AHRENDT, BECKERT, HÄHNLE, MENZEL, REIF, SCHELLHORN, SCHMITT

INTEGRATING AUTOMATED AND INTERACTIVE THEOREM PROVING

1. INTRODUCTION

Automated and *interactive* theorem proving are the two main directions in the field of deduction. Most chapters of this book belong to either the one or the other, whether focusing on theory, on methods or on systems. This reflects the fact that, for a long time, research in computer-aided reasoning was divided into these two directions, driven forward by different communities. Both groups offer powerful tools for different kinds of tasks, with different solutions, leading to different performance and application profiles. Some important examples are: ACL2 (Kaufmann and Moore, 1988), HOL (Gordon, 1988), IMPS (Farmer et al., 1996), Isabelle (Paulson, 1994), KIV (Reif et al., 1997) (see also Chapter I.3.14), NQTHM (Boyer and Moore, 1979), and PVS (Owre et al., 1992) for the interactive (or tactical) theorem proving community; and KoMeT (Bibel et al., 1994), Otter (Wos et al., 1992), Protein (Baumgartner and Furbach, 1994), Setheo (Goller et al., 1994), Spass (Weidenbach et al., 1996), and $_3T\!A\!P$ (Beckert et al., 1996) for the automated theorem proving community.

In this chapter we present a project to *integrate interactive and automated theorem proving*. Its aim is to combine the advantages of the two paradigms. We focus on one particular application domain, which is deduction for the purpose of software verification. Some of the reported facts may not be valid in other domains. We report on the integration concepts and on the experimental results with a prototype implementation.

Automatic provers are very fast for the majority of the problems they can solve at all. With increasing complexity, response time increases dramatically. Beyond a certain problem size, automated theorem provers produce reasonable results only in very exceptional cases. Interactive theorem provers on the other hand can be used even in very large case studies. For small problems, they do, however, require many user interactions, particularly when combinatorial exhaustive search has to be performed.

Concerning software verification and the typical proof tasks arising there, the gap between both methods (if applied naively) is even more dramatic.

There are essentially two reasons for that phenomenon. First, the theories occurring in verification projects are very large (hundreds of axioms). Second, the majority of these axioms use equality. Such theories are not well handled by automatic provers. We present techniques to relieve these problems.

We investigate a conceptual integration of interactive and automated theorem proving for software verification that goes beyond a loose coupling of two proof systems. Our concrete application domain turned out to have an enormous influence on the integration concepts. We have implemented a prototype system combining the advantages of both paradigms. In large applications, the integrated system incorporates the proof engineering capabilities of an interactive system and, at the same time, eliminates user interactions for those goals that can be solved by the efficient combinatorial proof search embodied in an automated prover. We report on the integration concept, on the encountered problems, and on experimental results with the prototype implementation. Furthermore, the current directions of our ongoing research are described.

The technical basis for the integration are the systems KIV (Reif, 1995) and $_3T\!A\!P$ (Beckert et al., 1996), both of which were developed in the research groups of the authors at Ulm and Karlsruhe. KIV (Karlsruhe Interactive Verifier) is an advanced verification system which has been applied in large realistic case studies in academia and industry for many years now. $_3T\!A\!P$ (Three-valued Tableau-based Automated Theorem Prover) is an automated tableau prover for full first-order logic with equality. It does not require normal forms, and it is easily extensible. Although we experimented with these particular systems, the conceptual results carry over to other provers.

Based on statistics from case studies in KIV, we estimate that in our application domain up to 30% of all user interactions needed by an interactive prover could be saved in principle by a first-order theorem prover. Current provers, however, are far from this goal, because they are in general not prepared for deduction in large software specifications (i.e., very large search spaces) or for typical domain specific reasoning. In Section 2 we describe these and other problems, and in the Sections 3 to 6 we present the solutions we came up with so far.

Many of our decisions are based on experimental evidence. Therefore, we put a lot of effort in a sophisticated verification case study: Correct compilation of Prolog programs into Warren Abstract Machine code ((Schellhorn and Ahrendt, 1997) and Chapter III.2.6). We use it as a reference or benchmark. Parts of it are repeated every now and then to evaluate the success of our integration concepts, see Section 7.

In realistic applications in software verification, proof attempts are more likely to fail than to go through. This is because specifications, programs, or user-defined lemmas typically are erroneous. Correct versions usually are only obtained after a number of corrections and failed proof attempts. Therefore, the question is not only how to produce powerful theorem provers but also how to integrate proving and error correction. Current research on this and related topics is discussed in Section 8.

There are different approaches of combining interactive methods with automated ones. Their relation to our approach is the subject of Section 9. Finally, in Section 10 we draw conclusions.

2. Identifying the Problems of Integration

Theorem proving with an interactive system typically proceeds by simplifying goals using backward reasoning with proof rules (tactics). Many proof rules may be applied automatically, but usually the tactics corresponding to the main line of argument in the proof must be supplied interactively. To allow the proof engineer to keep track of the details of a proof, system response time to the application of tactics should be short.

In the case of software verification, the initial goals contain programs. The tactics to reduce these goals make use of first-order lemmas and ultimately reduce the goal to a first-order formula. Usually, these first-order goals require interaction as well as the program goals. Using an (ideal) automated theorem prover would relieve the proof engineer from a lot of interaction. Therefore, the scenario we considered was to use $_3T^AP$ as a *tactic* to prove first-order goals, thus exploiting its capability of fast combinatorial search. A suitable interface was implemented such that $_3T^AP$ can be called either interactively or by KIV's heuristics. Termination of this tactic was guaranteed simply by imposing a time limit (usually between 15 seconds and 1 minute).

Based on this first, loosely integrated version, we started to experiment with using the automatic theorem prover to solve first-order theorems encountered in software verification. As expected, the automatic theorem prover initially could not meet the requirements found in software verification. Virtually no relevant theorem could be proved. Analysis of the proof attempts identified a number of reasons, some expected and some unexpected. The most important ones are:

Interface Automated theorem provers usually do not support separate input of axioms and goal. Instead, one is forced to prove the combined goal "axioms imply theorem" with universally quantified axioms and theorem. Most theorem provers do some preprocessing on formulas to speed up proof search. In $_3T^AP$, links between potentially unifiable terms are computed, whereas in many other theorem provers formulas are converted to clauses. We found that preprocessing 200 axioms with $_3TAP$ takes about 30 sec. (the same holds for other provers we tested, see Chapter III.2.9). Preprocessing the axioms at every proof attempt is obviously unacceptable for interactive theorem proving.

Correctness Management Automated provers do not record which assumptions were actually needed in a proof. But such information is necessary if the interactive theorem prover does not rely on strict bottom-up proving. In KIV, for example, the correctness management prevents cycles in the dependencies of lemmas and invalidates only a minimal set of previous work when goals or specifications are changed.

Different Logics Most automated theorem provers only support formulas of first-order logic without sorts (e.g. Otter) or even only clauses (e.g. Setheo) while interactive theorem provers often support more expressive logics like higher-order logic (e.g. HOL, Isabelle or PVS) or type theory (e.g. NuPRL (Constable et al., 1986) or Typelab, see Chapter I.3.16).

Inductive Theorems Many theorems can only be proved inductively from the axioms, but most automated theorem provers (including $_3TAP$) are not capable of finding inductive proofs.

Large Theories Automated provers are tuned to prove theorems over small theories and a small signature. Moreover, the given axioms are always relevant to prove the goal. In contrast, theories used in software verification usually contain hundreds of axioms of which only few are relevant for finding a particular proof. Still, all axioms contribute to the search space.

Domain Characteristics In our application domain, which is software verification, specifications are well structured and have specific properties (e.g. sorted theories, equality reasoning is important). Also, theorems used as lemmas often have an operational meaning in the interactive theorem prover (e.g. equations oriented as rewrite rules). Automated theorem provers do not exploit these properties.

The first two items above are technical problems requiring mere changes to the interface. Now, preprocessing of axioms is done by $_3TAP$ when the user of the integrated system selects a specification to work on. A separate command for initiating proof attempts refers to the preprocessed axioms by naming the specification in which they occur. Embedding the automated prover in the correctness management is done by converting proofs found by $_3T\!A\!P$ to proofs in the sequent calculus used in KIV.

To handle the problem of different logics, a suitable subset of the formulas treated by the interactive system must be selected and, in general, translated to first-order logic. For those automatic provers which only support clauses, even the first-order formulas have to be transformed by standard encoding techniques like the ones described in (Plaisted and Greenbaum, 1986).

In our case, the solution is simpler, since the logic used in KIV is actually an extension of many-sorted first-order logic by program formulas (Dynamic Logic), and $_3T\!A\!P$ supports full many-sorted first-order logic. For a discussion of some of the issues that have to be solved for higher-order logic (polymorphism, currying, problems arising from the identification of boolean terms with formulas) see (Archer et al., 1993).

The presence of inductive theorems is a fundamental problem. It can be mitigated by adding previously derived (inductive) theorems as lemmas. On the one hand, this reduces the number of theorems which require inductive proofs in our experiments to about 10% of all theorems (see Chapter III.2.9). On the other hand, there are roughly as many potentially useful or necessary lemmas as there are axioms, which adds considerably to the problem of large theories.

This leaves us with two problems, namely handling large theories and exploiting domain characteristics (of software verification). Both will be tackled in the following sections. The number of potentially relevant axioms in proving a goal is minimized by exploiting a specification's structure (Section 3). Measures we have taken to exploit the characteristics of software verification are presented next. Finally, Section 6 deals with the conversion of proofs from $_3T\!A\!P$ to KIV.

3. Reduction of the Axiom Set

Specifications in KIV are built up from elementary first-order theories with the usual operations of algebraic specification: union, enrichment, parameterization, and actualization. Their semantics is the whole class of models (loose semantics). Reachability constraints like “nat generated by 0, succ” are used to define induction principles. Typical specifications used to formally describe software systems contain several hundred axioms.

Structuring operations are not used arbitrarily in formal specifications of software systems. Almost all enrichments “**enrich** SPEC **by** Δ” are *hierarchy persistent*. This property means that every model of SPEC can be extended to a model of the whole (enriched) specification. Hierarchy persistency cannot be checked syntactically, but is usually guaranteed by a modular implementation of the specification (Reif, 1995).

Hierarchy persistency can be exploited to define simple, syntactic criteria for eliminating many irrelevant axioms, which then no longer must be passed to the automated theorem prover. This enables theorem proving in large theories, and is described in seperately in Chapter (III.2.9). There, the reduction technique used to identify the axioms in question is described in greater detail. It is sufficient, for example, to work merely with the axioms of the minimal specification whose signature contains that of the theorem.

4. Equality Handling

4.1. *Incremental Equality Reasoning*

KIV specifications—like most real word problems—make heavy use of equality. It is, therefore, essential for an automated deduction system that is integrated with an interactive prover to employ efficient equality reasoning techniques.

Part of $_3T^AP$ is a special equality background reasoner that uses a completion-based method (Beckert, 1994) for solving E-unification problems extracted from tableau branches. This equality reasoner is much more efficient than just including the equality axioms. In addition to the mere efficiency of the tableau-based foreground reasoner and that of the equality reasoner, the interaction between them plays a critical role for their combined efficiency. It is a difficult problem to decide when it is useful to call the equality reasoner and how much time to allow for its computations. Even with good heuristics at hand, one cannot avoid calling the equality reasoner either too early or too late. This problem is aggravated in the framework of integration by the fact that most equalities present on a branch are actually not needed to close it, such that computing a completion of all available equalities not only is expensive, but quite useless. These difficulties can (at least partially) be avoided by using incremental methods for equality reasoning (Beckert and Pape, 1996). These are algorithms that—after a futile try to solve an E-unification problem—allow to store the results of the equality reasoner's computations and to reuse them for a later call (with additional equalities). Then, in case of doubt, the equality reasoner can be called early without running the risk of doing useless computations. In addition, an incremental reasoner can reuse data for different extensions of a set of equalities.

Fortunately, due to the inherently incremental nature of $_3T^AP$'s algorithm for solving E-unification problems, it was possible to design and implement an incremental version of it: rewrite rules and unification problems that are extracted from new literals on a branch can simply be added to the data of the background reasoner. Previously, information computed by the equality reasoner of $_3T^AP$ could not be reused, and the background reasoner was either called (a) on exhausted tableau branches (i.e., no expansion rule is applicable; this meant that because of redundant equalities even simple theorems could not be proved) or (b) it was called each time before a branching rule was applied (which usually lead to early calls and repeated computation of the same information). Now $_3T^AP$ avoids both problems. The incremental equality reasoner may be called each time before a disjunctive rule is applied without risking useless computations.

4.2. *Generating Precedence Orders from Simplifier Rules*

As in most interactive theorem provers, in KIV simplification is a key issue. *Simplifier rules* are theorems over a data structure, which have been marked by the proof engineer for use in the simplifier. The way of use is determined by their syntactic shape. The most common simplifier rules are conditional rewrite rules of the form $\phi \rightarrow \sigma = \tau$, intended for rewriting instances of σ to instances of τ if ϕ is provable.

The fact that simplifier rules have an operational meaning within an interactive prover can be used in several ways to guide proof search in automated systems. One example is the automatic generation of useful precedence orders of function symbols (otherwise, an order must be provided manually or an arbitrary default order is used). Such orders are used in many refinements of calculi in theorem proving, hence our results are of general interest.

In $_3T^AP$, the precedence order is used to orient equations with a lexicographic path order based on it. The more equations occurring during a proof can be ordered, the smaller the search space becomes.

A first attempt to generate an order from the rewrite rules of KIV is to define $f > g$ for top-most function symbols f and g that occur on the left and on the right side of a rewrite rule, respectively, provided $f \neq g$. The attempt to generate a total order from this information (by a topological sort), however, typically fails due to a number of conflicts such as the following (*cons*, *car*, *cdr* and *append* are the usual list operations):

append(cons(a,x),y) = cons(a,append(x,y))
x $\neq$ nil $\rightarrow$ cons(car(x),append(cdr(x),y)) = append(x,y)

The first rule suggests, in accordance with intuition, that *append* $>$ *cons*, while the second one suggests the contrary. To avoid such conflicts, one excludes rewrite rules of the form $\phi \rightarrow \sigma = \tau$, where τ can be embedded into σ

(as is the case in the second rule above) from the order generation. We tested the resulting algorithm with five specifications from existing case studies in KIV, each having approximately 100 rewrite rules. The result was that the function symbols could *always* be topologically sorted, except one conflicting pair of rewrite rules. Additional restrictions on the order could be extracted by considering the first *differing* function symbols in rewrite rules instead of the top-most ones (no additional conflicts arose). For maximal flexibility KIV passes on only a partial order (if a conflict is found, no information on this function symbol is generated). The partial order is made total and used in the equality handling module of $_3T\!A\!P$. For yet another use, see the following section.

Our considerations show that rewrite rules, which are used in interactive theorem proving to "simplify" terms (relative to the user's intuition), can be translated rather directly into information used in automated theorem proving. Experiments showed that with suitable simplifier rules and a similar algorithm as the one above, one obtains an analogous result for predicate symbols (provided that the equality symbol is considered to be special).

5. Restricting the Search Space

5.1. *Problem-Specific Orders*

Calculi which incorporate search space restrictions based on atom and literal orders are relatively well investigated in the domain of resolution theory (Fermüller et al., 1993). In order to employ such restrictions in $_3T\!A\!P$, we could build on recent work on order-based refinements for tableau calculi (Hähnle and Klingenbeck, 1996; Hähnle and Pape, 1997). In fact this latter research was partially motivated by the integration of paradigms discussed in the present article.

Ordered tableaux constitute a refinement of free variable tableaux (Fitting, 1990). They have a number of advantages that become particularly important in the context of software verification: They are defined for *unrestricted first-order formulas* (Hähnle and Klingenbeck, 1996) (in contrast to mere clausal normal form) and they are compatible with another important refinement called *regularity* (Letz et al., 1992). It is possible to extend ordering restrictions to theory reasoning (Hähnle and Pape, 1997). Moreover, ordered tableaux are *proof confluent*: every partial tableau proof can be extended to a complete proof provided the theorem to be proven is valid. This property is an essential prerequisite for automated search for *counter examples*, see Section 8 below. Finally, *problem-specific knowledge* can be used to choose an order, which not only restricts the search space but, more importantly, rearranges it favorably.

The last point is difficult to exploit in general, but in the KIV-$_3T^AP$ integration one can take advantage of the same information computed already to reduce the number of axioms (Section 3) and to provide meaningful simplification orders for equality handling (Section 4.2). The hierarchy of specifications and the implicit hierarchy of function symbols within each specification defines a partial order $<$ of the function and predicate symbols occurring in a problem. Such an order $<$ can be naturally extended to an *A-order* $\prec$: a binary, irreflexive, transitive relation on atoms which is stable under substitutions. Ordered tableaux can be characterized by restricting branch extension as follows: a formula ϕ can be used to extend a tableau branch B iff either (i) ϕ has an $\prec$-maximal connection into B (i.e., the connection literal of ϕ occurs $\prec$-maximally in ϕ) or (ii) ϕ has an $\prec$-maximal connection into another input formula ψ (i.e., the connection literals of both ϕ and ψ occur $\prec$-maximally).

One particular *A-order*, based on $<$, is $\preccurlyeq$. For two terms/atoms $s = f(s_1,\ldots,s_n)$, $t = g(t_1,\ldots,t_m)$ is $s \preccurlyeq t$ iff (i) either $f < g$ or $f = g$, $n = m$, and $t_i \preccurlyeq s_i$ for $1 \leq i \leq n$ *and* (ii) the variables of s are a subset of the variables of t. The *automatically* computed $\preccurlyeq$ often prevents literals from unrelated hierarchies to be maximal and, as a consequence, a formula ϕ has no maximal connection into a branch with only literals from a hierarchy unreachable from the maximal literals of ϕ. Even when $\preccurlyeq$ does not perfectly reflect the hierarchy within a problem, completeness of the calculus still guarantees that a proof can be found, although proof search might not be influenced as favorably.

5.2. *Pragmatic Connectives*

As explained above, the automated part of the integrated system is used to prove those sub-tasks that are of first-order nature. In our context of software verification, first-order formulas appear in very different situations: as axioms of a specified theory, as rewrite lemmas, or as subgoals in a proof of a theorem involving programs. Each of these contexts carries its own pragmatics concerning the way formulas are used in proofs. So it is a basic task to enable the automated prover to make use of as much pragmatic information as possible. One solution is to add new logical connectives to the logic of the automated (tableau) prover.

Expansion of a disjunctive formula causes a splitting of the current branch, after which one of the two resulting branches has to be chosen as the new branch in focus. This choice heavily affects the search space. Consider, for example, the formula $p(x) \vee q(x)$: its expansion generates two (sub-)branches, say B_1 and B_2, with leaves $p(x)$ and $q(x)$, respectively. Suppose there are many different instantiations of x that allow to close B_1, but only few instantiations that allow to close B_2. In this case, the search for an instantiation that closes *both* branches should be done by first searching for an instantiation that allows to close B_2 and then checking whether it allows to close B_1 as well; obviously, a much smaller search space is spanned this way.

In general, one has no information that allows to decide which branch should be closed first, so the disjunctive connectives ($\vee$, $\rightarrow$ and $\leftrightarrow$) are treated in a standard way; by default, the left argument is handled first (some theorem provers take the relative size of p and q into account, which may or may not be beneficial). On the other hand, specific knowledge on the role of p or q in the proof would allow to rearrange and optimize the search space.

Typical candidates for this are implications that are intended to *exclude exceptions*. Consider the formula $n \neq 0 \rightarrow \mathit{property}(n)$. It states a property holding for all natural numbers *except* 0. Assume this formula occurs on the branch in focus and $\mathit{property}(n)$ is a complex formula. Expanding the implication, standard treatment puts the new branch B_1 with leaf $n = 0$ in focus. As each of the substitutions $\{n \leftarrow 1\}, \{n \leftarrow 2\}, \ldots$ closes B_1, the natural numbers are enumerated by backtracking. It is much better to examine the branch B_2 containing $\mathit{property}(n)$ first, and then check that the instantiation of n used to close B_2 is not equal to 0.

For this we added a version of implication to the $_3T^AP$ logic, called **if_then**, whose declarative semantics is the same as that of usual implication, but that carries the pragmatic information that the branch associated with the then-part should be closed first. In the integrated system, the control specific distinction between logically equivalent connectives is made by KIV, used by $_3T^AP$, but hidden to the user, protecting him or her from being confused by operational semantics.

6. Converting Proofs

Usually, automated and interactive theorem provers use quite different calculi, supporting machine-oriented proof search and, respectively, the intuition of a human user. To deal with this gap, we chose a dual approach, switching when necessary between the interactive part of the system (based on a sequent calculus) and the automated part of the system (based on free variable tableaux). Doing this we exploit the full power of both methods.

The resulting question is: which kind of information do both system parts have to exchange when they co-operate to construct a proof? First of all, there is *static* information depending on the signature, axioms, lemmas, and the specification structure, but not depending on the formula to prove. The *dynamically* exchanged information, depending on a current proof task, was at first a request to prove a formula, responded by *yes* or *no* (if not timed out). Even this short response is extremely valuable, because it can save the user time and effort otherwise spent in a boring and potentially long proof task.

But there are good reasons for a more informative response, providing the whole proof (if one was found) or at least the assumptions used in it. One important reason is the correctness management used in KIV, which automatically guarantees the absence of cycles in lemma dependencies and automatically invalidates proofs affected by the modification of a lemma. To integrate the proofs found by $_3T^AP$ into this management, KIV needs more than just a *yes*. The second reason (which requires the complete proof as a response) is the replay mechanism of KIV, which is able to rebuild the branches remaining valid of an invalidated proof tree. That is why all proofs found by $_3T^AP$ should be fully transformable into KIV proof trees. Finally, embedding $_3T^AP$ proofs into the KIV calculus makes it possible to visualize the overall proof in a single framework, enabling the user to understand what is going on without requiring him or her to deal with two different calculi.

Therefore, $_3T^AP$ proofs must be translated into KIV proofs. Since the first uses a tableau calculus and the second a sequent calculus, the main task seems to be the translation of tableaux into sequent proofs. But this is not difficult, because there is a one-to-one correspondence between tableaux and sequent proofs, provided we consider appropriate instances of the calculi consisting of corresponding sets of rules. In our case, they do *not* correspond one-to-one because they are designed for different purposes: automated search and interaction. These demand different features. For an interactive proof construction, the rules have a great degree of non-determinism (e.g. an arbitrary term has to be guessed) and, at the same time, they have to be simple. On the other hand, in the context of automated search, nothing is more important as the reduction of search space, which requires a very small degree of freedom, postponing decisions and removing non-determinism as much as possible.

A typical example for that difference is the cut-rule, which is used in an interactive calculus to enable case distinctions over arbitrary formulas. Such a rule is not feasible in an automatic proof procedure. Therefore, $_3T^AP$'s tableau calculus offers only a kind of restricted cut, called lemma generation, which handles disjunctions by adding the negation of one extension to the other extension. Here, the transformation is easy because lemma generation can be very simply simulated by cuts.

Much more interesting is the different handling of instantiations. KIV uses a "ground" calculus, where the system or the user has to guess a term as an instance of an $\forall$-quantified variable when applying a γ-rule. Given this, the handling of $\exists$-quantified variables by the δ-rule is very easy since Skolem *constants* suffice. On the other hand, $_3T^AP$ uses a *free variable* calculus. Here, the γ-rules introduce free variables, representing terms not yet guessed. As a consequence, the δ-rules have to introduce new Skolem terms containing the free variables of the formula. Older versions of these rules (Fitting, 1990) correspond one-to-one to rules of a simple "ground" calculus. But the research in the field of tableaux yielded several improvements. One is a modified δ-rule (δ^{++}), which is proven to allow exponential shorter proofs in extreme cases (Beckert et al., 1993). Here, we decided to provide a "ground" δ-rule that corresponds to δ^{++}. The rule itself has a simple description, but its soundness proof turned out to be difficult.

Another improvement of the free variable tableaux is the usage of *universal variables*. These are special free variables to which arbitrarily many instances can be assigned (see Chapter I.1.1). They have no counterpart in the simple "ground" calculus. To bridge this gap, proofs using universal variables are transformed into ground proofs by collecting all instances assigned to a universal variable and building several ground copies of the affected subproof, one for each instance. The crucial point here is the analysis and handling of dependencies between free variable instances (Stenz, 1997).

7. Evaluation: the WAM Case Study

To evaluate our project results and to demonstrate improvements in the integrated system, we chose to verify selected compilation steps from Prolog to the Warren Abstract Machine (WAM) as our major case study. The algorithms and a mathematical analysis of the compiler were already provided (Börger and Rosenzweig, 1995), allowing us to concentrate on the development of a formal specification and on theorem proving. A first analysis (Schmitt, 1994) of the associated correctness problems showed that verification of the first compilation step poses tasks challenging for both KIV and $_3T^AP$, which should lead to synergy effects. One important challenge for the interactive prover was the fact that, due to its complexity, the correctness proof could be developed only in an incremental process of failed proof attempts, error correction and re-proof (evolutionary verification). For the use of the automated prover an important aspect was, that a large number of standard data types (lists, sets, tuples, etc.) is required, thus a large number of first-order theorems had to be proved. These theorems were used as benchmarks to evaluate improvement of the integrated system.

The content of the case study is described in Chapter III.2.6, together with our solution for the problem of finding large invariants. Here, we just mention that verification revealed some formal gaps in the analysis of (Börger and Rosenzweig, 1995). Moreover, the operational semantics of WAM code, as formalized in that paper, was found to be non-deterministic, allowing non-termination of programs that should terminate according to Prolog semantics. Without giving an explanation here, we just show one example. The Prolog program consisting of the clauses "`p :- fail.`" and "`p.`", when asked the query "`?- p.`", has more than one behavior: it *may* terminate with "`yes`", but it may as well run forever.

It took three man months to verify the two correctness theorems including all required lemmas. The final proofs of the two main theorems in Dynamic Logic required 846 interactions with the user. The algebraic specifications contained 207 axioms; 184 first-order lemmas were used. These were relatively easy to prove compared with the complexity of the main theorems, and usually the first attempt to state these lemmas was correct. Nevertheless, additional 350 interactions were required. About 90% of these interactions could potentially be saved by the use of an (ideal) automated theorem prover (the remaining interactions involve induction). In reality, first tests with the 58 theorems of the top-level specification showed that $_3T^AP$ was only able to prove 5 of these, giving a ratio of only 8%. With the improvements of the integrated system we have implemented so far, the integrated system is now able to prove 21 (36%) theorems fully automatic. The most significant gain in the productivity of $_3T^AP$ (from 9 to 21) was achieved by the reduction of the set of axioms, which often left only 10–20 relevant axioms for each proof.

8. Current Directions of Research

Beyond the work described so far, current research concentrates on deepening the integration of the automated and the interactive proving paradigm by exploiting further domain specifics found in software verification. Four aspects of this work are described in the following.

Tableau Rewriting Using Simplifiers Simplifiers of the form $(\forall x)(p(x) \to \psi(x))$ are frequently used in KIV specifications. Typical examples are *definitions*[1], where the formula ψ is used to specify the meaning of predicate symbol p, such as in $(\forall x)(even(x) \to (\exists y)(x = 2 * y))$.

Simplifiers carry pragmatic information. In proofs, a simplifier is used solely to deduce an instance $\psi(t)$ of the conclusion from a given instance $p(t)$

[1] Definitions are usually equivalences of the form $(\forall x)(p(x) \leftrightarrow \psi(x))$; they replace the two simplifiers $(\forall x)(p(x) \to \psi(x))$ and $(\forall x)(\neg p(x) \to \neg\psi(x))$, but *not* the implication $(\forall x)(\psi(x) \to p(x))$, which is (in general) not a simplifier.

of the premise, thus it is known (and part of the pragmatics of the simplifier) that the contrapositive $(\forall x)(\neg\psi(x) \to \neg p(x))$ is not needed in proofs. Note, that although both simplifiers and if-then formulas are formal implications, the concept of a simplifier is complementary to that of the if-then operator and their pragmatics is completely different. Consequently, the search space should be organized differently.

As there is a strong relationship between simplifiers and equality reduction rules, the improvement one can hope to gain is similar to that of using reduction rules for rewriting terms (as compared to using equalities in an unrestricted way).

Extending the Application Domain Until now we have considered the use of automated theorem provers for goals occurring as lemmas or explicit subgoals during interactive correctness proofs of software systems. There are, however, a lot of other first-order theorems hidden in goals containing programs, which can be proved using automated systems. Some typical subgoals of this kind are:

– to test the applicability of a conditional rewrite rule;
– to check disjunctive goals containing programs for first-order validity;
– to check a program conditional (or its logical complement) for validity in the current context, which allows to omit the else- (then-) branch;
– to determine the reachability of a recursive call (which can be expressed in a first-order formula).

For all these proof tasks, KIV uses its general simplifier (possibly involving user interaction). But all these tasks are suitable for trying $_3T^AP$ as they do not require induction. Compared to the first-order theorems we proved until now, they have a very different characteristic: most of these goals are not theorems. Another new characteristic is that not only a lot of axioms (and usable lemmas) are irrelevant to the proof, but also the goals themselves contain a lot of subformulas which are irrelevant. For routine application of $_3T^AP$ these new problems have to be solved first.

Proof Engineering Beside a high degree of automation, an important criterion for the efficient verification of large software systems is the existence of powerful *proof engineering* tools. These allow incremental development of proofs and thus to cope with failures and resulting corrections and changes. Major components are:

– methods for the analysis of proofs,
– methods to construct counter examples,
– methods to reuse proofs and proof attempts.

There exists a method in KIV to reuse proofs on *program* changes (Reif and Stenzel, 1993). But often it is also necessary to correct *first-order formulas*, e.g. for the incremental development of the *coupling invariant* in the WAM case study. Therefore, one of our major efforts in the future will be to improve the possibilities to analyze and to reuse proofs with changed first-order formulas.

Computing Counter Examples for Non-Valid Problems First-order problems deriving from program verification often do not constitute provable formulas. The reasons range from bugs in the object program or the specification to erroneous tactical decisions supplied earlier by the user (e.g., too weak induction hypothesis). Also, extending the application of an automated prover, as discussed above, involves non-theorems. It is, therefore, an extremely desirable feature of an automated theorem prover to provide counter examples for non-valid formulas.

In general, of course, falsifying interpretations for non-valid first-order formulas are not computable, but in the limited context of program verification additional observations (which have so far been rarely named explicitly) simplify the situation considerably: if a program contains bugs it does not meet its specification for certain inputs. These critical inputs can be obtained from finite counter examples. In practice, such counter examples tend to be small, so there is no need to search for large instantiations.

We can exploit several features specifically present in problems derived from program verification. Counter examples correspond to initial models finitely generated by a known set of function symbols and constants. Moreover, variables are sorted, which further restricts the number of models. On the other hand, one needs to generate models relative to an equality theory.

9. Other Approaches to Integration

There are several approaches described in the literature, which have in common the aim to strengthen interactive proving methods (and tools) by adding automatic methods (and tools) which are able to handle a relevant amount of subtasks.

Apart from first-order logic theorem proving, the most important reasoning methods being integrated in interactive frameworks are: computer algebra (Harrison and Théry, 1993; Ballarin et al., 1995), model checking (Joyce and Seger, 1994; Owre et al., 1996) and decision procedures (Boyer and Moore, 1988). Each of them is typically applied to some particular domain(s). *Computer algebra* is incorporated into proof systems solving problems of mathematical or physical origin. These applications are very different from the problems occurring in software verification, on which we focused in this chapter. *Model checking* is often used in the context of hardware verification. This method is also applied to verify software, but then restricted to (sub-)systems that can be adequately represented by (not too many) finite states. Although the amount of potential applications is growing due to new techniques of abstraction, most problems arising in software verification are infinite-state. Finally, for our domain the *decision procedures* are the most relevant of the listed methods. They are not meant to be flexible but tailor-made for fixed theories that are used very often. Here, the crucial question is: how many datastructures we use often are decidable and, apart from that, unchanging? In the experience of our different case studies, most specifications had often to be modified. But, of course, there are a few decidable theories we use regularly (e.g. fragments of natural numbers, integers), where decision procedures would help.

Computer algebra, model checking, and decision procedures differ from our approach in that the incorporated algorithms handle *decidable* problems. Moreover, the correctness management problem (keeping everything consistent, while the underlying theories are changed and corrected continually) is not relevant for these approaches, because they handle only *fixed* standard theories (e.g. linear arithmetic, calculus and boolean algebra).

In contrast, the problem of impairing correctness of the interactive prover by "believing" the results of connected systems has been an issue in the approaches mentioned above (in particular for computer algebra). In comparison, this issue is less significant for the joint work of two provers.

Relevant work in the area of automated theorem proving are attempts to aggregate several automated provers under a common homogeneous user interface, e.g. ILF (Dahn et al., 1995) (for conceptual issues see also Chapter II.3.10), and experiments that extend existing automated provers by an interactive component, e.g. INKA (Hutter and Sengler, 1996).

We are also aware of several attempts to integrate automated theorem proving techniques with interactive systems, that have been made so far:

The homogeneous approach, which implements an automated theorem prover in the implementation language of the interactive system was used in the HOL system (Schneider et al., 1992). The automated theorem prover FAUST uses a variant of the sequent calculus. The sequent calculus proofs resulting from its application are converted back to HOL proofs. The approach resulted in speed-ups and reduced the necessary interactions. Nevertheless, a prototypical implementation like that of FAUST cannot compete with the results of efficient implementations of automated theorem provers.

Our heterogeneous approach, which combines two different existing provers, has previously been used to combine the HOL system with a resolution prover. The work described in (Archer et al., 1993) concentrates on the selection of a sub-language of higher-order logic suitable for the resolution prover and on proof visualization.

A direct comparison of a heterogeneous and a homogeneous approach can be found in (Busch, 1994). The first is a combination of the higher-order logic proof checker LAMBDA with the automatic first-order logic prover SEDUCT. The second is an extension of LAMBDA by rules suitable for first-order theorem proving, together with heuristics for quantifier instantiation.

Finally we should mention the recent integration of the interactive type theory prover Typelab (see Chapter I.3.16) with the equality prover Discount (Strecker and Sorea, 1997) and current work in progress on combining KIV and INKA in the VSE system (Hutter et al., 1995) and NuPRL with KoMeT.

All these experiments (except the VSE system) are based on higher-order logic languages, in which algorithms are denoted in a functional style. In contrast, our logical framework, the Dynamic Logic, deals with imperative programs combined with first-order formulas. Another difference is the main purpose of our system. It is not designed for formalizing mathematics, but specialized to verify modular programs w.r.t. structured algebraic specifications, i.e. structured first-order theories. Our work differs from others in heavily making use of this application domain, especially for reducing the search space of the automatic prover (see reduction of the axiom set, generating precedence orders and other issues above).

10. Conclusion

Ever since we began the project of integrating interactive and automatic theorem provers, we had the strong feeling to be working on a strategic and promising topic. Already the expected benefit from linking the two theorem proving programs provided sufficient motivation to set to work with extra effort. But this was only the beginning.

We anticipated to identify problems that are not particular to our two theorem proving systems, but would arise in any attempt to combine the two theorem proving paradigms. And in fact we can now name typical trouble spots: the difficulties of automated provers to cope with large sets of axioms, the mismatch between first-order automated theorem proving and higher-order interactive provers, the use of pragmatic information to guide proof search, and the internal communication between the interactive and the automated prover. We have made substantial progress towards finding solutions, which again have significance beyond the special situation we are dealing with.

From this experience and from observing the literature and the focus of conferences we see a new research area emerging that might be called “integrated theorem proving”.

REFERENCES

Archer, M., G. Fink, and L. Yang: 1993, ‘Linking Other Theorem Provers to HOL using PM: Proof Manager’. In: L. Claesen and M. Gordon (eds.): *Higher Order Logic Theorem Proving and its Applications: 4th International Workshop, HUG’92*.

Ballarin, C., K. Homann, and J. Calmet: 1995, ‘Theorems and Algorithms: An Interface between Isabelle and Maple’. In: A. H. M. Levelt (ed.): *International Symposium on Symbolic and Algebraic Computation*. pp. 150–157.

Baumgartner, P. and U. Furbach: 1994, ‘PROTEIN: A Prover with a Theory Extension Interface’. In: A. Bundy (ed.): *Proc. 12th CADE, Nancy, France*, Vol. 814 of *LNCS*. pp. 769–773.

Beckert, B.: 1994, ‘A Completion-Based Method for Mixed Universal and Rigid E-Unification’. In: A. Bundy (ed.): *Proc. 12th CADE, Nancy, France*, Vol. 814 of *LNCS*. pp. 678–692.

Beckert, B., R. Hähnle, P. Oel, and M. Sulzmann: 1996, ‘The Tableau-Based Theorem Prover $_3T^AP$, Version 4.0’. In: M. McRobbie and J. Slanley (eds.): *Proc. 13th CADE, New Brunswick/NJ, USA*, Vol. 1104 of *LNCS*. pp. 303–307.

Beckert, B., R. Hähnle, and P. H. Schmitt: 1993, ‘The Even More Liberalized δ-Rule in Free Variable Semantic Tableaux’. In: G. Gottlob, A. Leitsch, and D. Mundici (eds.): *Proc. 3rd Kurt Gödel Colloquium (KGC), Brno, Czech Republic*. pp. 108–119.

Beckert, B. and C. Pape: 1996, ‘Incremental Theory Reasoning Methods for Semantic Tableaux’. In: P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi (eds.): *Proc. 5th TABLEAUX, Terrasini/Palermo, Italy*, Vol. 1071 of *LNCS*. pp. 93–109.

Bibel, W., S. Brüning, U. Egly, and T. Rath: 1994, ‘KoMeT’. In: A. Bundy (ed.): *Proc. 12th CADE, Nancy, France*, Vol. 814 of *LNCS*. pp. 783–787.

Börger, E. and D. Rosenzweig: 1995, ‘The WAM—Definition and Compiler Correctness’. In: C. Beierle and L. Plümer (eds.): *Logic Programming: Formal Methods and Practical Applications*. North-Holland, pp. 21–90.

Boyer, R. and J. Moore: 1988, ‘Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic.’. *Machine Intelligence* **11**. Oxford University Press.

Boyer, R. S. and J. S. Moore: 1979, *A Computational Logic*. Academic Press.

Busch, H.: 1994, ‘First-Order Automation for Higher-Order-Logic Theorem Proving’. In: T. F. Melham and J. Camilleri (eds.): *Higher Order Logic Theorem Proving and its Applications: 7th International Workshop*, Vol. 859 of *LNCS*. pp. 97–112.

Constable, R. L., S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panagaden, J. T. Sasaki, and

S. F. Smith: 1986, *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall.

Dahn, B., J. Gehne, T. Honigmann, L. Walther, and A. Wolf: 1995, 'Integrating Logical Function with ILF'. System description, Humboldt-Universität zu Berlin.

Farmer, W. M., J. D. Guttman, and F. J. T. Fábrega: 1996, 'IMPS: an update description'. In: M. McRobbie and J. Slanley (eds.): *Proc. 13th CADE, New Brunswick/NJ, USA*. pp. 298–302.

Fermüller, C., A. Leitsch, T. Tammet, and N. Zamov: 1993, *Resolution Methods for the Decision Problem*, Vol. 679 of *LNCS*. Springer-Verlag.

Fitting, M.: 1990, *First-Order Logic and Automated Theorem Proving*. Springer, New York, first edition. Second edition appeared in 1996.

Goller, C., R. Letz, K. Mayr, and J. Schumann: 1994, 'SETHEO V3.2: Recent Developments – System Abstract'. In: A. Bundy (ed.): *Proc. 12th CADE, Nancy, France*, Vol. 814 of *LNCS*. pp. 778–782.

Gordon, M.: 1988, 'HOL: A Proof Generating System for Higher-order Logic'. In: G. Birtwistle and P. Subrahmanyam (eds.): *VLSI Specification and Synthesis*. Kluwer Academic Publishers.

Hähnle, R. and S. Klingenbeck: 1996, 'A-Ordered Tableaux'. *J. of Logic and Computation* **6**(6), 819–834.

Hähnle, R. and C. Pape: 1997, 'Ordered Tableaux: Extensions and Applications'. In: D. Galmiche (ed.): *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Pont-à-Mousson, France*, Vol. 1227 of *LNCS*. pp. 173–187.

Harrison, J. and L. Théry: 1993, 'Extending the HOL Theorem Prover with a Computer Algebra System to Reason about the Reals'. In: J. J. Joyce and C.-J. H. Seger (eds.): *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG'93, Vancouver, Canada*, Vol. 780 of *LNCS*. pp. 174–184.

Hutter, D., B. Langenstein, F. Koob, W. Reif, C. Sengler, W. Stephan, M. Ullmann, M. Wittmann, and A. Wolpers: 1995, 'The VSE Development Method - A Way to Engineer High-Assurance Software Systems'. In: B. Gotzheim (ed.): *GI/ITG Tagung Formale Beschreibungstechniken für verteilte Systeme.*

Hutter, D. and C. Sengler: 1996, 'INKA: The Next Generation'. In: M. McRobbie (ed.): *Proc. 13th CADE, New Brunswick/NJ, USA*, Vol. 1104 of *LNCS*. pp. 288–292.

Joyce, J. and C. Seger: 1994, 'The HOL-Voss System: Model-Checking inside a General-Purpose Theorem-Prover'. In: J. J. Joyce and C.-J. H. Seger (eds.): *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG'93*. pp. 185 – 198.

Kaufmann, M. and J. Moore: 1988, 'Design Goals of ACL2'. Technical report 101, Computational Logic Inc.

Letz, R., J. Schumann, S. Bayerl, and W. Bibel: 1992, 'SETHEO: A High-Perfomance Theorem Prover'. *J. of Automated Reasoning* **8**(2), 183–212.

Owre, S., S. Rajan, J. Rushby, N. Shankar, and M. Srivas: 1996, 'PVS: Combining Specification, Proof Checking, and Model Checking'. In: R. Alur and T. A.

Henzinger (eds.): *Computer-Aided Verification, CAV '96*, Vol. 1102 of *LNCS*. New Brunswick, NJ, pp. 411–414.

Owre, S., J. M. Rushby, and N. Shankar: 1992, 'PVS: A Prototype Verification System'. In: D. Kapur (ed.): *Proc. 11th CADE, Saratoga Springs/NY, USA*, Vol. 607 of *LNCS*. Saratoga, NY, pp. 748–752.

Paulson, L. C.: 1994, *Isabelle: A Generic Theorem Prover.* Springer LNCS 828.

Plaisted, D. A. and S. Greenbaum: 1986, 'A Structure-Preserving Clause Form Translation'. *Journal of Symbolic Computation* **2**, 293–304.

Reif, W.: 1995, 'The KIV-approach to Software Verification'. In: M. Broy and S. Jähnichen (eds.): *KORSO: Methods, Languages, and Tools for the Construction of Correct Software—Final Report*, LNCS 1009. Springer-Verlag.

Reif, W., G. Schellhorn, and K. Stenzel: 1997, 'Proving System Correctness with KIV 3.0'. In: *Proc. 14th CADE, Townsville, Australia*, Vol. 1249 of *LNCS*. pp. 69–72.

Reif, W. and K. Stenzel: 1993, 'Reuse of Proofs in Software Verification'. In: R. Shyamasundar (ed.): *Proc. FST TCS, Bombay, India*, Vol. 761 of *LNCS*. pp. 284–293.

Schellhorn, G. and W. Ahrendt: 1997, 'Reasoning about Abstract State Machines: The WAM Case Study'. *Journal of Universal Computer Science (JUCS)* **3**(4), 377–380. Available at the URL: http://hyperg.iicm.tu-graz.ac.at/jucs/.

Schmitt, P. H.: 1994, 'Proving WAM Compiler Correctness'. Interner Bericht 33/94, Universität Karlsruhe, Fakultät für Informatik.

Schneider, K., R. Kumar, and T. Kropf: 1992, 'Integrating a first-order automatic prover in the HOL environment'. In: *Higher Order Logic Theorem Proving and its Applications: 4th International Workshop, HUG'91*.

Stenz, G.: 1997, 'Beweistransformation in Gentzenkalkülen'. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

Strecker, M. and M. Sorea: 1997, 'Integrating an Equality Prover into a Software Development System based on Type Theory'. In: G. Brewka, C. Habel, and B. Nebel (eds.): *Proc. KI'97,* Vol. 1303 of *LNCS*. pp. 147–158.

Weidenbach, C., B. Gaede, and G. Rock: 1996, 'SPASS & FLOTTER, Version 0.42'. In: M. McRobbie and J. Slanley (eds.): *Proc. 13th CADE, New Brunswick/NJ, USA*, Vol. 1104 of *LNCS*. pp. 141–145.

Wos, L., R. Overbeek, E. Lusk, and J. Boyle: 1992, *Automated Reasoning, Introduction and Applications (2nd ed.).* McGraw Hill.