Theorem Proving in Large Theories*

Wolfgang Reif and Gerhard Schellhorn

Abt. Programmiermethodik, Universität Ulm, D-89069 Ulm, Germany

Abstract. This paper investigates the performance of automated theorem provers in formal software verification. The challenge for the provers in this application is the large number of (up to several hundred) axioms in typical software specifications. Both the success rates and the proof times strongly depend on how good the provers are able to find out the few relevant axioms that are really needed in the proofs. We present a reduction technique for this problem. It takes the axioms of a theory and a theorem and computes a reduced axiom set by eliminating as many irrelevant axioms as possible. The proof search for the theorem then is performed in the reduced set. Comparative experiments with five automated theorem provers show that with the reduction technique they can prove more theorems than before, and were faster for those that could be proved already without reduction.

1 Introduction

The motivation behind this paper is the question: how can formal software verification benefit from automated first-order theorem proving? To answer the question we used the software verification tool, KIV ([RSS95], [Rei95], [FRSS95]) as a test environment, and did comparative experiments with five automated theorem provers as dedicated subsystems for the non-inductive first-order goals that showed up during proofs of specification- and program properties. The five provers were Protein([BF94]), Otter ([WOLB92]), Setheo ([GLMS94]), $_3T^AP$ ([BHOS96]) and Spass ([WGR96]). The experiments were performed in the context of a joint project[2] on the conceptual integration of interactive and automated theorem proving using KIV and $_3T^AP$ as an experimental platform. Therefore, $_3T^AP$ could be called from inside KIV, whereas for the other provers the problems had to be transfered by hand.

As a general result in the experiments we found that the automated provers did not satisfy the expectations induced by the good results some of them normally get in standard benchmarks from the TPTP ([GSY94]) library. In our experiments either the success rate was rather low or the required time was unexpectedly high.

* This research was sponsored by the German Research Foundation (DFG) under grant Re 828/2-2.

[2] between the research groups of $_3T^AP$ and KIV at the Universities of Karlsruhe and Ulm

One major reason for this behaviour is the large number of axioms in typical software specifications. Whereas TPTP problems only have a few axioms, typical software specifications in KIV comprise up to several hundreds of axioms. The provers were misled or even got lost in the search space although the majority of the axioms were irrelevant to the proofs under consideration. We found that in the context of large software specifications it is important to find out the relevant axiom set (for the proof of a theorem) or at least a good approximation to it.

In this paper we present a reduction technique that takes a large theory and a goal, and computes a reduced axiom set by filtering out as many irrelevant axioms as possible. The proof search then is performed with the reduced axiom set. The reduction is independent from the actual prover and the calculus. Furthermore, it can be applied recursively during a proof to the subgoals and sublemmas of the original theorem. They often have progressively smaller sets of relevant axioms than the theorem itself.

To evaluate the reduction technique we repeated the original experiments once more, but now with the reduced axiom sets. In the largest example, the original theory had 500 axioms. The reduced axiom sets for the test examples had around 20 axioms.

With the reduction the provers were able to prove more theorems than before. Furthermore, for those theorems that could already be proved without the axiom reduction we got considerably shorter proof times.

As a side effect it turned out that the susceptibility of the five provers to the problem of large axiom sets is quite different. Furthermore, the experiments shed some light on the question what makes one large theory more harmful than another.

In the next section we illustrate the problem with an example. In section 3 we present the reduction criteria and the assumptions about the specification structure. The reduction procedure is explained in section 4. Section 5 reports on the experimental results, and section 6 draws some conclusions.

2 An Example

The example is a specification of a single datatype and not of a whole software system. But it is sufficient to demonstrate the basic reduction criteria, and large enough to cause some problems for the automated theorem provers.

It deals with the data type of finite enumerations. These are bijections from a finite subset of data elements to an initial segment of the natural numbers. An example for an enumeration is the function which maps all customers queueing for bread in a bakery to their position in the queue. Further examples are mappings which associate unique keys to database entries, or enumerate the nodes in a graph. Actually, the specification of enumerations was part of a larger KIV case study on the verification of depth-first search on graphs.

Enumerations can be constructed by $\emptyset$ (the empty bijection), and by $\mathrm{en} \oplus \mathrm{d}$ adding a data element d (with fresh code number) to the enumeration en. Adding

an element twice has no effect. The size function $\#$ en returns the number of elements recorded in the enumeration, and the predicate $d \in en$ tests for membership. The selector num_of(d, en) returns the number of the element d in en, and el_of(n, en) gives back the element number n in en. Both operations are unspecified if $d \notin en$ or if $n \geq \#$ en, respectively. Finally, the operation $en \ominus d$ removes the element d and its code number from the enumeration (if there is such an element). In addition all code numbers greater than num_of(d, en) are decremented by one in order to avoid gaps in the range of $en \ominus d$.

Back to the bakery: In the early morning the situation in the bakery is reflected by $\emptyset$. Then three customers arrive leading to the enumeration en = $((\emptyset \oplus \text{john}) \oplus \text{mary}) \oplus \text{peter}$, with num_of(john, en) = 0, num_of(mary, en) = 1, and num_of(peter, en) = 2. We have el_of(1, en) = mary and $\#$ en = 3. When mary leaves the bakery, we get $en_1 = en \ominus \text{mary}$, with num_of(john, en_1) = 0, and num_of(peter, en_1) = 1.

Enumerations can be implemented in various ways, depending on what operations are critical for efficiency. Possible implementations are duplicate free lists, arrays, or hashtables (to speed up num_of).

2.1 The formal specification

With a little experience the above description of finite enumerations can be directly translated into a bunch of axioms. However, from the software engineering point of view an amorphous list of formulas is a bad representation. In the software design process, the specification is preceded by a careful problem analysis identifying decomposable subproblems and their interrelations. With the standard algebraic specification language used in KIV, this information can be made explicit in the specification structure. This is not only good software engineering practice but also very helpful for automated deduction.

The formal specification of finite enumerations is shown in Fig. 1. It identifies seven specification modules. *DelEnum* is the toplevel specification describing the overall functionality of finite enumerations over the parameter *Elem*. It imports the subspecification *Enum* and enriches it by the axioms for $\ominus$ (called the Δ of the enrichment). *Enum* specifies the remaining operations for finite enumerations. The constraint "enum **generated by** $\emptyset$, $\oplus$" gives a structural induction principle.

The specification is formulated relative to a standard specification of natural numbers (actually taken from the KIV library): *NatBasic* is the freely generated fragment of the natural numbers with 0 and successor succ, with additional predecessor function pred and the ordering $<$. It is written like an ML ([MTH89]) datatype declaration for which the axioms can be generated automatically. The enrichments *Add* and *Sub* introduce addition and subtraction by recursive definitions. *Nat* is the union of *Add* and *Sub*.

The specification *DelEnum* (inclusive of all subspecifications) has 17 axioms that are directly given in Fig. 1. 8 axioms are generated automatically for the data specification *NatBasic*. Furthermore, the specification *Nat* from the library is associated with 77 additional standard lemmas to improve arithmetical reasoning. These are persistent over the lifetime of the specification, and have been

proved long time ago once and for all. Generally, the reuse of a formal library specification may include axioms that are relevant to one application but irrelevant to another. For *Enum* we get altogether 102 axioms.

As test examples we selected the 52 proof obligations for *DelEnum*, that showed up during a KIV case study on depth-first search in graphs. In a number of large case studies we found that in order to prove theorem n, it is a good idea to add all the n-1 previously proved theorems as lemmas to the theory. Although this enlarges the theory again, the effect is positive: With the redundant 77 lemmas of *Nat* and the discipline to add all previously proved test examples to the theory, the success rate of the four provers was doubled.

2.2 Proving a goal

Now we want to prove the theorem

$$\text{th-45: } \#\ \text{en} = 0 + 1 \rightarrow \emptyset \oplus \text{el_of}(0, \text{en}) = \text{en}$$

from our example theory. In the test suite, th-45 was the 45th theorem. Therefore we had at this stage 146 (= 102 + 44) axioms in the theory. Potentially all of them could be used in the proof. Actually an interactive proof in KIV used only 12 of them (6 of the previously proved theorems, 4 axioms from the *Enum*-specification and two axioms from *NatBasic*).

Of course, it is impossible by syntactic considerations to find out *exactly* which axioms will be relevant for a proof. But already the structure of the specification allows us to reduce the set of relevant axioms and lemmas considerably, as we will now show informally for the example. A precise definition of the reduction criteria is given in sect. 3.

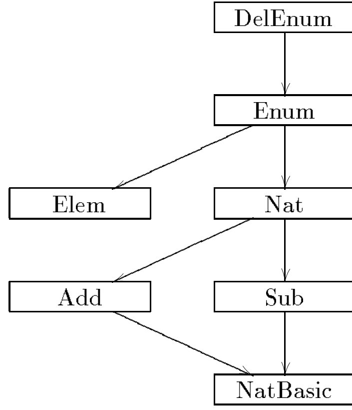
First, we select the minimal subspecification, whose signature comprises that of the theorem. In our case this is the specification *Enum*, since the operation $\ominus$ defined in *DelEnum* does not occur in the theorem. Thereby 5 axioms and 17 lemmas can be removed from the theory.

Second, the arithmetic operations + and − are not required. They are neither used to define any operation in the *Enum* specification nor do they occur in the theorem (this does not mean that *Add* and *Sub* are superfluous. They could be used for example in a superspecification of *DelEnum* in a larger case study). Removing all axioms and theorems for + and − saves additional 4 axioms and 57 lemmas, which no longer must be passed to the automated theorem prover.

Third, with the removal of $\ominus$ we have eliminated all axioms, which make use of the predecessor function. Since the theorem does not use it, and *NatBasic* could be split into a specification, which defines 0,succ and an enrichment for pred, we can also remove all axioms and lemmas which use the predecessor function (this is not a special case, it is true for all destructors of datatype specifications). Again this removes 2 axioms and 12 lemmas from the theory.

Finally, with a similar argument as for the predecessor function, the 4 axioms and 6 lemmas for the $<$ predicate can be dropped.

After four reductions we are left with 38 axioms (10 axioms from the specification, 28 additional lemmas), which may be relevant to the proof of the theorem.




```
NatBasic =
data specification
    nat =  0 | succ (pred : nat);
    variables m, n: nat;
    order predicates
        . < . : nat × nat;
end data specification
```

```
Add =
enrich Nat with
    functions
        . + . : nat × nat → nat ;
    axioms
        n + 0 = n,
        m + succ(n) = succ(m + n)
end enrich
```

```
Sub =
enrich NatBasic with
    functions
        . − . : nat × nat → nat;
    axioms
        m − 0 = m,
        m − succ(n) = pred(m − n)
end enrich
```

```
Nat =
Add + Sub
```

```
Elem =
specification
    sorts elem;
    variables d: elem;
end specification
```

```
Enum =
generic specification
    parameter Elem
    using Nat target
    sorts enum;
    constants ∅ : enum;
    functions
        . ⊕ .    : enum × elem → enum ;
        num_of : elem × enum → nat   ;
        el_of    : nat × enum  → elem  ;
        # .      : enum          → nat   ;
    predicates . ∈ . : elem × enum;
    variables en: enum;
    axioms
      enum generated by ∅, ⊕;
      d ∈ en → en ⊕ d = en,
      ¬ d ∈ ∅,
      d1 ∈ en ⊕ d ↔ d = d1 ∨ d1 ∈ en,
      # ∅ = 0,
      ¬ d ∈ en → #(en ⊕ d) = succ(# en),
      ¬ d ∈ en → num_of(d, en ⊕ d) = # en,
      d ≠ d1 → num_of(d, en ⊕ d1) = num_of(d, en),
      d ∈ en → el_of(num_of(d, en), en) = d
end generic specification
```

```
DelEnum =
enrich Enum with
    functions
        . ⊖ . : enum × elem → enum;
    axioms
      ¬ d ∈ en → en ⊖ d = en,
      d ∈ en → #(en ⊖ d) = pred(# en),
      d1 ∈ en ⊖ d ↔ d ≠ d1 ∧ d1 ∈ en,
            d ∈ en ∧ d1 ∈ en
        ∧ num_of(d, en) < num_of(d1, en)
      → num_of(d1, en ⊖ d) = pred(num_of(d1, en)),
            d ∈ en ∧ d1 ∈ en
        ∧ num_of(d1, en) < num_of(d, en)
      → num_of(d1, en ⊖ d) = num_of(d1, en),
end enrich
```

Fig. 1. The Example Specification

The set of relevant axioms was reduced by a factor of almost 4. Although, this is not optimal (only twelve of them are actually needed) it makes a big difference for automated theorem provers. E.g. Otter was not able to prove the theorem with the full set of axioms within 5 minutes. With the reduced set of axioms the time to prove the theorem was about 13 seconds (a summary of the results for all provers will be given in section 5).

3 Reduction Criteria

Specifications in KIV are built up from elementary first-order theories with the usual operations known in algebraic specification: union, enrichment, parameterization, actualization and renaming. Their semantics is the class of all models (loose semantics). Reachability constraints like "nat generated by 0, succ" or "list generated by nil, cons" restrict the semantics to term-generated models. The constraints are reflected by induction principles in the calculus for theorem proving used in KIV. We treat them as (higher-order) axioms. The structure of a specification is visualized as a specification graph, like the one we have seen in fig. 1.

Structuring operations are not used arbitrarily in formal specifications of software systems. Enrichments "ESPEC = **enrich** SPEC **by** Δ", where Δ consists of a signature sig(Δ) and axioms ax(Δ) to be added, are supposed to have the property of *hierarchy persistency*. This property says that every model of SPEC can be extended to a model of ESPEC (therefore such enrichments are sometimes also called model conservative).

Hierarchy persistency of an enrichment implies safe reduction of the set of necessary axioms to prove a theorem φ: Every theorem φ that holds in (all models of) ESPEC and uses only the signature defined in SPEC, already holds in SPEC. Without loss of generality we can also assume sig(Δ) $\neq \emptyset$ in the following, since otherwise the axioms in ax(Δ) would already be theorems over SPEC.

Hierarchy persistency cannot be checked syntactically, but there are sufficient criteria: e.g. definitional extensions (see e.g. [Far94]) and recursive definitions over free datatypes are easily proved to be hierarchy persistent. More generally, if the functions and predicates defined in the Δ of an enrichment **enrich** SPEC **with** Δ can be implemented by abstract programs, which use only the operations defined in SPEC, terminate on any input and satisfy the given axioms, the enrichment is hierarchy persistent. E.g. function $\ominus$ defined in *DelEnum* could be implemented by a program which given an enumeration en and an element d to delete, would construct the result by inserting (using a loop from 0 to # en) all elements of en except d in a new empty enumeration.

The criterion of hierarchy persistency for enrichments can be used to derive criteria for the safe reduction of axioms for all other types of specifications: A generic specification like *Enum* must be a hierarchy persistent enrichment of the parameter. Nothing is required for renamings, and actualized specifications must be removed from the specification graph by actually computing the instantiated specification (this may duplicate some specification structure).

For a union specification $\text{SPEC}_1 + \text{SPEC}_2$ the criterion required is that the operations common to SPEC_1 and SPEC_2 are defined in a set of subspecifications

SP_1, ... SP_n of both SPEC_1 and SPEC_2, which is shared in the specification graph and that both the enrichments of $SP_1 + \ldots + \mathrm{SP}_n$ to SPEC_1 and to SPEC_2 are hierarchy persistent. In this case the union is said to *have explicit sharing*, and the proof of a theorem φ with $\mathrm{sig}(\varphi) \subseteq \mathrm{sig}(\mathrm{SPEC}_i)$ requires only the axioms of SPEC_i (i=1,2).

Structured Specifications in which every enrichment is hierarchy persistent and every union has explicit sharing are called *modular*. An example is the specification from the last section given in fig. 1. All enrichments are hierarchy persistent, and the union specification *Nat* has explicit sharing: The common part of *Add* and *Sub* is the specification *NatBasic* shared by both specifications.

Modular specifications are the usual case in software development[3]. Often even the structure of an implementation by a system of software modules (for a formal definition of such implementations, as they are done in KIV, see [Rei95]) follows the structure of the specification. The specification is then called an *architectural specification* ([Mos96]).

For a modular specification we can now prove the following criteria for the reduction of axioms:

- *minimality criterion*: To prove a theorem one never needs more axioms than those of the minimal subspecification MSPEC whose signature covers the signature of the theorem. This criterion was applied to the example theorem in section 2 to eliminate the axioms of *DelEnum*. In practice, this criterion usually does not lead to much reduction, since usually theorems are formulated over the minimal subspecification.
- *structure criterion*: If the enrichment "'**enrich** SPEC **by** Δ"' is a subspecification of the minimal specification MSPEC, such that the operations from Δ are neither used in specifications above the enrichment nor in the theorem, the axioms from Δ can be dropped. This criterion was applied to eliminate the axioms for $+$ and $-$.
- *specification criterion*: For elementary specifications the criteria described above can also used to determine, if an operation defined in it is a hierarchy persistent enrichment of the rest. This criterion was applied to split the datatype specification *NatBasic* into a basic specification *NatB* and enrichments for the predecessor function and the less predicate.
- *recursion criterion*: The three previous criteria can be applied recursively. This criterion was used to eliminate the axioms for the predecessor function.

By applying the four criteria on a modular specification, a theory can be determined, over which the theorem can be *guaranteed* to hold, if it holds at all. Even if there are some non hierarchy persistent enrichments in a structured specification, the axiom reduction is still a very good heuristic.

For actually proving theorems, we use the automated theorem provers as "oracles", which relieve us, if successful, of some interaction in KIV. The approach is correct, but not complete, since we are dealing with inductive theories and reachability constraints such as "Nat generated by 0,succ" are not available to

[3] The situation is different in mathematics, consider e.g. the enrichment of rings to fields, which is not hierarchy persistent

the automated theorem provers. In rare cases, an inductively provable theorem may even require to be generalized to a theorem, which requires a larger theory (the only theorem in our example is th-19, which must be generalized to th-03, which requires additionally the $<$-predicate).

4 The Reduction Algorithm

With the reduction criteria from the previous section, a set of relevant axioms RelAx(SPEC,φ) for a structured specification SPEC and a theorem φ can be computed. Of course, "relevant" axioms are a superset of those actually needed.

In practical applications, there will be a lot of theorems φ_1, φ_2, ... which have to be proved over the same specification SPEC. Therefore, it is a bad idea to start the computation of relevant axioms and lemmas for every theorem anew.

Instead, the algorithm we present in the following splits computation of RelAx(SPEC,φ) in two phases: In the first phase, specification structure is compiled to information Comp(SPEC) = $\{$Info(Ax): Ax $\in$ ax(SPEC)$\}$ for every axiom of SPEC. This phase is independent of the theorem to prove, and can be computed when the specification and its structure are known.

In the second phase this information is evaluated to give the set of relevant axioms as Eval(Info,φ) for a theorem φ to prove. We will see that Comp does the main computation, Eval is a simple iteration function over all axioms.

To define the algorithm let us first state two theorems about modular specifications:

Theorem 1. *A modular specification* **enrich** *(***enrich** *SPEC* **with** Δ_1*)* **with** Δ_2 *is semantically equivalent to* **enrich** *(***enrich** *SPEC* **with** Δ_2*)* **with** Δ_1 *and the latter is also a modular specification, if the operations defined in* Δ_1 *are not used in the axioms of* Δ_2*.*

Theorem 2. *A modular specification (***enrich** *(SPEC*$_1$ *+ SPEC*$_2$*)* **with** Δ*) is semantically equivalent to (***enrich** *SPEC*$_1$ **with** Δ*) + SPEC*$_2$ *and the latter is also a modular specification, if the operations defined in* Δ *are not used in the axioms of SPEC*$_2$*.*

These two theorems allow to view the removal of irrelevant axioms as a process of "lifting irrelevant Δs up in the specification hierarchy", until they are outside the minimal subspecification relevant to the theorem. E.g. in our example two applications of the second theorem allow us to restructure **enrich** *Elem* + *Add* + *Sub* **with** Δ_{Enum} to (**enrich** *Elem* **with** Δ_{Enum}) + *Add* + *Sub* (Δ_{Enum} is the enrichment of *Enum*).

Examining the consequences of the two theorems, we find, that the original structure of a modular specification is completely irrelevant. If we view basic specifications as enrichments of an empty specification, all that matters are the Δs of its enrichments. These can be put together arbitrarily without changing the semantics as long as the resulting specification is syntactically valid (the proof is by induction on the number of Δs). This fact can be exploited to define a certain "normal form" of modular specifications, in which all irrelevant parts of subspecifications have been "lifted up".

To define this normal form, we first compute for an enrichment **enrich** SPEC **with** Δ the minimal necessary set sub(Δ) of Δ's, which must be contained in SPEC, such that the enrichment is syntactically valid. sub(Δ) can be computed by defining (read: Δ' is directly necessary for Δ):

$$\Delta' <^1 \Delta :\Leftrightarrow \mathrm{sig}(\Delta') \cap \mathrm{sig}(\mathrm{ax}(\Delta)) \neq \emptyset \text{ and } \Delta' \neq \Delta$$

If $<$ is the transitive closure of $<^1$ we have

$$\mathrm{sub}(\Delta) = \{\Delta' \colon \Delta' < \Delta\}$$

Setting max(sub(Δ)) to be the set of maximal elements in sub(Δ) with respect to $<$, we can now define the normal form, which is semantically equivalent to the original specification:

Definition 3. Let SPEC be a modular specification. Then for every Δ in SPEC the normal form of SPEC contains an enrichment **enrich** $\mathrm{SPEC}_1 + \ldots + \mathrm{SPEC}_n$ **with** Δ, such that $\mathrm{SPEC}_1, \ldots, \mathrm{SPEC}_n$ are the enrichments for the elements in max(sub(Δ)). Additionally the normal form contains a toplevel union specification for all maximal Δs from SPEC with respect to $<$.

Only the maximal elements in sub(Δ) are needed in $\mathrm{SPEC}_1, \ldots, \mathrm{SPEC}_n$, since the others are already contained in subspecifications of these specifications. The following theorem holds:

Theorem 4. *The normal form of a specification SPEC is unique up to reordering of arguments in the union specifications. It is semantically equivalent to SPEC.*

For the example specification from section 2 the specification graph of the normal form is shown in Fig. 2 (ignore specification *Enum + Less* for the moment). Before computing the normal form *NatBasic* has already been split into a basic specification *NatB* and two enrichments with less predicate (specification *Less*) and predecessor function (*Pred*) according to the specification criterion of section 3.

The normal form is used to determine the set of relevant axioms as follows:

Theorem 5. *Let SPEC be a modular specification and φ a formula with signature sig(φ). Set $\mathcal{S}_0(sig(\varphi)) := \{\Delta : sig(\Delta) \cap sig(\varphi) \neq \emptyset\}$ (the set of all Δ's which define some signature used in φ) and $\mathcal{S}(sig(\varphi)) := sub(\mathcal{S}_0(sig(\varphi))) \cup \mathcal{S}_0(sig(\varphi))$ (their subspecifications in the normal form of SPEC). Then φ holds over SPEC, if it already holds in the union of all enrichments with Δs in $\mathcal{S}(sig(\varphi))$ in the normal form of SPEC. This specification is minimal (i.e. there are theorems which will need the axioms of every Δ).*

Note that $\mathcal{S}(\mathrm{sig}(\varphi))$ is just the minimal set of Δs whose signatures include sig(φ) and which can be put together to a specification.

As an application consider the theorem

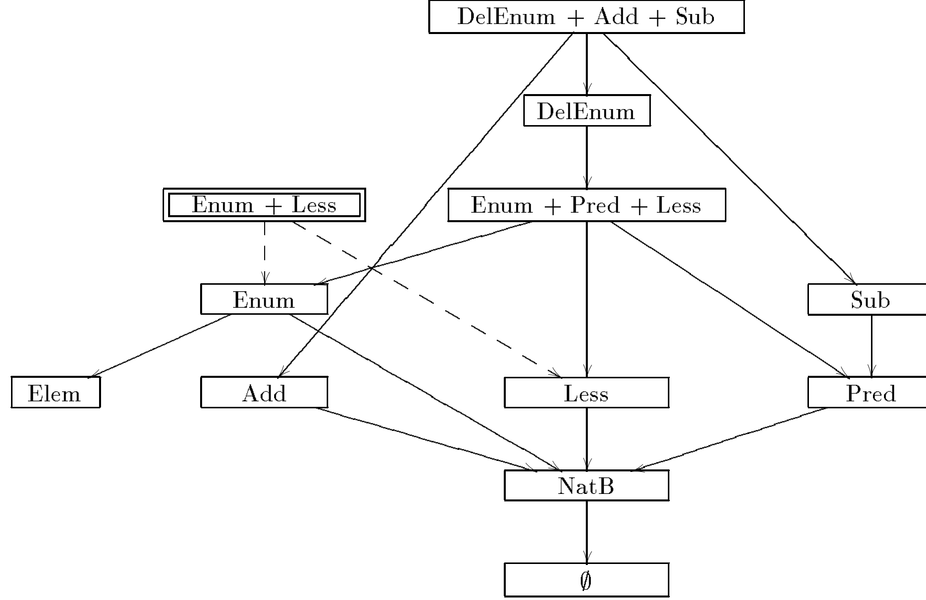


Fig. 2. Specification in normal form

th-19: n < # en → (num_of(el_of(n,en), en) = n

which is the dual theorem to the last axiom

ax-08: d ∈ en → el_of(num_of(d, en), en) = d

from specification *Enum*. It contains symbols from Δ_{Enum} (the Δ of the enrichment *Enum*) and Δ_{Less}, so $\mathcal{S}_0$(sig(th-19)) = $\{\Delta_{Enum}, \Delta_{Less}\}$. The only specification below *Enum* and *Less* in Fig. 2 is *NatB*, so $\mathcal{S}$(sig(th-19)) = $\{\Delta_{Enum}, \Delta_{Less}, \Delta_{NatB}\}$. According to theorem 5, φ is therefore provable over the sum of *Enum*, *Less* and *NatB*. Of course we can remove non maximal elements from this sum, so φ is provable in the union specification *Enum + Less*, which is added in Fig. 2 with a double frame. For our example theorem th-45 from section 2, things are easier. We have $\mathcal{S}_0$(sig(th-45)) = $\{\Delta_{Enum}\}$ and $\mathcal{S}$(sig(th-45)) = $\{\Delta_{Enum},\Delta_{NatB}\}$, so th-45 is provable with the axioms of those two Δs, as was stated informally in section 2. Note that for a theorem φ the sets $\mathcal{S}_0(\text{sig}(\varphi))$ and $\mathcal{S}(\text{sig}(\varphi))$ can be computed as the union of sets $\mathcal{S}(\{\text{f}\})$, for every symbol f from sig(φ).

With the computation of the minimal specification, in which a theorem φ should be proved, we have answered the question, which *axioms* may be relevant for the proof a theorem. For a previously proved *lemma*, the situation is slightly different, since a lemma is not necessarily connected to *one* of the Δs. E.g. th-19 is proved over the union of *Enum* and *Less*. But in fact, the answer is now easy

to find: Lemma ψ may be relevant for a proof of φ, if and only if it holds over the axioms of $\mathcal{S}(\text{sig}(\varphi))$ (consequences of irrelevant axioms are also irrelevant). Since it holds over the axioms in $\mathcal{S}(\text{sig}(\psi))$, we must test, whether $\mathcal{S}(\text{sig}(\psi)) \subseteq \mathcal{S}(\text{sig}(\varphi))$. This test can also be used for axioms. To gain efficiency, it suffices to use the maximal elements from $\mathcal{S}(\text{sig}(\psi))$ on the left side in the subset-test. For an axiom ψ defined in a Δ, $\max(\mathcal{S}(\text{sig}(\psi)))$ usually just contains this Δ (otherwise, if the axiom does not use any symbols from $\text{sig}(\Delta)$, it is redundant).

Now we are ready to define the two phases of an algorithm, which computes the relevant axioms and lemmas for the proof of a theorem, as we announced in the beginning of this section:

In a first phase, which is done as soon as the specification and its structure become known, function Comp computes

- all Δs from the specification structure
- the $<^1$-relation on specifications
- its transitive closure $<$
- $\mathcal{S}(\{\text{f}\})$ for every symbol (sort,function and predicate) of the specification
- $\text{Info}(\psi) := \max(\mathcal{S}(\text{sig}(\psi)))$ for every axiom ψ

$\text{Info}(\psi)$ is attached to every axiom ψ.

In the second phase, on an attempt to prove theorem φ, function Eval first computes $\mathcal{S}(\text{sig}(\varphi))$ (using the precomputed information on $\mathcal{S}(\{\text{f}\})$ for every symbol f). The relevant axioms then are

$$\text{Relax}(\varphi) := \{\ \psi\text{: Info}(\psi) \subseteq \mathcal{S}(\text{sig}(\varphi))\ \}$$

After a successful proof of theorem φ, we can add it to the now available axioms, and attach $\text{Info}(\varphi) := \max(\mathcal{S}(\text{sig}(\varphi)))$.

This algorithm can still be improved by two further observations:

- An inference step between an axiom ψ and a theorem φ will require, that $\text{sig}(\psi) \cap \text{sig}(\varphi) \neq \emptyset$ (at least in all calculi for theorem proving, we know).
- A theorem φ is always formulated over the union of the enrichments in $\max(\mathcal{S}(\text{sig}(\varphi))$. Its proof will almost always make use of axioms of these Δs, which are the uppermost in the specification graph in normal form (otherwise parts of the theorem could be dropped).

The idea is to add the axioms incrementally, with topmost axioms in the hierarchy specifications added first, others lower down in the hierarchy added later. This is done as follows: Instead of computing the full set $\mathcal{S}(\text{sig}(\varphi_0))$ for a theorem φ_0, we just compute $\mathcal{S}_0(\text{sig}(\varphi_0))$, the enrichments which define some signature used in φ. With these Δs a first approximation

$$\text{Relax}_0(\varphi_0) := \{\ \psi : \text{sig}(\varphi_0) \cap \text{sig}(\Delta) \neq \emptyset \text{ for every } \Delta \in \text{Info}(\psi)\}$$

of the relevant axioms can be computed. Adding these axioms as preconditions to the goal to prove, i.e. setting $\varphi_1 := \text{Cl}_\forall(\text{Relax}_0(\varphi_0)) \rightarrow \varphi_0$, where $\text{Cl}_\forall$ is universal closure, we can either compute a second approximation

$$\text{Relax}_1(\varphi_1) := \{\ \psi : \text{sig}(\varphi_1) \cap \text{sig}(\Delta) \neq \emptyset \text{ for every } \Delta \in \text{Info}(\psi)\}$$

immediately (and iteration will finally yield all relevant axioms), or first try some deduction steps before considering the axioms of the next approximation. Often the theorem will already be provable with the axioms from Relax_0. In fact, for the example theorem th-19, Relax_0 will include the necessary lemma (th-15) and axiom (ax-08) for the proof, but not the axioms from *NatB*.

Iterated computation of relevant axioms is even more beneficial when a calculus is used for theorem proving that has a notion of "subgoals" (like sequent- or tableau-calculus). Then the second iteration of computing relevant axioms may use the subgoals produced by the inference steps instead of φ_1, with the expectation that even fewer axioms should be relevant for subgoals with smaller signature.

5 Experimental Results

To evaluate the results of axiom reduction we first tried the automated theorem provers Setheo, Otter, Spass and $_3T^AP$ on 52 theorems defined over the *DelEnum* specification from section 2. No theorem was invented for this case study, all were drawn from the case study on verifying depth-first search on graphs. A full listing of the theorems, settings of the provers and the results can be found appendix A.

To summarize, all provers benefited from the reduction of axioms, but there were enormous differences. Very significant improvements were made by Spass and $_3T^AP$, Otter and Setheo benefited less.

A possible explanation for the good behaviour of Otter and Setheo is, that it concentrates proof search on the distinguished goal clause. E.g. if in the proof of th-10 we intentionally take a wrong goal clause (th-6), the time Setheo needs to find a proof increases from 1.2 seconds to 110 seconds (and this result is typical).

As a second case study we considered 54 simple non inductive first-order theorems that showed up during the verification of a Prolog compiler in KIV ([SA96]). These are formulated over a specification which is built up from a lot of standard datatypes (lists, tuples, pairs etc.) in various different instantiations. Therefore the specification structure contains many different sorts, but the hierarchy of specifications is relatively flat. The theorems are easier than the ones found in *DelEnum* example, but the initial set of axioms an lemmas is much larger (ca. 500). We found that Spass and $_3T^AP$ were initially only able to prove 1 resp. 9 theorems. Setheo could prove 44, Otter 48 theorems. The situation changes, when instead of the full set only the reduced set of axioms is considered, since this set contains in most cases only between 4 (!) and 25 axioms: For Spass and $_3T^AP$ the set of provable theorems increased to 25 resp. 35. Otter and Setheo cannot prove more theorems, but the time to prove them drops in several examples from over 30 seconds to about 2.

It seems, that Otter's and Setheo's heuristics for using axioms are already strong enough, to avoid all axioms involving sorts (encoded as constants), which

do not occur in the theorem. In a flat specification structure, this set is already a good approximation to the set of relevant axioms. To see, how the heuristics of Otter and Setheo would behave in general, we finally tried an example with the opposite characteristic: Only few sorts, but many operations. The example is from the KIV library of standard specifications: There, a specification *Graph* is defined. To define the set of nodes of a graph it uses a specification *EnrSet* of nodes, which is itself an enrichment of a basic specification *Set* of finite sets (with $\emptyset$, $\in$, $\subset$, cardinality, insert and delete) by the operations $\cup$, $\cap$, and $\setminus$ (set-difference), an ordering on elements and a selector of the minimal element of a set. The theorems we tried to prove with the axioms of *Set*,*Enrset* and *Graph* are the 62 theorems from the *Set* specification, which of course can be proved with the axioms of the smallest specification *Set* only.

Indeed, Otter (Setheo) can prove 31 (29) theorems of this specification when given the 8 axioms of the *Set* specification only. With the additional 60 axioms (including lemmas) of the *Enrset* specification, the number drops to 27 (24), and adding the axioms of the *Graph* specification, the number of provable theorems drops to 24 (18).

Let us finally make a technical remark: One criterion for the effective use of automated theorem provers in the verification of first-order theorems from software verification is that they separate expensive preprocessing of axioms from the actual proof attempts. This avoids preprocessing the axioms over and over again for several theorems over the same specification. The code generation for Setheo already takes a minute for the test examples of the Prolog compiler with the full set of axioms. This is quite unsatisfactory. The same problem exists for Otter's transformation of formulas to clauses, but it is less severe since generating clauses is done in about 5 seconds. Spass already separates clause generation from actual proof attempts. Here, $_3T^AP$ offers the best solution, in that it may keep an arbitrary number of preprocessed sets of axioms, which can be referred by name.

6 Conclusion

We have investigated the performance of automated theorem provers in formal software verification. To this purpose we used the software verification tool, KIV as a test environment, and did comparative experiments with the four automated theorem provers Otter , Setheo , $_3T^AP$ and Spass . In this application the challenge for the automated provers is the large number of (up to several hundred) axioms in typical software specifications. The major problem is to find out the few axioms that are really relevant to the proof of a given theorem. We have presented a reduction technique for this problem which eliminates for a given theory and a theorem as many irrelevant axioms from the theory as possible. The technique relies on good software specification practice to reflect design decisions in the specification structure. It is safe for hierarchy persistent specifications. The property of hierarchy persistency is proved separately by KIV. Reduction can be applied recursively also to the subgoals of a goal, and it is independent from the particular prover and calculus. The incremental version

of the reduction algorithm computes the reduced axiom set stepwise, using the entire set only when needed.

From the experiments we learned that all four provers benefited from the axiom reduction. They could prove more theorems than before, and were faster for those that could be proved already without reduction. However, the susceptibility of the four provers to the problem of large axiom sets was quite different. Furthermore, the experiments shed some light on the question, to what extent an interactive verification system could benefit from an automated prover. We expect that in an integrated system 30% of the user interactions (in the context of first-order goals) can be saved.

Acknowledgements

We thank Kurt Stenzel for reading the draft and making helpful comments. Thanks also to Harald Vogt and Michael Balser for their help with the experiments.

References

[BF94] P. Baumgartner and U. Furbach. Protein: A prover with a theory extension interface. In *Proc. 12th CADE*, LNCS 804. Springer, 1994.

[BHOS96] Bernhard Beckert, Reiner Hähnle, Peter Oel, and Martin Sulzmann. The tableau-based theorem prover $_3T^AP$, version 4.0. In Michael McRobbie, editor, *Proc. 13th CADE, New Brunswick/NJ, USA*, LNCS 1104, pages 303–307. Springer, 1996.

[Far94] W. M. Farmer. A general method for safely overwriting theories in mechanized mathematics systems. Technical report, The MITRE Corporation, 1994.

[FRSS95] T. Fuchß, W. Reif, G. Schellhorn, and K. Stenzel. Three Selected Case Studies in Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*. Springer LNCS 1009, 1995.

[GLMS94] C. Goller, R. Letz, K. Mayr, and J. Schumann. Setheo v3.2: Recent developments – system abstract. In A. Bundy, editor, *12th International Conference on Automated Deduction, CADE-12*, Springer LNCS 814. Nancy, France, 1994. for the newest version of SETHEO, see the URL: http://wwwjessen.informatik.tu-muencen.de/forschung/reasoning/-setheo.html.

[GSY94] C.B. Suttner G. Sutcliffe and T. Yemenis. The tptp problem library. In A. Bundy, editor, *12th International Conference on Automated Deduction, CADE-12*, Springer LNCS 814. Nancy, France, 1994.

[Mos96] P. D. Mosses. Cofi : The common framework initiative for algebraic specification. In H. Ehrig, F. v. Henke, J. Meseguer, and M. Wirsing, editors, *Specification and Semantics*. Dagstuhl-Seminar-Report 151, 1996. Further information available at http://www.brics.dk/Projects/CoFI.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[Rei95] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report.* Springer LNCS 1009, 1995.

[RSS95] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Tenth Annual Conference on Computer Assurance*, IEEE press. NIST, Gaithersburg (MD), USA, 1995.

[SA96] G. Schellhorn and W. Ahrendt. Verification of a Prolog Compiler – First Steps with KIV. Ulmer Informatik-Berichte 96-05, Universität Ulm, Fakultät für Informatik, 1996.

[WGR96] C. Weidenbach, B. Gaede, and G. Rock. Spass & flotter, version 0.42. In *13th International Conference on Automated Deduction, CADE-13*, Springer LNCS, 1996.

[WOLB92] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning, Introduction and Applications (2nd ed.).* McGraw Hill, 1992. for the newest version of OTTER, see the URL: http://www.mcs.anl.gov/home/mccune/ar/-otter/#doc.

A Theorems and Results for the Example

A.1 The theorems

th-01: $\neg\ d \in en \ominus d$

th-02: $d \in en \oplus d$

th-03: $d \in en \rightarrow \text{num_of}(d, en) < \#\ en$

th-04: $\neg\ d \in en \rightarrow \text{el_of}(\#\ en, en \oplus d) = d$

th-05: $d \neq d_1 \wedge d \in en \wedge d_1 \in en \rightarrow \text{num_of}(d, en) \neq \text{num_of}(d_1, en)$

th-06: $\emptyset \neq en \oplus d$

th-07: $\#\ en = 0 \leftrightarrow en = \emptyset$

th-08: $\neg\ d \in en \rightarrow \text{num_of}(d, en) = \text{num_of}(d, \emptyset)$

th-09: $\neg\ d \in en \wedge \neg\ d_0 \in en_0 \wedge d \neq d_0 \rightarrow en \oplus d \neq en_0 \oplus d_0$

th-10: $\#(\emptyset \oplus d) = 0\ {+1}$

th-11: $d_0 \neq d \wedge \neg\ d_0 \in en \rightarrow \#((en \oplus d) \oplus d_0) = (\#(en \oplus d)){+1}$

th-12: $\neg\ d \in en \wedge \neg\ d_0 \in en \rightarrow (en \oplus d = en \oplus d_0 \leftrightarrow d = d_0)$

th-13: $en \neq \emptyset \rightarrow (\exists\ d_1, en_1.\ en = en_1 \oplus d_1 \wedge \neg\ d_1 \in en_1)$

th-14: $d \in en \rightarrow \text{num_of}(d, en) \neq \#\ en$

th-15: $n < \#\ en \rightarrow (\exists\ d.\ d \in en \wedge \text{num_of}(d, en) = n)$

th-16: $\neg\ d \in en \wedge n < \#\ en \rightarrow \text{el_of}(n, en \oplus d) = \text{el_of}(n, en)$

th-17: $n < \#\ en \rightarrow \text{el_of}(n, en) \in en$

th-18: $\neg\ d \in en \wedge n < \#\ en \rightarrow \text{el_of}(n, en) \neq d$

th-19: $n < \#\ en \rightarrow \text{num_of}(\text{el_of}(n, en), en) = n$

th-20: $n_1 < \#\ en \wedge n_2 < \#\ en \rightarrow (\text{el_of}(n_1, en) = \text{el_of}(n_2, en) \leftrightarrow n_1 = n_2)$

th-21: $n \neq n_1 \wedge n < \#\ en \wedge n_1 < \#\ en \rightarrow \text{el_of}(n, en) \neq \text{el_of}(n_1, en)$

th-22: $en = en_0 \leftrightarrow \#\ en = \#\ en_0 \wedge (\forall\ n.\ n < \#\ en \rightarrow \text{el_of}(n, en) = \text{el_of}(n, en_0))$

th-23: $en = en_0 \leftrightarrow \#\ en = \#\ en_0 \wedge$
$(\forall\ d.\ d \in en \leftrightarrow d \in en_0) \wedge (\forall\ d.\ \text{num_of}(d, en) = \text{num_of}(d, en_0))$

th-24: $\neg\ d_0 \in en \wedge d_0 \neq d \rightarrow (en \oplus d) \ominus d_0 = en \oplus d$

th-25: $d_0 \neq d \wedge \neg\ d_0 \in en \rightarrow \#((en \ominus d) \oplus d_0) = (\#(en \ominus d)){+1}$

th-26: $d_0 \in en \wedge d_0 \neq d \rightarrow (en \ominus d) \oplus d_0 = en \ominus d$

th-27: $d_0 \in en \wedge d_0 \neq d \rightarrow (en \oplus d) \oplus d_0 = en \oplus d$
th-28: $\neg\ d_0 \in en \wedge d_0 \neq d \rightarrow (en \ominus d) \ominus d_0 = en \ominus d$
th-29: $d_0 \in en \wedge d_0 \neq d \rightarrow \#((en \oplus d) \ominus d_0) = (\#(en \oplus d))\text{-}1$
th-30: $\neg\ d \in en \wedge d_0 \neq d \rightarrow \#((en \oplus d) \ominus d_0) = (\#(en \ominus d_0))+1$
th-31: $num_of(d, en \ominus d) = num_of(d, \emptyset)$
th-32: $d_0 \neq d \rightarrow num_of(d, (en \ominus d) \oplus d_0) = num_of(d, \emptyset)$
th-33: $\neg\ d_1 \in en \wedge d \neq d_1 \rightarrow num_of(d_1, (en \ominus d) \oplus d_1) = \#(en \ominus d)$
th-34: $d \in en \rightarrow \neg\ \#\ en < num_of(d, en)$
th-35: $\neg\ d \in en \rightarrow num_of(d, en \ominus d_0) = num_of(d, \emptyset)$
th-36: $\neg\ d \in en \wedge d_1 \neq d \rightarrow num_of(d, (en \ominus d_0) \oplus d_1) = num_of(d, \emptyset)$
th-37: $\neg\ d \in en \wedge d_1 \neq d \rightarrow num_of(d, (en \oplus d_1) \ominus d_0) = num_of(d, \emptyset)$
th-38: $d \neq d_1 \rightarrow (en \oplus d_1) \ominus d = (en \ominus d) \oplus d_1$
th-39: $el_of(n, en) \in en \wedge n < \#\ en \rightarrow num_of(el_of(n, en), en) = n$
th-40: $d \in \emptyset \oplus d_1 \leftrightarrow d = d_1$
th-41: $(en \ominus d) \ominus d = en \ominus d$
th-42: $(\emptyset \oplus d) \ominus d = \emptyset$
th-43: $\emptyset \ominus d = \emptyset$
th-44: $el_of(0, \emptyset \oplus d) = d$
th-45: $\#\ en = 0 +1 \rightarrow \emptyset \oplus el_of(0, en) = en$
th-46: $\#\ en = 0 +1 \rightarrow (d \in en \leftrightarrow d = el_of(0, en))$
th-47: $\#\ en = 0 +1 \rightarrow (en = \emptyset \oplus d \leftrightarrow el_of(0, en) = d)$
th-48: $\#\ en = 0 +1 \rightarrow el_of(0, en) \in en$
th-49: $\#\ en = 0 +1 \rightarrow en \ominus el_of(0, en) = \emptyset$
th-50: $\neg\ d \in en \rightarrow (en \oplus d) \ominus d = en$
th-51: $\#\ en = 0 +1 \rightarrow (d \in en \leftrightarrow en = \emptyset \oplus d)$
th-52: $\neg\ d \in en \wedge \neg\ d_0 \in en_0 \rightarrow (en \oplus d = en_0 \oplus d_0 \leftrightarrow en = en_0 \wedge d = d_0)$

A.2 The Results

The following table gives the results of applying the theorem provers Otter, Setheo, Spass and $_3T^AP$ to the theorems listed above, one time with the original set of axioms and one time with the reduced set (columns marked with 'red.'). At the time we made the experiment the theorems were already proved in KIV in the order in which they are numbered. As in KIV, previously proved theorems were therefore available for the proofs in Otter, Setheo Spass and $_3T^AP$. All provers were given a time limit of 2 minutes on a SPARC 20. The proof times in the table are in seconds, and do not take preprocessing time into account.

The number of interactions in KIV is given in the last column, with an exclamation mark indicating, that the KIV proof made use of the induction rule. An inductive proof in KIV does not necessarily mean, that a noninductive proofs is impossible (see e.g. th-19), but for th-03, th-07, th-08, th-13, th-15, th-22 and th-23 (marked with '!') inductive proofs seem unavoidable (th-52 has a noniductive proof involving two applications of th-23, although Otter Setheo and Spass only find a proof involving $\ominus$). To prove the theorems with KIV 69 interactions and 4 hours of work were needed. The last line gives the number of proved theorems (Σ). For th-14 the positions marked with an 'X' refer to the

fact, that it is not provable with the reduced set of axioms, since it must be generalized to th-03 (see sect. 3). Finally, here are some informations on how the provers were set up and how the problems were prepared for them:

Otter Otter (version 3.0.4) has a built-in equality predicate, but no sorts. These were encoded by mapping terms t of sort s to pairs $mk(t, s)$, where s is a constant. Otter was used with the settings of auto-mode, but with the (negated) theorem to prove as the set of support (see p. 552 of [WOLB92]) and with binary resolution instead of hyper resolution (since the latter is not complete when combined with sos). We tried some other settings, but all resulted in fewer provable theorems. E.g. in auto-mode, 19 resp. 26 theorems could be proved without resp. with reduction. For th-48 auto-mode finds the rather trivial two-step proof (application of th-46 and reflexivity) in 0.10 seconds, while our setting only finds a complex proof in 138 seconds.

Setheo Setheo (V 3.3) can handle only clauses from an unsorted logic, and has no built-in equality predicate. Therefore we had to encode formulas as clauses. Sorts were encoded as for Otter. Equality was explicitly axiomatized. Setheo was used with the '-wdr' option, which performed significantly better than the '-dr' option.

Protein Protein (v 2.12) was given the same input as Setheo. It was used with the model elimination calculus, which gave the best results.

Spass Spass (v 0.55) is a theorem prover for unsorted first-order logic with equality. Since unary predicates are treated by special "sort" inference rules we encoded sorts as unary predicates.

$_3$TAP $_3$TAP (version 4.0) is a theorem prover for full first-order logic with equality. In a project together with the developers of $_3$TAP on the integration of interactive and automated theorem proving an interface between KIV and $_3$TAP has already been built. Therefore, in contrast to all other provers, $_3$TAP could be called directly from KIV. The experiments with $_3$TAP benefited from the fact, that KIV gives only a subset of the previously proved lemmas to $_3$TAP.

theorem	Protein	Protein red	Spass	Spass red	Otter	Otter red	Setheo	Setheo red	$_3$TAP	$_3$TAP-r red	KIV
th-01	0.00	0.00	—	51.48	0.03	0.02	0.99	0.40	6.54	0.34	0
th-02	0.00	0.00	78.72	0.23	0.02	0.01	0.99	0.18	0.31	0.13	0
th-03	!	!	!	!	!	!	!	!	!	!	0!
th-04	13.15	0.82	—	0.25	1.86	0.04	5.49	0.35	—	20.9	1
th-05	—	21.73	53.10	0.15	0.04	0.01	—	0.84	—	—	2
th-06	0.00	0.02	1.07	0.13	0.01	0.02	1.01	0.18	—	5.49	2
th-07	!	!	!	!	!	!	!	!	!	!	0!
th-08	!	!	!	!	!	!	!	!	!	!	0!
th-09	—	—	97.98	1.40	82.97	10.96	—	—	—	—	4
th-10	0.03	0.00	58.83	0.28	10.08	—	1.92	0.23	—	7.77	0
th-11	—	0.02	—	0.47	7.20	0.74	2.35	0.26	31.42	1.81	0
th-12	0.07	0.03	60.07	0.37	0.42	0.81	1.08	0.25	—	—	0
th-13	!	!	!	!	!	!	!	!	!	!	0!
th-14	0.18	X	46.40	X	0.01	X	1.97	X	—	X	1

theorem	Protein	Protein red	Spass	Spass red	Otter	Otter red	Setheo	Setheo red	$_3T^AP$	$_3T^AP$-r red	KIV
th-15	!	!	!	!	!	!	!	!	!	!	3!
th-16	—	—	—	—	—	—	—	—	—	—	3
th-17	—	—	97.92	16.78	0.27	0.05	—	—	—	—	1
th-18	0.02	0.03	48.63	7.92	0.15	0.08	1.20	0.40	3.64	0.45	0
th-19	—	—	—	20.83	0.28	0.05	—	40.51	—	—	2!
th-20	—	8.42	—	17.87	—	—	11.41	0.49	—	—	1
th-21	0.00	0.00	119.88	19.52	0.49	0.16	1.14	0.42	—	—	0
th-22	!	!	!	!	!	!	!	!	!	!	5!
th-23	!	!	!	!	!	!	!	!	!	!	9!
th-24	0.48	0.27	—	89.57	1.00	0.86	1.31	0.68	—	5.92	0
th-25	0.05	0.02	108.33	56.43	0.16	0.16	1.31	0.63	—	12.53	0
th-26	0.57	0.38	—	71.50	1.70	1.16	1.28	0.68	—	—	0
th-27	0.02	0.00	117.92	1.12	0.42	0.16	1.26	0.31	—	1.41	0
th-28	0.02	0.02	90.07	53.20	0.19	0.17	1.24	0.65	5.49	1.91	0
th-29	0.05	0.02	—	86.52	0.41	0.31	1.32	0.70	—	15.93	0
th-30	—	—	—	—	—	—	—	—	—	—	0
th-31	0.00	0.00	45.02	28.15	0.00	0.00	1.32	0.75	—	—	1
th-32	0.12	0.05	—	104.28	2.76	2.53	1.39	0.68	5.2	1.5	0
th-33	0.05	0.00	—	56.23	0.21	0.20	1.32	0.70	—	0.65	0
th-34	0.07	0.00	46.12	9.22	31.50	5.85	1.58	0.53	5.57	1.06	3
th-35	0.12	0.07	104.48	52.63	0.16	0.15	1.35	0.68	4.31	0.72	1
th-36	45.87	0.30	—	109.28	—	105.21	2.87	0.89	23.29	0.67	0
th-37	0.13	0.05	—	105.13	23.29	21.27	1.31	0.74	—	1.03	1
th-38	—	—	—	—	—	—	—	—	—	—	11
th-39	0.00	0.02	109.79	22.95	0.36	0.17	1.35	0.48	—	—	0
th-40	15.50	0.42	—	1.67	0.11	0.03	7.94	0.51	—	—	0
th-41	0.00	0.00	1.97	1.02	0.03	0.02	1.33	0.78	—	—	0
th-42	—	—	—	—	0.68	0.61	—	11.07	—	23.47	4
th-43	0.00	0.00	2.08	1.50	0.01	0.70	1.45	0.76	—	4.59	0
th-44	7.83	0.48	99.73	3.53	—	3.23	6.94	0.58	—	—	1
th-45	—	—	—	66.22	—	11.05	—	—	—	—	1
th-46	—	—	—	6.22	—	105.09	90.54	4.45	—	—	0
th-47	119.03	2.88	—	2.08	36.18	6.94	2.40	0.53	—	—	0
th-48	0.00	0.00	53.22	2.93	1.19	—	1.38	0.34	—	—	0
th-49	0.65	0.43	—	—	1.76	0.70	5.90	1.42	—	—	1
th-50	—	—	—	—	—	—	—	—	—	—	5
th-51	—	—	—	9.47	—	21.60	—	56.54	—	20.19	0!
th-52	—	—	118.03	—	12.29	—	117.15	—	—	—	5
Σ	30	32	22	37	35	36	34	36	9	21	52

This article was processed using the LaTeX macro package with LLNCS style