Theorem Proving in Large Theories*

Wolfgang Reif**
Universität Ulm

Gerhard Schellhorn***
Universität Ulm

1 Introduction

This paper investigates the performance of automated first-order theorem provers in formal software verification. We used the software verification tool, KIV ([5], [6]) as a test environment, and did comparative experiments with five automated theorem provers as dedicated subsystems for the non-inductive first-order theorems that showed up during proofs of specification- and program properties in KIV. The five provers were Otter ([11]), Protein ([1]), Setheo ([3]), Spass ([9]) and $_3T\!A\!P$ ([2]).

The challenge for the provers in this application domain (unlike in standard TPTP benchmarks, [8]) is the large number of up to several hundred axioms in typical software specifications. We found that both the success rates and the proof times of the automated provers strongly depend on how good they are able to find out the few relevant axioms that are really needed in the proofs.

In this paper we present a reduction technique that helps automated provers to concentrate on the right axioms. The reduction technique 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. To evaluate the reduction technique we repeated the original experiments once more, but now with the reduced axiom sets. The largest theory in our experiments had 500 axioms. The reduced axiom sets for the test theorems 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.

The reduction works because software specifications are well structured theories. Finding their structure is part of the early phases in the software life cycle. The reduction not only exploits this structure (given by the software engineer) but to a certain extent also explores resolvable axiom dependencies within unstructured specification components.

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. Section 4 reports on the experimental results.

* This research was sponsored by the German Research Foundation (DFG) under grant Re 828/2-2, SPP 'Deduktion', project 'Integration of Automated and Interactive Theorem Proving', Universities Karlsruhe and Ulm.

** Abt. Programmiermethodik, Universität Ulm, D-89069 Ulm, Germany, reif@informatik.uni-ulm.de

*** Abt. Progr.meth., Universität Ulm, D-89069 Ulm, Germany, schellhorn@informatik.uni-ulm.de

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. Examples for enumerations 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.

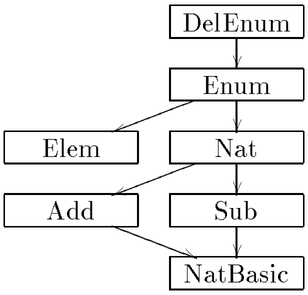
Enumerations can be constructed by $\emptyset$ (the empty bijection), and by $en \oplus d$ adding a data element d (with fresh code number) to the enumeration en. Adding an element twice has no effect. The size function $\#en$ returns the number of elements recorded in the enumeration, and the predicate $d \in en$ 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 $en \ominus d$.

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 is made explicit in the specification structure.

The formal specification of finite enumerations is partly 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, omitted in Fig. 1): *NatBasic* is the freely generated fragment of the natural numbers with 0, successor $succ$, predecessor $pred$ and ordering $<$. The enrichments *Add* and *Sub* introduce addition and subtraction by recursive definitions. *Nat* is the union of *Add* and *Sub*. The formal semantics of the specification language is described in [10].

The specification *DelEnum* (inclusive of all subspecifications) has 25 axioms, 13 of which are given in Fig. 1. 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 library specification may include axioms that are relevant to one application but irrelevant to another. Altogether we get 102 axioms.

As test theorems we selected the 52 proof obligations for *DelEnum*, that showed up during a KIV case study on depth-first search in graphs. 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 five provers was doubled.



DelEnum =
enrich Enum **with**
functions
$\quad . \ominus . : \text{enum} \times \text{elem} \rightarrow \text{enum};$
axioms
$\neg\, d \in en \rightarrow en \ominus d = en,$
$d \in en \rightarrow \#(en \ominus d) = \text{pred}(\#\, en),$
$d_1 \in en \ominus d \leftrightarrow d \neq d_1 \wedge d_1 \in en,$
$\quad d \in en \wedge d_1 \in en$
$\quad \wedge\ \text{num\_of}(d, en) < \text{num\_of}(d_1, en)$
$\rightarrow \quad \text{num\_of}(d_1, en \ominus d)$
$\quad = \text{pred}(\text{num\_of}(d_1, en)),$
$\quad d \in en \wedge d_1 \in en$
$\quad \wedge\ \text{num\_of}(d_1, en) < \text{num\_of}(d, en)$
$\rightarrow \text{num\_of}(d_1, en \ominus d) = \text{num\_of}(d_1, en),$
end enrich

Enum =
generic specification
parameter Elem
using Nat **target**
sorts enum;
constants $\emptyset$: enum;
functions
$\quad . \oplus . \quad : \text{enum} \times \text{elem} \rightarrow \text{enum};$
$\quad \text{num\_of} : \text{elem} \times \text{enum} \rightarrow \text{nat};$
$\quad \text{el\_of} \quad : \text{nat} \times \text{enum} \rightarrow \text{elem};$
$\quad \#\, . \quad : \text{enum} \rightarrow \text{nat};$
predicates $. \in . : \text{elem} \times \text{enum};$
variables en: enum;
axioms
enum **generated by** $\emptyset$, $\oplus$;
$d \in en \rightarrow en \oplus d = en,$
$\neg\, d \in \emptyset,$
$d_1 \in en \oplus d \leftrightarrow d = d_1 \vee d_1 \in en,$
$\#\, \emptyset = 0,$
$\neg\, d \in en \rightarrow \#(en \oplus d) = \text{succ}(\#\, en),$
$\neg\, d \in en \rightarrow \text{num\_of}(d, en \oplus d) = \#\, en,$
$\quad d \neq d_1$
$\rightarrow \text{num\_of}(d, en \oplus d_1) = \text{num\_of}(d, en),$
$d \in en \rightarrow \text{el\_of}(\text{num\_of}(d, en), en) = d$
end generic specification

Fig. 1. The Example Specification

3 Reduction Criteria

Specifications in KIV are built up from elementary first-order theories (with loose semantics) with the usual operations known in algebraic specification: union, enrichment, parameterization, actualization and renaming.

Structuring operations are not used arbitrarily in formal specifications of software systems. Enrichments “ESPEC = **enrich** SPEC **by** Δ”, where Δ consists of a signature and axioms 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.

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 of SPEC, already holds in SPEC.

For structuring operations other than enrichment there are criteria similar to hierarchy persistency which also allow safe reduction of axioms. A specification in which all structuring operations fulfil these criteria is called *modular.* Modular specifications are very natural in software development[1]. The specification from the last section given in fig. 1 is a modular specification. Often the structure of an implementation by a system of software modules (for the definition in KIV see [5]) follows the structure of the specification. The specification is then called an *architectural specification* ([4]).

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

We will now give several axiom reduction criteria and demonstrate them by an example. For modular specifications, these criteria can be proved to be safe, i.e. a formula follows from a set of axioms if it follows already from the reduced set of axioms. Even if there are some non hierarchy persistent enrichments in a structured specification, the axiom reduction is still a very good heuristic.

Suppose we wanted to prove

th-45: $\#en = succ(0) \rightarrow \emptyset \oplus el\_of(0, en) = en$

from our example theory. In the test suite, th-45 is the 45^{th} theorem. Therefore at this stage there are 146 (= 102 + 44) axioms in the theory. Potentially all of them can 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 *Enum* fuand two axioms from *NatBasic*).

The first criterion we apply is the

> *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.

In our case the minimal specification MSPEC is *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. In practice, this criterion does not lead to much reduction, since theorems are usually formulated over the minimal subspecification. Second, we apply the

> *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.

From this criterion we find, that 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. 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, we can apply the

> *specification criterion*: Specifications, which contain operations, which have hierarchy persistent definitions — typical cases are operations over free datatypes, which are defined by (recursive or nonrecursive) definitions — can be split into a basic specification and into enrichments for the defined operations.

This criterion splits *NatBasic* into a specification, which defines 0, *succ* and two enrichments for the predecessor function and the $<$ predicate. Now we can apply the

> *recursion criterion*: The three previous criteria can be applied recursively, until the set of axioms gets stable.

This eliminates the less predicate and the predecessor function. Again this removes 6 axioms and 18 lemmas from the theory.

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. 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 11 seconds.

4 Experimental Results

To evaluate the results of the axiom reduction we first tried the automated theorem provers Otter, Protein, Setheo, Spass and $_3T^AP$ on 45 noninductive (+ 7 inductive) theorems defined over the *DelEnum* specification from section 2. No theorem was invented for this case study, all were existing theorems from a larger KIV case study. All provers were given 2 minutes of proof time on a SPARC 20. The results are summarized in Table 1. The first line in the table gives the number of theorems which could be proved with the full set of axioms, the second line gives the number for the reduced set of axioms.

	Otter	Protein	Setheo	Spass	$_3T^AP$
full axiom set	35	30	34	22	9
reduced set	36	32	36	37	21

Table 1. Results for the Enum example

The numbers show that all provers benefited from the reduction of axioms, but there were enormous differences: Very significant improvements were made by Spass and $_3T^AP$, while the other three provers benefited less. The time necessary to prove theorems was reduced by a factor of three on average.

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 ([7]). These are formulated over a specification which is built up from a lot of standard datatypes (lists, tuples, pairs etc.).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 is much larger (ca. 400). The axiom reduction is more effective than in the previous example: The reduced set contains in most cases only between 4 (!) and 25 axioms. Again the results varied largely as can be seen in table 2. Otter cannot prove more theorems, but the time to prove them decreases in several examples from over 30 seconds to about 2.

	Otter	Protein	Setheo	Spass	$_3T^AP$
full set	48	39	45	4	8
red. set	48	45	48	45	21

Table 2. Results for the compiler verification example

	Otter	Setheo
full set	24	18
red. set	31	29

Table 3. Results for the graph example

It seems, that the syntacic criteria for using axioms built into Otter, Setheo and Proteinare already are already strong enough, to avoid deduction with most irrelevant axioms (for Otter the goal was distinguished by using the set of support strategy and binary resolution; auto-mode, which does not distinguish the goal, can only prove 19 resp. 26 theorems). This is not too surprising, since in a flat specification structure using only axioms involving sorts (encoded as constants), which also occur in the theorem is already a good approximation to the set of relevant axioms. To see, how Otter and Setheo (Protein is very similar to Setheo, so we did not try it) 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. The full set of axioms contains over 500 axioms. The 40 theorems considered were theorems on the sets of nodes. Axiom reduction yields below 100 relevant axioms for all these theorems. Table 3 gives the results. They clearly show the positive effect of axiom reduction. It also shows that the syntactic criteria built into the provers is orthogonal to exploiting the specification structure with axiom reduction.

References

[1] P. Baumgartner and U. Furbach. Protein: A prover with a theory extension interface. In *Proc. 12th CADE*, LNCS 804. Springer, 1994. for the newest version of Protein, see the URL: http://www.uni-koblenz.de/ag-ki/Implementierungen/Protein/.

[2] B. Beckert, R. Hähnle, P. Oel, and M. 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. for the newest version of $_3T^AP$, see the URL: http://i12www.ira.uka.de/˜threetap/.

[3] C. Goller, R. Letz, K. Mayr, and J. Schumann. Setheo v3.2: Recent developments – system abstract. In A. Bundy, (ed.), *12th Int. Conf. on Automated Deduction*, Springer LNCS 814. Nancy, France, 1994. for the newest version of Setheo, see the URL: http://wwwjessen.informatik.tu-muenchen.de/forschung/reasoning/-setheo.html.

[4] 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.

[5] 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.

[6] W. Reif, G. Schellhorn, and K. Stenzel. Proving System Correctness with KIV 3.0. In *14th International Conference on Automated Deduction. Proceedings*, pages 69 – 72. Townsville, Australia, Springer LNCS 1249, 1997.

[7] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 1997. available at the URL: http://hyperg.iicm.tu-graz.ac.at/jucs/.

[8] G. Sutcliffe, C. Suttner, 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.

[9] C. Weidenbach, B. Gaede, and G. Rock. Spass & flotter, version 0.42. In *13th Int. Conf. on Automated Deduction, CADE-13*, Springer LNCS, 1996. for the newest version of Spass, see the URL: http://www.mpi-sb.mpg.de/guide/software/spass.html.

[10] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675 – 788. Elsevier, 1990.

[11] 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.