the interactive prover must verify the proofs. In this paper, we will describe our translations and experimental data that compares the three translations in respect of their success rates for various automatic provers.

1 Introduction

Interactive theorem provers, such as HOL4 [GM93], Isabelle [NPW02] and PVS [ORR+96] are widely used for formal verifications and specifications. They provide expressive formalisms and tools for managing large scale proof projects. However, a major weakness of interactive provers is the lack of automation. In order to overcome the problem, we have integrated Isabelle with automatic theorem provers (ATPs) [MQP06]. ATPs combine a variety of reasoning methods and do not require users' instructions on how to use an axiom or when to use an axiom. For example, they do not require equalities to be oriented, but use them as undirected equations.

Among the many logics used by interactive provers, higher-order logic (HOL) is the most popular one because of its expressiveness. One of the most widely used logics in Isabelle is Isabelle/HOL. In contrast, most of the powerful ATPs are based on first-order logic (FOL). Therefore, it is important to translate HOL problems into FOL format.

Those HOL problems that do not involve function variables, predicate variables or λ-abstractions can be translated directly to FOL format. However, we must be careful about how to translate HOL problems that are truly higher-order. In particular, we need to include the problem's type information to preserve soundness. A sound approach is to include all types for all terms. Unfortunately, this will result in large terms and clauses, and much of the type information is redundant. A more compact type representation

could enhance the performance of ATPs. Omitting some type information could lead to unsoundness, which ultimately we will prevent through proof reconstruction (Sect. 2.5).

We have implemented three HOL to FOL translations, and we believe two of them are new. We have also carried out extensive experiments in order to assess their effectiveness with the provers E [Sch04], SPASS [Wei01] and Vampire [RV01].

Paper outline. We first describe three HOL to FOL translations and discuss their soundness (Sect. 2). We then describe the experiments we ran (Sect. 3) and finally offer some conclusions (Sect. 4).

2 Background

Higher-order logic (HOL) extends first-order logic (FOL) in several respects. The main difference is that HOL terms can denote truth values and functions. Function values can be expressed using λ-abstractions or by *currying*: that is, by applying a function to fewer than the maximum number of arguments. In FOL, a function must always be supplied the same number of arguments. In translating from HOL to FOL, the only way to reconcile this difference is to regard all HOL functions as constants while providing a two-argument function (called `app` and is abbreviated by `@` below) to express function application. In addition, we need a predicate `B` to convert all top-level FOL terms of boolean type to predicates. These translations allow first-order provers to solve many problems that contain higher-order features, though they do not yield the full power of higher-order logic.

For example, the HOL formula $\forall F\, p(F(x))$ is translated to

```
{++B(@(p,@(F,x)))}
```

We use the combinators **I**, **K**, **C**, **B** and **S** to represent λ-abstractions, asserting the combinator reduction equations as axioms. (Although **K** and **S** suffice in theory, the resulting translation is exponential in the number of abstractions.) Another axiom we assert is function extensionality:

$$\forall f g\, [(\forall x\, f(x) = g(x)) \rightarrow f = g].$$

It has the following clause form, where `e` is a reserved Skolem function symbol, yielding some x such that $f(x) \neq g(x)$.

```
{--equal(@(F,@(@(e,F),G)),@(G,@(@(e,F),G))),
 ++equal(F,G)}
```

Finally, equality in Isabelle may appear in a λ-abstraction, and thus is treated as an ordinary function symbol. We use a new function symbol `fequal` to represent the function version of equality. Through λ-reduction, this equality may be promoted to predicate level, becoming an ordinary equality. Promotion requires two additional axioms:

```
{++B(@(@(fequal,X),Y)),--equal(X,Y)}
{--B(@(@(fequal,X),Y)),++equal(X,Y)}
```

Not all subgoals require the full power of HOL. Often the initial steps of the proof replace complicated constructions by simple ones. Of the remaining subgoals, many are purely first-order. Others are higher-order but use no λ-abstractions. These special cases admit more efficient translations into first-order clauses, though naturally we must also provide a translation that accommodates the general case.

2.1 Types in Isabelle/HOL

We have just seen how to represent HOL formulae in FOL form. A more important issue is to embed type information of HOL formulae in FOL clauses. Let us review how this works for problems that are already first-order [MP04, MQP06]. Before that, we give a brief overview of Isabelle/HOL's polymorphically sorted type system. We refer readers to the two papers above for more information.

Isabelle/HOL supports *axiomatic type classes*, where a type class is a set of types. For example, the type for real numbers `real` is a member of type class `linorder`. A type class is axiomatic because it may have a set of properties—specified by axioms—that all its member types should satisfy. A type may belong to several type classes and an intersection of type classes is a *sort*. Moreover, each type constructor has one or more *arities*, which describe how the result type class depends upon the arguments' type classes. For example, type constructor `list` has an arity that says if its argument is a member of class `linorder` then the resulting list's type is also a member of `linorder`.

Constants can be overloaded and types can be *polymorphic*, allowing instantiation to more specific types. For example, the $\leq$ operator has the polymorphic type $\alpha \to \alpha \to bool$; when it has type $nat \to nat \to bool$ it denotes the usual ordering of the natural numbers, and when it has type $\alpha\ set \to \alpha\ set \to bool$ it denotes the subset relation. The latter type is still polymorphic in the type of the set's elements. Isabelle's overloading cannot be eliminated by preprocessing because polymorphic theorems about $\leq$ are applicable to all instances of this function, despite their different meanings.

When we translate Isabelle formulae to FOL clauses, we need to formalize types, especially in view of Isabelle's heavy use of overloading. We need to ensure that Isabelle theorems involving polymorphic functions are only used for appropriate types. To accomplish this, polymorphic functions carry type information as additional arguments and we translate Isabelle types to FOL terms. For example, we translate $x \leq y$ where x and y are $\alpha\ set$ to $le(x, y, set(\alpha))$. We also translate Isabelle's axiomatic type classes into first-order clauses. For this, we translate type classes to FOL predicates and types to FOL terms.

This translation is reasonably compact, and it enforces overloading ($\leq$ on sets is not confused with $\leq$ on integers), but it does not capture other aspects of types. For example, if we declare a two-element datatype `two`, then we obtain the theorem

$$\forall x\, [x = a \vee x = b].$$

The corresponding clause does not mention type `two`:

```
{++equal(X,a), ++equal(X,b)}
```

It therefore asserts that the universe consists of two elements; given our other axioms, ATPs easily detect the inconsistency. We simply live with this risk for the moment,

pending the implementation of proof reconstruction. A simple way of detecting such issues is to check whether a proof refers to at least one conjecture clause: if not, then the axiom clauses by themselves are inconsistent. This method detects some invalid proofs, but not all of them.

Since HOL problems require currying, we need a different type embedding method from first-order ones. We have implemented three type translations, namely fully-typed, partial-typed and constant-typed.

2.2 The Fully-Typed Translation

The fully-typed translation, which resembles Hurd's translation [Hur02], is sound. The special function `typeinfo`, which we abbreviate to `T`, pairs each term with its type. For instance, the formula $P < Q$ is translated to

```
{++B(T(@(T(@(T(<, a=>a=>bool), T(P,a)), a=>bool), T(Q,a)), bool))}
```

This translation is sound because it includes types for all terms and subterms, right down to the variables. When two terms are unified during a resolution step, their types are unified as well. This instantiation of types guarantees that terms created in the course of a proof continue to carry correct types. Isabelle unifies polymorphic terms similarly. In fact, the resolution steps performed by an ATP could in principle be reconstructed in Isabelle. Each FOL axiom clause corresponds to an Isabelle theorem. If two FOL clauses are resolved, then the resolvant FOL clause will correspond to the Isabelle theorem produced by Isabelle's own resolution rule.

The fully-typed translation introduces much redundancy. Every part of a function application is typed: the function's type includes its argument and result types, which are repeated in the translation of the function's argument and by including the type of the returned result. Through experiments (Sect. 3), we have found that these large terms degrade the ATPs' performance. A more compact HOL translation should improve the success rate.

Hurd [Hur03] uses an untyped translation for the same reason. No term or predicate has any type information. Because this translation can produce unsound proofs, Hurd relies on proof reconstruction to verify them. If reconstruction fails, Hurd calls the ATP again, using a typed translation. Hurd says that this happens less than one percent of the time. This combination of an efficient but unsound translation with a soundness check achieves both efficiency and soundness. We intend to take the same approach.

If we are to achieve a compact HOL translation, we will have to omit some types, potentially admitting some unsound proofs. We cannot use a completely untyped translation because our requirements differ from Hurd's. His tactic sends to ATPs a few theorems that are chosen by users. In contrast, we send ATPs hundreds of theorems, many involving overloading. Omitting the types from this large collection would result in many absurd proofs, where for example, the operator $\leq$ simultaneously denoted "less than" on integers and the subset relation. We have designed and experimented with two compact HOL translations: the partial-typed and constant-typed translations. These translations attach the most important type information (such as type instantiations of polymorphic constants) that can block some incorrect resolutions.

2.3 The Partial-Typed Translation

The partial-typed translation only includes the types of functions in function calls. The type is translated to a FOL term and is inserted as a third argument of the application operator (`@`). Taking the previous formula $P < Q$ as an example, we translate it to

```
{++B(@(@(<, P, a=>a=>bool), Q, a=>bool))}.
```

Here, the type of `<` is `a=>a=>bool`, and we include this type as an additional argument of function application `@`.

In a HOL formula, a function may be passed to another function as an argument. If a function appears without arguments, we do not include its type. The FOL clauses are derived from Isabelle formulae, which we know to be well-formed and type correct. The partial-typed translation avoids the redundancy of the fully-typed translation. Most of the time, this type encoding also ensures correct treatments of Isabelle overloading: Isabelle overloaded constants are most likely to appear as operators (functions and predicates) in formulae, whose types are inserted by the partial-typed encoding.

However, the partial-typed translation can still yield unsound proofs. It is vulnerable to the example involving datatype `two`, described in Sect. 2.1 above.

2.4 The Constant-Typed Translation

In the constant-typed translation, we include types of polymorphic constants only. Furthermore, we do not include a constant's full type but only the instantiated values of its type variables. Monomorphic constants do not need to carry types because their names alone determine the types of their arguments. A polymorphic constant is translated to a first-order function symbol. Its arguments, which represent types, are obtained by matching its actual type against its declared type. This treatment of types is similar to the one we use for problems that are already first-order.

Again considering our standard example, if P and Q are natural numbers (type `nat`), we translate the formula $P < Q$ to

```
{++B(@(@(<(nat), P), Q))}.
```

Similarly, if P and Q are sets (type α `set`), it becomes

```
{++B(@(@(<(set(a)), P), Q))}.
```

As for equality, if it appears as a predicate, then we do not insert its type. However, if it appears as a constant in a combinator term, then we include its argument's type as its argument. Similarly, we translated the equality axiom above to the two clauses

```
{++B(@(@(fequal(T),X),Y)),--equal(X,Y)}
{--B(@(@(fequal(T),X),Y)),++equal(X,Y)}
```

This translation can reduce the size of terms significantly. However, like the partial-typed one, it can be unsound.

2.5 Which Translation to Use and Soundness Issues

Of the three HOL to FOL translations above, the fully-typed one is sound but produces excessively large terms. The partial-typed and constant-typed translations are more compact, but may introduce unsound proofs. If we use either of the compact translations, then we must verify proofs in Isabelle to ensure soundness.

There are several factors that affect the decision about which translation should be used as the default.

- Can we verify the proofs for partial-typed and constant-typed translations? The answer is yes. Although the FOL clauses carry insufficient type information, the clauses still correspond to Isabelle lemmas and goals. Our approach to proof reconstruction, which is currently being implemented, involves following the low-level resolution steps. If two clauses cannot be resolved due to incompatible types, Isabelle will detect this.

- What benefit do we obtain from using the compact translations? Our experimental results (Sect. 3) show that the compact translations can boost the success rate significantly. Therefore, it is worthwhile to use a compact translation, even if occasional unsound proofs require retrying the problem using the fully-typed translation.

Moreover, we aim to reconstruct proofs in Isabelle even if we use fully-typed translation. This is so that proofs can go through Isabelle kernel. Since proof reconstruction is needed regardless of which translation is used, the only potential extra cost involved in using a compact translation is that of occasional retries.

3 Experiments

It is obvious that the constant-typed translation is the most compact, while the fully-typed one is the least compact. We can therefore predict that the constant-typed translation will deliver the best results with ATPs, while the fully-typed one will turn out to be the worst. However, such claims need to be backed up by observations, especially given that the fully-typed translation is the best for soundness.

For our experiments, we took 79 problems generated by Isabelle, most of which are higher-order. Since our HOL translation can also be used for purely FOL problems and our experiments were aimed at testing efficiency of the translation methods, we translated all problems (both HOL and FOL) using the three translation methods we mentioned in the previous section. We used our relevance filter [MP06] to reduce the sizes of the problem. We ran these tests on a bank of Dual AMD Opteron processors running at 2400MHz, using Condor[1] to manage our batch jobs.

Each graph compares the success rates of the three translations, for some prover, as the runtime per problem increases from 10 to 300 seconds. These short runtimes are appropriate for our application of ATPs to support interactive proofs, We tested three provers: E (Fig. 1), SPASS 2.2 (Fig. 2) and Vampire 8 (Fig. 3). We used E version

[1] `http://www.cs.wisc.edu/condor/`

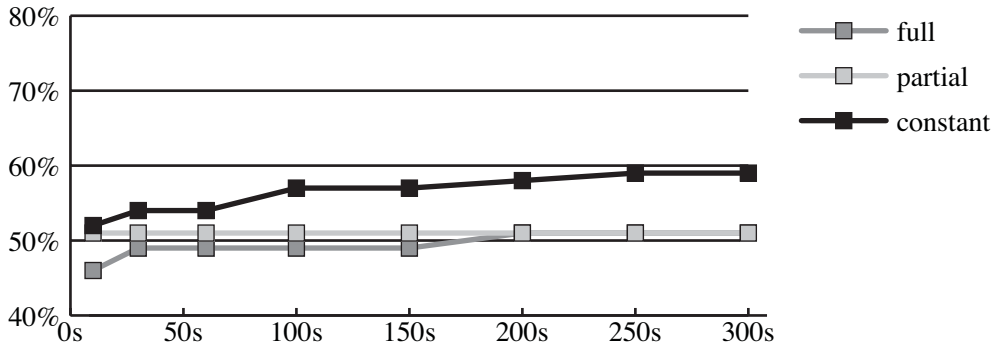

Figure 1: E, Version 0.91dev001

0.91dev001, a development version that surpasses E 0.9 "Soom". SPASS ran in automatic mode and, in a second run, with SOS enabled and splitting disabled.[2] Vampire ran in its default mode and with its CASC option.

On the whole, the constant-typed translation did indeed yield the highest success rate, while the fully-typed translation yielded the lowest, especially when runtime is increased. This is as we would expect, but a glance at the graphs shows that the situation is more complex than this. Against expectations, the partial-typed translation frequently outperforms the constant-typed one. For runtimes below about 200 seconds, the partial-typed translation gives the best results with SPASS (default settings) and Vampire. As runtime increases to 300 seconds, the constant-typed translation comes out top (or nearly) in all five graphs.

At its default settings, SPASS does not perform well with any translation. It is safe to conjecture that only trivial problems are being proved, where the translation makes little difference. By enabling SOS, which makes the proof search more goal-directed, SPASS delivers excellent results with the constant-typed translation.

Readers may ask whether the fully-typed translation has a lower success rate because the other translations are finding unsound proofs for our problems. We have found that three of the 79 problems have unsound proofs. For one of the problems, incorrect axioms cause all three of its translations to be unsound, so this error does not bias the results. For the other two problems, the compact translations are indeed to blame, giving a bias of 2/79 or 2.5% against the fully-typed translation. The advantage given by the compact translations is much greater than this. We do regard this unsoundness rate as too high, and we are considering a number of options for reducing it.

To obtain a quantitative picture of the differences between the three translations, we chose one of the problems and used tptp2X [SS04] to summarize its syntactic features. This problem is of median size in our problem set. It has 1150 clauses after relevance filtering, of which 105 are non-trivial: the remainder constitute a monadic Horn theory that describes Isabelle's type class system. Table 1 shows the figures common to all three translations. Table 2 shows variations among the translations.

A major difference is the maximal term depth, where fully-typed appears to have the greatest maximal term depth while constant-typed has the least. Shallower terms

[2]The precise option string is `-Splits=0 -FullRed=0 -SOS=1`.

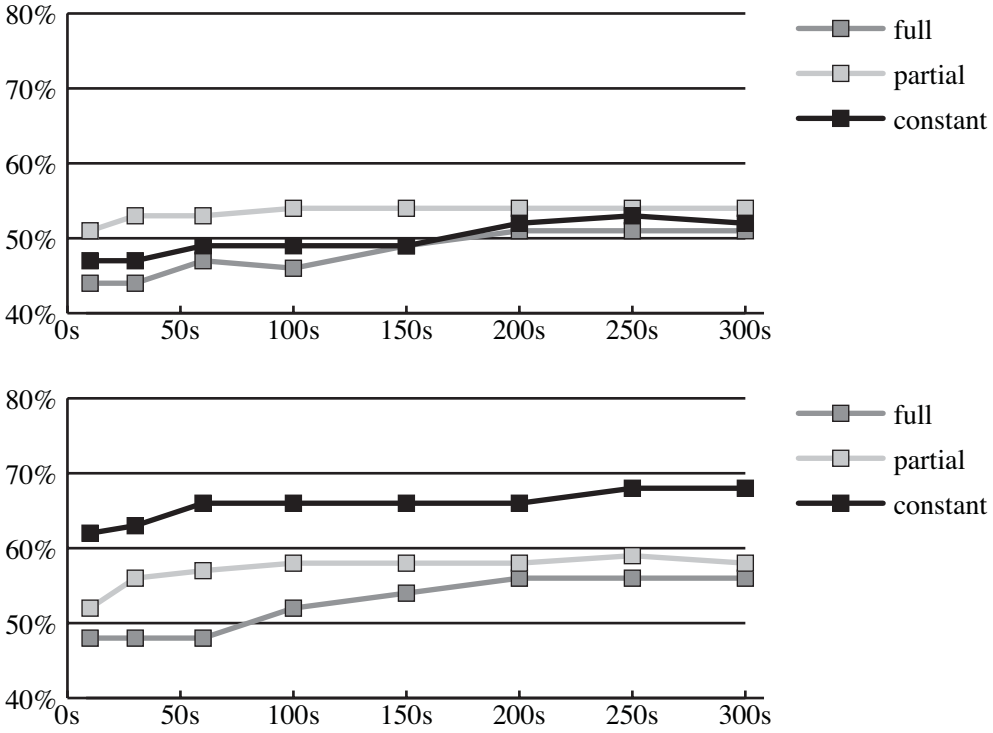

Figure 2: SPASS, Default Settings and with SOS

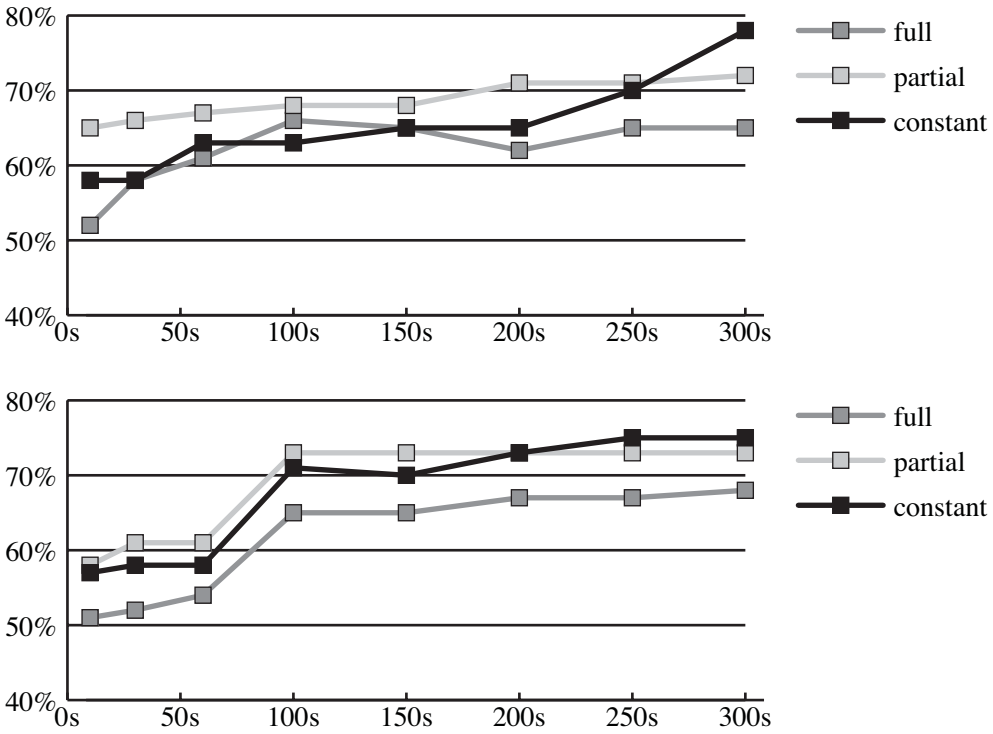

Figure 3: Vampire, Default Settings and in CASC Mode

Number of clauses	1150 (6 non-Horn; 198 unit)
Number of literals	2105 (152 equality)
Maximal clause size	3 (1 average)
Number of predicates	74 (0 propositional; 1–2 arity)

Table 1: Common to All Translations

	No. of functors	*No. of variables*	*Max. term depth*	*File size*
full	51 (42 constant)	1292 (69 singleton)	19 (1 average)	223460
partial	50 (42 constant)	1265 (66 singleton)	13 (1 average)	196663
constant	49 (10 constant)	1292 (106 singleton)	8 (1 average)	171724

Table 2: Differences between Three Translations

may be easier to process by ATPs, which may be a reason why constant-typed performs better overall. In addition, constant-typed produces the smallest problem file. Finally, although the fully-typed and constant-typed translations have the same number of variables, the latter has more singleton variables, which are variables that occur in a clause only once.

These statistics again suggest that the constant-typed translation should give the best results, so we have no explanation for the many situations in which the partial-typed translation performs best. The dips in the graphs, where the success rate drops as the runtime goes up, are also puzzling. A notable one is with Vampire, default settings, with the fully-typed translation (Fig. 3). Vampire's *limited resource strategy*, which discards clauses that cannot be used within the time limit, may explain its dips. For E and SPASS we have no explanation, but we have no doubt that the dips are real. Similar dips appear in our other experiments [MP06], and all of these measurements are made by automatic procedures.

4 Conclusions

We have described three HOL to FOL translations, which differ in their treatment of types. We have carried out extensive experiments to evaluate the effectiveness of the three translations. We have also obtained some statistics concerning how compact our translations are. Of the three translations—fully-typed, partial-typed and constant-typed—the constant-typed translation produces the most compact output.

Naturally, we would expect a provers' success rate to increase with a more compact clause form. This is what we have seen with E and with SPASS (provided SOS is enabled). However, we are surprised to see the simplest and most compact format does not always yield the best results with Vampire. Given that Vampire gives the best overall results, the partial-typed translation is worth considering. Because the more compact translations can be unsound, proofs found using them must be validated in some way, such as by proof reconstruction.

The higher-order logic we have investigated is Isabelle/HOL. However, our translations should be equally applicable to the similar logic implemented in the HOL4 system.

Any translations for PVS would have to take account of predicate subtyping, but their treatment of basic types might be based on our techniques.

Acknowledgements

The research was funded by the EPSRC grant GR/S57198/01 *Automation for Interactive Proof* and by the L4.verified project of National ICT Australia. Joe Hurd has given much helpful advice on how to translate from HOL to FOL. The referees made many useful suggestions for improving this paper.

References

[BR04] David Basin and Michaël Rusinowitch, editors. *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097. Springer, 2004.

[GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge Univ. Press, 1993.

[Hur02] Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Automated Deduction — CADE-18 International Conference*, LNAI 2392, pages 134–138. Springer, 2002.

[Hur03] 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.

[MP04] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In Basin and Rusinowitch [BR04], pages 372–384.

[MP06] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. These proceedings, 2006.

[MQP06] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 2006. in press.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification: 8th International Conference, CAV '96*, LNCS 1102, pages 411–414. Springer, 1996.

[RV01] 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.

[Sch04] Stephan Schulz. System description: E 0.81. In Basin and Rusinowitch [BR04], pages 223–228.

[SS04] Geoff Sutcliffe and Christian Suttner. The TPTP problem library for automated theorem proving. On the Internet at `http://www.cs.miami.edu/~tptp/`, 2004.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

Three Stories on Automated Reasoning for Natural Language Understanding

Johan Bos
University of Rome "La Sapienza"
bos@di.uniroma1.it

Abstract

Three recent applications of computerised reasoning in natural language processing are presented. The first is a text understanding system developed in the late 1990s, the second is a spoken-dialogue interface to a mobile robot and automated home, and the third is a system that determines textual entailment. In all of these applications, off-the-shelf tools for reasoning with first-order logic (theorem provers as well as model builders) are employed to assist in natural language understanding. This overview is not only an attempt to identify the added value of computerised reasoning in natural language understanding, but also to point out the limitations of the first-order inference techniques currently used in natural language processing.

Introduction

Since the mid 1990s I've been using tools from automated deduction (mainly theorem provers and model builders) to solve problems in natural language understanding. I've done this both from the perspective of *computational linguistics* (testing and improving a linguistic theory by means of a computational implementation) as well as that of *natural language processing* (using inference in applications such as question answering and textual entailment). In this paper I will write about some recent projects that I was involved in, with the aim of convincing researchers — both from the natural language engineering and the automated deduction communities — that computerised reasoning can be successfully applied to natural language understanding.

Right — what's new about this claim? Isn't it obvious that one needs some form of computerised reasoning to model natural language understanding? Indeed, in my opinion it is. However, it is astonishing to see how few tools from automated reasoning have made it in real applications. Why is that? I think there are a couple of reasons for this.

First of all, a rather high level of interdisciplinarity is required. One needs not only to know about linguistics (and in particular (formal) semantics), but also about natural language processing, knowledge representation, and automated inference. As a matter of fact, not many researchers match this profile, and looking back to the 1980s and 1990s, there seem fewer and fewer interested in taking on the pursuit. Or as Steve Pulman put it on the recent ICoS (Inference in Computational Semantics) conference: "I feel like an endangered species."

Secondly, there is an enormous gap between formal linguistic theory and practical implementation. There is a vast amount of formal linguistics theory on the semantics of natural language. However interesting most of it is, it doesn't always lead directly to computational implementation. Many of the phenomena that are accounted for in formal theory are quite rare in natural data, so a natural language engineer won't loose much by ignoring it. But more crucially, from a an automated reasoning point of view, almost all semantic phenomena are formalised in higher-order logic (a trend set by noone less than Richard Montague), which is, as is well-known, a computational nightmare.

Third, it is not trendy to use theorem proving in natural language processing. In contrast, stochastic approaches dominate the field, and after having been successfully applied to speech recognition and syntactic processing, it won't take long until statistics will play a major role in semantic processing. Having said that, time and time again I am surprised by the opinion that using theorem proving in natural language understanding is classified as the "traditional approach." What tradition? OK – it was tried in the 1970s and 1980s, but it never got to work really well, for reasons that should not really surprise us: theorem proving hadn't matured into a state as we know it today, and moreover, trivially, computers lacked the memory and speed to perform the computations required by inference. Yet, we have only started to understand the limitations and opportunities of computerised reasoning in natural language understanding.

After this introduction I will present three applications demonstrating successful use of first-order inference tools. The first is a text-understanding system that calculates presuppositions of sentences and performs consistency and informativeness checks on texts. The second is a spoken dialogue system, interfaced to a mobile robot and an automated home environment, that uses theorem proving and model building for planning its linguistic and non-linguistic actions. The third is a system for recognising textual entailment. In all of these applications I will discuss the reasoning tools used, how they are used, what added value they had, and what their limitations were.

1 Presupposition Projection

In the mid 1990s I started to implement tools for the semantic analysis of English texts, as part of my thesis work at the University of the Saarland in Saarbrücken, Germany. One of the aims was to follow linguistic theory as close as possible and see how much of it could be implemented straight away. I adopted Discourse Representation Theory (DRT), initially developed by Hans Kamp, because it accounted for a wide range of linguistic phenomena in a unified framework, and, crucially, had a model-theoretic semantics [KR93]. In particular I was interested in modelling presupposition projection, and I followed a recent proposal by Rob van der Sandt, whose theory of presupposition was casted in DRT [VdS92].

My implementation efforts resulted in the DORIS system (Figure 1). It had a reasonable grammer coverage (substantially more than a toy grammar, but certainly not reaching the level of today's wide coverage grammars). It parsed English sentences and computed an underspecified discourse representation of the input sentence, followed by resolving ambiguities. The linguistic phenomena covered included pronouns, quantifier and negation scope, and presuppositions [Bos01].

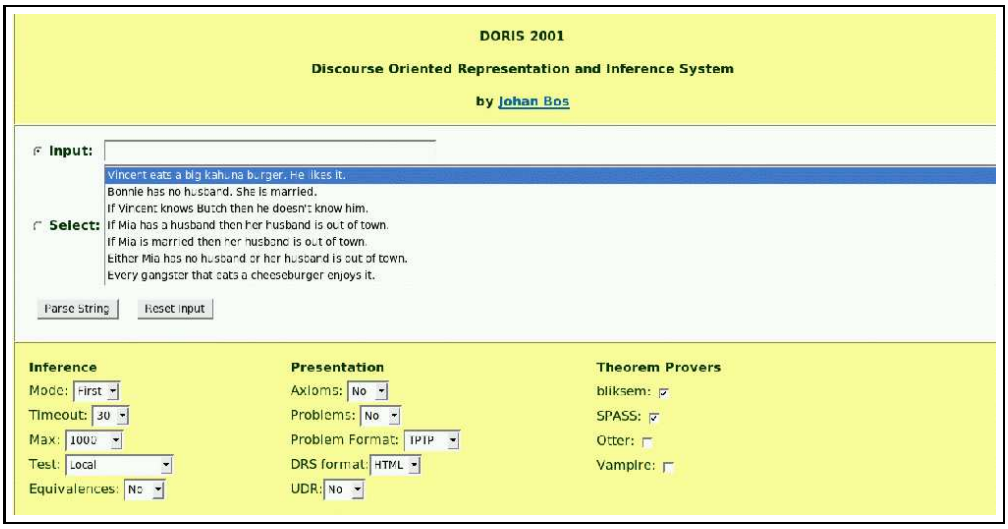

Figure 1: Screenshot of the DORIS system. In the upper window users can type or select an English sentence. The lower window provides several parameters, for instance the selection of various theorem provers.

Presuppositions are propositions taken for granted by the speaker, and "triggered" by words or phrases. Rather informally, p presupposes q if both p and $\neg p$ entail q. For instance, the phrase "Mia's husband" presupposes that Mia is married, and that Mia is a woman, because "Jody likes Mia's husband" and the negation of this sentence "Jody doesn't like Mia's husband" both entail that Mia is married and is a woman. The problem with presuppositions is that they are sometimes neutralised by the linguistic context, and that it is quite hard to pin down exactly when they are and when they are not, especially in sentences that contain some form of implication, negation, or disjunction. Consider, for instance, the following three sentences (with the relevant presupposition trigger typeset in bold face):

(a) If Mia has a date with Vincent, then **her husband** is out of town.
(b) If Mia has a husband, then **her husband** is out of town.
(c) If Mia is married, then **her husband** is out of town.

Here (a) presupposes that Mia is married, but (b) and (c) do not. Van der Sandt's theory explained this by constructing various possibilities of positioning the presupposition, and then checking whether these were acceptable by posing acceptability constraints upon them: consistency as well as informativeness of the resulting text, both on the global and local level of discourse. For (c), there are two positions where the presupposition can "land" (which are underlined):

(c-1) <u>Mia has a husband</u>. If Mia is married, then **her husband** is out of town.
(c-2) If Mia is married/<u>has a husband</u>, then **her husband** is out of town.

However, in the first paraphrase (c-1) the antecedent of the conditional violates the constraint of (local) informativeness: if Mia has a husband, then the fact that she

is married is not new information. In the second paraphrase (c-2) all acceptability constraints are satisfied. As a consequence, (c-1) is rejected, and (c-2) is accepted, as possible interpretation.

Despite the adequacy of the predictions of the theory, there was a still a problem: the acceptability contraints required logical inference. But how cuold you implement this? Even though DRT was an established semantic theory backed up with a model-theory, there were no (efficient) theorem provers available that could reason with the semantic representations employed by DRT. Discussions with Patrick Blackburn and Michael Kohlhase (both in Saarbrücken, at the time) developed the idea of using first-order theorem provers to implement Van der Sandt's acceptability constraints. As the core of DRT is a first-order language, it turned out to be pretty straightforward to translate the DRT representations into ordinary first-order formula syntax, something that first-order theorem provers could digest. Soon after, contacts were made with Hans de Nivelle, whose theorem prover BLIKSEM was among the first that was put to the test [DN98]. And it looked promising: BLIKSEM could handle most of the problems given to it in reasonable time.

However, as some natural language examples could cause hundreds of consistency checking tasks (due to a combinatorial explosion of linguistic ambiguities), it took a long time before BLIKSEM had dealt with them all. Michael Kohlhase and Andreas Franke came to the rescue, by offering MATHWEB, a web-based inference service [FK99]. MATHWEB farmed out a set of inference problems to different machines using a common software bus. Using the internet and many machines around the world (I recall that there were machines running in Edinburgh, Budapest, Saarbrücken, and Sydney, among other sites), MathWeb could basically be viewed as a parallel supercomputer. (This sounds perhaps quite ordinary right now, but at the time it was a sensation.) To cut a long story short, DORIS was interfaced directly to MATHWEB, and many different theorem provers for first-order logic were added: SPASS [WAB+99], FDPLL [Bau00], OTTER [MP96], and VAMPIRE [RV02].

In sum, the DORIS system demonstrated that first-order inference could play an interesting role in natural language processing, albeit with limitations [BBKdN01]. It generated a new application area for automated deduction (in fact, some of the problems generated by DORIS made it to the TPTP collection, thanks to Geoff Sutcliffe), and it opened a whole new vista of research in computational semantics. (Incidentally, it also helped to precisely formulate the acceptability constraints of Van der Sandt's theory of presupposition projection.)

So, what were these limitations? Scalability was one of them. A theorem prover would do well on a couple of sentences, but — not surprisingly given the computational properties of first-order logic — it would just choke on larger texts. Linguistic coverage was another. Some linguistic phenomona require richer semantic representations and therefore harder problems (for instance, tense and aspect require a richer sortal hierarchy, cardinal expression require counting, and plurals require elements of set theory).

The last version of DORIS was released in 2001 [Bos01]. Although it was an important step in the development of computational semantics, its limited grammatical coverage and unfocussed application domain left it without a future. At the time I thought that it would take at least twenty years to develop a parser that achieved both wide-coverage *and* syntactic representations of enough detail to construct meaningful semantic repre-

sentations (I was, fortunately, very wrong! See Section 3). In order to reach a new level of sophistication in computational semantics, I instead focussed on small domains, in which the grammatical coverage and necessary background knowledge could be specified a-priory. Human-computer dialogue systems turned out to be the killer application.

2 Spoken Dialogue Systems

At the University of Edinburgh I was involved in developing a spoken dialogue system which was interfaced with a (real) robot. The robot was a RWI Magellan Pro robot, with sonars, infrared sensors, bumpers, and a video camera. It had an on-board computer connected to the local network via a wireless LAN interface. The robot moved about at the basement of Buccleuch Place, and people could direct it, ask it questions, or provide it with new information, all via speech. A typical conversation could be:

Human: Robot?
Robot: Yes?
Human: Where are you?
Robot: I am in the hallway.
Human: OK. Go to the rest room!

Figure 2: An early version of Godot the talking robot with the roboticist Tetsushi Oka (6 Buccleuch Place, Edinburgh, 2001).

Such kinds of dialogues were relatively straightforward to model with the then state-of-the-art in human-machine dialogue. Yet, the robot was still "semantically challenging": it had no means to draw inferences. What I aimed to do was using components of the DORIS system to give the robot means to do consistency checking, answer questions,

and calculate its next actions. In particular, I was interested in letting the robot react to inconsistent information or obvious information, envisioning dialogues such as:

Human: Where are you?
Robot: I am in the hallway.
Human: You are in my office.
Robot: No, that's not true!
Human: You are in the hallway.
Robot: Yes, I know.

I knew this was feasible because of the DORIS experience: theorem provers can easily handle the amount of information, and the amount of background knowledge, given the limited domain and environment, was easy to compute and maintain. And so it turned out to be: using SPASS as theorem prover, the robot was checking whether each assertion of the user was consistent with its current state and knowledge of the dialogue.

After having implemented this, I was interested in using first-order theorem proving for planning the actions of a directive, primarily from a semantic point of view. I considered commands that involved negation, disjunction, or quantification. Examples of utterances that I had in mind included:

(a) Turn on a light.
(b) Switch in every light.
(c) Switch on every light in the hallway.
(d) Turn off every light except the light in the office.
(e) Go to the kitchen or the rest room.

In (a), the robot had to turn on a light that was currently switched off (and of course it had to complain when all the lights were already on). In (b), it had to turn on all lights that were currently off (some of the lights could be on already). In (c), it should only consider lights in the hallway (restrictive quantification). In (d), it should consider all lights minus those in the office (negation). In (e), it should have either gone to the kitchen or to the rest room (disjunction).

This was hard to do with a theorem prover. It seemed a natural task for a finite model builder though. Via the DORIS system I already came into contact with Karsten Konrad's model builder KIMBA [KW99], but I had never used model builders other than checking for satisfiability. I started using Bill McCune's MACE because it searched models by iteration over domain size, and generally generating models that were both domain-minimal and minimal in the extensions of the predicates [McC98]. Model building was successfully integrated such that the minimal models produced by MACE where used to determine the actions that the robot had to perform [BO02]. The robot in action was regularly demonstrated to visitors at Buccleuch Place (Figure 2) as well as to the general public at the Scottish museum in Edinburgh (Figure 3).

Of course there were the usual limitations. As I was using a first-order language with possible worlds to model the semantics of actions, I was forced to erase the dialogue memory after every second utterance in order to keep the response time acceptable. Also, the number of different objects in the domain was very limited (given the background axioms of the robot, MACE produces models in reasonable time for models up to domain size 20). Nevertheless, the overall system was impressive, and showed what one could do with general purpose, off-the-shelf, first-order inference tools in a practical system.

Figure 3: Godot the talking robot at the Scottish Museum (Edinburgh, 2003).

3 Recognising Textual Entailment

The rapid developments in statistical parsing were of course of interest to the semanticist. Yet most of these parsers produced syntactic derivations that were unsuitable to produce semantic representations in a systematic, principled way. It is not an exaggeration to say that the release of a stastistical wide-coverage parser for CCG (Combinatorial Categorial Grammar) in 2004 corresponded to a breakthrough in computational semantics. This CCG parser, implemented by Stephen Clark and James Curran [CC04], and trained on an annotated treebank developed by Julia Hockenmaier and Mark Steedman [HS02], had the best of both worlds: it achieved wide coverage on texts, and produced very detailed syntactic derivations. Because of the correspondence between syntax and semantic rules in CCG, this framework was the ideal setting for doing semantics.

Because CCG is a heavily lexicalised theory, it has a large number of lexical categories, and very few rules. In order to translate the output of the parser (a CCG derivation) into a DRT representation (which was my main aim, in order to reuse the existing tools that I developed for DRT and inference), I coded a lambda-expression for each of the ca. syntactic 400 categories that were known to the parser. Using the lambda-calculus to produce DRT representations, we simply had a parser that translated newspaper texts into semantic representations, with a coverage of around 95% [BCS$^+$04, Bos05]. This was a great starting point for doing computerised reasoning for natural language on a wider scale.

In the same year the first challenge to recognising textual entailment (RTE) were organised. This is basically a competition for implemented systems to detect whether one (small) text entails another (small) text. To give an impression of the task, consider

an example of a positive and and an example of a negative entailment pair are (where T is the text, and H the hypothesis, using the terminology of the RTE):

Example: 115 (TRUE)

T: On Friday evening, a car bomb exploded outside a Shiite mosque in Iskandariyah, 30 miles south of the capital, killing seven people and wounding 10, doctors said on condition of anonymity.
H: A bomb exploded outside a mosque.

Example: 117 (FALSE)

T: The release of its report led to calls for a complete ivory trade ban, and at the seventh conference in 1989, the African Elephant was moved to appendix one of the treaty.
H: The ban on ivory trade has been effective in protecting the elephant from extinction.

In the RTE challenge a participating system is given a set of entailment pairs and has to decide, for each T-H pair, whether T entails H or not. This turned out to be a very hard task. The baseline (randomly guessing) already gives a score of 50%, as half of the dataset correspond to true entailments, and the other half to false ones. The best systems on the RTE-1 campaign achieved a score approaching 60%.

With the CCG parser and semantics at my disposal, I decided to implement a system that used logical inference to approach the RTE challenge. The overall idea, given the available tools, was straightforward: produce a DRT representation for T and H, translate these to first-order logic, and then use an off-the-shelf prover to check whether $T' \rightarrow H'$. We used VAMPIRE as theorem prover [RV02], motivated by its performance at the recent CASCs.

At first, the performance was limited. The system performed quite well on cases such as 115 above, but unfortunately the RTE challenge doesn't contain many of these. Most of the examples require a lot of background knowledge to draw the correct inference. I used WordNet (an electronic dictionary) to compute some of these background axioms automatically. Yet still there were knowledge gaps. It would be nice to have a theorem prover that would be able to say that it "almost found a proof", instead of just saying "yes" or "(probably) no".

Back to model building. Finite model builders, such as MACE [McC98], said more than just "yes" or "no". They also give you a finite model if there is one. As this model is a clearly defined mathematical entity, it is ideal to simulate the "almost found a proof" scenario. That is, by generating minimal models for T' and for $T' \wedge H'$, we hypothesised that comparing the number of entities of the two models would give us a useful handle on estimating entailment. If the difference is relatively small, it is likely that T entails H. Otherwise it is not. (To deal with negation, one also has to calculate the models for $\neg T'$ and $\neg(T' \wedge H')$ and compare the model sizes. To deal with disjunction, one has to do this for all minimal models for T and H — but as of now it is unclear to me how to implement the latter in a neat way...)

This turned out to work well — not as good as one would hope (because it is hard to get the right background knowledge), but significantly better than the baseline. We used standard machine learning techniques to estimate the thresholds of the model sizes. We used both the size of the domain as well as the number of instantiated relations in the model. It turned out that MACE was quite slow for longer sentences. We tried

PARADOX [CS03], which is faster, but it does not always return minimal models (with respect to the predicate extensions). To overcome this, we used PARADOX to calculate the domain size, and then called MACE to generate the model given that domain size.

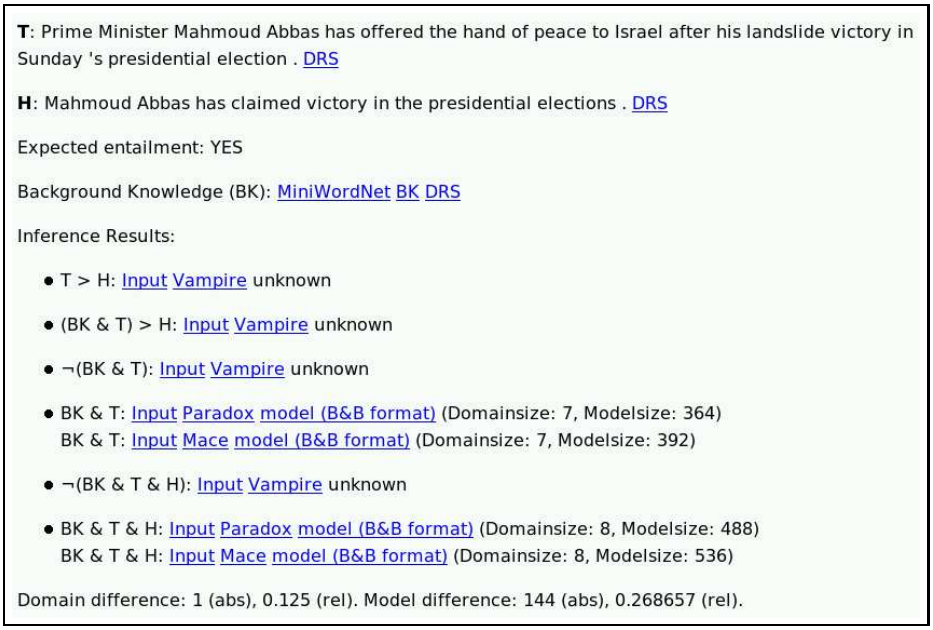

Figure 4: Example output screen for recognising textual entailment using the theorem prover VAMPIRE and the model builders MACE and PARADOX.

Conclusion

First-order inference tools, such as automated theorem provers and model builders, can be successfully used in natural language understanding applications. Obviously, there are limitations, but in many interesting applications these limitations play a subordinate role. Whether computerised (logical) reasoning will ever become part of mainstream research in natural language processing is questionable, though. We will have to see to what extent statistical approaches (currently dominating computational linguistics) can be applied to natural language understanding tasks. Meanwhile, collaboration between researchers working in automated deduction and computational linguistics should be stimulated to get a better understanding of the boundaries of applying automated reasoning to natural language understanding.

References

[Bau00] Peter Baumgartner. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In David McAllester, editor, *CADE-17 – The 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 200–219. Springer, 2000.

[BBKdN01] Patrick Blackburn, Johan Bos, Michael Kohlhase, and Hans de Nivelle. Inference and Computational Semantics. In Harry Bunt, Reinhard Muskens, and Elias Thijsse, editors, *Computing Meaning Vol.2*, pages 11–28. Kluwer, 2001.

[BCS+04] J. Bos, S. Clark, M. Steedman, J.R. Curran, and Hockenmaier J. Wide-Coverage Semantic Representations from a CCG Parser. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING '04)*, Geneva, Switzerland, 2004.

[BO02] Johan Bos and Tetsushi Oka. An Inference-based Approach to Dialogue System Design. In Shu-Chuan Tseng, editor, *COLING 2002. Proceedings of the 19th International Conference on Computational Linguistics*, pages 113–119, Taipei, Taiwan, 2002.

[Bos01] Johan Bos. DORIS 2001: Underspecification, Resolution and Inference for Discourse Representation Structures. In Patrick Blackburn and Michael Kohlhase, editors, *ICoS-3, Inference in Computational Semantics*, pages 117–124, 2001.

[Bos05] Johan Bos. Towards wide-coverage semantic interpretation. In *Proceedings of Sixth International Workshop on Computational Semantics IWCS-6*, pages 42–53, 2005.

[CC04] S. Clark and J.R. Curran. Parsing the WSJ using CCG and Log-Linear Models. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL '04)*, Barcelona, Spain, 2004.

[CS03] K. Claessen and N. Sörensson. New techniques that improve mace-style model finding. In *Model Computation – Principles, Algorithms, Applications (Cade-19 Workshop)*, Miami, Florida, USA, 2003.

[DN98] Hans De Nivelle. A Resolution Decision Procedure for the Guarded Fragment. In *Automated Deduction - CADE-15. 15th International Conference on Automated Deduction*, pages 191–204. Springer-Verlag Berlin Heidelberg, 1998.

[FK99] Andreas Franke and Michael Kohlhase. System description: Mathweb, an agent-based communication layer for distributed automated theorem proving. In *CADE'99*, 1999.

[HS02] J. Hockenmaier and M. Steedman. Generative Models for Statistical Parsing with Combinatory Categorial Grammar. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, Philadelphia, PA, 2002.

[KR93] H. Kamp and U. Reyle. *From Discourse to Logic; An Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and DRT.* Kluwer, Dordrecht, 1993.

[KW99] Karsten Konrad and D.A. Wolfram. System description: Kimba, a model generator for many-valued first-order logics. In *16th International Conference on Automated Deduction CADE-16*, 1999.

[McC98] W. McCune. Automatic Proofs and Counterexamples for Some Ortholattice Identities. *Information Processing Letters*, 65(6):285–291, 1998.

[MP96] W. McCune and R. Padmanabhan. *Automated Deduction in Equational Logic and Cubic Curves*. Lecture Notes in Computer Science (AI subseries). Springer-Verlag, 1996.

[RV02] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2–3), 2002.

[VdS92] R.A. Van der Sandt. Presupposition Projection as Anaphora Resolution. *Journal of Semantics*, 9:333–377, 1992.

[WAB+99] Christoph Weidenbach, Bijan Afshordel, Uwe Brahm, Christian Cohrs, Thorsten Engel, Enno Keen, Christian Theobalt, and Dalibor Topic. System description: Spass version 1.0.0. In Harald Ganzinger, editor, *16th International Conference on Automated Deduction, CADE-16*, volume 1632 of *LNAI*, pages 314–318. Springer-Verlag, Berlin, 1999.

A Flexible DL-based Architecture for Deductive Information Systems

Michael Wessel, Ralf Möller
Hamburg University of Technology (TUHH), Germany
{mi.wessel, r.f.moeller}@tuhh.de

1 Introduction

Description Logics (DLs) are nowadays an accepted standard for decidable knowledge representation. DLs also provide the theoretical foundation for representing and reasoning with ontologies as well as for the Semantic Web. Thus, DL systems (e.g., Fact++, Pellet, RacerPro) and DL-based technology in general will play an increasingly important role for building the next generation of deductive, ontology-based information systems.

The RacerPro description logic system [HM01a] implements the very expressive DL $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$, also known as $\mathcal{SHIQ}(D^-)$ [HST99, HM01b]. RacerPro's most prominent feature is the support for so-called *ABoxes* which allow for the representation of a "concrete state of the world" in terms of individuals and relationships. In the following, we assume familiarity with DLs as well as DL systems.

In order to provide the information technology backbone for next generation information systems (IS), current DL systems still have a long way to go. Nowadays, building ontology-based IS with a DL system is still a non-trivial task: One the one hand, DLs somehow live in their own "realm" and are thus not really interoperable with the rest of "more conventional" IS infrastructure (e.g., existing relational databases). One the other hand, only recently issues such as persistency and powerful *query languages (QLs)* have been considered and incorporated into DL systems. DLs itself have their deficiencies as well and are thus not a panacea for arbitrary information representation and (ontology-based) retrieval tasks: Due to the complexity of the inference problems, scalability (in the average case) is not easy to achieve and nowadays only achievable for *knowledge bases (KBs)* which use simple DLs. Standard DLs are well suited for the representation of (and reasoning about) semi-structured (or even unknown/indeterminate) information, but things become more complicated if, say, n-ary relationships or special "non abstract" domains such as space are considered. Then, either non-standard DLs or complicated logical encodings are needed. For non-standard DLs, no working systems exist, and complicated logical encodings are likely to decrease the performance and complicate the information handling and maintenance.

To get working systems on time *today*, we must thus agree upon pragmatic solutions. This also implies that existing technology (i.e., existing DL system such as RacerPro) must be exploited and possibly extended for the task of IS building. Due to the intellectual inherent complexity of the field, application and system studies on how to use DL system in real-world IS scenarios are thus of utmost importance in order to provide guidance.

In this paper, we consider the IS domain of deductive geographic information systems (GIS), which can be called a "non-standard domain". We describe the problems encountered and pragmatic solutions found during the endeavor of *building an ontology-based query answering system for digital city maps*, the DLMAPS system [Wes03a]. We use RacerPro as a DL system, which can be called an empirically successful system, since it is widely used.[1] In this IS domain of digital city maps, we must

1. pragmatically solve the map representation problem, especially regarding the spatial and thematic aspects,

2. provide an expressive *spatio-thematic query language* (QL) which supports ontology-based query answering (w.r.t a "city map background ontology"). This QL must be able to address spatial as well as thematic aspects of the map objects.

The paper provides the following contributions:

We describe how an existing DL system such as RacerPro can be used for building an ontology-based IS in such a "non-standard" IS domain for which a standard DL such as $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ is not very well suited. This mainly concerns the representation of the spatial aspects of the maps, which we call the "spatial representation problem" in the following. We will demonstrate which representation options are available in this setting.

We demonstrate what can be done with nowadays available state-of-the art DL (and Semantic Web) technology such as RacerPro, and identify and motivate and design pragmatic extensions to tackle the spatial representation problem. Making this explicit by means of this paper will help other users to build their own IS with enabling DL technology by exploiting similar "design patterns" as we did.

Spatial representations are, in principle, possible with expressive spatial *concrete domains (CDs)* [HLM99, LM05] or specialized DLs [Wes03b] or spatial modal logics [LW04]. However, no optimized mature DL system supporting these non-standard DLs exist, and building a DL system which supports them is a non-trivial task.

Even if the spatial representation is achieved, then additionally an expressive spatio-thematic QL is needed for the IS. Designing and implementing such a QL is again a non-trivial task. We will demonstrate (from a pragmatic perspective) how such a QL can be designed. Moreover, the QL is also implemented and available for other users.

RacerPro has been extended in 2003 by an expressive QL called nRQL [WM05, KG06]. Given that we have solved the spatial representation problem, we can use and extend nRQL by "spatial atoms" to get the desired spatio-thematic query language. The nRQL query answering engine (which is an integral part of RacerPro) can be called an empirically successful system, since it is used by many RacerPro users. At the time of this writing, this engine is (to the best of our knowledge) the only optimized ABox query answering engine which provides complex ABox queries and also addresses issues such as life cycle management of queries (queries are managed as objects which have a state), concurrent and incremental query answering, version control of query answers, defined queries, etc.

Another contribution of this paper is the identification and description of (software) abstractions which we claim can be useful for other developers of ontology-based IS with

[1] And is commercially distributed by the start-up company Racer Systems.

DL system components as well: We have based the DLMAPS system on the so-called *substrate data model.* This data model serves as a *layer of indirection and abstraction*, shielding the IS from the details of the used DL system (a kind of "semantic middle ware"). But more importantly, the substrate data model enables us to attach or associate additional information on or with ABox individuals, for which no encoding in an ABox is possible or appropriate. We will show how the substrate data model can be used to solve the spatial representation problem of the map. The resulting representations will by hybrid; thus, a *hybrid QL* will be needed to combine the information distributed on the different substrate layers.

The paper is structured as follows. We first describe the IS domain of digital city maps, the concrete map data we use, and the idea of ontology-based queries to such city maps. We then present various options how to represent the maps. Next we describe the nRQL ABox QL which plays a crucial role here, since nRQL is the QL which is extended to become a hybrid spatio-thematic QL in this paper. Finally, the resulting QL is presented. Then comes the conclusion.

2 The Scenario - Ontology-Based Queries To a City Map

For what follows, we call the TBox or ontology of the DLMAPS IS the *intensional component,* whereas the actual map data is kept in the *extensional component. Ontology-based query answering* then means that the (defined) vocabulary from the intensional component can be used in queries to retrieve the desired information (by means of query answers) from the extensional component. The *query answering component* is responsible for computing these answers.

The data in the DLMAPS IS are digital vector maps from the city of Hamburg provided by the land surveying office ("Amt für Geoinformation und Vermessungswesen Hamburg"); these maps are called the DISK ("Digitale Stadtkarte"). Part of the DISK is visualized by the Map Viewer component of our system in Fig. 1. Each map object (also called *geographic feature*) is *thematically annotated.* The basic *thematic annotation (TA)* has been established by the land surveying office itself. The TA says something about the "theme" or semantics of the map object. Simple symbols/names such as "green area", "meadow", "public park", "lake" are used. A few hundred TAs are used and documented in a so-called *thematic dictionary (TD)*, which is GIS-typically organized in thematic layers (e.g., one layer for infrastructure, one for vegetation, etc.).

Sometimes, only highly specific TAs are available, such as "cemetery for non Christians", and generalizing "common sense" vocabulary such 'as "cemetery" is often missing. This is unfortunate, since it prevents the usage of common sense vocabulary for query formulation. We can repair this defect by adding a background ontology (in the form of a TBox) providing generalizing TAs by means of taxonomic relationships. On the other hand, *defined concepts* ("if and only if") can be added and exploited to *automatically enrich* the given basic annotations. Thus, we might define our own needed TA "public park containing a lake" as a "park which is public which contains a lake" with a TBox axioms such as

$$public\_park\_containing\_a\_lake \doteq park \sqcap public \sqcap \exists contains.lake$$

or

$$bird\_sanctuary\_park \doteq park \sqcap \forall contains.\neg building$$

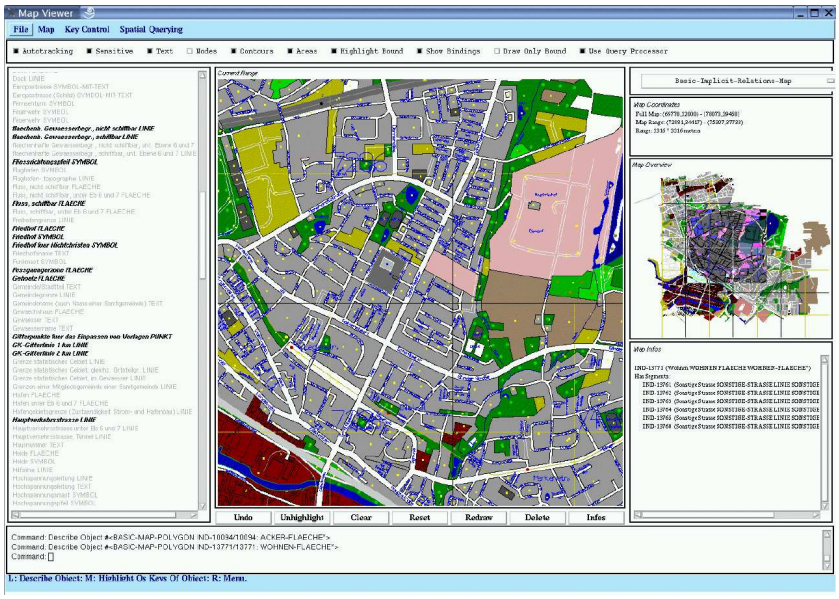

Figure 1: The Map Viewer of the DLMAPS System

and we might want to retrieve all instances of these concepts. This is what ontology based query answering is all about. Inference is required to obtain these instances, since there are no *known* instances of *public_park_containing_a_lake* (this is not among the basic TAs provided by the land surveying office). For simple queries, *instance retrieval queries* might be sufficient as a QL. However, in our scenario one must retrieve *constellations*[2] *of map objects* which satisfy a *complex query expression;* thus, a QL with variables is needed.

A definition such as *public_park_containing_a_lake* refers to *thematic as well as to spatial aspects* of the map objects:

• **Thematic aspects:** the name of the park, that the park is public, the amount of water contained in the lake, etc.

• **Spatial aspects:** the *spatial attributes* such as the area of the park (or lake), the concrete shape, qualitative *spatial relationship* such as "contain", quantitative (metric) spatial relationships such as the distance between two objects, etc.

We use the following terminology: a *thematic concept* refers only to thematic aspects, similarly a *spatial concept* solely to spatial aspects, whereas a *spatio-thematic concept* refers to both. We also talk of *thematic*, *spatial* and *spatio-thematic queries*. A strict separation might be difficult sometimes.

Thus, there are different thematic and spatial aspects one would like to represent and address with the spatial QL. Since the concrete geometry is given by the map, the spatial aspects of the map objects are in principle intrinsically represented and available. This mainly concerns the spatial relationships which are depicted in the map. However, also attributes like the area etc. can in principle be computed from the geometry (although this will not be very accurate). A function that exploits the geometry to dynamically check the requested spatial aspect (i.e., spatial property or

[2]We use the term "constellation" to stress that a certain spatial arrangement of map objects is requested with a query.

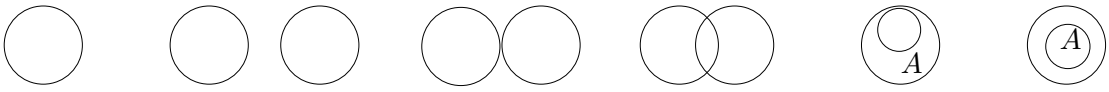

Figure 2: RCC8 base relations: EQ = Equal, DC = Disconnected , EC = Externally Connected, PO = Partial Overlap, TPP = Tangential Proper Part, $NTPP$ = Non-Tangential Proper Part. All relations with the exception of TPP and $NTPP$ are symmetric; the inverse relations of TPP and $NTPP$ are called $TPPI$ and $NTPPI$.

relation) is called an *inspection method* in the following.

It is obvious that *qualitative* spatial descriptions are of great importance. On the one hand, they are needed in order to be able to define concepts in the TBox such as "public park containing a lake". On the other hand, they are needed in the spatial QL ("retrieve all public parks containing a lake"). A popular and well-known set of qualitative spatial relationships is given by the RCC8 relations, see Fig. 2.

One the other hand, since the concrete geometry is given by means of the map, in principle, *no qualitative representation is needed* in the extensional component. However, if we want to use a RacerPro ABox for the extensional component, then the spatial representation options are limited, and we must necessarily make use qualitative descriptions for the map representation, see below.

The DLMAPS system will support ontology-based spatio-thematic conjunctive queries to the DISK. To give a first impression of what will be possible, suppose we want to identify the chemical plants in the DISK which might be responsible for a chemically contaminated lake:

$$\begin{aligned} &ans(?lake, ?park, ?creek, ?industrial\_area, ?chemical\_plant) \leftarrow \\ &\quad lake(?lake), chemically\_contaminated(?lake), \\ &\quad park(?park), public(?park), contains(?park, ?lake), \\ &\quad creek(?creek), flows\_in(?creek, ?lake), \\ &\quad crosses(?creek, ?industrial\_area), \\ &\quad contains(?industrial\_area, ?chemical\_plant), unreliable(?chemical\_plant). \end{aligned}$$

We assume that the reader has an intuitive understanding of such a query. The different conjuncts of such a query are called *query atoms* in the following. A *ground query atom* is an atom in which the free variables have been substituted with (satisfying) individuals resp. "constants". We formalize these notions subsequently. An atom with one variable is equivalent to a simple *instance retrieval query*.

3 Representing and Querying the DISK

It is clear that the kind of representation we will devise for the DISK in the extensional component will also determine what we can query, and how we can query (i.e., which techniques to be used for computing the query answers). Without doubt, the thematic aspects of the DISK map objects can be represented satisfactory with a standard DL.

To solve the spatial representation problem of the DISK in the extensional component, we consider three representation options and analyze their impacts:

• **Representation Option 1 – Use a Single ABox:** We can represent "as much spatial aspects as possible" in a $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ ABox. Regarding the spatial

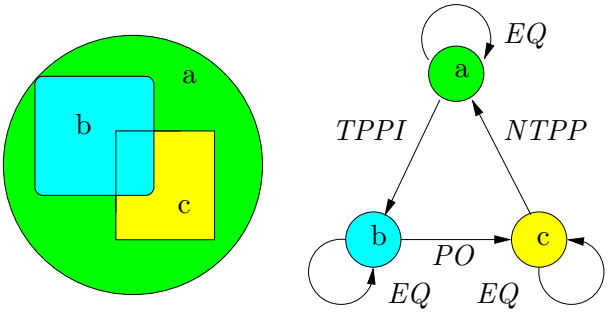

Figure 3(a): Concrete geometric scene and corresponding RCC8 network (inverse edges omitted)

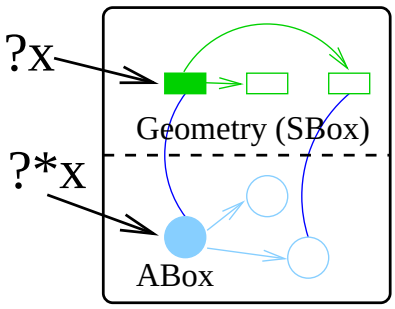

Figure 3(b): Hybrid map substrate: variables are bound in parallel

relationships, we can only represent *qualitative* relationships. From the concrete map geometry, we can compute a so-called *RCC network* and represent this by means of *RCC role assertions* in the ABox, e.g. $(i, j) : TPPI$ etc. In Abb. 3(a) a "scene" and its corresponding RCC8 network is depicted. Such a network will always take the form of a an edge labeled complete graph, a so-called K_n (see graph theory), due to the *JEPD property* of the RCC base relations: The base relations are *j*ointly *e*xhaustive, *p*airwise *d*isjoint). Moreover, an RCC network derived from a geometric scene will always be *RCC consistent* (see below).

Moreover, selected spatial attributes such as area and length can be represented in the CD of the ABox; we then use *concept assertions* such as $i : \exists(has\_area).=_{12.345}$.

Moreover, since the represented spatial aspects are accessible to RacerPro, this supports richer spatio-thematic concept definitions in the TBox, for example

$$public\_park\_containing\_a\_lake \doteq park \sqcap public \sqcap \exists contains.lake$$

(we are using $\exists contains.lake$ instead of $(\exists TPPI.lake) \sqcup (\exists NTPPI.lake)$). Obviously, an individual i in the ABox can only be recognized as an instance of that concept if RCC role assertions are present in that ABox.

In principle, the specific properties of qualitative spatial relationships resp. roles cannot be captured completely within $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ (we will elaborate this point below when we discuss qualitative spatial reasoning with the RCC substrate). This means that the computed taxonomy of the TBox will not correctly reflect the "real" subsumption relationships, and that the TBox might be incoherent without being noticed. We believe that careful modeling can avoid this. Another option is to compute the *taxonomy* of the TBox "offline" with a specialized (non-optimized, prototypically) DL system with space semantics such as an $\mathcal{ALCI}_{RCC8}$ prover (see [Wes03b]), even though this DL is undecidable, [LW04].[3] The deduced implied subsumption relationships can be made syntactically explicit by means of additional TBox axioms, and the augmented TBox can replace the original one.

Much more importantly in our setting is the observation that *ontology-based query answering* can still be achieved in a way that correctly reflects the semantics of the spatial relationships with RacerPro. Consider the *instance retrieval query* $(?x, public\_park\_containing\_a\_lake)$ on the ABox

[3] One could at least try to compute the taxonomy.

$$\mathcal{A} = \{i : park \sqcap public, k : lake, j : meadow, (i,j) : TPPI, (j,k) : NTPPI\}.$$

If this ABox has been computed from the concrete geometry of the map, then it must also contains $(i,k) : NTPPI$, since each RCC network which has been computed from a spatial constellation which shows $(i,j) : TPPI$ and $(j,k) : NTPPI$ must also show $(i,k) : NTPPI$. In order to retrieve the instances of *public_park_containing_a_lake*, we check each individual separately. Let us consider i. Checking that i is an instance of *public_park_containing_a_lake* is reduced to checking the unsatisfiability of the ABox

$$\begin{aligned}\mathcal{A} \cup\ & \{(i,k) : NTPPI\} \cup \\ & \{i : (\neg park \sqcup \neg public \sqcup ((\forall NTPPI.\neg lake) \sqcap (\forall TPPI.\neg lake)))\}\end{aligned}$$

This ABox is obviously unsatisfiable and thus, i is a *public_park_containing_a_lake*.

Regarding concepts that "contain or imply" *universal role or number restrictions,* we can answer queries completely only if we turn on a "closed domain reasoning mode", we must *close the ABox w.r.t. the RCC role assertions* and *enable the unique name assumption (UNA)* in order to keep the semantics of the RCC roles.

To close the ABox $\mathcal{A}$ w.r.t. the RCC role assertion, we count for each individual $i \in \mathsf{individuals}(\mathcal{A})$ and each RCC role R the number of R-successors, $n = |\{\, j \mid\ (i,j) : R \in \mathcal{A}\,\}|$, and add so-called *number restrictions* $i : (\leq\ R\ n) \sqcap (\geq\ R\ n)$ to $\mathcal{A}$. This assertion is satisfied in an interpretation $\mathcal{I}$ iff $n = \{\, j \mid\ (i^{\mathcal{I}}, j) \in R^{\mathcal{I}}\,\}$; thus, i must have exactly n R successors in every model.

In combination with the UNA, this enables a *closed domain reasoning* on the individuals which are mentioned in the RCC role assertions and thus prevents the reasoner from generating "new anonymous RCC role successors" in order to satisfy an existential restriction such as $\exists NTPPI.lake$. In order to satisfy $\exists NTPPI.lake$, it must then necessarily *reuse* one of the existing individuals in the ABox, thus the domain is "closed" [Wes03a]. Let us demonstrate this using the concept

$$bird\_sanctuary\_park \doteq park \sqcap \forall contains.\neg building.$$

Assuming that both *lake* and *meadow* imply $\neg building$, we can show that i is an instance of a *bird_sanctuary*, since the ABox

$$\begin{aligned}\mathcal{A}\ \cup\ & \{(i,k) : NTPPI\} \cup \\ & \{i :\leq_1 TPPI \sqcap \geq_1 TPPI, i :\leq_1 NTPPI \sqcap \geq_1 NTPPI, \ldots\} \cup \\ & \{i : (\neg park \sqcup ((\exists TPPI.building) \sqcap (\exists NTPPI.building)))\}\end{aligned}$$

is again unsatisfiable, because the alternative $i : \neg park$ immediately produces an inconsistency. Thus, the alternative $\{i : (\exists TPPI.building) \sqcap (\exists NTPPI.building)\}$ is considered. Due to $i :\leq_1 TPPI \sqcap \geq_1 TPPI$, only j can be used to satisfy $\exists TPPI.building$, and only k to satisfy $\exists NTPPI.building$. Since $j : meadow$ and thus $j : \neg building$, $k : lake$ and thus $k : \neg building$, the ABox must be unsatisfiable.

Thus, we have argued that spatio-thematic ontology-based query answering can be done on such an ABox representation of the DISK.

Moreover, we must not resort to simple instance retrieval queries, but can use a powerful ABox QL such as nRQL. With nRQL we can pose queries (here in mathematical syntax) such as

$$\begin{aligned}ans(?living\_area, ?park, ?lake) \leftarrow\ & \\ & living\_area(?living\_area), park(?park), \\ & contains(?park, ?lake), adjacent(?living\_area, ?park), \\ & (\forall adjacent.\neg industrial\_area)(?living\_area)\end{aligned}$$

Note that $\forall adjacent.\neg industrial\_area$ is a complex query concept; we could have put

a concept definition such as $healthy\_living\_area \doteq \forall adjacent.\neg industrial\_area$ into the TBox and used the atom $healty\_living\_area(?living\_area)$ instead. The concrete nRQL syntax will be shown and used below.

However, the discussed ABox representation has the *following drawbacks:*

• **1.** The size of the generated ABoxes is huge. Since the RCC network is explictly encoded in the ABox, the number of required role assertions is quadratic in the number of map objects (resp. individuals).

• **2.** Most spatial aspects cannot be handled that way. For example, distance relations are very important for map queries. It is thus not possible to retrieve all subway stations within a distance of 100 meters from a certain point.

• **3.** Query processing will not be efficient. More efficient query processing can be done if spatial index structures are added.

• **4.** In the DLMAPS system, the geometric representation of the map is needed anyway, at least for presentation purposes. Thus, from a non-logical point of view, the ABox cannot be the only representation used by the extensional component of such a system. Thus, it seems plausible to exploit the representation for query answering as well.

• **5.** Most importantly, we have demonstrated that this kind of ontology-based query answering only works if the domain is closed. However, DL systems and thus RacerPro are not really good at closed domain reasoning, since the *open domain assumption* is made in DLs. In contrast, since the geometry of the map is completely specified, there is neither unknown nor underspecified spatial information. This motivates the classification of the map as "spatial data", in contrast to "spatial information". Thus, regarding the spatial aspects, deduction or logical inference is not needed. Instead, *model checking* would be sufficient regarding the spatial aspects. All these points thus motivate

• **Representation Option 2 – Use a Map Substrate:** Due to the problems with spatio-thematic concepts and since closed domain reasoning is anyway all that we can achieve here, it seems more appropriate to represent the spatial aspects *primarily* in a different representation layer which we can *associate* with an ABox, a kind of "spatial database", that contains instances of spatial datatypes (polygons etc.). We already mentioned that the geometry of the map must be represented in the extensional component anyway (at least for presentation purposes). This also reflects appropriately that there is neither unknown nor underspecified information regarding the spatial aspects of the map. Everything is explictly given (the "logical theory" of the map is thus complete). This is a reasonable assumption in this scenario.

In the following, this spatial medium is called an *SBox.* If we say that spatial aspects are *primarily* represented in the SBox, then this does *not* necessarily exclude the (additional) representation possibilities of dedicated spatial aspects in the ABox as just discussed. The resulting hybrid $(SBox, ABox)$ representation is illustrated in Fig. 3(b), we call it a *map substrate.* It is shown that some ABox individuals have corresponding instances in the SBox, and vice versa, since the mapping function is partial and injective. This association / mapping function is called the $*$-function in the following.

If spatial information about the map is now primarily kept in the SBox, then it is no longer available for ABox reasoning. Thus, nRQL (or instance retrieval) queries are no

longer sufficient to address the spatial aspects – we will thus extend nRQL to become a hybrid spatio-thematic QL, offering also *spatial query atoms* to query the SBox: SnRQL. The SnRQL query answering engine will combine the retrieved results from the SBox and ABox part of the hybrid map substrate representation. The purely thematic part of the query will be a plain nRQL query which will be answered on the ABox, and the spatial part of the query will contain spatial atoms which are evaluated on the SBox.

Since the SBox represents the geometry of the map, it can evaluate the requested spatial aspects on the SBox "on the fly" during query evaluation by means of *inspection methods* (see Page 5). The SBox provides a *spatial index*, supporting the efficient computation of spatial relationships by means of spatial selection operations. Computed spatial aspects can also be made explicit and "materialized" in order to avoid repeated re-computation (e.g., computed RCC relations can be materialized as edges).

By means of query rewriting and expansion, the hybridness of SnRQL can be made transparent for the users. Ideally, a user can abstract from the details of the DISK representation in the extensional component of the DLMAPS system. He/she must not known how and where a special aspect of the DISK is physically represented (in the ABox or SBox) in order to be able to formulate queries which return the intended results. The exploited query expansion and rewriting procedures are currently hard-coded, though.

In order to provide these flexible representation options, to "break out" of the strict DL ABox framework, but nevertheless keep a clear mathematical semantics, we base the DLMAPS IS on a semi-structured graph-based *substrate data model* which provides this flexibility. As explained, it serves both as a mediator and software abstraction ("semantic middle ware"), but also enables us to formally describe and combine different representation layers to build hybrid representations such as a map substrate. Since ABoxes play a role here as well, the data model must also generalize the notion of an ABox:

Definition 1 A *substrate* is an edge- and node-labeled directed graph (V, E, L_V, L_E), with V being the set of *substrate nodes*, and E being a set of *substrate edges.* The node labeling function $L_V : V \rightarrow \mathcal{L}_\mathcal{V}$ maps vertices to descriptions in an appropriate node description language $\mathcal{L}_\mathcal{V}$, and likewise for $L_E : E \rightarrow \mathcal{L}_\mathcal{E}$, where $\mathcal{L}_\mathcal{E}$ is an edge description language. ■

The languages $\mathcal{L}_\mathcal{V}$ and $\mathcal{L}_\mathcal{E}$ are not fixed and can be seen as *subsets of first-order predicate logic* (in appropriate syntax). For example, we can consider an ABox as a substrate if we identify V with the ABox individuals, E with the pairs of individuals mentioned as arguments in role assertions in that ABox, $\mathcal{L}_\mathcal{V}$ with the set of $\mathcal{ALC}$ concepts, and $\mathcal{L}_\mathcal{E}$ with the set of $\mathcal{ALC}$ role names closed under conjunction. Using the first-order perspective, V is a set of constant symbols, and L_V and L_E are *indexing functions.* Obviously, an encoding of $\mathcal{L}_\mathcal{E}$ and $\mathcal{L}_\mathcal{V}$ into first-order logic is possible. Moreover, an associated TBox manifests itself in additional first-order sentences which are assumed to be "intrinsically encoded" into the substrate as well.

We do not claim that this data model is interesting from a theoretical perspective; its "generic definition" is of course also its weakness. Thus, it must be specifically *instantiated.* However, the given definition enables a formal specification of the semantics of our substrate representation and query answering framework on which we have based the DLMAPS system.

The data model is somehow inspired by the work on $\mathcal{E}$-Connections [KLWZ04] or the tableaux data structure [HST99], as well as by RDF. However, we feel that we need more flexibility than, e.g., $\mathcal{E}$-Connections or RDF can provide; e.g., RDF does not allow us to describe the geometry of map object. Moreover, a substrate is not simply a "graph database", since DL ABoxes are considered part of the framework and thus, inference is required in order to answer queries. We have the feeling that first-order logic is all we need here in order to formally describe the data model, it semantics and its inference services as well as the *generic substrate query language* (see below).

As just explained, an ABox can be seen as a specialized substrate. The substrate establishes a graph perspective on the ABox (by means of the indexing functions). Moreover, an SBox can be seen as a specialized substrate as well. Here, the nodes are instances of spatial datatypes. We need not leave the framework of first-order logic if we agree that the geometry of such nodes can be described using an appropriate *geometry description language*. We do not want to go into the details (of logical encoding of instances of spatial data types) here.

In order to make use of the substrate representation, a *substrate QL* is needed. The *substrate QL framework* enables the creation of specialized substrate QLs which can be tailored for special kinds of substrates (e.g., ABox, SBoxes). This QL framework is based on the general notion of *ground query atom entailment*. All that matters is that a *notion of logical entailment* ($\models$) between a substrate S and a *ground query atom for* S is defined and decidable. Thus, give the (unary and binary) ground query atoms $P(i)$ and $Q(i,j)$ for $i,j \in V$, $S \models P(i)$ and $S \models Q(i,j)$ must be defined and decidable.

The $\models$ relation is, in principle, the standard first-order one. The $\models$ relation defined for a substrate can *intrinsically encode a rich background theory,* for example, the additional axioms of a background TBox, or *domain closure axioms,* or even the full set of *Clark completion axioms.* These additional axioms are thus not explicitly represented in the substrate as sentences, but are assumed to be part of the inherent structure of the substrate. The models relation is thus further constrained. For example, in the case of an SBox, deciding the $\models$ relation boils down to simple *model checking*. We will give an example for such a background theory when we discuss the RCC substrate.

The substrate QL framework provides a great flexibility, extensibility and adaptability, since specialized query atoms can be tailored for specific instantiations of the substrate data model. If only a notion of entailment is defined and decidable for these specialized atoms, then the substrate query answering engine immediately supports the evaluation of these atoms. In fact, it suffices to overload a single CLOS multimethod `entails-p`. Moreover, decidability of the QL is automatically guaranteed by the framework. *Complex substrate queries* are combined from query atoms by means of body constructors. The substrate QL framework provides the following constructors: negation as failure (NAF, "\"), conjunction (";" and "$\wedge$"), union ("$\vee$"), as well as a *projection operator* ("π"). The semantics (and decidability) of these generic QL operators will become clear if we consider nRQL, which is a specialized substrate QL tailored for RacerPro ABoxes; it thus shares the same semantics for the body constructors as all substrate QLs.

Definition 2 A *hybrid substrate* is a triple $(S_1, S_2, *)$, with S_i, $i \in \{1,2\}$ being substrates $(V_i, E_i, L_{V_i}, L_{E_i})$ using $\mathcal{L}_{\mathcal{V}i}$ and $\mathcal{L}_{\mathcal{E}i}$, $*$ being a partial and injective function

$* : V_1 \mapsto V_2$. ■

The DLMAPS system is designed in such a way to work on ABoxes, SBoxes, or *map substrates:*

Definition 3 A *map substrate* is a hybrid substrate $(S_1, S_2, *)$, where S_1 is an SBox, and S_2 is an ABox (substrate). ■

Given the hybrid representation, a *hybrid query* now contains two kinds of query atoms: Those for S_1, and those for S_2. In order to distinguish atoms meant for S_1 from atoms meant for S_2 easily, we simply prefix variables in query atoms which for S_2 with a "?*" instead of "?"; the same applies to constants / individuals. Intuitively, the *bindings* which will be established for variables must reflect the $*$-function: If $?x$ is bound to $i \in V_1$, then $? * x$ will automatically be bound to $*(i) \in V_2$, and vice versa (w.r.t. $*^{-1}$). Such a binding is called $*$-*consistent.* We will only consider $*$-consistent bindings. The notion of $*$-consistent bindings is also depicted in Fig. 3(b).

Assume we use a map substrate for the DISK representation now. Then, the nRQL example query given above will become a hybrid SnRQL query; nRQL atoms now use $*$-prefixed variables (since the ABox is S_2, and the SBox is S_1):

$$
\begin{array}{l}
ans(?living\_area, ?park, ?lake) \leftarrow \\
\quad living\_area(? * living\_area), park(? * park), \\
\quad contains(?park, ?lake), adjacent(?living\_area, ?park), \\
\quad \backslash \ (\ \pi(?living\_area) \ (\ adjacent(?living\_area, ?industrial\_area), \\
\qquad\qquad\qquad\qquad industrial\_area(? * industrial\_area)))
\end{array}
$$

Please note that the last conjunct in the query is the "equivalent" of the nRQL atom $(\forall adjacent.\neg industrial\_area)(?living\_area)$.

By means of the definition of the semantics of the QL (see below) it will become clear that $\pi(? * living\_area)$ $(adjacent(? * living\_area, ? * industrial\_area), \ldots)$ is in fact equivalent to (resp. has the same extension as) $ans(? * living\_area) \leftarrow adjacent(? * living\_area, ? * industrial\_area), \ldots$. This subquery therefore retrieves all map objects which are adjacent to an industrial area. The NAF operator *complements* this subquery result (see next Section on nRQL semantics). Thus, the binding to $?living\_area$ must be an instance of *living_area* which *does not have a (known) adjacent industrial area.* This implies that that all known adjacent areas are not industrial areas.

• **Representation Option 3 – Use an ABox + RCC Substrate:** We have argued that in general we cannot completely capture the inherent properties of the RCC relations in an $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ ABox, since $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ lacks the required expressivity. However, we have argued that (see Option 1) at least for ontology-based spatio-thematic query answering a $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ ABox can still be used, given that the RCC network in the ABox was computed from a concrete map. The RCC network is thus complete (lacks no edges) and automatically RCC consistent. Moreover, we must enable closed domain reasoning for the RCC roles.

In order to make a comparable query answering functionality available to other users of the RacerPro system without having to add the whole SBox functionality to RacerPro (spatial datatypes and spatial indexes, etc.), we devise yet another kind of substrate, the *RCC substrate*, which captures the semantics of the RCC relations by exploiting

techniques from *qualitative spatial reasoning.* Users of RacerPro can associate an ABox $\mathcal{A}$ with an RCC substrate $\mathcal{RCC}$ by means of a hybrid substrate $(\mathcal{A}, \mathcal{RCC}, *)$ and query this hybrid substrate with *nRQL + RCC query atoms* (see below). Note that, unlike a map substrate, there the ABox is S_1 and thus the "primary" substrate.

The RCC substrate is basically an RCC network consistency checker which can decide *(relational) consistency of RCC networks* and *entailment of RCC relations* resp. RCC ground query atoms:

Definition 4 An RCC substrate $\mathcal{RCC}$ is a substrate such that V is a set of RCC nodes with $\mathcal{L}_\mathcal{V} = \emptyset$, and $\mathcal{L}_\mathcal{E} = 2^{\{EQ,DC,EC,PO,TPP,TPPI,NTPP,NTPPI\}}$. ■

An edge label represents a *disjunction of RCC base relations*, representing coarser or even unknown knowledge regarding the spatial relation. Disjunctions of base relations are thus RCC relations as well. The *properties* of the RCC relations are captured by the so-called JEPD property (see Page 6) as well as the so-called *RCC composition table.* This table is used for solving the following basic inference problem: **Given: RCC relations** $R(a,b)$ **and** $S(b,c)$**. Question: Which relation** T **holds between** a **and** c**?** The table thus lists, at column for base relation R and row for base relation S, the possible values for T. In general, T will not be a base relation, but a set: $\{T_1, \ldots T_n\}$. The RCC table is given as a set $\mathcal{RCC}_\mathcal{T}$ of sentences of the form $\{R \circ S = \{T_1, \ldots, T_n\}, \ldots\}$.

An RCC network $\mathcal{RCC}$ (viewed as set of first-order ground atoms) containing only base relations is said to be consistent iff it satisfies a number of first-order sentences:

$$
\begin{aligned}
\mathcal{RCC}' = \; & \mathcal{RCC} \cup \\
& \{ \forall x,y,z.R(x,y) \wedge S(y,z) \rightarrow T_1(x,z) \vee \cdots \vee T_n(x,z) \mid \\
& \qquad\qquad\qquad\qquad R \circ S = \{T_1, \ldots, T_n\} \in \mathcal{RCC}_\mathcal{T} \} \cup \\
& \{\forall x,y. \bigvee_{R \in \mathrm{RCC}} R(x,y)\} \cup \\
& \{\forall x,y. \bigvee_{R,S \in \mathrm{RCC}, R \neq S} R(x,y) \wedge \neg S(x,y)\} \cup \\
& \{\forall x.EQ(x,x)\}
\end{aligned}
$$

For example, the network $\mathcal{RCC} = \{NTPP(a,b), DC(b,c), PO(a,c)\}$ is inconsistent, because if a is contained in b (atom $NTPPI(a,b)$), and b is disconnected from c (atom $DC(b,c)$), then a must be disconnected from c as well. The *RCC8 composition table* contains the axiom $NTPP \circ DC = DC$. Thus, $\mathcal{RCC}' \models DC(a,c)$, which contradicts $PO(a,c)$, due to the JEPD property. *Entailment of RCC relations* or RCC ground query atoms can be *reduced to inconsistency checking:* $\mathcal{RCC}' \models R(a,b)$ **iff** $\mathcal{S} \cup (\{EQ, DC, EC, PO, TPP, TPPI, NTPP, NTPPI\} \setminus R)$ **is not consistent.** An RCC network is consistent iff at least one of its *configurations* is consistent. A configuration of an RCC network is obtained by choosing (and adding) one disjunct from every non-base relation in that network. Thus, a configuration contains only base relations. Consider now $\mathcal{RCC} = \{NTPP(a,b), DC(b,c)\}$. We have $\mathcal{RCC}' \models DC(a,c)$, since $\mathcal{RCC}' \cup \{(EQ \vee EC \vee PO \vee TPP \vee TPPI \vee NTPP \vee NTPPI)(a,c)\}$ is inconsistent, because its *configurations* $\mathcal{RCC}' \cup \{EQ(a,c)\}$ $\ldots$ $\mathcal{RCC}' \cup \{NTPPI(a,c)\}$ are all inconsistent.

Since the RCC substrate defines a notion of logical entailment, the semantics of the RCC relations will be correctly captured for query answering: Consider the hybrid substrate $(\mathcal{A}, \mathcal{RCC}, *)$

$$\mathcal{A} = \{hamburg : german\_city, paris : french\_city, france : country, germany : country\}$$

with

$$\mathcal{RCC} = \{NTPP(*hamburg, *germany), EC(*germany, *france), NTPP(*paris, *france)\}$$

and with the obvious (trivial) mapping $*$

$$* = \{(hamburg, *hamburg), (paris, *paris), (france, *france), (germany, *germany)\}.$$

Then, the query

$$ans(?city1, ?city2) \leftarrow city(?city1), city(?city2), DC(? * city1, ? * city2)$$

will correctly return

$?city1 = hamburg, ?city2 = paris$, and vice versa, even though $DC(*paris, *hamburg)$ is not explicitly present in $\mathcal{RCC}$. Thus, unlike the $\models$ relation for the SBox which only requires model checking, "spatial inference" is required for query answering on the RCC substrate.

4 The nRQL ABox Query Language

The nRQL ABox QL is now described in some detail here, since this is the QL which is used and extended to a spatial QL in this paper.

At a first glance, it will seem that nRQL is in a line of QLs which is closely related to Horn logic (query) languages such as Datalog: After all, nRQL provides some form of conjunctive queries. As such, one might criticize nRQL for not being "state of the art in logic programming techniques". However, considering nRQL as some-kind of non-recursive Datalog with Negation is inappropriate and *pragmatically misleading,* since the language has been designed under a different perspective. nRQL is an *instantiation* of a more general QL framework, the *substrate QL framework*. We claim that this framework provides more flexibility and options for extensions than, for example, Datalog, and demonstrate this in this paper. In order to support this claim, it is thus crucial that we formally define syntax and semantics of nRQL resp. of the generic substrate QL framework.

In the following we will use the *concrete syntax of nRQL*. A nRQL query consists of a *query head* and a *query body*:

```
(retrieve (?x ?y) (and (?x woman) (?x ?y has-child)))
```

has the head `(?x ?y)` and the body `(and (?x woman) (?x ?y has-child))`. The query returns all mother-child pairs from the ABox which is queried. In a nutshell, nRQL can be characterized as follows:

• **Variables and individuals** can be used in queries. The variables range over the individuals of an ABox (this is called *active domain semantics*), and are bound to those ABox individuals which *satisfy* the query. The notion of satisfiability of a query used in nRQL is defined in terms of logical entailment. A variable is bound to an ABox individual iff it can be proven that this binding holds in *all* models of the knowledge base. Returning to our example query body `(and (?x woman) (?x ?y has-child))`, `?x` is only bound to those individuals which are instances of the concept `woman` having a *known* child `?y` in *all* models of the KB.

nRQL distinguishes variables which must be bound to *differently named individuals* (prefix `?`, e.g., `?x, ?y` cannot be bound to the same ABox individual) from variables for which this does not hold (prefix `$?`, e.g., `$?x, $?y`). Individuals from an ABox are identified by using them directly in the query, e.g. `betty`.

• **Different types of query atoms** are available: these include concept query atoms, role query atoms, constraint query atoms, and same-as query atoms. To give some examples, the atom `(?x (and woman (some has-child female)))` is a concept

query atom, `(?x ?y has-child)` is a role query atom, `(?x ?x (constraint (has-father age) (has-mother age) =))` is a constraint query atom (asking for the persons `?x` whose parents have equal age), and `(same-as ?x betty)` is a same-as query atom, enforcing the binding of `?x` to `betty`.

As the given example concept query atom demonstrates, it is possible to use complex concept expressions within concept query atoms. Regarding role query atoms, the set of role expressions is more limited. However, it is possible to use inverted roles (e.g., role expressions such as `(inv R)`) as well as *negated roles* within role query atoms. Note that negated roles are not supported in the concept expression language of $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$; thus, they are only available in nRQL. For example, the atom `(?x ?y (not has-father))` will return those bindings for `?x, ?y` for which RacerPro *can prove* that the individual bound to `?x` cannot have the individual bound to `?y` as a father. If the role `has-father` was defined as having the concept `male` as a range, then at least all pairs of individuals in which `?y` is bound to a `female` person are returned, given `male` and `female` can be proven to be disjoint.

• **Complex queries** are built from query atoms using the boolean constructors `and, union, neg, project-to`. This holds for *all substrate QLs*. We have already seen an example: `(and (?x woman) (?x ?y has-child))` is a simple *conjunctive query body.* These constructors can be combined in an arbitrary (orthogonal) way to create complex queries. This is why we call nRQL an orthogonal language.

The `neg` operator implements a *negation as failure semantics (NAF)*: `(neg (?x woman))` returns all ABox individuals for which RacerPro *cannot prove* that they are instances of `woman`. Thus, `(neg (?x woman))` returns the complement set of `(?x woman)` w.r.t. the set of all ABox individuals. If used in front of a role query atom, e.g. `(neg (?x ?y has-child))`, then this returns the complement of `(?x ?y has-child)` (w.r.t. to all pairs of ABox individuals), i.e. all pairs of individuals which are not in the extension of `has-child` in all models. The semantics of nRQL ensures that DeMorgan's Laws hold: `(neg (and` $a_1 \ldots\ a_n$`))` is equivalent to `(union (neg` a_1`) ... (neg` a_n`))`.

Note that `(?x (not woman))` has a different semantics from `(neg (?x woman))`, since the former returns the individuals for which RacerPro *can prove* that they are *not instances* of woman, whereas the latter returns all instances for which RacerPro *cannot prove* that they are *instances* of woman.

• **Support for retrieving *told values* from the CD:** Suppose that `age` is a so-called *concrete domain attribute* of type integer. Thus, the `age` attribute fillers of a certain individual must be concrete domain values of type `integer`. We can use the following query to retrieve all adults as well as their ages: `(retrieve (?x (told-value (age ?x)) (?x (min age 18))))`, and a possible answer might be `(((?x michael) ((told-value (age michael)) 34)))`. Please refer to [WM05, KG06] for more details and the design rationale.

• **Support for CD constraint checking :** The so-called *constraint query atoms* allow one to "compare" concrete domain attribute fillers of different individuals. Consider the query

```
(retrieve (?x (told-value (age ?x)))
      (and (?x (and woman (an age))) (?x ?y has-child)
           (?y ?y (constraint (has-father age) (has-mother age)
                              (<= (+ age-2 8) age-1)))))
```

which returns the list of women and their ages. The women are required to have children whose fathers are at least 8 years older than their mothers. Note that `(has-father age)` denotes a "path expression": starting from the individual bound to `?y` we retrieve the value of the concrete domain attribute `age` of the individual which is the filler of the `has-father` role (feature) of this individual. In a similar way, the age of the mother of `?y` is retrieved. These concrete domain values are then used as actual arguments to evaluate the compound concrete domain predicate `(<= (+ age-2 8) age-1)`. Here, `age-2` refers to `(has-mother age)`, and `age-1` refers to `(has-father age)`. Note that the suffixes `-1, -2` have been added to the `age` attribute in order to differentiate the two values. Obviously, this mechanism is not needed if the two chains are ended by different attributes.

• **Special support for querying OWL documents**, e.g., retrieving told datatype value fillers of OWL datatype and OWL annotation properties. Retrieval of these datatype values is supported in a similar style as in the concrete domain case, by means of concept query atoms and head projection operators. We do not go into detail here, please refer to [KG06].

• **The body projection operator (`project-to`):** Sometimes this operator is required in order to reduce the "dimensionality" of a tuple set, for example, before computing its complement with `neg`.

Let us motivate the necessity for such an operator: Consider `(retrieve (?x) (and (?x mother) (?x ?y has-child)))`. This query returns all mothers having a *known child* in the ABox. Now, how can we query for mothers which do *not* have a *known* child? Our first attempt will be the query `(retrieve (?x) (and (?x mother) (neg (?x ?y has-child))))`. A bit of thought and recalling that `(neg (?x ?y has-child))` returns the complement set of `(?x ?y has-child)` w.r.t. the Cartesian product of all ABox individuals will reveal that this query doesn't solve the task. In a second attempt, we will probably try `(retrieve (?x) (neg (and (?x mother) (?x ?y has-child))))`. However, due to DeMorgan's Law and nRQL's semantics, this query is equivalent to `(retrieve (?x) (union (and (neg (?x mother)) (?y top)) (neg (?x ?y has-child))))` – first the union of two two-dimensional tuple sets is constructed, and then only the projection to the first element of these pairs (`?x`) is returned. Obviously, this set contains also the instances which are *not* known to be mothers, which is wrong as well. Thus, the need for the projection operator becomes apparent: `(retrieve (?x) (and (?x mother) (neg (project-to (?x) (?x ?y has-child)))))` solves the task. This body projection operator was not present in earlier versions of nRQL, special syntax was introduced to address these problems, namely the special unary atoms `(?x (has-known-successor has-child))`, `(?x NIL has-child)` and `(NIL ?X child-of)`. These atoms (which still work) can now be seen as "syntactic sugar" for the bodies `(project-to (?x) (?x ?y has-child))`, `(neg (project-to (?x) (?x ?y has-child)))` and `(neg (project-to (?x) (?y ?x has-child)))`. The `project-to` operator can be used at any position in a query body.

We can now define syntax and semantics of nRQL. Please note that all substrate QLs share in principle the same syntax and semantics, only *atom* (and possibly also *head_projection_operator*) are redefined appropriately, as already mentioned:

Definition 5 (Syntax of nRQL) A nRQL query has a *head* and a *body*.

A *head* can contain the following (note that {a|b} represents a or b):

head	:=	(*head_entry**)
object	:=	*variable* | *individual*
variable	:=	a symbol beginning with "?"
individual	:=	a symbol
head_entry	:=	*object* | *head_projection_operator*
head_projection_operator	:=	(*cd_attribute object*) |
		(told-value (*cd_attribute object*)) |
		(told-value (*datatype_property object*)) |
		(annotations (*annotation_property object*))

The *body* is defined as follows:

body	:=	*atom* | ({and | union} *body**) | (neg *body*) |
		(project-to (*object**) *body*)
atom	:=	(*object concept_expr*) | (*object object role_expr*) |
		(*object object* (constraint *chain chain constraint_expr*)) |
		(same-as *object object*)
chain	:=	(*role_expr** *cd_attribute*)

The "bridge" to the RacerPro syntax is given by the following rules:

concept_expr	:=	a RacerPro concept, with some extensions for OWL
role_expr	:=	a RacerPro role or the *special role* equal |
		(inv *role_expr*) | (not *role_expr*)
constraint_expr	:=	a (possibly compound) RacerPro concrete domain predicate, e.g. (< (+ age-1 20) age-2)
cd_attribute	:=	a RacerPro concrete domain attribute
datatype_property	:=	a RacerPro role used as OWL datatype property

■

Note that nRQL permits the use of negated roles in role atoms, as well as the special role equal. The equal role can only be used in nRQL, not in $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ concept expressions. The semantics of equal is fixed: $\texttt{equal}^{\mathcal{I}} =_{def} \mathcal{ID}_{2,\Delta^{\mathcal{I}}} = \{\, \langle i, i\rangle \mid i \in \Delta^{\mathcal{I}} \,\}$.

Definition 6 (Semantics of nRQL) Let $\mathcal{A}$ be a RacerPro ABox, and $\mathcal{T}_{\mathcal{A}}$ denote its associated TBox. Denote the set of individuals used in $\mathcal{A}$ with $\mathsf{Inds}_{\mathcal{A}}$.

Let q be a nRQL query *body*. The function $\mathsf{vars}(q)$ is defined inductively:
$\mathsf{vars}(\texttt{(}x \ \textit{concept\_expr}\texttt{)}) =_{def} \{x\}$, $\mathsf{vars}(\texttt{(}x_1 \ x_2 \ \textit{role\_expr}\texttt{)}) =_{def} \{x_1, x_2\}$,
$\mathsf{vars}(\texttt{(}x_1\ x_2\ \texttt{(constraint ...)))} =_{def} \{x_1, x_2\}$,
$\mathsf{vars}(\texttt{(\{ and | union | neg \}}\ q_1 \ldots q_m\texttt{)}) =_{def} \bigcup_{1\leq i\leq m} \mathsf{vars}(q_i)$, **BUT**
$\mathsf{vars}(\texttt{(project-to (}x_1 \ldots x_m\texttt{) ...)}) =_{def} \{x_1 \ldots x_m\}$. Thus, vars "stops at projections".

Assume that $\langle x_{1,q}, \ldots, x_{n,q}\rangle$ is a lexicographic enumeration of $\mathsf{vars}(q)$. Denote the ith element in this vector with $x_{i,q}$, indicating its position in the vector.

Let $\mathcal{T}$ be a set of n-ary tuples $\langle t_1, \ldots, t_n\rangle$ and $\langle i_1, \ldots, i_m\rangle$ be an *index vector* with $1 \leq i_j \leq n$ for all $1 \leq j \leq m$. Then we denote the set $\mathcal{T}'$ of m-ary tuples

with $\mathcal{T}' =_{def} \{ \langle t_{i_1}, \ldots, t_{i_m} \rangle \mid \langle t_1, \ldots, t_n \rangle \in \mathcal{T} \} = \pi_{\langle i_1,\ldots,i_m \rangle}(\mathcal{T})$, called the *projection* of $\mathcal{T}$ to the components mentioned in the index vector $\langle i_1, \ldots, i_m \rangle$. For example, $\pi_{\langle 1,3 \rangle}\{\langle 1,2,3 \rangle, \langle 2,3,4 \rangle\} = \{\langle 1,3 \rangle, \langle 2,4 \rangle\}$.

Let $\vec{b} = \langle b_1, \ldots, b_n \rangle$ be a *bit vector* of length n, $b_i \in \{0,1\}$. Let $m \leq n$. If $\vec{b}$ is a bit vector which contains exactly m ones, and $\mathcal{B}$ is a set, $\mathcal{T}$ is a set of m-ary tuples, then the n-dimensional *cylindrical extension* $\mathcal{T}'$ of $\mathcal{T}$ w.r.t. $\mathcal{B}$ and $\vec{b}$ is defined as

$$\begin{aligned}\mathcal{T}' =_{def} \{ \langle i_1, \ldots, i_n \rangle \mid\ & \langle j_1, \ldots, j_m \rangle \in \mathcal{T}, 1 \leq l \leq m, 1 \leq k \leq n \\ & \text{with } i_k = j_l \text{ if } b_k = 1, \\ & \text{and } b_k \text{ is the } l\text{th one (1) in } \vec{b}, \\ & \text{otherwise, } i_k \in \mathcal{B} \ \ \}\end{aligned}$$

and denoted by $\chi_{\mathcal{B},\langle b_1,\ldots,b_n \rangle}(\mathcal{T})$. For example,
$\chi_{\{a,b\},\langle 0,1,0,1 \rangle}(\{\langle x, y \rangle\}) = \{\langle a,x,a,y \rangle, \langle a,x,b,y \rangle, \langle b,x,a,y \rangle, \langle b,x,b,y \rangle\}$.

We denote an n-dimensional bit vector having ones at positions specified by the index set $\mathcal{I} \subseteq 1 \ldots n$ as $\vec{\mathbf{1}}_{n,\mathcal{I}}$. For example, $\vec{\mathbf{1}}_{4,\{1,3\}} = \langle 1,0,1,0 \rangle$. Moreover, with $\mathcal{ID}_{n,\mathcal{B}}$ we denote the n-dimensional identity relation over the set $\mathcal{B}$: $\mathcal{ID}_{n,\mathcal{B}} =_{def} \{ \underbrace{\langle x, \ldots, x \rangle}_{n} \mid x \in \mathcal{B} \}$.

The semantics of a nRQL query is given by the set of tuples it returns if posed to an ABox $\mathcal{A}$. This set of answer tuples is called the extension of q and denoted by $q^{\mathcal{E}}$.

In order to simplify the specification of the semantics, the query body q first undergoes some syntactical transformations. In a first step, q is rewritten by consistently replacing all its individuals with representative (fresh) variables throughout the body: If the individual i has been replaced with the variable x_i, then we also add the conjunct `(same-as` x_i i`)` to q, yielding a body of the form `(and` q `(same-as` x_i i`)` $\ldots$`)`. In a second step, the role chains (see syntax rule *chain*) possibly present in constraint query atoms are decomposed. This means they are replaced by conjunctions of role query atoms such that only concrete domain attributes remain in chains of constraint query atoms. Fresh variables are used for this purpose.

We can now define the semantics of the different query atoms, being part of q'. Keep in mind that $\langle x_{1,q'}, \ldots, x_{n,q'} \rangle$ is the variable vector of q' and that n is the total number of variables returned by $\mathsf{vars}(q')$. The semantics for the different nRQL query atoms is given as:

$$\begin{aligned}
(q'_{x_i}\ \mathit{concept_expr})^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{1}}_{n,\{i\}}}(\mathsf{concept_instances}(\mathcal{A}, \mathit{concept_expr})) \\
(q'_{x_i}\ q'_{x_j}\ \mathit{rolen_expr})^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{1}}_{n,\{i,j\}}}(\mathsf{role_pairs}(\mathcal{A}, \mathit{role_expr})),\ \text{if } i \neq j \\
(q'_{x_i}\ q'_{x_i}\ \mathit{role_expr})^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{1}}_{n,\{i\}}}(\mathsf{role_pairs}(\mathcal{A}, \mathit{role_expr}) \cap \mathcal{ID}_{2,\mathsf{Inds}_{\mathcal{A}}}) \\
\texttt{(same-as}\ q'_{x_i}\ \mathit{ind}\texttt{)}^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{1}}_{n,\{i\}}}(\{\mathit{ind}\}) \\
\texttt{(same-as}\ \mathit{ind}\ q'_{x_i}\texttt{)}^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{1}}_{n,\{i\}}}(\{\mathit{ind}\}) \\
\texttt{(same-as}\ q'_{x_i}\ q'_{x_j}\texttt{)}^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{1}}_{n,\{i,j\}}}(\mathcal{ID}_{2,\mathsf{Inds}_{\mathcal{A}}}),\ \text{if } i \neq j \\
\texttt{(same-as}\ q'_{x_i}\ q'_{x_i}\texttt{)}^{\mathcal{E}} &=_{def} \chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{1}}_{n,\{i\}}}(\mathcal{ID}_{2,\mathsf{Inds}_{\mathcal{A}}})
\end{aligned}$$

$$\begin{aligned}
(q'_{x_i}\ q'_{x_j}\ \texttt{(constraint}\ \mathit{attrib}_1\ \mathit{attrib}_2\ P\texttt{))}^{\mathcal{E}} &=_{def} \\
&\chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{1}}_{n,\{i,j\}}}(\mathsf{predicate_pairs}(\mathcal{A}, \mathit{attrib}_1, \mathit{attrib}_2, P)),\ \text{if } i \neq j \\
(q'_{x_i}\ q'_{x_i}\ \texttt{(constraint}\ \mathit{attrib}_1\ \mathit{attrib}_2\ P\texttt{))}^{\mathcal{E}} &=_{def} \\
&\chi_{\mathsf{Inds}_{\mathcal{A}},\vec{\mathbf{1}}_{n,\{i\}}}(\mathsf{predicate_pairs}(\mathcal{A}, \mathit{attrib}_1, \mathit{attrib}_2, P) \cap \mathcal{ID}_{2,\mathsf{Inds}_{\mathcal{A}}})
\end{aligned}$$

This definition uses some auxiliary functions, providing the bridge to the classical ABox retrieval functions offered by RacerPro. These functions have the standard DL semantics in terms of logical entailment:

$$\begin{array}{l}\mathsf{concept\_instances}(\mathcal{A}, concept\_expr) =_{def} \\ \quad \{\, i \mid\ i \in \mathsf{Inds}_{\mathcal{A}}, (\mathcal{A}, \mathcal{T}_{\mathcal{A}}) \models concept\_expr(i) \,\} \\ \mathsf{role\_pairs}(\mathcal{A}, role\_expr) =_{def} \\ \quad \{\, \langle i, j\rangle \mid\ i, j \in \mathsf{Inds}_{\mathcal{A}}, (\mathcal{A}, \mathcal{T}_{\mathcal{A}}) \models role\_expr(i, j) \,\} \\ \mathsf{predicate\_pairs}(\mathcal{A}, attrib_1, attrib_2, P) =_{def} \\ \quad \{\, \langle i, j\rangle \mid\ i, j \in \mathsf{Inds}_{\mathcal{A}}, (\mathcal{A}, \mathcal{T}_{\mathcal{A}}) \models \\ \qquad\qquad \exists x, y : attrib_1(i, x) \wedge attrib_2(j, y) \wedge P(x, y) \,\}\end{array}$$

Moreover, a nRQL *role expression (role_expr)* can also be a *negated* (or inverse) role. In the case of $role\_expr = \texttt{equal}$ we have $(\mathcal{A}, \mathcal{T}_{\mathcal{A}}) \models \texttt{equal}(i, j)$ iff $i^{\mathcal{I}} = j^{\mathcal{I}}$ in all models $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $(\mathcal{A}, \mathcal{T}_{\mathcal{A}})$, and in the case of $role\_expr = \neg\texttt{equal}$ we must have $i^{\mathcal{I}} \neq j^{\mathcal{I}}$ in all models. The predicates P are made from the vocabulary offered by RacerPro for building CD predicates (e.g., $<(+(age_2, 8), age_1)$ is a CD predicate with free variables age_1, age_2, and $+$ is a functor with the standard semantics).

The semantics of complex nRQL bodies can be defined easily now:

$$\begin{array}{rcl}\texttt{(and}\ q_1 \ldots q_i\texttt{)}^{\mathcal{E}} & =_{def} & \bigcap_{1\leq j\leq i} q_j^{\mathcal{E}} \\ \texttt{(union}\ q_1 \ldots q_i\texttt{)}^{\mathcal{E}} & =_{def} & \bigcup_{1\leq j\leq i} q_j^{\mathcal{E}} \\ \texttt{(neg}\ q\texttt{)}^{\mathcal{E}} & =_{def} & (\mathsf{Inds}_{\mathcal{A}})^n \setminus q^{\mathcal{E}} \\ \texttt{(project-to (}x_{i_1,q} \ldots x_{i_k,q}\texttt{)}\ q\texttt{)}^{\mathcal{E}} & =_{def} & \pi_{\langle i_1,\ldots,i_k\rangle}(q^{\mathcal{E}})\end{array}$$

So far we have specified the semantics of a query body. To get the answer of a query, the *head* has to be considered. This can be seen as a further projection of $q^{\mathcal{E}}$ to the variables mentioned in the head. If the head contained an individual, then this individual has also been replaced by the representative variable in the head. If a head projection operator is encountered, the functional operator is applied to the binding of the argument variable. The value is included in the query answer (producing nested binding lists); a more formal definition of this function application is omitted here. ■

5 Hybrid Spatio-Thematic Queries

According to the discussion in Section 3, we are either using a single ABox $\mathcal{A}$ (option 1), a (hybrid) map substrate $(SBox, \mathcal{A}, *)$ (option 2), or or a hybrid substrate $(\mathcal{A}, \mathcal{RCC}, *)$ (option 3) for DISK representation in the DLMAPS system. nRQL is used in all settings. We already explained how we make a substrate QL such as nRQL hybrid: A hybrid query uses now two kinds of query atoms, those for S_1, and those for S_2. Variables / individuals in atoms for S_2 are simply prefixed with $*$. Then, in a query body, any combination of atoms for S_1 and S_2 is permitted, and variables are bound in a $*$-consistent way, see Page 11.

Basically, all substrate QLs share the same *body* syntax (Def. 5). The *atom*s are substrate specific, as explained. Also the head projection operators can be specific. The

semantics of complex query bodies in Def. 6 is shared by all substrate QLs. However, each specialized atom must define its own extension by means of the dedicated $\models$ relation; e.g., for an arbitrary *atom*, its extension $atom^{\mathcal{E}}$ must be defined (w.r.t. substrate S). That's all. For a hybrid query language, we additionally require that the computed binding tuples respect the $*$-function, that is, computed bindings must $*$-consistent. We do not go into formal details here.

Which spatio-thematic QL is now applicable for the different representation options in the DLMAPS system?

• **For option 1:** We can only use plain nRQL, as explained.

• **For option 2:** The resulting hybrid QL is called SnRQL. It provides the following spatial atoms in addition to nRQL:

• **RCC atoms:** Atoms such as `(?x ?y (:tppi :ntppi))` etc. *Spatial prepositions* such as `:contains, :adjacent, :crosses, :overlaps, :flows-in` are available.

• **Distance Atoms:** `(?x ?y (:inside-distance <min> <max>))`, where `<min>, <max` specifies an interval $[min; max]$; `NIL` can be used for 0 resp. ∞ (this applies to the subsequent interval specifications as well). For example, the extension of `(i ?x (:instance-distance nil 100))` consists of all SBox objects which are *not further away than 100 meters from* i. Either the shortest distance or the distance of the centroids of these objects can be used for distance computation.

• **Epsilon Atoms:** `(?x ?y (:inside-epsilon <min> <max>))`. With that atom, all objects `?y` are retrieved, such that `?y` is contained within the *buffer zone* of size $[min; max]$ around `?x`. This buffer zone consists of all points (x, y) whose *shortest distance to the fringe of* `?x` is contained within $[min; max]$.

• **Geometric Attribute Atoms:** Atoms regarding geometric attributes, e.g. length and area: The extension of `(?x (:area 100 1000))` consists of all polygons whose area is in $[100; 1000]$. Also `:length` is understood.

Moreover, simple type checking atoms such as `(?x :is-polygon)`, `(?x :is-line)` etc. are available.

To give a final example, consider this SnRQL query in concrete syntax which selects an appropriate home for a millionaire:

```
(retrieve (?villa ?living-area  ?golf-club ?church)
  (and (?*living-area (and living-area
                           (or (all classification first-class-area)
                               (string= name "Beverley Hills"))))
       (?living-area ?villa :contains)
       (?*villa (and villa (all status for-sale) (> has-price 10000000)
                    (some has-comfort swimming-pool)))
       (?church ?living-area (:inside-epsilon nil 200))
       (?living-area ?golf-club :adjacent)
       (?*golf-club (and golf-club (all members millionaire)))))
```

The extensions of the atoms are computed on the fly from the geometry with *spatial inspections methods* (see Page 5). As argued, this can be understood as a kind of (spatial) model checking.

• **For option 3,** the resulting hybrid QL is called *nRQL + RCC atoms*. This language can only offer RCC atoms, since the geometry of the map is not represented. The syntax of the SnRQL RCC atoms is identical to the RCC atoms just discussed.

6 Conclusion

Building ontology-based IS with enabling DL technology is a non-trivial task, especially for IS in non-standard domains such as the one considered here. The space of design decisions is very large. Since decidability is not always easy to achieve it is thus of even more importance to identify pragmatic and practical solutions which, even though if they do not exploit or advance the latest theoretical state-of-the-art techniques in DL research, can nevertheless be considered an advance w.r.t. the current state-of-the-art of IS technology and provide guidance and "road maps" for similar designs. We claim that our framework for building pragmatic combinations of specialized representation layers (including DL ABoxes) for which orthogonal specialized substrate QLs can be defined provides a great deal of flexibility for building similar ontology-based IS. Moreover, the implemented functionality is available for other users and system designers. Thus, we believe that these pragmatic solutions can be called empirically successful. DLMAPS can be found under `http://www.sts.tu-harburg.de/~mi.wessel/dlmaps/dlmaps.html`.

References

[HLM99] V. Haarslev, C. Lutz, and R. Möller. A description logic with concrete domains and a role-forming predicate operator. *Journal of Logic and Computation*, 9(3):351–384, 1999.

[HM01a] V. Haarslev and R. Möller. RACER System Description. In *Int. Joint Conference on Automated Reasoning*, 2001.

[HM01b] V. Haarslev and R. Möller. The Description Logic ALCNHR+ Extended with Concrete Domains: A Practically Motivated Approach. In *International Joint Conference on Automated Reasoning*, 2001.

[HST99] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In *Proc. International Conference on Logic for Programming and Automated Reasoning*, 1999.

[KG06] Racer Systems GmbH & Co. KG. RacerPro User's Guide 1.9.0. Technical report, Racer Systems GmbH & Co. KG, `http://www.racer-systems.com/products/racerpro/users-guide-1-9.pdf`, 2006.

[KLWZ04] O. Kutz, C. Lutz, F. Wolter, and M. Zakharyaschev. E-Connections of Abstract Description Systems. *Artificial Intelligence*, 156(1):1–73, 2004.

[LM05] C. Lutz and M. Milicic. A Tableau Algorithm for DLs with Concrete Domains and GCIs. In *Proc. of the International Workshop on Description Logics*, 2005.

[LW04] C. Lutz and F. Wolter. Modal logics of topological relations. In *Proc. of Advances in Modal Logics*, 2004.

[Wes03a] M. Wessel. Some Practical Issues in Building a Hybrid Deductive Geographic Information System with a DL Component. In *Proc. of the 10th Int. Workshop on Knowledge Representation meets Databases*, 2003.

[Wes03b] M. Wessel. Qualitative Spatial Reasoning with the $\mathcal{ALCI}_{\mathcal{RCC}}$-family – First Results and Unanswered Questions. Technical report, May 2003. Available at `http://www.sts.tu-harburg.de/~mi.wessel/papers/report7.pdf`.

[WM05] M. Wessel and R. Möller. A High Performance Semantic Web Query Answering Engine. In *Proc. International Workshop on Description Logics*, 2005.

Real World Verification
Experiences from the Verisoft Email Client

Gerd Beuster, Niklas Henrich, Markus Wagner*
University Koblenz-Landau
{gb}|{nikhen}|{wagnermar}@uni-koblenz.de

Abstract

This paper reports our experiences developing a completely verified email client. The formal specification of the email client includes all informal requirements and security goals. Compliance to the formal specification has been proven for the complete source code. The email client is part of project Verisoft, where pervasively verified systems are developed.

1 Introduction

The goal of the Verisoft project is to create the tools and methods to allow the pervasive formal verification of computer systems, and to show that verification of real world systems is viable [Pau05]. In Verisoft, formal methods and verification technology are used throughout all aspects of system developing, including verified hardware, verified development tools, and verified operating systems and verified application programs. Four concrete systems are developed in Verisoft. Of these four systems, three are developed by or in cooperation with partners from the industry, and one is developed by the academic partners. The industry projects include an *Emergency Call System* developed in cooperation with the BMW group, a *Biometric Identification System* developed in cooperation with T-Systems, and *Hardware verification* developed in cooperation with Infineon Technologies. The academic project develops a secure email system. This paper reports our experiences developing a completely verified email client as part of the academic system.

1.1 The Academic System

The goal of the academic project is to show that common desktop technology can be formally specified and verified. For this reason, the technology used in the academic system tries to stay as close to "normal" systems, technologies, and standards as possible. The academic system is made up of different parts, as depicted in figure 1. The

*This work was funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors. See http://www.verisoft.de for more information about Verisoft.

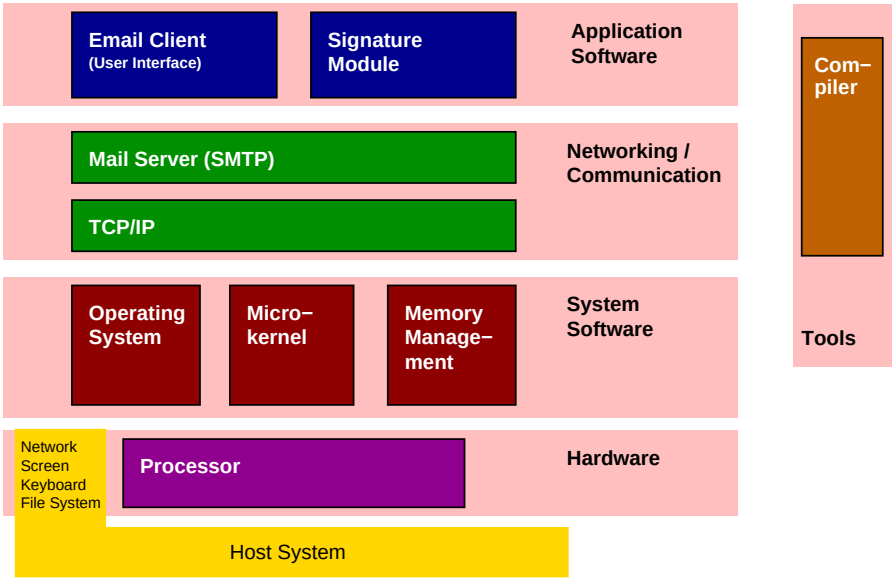

Figure 1: Components of the academic system

verified compiler compiles programs written in the C dialect C0 [LPP05]. The machine code is run on fully verified hardware (processor) [ABKS05]. Three layers of software build upon the hardware. The first layer consists of a fully verified micro-kernel, memory management unit and an accompanying operating system called *Simple Operating System* [GHLP05]. The networking and communication layer consists of a fully verified SMTP mail server using a fully verified TCP/IP stack. This allows the academic Verisoft system to interconnect with the "real world" like Intranets or the Internet. The application software sits on top of the system software and communication layer.

As part of the Academic Verisoft System, we developed a completely verified email client. The formal specification of the email client includes all informal requirements and security goals. Compliance to the formal specification has been proven for the complete source code. Each of the three main Sections 2–4 deals with one of the core results of our work on the Verisoft email client. Section 2 explains how we formally specified secure user interfaces. Section 3 describes the lessons learned from the early stages of specification, and why developing a prototype was important. Section 4 reports our experiences from verifying the Verisoft email client.

1.2 Related Projects

Another important fundamental research project in the area of verification and analysis is the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (AVACS), which is funded by the Deutsche Forschungsgemeinschaft (DFG). About 70 scientists of the Universities of Oldenburg, Freiburg and Saarbruecken as well as the Max-Planck-Institute of Computer Sciences of Saarbruecken are working on the improvement of techniques for mathematically precise verification, including the development of tools. The goal of their work is to automate safety analyses of critical embedded systems which are used for example in aircrafts, motor vehicles or

railway transportation [DHO04].

Significant prior projects are DAEDALUS and VERIFIX. The DEADALUS consortium was a research and technology development project in the Fifth Framework Programme (FP5) of the European Union. With the contributions of universities from France, Germany, Denmark, and Israel, the project developed methods and tools to support the industrial validation of critical concurrent software by static analysis and abstract testing [Gou01, CC02]. The goal of VERIFIX, another project funded by DFG, was the construction of mathematically correct compilers, which included the development of formal methods for specification and implementation of a compiler. One of the project's results was a fully verified LISP interpreter [GZ99].

2 User Interface Specification

Within the academic part of the Verisoft project, the Verisoft email client, for short *Vericlient*, provides the interface to the user. When a user accesses the academic system, he interacts with the email client. The email client itself has internal interfaces to four components: The I/O facilities (via the operating system), the SMTP server for delivery and reception of emails, and the signature module for generation and checking of signatures. For its internal operation, the email client makes use of data structures provided by the C library.

Providing a user interface is the core functionality of the email client. Vericlient provides a text-based user interface as shown in Figure 2. Since a general design goal of the Verisoft email system was to provide a secure environment for using email services, special emphasis was put onto the security of the user interface.

2.1 User Interface Security Requirements

The definition of a secure user interface is based on the common definition of secure computing as

Confidentiality	Information is available to authorized parties only.
Integrity	Neither the system nor services provided by and data processed by the system can be manipulated.
Availability	Accessibility of services and data is guaranteed.

We adapted these concepts to user interface security by restricting these definitions to the aspects involving the user interface and human-computer interaction. For Confidentiality, this means that eavesdropping on the input/output facilities must not be possible. Integrity of the user interface is guaranteed if manipulation of the user interface is not possible, i.e. if the user's assumptions about the state of the application, gained by observing and manipulating the application via the user interface, corresponds to the actual state of the application. Availability of the user interface means that an attacker can not get the user interface into a state where the full functionality is no longer accessible.

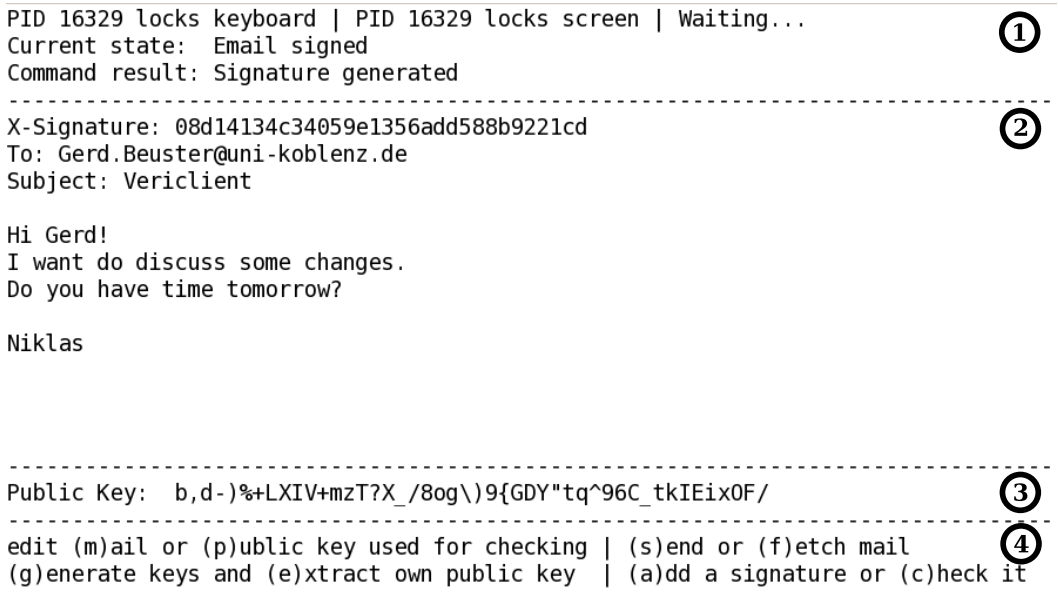

Figure 2: Vericlient prototype running: The numbers indicate the following screen areas: (1) Status / current state of the email client (2) Editing area (3) Public key (4) Commands available

Confidentiality	A third party can not gain information from observing human-computer interaction.
Integrity	Whenever the user issues a command, all relevant information, most notably the state of the program and the data processed, is shown on the screen correctly.
Availability	The functionality provided by the user interface is always accessible.

Translating these security constraints into a formal specification and writing an email client application satisfying the constraints posed a number of challenges. It turned out that a number of constraints raised by the email client development group required functionality from outside the email client. For example, neither Confidentiality, nor Integrity, nor Availability can be guaranteed if an attacker can manipulate the I/O devices. Therefore a key requirement for a system using keyboard input and screen output is the impossibility of man-in-the-middle attacks against the keyboard and the screen. If an attacker can get in between the legitimate application and its input/output facilities, the attacker can manipulate the user at will.

There is no easy way to prevent physical man-in-the-middle attacks like, for example, covering the real keyboard with a faked keyboard as described in [BR02]. However, the prevention of software-based attacks with Trojan horses, worms, viruses etc. is possible if the operating system provides means to guarantee exclusive access to the keyboard and screen. We call the process of acquiring exclusive access "locking" and the release of the lock "unlocking." Locking a resource is not sufficient to guarantee security. The user must also *know* which process locks a resource and whether the system is busy

or not. Providing (and verifying) this functionality is beyond the realm of a client application like Vericlient; it has to be provided by the operating system. Therefore, in the specification phase of the project a lot of communication with other development group was necessary. Most changes where requirement from the operating system group.

2.2 User Interface Specification

In order to formally specify and verify the security of a user interface, it was necessary to bring together formal methods, human computer interaction, and computer security. All three of them are established fields of research. There are also works combining each two of the fields. Formal methods have been used to specify human computer interaction. User interfaces have been designed and evaluated under security aspects. System security has been treated with formal methods. For the Verisoft email client, we had to combine all *three* fields.

Confidentiality relies mainly on the operating system, which has to ensure that eavesdropping on the application and the communication channel is not possible. Availability depends on all components involved. For the email client, it has to be shown that it is always possible to write, sign and send email, and that is always possible to receive email, check the signature, and read it. Integrity, defined as the requirement that data is displayed correctly whenever the user issues a command, is primarily the responsibility of the email client.

Usually, the specification of user input and system output is rather informal. Specifications declare that something "is shown on the screen" and the user "enters a text." In most cases, this informal description is sufficient. However, if we want to formally verify the integrity of a system, a formal definition is required.

In order to ensure integrity with formal methods, it is necessary that a) the output device provides the "right" information, b) the information is up-to-date whenever the user issues a command, and c) that the user is able to understand the information shown by the output device. For the latter, a formal user model is required. While there are some (semi-)formal methods for the description of user interfaces and human-computer interaction, these are usually not suited for automatic reasoning. Our approach was to formalize and extend the GOMS user model technique in a way that makes it suitable for modeling human-computer interaction, including potential human errors, and for automated reasoning [BB]. GOMS is a modeling technique (more specifically, a family of modeling techniques) for analyzing the complexity of interactive systems. The user's behavior is modeled in terms of *Goals*, *Operators*, *Methods* and *Selection rules*. Briefly, a GOMS model consists of methods that are used to achieve goals. A method is a sequential list of operators that the user performs and (sub)goals that must be achieved. If there is more than one method which may be employed to achieve a goal, a selection rule is invoked to determine what method to choose, depending on the context.

For the Vericlient, we assume that the user knows about the system state if he gets information about the last operation of the system ("email has been sent"), and the data on which the operation was performed (i.e. the actual email). Guaranteeing that screen output is up-to-date whenever the user issues a command is tricky, because user interfaces are inherently asynchronous. There will always be moments during the execution of the application, when the screen output does not reflect the actual state of

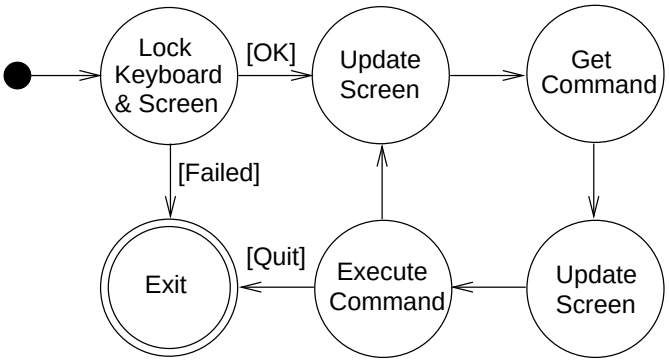

Figure 3: Statechart describing the main event loop.

the system, because only parts of the output screen have been updated, or because the system just finished an internal operation and the screen output had not been updated. The same is true for user input. Since keyboard input is usually buffered, it is possible that the user triggers actions without having seen the current screen output.

We solved the problem of asynchronous input/output by defining strict points in the main execution loop when the screen is updated, and by imposing restrictions on the input buffer. A statechart of the main execution loop is given in Figure 3.

After locking keyboard and screen (required to guarantee Confidentiality and Authenticity), the event loop receives keystrokes and executes the commands associated with the keystrokes. It also takes care of keeping the screen up-to-date. Since the screen may be inconsistent during state updates (i.e., the current screen display may not reflect the internal state of the system), the screen update function is called twice: Once before the system waits for the next keystroke, and again before command execution. In the second update, the screen area for displaying the current state shows the message "processing." When processing is finished, the loop starts over again, unless the user has issued the command "quit." Before the next command is accepted, the input buffer is emptied. This way, the user can be ensured that the screen display is consistent with the actual state of the application whenever the message "processing" is *not* shown.

The functional behavior of the email client, and thus the information to be displayed, is defined by the statechart shown in Figure 4. For example, the system transits from state Unsigned to state Signed if command "sign" was issued and the signing operation was successful. Of course, the states in such a statechart are abstractions of the application's actual internal configuration, which is much richer in detail. Nevertheless, we assume that these states are the *right* abstraction in that the user has sufficient information about the internal configuration of the application if he or she knows in what abstract state the application is.

2.3 Example: Editing Mail

Not only the state of the system, but also the data has to be displayed correctly. Defining "correct" display of an email under security aspects is a challenging task. In the real world, "phishing" attacks are major form of electronic fraud [Bac05]. Many of

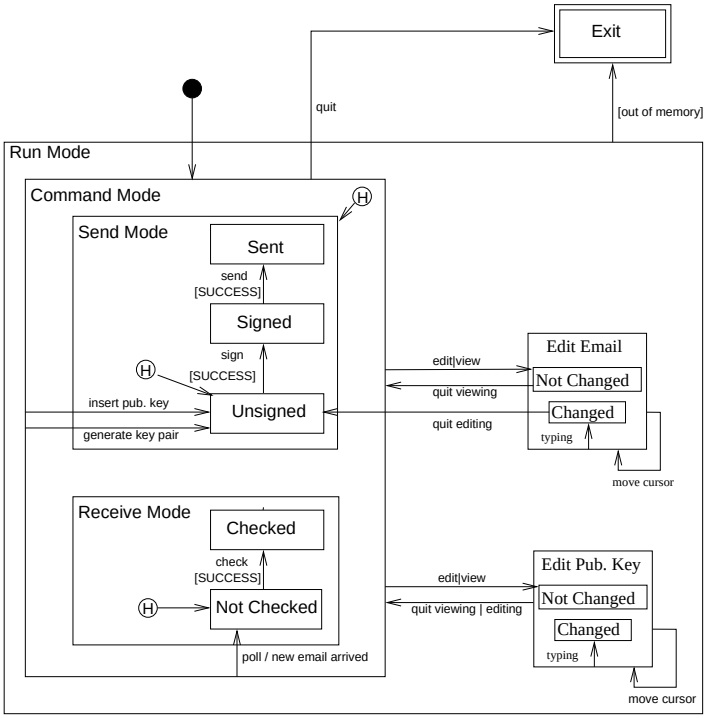

Figure 4: Statechart of email client applications. State transitions represent execution of program functions.

these attacks are based on exploitation of incorrect or ambiguous display of email messages. General concepts against these attacks are beyond the scope of this paper. For the Verisoft email client, these attacks are prevented by restricting the way emails are displayed. The Verisoft module for viewing and editing shows the pure ASCII representation of the email.

In the following, we present a short excerpt of the specification of the Verisoft email editing module. This example allows us to demonstrate how an interactive user interface component can be specified. The email viewing and editing component has the following characteristics: It is a full screen editor; the user can roam freely over the text using the cursor keys. The text edited may not fit the screen. In that case, the editor will scroll when the cursor reaches the screen borders.

The email message editing field is represented by a data structure $textEdit := (s, cx, cy, co, ro)$ with s a list of strings where each element represents a line of the text, (cx, cy) the cursor position and (ro, co) row and column offsets. If the text is larger than the size of the screen, the offsets indicate which part of the email are shown.

The part of the main execution loop's `updateScreen` responsible for showing the email (with (x, y) a position on the screen) is specified as:

$$updateScreen[y, x] = \begin{cases} s[y + ro][x + co] & \text{if } length(s) < y + ro \text{ and} \\ & \quad length(s[y + ro]) < x + co \\ \text{blank} & \text{otherwise} \end{cases}$$

The specification of main execution loop function `execute` for email editing is defined by the OCL specification given in Table 1. Note that `INSERT_CHAR` represents the set of all printable characters.

```
context execute(cmd, textEdit)
pre     cmd ∈ { CURSOR_LEFT, CURSOR_RIGHT, CURSOR_UP,
                CURSOR_DOWN, INSERT_CHAR, DELETE_CHAR,
                QUIT }
post    if cmd = CURSOR_LEFT then
          textEdit = cursorLeft(textEdit@pre) and
          result = CURSOR_MOVED
        else if cmd = CURSOR_RIGHT then
          textEdit = cursorRight(textEdit@pre) and
          result = CURSOR_MOVED
        else if cmd = CURSOR_UP then
          textEdit = cursorUp(textEdit@pre) and
          result = CURSOR_MOVED
        else if cmd = CURSOR_DOWN then
          textEdit = cursorDown(textEdit@pre) and
          result = CURSOR_MOVED
        else if cmd ∈ INSERT_CHAR then
          textEdit =
              insertChar(cmd, textEdit@pre) and
          result = CHAR_INSERTED
        else if cmd = DELETE_CHAR then
          textEdit = deleteChar(cmd, textEdit@pre) and
          result = CHAR_DELETED
        else
          result = QUIT
        end if
```

Table 1: Command execution function

The auxiliary functions describing the effects of cursor movements and inserting/deleting characters are straightforward. As an example, we only provide a definition for $cursorRight$:

$$cursorRight(a) = (s, cx', cy', co', ro')$$

with

$$cx' = \begin{cases} cx+1 & \text{if } a.cx+1 < length(a.s[a.cy]) \\ 0 & \text{if } a.cx+1 \geq length(a.s[a.cy]) \text{ and} \\ & \quad a.cy+1 < length(a.s) \\ cx & \text{otherwise} \end{cases}$$

$$cy' = \begin{cases} cy+1 & \text{if } a.cx+1 \geq length(a.s[a.cy]) \text{ and} \\ & \quad a.cy+1 < length(a.s) \\ cy & \text{otherwise} \end{cases}$$

$$co' = \begin{cases} co+1 & \text{if } cx' = co + \texttt{screenWidth} \\ 0 & \text{if } cx' = 0 \\ co & \text{otherwise} \end{cases}$$

$$ro' = \begin{cases} ro+1 & \text{if } cy' = ro + \texttt{screenHeight} \\ ro & \text{otherwise} \end{cases}$$

The correctness of the specification was ensured in two ways: It has been shown that the editing component specification allows to enter an arbitrary text, and it has been shown that the order of characters is preserved when an arbitrary text is shown by the email client. While these two refinement proofs ensure basic correctness of the editing component, they do not capture the interactive behavior of the editing component. A prototypical implementation (see also Section 3) was used to ensure that the specification of the editing component follows the user's intuition about an interactive editor.

We have shown how user interface security is formalized and specified for the Verisoft email client. In the next Sections, we report our experiences from implementing and verifying the Verisoft email client specification.

3 Specification and Prototypical Implementation

The most direct way to develop a fully formally specified and verified application would be to start by writing a formal specification. From this, one would either write an implementation and proof its correctness, or refine the specification down to the implementation level, generating the correctness proofs on the way.

Because of the character of the Verisoft project, this approach could not be followed strictly. Since Verisoft started largely from scratch, all parts of Verisoft, including the specification languages, the calculus, and the system components the email client relies on, were developed in parallel. Over the course of the projects, more and more tools were finished, the calculus and languages got fixed, and specifications and implementations of other components became available. This led to a somewhat different design model.

At the beginning of the project, we started by informally defining the global design goals of the Verisoft email client. We developed a semi-formal specification using OCL and statecharts. Based on this specification, a prototype was developed. The prototype served two purposes. First, it allowed us to test the informal specification. Verifying software is even more costly in terms of time and money than normal software development. Therefore we wanted to ensure that the specification of the email clients did not contain design errors. The main functionality of the email client is to provide the user interface for other system parts, like the SMTP component and the signature component. The design of the user interface of the email client must not only comply to security requirements ("the email is shown correctly"), it must also comply to the user's expectation about "proper behavior" of a user interface. The prototype allowed us to test our email client design before finalizing the formal specification.

For the development of the prototype, we used ordinary C, not C0, and we used the *ncurses* library[Str91] instead of Verisoft's operating system functionality for screen output. The client compiles in a standard Linux environment. This has the advantage, that we could provide a working prototypical version of the email client without being

dependent on other system parts. C0 is a subset of C. By restricting our coding style to the constructs allowed by C0, we learned how to deal with the limitations inherent to the language. The prototype was more an evolutionary prototype than a throwaway prototype. After more and more libraries and the final definition of C0 became available, we adjusted the prototype step-by-step until it become the final implementation of Vericlient.

The development of a prototype was crucial. We gained several insights on the run-time behavior of the Vericlient and it helped to improve and even to correct the specification. With the help of the prototype, we detected glitches in the user-interface (for example, characters were inserted at the wrong place in email messages because of an off-by-one error) as well as errors in the statechart specification of Vericlient's overall behavior. Since these errors were in the specification, they would not have been noticed until Vericlient would have been fished.

Errors found at the earlier stages are easier to correct than errors found in a late stage. This is even more the case for a verification projects, where errors in the specification may require the component to be proved again. A prototype helps to identify errors or wrong decisions in the early design and specification phase of a project. Since it was possible to evolutionary develop the final version from the prototype, the work spent on the prototype was efficiently integrated into the project.

Over the course of the project, more functionality from other parts of the academic Verisoft project became available. The Vericlient prototype gradually turned into the final implementation of the Verisoft email client by integrating these modules once they became available. The prototypical code developed to run in a normal Linux environment was maintained in parallel to the code developed for the Verisoft environment. The advantage of this approach was that from the very beginning of the project, a working version existed and changes in the specification could be tested. From our experience, the little extra work required to develop two versions in parallel pays out enormously. Since a working version of the email client existed from the very beginning, we always knew precisely if the interfaces and specifications provided by other modules fit into the email client, or if changes were required.

4 Verification and Integration

The goal of Verisoft is the pervasive verification of both system hardware and software. In order to allow integration of the components developed in different parts of the project, a common set of formal methods and tools is required. As a general design decision, the Verisoft participants agreed to use Isabelle/HOL [NPW02] as the main verification tool. Norbert Schirmer developed an Isabelle theory for the verification of C0 programs in Isabelle/HOL. The core of this theory is a Verification Condition Generator for translation of specifications and code into HOL [Sch05]. C0 is a subset of C with some limitations for easier verification. Side effects are not allowed in expressions, and there can only be one return statement in each function. Pointers are typed. Pointer arithmetic is forbidden, and arrays can not be allocated dynamically. The verified compiler developed by Leinenbach et al.[LPP05] translates C0 programs into machine code and into a format suitable for input into the Isabelle system.

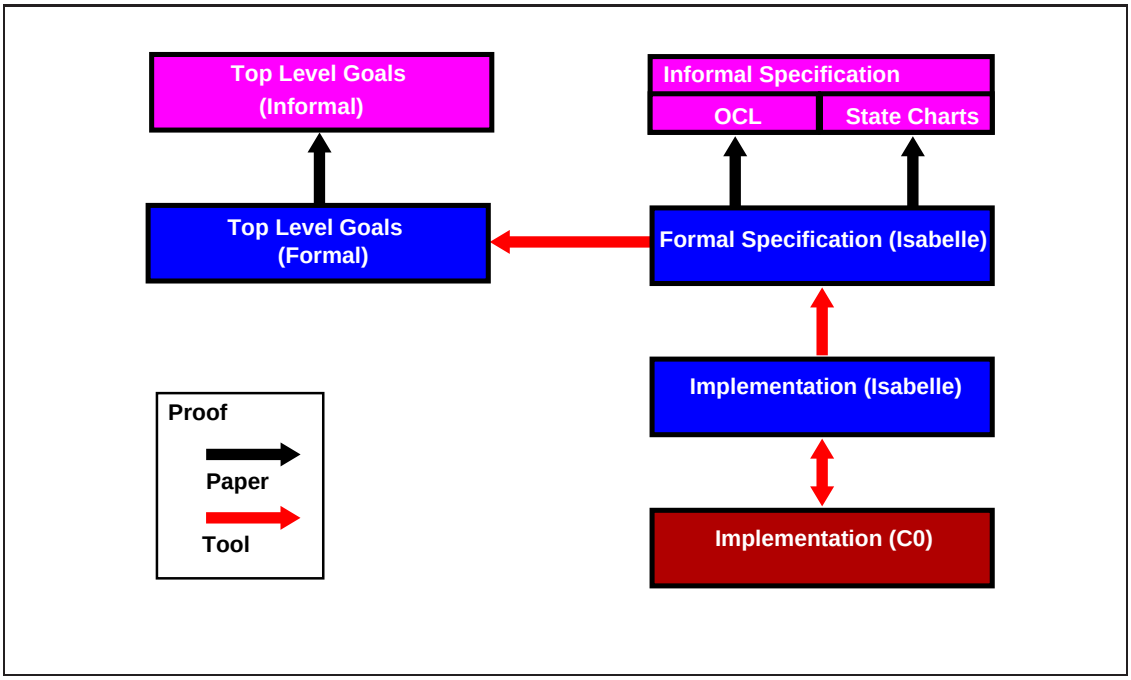

Figure 5: Types of models used in the specification of the Verisoft email client

The verification process of the Verisoft email client is depicted in Figure 5. Based on the informal specifications, formal Isabelle specifications were developed. The prototypical C implementation of the email client was adapted to C0, and correctness of the automatically generated Isabelle translation of the code was proven. Independent of this, formal definitions of the email client security requirements given in Section 2.1 were developed. From the functional correctness of the email client specification, compliance to the top level verification goals was deduced.

4.1 Verifying the Email Client

An important lesson from verifying the Verisoft email client is that *implementation follows verification*. When there was a problem in verifying a piece of code, we did not hesitate to change the implementation in order to make it easier to verify. Since the tool for automatic translation from C0 to the Isabelle representation of C0 was developed in Verisoft, and was therefore not available at the begin of the project, we started by manually translating parts of the code and verifying it. We learned that this goes very well with verification of the large and complex functions of the Verisoft email client. In order to verify a large function, we started with small pieces of the code and the specification. Once these fractions where verified successfully, more parts of the specification and of the code were added. The final implementation of the Verisoft email client consists of about 2800 lines of code. Verifying the Verisoft email client required an Isabelle proof script of twice the size of the code. Compared to typical text book examples, the proofs were rather simple. The Verisoft email client does not use data structures more complex than lists of strings, and operations on the data structures do

not involve recursion etc.

4.2 Interfaces to other components

The Verisoft email client has interfaces to a number of other components of the Verisoft system: The operating system provides system calls to the I/O devices. Remote procedure calls are used to pass mail to and from the SMTP server, and to the cryptographic module for signing and checking signatures. The C0 library provides essential data structures used by the email client. Since all of these components were developed in parallel to the Verisoft email client, their specifications and implementations became available over the course of the project.

In Section 3, we explained how we solved the problem of not having implementations of other components available at the begin of the project. For the specification and verification part, our approach to deal with this problem depended on the type of missing component.

Some of the components are essentially "black boxes" for the email client. For verifying the functional properties of the email client, the actual specification of the signature component and the SMTP server are irrelevant. For the email client, we just have to show that the SMTP and cryptographic functionality is executed at certain points during the execution of the email client. Specification of these components is needed only in the last step, when the email system is integrated and the top level goals of the email system are proven. Therefore, we were able to specify and verify the email client independent of these modules.

The situation was different for the data structures. For the specification and correctness proofs about the email client, we had to make use of the specification of the string and list data structures. Here, the Isabelle/HOL verification environment was beneficial. HOL provides native string and data structures. In the first phases of the project, we replaced the C0 data structures by their corresponding HOL data structures. Since the C0 data structures are defined as refinements of the native HOL data structures, integration of the real data structures was fairly straightforward. Only some additional pre-conditions had to be changed in order to take into account the cases where the behavior of the C0 data structure differs from the behavior of the HOL data structures. For example while HOL data structures do not have upper bounds, the length of C0 data structures is limited.

It turned out that this two-step approach did not cause a significant overhead, because the old proofs, conducted with the HOL data structures, did not become obsolete. They just had to be extended to deal with the additional constraints of the C0 data structures.

In conclusion, using non-verified, interface compatible libraries built-in into the tools and replacing them later on with their verified counterpart turned out to be a good approach. It allowed us to start verifying at a time when the final libraries were not available.

4.3 Integration

Different parts of the Verisoft academic system, and even different part of the Verisoft email client, use different kinds of formal methods. Parts are specified in terms of pre-

and post-conditions. These were verified in Hoare calculus. Other parts rely on temporal properties. These were specified in temporal logics and proven by model checking. In order to integrate both aspects, the specification and implementation of the email client was split in two parts: The temporal properties were modeled as the state transition diagram shown in Figure 4. The functional properties were embedded in this state transition diagram. Each state transition represented a function call specified in Hoare logics. Since all states are represented explicitly in Vericlient, and the main execution loop executes the statechart, integration was achieved by showing that each functional call executes the state transitions defined in the statechart.

5 Conclusions

Completely verifying the Verisoft email client posed a number of challenges:

- Other parts of the system, including the specification and implementation language and the calculus, were not available when the project started.
- Interactive user interfaces and secure human-computer interaction had to be specified and verified.
- Theorems proven in different formal methods had to be integrated.

All of these problems were solved. Implementing a prototype and using prototypical specifications of the other components proved to be a viable solution for the problem that not all system parts were available in the beginning, a situation quite typical for large-scale academic and industrial projects. Starting with a prototype, we could test the viability of our specification and start verifying without having all other system parts available. When more and more parts of the system became available, most of the work spent on the prototype could be reused.

Formally specifying secure human-computer interaction required genuine scientific work. By formalizing the user model and by adapting and formalizing secure computing for human-computer interaction, it was possible to verify user interface security with the formal methods employed in the Verisoft project. Here, it was beneficial that not all other parts of the Verisoft academic system were already available at the start of the project, because the requirements for a secure user interface directly affected the specification of other parts, like the I/O device interface of the operating system.

For the integration of results achieved by Hoare calculus verification with results achieved by model checking, it was a big advantage to have both methodologies integrated into Isabelle/HOL. This way, the same statechart specification could be used for model checking, and in the definition of pre- and post-conditions of the individual functions.

References

[ABKS05] Nathaniel Ayewah, Sven Beyer, Nikhil Kikkeri, and Peter-Michael Seidel. Challenges in the formal verification of complete state-of-the-art processors. In *International Conference on Computer Design*, San Jose, 2005.

[Bac05] Daniel Bachfeld. Nepper, Schlepper, Bauernfänger — Risiken beim Online-Banking. *c't magazin für Computertechnik*, pages 148–153, 2005.

[BB] Bernhard Beckert and Gerd Beuster. A method for formalizing secure user interfaces. In *Submitted to Eighth International Conference on Formal Engineering Methods (ICFEM 2006)*.

[BR02] L. Bussard and Y. Roudier. Authentication in ubiquitous computing. In *UBICOMP 2002, Workshop on Security in Ubiquitous Computing*, Göteborg, Sweden, September 2002.

[CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis. In N. Horspool, editor, *Proceedings of the International Conference on Compiler Construction (CC 2002)*, LNCS 2304, pages 159–178, Grenoble, France, April 6–14 2002.

[DHO04] W. Damm, H. Hungar, and E.-R. Olderog. On the verification of cooperating traffic agents. In F.S. de Boer, M.M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. FMCO '03: Formal Methods for Components and Objects*, LNCS 3188, pages 78–110, 2004.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In *Proceedings, 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS 3603, pages 2–16. Springer, 2005.

[Gou01] Éric Goubault. Static analyses of floating-point operations. In P. Cousot, editor, *SAS'01*, LNCS 2126, pages 233–258, Paris, July 2001.

[GZ99] Gerhard Goos and Wolf Zimmermann. Verification of compilers. In *Correct System Design*, pages 201–230, 1999.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a c0 compiler. In *Proceedings, 3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)*, Koblenz, Germany, 5–9 September 2005.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pau05] Wolfgang Paul. Towards a worldwide verification technology. In *Proceedings of the Verified Software: Theories, Tools, Experiments Conference (VSTTE 2005)*, Zurich, Switzerland, October 2005.

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452, pages 398–414, 2005.

[Str91] John Strang. *Programming with curses*. O'Reilly & Associates, Inc., 1991.

Towards Determining the Subset Relation between Propositional Modal Logics

Florian Rabe[1,*]
[1]Carnegie Mellon University, International University Bremen

July 3, 2006

Abstract

We present design and implementation of a system that tries to automatically determine the subset relation between two given axiomatizations of almost arbitrary propositional modal logics, which is an open challenge problem for automated theorem proving. A test suite shows that relatively simple strategies can lead to satisfactory results, but also that certain subproblems are hard for current automated theorem provers.

1 Introduction

Plain propositional modal logic is standard knowledge (see, e.g., [HC96]). However, there is a fundamental problem that has not been addressed in a focussed way, namely the automated decision whether among two given axiomatizations one is weaker or stronger than the other one. For example, the Modal Logic $ 100 Challenge ([Sut]) calls for a program that can answer this problem for the most common Hilbert-style axiomatizations of K, T, S1, $S1_0$, S3, S4 and S5 (see [Hal] for an overview and a list of references to various modal logics and axiomatizations).

In general, this problem is undecidable (see [HV89] for an example). But decidability can be established separately for lots of modal logics (e.g., [HM92]), e.g., using a complete Kripke semantics ([Kri63]) and methods like filtration ([Lem66]). There is a variety of implementations (see, e.g., [HS00], [GTG02]) of theorem provers for specific modal logics.

However, such separate implementations are relative to a specific modal logic. Sometimes the set of axioms can be varied, in particular by imposing additional restrictions on the used Kripke frames like reflexivity and transitivity. But we are not aware of a modal theorem prover permitting an arbitrary set of inference rules. Furthermore, to decide whether two distinct axiomatizations generate the same logic, a decision procedure for that logic is difficult to use since it is not always clear which derived rules it uses implicitly. Also, checking the subset relation between modal logics requires to derive rules whereas theorem provers focus on deriving axioms.

*The author was supported by a fellowship for Ph.D. research of the German Academic Exchange Service.

The presented work describes a general approach to the above-mentioned challenge problem. We provide a modular framework into which different strategies can be plugged in. Thus incomplete strategies covering different cases can be combined. Experimental results show that in certain cases, this works surprisingly well and identify the subproblems that still present the greatest challenges.

Sect. 2 gives the main definitions, Sect. 3 and 4 describe the currently implemented strategies, Sect. 5 introduces the system, and Sect. 6 analyzes its current and potential future strength.

2 Definitions

We encode modal logic in first-order logic, i.e., every modal formula is a first-order term. We use the following symbols for the first-order meta logic: $\forall$, $\exists$, $\neg$, $\wedge$, $\Rightarrow$, $\doteq$. The theory $\mathcal{T}$ of first-order logic with equality is defined as follows:

- primitive function symbols *or* (disjunction, binary), *not* (negation) and *box* (necessity, both unary),
- further function symbols *and* (conjunction), *imp* (implication), *eqv* (equivalence), *simp* (strict implication), *seqv* (strict equivalence, all binary) and *dia* (possibility, unary) along with equality axioms that (classically) define these symbols in terms of the primitive ones, in particular strict implication and strict equivalence as
  - $\forall x, y\ simp(x, y) \doteq box(imp(x, y))$,
  - $\forall x, y\ seqv(x, y) \doteq and(box(imp(x, y)), box(imp(y, x)))$,
- a unary predicate symbol p (expressing provability).

A modal formula is a term of this first-order language, in particular propositional variables of modal logic are identified with first-order variables, and henceforth, just called variables. Provability of $f(X)$ is encoded as $\forall X\ p(f(X))$ (see below for the implicit use of uniform substitution). Here, and throughout the paper, X abbreviates $x_1, \ldots, x_m$.

Axiom/rule	Encoding
$\Box(A \to B) \to (\Box A \to \Box B)$	$\forall x, y\ p\big(imp(box(imp(x, y)), imp(box(y), box(x)))\big)$
$\dfrac{A \quad A \to B}{B}$	$\forall x, y\ \big((p(y) \wedge p(imp(x, y))) \Rightarrow p(y)\big)$
$\dfrac{A}{\Box A}$	$\forall x\ (p(x) \Rightarrow p(box(x)))$

Figure 1: Encoding of K

Furthermore, we define:

- A rule is an additional axiom for $\mathcal{T}$ that is in Horn form and does not contain equality. In other words, the rule scheme

$$\frac{h_1(X) \quad \ldots \quad h_n(X)}{c(X)}$$

  is encoded as

$$\forall X \big((p(h_1(X) \wedge \ldots \wedge p(h_n(X))) \Rightarrow p(c(X)) \big).$$

  If $n = 0$, the rule is called an axiom.

- A modal logic is an extension of $\mathcal{T}$ with a finite set of rules.

- A theorem of the modal logic L is a formula $f(X)$ such that $\forall X \ p(f(X))$ is a consequence of L. We write this as $f \in L$.

- A modal logic M is a subset of the modal logic L, written $M \subseteq L$, iff all theorems of M are theorems of L.

As an example, Fig. 1 gives the common and the encoded axiomatization of the modal logic K.

Let US be the rule of uniform substitution, i.e., substitution of all occurrences of a variable with the same modal formula. Then the above encoding is sound in the following sense: If L is a modal logic in our sense arising as the encoding of a Hilbert-style set S of axioms and rules and if f is the encoding a modal formula ϕ, then $f \in L$ iff ϕ is a theorem of $S \cup \{US\}$.

We only consider modal logics with the following two properties.

Firstly, the set of theorems is closed under uniform substitution. This is fundamental for the soundness of the above encoding. We do not know of any used modal logic that does not have this property.

Secondly, the set of theorems contains all classical propositional tautologies. This assumption is connected to a principal simplification of the challenge problem: We do not assume specific, let alone minimal axiomatizations for the propositional part. This greatly increases performance without dropping too much of the original challenge. In fact, it can be argued that proving propositional theorems from specific axiom sets should be another challenge independent from the modal logic challenge (see for example the problems in the Logic Calculi domain of TPTP ([SS98])). Furthermore, there is no discussion which propositional formulas are considered theorems (except for intuitionistic logic, but all modal logics considered in the challenge are classical) so that the choice of axioms becomes just a matter of implementation. For modal logic on the contrary, there are different applications and interpretations that naturally lead to different and not necessarily equivalent axiomatizations.

3 Positive Criteria

The most straightforward approach would be to let a first-order theorem prover prove the inclusion directly, e.g., by renaming the predicate p to q in one of two modal logics, say L_p and L_q and then trying to prove $\forall x(p(x) \Rightarrow q(x))$, which corresponds to $L_p \subseteq L_q$.

This is far from possible. Instead we break the problem down into subproblems treating every rule of L_q separately. The resulting problems can be attacked by existing tools.

In the sequel, we present simple criteria how to do that. These criteria are not necessarily complete and cover different situations. For the remainder of this section, let L and M be modal logics and let M' be as M but with an additional rule r.

Let $n = 0$, i.e., r is an axiom, say $r = \forall X p(c(X))$. Clearly, we have

Lemma 1. *$M' \subseteq L$ iff $M \subseteq L$ and $c \in L$*

This criterion can be used quite well by theorem provers. It can fail if r is complex. The following more sophisticated criterion using relational semantics and correspondence results can be applied less often but is successful in some cases in which Lem. 1 fails in practice.

Lemma 2. *Let*

1. *L be normal, i.e., $K \in L$ and L is closed under MP (modus ponens) and N (necessitation) (see Fig. 2 for the formulas; most of the time we use the same names as [Hal]),*
2. *F be a set of axioms of L that are Sahlqvist formulas,*
3. *P be the first-order property of Kripke frames completely characterized by F,*
4. *c' be the standard first-order translation of the modal formula c by making worlds explicit,*
5. *c' be first-order provable from P.*

Then $c \in L$.

By standard first-order translation, we mean the relational semantics of modal logics, e.g., $\Box P$ is translatated to $\forall w \forall x(Acc(w, x) \rightarrow P(x))$ for an accessibility realtion Acc.

This result is due to Sahlqvist [Sah75]. Lots of practically relevant axioms are Sahlqvist formulas, e.g., any formula of the form $A \rightarrow B$ where B is a positive formula and A is constructed by applying conjunction, disjuncton and possiblity to boxed atoms and negative formulas. P can be computed from F using the SCAN algorithm ([GO92]) for second-order quantifier elimination, for which an implementation is available. We are currently working on implementing this strategy, so far only the special case where P does not exclude any Kripke frames is used.

Note that we cannot use relational semantics in general since Kripke semantics may be not sound (e.g., for S1) or not complete (see, e.g., [Tho74]) for a given modal logic. It is necessary to find a set of Kripke frames that corresponds to the modal logic and show that this set of frames is complete for it. The above criterion gives the most important class of modal logics where this has been done.

If r is not an axiom, the situation is more complicated. The obvious naive approach is given by

Lemma 3. *$M' \subseteq L$ if $M \subseteq L$ and r is a first-order consequence of L.*

Axioms	
K	$\forall X, Y\ p(imp(box(imp(X,Y)), imp(box(X), box(Y))))$
T	$\forall X\ p(imp(box(X), X))$
4	$\forall X\ p(imp(box(X), box(box(X))))$
B	$\forall X\ p(imp(X, box(dia(X))))$
5	$\forall X\ p(imp(dia(X), box(dia(X))))$
T2_1	$\forall X, Y\ p(eqv(dia(or(X,Y)), or(dia(X), dia(Y))))$
T2_2	$\forall X\ p(imp(X, dia(X)))$
M1	$\forall X, Y\ p(simp(and(X,Y), and(Y,X)))$
M2, H2	$\forall X, Y\ p(simp(and(X,Y), X))$
M3	$\forall X, Y, Z\ p(simp(and(and(X,Y),Z), and(X, and(Y,Z))))$
M4, H1	$\forall X\ p(simp(X, and(X,X)))$
M5	$\forall X, Y, Z\ p(simp(and(simp(X,Y), simp(Y,Z)), simp(X,Z)))$
M6, H5	$\forall X\ p(simp(X, dia(X)))$
M8	$\forall X, Y\ p(simp(simp(X,Y), simp(dia(X), dia(Y))))$
M9, H7	$\forall X\ p(simp(dia(dia(X)), dia(X)))$
M10	$\forall X\ p(simp(dia(X), box(dia(X))))$
AS1_6	$\forall X, Y\ p(simp(and(X, simp(X,Y)), Y))$
H3	$\forall X, Y, Z\ p(simp(and(and(Z,X), not(and(Y,Z))), and(X, not(Y))))$
H4	$\forall X\ p(imp(not(dia(X)), not(X)))$
H6	$\forall X, Y\ p(simp(simp(X,Y), simp(not(dia(Y)), not(dia(X)))))$
Other Rules	
MP	$\forall X, Y\ ((p(X) \wedge p(imp(X,Y))) \Rightarrow p(Y))$
N	$\forall X\ (p(X) \Rightarrow p(box(X)))$
REM	$\forall X, Y\ (p(eqv(X,Y)) \Rightarrow p(eqv(dia(X), dia(Y))))$
SMP	$\forall X, Y\ ((p(X) \wedge p(simp(X,Y))) \Rightarrow p(Y))$
AD	$\forall X, Y\ ((p(X) \wedge p(Y)) \Rightarrow p(and(X,Y)))$

Figure 2: Rules

The opposite direction does not hold making this criterion incomplete. Essentially, two things can go wrong.

Firstly, deriving r from L means to show that whenever L contains instances of the hypotheses of r, it also contains the appropriate instance of the conclusion. The necessary and sufficient condition, however, is that whenever M' contains instances of the hypotheses, then L contains the appropriate instance of the conclusion. For a trivial example, let M be empty, r be the rule $\forall x(p(x) \Rightarrow p(not(x)))$, and L be any consistent non-empty modal logic. Clearly r is not derivable from L, but M' is empty (An axiomatization without axioms has no theorems.) and therefore, a subset of L.

Secondly, even if the opposite direction holds, Lem. 3 may be ineffective in practice, namely if r is only admissible in M but not derivable. For a simple special case of that, the following lemma gives an inductive admissibility criterion.

Lemma 4. *Let r be of the form $\forall x(p(x) \Rightarrow p(f(x)))$ for some formula f in one variable. Then $M' \subseteq L$ if*

- *$M \subseteq L$ and*
- *for every rule of L with hypotheses $h_1, \ldots, h_n$ and conclusion c,*

$$\forall X\big(\bigwedge_{i=1}^{n}\big(p(h_i(X)) \wedge p(f(h_i(X)))\big) \Rightarrow p(f(c(X)))\big)$$

is a first-order consequence of L.

This can be proven by a straightforward induction over the theorems of L (The second assumption is just what is needed to make the induction step.). The most important application is the case where $f(x) = box(x)$, i.e., where r is the necessitation rule.

Not all rules of modal logics can be easily encoded in the above way since some rules have complicated side conditions. The most important examples are (We refer to rules by abbreviations of their names, see Fig. 2 for the definitions.)

- substitution of strict equivalents (EQS): if f and $seqv(a, b)$ are theorems then f' which is as f but with a subformula a replaced with b is a theorem,
- propositional necessitation (Ga): if a is a theorem with no occurrences of box or dia, then $box(a)$ is a theorem.

It is reasonable to assume that any meta-language allowing for a natural encoding of these rules will be so strong that efficient automated solutions are unlikely. Instead we use

Lemma 5. *If M contains the rules SMP, AD, M1, M2, M4 and M5, then M enriched with EQS has the same theorems as M enriched with the following rules:*

- $\forall x, y, z(p(seqv(x, y)) \Rightarrow p(seqv(or(z, x), or(z, y))))$,
- $\forall x, y, z(p(seqv(x, y)) \Rightarrow p(seqv(or(x, z), or(y, z))))$,

- $\forall x, y(p(seqv(x, y)) \Rightarrow p(seqv(not(x), not(y))))$,
- $\forall x, y(p(seqv(x, y)) \Rightarrow p(seqv(box(x), box(y))))$,
- $\forall x, y((p(x) \wedge p(seqv(x, y))) \Rightarrow p(y))$.

This can be proven by using the assumptions made about M (without EQS) to show that strict equivalence is a congruence relation in M. That is equivalent to M being closed under EQS (see [Por80], [Tho68]). This lemma covers the most important case, in which EQS occurs, namely Lewis's family S1 to S5 ([Lew18]). In other cases, we are not able to express EQS.

A similar criterion for Ga is more complicated. While the necessary axiomatization of a further predicate *prop* that expresses the side condition is trivial, this predicate would not be preserved under uniform substitution and therefore, cannot be used. Instead a second sort, say s_2, with function symbols for propositional logic must be used along with with a predicate symbol t and an axiomatization such that $\forall X : s_2.\ t(f(X))$ expresses that f is a propositional tautology. Furthermore, a function symbol $i : s_2 \rightarrow s_1$ and axioms of the form

$$\forall x_1, x_2 : s_1, y_1, y_2 : s_2.\ \big((i(y_1) = x_1 \wedge i(y_2) = x_2) \Rightarrow i(and_2(y_1, y_2)) = and(x_1, x_2)\big)$$

are necessary.

If all previous definitions are appropriately extended to this more general setting, we have

Lemma 6. *If L has all propositional tautologies as theorems, then L enriched with Ga has the same theorems as L enriched with the rule $\forall y : s_2.(t(y) \Rightarrow p(box(i(y))))$.*

Currently, this is not supported by the implementation.

4 Negative Criteria

The simplest approach to disprove a subset relation $M \subseteq L$ is to use a first-order model finder to show that a theorem of M does not follow from L.

Lemma 7. *If f is a theorem of M, and if there is a first-order model for L enriched with $\exists X \neg p(f(X))$, then $M \not\subseteq L$.*

This approach is essentially the same as using matrix models for modal logics (see, e.g., [McK41]). It is not complete since only finite models can be checked, but this is often sufficient in practice.

Kripke models provide another strong criterion. It is clearly not complete but successful in many cases. The idea of this approach is to find a Kripke model $m' = (U, R, \alpha)$ such that the formulas satisfied by m' include the theorems of L but not f for some $f \in M$; here U is the set of worlds, R the accessibility relation, and α an assignment of truth values to the variables of f. Firstly, m' must satisfy all rules of L, i.e., an instance of the conclusion of a rule holds in all worlds whenever the appropriate instances of all hypotheses of the rule hold in all worlds. And secondly m' must satisfy $not(f)$ in one world.

Implementing that is less trivial than it sounds. If Kripke semantics is used to translate modal logic to first-order logic, the possibility to quantify over all formulas is lost, which is necessary to check whether a model satisfies a rule. And we are not aware of a model finder for (monadic) second-order logic. To circumvent this problem, we fix the number of worlds in U, say n, and proceed as follows.

We search for a first-order model m, from which m' can be generated. The signature for m contains constants $1, \ldots, n$ (intended semantics: one constant per world of U), the constant t (intended semantics: truth value of truth; we use any of the worlds as the truth value for falsity), and the binary predicate Acc (intended semantics: the accessibility relation R). m must satisfy that all worlds are different from each other and from t. Furthermore, the signature contains one constant x_i for every variable x occurring in f and for every $i = 1, \ldots, n$ (intended semantics: x_i gives the value of the assignment α to x in world i).

Let $\overline{\cdot}$ be the following translation from the language over $\mathcal{T}$ to this new signature:

- $\overline{\forall x\ F} = \forall x_1, \ldots, x_n\ \overline{F}$,
- the meta language connectives $\wedge$ and $\Rightarrow$ remain unchanged,
- $\overline{p(f(X))} = \bigwedge_{i=1}^{n} \overline{f(X)}(i)$ where $\overline{f(X)}(i)$ is given by the remaining cases,
- $\overline{and(f_1(X), f_2(X))}(i) = \overline{f_1(X)}(i) \wedge \overline{f_2(X)}(i)$ and accordingly for the other propositional connectives,
- $\overline{x}(i) = \ x_i \doteq t$ for a variable x,
- $\overline{box(f_1(X))}(i) = \bigwedge_{j=1}^{n} (Acc(i,j) \Rightarrow \overline{f_1(X)}(j))$,
- $\overline{dia(f_1(X))}(i) = \bigvee_{j=1}^{n} (Acc(i,j) \wedge \overline{f_1(X)}(j))$.

Here, the intended semantics of $\overline{\forall X\ p(f(X))}$ is that f holds in all worlds of m' and that of $\overline{f(X)}(i)$ is that f holds in the world i. Then the needed requirements for m are that it satisfies $\overline{r}$ for all rules r of L and $\overline{not(f(X))}(1)$.

These requirements can be sent to a first-order model finder. From the generated model m, m' is constructed by

- U: the universe of m minus the interpretation of t,
- R: the restriction of the interpretation of Acc to U,
- for a variable x of f and a world i of U, $\alpha(x)(i)$ is true if x_i is equal to t in m, and it is false otherwise.

Then we have

Lemma 8. *If f is a theorem of M and there is an n such that a model for L and $not(f)$ as described above can be found, then $M \not\subseteq L$.*

The follows directly from the above construction.

5 Implementation

The implementation of the presented criteria was done in SMLNJ ([SML]) and is available as [Rab]. It requires binary files for Vampire ([RV02]) and Paradox ([CS03]) to be present in the current directory because these are invoked to solve the generated subproblems.

The main function is invoked as `Main.compare(f1,f2)` where `f1` and `f2` are the names of files containing the modal logics in TPTP syntax ([SS98]). The content of the file `definitions.tptp` contains the propositional axiomatization and is added to each logic.

`compare` calls both `subsumes(f1,f2)` and `subsumes(f2,f1)`. Firstly, `subsumes(big,small)` calls `inclusion(big,small)` to prove `small` $\subseteq$ `big`. It tries all strategies given in the list `incl_strategies` trying to derive each rule of `small` as a consequence of `big`. Proven rules are added to `big`. If all rules are proven, a proof object is returned, otherwise the exception `fail` is raised.

In the latter case, the underived rules, say `l`, are passed to `noninclusion(big,l)`, which calls each disproving strategy given in the list `nonincl_strategies` with each element of `l`. If a counter-example is found a proof object is returned. `compare` collects these results and outputs the final result.

Optionally, whenever an external call is carried out the user is prompted for confirmation. It is possible to skip a call and choose whether the program should assume that the call has succeeded or failed.

Strategy	Criterion	Prover/Model finder	Remarks
`Direct`	Lem. 1, 3	Vampire	
`Inductive`	Lem. 4	Vampire	
`Sahlqvist`	Lem. 2	–	not implemented yet
`Containsk`	Lem. 2	Vampire	special case of Sahlqvist, implemented but not yet used in performance test
`Naivemodel`	Lem. 7	Paradox	implemented but not yet used in performance test
`Kripke`	Lem. 8	Paradox	

Figure 3: Strategies and provers

Strategies are functors that take a prover (or model finder) as an argument. Each strategy implements a sufficient criterion and tries to establish it by calling the prover. The provers are wrapper structures that invoke external first-order tools.

Strategies and provers are designed to be modular, which makes it easy to add further strategies and provers in the future. Currently only a minimal set of strategies is implemented, see Fig. 3. Testing showed that using the CASC competition ([PSS02]) winners Vampire and Paradox already gives quite satisfactory results.

$Log1$	Axiomatization	$Log2$	Axiomatization	R	R'	Pr. +	Pr. −
K	MP, N, K	M	MP, N, REM, T2_1, T2_2	$\subset$	$\not\supseteq$	K	
K	MP, N, K	S1_0	SMP, AD, EQS, M1-M5	$\neq$	−		inc.
S1_0	SMP, AD, EQS, M1-M5	S4	MP, N, K, T, 4	$\subset$	$\not\supseteq$	M5	
S1	SMP, AD, EQS, M1-M5, AS1.6	S4	MP, H1-H7	$\subset$	$\not\supseteq$	M5	
S3	SMP, AD, EQS, M1-M6, M8	S5	MP, N, K, T, B, 4	$\subset$	$\not\supseteq$	M5, M8	
S5	MP, N, K, T, 5	S5	SMP, AD, EQS, M1-M5, M10	$=$	$\subseteq$	M5	

Figure 4: Example instances and results

6 Experimental Results

The 100 $ challenge mentions eight logics with a total of about 25 axiomatizations that can be expressed in our encoding. This leads to about 600 combinations of logics. Fig. 4 shows the experimental results for a set of six examples. Running the program partially for other cases showed that this set is fairly representative for difficult instances. Here, the relationship between the logics (fifth column) is $Log1\ R\ Log2$ where $R \in \{\subset, \supset, =, \neq\}$. The program's answer R' is given in the sixth column: For partial success of the search, R' can also be in $\{\subseteq, \supseteq, \not\subseteq, \not\supseteq, -\}$. The symbol $\neq$ denotes incomparability, and $-$ denotes complete failure. Note that the strategies Containsk and Naivemodelwere not used for this test.

As can be seen, none of the cases could be solved completely. The seventh column gives the problems with positive criteria, which consisted of rules that could not be proven although they were provable. The eighth column gives the problems with negative criteria, which occurred only in one case where the used criterion was incomplete.

Comparing the run time shows that if a relation could be determined, this was done relatively fast: less than 30 minutes on a 3 GHz Intel Xeon machine. And this time was mainly spent waiting for the five-minute time limit of a failing Vampire call. Successful Vampire calls returned results within seconds, only sometimes minutes. Even more interesting, hardly any improvement could be observed by increasing Vampire's time limit: In the case of the third row, even 20 hours did not suffice to derive M5. Since M5 is a base axiom occurring in several natural axiomatizations, this becomes an interesting challenge problem for automated theorem proving. Calls to negative criteria did not contribute significantly to the run time.

This shows that the run time is essentially determined by the time limit for failing prover calls. Due to the nature of the problem, such calls occur frequently. This suggests that a more sophisticated switching between trying to prove inclusion or non-inclusion and varying the time limit for porver calls may lower the run time significantly.

The most interesting conclusion to be drawn about proving inclusion is that the complicated theoretical problems of rules that are only admissible but not derivable and of the incompleteness of the criterion in Lem. 3 are less relevant than expected: Both problems did not strike at all. All rules that could not be derived are axioms (K, M5 and M8). For these, however, the strategy `Direct` is complete. Apparently even a relatively small complexity of an axiom leads to de-facto unprovability.

As to proving non-inclusion, the incompleteness of the strategy `Kripke` hit twice: Both K and M1 to M5 hold in all Kripke models and their negation can never be satisfied. An obvious improvement of the current implementation is to apply rules that could not be derived to axioms in order to obtain more potential counter-examples. It is not clear if this will be successful in practice or if other approaches (like the recently implemented strategy `Naivemodel`) have to be considered.

It is not clear whether the strategy `Sahlqvist` obeys the spirit of the 100 $ challenge: Since it uses an external completeness result, it can be argued that the strategy uses hidden knowledge about the problems and is not a pure theorem prover. However, the performance of this strategy provides an enticing argument: Already the very special case implemented in the strategy `Containsk` (which does not even use the Sahlqvist results) can derive M5 in all four critical cases within seconds, and M8 can be derived by `Sahlqvist`. Thus, although this strategy can never help to prove K itself or non-normal axioms, it solves the problems in the last four rows of Fig. 4.

7 Conclusion and Future Work

We are bringing together different techniques to implement and integrate them into a framework directed at a specific challenge problem of theorem proving. Clearly, the current implementation is not much more than a proof of concept, but the current version already brings promising results. Implementing the algorithm of Lem. 2 fully or even partially will strengthen it significantly.

We also found that pure proof search without using semantical correspondence results seems to be practically incomplete for relevant problems. And the question how to derive non-normal axioms or K is open. However, we expect that a larger set of strategies (e.g., splitting as described in [Kra99]) will cover almost all interesting cases.

Less sophisticated possible future work includes

- utilizing different provers to exploit their respective strengths,
- tweaking, dynamically assigning or iteratively increasing the search depth, which is now fixed (5 minutes for Vampire, 4 worlds for Kripke models),
- redesigning the main functions to minimize the time spent with failing calls, in particular dynamically switching between trying to prove inclusion or non-inclusion, and keeping better track of positive and negative partial results,
- parallelizing external calls.

By publishing this work in progress, we hope to gain support and feedback for the further attack of this problem.

References

[CS03] K. Claessen and N. Sorensson. New techniques that improve MACE-style finite model finding. In *CADE-19 Workshop on Model Computation - Principles, Algorithms, Applications*, 2003.

[GO92] D. Gabbay and H. Ohlbach. Quantifier elimination in second-order predicate logic. *South African Computer Journal*, 1992.

[GTG02] E. Giunchiglia, A. Tacchella, and F. Giunchiglia. SAT-based decision procedures for classical modal logics. *Journal of Automated Reasoning*, 28(2):143–171, 2002.

[Hal] J. Halleck. Logic systems. See `http://www.cc.utah.edu/~nahaj/logic/structures/systems/index.html`.

[HC96] G. E. Hughes and M. J. Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996.

[HM92] J. Y. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54:319–379, 1992.

[HS00] U. Hustadt and R. A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2000)*, pages 67–71, 2000.

[HV89] J. Y. Halpern and M. Y. Vardi. The complexity of reasoning about knowledge and time. *Journal of Computer and System Sciences*, 38(1):195–237, 1989.

[Kra99] M. Kracht. *Tools and Techniques in Modal Logic*. Elsevier, 1999.

[Kri63] S. A. Kripke. Semantical analysis of modal logic I. Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Lem66] E. J. Lemmon. Algebraic Semantics for Modal Logics II. *The Journal of Symbolic Logic*, 31:191–218, 1966.

[Lew18] C. Lewis. *A Survey of Symbolic Logic*. University of California Press, 1918.

[McK41] J. C. McKinsey. A solution of the decision problem for the lewis systems s2 and s4 with an application to topology. *The Journal of Symbolic Logic*, 6:117–134, 1941.

[Por80] J. Porte. Congruences in Lemmon's S0.5. *Notre Dame Journal of Formal Logic*, 21(4):672–678, 1980.

[PSS02] F. Pelletier, G. Sutcliffe, and C. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[Rab] F. Rabe. Determining the subset relation between propositional modal logics. http://kwarc.eecs.iu-bremen.de/frabe/Research/moloss.

[RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15:91–110, 2002.

[Sah75] H. Sahlqvist. Completeness and correspondence in the first and second order semantics for modal logic. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, pages 110–143. North-Holland, 1975.

[SML] Standard ML of New Jersey. See http://www.smlnj.org.

[SS98] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[Sut] G. Sutcliffe. The modal logic $100 challenge. See http://www.cs.miami.edu/~tptp/HHDC/.

[Tho68] I. Thomas. Replacement in Some Modal Systems. *The Journal of Symbolic Logic*, 33(4):569–570, 1968.

[Tho74] S. K. Thomason. An incompleteness theorem in modal logic. *Theoria*, 40:30–34, 1974.

Prospective Logic Programming with ACORDA

Gonçalo Lopes and Luís Moniz Pereira
Centro de Inteligência Artificial - CENTRIA
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
goncaloclopes@gmail.com
lmp@di.fct.unl.pt

Abstract

As we face the real possibility of modelling programs that are capable of non-deterministic self-evolution, we are confronted with the problem of having several different possible futures for a single such program. It is desirable that such a system be somehow able to *look ahead*, prospectively, into such possible futures, in order to determine the best courses of evolution from its own present, and then to prefer amongst them. This is the objective of the ACORDA, a prospective logic programming system. We start from a real-life working example of differential medical diagnosis illustrating the benefits of addressing these concerns, and follow with a brief description of the concepts and research results supporting ACORDA, and on to their implementation. Then we proceed to fully specify the implemented system and how we addressed each of the enounced challenges. Next, we take on the proffered example, as codified into the system, and describe the behaviour of ACORDA as we carefully detail the resulting steps involved. Finally, we elaborate upon several considerations regarding the current limitations of the system, and conclude with the examination of possibilities for future work.

1 Introduction

Part of the Artificial Intelligence community has struggled, for some time now, to make viable the proposition of turning logic into an effective programming language, allowing it to be used as a system and application specification language which is not only executable, but on top of which one can demonstrate properties and proofs of correctness that validate the very self-describing programs which are produced. At the same time, AI has developed logic beyond the confines of monotonic cumulativity and into the non-monotonic realms that are typical of the real world of incomplete, contradictory, arguable, revised, distributed and evolving knowledge. Over the years, enormous amount of work and results have been achieved on separate topics in logic programming (LP) language semantics, belief revision, preferences, and evolving programs with updates[DP07, PP05, ABLP02].

As we now have the real possibility of modelling programs that are capable of non-deterministic self-evolution, through self-updating, we are confronted with the problem of having several different possible futures for a single starting program. It is desirable that such a system be able to somehow *look ahead* into such possible futures to determine the best paths of evolution from its present, at any moment. This involves

a notion of simulation, in which the program is capable of conjuring up hypothetical *what-if* scenarios and formulating abductive explanations for both external and internal observations. Since we have multiple possible scenarios to choose from, we need some form of preference specification, which can be either a priori or a posteriori. A priori preferences are embedded in the program's own knowledge representation theory and can be used to produce the most relevant hypothetical abductions for a given state and observations, in order to conjecture possible future states. A posteriori preferences represent choice mechanisms, which enable the program to commit to one of the hypothetical scenarios engendered by the relevant abductive theories. These mechanisms may trigger additional simulations in order to posit which new information to acquire, so more informed choices can be enacted, in particular by restricting and committing to some of the abductive explanations along the way.

A large class of higher-order problems and system specifications exist which would benefit greatly from this strategy for evolving knowledge representation, especially in domains closer to the human level of reasoning such as expert systems for medical diagnosis, and systems for enhanced proactive behavioural control.

1.1 Iterated Differential Diagnosis

In medicine, differential diagnosis is the systematic method physicians use to identify the disease causing a patient's symptoms. Before a medical condition can be treated it must first be correctly diagnosed. The physician begins by observing the patient's symptoms, taking into consideration the patient's personal and family medical history and performing additional examinations if current information is lacking. Then the physician lists the most likely causes. The physician asks questions and performs tests to eliminate possibilities until he or she is satisfied that the single most likely cause has been identified. The term differential diagnosis also refers to medical information specially organized to aid in diagnosis, particularly a list of the most common causes of a given symptom, annotated with advice on how to narrow down the list.

This listing of the most likely causes is clearly an abduction process, supported by initial observations. Encoding this process in logic programming would result in a set of literals and rules representing both the expected causes and the knowledge representation theory leading up to them. This set could be subsequently refined through a method of prospection with a priori and a posteriori preferences, like the one described above. This prospective process can result in the decision to perform additional observations on the external environment, and be iterated.

In this case, the external observations correspond to the disease's signs and other examinations performed by the physician during the process of diagnosis. The abductive explanations correspond to the possible causes (e.g. medical conditions or simply bad habits) responsible for the symptoms. A priori preferences and expectations are necessarily determined according to the patient's symptoms and they can be updated during the course of the prospection mechanism. An appropriate knowledge representation theory can be derived from current medical knowledge about the causes of a given disease.

Being such a good example of abduction, we provide next a practical example of iterated differential diagnosis, based upon real medical knowledge from the field of den-

tistry, in order to better illustrate the kinds of problems involved. Further on we shall proffer an encoding of the problem on our system, along with results from an interactive diagnosis session.

1.1.1 Differential Diagnosis in Dentistry: A Use Case

A patient shows up at the dentist with signs of pain upon teeth percussion. The expected causes for the observed signs are:

- Periapical lesion (endodontic or periodontal source)
- Horizontal Fracture of the root and/or crown
- Vertical Fracture of the root and/or crown

Several additional examinations can be conducted to determine the exact cause, namely:

- X-Ray for determination of radiolucency or fracture traces
- X-Ray for determination of periapical lesion source
- Check for tooth mobility
- Measurement of gingival pockets

Aside from presenting multiple hypotheses for diagnosis, the knowledge exhibited by the practitioner must necessarily evolve in time, as he or she performs relevant examinations which will attempt to disqualify all but one of the possible explanations. Current examinations depend on knowledge acquired in the past, which, in turn, will end up influencing the observations and inferences which will be drawn in the future. Developing a system that is capable of modelling the evolution of a program in order to draw such inferences is a demanding but obviously useful challenge.

1.2 Prospective Logic Programming

This problem is already at hand, since there are working implementations of a logic programming language which was specifically designed to model such program evolutions. EVOLP [ABLP02] is a non-deterministic LP language, having a well-defined semantics that accounts for self-updates, and we intend to use it to model autonomous agents capable of evolving by making proactive commitments concerning their imagined prospective futures. Such futures can also be prospections of actions, either from the outside environment or originating in the agent itself. This implies the existence of partial internal and external models, which can already be modelled and codified with logic programming.

It is important to distinguish this mechanism of prospection from your average planning problem, in which we search a state space for the means to achieve a certain goal. In planning, this search space is inherently fixed and implicit, by means of the set of possible actions that we can use to search for the goal. In future preference, we are precisely establishing that search space, defining our horizon of search, which isn't a priori known and can even change dynamically with the very process of searching. It

intends to be a pre-selection prior to planning, but essentially distinct from planning itself.

A full-blown theory of preferences[AP00] has also been developed in LP along with EVOLP, as well as an abduction theory [APS04] and several different working semantics for just such non-monotonic reasoning, such as the Stable Models [GL88] and Well-Founded Semantics [vGRS91]. All of these have thoroughly tested working implementations. The problem is then how to effectively combine these diverse ingredients in order to avail ourselves of the wherewithal to solve the problems described above. We are now ready to begin addressing such general issues with the tools already at hand, unifying several different research results into powerful implementations of systems exhibiting promising new computational properties. This is the main objective of the ACORDA system[1].

First, we provide a brief description of the concepts and previous research results supporting ACORDA, and their implementations[2]. Then we proceed to fully specify the implemented system and how we have addressed each of the enounced challenges. Next, we take on the real-life working example of differential medical diagnosis, showing how it can be codified into the system, and describing the behaviour of ACORDA, as we carefully detail the resulting steps. Finally, we elaborate upon several considerations regarding the current limitations of the system and conclude examining possibilities for future work.

2 Logic Programming Framework

2.1 XSB-XASP Interface

The Prolog language has been for quite some time one of the most accepted means to codify and execute logic programs, and as such has become a useful tool for research and application development in logic programming. Several stable implementations have been developed and refined over the years, with plenty of working solutions to pragmatic issues ranging from efficiency and portability to explorations of language extensions. The XSB Prolog system is one of the most sophisticated, powerful, efficient and versatile among these implementations, with a focus on execution efficiency and interaction with external systems, implementing program evaluation following the Well-Founded Semantics (WFS) for normal logic programs.

Two of its hallmark characteristics make it a particularly useful system on top of which to implement ACORDA, and many of its supporting subsystems. First of all, the tabling mechanism [Swi99], in which the results of particular queries are stored for later reuse, can provide not only an enormous decrease in time complexity, but also allow for solutions to well-known problems in the LP community, such as query loop detection.

Secondly, its aiming for external systems interaction eventually resulted in the development of an interface to Smodels [NS97], one of the most successful implementations of the Stable Models semantics over generalized logic programs, also known as the Answer

[1] ACORDA means literally "wake-up" in Portuguese. The *ACORDA* system project page is temporarily set up at: `http://articaserv.ath.cx/`

[2] Working implementations of *Dynamic Logic Programming*, *EVOLP* and *Updates plus Preferences using DLP* available online at: `http://centria.di.fct.unl.pt/~jja/updates`

Set semantics. The SM semantics has become the cornerstone for the definition of some of the most important results in logic programming of the past decade, providing an increase in logic program declarativity and a new paradigm for program evaluation. Many of the ACORDA subsystems are defined on top of the Stable Models (SM) semantics, and as such, this integration proves extremely useful, and even accounts for new and desirable computational properties that neither of the systems could provide on its own.

The XASP interface [CSW] (standing for XSB Answer Set Programming) provides two distinct methods of accessing Smodels[3]. The first one is for using Smodels to obtain the stable models of the so-called residual program, the one that results from a query evaluated in XSB using tabling. This residual program is represented by delay lists, that is, the set of undefined literals for which the program could not find a complete proof, due to mutual dependencies or loops over default negation for that set of literals, which are detected by the XSB tabling mechanism. This method allows us to obtain any two-valued semantics in completion to the three-valued semantics the XSB system provides. The second method is to build up a clause store, adding rules and facts to compose a generalized logic program that is then parsed and sent to Smodels for evaluation, thereafter providing access to the computed stable models back to the XSB system.

This kind of integration allows one to maintain the relevance property for queries over our programs, something that the Stable Models semantics does not originally enjoy. In Stable Models, by the very definition of the semantics, it is necessary to compute all the models for the whole program. In our system, we sidestep this issue, using XASP to compute the relevant residual program on demand, usually after some degree of transformation. Only the resulting program is then sent to Smodels for computation of possible futures. We believe that such system integrations are crucial in order to extend the applicability of the more refined and declarative semantics that have been developed in the field of AI.

2.2 Evolving Logic Programs

Modelling the dynamics of knowledge changing over time has been an important challenge for LP. Accounting for the specification of a program's own evolution is essential for a wide variety of modern applications, and necessary if one is to model the dynamics of real world knowledge. Several efforts were conducted in lieu of developing a unified language that could be both expressive and simple, following the spirit of declarative programming that is characteristic of LP. The language EVOLP [ABLP02] is one of the most powerful results from this research area with working implementations.

EVOLP generalizes LP in order to provide a general formulation of logic program updating, by permitting rules to indicate assertive conclusions having the form of program rules. Such assertions, whenever they belong to a model of the program P, can be employed to generate an updated version of P. This process can then be iterated on the basis of the new program. When the program semantics affords several program models, branching evolution will occur and several evolution sequences are possible. The ability of EVOLP to nest rule assertions within assertions allows rule updates to be themselves updated over time, conditional on each evolution strand. The ability to include assertive

[3]The XSB Logic Programming system and Smodels are freely available at:
`http://xsb.sourceforge.net` and `http://www.tcs.hut.fi/Software/smodels`

literals in rule bodies allows for looking ahead on program changes and acting on that knowledge before the changes actually take place.

2.2.1 Self-evolving Logic Programs

The approach used to define EVOLP aimed for minimality in regard to classic logic programs. The objective was to identify the necessary conditions allowing the new capabilities of evolution and updating, and then minimally adding constructs to LP in order to account for them. Since we are envisaging updates to logic programs, it is necessary to provide a means to state that, under some conditions, some rule is to be added to the program. However, to allow for the non-monotonicity of rules, it is also necessary to provide some sort of negation in rule heads, so as to let instances of older rules be supervened by more recent rules updating them. Under these minimal conditions it is possible to express any kind of logic program evolution and, as such, EVOLP aimed to satisfy both.

The negation in rule heads is provided by the very definition of generalized LPs, while the statement of updates can be specified by augmenting this language with the reserved predicate *assert/1*, whether as the rule head or in its body. The sole argument of this predicate is itself a full blown rule in order to account for the possibility of arbitrary nesting. The formal inductive definition of the EVOLP language can be presented thus:

Definition 1[ABLP02]. *Let $\mathcal{L}$ be any propositional language (not containing the predicate assert/1). The extended language $\mathcal{L}_{assert}$ is defined inductively as follows:*

1. *All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}_{assert}$.*
2. *If each of $L_0, \ldots, L_n$ is a literal in $\mathcal{L}_{assert}$ (i.e. a propositional atom A or its default negation not A), then $L_0 \leftarrow L_1, \ldots, L_n$ is a generalized logic program rule over $\mathcal{L}_{assert}$.*
3. *If R is a rule over $\mathcal{L}_{assert}$ then assert(R) is a propositional atom of $\mathcal{L}_{assert}$.*
4. *Nothing else is a propositional atom in $\mathcal{L}_{assert}$.*

An evolving logic program over a language $\mathcal{L}$ is a (possibly infinite) set of generalized logic program rules over $\mathcal{L}_{assert}$.

It should be noted that the basic syntax provides no explicit retract construct because in fact it has no need of one. Retraction of rules can be encoded in EVOLP simply by allowing for default negation to appear in rule heads.

The semantics of such self-evolving logic programs is provided by a set of *evolution stable models*, each of which is a sequence of interpretations or states. Each evolution stable model describes some possible self-evolution of one initial program after a given number n of evolution steps. Each self-evolution is represented by a sequence of programs, each program corresponding to a state.

These sequences of programs are treated as in Dynamic Logic Programs [ALP+00], where the most recent rules are put in force, and previous rule instances are valid by inertia insofar as possible, as long as they do not conflict with more recent ones.

2.3 Preferential Theory Revision

The application of preferential reasoning over logic programs in the form of preferences between rules has been successfully attempted, including combinations of such rule preferences with program updates[AP00], and the updating of preferences themselves. However, a crucial ingredient had been missing, that of considering the possible abductive extensions to a theory, as expressed by means of a logic program and the integration in it of preferences over such extensions.

Abduction plays a crucial role in belief revision and diagnosis, and also in the development of hypotheses to explain some set of observations, a common consequence of working under the scientific method. Such abductive extensions to a theory can be expressed by sets of *abducibles*, over which we should be able to express conditional priority relations. Abducibles may be thought of as the hypothetical solutions or possible explanations that are available for conditional proof of a given query.

This ability of construing plausible extensions to one's theory is also vital for logic program evolution, so that the program is capable of self-revision and theorizing, providing new and powerful ways on which it can guide the evolution of its knowledge base, by revising incorrect or incomplete behaviour.

Such preferential theory revision has been approached in [DP05, DP07] and will be presented next as part of the logic programming framework of the ACORDA prospective system.

2.3.1 Language

Let $\mathcal{L}$ be a first order language. A domain literal in $\mathcal{L}$ is a domain atom A or its default negation *not* A, the latter expressing that the atom is false by default (CWA). A domain rule in $\mathcal{L}$ is a rule of the form:

$$A \leftarrow L_1, \ldots, L_t \ \ (t \geq 0)$$

where A is a domain atom and $L_1, \ldots, L_t$ are domain literals. The following convention is used. Given a rule r of the form $L_0 \leftarrow L_1, \ldots, L_t$, we write $H(r)$ to indicate L_0 and $B(r)$ to indicate the conjunction $L_1, \ldots, L_t$. We write $B^+(r)$ to indicate the conjunction of all positive literals in $B(r)$, and $B^-(r)$ to indicate the conjunction of all negated literals in $B(r)$. When $t = 0$ we write the rule r simply as L_0. Let $\mathcal{A} \subseteq \mathcal{L}$ be a set of domain atoms, the *abducibles*.

2.3.2 Preferring Abducibles

To express preference criteria among abducibles, we introduce the language $\mathcal{L}^*$. A relevance atom is one of the form $a \lhd b$, where a and b are abducibles. $a \lhd b$ means that the abducible a is more relevant than the abducible b. A relevance rule is one of the form:

$$a \lhd b \leftarrow L_1, \ldots, L_t \ \ (t \geq 0)$$

where $a \lhd b$ is a relevance atom and every $L_i (1 \leq i \leq t)$ is a domain literal or a relevance literal. Let $\mathcal{L}^*$ be a language consisting of domain rules and relevance rules.

The original definition of *abductive stable models* given in [DP07] considered logic programs already infused with preferences between rules. The framework considered in this work does not assume this, and as such the transformation of programs over $\mathcal{L}^*$ is slightly different, as it considers only generalized LPs.

The following definition provides us with the syntactical transformation from the language $\mathcal{L}^*$ to normal logic programs, with the abducibles being codified as simultaneous even-loops that guarantee mutual exclusion, i.e. that only one abducible is present in each model. Relevance rules are also codified so as to defeat the abducibles which are less preferred when the body of the rule is satisfied.

Definition 2[DP05, DP07]. *Let Q be a program over $\mathcal{L}^*$ with set of abducibles $\mathcal{A}_Q = \{a_1, \ldots, a_m\}$. The program $P = \Sigma(Q)$ with abducibles $\mathcal{A}_P = \{abduce\}$ is obtained as follows:*

1. *P contains all the domain rules in Q*
2. *for every $a_i \in \mathcal{A}_Q$, P contains the domain rule:*
   $confirm(a_i) \leftarrow expect(a_i), not\ expect\_not(a_i)$
3. *for every $a_i \in \mathcal{A}_Q$, P contains the domain rule:*
   $a_i \leftarrow abduce, not\ a_1, \ldots, not\ a_{i-1}, not\ a_{i+1}, \ldots, not\ a_m,$
   $confirm(a_i), not\ neg\_a_i$
4. *for every relevance rule r in Q, P contains a set of domain rules obtained from r by replacing every relevance atom $x \lhd y$ in r with the following domain rule:*

   $neg_y \leftarrow L_0, \ldots, L_n, B_x^+, B_y, not\ neg_x$

3 ACORDA Architecture and Implementation

The basis for the ACORDA architecture is a working implementation of EVOLP on which we can evaluate for truth literals following both a three-valued or a two-valued logic. Since we aim for autonomous abduction processes that are triggered by constraints or observations, from within the knowledge state of an agent, we need a means to express how this triggering is accomplished. In our system, we resort to the notion of *observable* in order to model and specify this kind of behaviour.

An observable is a quaternary relation amongst the *observer*, i.e. that which is performing the observation; the *observed*, that which is the target of the observation; the result of the *observation* itself; and the *truth value* associated with the observation. This relation is valid both to express program based self-triggering of internal queries, as well as observations requested from an exterior oracle to the program, and from the program directly to the environment. For example, the observable

```
observable(prog,prog,Query,true)
```

represents an observation in which the observer is the program, that which is observed is also the program, the observation is the specified Query and the positive truth value means the observation must be proven true. In this case, we expect that such an

observable, by becoming active, triggers the self-evaluation of Query in the current knowledge state, in the process resorting to any relevant and expected abducibles that can account for the observation.

As a consequence of the fact that only one of the relevant abducibles can effectively be chosen, corresponding to some possible world model, we may have several different results for explaining such self-observations. These results represent the possible knowledge futures that the program can infer from the current knowledge state towards explaining a given hypothetical observation, and also the choices that the program can make in order to evolve towards a new state where the observation holds. To make the correct choice, we implement a priori and a posteriori prospective systems of preferences. After this prospection takes place, the system is updated with the relevant literals that explain the observation, and the transition to the next knowledge state is then complete. This iterative process can be repeated for as long as we like, producing autonomous evolution of the program, interleaved with external updates that can reach the system independently, in between every such iteration. For simplicity, we presently assume that external updates do not reach the system during each abductive process. The steps in an ACORDA evolution loop are detailed below, and we proceed to explain them in detail in the following sections.

1. Determine the active program-to-program generated observations for the current state and their respective queries.

2. Determine the expected abducibles which are confirmed relevant to support the active observations and which do not violate the integrity constraints.

3. Compute the abductive stable models generated by virtue of those confirmed abducibles, considering the currently active a priori preferences amongst them.

4. If necessary, activate a posteriori choice mechanisms to possibly defeat some of the resulting abductive stable models.

5. Update the current knowledge state with the chosen abducible that can effectively satisfy the activated observations and integrity constraints.

3.1 Active Program-to-Program Generated Observations

The active self-observations for a given state are obtained from a special kind of clause which indicates that certain program-to-program observations should be made, in exclusion of all others. These *on_observable/4* clauses follow the same structure of the *observable/4* clauses, but they do not represent the observations themselves. They point rather to the self-observations that the ACORDA system should perform in the next step of evolution. In order to ease codification in the ACORDA system, we consider additional *observable/3* and *on_observable/3* clauses, which default the truth value to `true`.

Those *on_observable/4* clauses whose body holds true in the Well-Founded Model (WFM), the three-valued logic model derived from the current knowledge state, will trigger an internal observation expressed by their conclusion. The reason why we opt for this skeptical semantics in this step is that we do not wish to allow just any kind of

abduction to determine such activated observations. Since we codify abduction using even-loops over default negation, we would risk having those even-loops generate several possible models for active observations if we considered the Stable Models semantics. With the Well-Founded Semantics, we guarantee that a single model of active observations exists, composed of those literals which can be proven true in the WFM, with any literals supported by abductions being marked as undefined. A much more expensive alternative, of course, would be to find those observations true in the intersection of all abductive Stable Models.

Also, in a knowledge base of considerable dimension, the number of active program-to-program observations for a given knowledge state can be huge. Typically, we want to give attention to only a subset of the observations that are active. There are multiple ways in which this subset could be determined, either by using utility functions, additional preference relations or priorities, or just any other means of determining the most important observables. In our system, we leave this function to be determined by the user, since it will hinge much on the kind of application being developed. By default, the system attends to all the observations that are active.

3.2 Expected and Confirmed Abducibles

Each active program-to-program observation will be launched as part of a conjunctive query, which the system will attempt to satisfy by relying on any expected abducibles which become confirmed by not otherwise being defeated, along with enforcing satisfaction of integrity constraints. Since only a single abducible can currently be derived from the even loops expressing alternative abductions, it follows that this abducible must be able to satisfy the conjunction of active observations and attending integrity constraints in order for evolution to succeed. This notion can of course be extended to support abducible sets, and in that case one may want to consider the minimal sets of abducibles that satisfy the conjunction. One way to allow for sets is to attach a single abducible to each desired alternative set. This is currently outside the scope and essence of the project, but will be considered in future developments, as mentioned in section 5.

Each abducible needs first to be expected (i.e. made available) by a model of the current knowledge state. Next it needs to be confirmed in that model before it can be used to satisfy or explain any kind of prospective observation. This is achieved via the *expect/1* and *expect_not/1* clauses, which indicate conditions under which an expectable abducible is indeed expected and confirmed for an observation given the current knowledge state. A relevant expected and confirmed abducible must be present in the current derivation tree for the conjunction of all observations, satisfy at least one *expect/1* clause and none of its *expect_not/1* clauses. It must also be coded over an even-loop using the system's general *abduce/1* clause, detailed below along with the general *confirm/1* clause encoding the requisite properties of an expected and confirmed abducible.

```
    confirm(X)  <- expect(X), not expect_not(X), abduce(X).
     abduce(X)  <- not abduce_not(X).
abduce_not(X)  <- abduce(X).
```

The latter two clauses are defined for every abducible by instantiating its variable to it, so as to produce an even-loop with the *abduce_not/1* clause, and guarantee that

the confirmed abducibles come up undefined in the WFM of the program, and hence in the residual program computed by the XSB Prolog-based EVOLP meta-interpreter as explained in section 2.1. This residual program is a set of cyclic strongly connected graphs of the interdependencies among the undefined atoms in the Well-Founded Model of a program, without the presence of any true or false literals, for these have already been partially evaluated[DMS].

3.3 Computation of the Abductive Stable Models

Determination of relevant abducibles can be performed by examination of the residual program for ground literals which are arguments to *confirm/1* clauses. Currently we need to assume ground literals since we will be producing a program transformation over the residual program based on the set of confirmed abducibles, which will be sent directly to Smodels for computation of the hypothetical confirmed abducibles' generated scenarios. Smodels needs all literals to be instantiated, or have an associated domain for instantiation. This limitation is present in many other state-of-the-art logic programming systems and its solution is not the main point of this work. Nevertheless, it is worth considering that XSB's unification algorithm can ground some of the variables automatically during the top-down derivation of observations hinging on abducibles.

Once the set of relevant confirmed abducibles is determined from the program's current knowledge state, all that remains before applying the preference transformation described in section 2.3 is to determine the active a priori preferences that are relevant for that set. This is merely a query for all preference literals whose heads indicate a preference between two abducibles that belong to the set, and whose body is true in the Well-Founded Model of the current knowledge state.

The XASP package[CSW] allows the programmer to collect rules in an XSB clause store. When the programmer has determined that enough clauses have been added to the store to form a semantically complete sub-program, the program is then *committed*. This means that information in the clauses is copied to Smodels, codified using Smodels data structures so that stable models of those clauses can be computed and examined.

When both the relevant abducibles and the active preferences are determined, the transformation is applied, and every resulting clause is sent to the XASP store, which is reset beforehand in preparation for the stable models computation. The transformed residual program is then committed to Smodels and we make use of the XASP interface to obtain back the literals corresponding to abducibles from all the resulting stable models of the transformed program. These models form the Abductive Stable Models of the current knowledge state given the active observations and available relevant abducibles.

3.4 A Posteriori Choice Mechanisms

If everything goes well and only a single model emerges from computation of the abductive stable models, the ACORDA cycle terminates, and the resulting abducible is updated into the next state of the knowledge base. In most cases, however, we cannot guarantee the emergence of a single model, since the active preferences may not be sufficient to defeat enough abducibles. In these situations, the ACORDA system has to resort on additional information for making further choices, but supported on what?

A given abducible can be defeated in any one of two cases: either by satisfaction of an *expect_not/1* clause for that abducible, or by satisfaction of a preference rule that prefers another abducible instead. However, the current knowledge state may be insufficient to satisfy any of these cases for all abducibles except one, or else a single model would have already been abduced. It is then necessary that the system obtain the answers it needs from somewhere else, namely from making experiments on the environment or from querying an outside entity.

An agent or program that is completely isolated from its environment is limited in the kinds of reasoning it can produce. If we are to model agents which can face up proactively to the problems of the real world, they need to have the ability to probe the outside environment in order to make experiments which permit it to enact a more informed choice. We next consider the mechanism by means of which the program can pose questions to external systems, be they other agents, actuators, sensors or other procedures. Each of these serves the purpose of an *oracle*, which the program can probe with observations of its own. These observations are of the form

```
observable(prog,Oracle,Query,Value)
```

representing that the program is performing the observation Query on the specified Oracle. These clauses can normally only be satisfied if the system has activated oracle probing, and if the conditions are met for using the prescribed oracle. In some situations, not all oracles may be available for the agent to query.

ACORDA consequently activates its a posteriori choice mechanisms, which consist in attempting to satisfy additional self-observations of the top-down conjunctive query taking into account the possible satisfaction of more *expect_not/1* clauses or preference rules. Each such observation is an ACORDA cycle in its own right, counting on enabled oracles' activation to perform external queries if necessary. Any such observation can spawn even more observations, as long as there are clauses to inspect and oracles to interrogate. Information gleaned from the oracles can produce numerous side-effects, besides the possible defeat of previous abducibles, namely: supporting new abductions; activating preference relations previously unaccounted for; or even lending support by themselves to the satisfaction of the original query.

The results from performed experiments are currently tabled, to allow for reuse of the retrieved information for the current knowledge state. We assume the environment does not change while the ACORDA system is performing its abductive reasoning. There are many situations in which this assumption would be unacceptable but, once again, these fall outside the immediate scope of the project. They are discussed as current system limitations in section 5.

Eventually, the ACORDA system will have defeated enough models that it can guarantee a single one will emerge with the additional information gathered, or that it has exhausted every means to make the choice. In any case, it will commit to the literals gathered from the observations, querying the user for the final choice, if indeed it fails to produce a single course of evolution by itself.

The finalizing committing update can, of course, trigger additional observations and turn the attention of the system to other matters, but always in a well-founded manner, so that the agent is capable of autonomous reasoning justified by its initial program

and self-guided evolution. We believe that the ACORDA system is an interesting experiment on new forms of combined reasoning that deal with new emerging problems already beginning to present themselves. Moreover, even though an experimental system, it's already able to solve empirically concrete real-world problems of some degree of complexity with classical LP declarative programming style, as the next section will demonstrate.

4 Modelling Prospective Logic Programs

We will now resume the issue of codifying the example of differential diagnosis in dentistry presented in section 1.1.1 using the ACORDA system. The results obtained during an interactive diagnosis section are then detailed, along with annotations on the workings of the system.

The scenario can be intuitively modelled using the provided syntax for prospective logic programs on the ACORDA system, resulting in the initial program exhibited below. It should be noted that, as a matter of coding syntax, the relevance operator $\triangleleft$ is represented by the atom `<|`.

```
==== Initial Signs ====

on_observable(prog,prog,percussion_pain_cause) <- percussion_pain. (A)

==== First Phase of Differential Diagnosis ====

percussion_pain_cause <- periapical_lesion.
percussion_pain_cause <- vertical_fracture.
percussion_pain_cause <- horizontal_fracture.

periapical_lesion <- confirm(periapical_lesion).
expect(periapical_lesion) <- profound_caries.
expect(periapical_lesion) <- % expected in the context of an observation:
    on_observable(prog,prog,percussion_pain_cause).                (B)
expect_not(periapical_lesion) <- fracture_traces, not radiolucency.

vertical_fracture <- confirm(vertical_fracture).
expect(vertical_fracture) <- on_observable(prog,prog,percussion_pain_cause).
expect_not(vertical_fracture) <- radiolucency, not fracture_traces.

horizontal_fracture <- confirm(horizontal_fracture).
expect(horizontal_fracture) <- on_observable(prog,prog,percussion_pain_cause).
expect_not(horizontal_fracture) <- radiolucency, not fracture_traces.

periapical_lesion <| horizontal_fracture <-
    profound_caries, not percussion_pain.

periapical_lesion <| vertical_fracture <-
    profound_caries, not percussion_pain.

horizontal_fracture <| vertical_fracture <- low_mobility.
vertical_fracture <| horizontal_fracture <- high_mobility.
```

```
==== Second Phase of Differential Diagnosis ====

on_observable(prog,prog,periapical_lesion_source) <-                (C)
    periapical_lesion.

periapical_lesion_source <- endodontic_lesion.
periapical_lesion_source <- periodontal_lesion.

endodontic_lesion <- confirm(endodontic_lesion).
expect(endodontic_lesion) <-
    on_observable(prog,prog,periapical_lesion_source).
expect(endodontic_lesion) <- devitalization.
expect(endodontic_lesion) <- not gingival_pockets.
expect_not(endodontic_lesion) <-                                    (D)
    gingival_pockets, not devitalization.

periodontal_lesion <- confirm(periodontal_lesion).
expect(periodontal_lesion) <-
    on_observable(prog,prog,periapical_lesion_source).
expect(periodontal_lesion) <- devitalization.
expect(periodontal_lesion) <- gingival_pockets.
expect_not(periodontal_lesion) <- neg_gingival_pockets.

periodontal_lesion <| endodontic_lesion <- gingival_pockets.

==== Available Experiments ====

radiolucency <- observable(prog,xray,radiolucency).
fracture_traces <- observable(prog,xray,fracture_traces).           (E)
high_mobility <- observable(prog,mobility_check,high_mobility).
low_mobility <- observable(prog,mobility_check,low_mobility).
gingival_pockets <- observable(prog,pockets_check,gingival_pockets).
neg_gingival_pockets <-
    observable(prog,pockets_check,gingival_pockets,false).
devitalization <- observable(prog,periapical_xray,devitalization).

==== Available Oracles ====

observable(prog,xray,Q,S) <- oracle,prolog((oracleQuery(xray(Q),T),S = T)).
observable(prog,mobility_check,Q,S) <-
    oracle,prolog((oracleQuery(mobility_check(Q),T),S = T)).
observable(prog,pockets_check,Q,S) <-
    oracle,prolog((oracleQuery(pockets_check(Q),T),S = T)).
observable(prog,periapical_xray,Q,S) <-
    oracle,prolog((oracleQuery(periapical_xray(Q),T),S = T)).
```

4.1 Interactive Session Results with ACORDA

After the initial program is loaded into the ACORDA evolving knowledge base, we provide the patient's signs (i.e. `percussion_pain`) as an external update to the system.

Running a cycle of introspection over the ACORDA system produces the reasoning steps detailed below. The question marks are used to indicate queries to an external oracle, followed by the answer instances given by it.

1. `About to launch new introspection on selected active observables:`
   `[on_observable(prog,prog,percussion_pain_cause)]`
2. `Relevant abducibles for current introspection:`
   `[horizontal_fracture,periapical_lesion,vertical_fracture]`
3. `Partial models remaining after a priori preferences:`
   `[[horizontal_fracture],[periapical_lesion],[vertical_fracture]]`
4. `About to launch new introspection on selected active observables:`
   `[on_observable(prog,prog,percussion_pain_cause)]`
5. `Confirm observation: xray(fracture_traces)`
   `(true, false or unknown)? false.`
6. `Confirm observation: xray(radiolucency)`
   `(true, false or unknown)? true.`
7. `Relevant abducibles for current introspection:`
   `[periapical_lesion]`
8. `Partial models remaining after a priori preferences:`
   `[[periapical_lesion]]`

The event `percussion_pain` triggers the active observable `percussion_pain_cause` (cf. rule A) which is a program-to-program generated observation, that is, it will cause a top-down query to be launched to attempt satisfaction of the observable. The system goes down the derivation tree for `percussion_pain_cause`, encountering even-loops over default negation for each relevant abducible in the query, i.e. the literals that are output in 2. In this case, the expectations are triggered only in the context of the currently active observation (cf. rule B).

Afterwards, ACORDA launches queries for the a priori preferences which concern the relevant abducibles, which can themselves be dependent on abductive or observable loops. These abducibles and preferences determine the transformation specified in section 2.3. For the computed residual program and the set of abducibles in 2, we show part of the performed transformation, which is then placed in the XASP clause store:

```
percussion_pain_cause <- horizontal_fracture.
percussion_pain_cause <- periapical_lesion.
percussion_pain_cause <- vertical_fracture.
confirm(horizontal_fracture) <- not expect_not(horizontal_fracture).
confirm(periapical_lesion) <- not expect_not(periapical_lesion).
confirm(vertical_fracture) <- not expect_not(vertical_fracture).
expect_not(horizontal_fracture) <- radiolucency,not fracture_traces.
expect_not(periapical_lesion) <- fracture_traces,not radiolucency.
expect_not(vertical_fracture) <- radiolucency,not fracture_traces.

horizontal_fracture <-
    abduce, confirm(horizontal_fracture), not periapical_lesion,
    not vertical_fracture, not neg_horizontal_fracture.
```

```
false <- abduce, horizontal_fracture, neg_horizontal_fracture.

periapical_lesion <-
    abduce, confirm(periapical_lesion), not horizontal_fracture,
    not vertical_fracture, not neg_periapical_lesion.
false <- abduce, periapical_lesion, neg_periapical_lesion.

vertical_fracture <-
    abduce, confirm(vertical_fracture), not horizontal_fracture,
    not periapical_lesion, not neg_vertical_fracture.
false <- abduce,vertical_fracture,neg_vertical_fracture.
```

The 'false' literal is equivalently used here to model integrity constraints in our program, instead of the usual empty head that is adopted in Smodels' implementation. The potential observations which can satisfy some of the literals in the residual programs, namely the *expect_not/1* clauses are codified over even-loops, but are ommitted for clarity, since in the first phase of the system no external observations are allowed.

After the clauses are stored, the stable models of the processed residual program are computed, resulting in at least a model for each abducible that could not be defeated by contextual preference rules. In this case no active preferences hold and no observations are allowed, so the resulting stable models correspond to all the confirmed abducibles: `periapical_lesion`, `horizontal_fracture` and `vertical_fracture`. Since the system was not able to abduce just one model from a priori preferences, the choice mechanisms are activated and a new prospective search is launched for ways to defeat some of the models, as described in section 3.

However, the current knowledge state is insufficient to enact any kind of choice, so the system is forced to probe the external environment for additional information. This means that the oracle mechanisms are activated, and for the new observations the system decrees that experiments can now be performed. A new top-down query is launched, and as the derivation tree is traversed for new confirmation of relevant abducibles, opportunities to perform experiments are encountered. This is the case for the satisfaction of `expect_not(periapical_lesion)` which depends on `fracture_traces` and `not radiolucency`.

In an attempt to satisfy `fracture_traces`, ACORDA encounters an observable clause (cf. rule E). This clause depends on fracture traces being found on X-Ray imaging, so it probes the environment for just such an exam. No fracture traces are found, so periapical lesion can be confirmed under support from the oracle experiment. However, the subsequent experiment for `radiolucency` reveals positive results which can guarantee defeat of the other two abducibles, meaning that `periapical_lesion` is now the only expected and confirmed abducible. ACORDA can now successfully commit to a single diagnosis.

As mentioned above, in the cases where the system cannot guarantee a single abductive stable model after exhausting all the introspections, it queries the user to perform the final choice over the list of surviving abducibles. This behaviour could be refined, however, by launching separate simulations assuming each of the remaining solutions, and performing extra levels of prospective lookahead for additional ways to defeat further models. This addition is considered in the future work mentioned in section 5.

After the abduction of periapical lesion as the most likely cause for the sign of percussion pain on the patient's tooth, the attention of the system turns to satisfy a new observation for the source of such lesion. As such, a new cycle of introspection on top of these results can produce a more detailed diagnosis, as depicted below:

1. `About to launch new introspection on selected active observables:`
   `[on_observable(prog,prog,periapical_lesion_source)]`
2. `Relevant abducibles for current introspection:`
   `[endodontic_lesion,periodontal_lesion]`
3. `Partial models remaining after a priori preferences:`
   `[[endodontic_lesion],[periodontal_lesion]]`
4. `About to launch new introspection on selected active observables:`
   `[on_observable(prog,prog,periapical_lesion_source)]`
5. `Confirm observation: pockets_check(gingival_pockets)`
   `(true, false or unknown)? true.`
6. `Confirm observation: periapical_xray(devitalization)`
   `(true, false or unknown)? true.`
7. `Relevant abducibles for current introspection:`
   `[endodontic_lesion,periodontal_lesion]`
8. `Partial models remaining after a priori preferences:`
   `[[periodontal_lesion]]`

This time, the commitment to the abduction of `periapical_lesion` triggers the activation of program-to-program observable `periapical_lesion_source` (cf. rule C). The relevant expected abducibles that satisfy this observation are `endodontic_lesion` and `periodontal_lesion`, both being confirmed under the current knowledge state. Again, no active preferences hold a priori, that is, there are no preference rules relevant to the current abducibles which do not depend on any external observations. As a result, the Stable Models of the transformed partial residual program correspond once again to all relevant abducibles.

The choice mechanisms are activated and the top goal is relaunched in an attempt to acquire additional information, activating the oracles for external environment probing. The attempt to defeat `endodontic_lesion` calls for a gingival pockets measurement (cf. rule D), which reveals the existence of pockets in the vicinity of the patient's tooth. On the other hand, a periapical X-Ray is needed to confirm that no endodontic therapy was performed on that tooth. This experiment, however, reveals just that, and so ACORDA is unable to defeat the `endodontic_lesion` abducible using this *expect_not/1* clause.

Attempting to defeat `periodontal_lesion` by means of *expect_not/1* clauses proves impossible as well, due once again to the ambiguous results from the experiments. Both abducibles are again confirmed. However, a preference for `periodontal_lesion` is in place, given the existence of gingival pockets, so a unique Stable Model emerges, yielding the final diagnosis of `periodontal_lesion`.

The system could now be extended to handle different treatment hypotheses according to the result of the diagnosis. It would just be a matter of adding new triggers for additional observations that would represent the adequate treatment. Abducibles in this case would be the expected treatments and the preference model could be extended to include the patient's own preferences.

5 Conclusions and Future Work

The ACORDA project is still in its early beginnings, but even now it addresses several new pragmatic problems involving self-modifying logic programs by combining some of the best results from research and development of logic-based systems in the last decade. Preferential reasoning techniques are applied and the stable models semantics is used in an effective way by means of the XASP package. The relevance property enjoyed by top-down querying in XSB Prolog is crucial in preparation for our use of Smodels, and so we can expect to handle problems with a large rule-base, as we do not need to compute whole models, and none of those with irrelevant abducibles, in order to derive the relevant abductive stable models specific to the problem at hand.

This is one of the main problems which abduction over stable models has been facing, in that it always has to consider all the abducibles in a program and then progressively defeat all those that are irrelevant for the problem at hand. This is not so in the ACORDA system, since we begin by a top-down derivation that immediately constrains the set of abducibles that are relevant to the agent's observations and meta-goals. However, the encoding of abducibles as even-loops over default negation provides a declarative way in which to describe the abductive process, and the computation of stable models of the residual program is a natural way to obtain all the possible 2-valued models, considering also the a priori preferences, themselves coded as even loops over default negation.

Since virtually every element that is necessary to the process of abduction is encoded in the system's knowledge base, it automatically becomes a possible target of revision, meaning that the agent can update itself in order to change its preferences, and even the abductive rules, supervening old rules and adding new ones at will. Only the process of simulation in itself needs to remain separate, but that is just a practical consequence of the well-known self-reference problem. A program that simulates another program's future cannot be inside that very program. The process must be executed by an external system that is not included in the model that it is trying to simulate.

There are currently several avenues of improvement of the ACORDA system, orthogonal to several fields of active research in logic programming. To begin with, the stable models semantics may not be sufficient to model program evolution, since it does not guarantee the existence of models. Once we consider non-deterministic evolution of an agent, especially if it is expected to retrieve new rules from the environment or to compose old rules to form new ones or learn them, we cannot guarantee that odd-loops over default negation will not be part of the program somewhere in the future. If that happens, evolution is immediately halted, as stable models does not provide any model in that case.

It can be argued that inserting an odd-loop over default negation is simply a case of bad programming, and as such, it should never appear at all. However, if our agent makes a mistake, we want at least to give it an opportunity to revise that mistake, and not kill its evolution altogether. Fortunately, recent research on logic programming semantics has produced the Revised Stable Models [PP05] semantics, which elegantly solves this problem and brings a whole new host of desirable properties to the original SM semantics. Working implementations have already been developed and they will soon be integrated into the ACORDA system.

The abductive process and the system of a priori preferences can be improved as well, in order to allow for the abduction of a set of abducibles. It will be necessary to specify new ways in which we can express preferences over these sets, but work is already well underway in this regard. Abduction of multiple literals is, however, already possible in the current system, by making single abducibles themselves stand for sets of "abducible" literals. One just needs to carefully model all the relevant combinations of literals that one would be inclined to expect.

The integration of action prospections is also a must to tackle even more complex problems and applications, in order to deal with pre- and post-conditions. Namely, the pre-conditions of an action must be evaluated before the update for the action actually takes place, but the post-conditions must also be taken in consideration during the simulation and before the real action is executed. Also, we would like to extend the mechanism to an arbitrarily long sequence of actions, which would require an extensible amount of lookahead into the future.

In fact, there is a wide variety of problems for which we would like to be able to arbitrarily extend the future lookahead that is possible within the ACORDA system. One of the most important challenges in future developments is devising a way to do this that will also allow the specification of the degree of self-control over the amount of lookahead that the agent performs.

The need for a finer time stamping of events is also growing, since it would permit generalization of the validity of experiments. Instead of using tabling to hold the results of experiments, we would like to specify a time interval during which those results are assumed valid. This would be the first step in which the system could be extended to handle environment changes during simulation, and could even be used to model reactions of the system to such changes, or to introduce timeouts on the abductive process, if it proves to be taking too long.

Preferences over observables will also be desirable, since not every observation costs the same for the agent. Performing an X-Ray costs more than checking for tooth mobility or gingival pockets, and these differences should be modelled directly in the system. It would also be interesting to study more general ways of selecting the most interesting internal observations for ACORDA to pay attention to at each evolution step.

We have just started exploration of a territory that is vast in possibilities and, as such, we come out bearing more questions than answers. Even if we have a working system for such autonomous preferential reasoning and self-updating, the more interesting possibilities are still farther ahead, as we attempt to broach more and more of the goal of completely automated artificial reasoning, adumbrating its combinations.

6 Acknowledgments

We would like to thank Joana Nogueira for preparing and supplying scientific information for the case study on differential medical diagnosis, Pierangelo Dell'Acqua for constructive criticism on the definition of several specification and operational issues regarding the ACORDA system, and a number of colleagues at DEIS, U. Bologna, for prior stimulating discussions.

References

[ABLP02] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Procs. of the 8th European Conf. on Logics in Artificial Intelligence (JELIA'02)*, LNCS 2424, pages 50–61, Cosenza, Italy, September 2002. Springer.

[ALP+00] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *J. Logic Programming*, 45(1-3):43–70, September/October 2000.

[AP00] J. J. Alferes and L. M. Pereira. Updates plus preferences. In M. O. Aciego, I. P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *Procs. of JELIA'00*, LNAI 1919, pages 345–360, Málaga, Spain, 2000. Springer.

[APS04] J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, July 2004.

[CSW] L. Castro, T. Swift, and D. S. Warren. *XASP: Answer Set Programming with XSB and Smodels*. `http://xsb.sourceforge.net/packages/xasp.pdf`.

[DMS] N. Drakos, R. Moore, and H. Singh. *The XSB System Programmer's Manual*. `http://xsb.sourceforge.net/manual1/`.

[DP05] P. Dell'Acqua and L. M. Pereira. Preferential theory revision. In L. M. Pereira and G. Wheeler, editors, *Procs. Computational Models of Scientific Reasoning and Applications*, pages 69–84, Universidade Nova de Lisboa, Lisbon, Portugal, September 2005.

[DP07] P. Dell'Acqua and L. M. Pereira. Preferential theory revision (extended version). *J. Applied Logic*, 2007. Forthcoming.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th Intl. Logic Programming Conf.*, pages 1070–1080. MIT Press, 1988.

[NS97] I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *4th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning*, LNAI 1265, pages 420–429, Berlin, 1997. Springer.

[PP05] L. M. Pereira and A. M. Pinto. Revised stable models - a semantics for logic programs. In *12th Portuguese Intl. Conf. on Artificial Intelligence (EPIA'05)*, LNAI 3808, pages 29–42, Covilhã, December 2005. Springer.

[Swi99] T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):201–240, 1999.

[vGRS91] A. van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.

System Description: The Cut-Elimination System CERES *

Matthias Baaz[1], Stefan Hetzl[2], Alexander Leitsch[2], Clemens Richter[2], Hendrik Spohr[2]

[1]Institute of Discrete Mathematics and Geometry (E104), Vienna University of Technology, Wiedner Hauptstraße 8-10, 1040 Vienna, Austria
baaz@logic.at

[2]Institute of Computer Languages (E185), Vienna University of Technology, Favoritenstraße 9, 1040 Vienna, Austria
{hetzl|leitsch|richter|spohr}@logic.at

Abstract

Cut-elimination is the most prominent form of proof transformation in logic. The elimination of cuts in formal proofs corresponds to the removal of intermediate statements (lemmas) in mathematical proofs. The cut-elimination method CERES (cut-elimination by resolution) works by constructing a set of clauses from a proof with cuts. Any resolution refutation of this set then serves as a skeleton of an **LK**-proof with only atomic cuts.

The use of resolution and the enormous size of (formalized) mathematically relevant proofs suggest an implementation able to handle rather complex cut-elimination problems. In this paper we present an implementation of CERES: the system CERES. It already implements an improvement based on an extension of **LK** to the calculus **LKDe** containing equality rules and rules for introduction of definitions which makes it easier to formalize and interpret mathematical proofs in **LK**. Furthermore it increases the efficiency of the cut-elimination method. The system CERES already performs well in handling quite large proofs.

1 Introduction

Proof analysis is a central mathematical activity which proves crucial to the development of mathematics. Indeed many mathematical concepts such as the notion of group or the notion of probability were introduced by analyzing existing arguments. In some sense the analysis and synthesis of proofs form the very core of mathematical progress [12, 13].

Cut-elimination introduced by Gentzen [8] is the most prominent form of proof transformation in logic and plays an important role in automating the analysis of mathematical proofs. The removal of cuts corresponds to the elimination of intermediate

*supported by the Austrian Science Fund (project no. P17995-N12)

statements (lemmas) from proofs resulting in a proof which is analytic in the sense, that all statements in the proof are subformulas of the result. Therefore, the proof of a combinatorial statement is converted into a purely combinatorial proof. Cut elimination is therefore an essential tool for the analysis of mathematical proofs, especially to make implicit parameters explicit. Cut free derivations allow for

- the extraction of Herbrand disjunctions, which can be used to establish bounds on existential quantifiers (e.g. Luckhardt's analysis of the Theorem of Roth [11]).
- the construction of interpolants, which allow for the replacement of implicit definitions by explicit definitions according to Beth's Theorem.
- the calculation of generalized variants of the end formula.

In a formal sense Girard's analysis of van der Waerden's theorem [9] is the application of cut-elimination to the proof of Fürstenberg/Weiss with the "perspective" of obtaining van der Waerden's proof. Indeed an application of a complex proof transformation like cut-elimination by humans requires a goal oriented strategy.

Note that cut-elimination is *non-unique*, i.e. there is no single cut-free proof which represents *the* analytic version of a proof with lemmas. Therefore the application of purely computational methods on existing proofs may even produce new interesting proofs. Indeed, it is non-uniqueness which makes computational experiments with cut-elimination interesting [3]. The experiments can be considered as a source for a base of proofs in formal format which provide different mathematical and computational information.

CERES [5, 6] is a cut-elimination method that is based on resolution. The method roughly works as follows: The structure of the proof containing cuts is mapped to an unsatisfiable set of clauses $\mathcal{C}$ (the *characteristic clause set*). A resolution refutation of $\mathcal{C}$, which is obtained using a first-order theorem prover, serves as a skeleton for the new proof which contains only atomic cuts. In a final step also these atomic cuts can be eliminated, provided the (atomic) axioms are valid sequents; but this step is of minor mathematical interest and of low complexity. In the system `CERES`[1] this method of cut-elimination has been implemented. The system is capable of dealing with formal proofs in an extended version of **LK**, among them also very large ones (i.e. proofs with more than 1500 nodes and cut-elimination has been done within 14 seconds).

The extension of CERES to **LKDe** a calculus containing definition introductions and equality rules [10] moves the system closer to real mathematical proofs. In particular, introduction of definitions and concepts is perhaps the most significant activity of a mathematician in structuring proofs and theories. By its high efficiency (the core of the method is first-order theorem proving by resolution and paramodulation) `CERES` might become a strong tool in *automated proof mining* and contribute to an experimental culture of *computer-aided proof analysis* in mathematics.

[1] available at `http://www.logic.at/ceres/`

2 The System CERES

The cut-elimination system CERES is written in ANSI-C++. There are two main tasks. One is to compute an unsatisfiable set of clauses characterizing the cut formulas. This is done by automatically extracting the so-called *characteristic clause set* $\mathrm{CL}(\varphi)$. The other is to generate a resolution refutation of $\mathrm{CL}(\varphi)$ by an external theorem prover[2], and to compute the necessary projections of φ w.r.t. the clauses in $\mathrm{CL}(\varphi)$ actually used for the refutation. A projection of φ modulo a clause $C \in \mathrm{CL}(\varphi)$ is gained by applying all inference rules of φ not operating on cut ancestors to the initial sequents of C. The properly instantiated projections are concatenated, using the refutation obtained by the theorem prover as a skeleton of a proof with only atomic cuts.

The extraction of the *characteristic clause set* $\mathrm{CL}(\varphi)$ from an **LKDe**-proof φ is defined inductively. For every node ν of φ either:

- If ν is an occurrence of an axiom S and S' is the subsequent of S containing only ancestors of cut formulas then $\mathcal{C}_\nu = \{ S' \}$.
- Let ν_1, ν_2 be the predecessors of ν in a binary inference then we distinguish:
  - The auxiliary formulas of ν_1, ν_2 are ancestors of cut formulas then $\mathcal{C}_\nu = \mathcal{C}_{\nu_1} \cup \mathcal{C}_{\nu_2}$.
  - Otherwise $\mathcal{C}_\nu = \mathcal{C}_{\nu_1} \times \mathcal{C}_{\nu_2}$, where $\mathcal{C} \times \mathcal{D} = \{ C \circ D \mid C \in \mathcal{C}, D \in \mathcal{D} \}$ and $C \circ D$ is the merge of the clauses C and D.
- Let ν' be the predecessor of ν in a unary inference then $\mathcal{C}_\nu = \mathcal{C}_{\nu'}$.

The characteristic clause set $\mathrm{CL}(\varphi)$ is defined as $\mathcal{C}_\nu$ where ν is the root node of φ.

The *definition rules* of **LKDe** directly correspond to the *extension principle* (see [7]) in predicate logic. It simply consists in introducing new predicate- and function symbols as abbreviations for formulas and terms. Mathematically this corresponds to the introduction of new concepts in theories. Let A be a first-order formula with the free variables $x_1, \ldots, x_k$ (denoted by $A(x_1, \ldots, x_k)$ and P be a *new* k-ary predicate symbol (corresponding to the formula A). Then the rules are:

$$\frac{A(t_1, \ldots, t_k), \Gamma \vdash \Delta}{P(t_1, \ldots, t_k), \Gamma \vdash \Delta}\ \mathrm{def}_P\!: l \qquad \frac{\Gamma \vdash \Delta, A(t_1, \ldots, t_k)}{\Gamma \vdash \Delta, P(t_1, \ldots, t_k)}\ \mathrm{def}_P\!: r$$

for arbitrary sequences of terms $t_1, \ldots, t_k$. There are also definition introduction rules for new function symbols which are of similar type. The *equality rules* are:

$$\frac{\Gamma_1 \vdash \Delta_1, s = t \quad A[s]_\Lambda, \Gamma_2 \vdash \Delta_2}{A[t]_\Lambda, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2} =\!: l1 \qquad \frac{\Gamma_1 \vdash \Delta_1, t = s \quad A[s]_\Lambda, \Gamma_2 \vdash \Delta_2}{A[t]_\Lambda, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2} =\!: l2$$

for inference on the left and

$$\frac{\Gamma_1 \vdash \Delta_1, s = t \quad \Gamma_2 \vdash \Delta_2, A[s]_\Lambda}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, A[t]_\Lambda} =\!: r1 \qquad \frac{\Gamma_1 \vdash \Delta_1, t = s \quad \Gamma_2 \vdash \Delta_2, A[s]_\Lambda}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, A[t]_\Lambda} =\!: r2$$

[2] The current version of CERES uses the automated theorem prover Otter (see http://www-unix.mcs.anl.gov/AR/otter/), but any refutational theorem prover based on resolution and paramodulation may be used.

on the right, where Λ denotes a set of positions of subterms where replacement of s by t has to be performed. We call $s = t$ the *active equation* of the rules. Note that, on atomic sequents, the rules coincide with *paramodulation* – under previous application of the most general unifier.

Concerning the extension of the CERES-method from **LK** to **LKDe**, equality rules appearing within the input proof are propagated to the projections like any other binary rules. During theorem proving equality is treated by paramodulation; its application within the final clausal refutation is then transformed to the appropriate equality rules in **LKDe**. The definition introductions do not require any other special treatment within `CERES` than all other unary rules; in particular, they have no influence on the theorem proving part.

Since the restriction to skolemized proofs is crucial to the CERES-method, the system also performs skolemization (according to Andrew's method [2]) on the input proof.

The system `CERES` expects an **LKDe** proof of a sequent S and a set of axioms as input, and, after validation of the input proof, computes a proof of S containing at most atomic-cuts. Input and output are formatted using the well known data representation language XML. This allows the use of arbitrary and well known utilities for editing, transformation and presentation and standardized programming libraries. To increase the performance and avoid redundancy, most parts of the proofs are internally represented as directed acyclic graphs. This representation turns out to be very handy, also for the internal unification algorithms.

The formal analysis of mathematical proofs (especially by a mathematician as a pre- and post-"processor") relies on a suitable format for the input and output of proofs, and on an appropriate aid in dealing with them. We developed an intermediary proof language connecting the language of mathematical proofs with **LKDe** and the compiler `HLK`[3] transforming proofs written in this higher-level language to **LKDe**. Furthermore we implemented a proof tool[4] with a graphical user interface, allowing a convenient input and analysis of the output of `CERES`. Thereby the integration of definition- and equality-rules into the underlying calculus plays an essential role in overlooking, understanding and analyzing complex mathematical proofs by humans.

3 Example

The example proof below is taken from [14]; it was formalized in **LK** and analyzed by a former version of CERES in the paper [3]. Here we use the extensions by equality rules and definition-introduction in **LKDe** to give a simpler formalization and analysis of the proof. The end-sequent formalizes the statement: on a tape with infinitely many cells which are all labelled by 0 or by 1 there are at least two cells labelled by the same number. $f(x) = 0$ expresses that the cell number x is labelled by 0. Indexing of cells is done by number terms defined over $0, 1$ and $+$. The proof φ below uses two lemmas: (1) there are infinitely many cells labelled by 0 and (2) there are infinitely many cells labelled by 1. The proof shows the statement by a case distinction: If there are infinitely many cells labelled by 0 then choose two of them, otherwise there are infinitely many

[3] available at `http://www.logic.at/hlk/`

[4] available at `http://www.logic.at/prooftool/`

cells labelled by 1 and we can then choose two of them. In each case there are two cells with the same value. These lemmas are eliminated by CERES and a more direct argument is obtained in the resulting proof φ'. Ancestors of the cuts in φ are indicated in boldface.

Let φ be the proof

$$\dfrac{\dfrac{\begin{matrix}(\tau) \\ A \vdash \mathbf{I_0}, \mathbf{I_1}\end{matrix} \quad \begin{matrix}(\epsilon_0) \\ \mathbf{I_0} \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))\end{matrix}}{A \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q)), \mathbf{I_1}}\ cut \quad \begin{matrix}(\epsilon_1) \\ \mathbf{I_1} \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))\end{matrix}}{A \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))}\ cut$$

where $\tau =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix}(\tau') \\ A \vdash \mathbf{f(n_0 + n_1) = 0}, \mathbf{f(n_1 + n_0) = 1}\end{matrix}}{A \vdash \mathbf{f(n_0 + n_1) = 0}, \exists \mathbf{k}.\mathbf{f(n_1 + k) = 1}}\ \exists r}{A \vdash \exists \mathbf{k}.\mathbf{f(n_0 + k) = 0}, \exists \mathbf{k}.\mathbf{f(n_1 + k) = 1}}\ \exists r}{A \vdash \exists \mathbf{k}.\mathbf{f(n_0 + k) = 0}, \forall \mathbf{n} \exists \mathbf{k}.\mathbf{f(n + k) = 1}}\ \forall : r}{A \vdash \forall \mathbf{n} \exists \mathbf{k}.\mathbf{f(n + k) = 0}, \forall \mathbf{n} \exists \mathbf{k}.\mathbf{f(n + k) = 1}}\ \forall : r}{A \vdash \mathbf{I_0}, \forall \mathbf{n} \exists \mathbf{k}.\mathbf{f(n + k) = 1}}\ \mathrm{def}_{I_0} : r}{A \vdash \mathbf{I_0}, \mathbf{I_1}}\ \mathrm{def}_{I_1} : r$$

For $\tau' =$

$$\dfrac{\dfrac{\dfrac{f(n_0 + n_1) = 0 \vdash \mathbf{f(n_0 + n_1) = 0} \quad \dfrac{\begin{matrix}(\text{Axiom}) \\ \vdash n_1 + n_0 = n_0 + n_1\end{matrix} \quad f(n_1 + n_0) = 1 \vdash \mathbf{f(n_1 + n_0) = 1}}{f(n_0 + n_1) = 1 \vdash \mathbf{f(n_1 + n_0) = 1}}\ =: l1}{f(n_0 + n_1) = 0 \vee f(n_0 + n_1) = 1 \vdash \mathbf{f(n_0 + n_1) = 0}, \mathbf{f(n_1 + n_0) = 1}}\ \vee : l}{\forall x (f(x) = 0 \vee f(x) = 1) \vdash \mathbf{f(n_0 + n_1) = 0}, \mathbf{f(n_1 + n_0) = 1}}\ \forall : l}{A \vdash \mathbf{f(n_0 + n_1) = 0}, \mathbf{f(n_1 + n_0) = 1}}\ \mathrm{def}_A : l$$

And for $i = 1, 2$ we define the proofs $\epsilon_i =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\psi \quad \eta_i}{\mathbf{f(s) = i}, \mathbf{f(t) = i} \vdash s \neq t \wedge f(s) = f(t)}\ \wedge : r}{\mathbf{f(s) = i}, \mathbf{f(t) = i} \vdash \exists q (s \neq q \wedge f(s) = f(q))}\ \exists : r}{\mathbf{f(s) = i}, \mathbf{f(t) = i} \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))}\ \exists : r}{\mathbf{f(n_0 + k_0) = i}, \exists \mathbf{k}.\mathbf{f(((n_0 + k_0) + 1) + k) = i} \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))}\ \exists : l}{\mathbf{f(n_0 + k_0) = i}, \forall \mathbf{n} \exists \mathbf{k}.\mathbf{f(n + k) = i} \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))}\ \forall : l}{\exists \mathbf{k}.\mathbf{f(n_0 + k) = i}, \forall \mathbf{n} \exists \mathbf{k}.\mathbf{f(n + k) = i} \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))}\ \exists : l}{\forall \mathbf{n} \exists \mathbf{k}.\mathbf{f(n + k) = i}, \forall \mathbf{n} \exists \mathbf{k}.\mathbf{f(n + k) = i} \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))}\ \forall : l}{\dfrac{\forall \mathbf{n} \exists \mathbf{k}.\mathbf{f(n + k) = i} \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))}{\mathbf{I_i} \vdash \exists p \exists q (p \neq q \wedge f(p) = f(q))}\ \mathrm{def}_{I_i} : l}\ c : l$$

for $s = n_0 + k_0$, $t = ((n_0 + k_0) + 1) + k_1$, and the proofs

$\psi =$

$$\dfrac{\dfrac{\begin{matrix}(\text{axiom}) \\ \vdash (n_0 + k_0) + (1 + k_1) = ((n_0 + k_0) + 1) + k_1\end{matrix} \quad \begin{matrix}(\text{axiom}) \\ n_0 + k_0 = (n_0 + k_0) + (1 + k_1) \vdash\end{matrix}}{n_0 + k_0 = ((n_0 + k_0) + 1) + k_1 \vdash}\ =: l1}{\vdash n_0 + k_0 \neq ((n_0 + k_0) + 1) + k_1}\ \neg : r$$

and $\eta_i =$

$$\dfrac{\mathbf{f(s)} = \mathbf{i} \vdash f(s) = i \qquad \dfrac{\mathbf{f(t)} = \mathbf{i} \vdash f(t) = i \quad \overset{\text{(axiom)}}{\vdash i = i}}{\mathbf{f(t)} = \mathbf{i} \vdash i = f(t)} =: r2}{\mathbf{f(s)} = \mathbf{i}, \mathbf{f(t)} = \mathbf{i} \vdash f(s) = f(t)} =: r2$$

The characteristic clause set is (after variable renaming)

$$\begin{aligned} \mathrm{CL}(\varphi) \quad = \quad & \{\vdash f(x+y) = 0, f(y+x) = 1; \quad (C_1) \\ & f(x+y) = 0, f(((x+y)+1)+z) = 0 \vdash; \quad (C_2) \\ & f(x+y) = 1, f(((x+y)+1)+z) = 1 \vdash\} \quad (C_3). \end{aligned}$$

The axioms used for the proof are the standard axioms of type $A \vdash A$ and instances of $\vdash x = x$, of commutativity $\vdash x + y = y + x$, of associativity $\vdash (x + y) + z = x + (y + z)$, and of the axiom

$$x = x + (1 + y) \vdash,$$

expressing that $x + (1 + y) \neq x$ for all natural numbers x, y.

The comparison with the analysis of Urban's proof formulated in **LK** (without equality) [3] shows that the proof based on equality is much more transparent. In fact the set of characteristic clauses contains only 3 elements (instead of 5), which are also simpler. This also facilitates the refutation of the clause set and makes the output proof simpler and more transparent. On the other hand, the analysis below shows that the mathematical argument obtained by cut-elimination is the same as in [3].

The program Otter found the following refutation of CL(φ) (based on hyperresolution only — without equality inference):

The first hyperesolvent, based on the clash sequence $(C_2; C_1, C_1)$, is

$$\begin{aligned} C_4 \quad &= \quad \vdash f(y+x) = 1,\ f(z + ((x+y)+1)) = 1, \text{ with the intermediary clause} \\ D_1 \quad &= \quad f(((x+y)+1)+z) = 0 \vdash f(y+x) = 1. \end{aligned}$$

The next clash is sequence is $(C_3; C_4, C_4)$ which gives C_5 with intermediary clause D_2, where:

$$\begin{aligned} C_5 \quad &= \quad \vdash f(v' + u') = 1,\ f(v+u) = 1, \\ D_2 \quad &= \quad f(x+y) = 1 \vdash f(v+u) = 1. \end{aligned}$$

Factoring C_5 gives C_6: $\vdash f(v+u) = 1$ (which roughly expresses that all fields are labelled by 1). The final clash sequence $(C_3; C_6, C_6)$ obviously results in the empty clause $\vdash$ with intermediary clause D_3: $f(((x+y)+1)+z) = 1 \vdash$. The hyperresolution proof ψ_3 in form of a tree can be obtained from the following resolution trees, where C' and ψ' stand for renamed variants of C and of ψ, respectively:

$\psi_1 :=$

$$\dfrac{\dfrac{C_1 \quad C_2}{D_1}\ \text{res} \qquad C_1'}{C_4}\ \text{res}$$

$\psi_2 :=$

$$\dfrac{\dfrac{\dfrac{C_3' \quad \psi_1}{D_2}\,\text{res} \qquad \psi_1'}{\dfrac{C_5}{C_6}\,\mathit{factor}}\,\text{res}}{}$$

$\psi_3 :=$

$$\dfrac{\dfrac{\dfrac{\psi_2 \quad C_3''}{D_3}\,\text{res} \qquad \psi_2'}{\vdash}\,\text{res}}{}$$

Instantiation of ψ_3 by the uniform most general unifier σ of all resolutions gives a deduction tree $\psi_3\sigma$ in **LKDe**; indeed, after application of σ, resolution becomes cut and factoring becomes contraction. The proof $\psi_3\sigma$ is the skeleton of an **LKDe**-proof of the end-sequent with only atomic cuts. Then the leaves of the tree $\psi_3\sigma$ have to be replaced by the proof projections. E.g., the clause C_1 is replaced by the proof $\varphi[C_1]$, where $s = n_0 + n_1$ and $t = n_1 + n_0$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash t = s}\,\text{(Axiom)} \qquad \dfrac{}{f(t) = 1 \vdash f(t) = 1}\,\text{(Axiom)}}{f(s) = 1 \vdash f(t) = 1}\,{=}{:}\,l}{f(s) = 0 \vdash f(s) = 0 \qquad f(s) = 1 \vdash f(t) = 1}\,{\vee}{:}\,l}{f(s) = 0 \vee f(s) = 1 \vdash f(s) = 0, f(t) = 1}\,{\forall}{:}\,l}{\forall x(f(x) = 0 \vee f(x) = 1) \vdash f(s) = 0, f(t) = 1}\,\text{def}_A{:}\,l}{A \vdash f(s) = 0, f(t) = 1}\,w{:}\,r}{A \vdash \exists p \exists q(p \neq q \wedge f(p) = f(q)), f(s) = 0, f(t) = 1}$$

Furthermore C_2 is replaced by the projection $\varphi[C_2]$ and C_3 by $\varphi[C_3]$, where (for $i = 0, 1$) $\varphi[C_{2+i}] =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\psi \quad \eta_i}{f(s) = i, f(t) = i \vdash s \neq t \wedge f(s) = f(t)}\,{\wedge}{:}\,r}{f(s) = i, f(t) = i \vdash \exists q(s \neq q \wedge f(s) = f(q))}\,{\exists}{:}\,r}{f(s) = i, f(t) = i \vdash \exists p \exists q(p \neq q \wedge f(p) = f(q))}\,{\exists}{:}\,r}{f(s) = i, f(t) = i, A \vdash \exists p \exists q(p \neq q \wedge f(p) = f(q))}\,w{:}\,l$$

Note that ψ, η_0, η_1 are the same as in the definition of ϵ_0, ϵ_1 above.

By inserting the σ-instances of the projections into the resolution proof $\psi_3\sigma$ and performing some additional contractions, we eventually obtain the desired proof φ' of the end-sequent

$$A \vdash \exists p \exists q(p \neq q \wedge f(p) = f(q))$$

with only *atomic* cuts. φ' no longer uses the lemmas that infinitely many cells are labelled by 0 and by 1, respectively.

4 Intended Extensions of CERES And Future Work

We plan to develop the following extensions of CERES:

- As the cut-free proofs are often very large and difficult to interpret, we intend to provide the possibility to analyze certain characteristics of the cut-free proof (which are simpler than the proof itself). An important example are Herbrand sequents which may serve to extract bounds from proofs (see e.g. [11]). We plan to develop algorithms for extracting Herbrand sequents (also from proofs of non-prenex sequents as indicated in [4]) and for computing interpolants, which allow for the replacement of implicit definitions by explicit ones according to Beth's Theorem.

- In the present version `CERES` eliminates all cuts at once. But - for the application to real mathematical proofs human user interaction is required — in particular only interesting cuts (i.e. lemmas of mathematical importance) deserve to be eliminated, others should be integrated as additional axioms.

- As CERES requires the skolemization of the end-sequent the original proof must be transformed to Skolem form. We plan to develop an efficient *reversed-skolemization*-algorithm, which transforms the theorem to be proved into its original form.

- A great challenge in the formal analysis of mathematical proofs lies in providing a suitable format for the input and output of proofs. We plan to improve our intermediary proof language and to move closer to the "natural" language of mathematical proofs.

To demonstrate the abilities of `CERES` and the feasibility of formalizing and analyzing complex proofs of mathematical relevance, we currently investigate a well known proof of the infinity of primes using topology (which may be found in [1]). Our aim is to eliminate the topological concepts from the proof by means of CERES, breaking it down to a proof solely based on elementary number arithmetic. This way one can obtain new insights into the construction of prime numbers contained in the topological proof.

References

[1] M. Aigner, G. M. Ziegler: *Proofs from THE BOOK*. Springer, 1998.

[2] P. B. Andrews: Resolution in Type Theory, *Journal of Symbolic Logic*, 36, pp. 414–432, 1971.

[3] M. Baaz, S. Hetzl, A. Leitsch, C. Richter, H. Spohr: Cut-Elimination: Experiments with CERES, *Proc. LPAR 2004*, pp. 481–495, Springer, 2004.

[4] M. Baaz, A. Leitsch: On Skolemization and Proof Complexity, *Fundamenta Informaticae*, 20(4), pp. 353–379, 1994.

[5] M. Baaz, A. Leitsch: Cut-Elimination and Redundancy-Elimination by Resolution, *Journal of Symbolic Computation*, 29, pp. 149–176, 2000.

[6] M. Baaz, A. Leitsch: Towards a Clausal Analysis of Cut-Elimination, *Journal of Symbolic Computation*, 41, pp. 381–410, 2006.

[7] E. Eder: Relative complexities of first-order calculi, Vieweg, 1992.

[8] G. Gentzen: Untersuchungen über das logische Schließen, *Mathematische Zeitschrift*, 39, pp. 405–431, 1934–1935.

[9] J. Y. Girard: *Proof Theory and Logical Complexity*, in Studies in Proof Theory, Bibliopolis, Napoli, 1987.

[10] A. Leitsch, C. Richter: Equational Theories in CERES, unpublished (available at `http://www.logic.at/ceres/`), 2005.

[11] H. Luckhardt: Herbrand-Analysen zweier Beweise des Satzes von Roth: polynomiale Anzahlschranken, *The Journal of Symbolic Logic*, 54, pp. 234–263, 1989.

[12] G. Polya: *Mathematics and plausible reasoning, Volume I: Induction and Analogy in Mathematics*, Princeton University Press, Princeton, New Jersey, 1954.

[13] G. Polya: *Mathematics and plausible reasoning, Volume II: Patterns of Plausible Inference*, Princeton University Press, Princeton, New Jersey, 1954.

[14] C. Urban: *Classical Logic and Computation* Ph.D. Thesis, University of Cambridge Computer Laboratory, 2000.

CEL—A Short System Demonstration*

Boontawee Suntisrivaraporn, Franz Baader, Carsten Lutz
Theoretical Computer Science, TU Dresden, Germany
{meng,baader,clu}@tcs.inf.tu-dresden.de

Introduction to the CEL system.

Description logics (DLs) are an important family of formalisms for reasoning about ontologies. CEL (Classifier for $\mathcal{EL}$) is a free (for non-commercial use) LISP-based reasoner for the description logic $\mathcal{EL}^+$ [2], supporting as its main reasoning task the computation of the subsumption hierarchy induced by $\mathcal{EL}^+$ ontologies. The most distinguishing feature of CEL is that, unlike other modern DL reasoners, it implements a polynomial-time algorithm, which allows it to process very large ontologies in reasonable time. The underlying description logic $\mathcal{EL}^+$ is a practically useful sub-language of $\mathcal{EL}^{++}$ introduced in [1]. Despite being less expressive than other description logics such as $\mathcal{SHIQ}$ and OWL, it offers a selected set of expressive means that are tailored towards the formulation of biological and medical ontologies, some of the most prominent of which are the Gene Ontology (GO) [7], the Galen Medical Knowledge Base (GALEN) [5], and the Systematized Nomenclature of Medicine (SNOMED) [3, 6].

The purpose of the present short paper is to demonstrate the functionalities of CEL and their use in applications. The implemented algorithm is described in detail in [2]. In the following section, a quick guide to system usage will be given, including the system requirements. In the last section, performance evaluation of CEL on real-world life science ontologies will be discussed.

CEL at work.

The CEL system is available as a binary executable which can run on most Linux platforms. The latest version is CEL `v0.9b` which includes all features illustrated in this system demonstration. The distribution bundle can be obtained from:

`http://lat.inf.tu-dresden.de/systems/cel/`

The package consists of the CEL executable, the user manual, and some example $\mathcal{EL}^+$ ontologies. After extracting the bundle, the executable `cel` under `bin` can be invoked without installation. However, the following system requirements are assumed:

- Linux operating system;[1]

*This work was partially supported by the EU funded IST-2005-7603 FET Project Thinking Ontologies (TONES).

[1] It has been tested successfully on RedHat, Debian, and SuSE.

Concept	DL Syntax	CEL Syntax
the top concept	$\top$	top
the bottom concept	$\bot$	bottom
conjunction	$C_1 \sqcap \cdots \sqcap C_n$	(and $C_1 \cdots C_n$)
existential restriction	$\exists r.C$	(some r C)

Table 1: Syntax of $\mathcal{EL}^+$ concepts.

Ontology axioms	DL Syntax	CEL Syntax
primitive concept definition	$A \sqsubseteq D$	(define-primitive-concept A D)
concept definition	$A \equiv D$	(define-concept A D)
general concept inclusion	$C \sqsubseteq D$	(implies C D)
concept equivalence axiom	$C \equiv D$	(equivalent C D)
concept disjointness axiom	$C_1 \sqcap \cdots \sqcap C_n \sqsubseteq \bot$	(disjoint $C_1 \cdots C_n$)
role domain axiom	$\exists r.\top \sqsubseteq C$	(define-primitive-role r :domain C)
role hierarchy axiom	$r \sqsubseteq s$	(define-primitive-role r :parent s)
transitive role axiom	$r \circ r \sqsubseteq r$	(define-primitive-role r :transitive t)
right-identity axiom	$r \circ s \sqsubseteq r$	(define-primitive-role r :right-identity s)
left-identity axiom	$s \circ r \sqsubseteq r$	(define-primitive-role r :left-identity s)
complex role inclusion	$r_1 \circ r_2 \sqsubseteq s$	(role-inclusion (compose r_1 r_2) s)

Table 2: Syntax $\mathcal{EL}^+$ ontology axioms.

- Physical memory at least 128MB;[2]
- At least 8MB of available hard-disk space.

CEL as a stand-alone reasoner. In order to use CEL to classify an ontology, the user must already have the ontology formulated in $\mathcal{EL}^+$ in a small extension of the KRSS syntax [4], henceforth called CEL syntax. With this LISP-like syntax, it is easy to port existing ontologies that have been used with well-known DL reasoners like FaCT and RACER. Table 1 shows the concept constructors available in $\mathcal{EL}^+$ that can be used to form concept descriptions. For building up ontologies, the expressive means shown in Table 2 can be used, where conventionally A, B denotes a named concept, C, D concept descriptions, and r, s named roles. Though only implies and role-inclusion axioms can sufficiently model any $\mathcal{EL}^+$ ontology, it is often very useful and also makes the ontology more comprehensible to provide auxiliary axioms. An $\mathcal{EL}^+$ ontology is effectively a text file containing axioms of the forms shown in the right column of Table 1.

For example, Figure 1 shows a medical ontology about the concept Endocarditis (an inflammation of a heart valve) formulated in CEL syntax. The user can either load this ontology into the system by calling `(load-ontology "med.tbox")` or enter interactively at the prompt each axiom of the ontology. The preprocess is carried out while the ontology is being loaded, and once this is finished, `(classify-ontology)` can be invoked

[2]Considerably more memory may be needed for larger ontologies.

```
(define-primitive-role contained-in)
(define-primitive-role part-of :parent contained-in :transitive t)
(define-primitive-role has-location :right-identity contained-in)

(define-primitive-concept Endocardium
                          (and Tissue
                               (some contained-in HeartWall)
                               (some contained-in HeartValve)))
(define-primitive-concept HeartWall
                          (and BodyWall
                               (some part-of Heart)))
(define-primitive-concept HeartValve
                          (and BodyValve
                               (some part-of Heart)))
(define-primitive-concept Endocarditis
                          (and Inflammation
                               (some has-location Endocardium)))
(define-primitive-concept Inflammation
                          (and Disease
                               (some acts-on Tissue)))

(define-concept Heartdisease
                (and Disease
                     (some has-location Heart)))

(implies (and Heartdisease
              (some has-location HeartValve))
         CriticalHeartDisease)
```

Figure 1: An $\mathcal{EL}^+$ ontology from file `med.tbox`.

to classify all concept names occurring in the ontology (eager subsumption approach). Subsumption query between two concept names can be queried using `(subsumes? B A)`. If this is called after classification, it simply looks up in the computed subsumption hierarchy. Otherwise, it runs a single subsumption test and answers without needing to classify the whole ontology first (lazy subsumption approach). After having classified the whole ontology, CEL allows the user to output the classification results in different formats: `(output-supers)` to output the subsumer sets for all concept names occurring in the ontology; `(output-taxonomy)` to output the Hasse diagram of the subsumption hierarchy, i.e., just the *direct* parent-child relationships; and `(output-hierarchy)` to output the hierarchy as a graphical indented tree. As an example, Figure 2 depicts screen shots of the results of `(output-hierarchy)` and `(output-taxonomy)` after classifying the ontology `med.tbox`.

Through its command-line options, CEL can also work as a stand-alone reasoner without interaction from users. For instance, the command line:

```
$cel -l filename -c -outputHierarchy -q
```

can be entered to load and classify an ontology from *`filename`*, and then output the hierarchy. For a more detailed description of the CEL interface, we refer to the CEL user

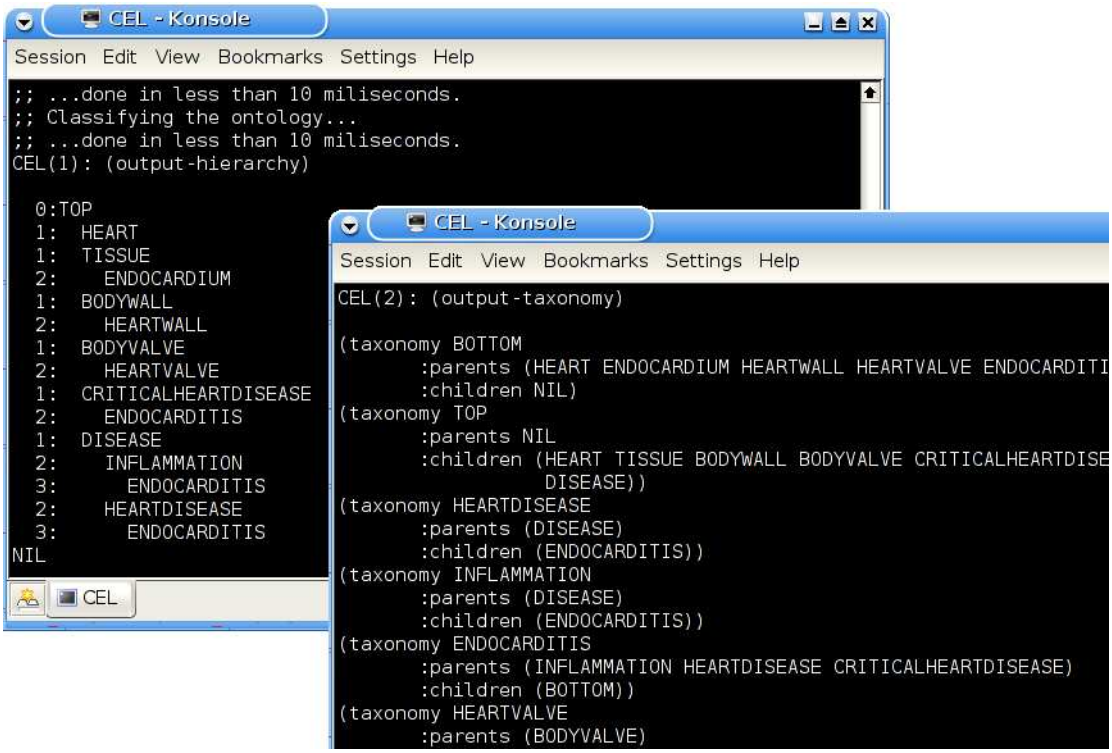

Figure 2: CEL with its innate interactive interface

manual (available on the CEL homepage).

CEL as a backend reasoner. Alternatively, the user can also exploit CEL reasoning capabilities through the DIG interface[3] and a graphical ontology editor. To do this, CEL has to be started as a DIG reasoning server by the following command line:

```
$cel -digServer [port]
```

where *port* is defaulted to 8080 but can be overridden.

Once started in this mode, ontology editors can be connected to CEL and exploit its reasoning power either locally or remotely via the Internet. The upper floating dialog in Figure 3, "Reasoner Inspector," displays the expressive means that can be handled by CEL in terms of DIG language. The DIG interface for CEL has been tested successfully with Protégé OWL editor.[4] The main window in Figure 3 illustrates the asserted subsumption hierarchy (input) and the inferred subsumption hierarchy (output) within the editor, whereas the small floating dialog, "Connected to CEL 0.9," displays the interactions between the DIG client and the DIG server.

[3]The DIG (DL Implementation Group) interface is an XML-based standard that defines an interfacing language for seamless communication between a DL service provider (DIG server) and a DL application (DIG client). See `http://dl.kr.org/dig/`

[4]See `http://protege.stanford.edu/plugins/owl/`

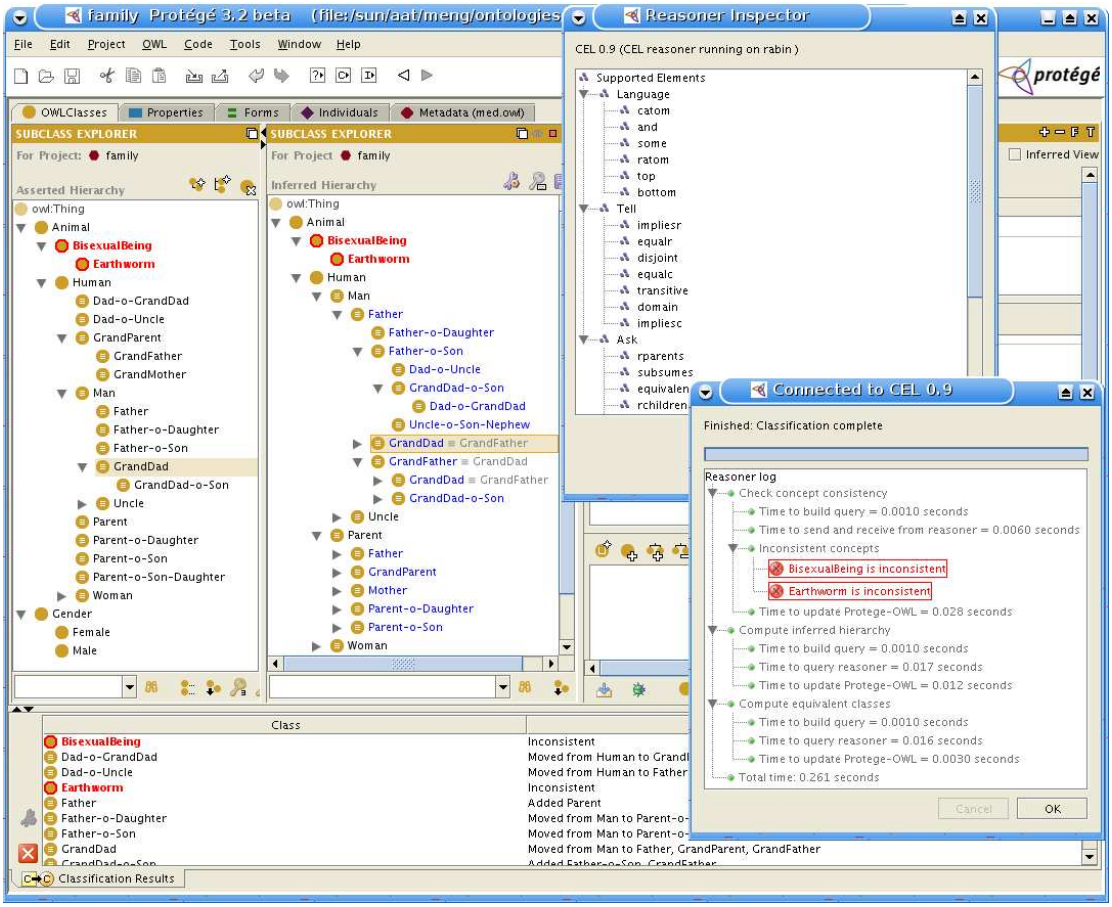

Figure 3: CEL as a DIG reasoner and Protégé OWL editor

Empirical Success of CEL.

We have compared the performance of CEL with three of the most advanced DL systems: FaCT^{++} (v1.1.0), RacerMaster (v1.9.0), and Pellet (v1.3b). These systems implement tableau-based decision procedures for expressive DLs in which subsumption is ExpTime-complete. All experiments have been performed on a PC with 2.8GHz Intel Pentium 4 processor and 512MB memory running Linux v2.6.14. For Pellet, we used JVM v1.5 and set the Java heap space to 256MB (as recommended by the implementers).

Our experiments are based on three important bio-medical ontologies: Go, Galen, and Snomed. Since Galen uses some expressivity that CEL cannot handle, we have simplified it by removing inverse role axioms and treating functional roles as ordinary ones, and obtained an $\mathcal{EL}^+$ ontology $\mathcal{O}^{\text{Galen}}$. (Of course, also the other reasoners, which could have handled inverse and functional roles, were applied to $\mathcal{O}^{\text{Galen}}$ rather than full Galen.) We have obtained two other benchmarks, $\mathcal{O}^{\text{Go}}$ and $\mathcal{O}^{\text{Snomed}}$, from the other two ontologies. However, Snomed has two right-identity rules similar to the third role axiom in our example (see Fig. 1). These axioms are passed to CEL, but not to the other reasoners, as the latter do not support right identities. Additionally, to get a smaller version of Snomed that can be dealt with by standard DL reasoners,

	$\mathcal{O}^{\text{GO}}$	$\mathcal{O}^{\text{GALEN}}$	$\mathcal{O}^{\text{SNOMED}}_{\mathsf{core}}$	$\mathcal{O}^{\text{SNOMED}}$
concept axioms	20,465/0/0	2,041/699/1,214	0/38,719/0	340,972/38,719/0
role axioms	1	438	0	11 + 2
concept names	20,465	2,740	53,234	379,691
role names	1	413	52	52
CEL	5.8	14	95	1,782
FaCT^{++}	6.9	50	740	3,859
RacerMaster	19	14	34,709	*unattainable*
Pellet	1,357	75	*unattainable*	*unattainable*

Table 3: Benchmarks and Evaluation Results

we also consider a fragment obtained by keeping only concept definitions, and call it $\mathcal{O}^{\text{SNOMED}}_{\mathsf{core}}$. Some information on the size and structure of these benchmarks is given in the upper part of Table 3, where the first row shows the numbers of primitive concept definitions, concept definitions, and general concept inclusions, respectively. The results of our experiments are summarized in the lower part of Table 3, where all classification times are shown in seconds and *unattainable* means that the reasoner failed due to memory exhaustion. Notable, CEL outperforms all the reasoners in all benchmarks except $\mathcal{O}^{\text{GALEN}}$, where RacerMaster is as fast. CEL and FaCT^{++} are the only reasoners that can classify $\mathcal{O}^{\text{SNOMED}}$, whereas RacerMaster and Pellet fail. Pellet and the original version of FaCT (not shown in the table) even fail to classify $\mathcal{O}^{\text{SNOMED}}_{\mathsf{core}}$.

The empirical results for the performance of CEL described above show that it can compete with, and often outperforms, the fastest tableau-based DL systems. The robustness and scalability of tractable reasoning is visible, especially in the case of SNOMED with almost four hundred thousand concepts. We view these results as evidence of empirical success of the CEL system, and also as a strong argument for the use of tractable DLs based on extensions of $\mathcal{EL}$ provided that the expresivity is sufficient in the domain of interest.

References

[1] F. Baader, S. Brandt, and C. Lutz. Pushing the $\mathcal{EL}$ envelope. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, Edinburgh, UK, 2005. Morgan-Kaufmann Publishers.

[2] F. Baader, C. Lutz, and B. Suntisrivaraporn. Efficient reasoning in $\mathcal{EL}^+$. In *Proceedings of the 2006 International Workshop on Description Logics (DL2006)*, CEUR-WS, 2006.

[3] R. Cote, D. Rothwell, J Palotay, R. Beckett, and L. Brochu. The systematized nomenclature of human and veterinary medicine. Technical report, SNOMED International, Northfield, IL: College of American Pathologists, 1993.

[4] P. Patel-Schneider and B. Swartout. Description-logic knowledge representation system specification from the krss group of the arpa knowledge sharing effort. Technical report, DARPA Knowledge Representation System Specification (KRSS) Group of the Knowledge Sharing Initiative, 1993.

[5] A. Rector and I. Horrocks. Experience building a large, re-usable medical ontology using a description logic with transitivity and concept inclusions. In *Proceedings of the Workshop on Ontological Engineering, AAAI Spring Symposium (AAAI'97)*, Stanford, CA, 1997. AAAI Press.

[6] K.A. Spackman. Managing clinical terminology hierarchies using algorithmic calculation of subsumption: Experience with SNOMED-RT. *Journal of the American Medical Informatics Association (JAMIA)*, 2000. Fall Symposium Special Issue.

[7] The Gene Ontology Consortium. Gene Ontology: Tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.