# Integrating Automated and Interactive Theorem Proving*

Wolfgang Ahrendt[1], Bernhard Beckert[1], Reiner Hähnle[1], Wolfram Menzel[1], Wolfgang Reif[2], Gerhard Schellhorn[2] and Peter Schmitt[1]

[1] Universität Karlsruhe, Inst. für Logik, Komplexität und Deduktionssysteme, D-76128 Karlsruhe

[2] Universität Ulm, Abt. Programmiermethodik, D-89069 Ulm

**Abstract.** This paper highlights a project to integrate interactive and automated theorem proving in Software Verification. Its aim is to combine the advantages of the two paradigms. We report on the integration concepts, and on the experimental results with a prototype implementation.

## 1 Introduction

For a long time, research in computer-aided reasoning was divided into two major branches: The interactive (or tactical) theorem proving community, and the supporters of fully automatic proof search. The two communities came up with powerful tools for different target groups, with different solutions, leading to different performance and application profiles.

In experiments with both interactive and automated theorem provers in Software Verification the following phenomenon can be observed: Automatic provers are very fast for the majority of the problems they can solve at all. When the problems become increasingly complex, response time grows superproportional. Beyond a certain problem size they cannot be applied reasonably. Interactive provers can be used even in very large case studies because of the cooperation with the user. However, most interactive provers require too many user interactions for small problems, particularly when combinatorial exhaustive search has to be performed.

Since strengths and weaknesses of the two paradigms are complementary, there is a current trend in application oriented deduction towards their integration. At the moment there are three approaches to integration: Some try to aggregate several automated provers under a common homogeneous user interface (e.g. ILF [8]). Others try to extend existing automated provers by an interactive component (e.g. INKA [17]).

Our approach explores the third way. We have investigated a conceptual integration of interactive and automated theorem proving for Software Verification that goes beyond a loose coupling of two proof systems. We have fixed a concrete application domain, because it turned out that the the application domain

* Research supported by DFG, grant No. Re 828/2-2.

has 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. In this paper we report on the integration concept, on the encountered problems, and on first experimental results with the prototype implementation. Furthermore we describe ongoing work.

The technical basis for the integration are the systems KIV [21] and $_3T^AP$ [3] both of which were developed in the research groups of the authors of this paper at Ulm and Karlsruhe. KIV is an advanced verification system which is applied in large realistic case studies in academia and industry for many years now. $_3T^AP$ 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.

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 we present the solutions we came up with so far.

Many of our decisions were 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. Although this case study is interesting in its own right, we mainly used it as a reference or benchmark. Parts of it are repeated every now and then to evaluate the success of our integration concepts. This case study is presented in Section 3.

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. Ongoing research on this and related topics is presented in Section 4. In Section 5 we draw conclusions.

## 2 Integration Concepts

### 2.1 Direct Application of $_3T^AP$ in KIV

Theorem proving with an interactive system typically proceeds by simplifying goals using backward reasoning with proof rules ("tactics"). Some proof rules may be applied automatically, but usually the tactics corresponding to the main line of argument in the proof must be supplied interactively. In order to allow the "proof engineer" to keep track of the proof state, 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, interactive theorem provers do not treat first-order goals specially: Interaction is required to prove them. Using an (ideal) automated theorem prover would relieve the proof engineer from a lot of interaction. Therefore, the scenario we considered was to use $_{3}T^{A}P$ as a "tactic" to prove first-order goals, thus exploiting its capability of fast combinatorial search. A suitable interface was implemented such that $_{3}T^{A}P$ 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 $_{3}T^{A}P$ links between potentially unifiable terms are computed, in many other theorem provers formulas are converted to clauses. We found that preprocessing 200 axioms with $_{3}T^{A}P$ takes about 30 secs. (the same holds for other provers we tested). Preprocessing the axioms at every proof attempt is clearly 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 for the "correctness management" of the interactive theorem prover, which prevents cycles in the dependencies of lemmas and invalidates only a minimal set of previous work when goals or specifications are changed.

**Inductive Theorems** Many theorems can only be proved inductively from the axioms, but most automated theorem provers (including $_{3}T^{A}P$) are not capable of finding inductive proofs.

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

**Domain Characteristics** In the application domain "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 semantics 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 $_3T^AP$ 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 $_3T^AP$ to proofs in the sequent calculus used in KIV.

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 to about 10 % of all theorems. 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 remainder of this section, which is organized as follows: The number of potentially relevant axioms in proving a goal is minimized by exploiting a specification's structure (Section 2.2). Measures we have taken to exploit the characteristics of software verification are presented next. Finally, in Section 2.6 details on converting proofs from $_3T^AP$ to KIV are given.

### 2.2 Reduction of the Axiom Set in KIV

Specifications in KIV are built up from elementary first-order theories with the usual operations of algebraic specification: union, enrichment, parameterization, actualization, and renaming. 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** $\Delta$" 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 [21].

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. It is sufficient, for example, to work merely with the axioms of the minimal specification whose signature contains that of the theorem. This results in drastic reduction for the WAM case study, because its specification hierarchy is flat (cf. Section 3).

### 2.3 Equality Handling

**Incremental Equality Reasoning** KIV specifications—as 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 [2] for solving $E$-unification problems extracted from tableau branches. This equality reasoner is much more efficient than just adding the equality axioms to the data base. 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 [4]. 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.

**Generating Precedence Orders from Simplifier Rules** As in most interactive theorem provers proof search in KIV is based on *simplifier rules*. 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 $\sigma$ to instances of $\tau$ if $\phi$ is provable.

The fact that simplifier rules have an operational semantics 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 rewrite rules 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, mostly fails due to conflicts such as the following (cons, car, cdr and append are the usual list operations):

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

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 $\tau$ can be homomorphically embedded into $\sigma$ (as is the case in the second rule above). We tested the resulting algorithm with five specifications from existing case studies in KIV, each having ca. 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 arised). 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 $_3T^AP$. For yet another use, see the following section.

Our considerations show that rules which are used to "simplify" terms in an intuitive sense in interactive theorem proving 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).

### 2.4 Restricting the Search Space by 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 [10] (although rarely used in practice). In order to employ such restrictions in $_3T^AP$, we could build on recent work on order-based refinements for tableau calculi [13, 14]. 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 [11]. They have a number of advantages that become particularly important in the context of software verification: They are defined for *unrestricted first-order formulas* [13] (in contrast to mere clausal normal form) and they are compatible with another important refinement called *regularity* [20]. It is possible to extend ordering restrictions to theory reasoning [14]. Moreover, ordered tableaux are *proof confluent* [19]: 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*, cf. Section 4.4 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 2.2) and to provide meaningful simplification orders for equality handling (Section 2.3).

Ordered tableaux can be characterized by restricting branch extension as follows: a formula $\phi$ can be used to extend a tableau branch $B$ iff either (i) $\phi$ has an order-maximal connection into $B$ (i.e., the connection literal of $\phi$ occurs order-maximally in $\phi$) or (ii) $\phi$ has an order-maximal connection into another input formula $\psi$ (i.e., the connection literals of both $\phi$ and $\psi$ occur order-maximally).

A partial order $<$ of the function and predicate symbols occurring in a problem can be naturally extended to an *A-order* $\prec$: a binary, irreflexive, transitive relation on atoms which is stable under substitutions. It suffices to stipulate for terms/atoms $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$ that $s \prec t$ iff (i) either $f < g$ or $f = g$, $n = m$, and $t_i \prec s_i$ for $1 \leq i \leq n$ *and* (ii) the variables of $s$ are a subset of the variables of $t$.

Thus $\prec$ can be *automatically* computed in such a way as to reflect the hierarchy of specifications and the implicit hierarchy of function symbols within each specification. $\prec$ often prevents literals from unrelated hierarchies to be maximal and, as a consequence, a formula $\phi$ has no maximal connection into a branch with only literals from a hierarchy unreachable from the maximal literals of $\phi$.

Even when $\prec$ 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.

### 2.5 Application-Specific Search Space Optimization

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 (dynamic logic) theorem. Each of these formula 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 to be 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 $_3T\!A\!P$ 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 $_3T\!A\!P$, but hidden to the user, protecting him from being confused by operational semantics.

## 2.6 Converting Proofs

Usually, automated and interactive theorem provers use strongly differing calculi, supporting machine-oriented proof search, respectively, the intuition of a human user. In an integrated system there are two alternatives for dealing with this gap. The first is to look for a monolithic method that constitutes a compromise between both sides' divergent requirements. Such a homogeneous solution does necessarily yield a system which is less adapted to the needs of the machine and the user. As can be seen in the sections above, this is not the alternative we choose.

Instead we use a dual approach, switching when required 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: What kind of information do both system parts have to exchange when they co-operate to construct a proof? Focusing on the *dynamic* information depending on a current goal,[3] the answer is roughly: KIV asks $_3T^AP$ to prove a goal and $_3T^AP$ responds with *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, yielding 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 which automatically invalidates proofs affected by the modification of a lemma. To embed 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 remaining valid branches 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. The technical details of this translation go far beyond the scope and size of this paper, thus we only mention its basic properties: (i) to strip whatever techniques automated provers use in order to restrict the search space (these techniques have no counterpart in the *interactively* used calculus, because they reduce legibility and are not required in a proof environment where the user guides the search); (ii) the transformation enlargens proofs by at most a constant factor by judicious introduction of *cuts* (which, on the other hand, are not allowed in the *automated* prover's calculus, where they would cause a search space explosion).

## 3 Compilation of Prolog to WAM

### 3.1 The Role of the WAM in the Project

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 [5], allowing us to concentrate on the development of a formal specification and on theorem proving. A first analysis [24] 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

[3] Examples for *static* information are: signature, axioms, lemmas and the specification structure.

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, ...) 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 following section is a short summary of the content of the case study and its results.

### 3.2 Content and Results of the WAM Case Study

Most Prolog implementations use the WAM [27] (or a variant of it) as an abstract target machine for compilation. Recently, correctness of the WAM-based Prolog compilation was mathematically analyzed by Börger & Rosenzweig [5].

As a starting point they formalized Prolog's semantics with an interpreter given as a *Gurevich Abstract State Machine* (formerly known as Evolving Algebra) [12]. This interpreter is then refined stepwise in altogether twelve steps, introducing more and more WAM concepts, until it serves as an interpreter of pure WAM code. In parallel, the Prolog program is compiled stepwise, which means that at intermediate levels it consists of WAM instructions interspersed with Prolog.

To verify selected compilation steps, we first specified the first six interpreters as imperative programs with 100–150 lines of code each. The data types in these programs were specified algebraically. Correctness of the interpreters was expressed as *program equivalence* in Dynamic Logic (the logic used in KIV to specify properties of imperative programs, see [15]).

As typical examples we then verified the first step, which transforms a Prolog search tree into a stack of "choicepoints", and the first proper compilation step (step four) with KIV. Details on the verification are given in [23, 1]. The equivalence proofs turned out to be very complex. Each proof requires a formula to describe the invariant part of the correspondence between states of the two programs (the *coupling invariant*), which covers about one page! Writing down a sufficient version *ad hoc* is just impossible. Instead, the correct version had to be developed by proof iterations. Whenever a proof attempt failed, the resulting unprovable goal told us what was missing in the coupling invariant and how it had to be improved. This led to an evolutionary process of completing the invariant by verification attempts. The most important feature for this technique was the powerful reuse of failed proofs as implemented in our system offering good support for the evolutionary verification process.

Verification revealed some formal gaps in the analysis of [5]. Moreover, the rule system for one of the interpreters was found to be indeterministic, which can cause non-termination, for example, for the Prolog program consisting of clauses "`p :- fail.`" and "`p.`" with the query "`?- p.`".

It took three man months of effort 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.

## 4 Ongoing Research

### 4.1 Tableau Rewriting Using Simplifiers

Simplifiers of the form $(\forall x)(p(x) \rightarrow \psi(x))$ are frequently used in KIV specifications. Typical examples are *definitions*, where the formula $\psi$ is used to specify the meaning of the predicate symbol $p$,[4] such as in $(\forall x)(even(x) \rightarrow (\exists y)(x = 2 * y))$.

Simplifiers carry pragmatic information: In proofs a simplifier is used solely to deduce an instance $\psi(t)$ of the conclusion from a given instance $p(t)$ of the premise, thus it is known (and part of the pragmatics of the simplifier) that the contrapositive $(\forall x)(\neg\psi(x) \rightarrow \neg p(x))$ 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.

When a simplifier is used in free-variable tableaux (where it is just an implication) to derive $\psi(t)$ from $p(t)$, then first two new branches are generated; one of these branches contains the formula $\neg p(x)$ and the other the formula $\psi(x)$, the first branch can be closed immediately, instantiating the free variable $x$ with $t$.

It is, however, much better to handle simplifiers differently using their pragmatic information: If, and only if, a branch $B$ contains $(\forall x)(p(x) \rightarrow \psi(x))$ (which must be classified as a simplifier) and a literal $p(t)$, the formula $\psi(t)$ is added to $B$. Thus, there is no need to generate a new branch. And, what is more important, a simplifier is only used, when a corresponding literal is present on the branch, such that unnecessary branching of the tableau is avoided.

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). The effects are even more drastic, if techniques similar to completion-based equality handling are used to make the set of simplifiers confluent. Then the conclusion $\psi(t)$ can *replace* the premise $p(t)$ on a branch.

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

### 4.2 Extending the Application Domain

Until now we have considered the use of automated theorem provers for goals ocurring 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, the validity of its condition in the present context has to be proved.
- Often, disjunctive goals containing programs can be proved by considering the first-order disjuncts only. To avoid the unnecessary dynamic logic proof attempts on such goals, KIV routinely checks for first-order validity.
- If the test (or its logical complement) of a conditional in a program is valid in a given context, then the else- (then-) branch needs not be proved.
- For recursive procedures KIV does a recursion analysis. This analysis involves to determine in a given context, whether the recursive call is reachable. This check is again a first-order goal.

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 no theorems. The ones which are should be filtered out quickly. The performance of the theorem prover becomes even more important, because a proof engineer is usually willing to wait a few seconds for the proof of a goal which he expects to be a theorem, but not for proof attempts on goals he expects (or even knows) to be non-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.

### 4.3 Proof Engineering

Beside a large 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 (apart from friendly heuristics and visualization of proof trees) are:

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

The importance of these tools reappeared in the incremental development of the “coupling invariant” in the WAM case study. Currently there exists a method in KIV to reuse proofs on program changes [22]. It is based on the idea to calculate the differences between proofs from the differences of programs. The correspondence between parts of a proof and parts of a program is given by the proof rules.

This method could not be applied often in the WAM case study, because programs were already given and only one major correction (to correct the "indeterminism" problem, see Section 3) was necessary. However, a lot of corrections were necessary to first-order formulas. Here the replay of proofs was often helpful as it already handles a lot of deviations between old (wrong) and new (corrected) goals. But still too many parts of proofs had to be redone following changes.

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.

## 4.4 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 in Section 4.2, 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 uttered explicitly) simplify the situation considerably: if a program contains bugs then only those occurring after finitely many steps on finite input are worth knowing. Thus, the corresponding counter examples are finite, too. In practice, counter examples deriving from buggy programs and specifications tend to be very small. So there is no need to deal with large terms while looking for them.

Several approaches to automated model building exist [26, 25, 7, 6, 18, 9], but none of them exploits further 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.

We believe that in our specific context a suitable extension (by equality and sorts) of the ground model generator MGTP [16] is most promising.

## 4.5 Concurrency

The integrated system described so far coordinates two sub-systems, an interactive and an automated theorem prover. In the current version, the two parts interact sequentially: KIV calls $_3T^AP$ and *waits* for an answer, at most until the timeout. The next step is to let both parts work in parallel. There are many possibilities for interleaving proof attempts. The most effective would surely be to exploit the time span in which the user thinks about his next decision on how to continue the proof. In the meantime the automated prover should try to close the current goal. Even several instances of an automated prover could work in parallel (maybe in a network) on all currently open goals of a proof. To evaluate the tradeoff between the complexity of realizing such an interface and the benefits of such a scenario, still a lot of experience has to be collected, which can only be gained by running case studies.

## 5 Conclusion

When 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 combining the two provers, that had been developed over the last years in our research teams, provided sufficient motivation to set to work with extra effort. But even in our most modest moments we had hoped to get more.

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 tactic provers, the use of pragmatic information to guide proof search. We have made substantial progress towards finding solutions, which again have significance beyond the special situation we are dealing with. We feel thus justified to call what we are doing not just combining *provers* but speak of integrating interactive and automatic theorem *proving*.

## References

1. W. Ahrendt. Von PROLOG zur WAM—Verifikation der Prozedurübersetzung mit KIV. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, Dec. 1995.
2. B. Beckert. A completion-based method for mixed universal and rigid $E$-unification. In A. Bundy, editor, *Proc. 12th CADE, Nancy, France*, LNCS 814, pages 678–692. Springer, 1994.
3. B. Beckert, R. Hähnle, P. Oel, and M. Sulzmann. The tableau-based theorem prover $_3T^AP$, version 4.0. In M. McRobbie, editor, *Proc. 13th CADE, New Brunswick/NJ, USA*, LNCS 1104, pages 303–307. Springer, 1996.
4. B. Beckert and C. Pape. Incremental theory reasoning methods for semantic tableaux. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Proc. 5th TABLEAUX, Terrasini/Palermo, Italy*, LNCS 1071, pages 93–109. Springer, 1996.
5. E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*. North-Holland, 1995.
6. R. Caferra and N. Peltier. Model building and interactive theory discovery. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Proc. 4th TABLEAUX, St. Goar, Germany*, LNCS 918, pages 154–168. Springer, 1995.

7. R. Caferra and N. Zabel. A tableaux method for systematic simultaneous search for refutations and models using equational problems. *J. of Logic and Computation*, 3(1):3–26, 1993.
8. B. Dahn, J. Gehne, T. Honigmann, L. Walther, and A. Wolf. Integrating logical function with ILF. System description, Humboldt-Universität zu Berlin, 1995.
9. C. Fermüller and A. Leitsch. Hyperresolution and automated model building. *J. of Logic and Computation*, 6(2), 1996.
10. C. Fermüller, A. Leitsch, T. Tammet, and N. Zamov. *Resolution Methods for the Decision Problem*. LNCS 679. Springer, 1993.
11. M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, New York, second edition, 1996.
12. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
13. R. Hähnle and S. Klingenbeck. A-ordered tableaux. *J. of Logic and Computation*, 6(6):819–834, 1996.
14. R. Hähnle and C. Pape. Ordered tableaux: Extensions and applications. Submitted, 1996.
15. D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984.
16. R. Hasegawa, M. Koshimura, and H. Fujita. MGTP: A parallel theorem prover based on lazy model generation. In D. Kapur, editor, *Proc. 11th CADE, Saratoga Springs/NY, USA*, LNCS 607, pages 776–780. Springer, 1992.
17. D. Hutter and C. Sengler. INKA: The next generation. In M. McRobbie, editor, *Proc. 13th CADE, New Brunswick/NJ, USA*, LNCS 1104, pages 288–292. Springer, 1996.
18. S. Klingenbeck. Generating finite counter examples with semantic tableaux. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Proc. 4th TABLEAUX, St. Goar, Germany*, LNCS 918, pages 31–46. Springer, 1995.
19. R. Letz. *First-Order Calculi and Proof Procedures for Automated Deduction*. PhD thesis, TH Darmstadt, June 1993.
20. R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A high-perfomance theorem prover. *J. of Automated Reasoning*, 8(2):183–212, 1992.
21. 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*, LNCS 1009. Springer, 1995.
22. W. Reif and K. Stenzel. Reuse of proofs in software verification. *SADHANA: Academy Proceedings in Engineering Sciences*, 21(2):229–244, 1996.
23. G. Schellhorn and W. Ahrendt. Verification of a Prolog compiler—first steps with KIV. Ulmer Informatik-Berichte 96-05, Univ. Ulm, Fakultät für Informatik, 1996.
24. P. H. Schmitt. Proving WAM compiler correctness. Interner Bericht 33/94, Universität Karlsruhe, Fakultät für Informatik, 1994.
25. J. Slaney. FINDER: finite domain enumerator. In A. Bundy, editor, *Proc. 12th CADE, Nancy, France*, LNCS 814, pages 798–801. Springer, 1994.
26. J. Slaney, M. Fujita, and M. Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. *Computers and Mathematics with Applications*, 1993.
27. D. Warren. An abstract Prolog instruction set. Technical Note 309, Artificial Intelligence Center, SRI International, 1983.