Tactical theorem proving in program verification

M. Heisel, Wolfgang Reif, W. Stephan

Angaben zur Veröffentlichung / Publication details:

Heisel, M., Wolfgang Reif, and W. Stephan. 1990. "Tactical theorem proving in program verification." In 10th International Conference on Automated Deduction: Kaiserslautern, FRG, July 24–27, 1990; proceedings, edited by Mark E. Stickel, 117–31. Berlin: Springer. https://doi.org/10.1007/3-540-52885-7_83.

Tactical Theorem Proving in Program Verification

M. Heisel W. Reif W. Stephan
Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Postfach 6980, D-7500 Karlsruhe

Abstract
A "shell" for program verification and development is presented which is obtained by using a variant of Dynamic Logic in combination with tactical theorem proving. This shell (Karlsruhe Interactive Verifier) allows easy implementations of various strategies which are guaranteed to be correct with respect to the basic logic. It is shown how tactical theorem proving is adapted for the above purposes.

1 Introduction

In this paper we describe how the paradigm of *tactical theorem proving* is used in the Karlsruhe Interactive Verifier (KIV) to realize a "verification shell" which allows the informed user to implement various verification and program development strategies within a common logical framework. These strategies can then be used without any deep knowledge of the KIV metalanguage (Proof Programming Language) or the underlying logic.

Starting with Edinburgh LCF, [GMW 79], a large number of systems have emerged in the last ten years that use tactical theorem proving. The common basic idea of all these systems is that they offer the user the possibility to *program extensions* of some basic formalism. They consist of a *metalanguage* that allows to represent and manipulate formulas, (proof-) rules, and whole proofs of the underlying logical system. Among other things, this metalanguage enables the user to write programs (tactics) which generate subgoals from a given goal. All the systems guarantee *soundness* by built-in mechanisms which make sure that all derivations obtained by using tactics can be transformed to proofs using only basic rules and axioms. Large proofs are typically generated in an *interactive* way. But since the user relies on a (possibly large) database of tactics and lemmata, these systems are far away from being simple proof-checkers.

Among the systems based on the paradigm of tactical theorem proving are the Cambridge LCF system, [Pa 87], which like Edinburgh LCF can be used for proofs about complex functional programs, the Nuprl system, [Co 86], for proofs in constructive type theory, the λProlog system, [FM 88], which extends Prolog by λ–terms, and the Oyster system, [Bun 89], which is similar to the Nuprl system and uses Boyer-Moore-like heuristics, [BM 79], to synthesize functional programs.

One major application of systems for tactical theorem proving has been the generation of completely formal versions of complex proofs. To perform this task the user is provided with a system, which as the authors of [Co 86] put it, "helps fill in the details and keeps track of the

proof obligations that must be met at the critical points of an argument." A good example of such a proof is given in [Mu 84]. In contrast to that, proofs about realistic programs rarely require deep mathematical results. Nearly the same holds for the derivation of special proof rules from a general axiomatization. We are trying to build an environment which makes proofs about individual programs as well as the development of entire proof strategies routine tasks.

In a first step the KIV system uses tactical theorem proving to derive special calculi (for program verification and development) from a powerful general logical formalism. The reduction of the logical building blocks of a special method, like for example Burstall's strategy for proving total correctness assertions, [Bur 74], [HRS 87], to the basic formalism is carried out completely automatically. Our main interest is to guarantee the soundness of the newly added rules and tactics, since as anyone will agree, a flexible verification system where one experiments with numerous proof-rules makes no sense if there is not a rigorous proof for each of them.

To people concerned with the verification or development of individual programs the KIV system offers a (growing) collection of strategies which leave to the user only the key decisions of a proof. Typically these strategies ask the user for invariants or inductive hypotheses. In most cases when the strategy has worked through a program some first-order verification conditions remain to be proven. In principle this can be done within the KIV system itself since we have implemented some versions of a theorem prover for first-order logic. However, it seems to be an interesting topic to combine our system with classical provers or simplifiers.

We use Dynamic Logic (DL) as the object language. Since DL allows formulas of the form $[\alpha]\varphi$, where α is a (PASCAL-like) program, this language exhibits a much richer syntactical structure than do most of the other formalisms. The soundness of rules and tactics very often depends on syntactic restrictions for the programs involved. Although no deep mathematical insight is required, without long and tedious proofs these restrictions are subject to subtle errors.

Most calculi for DL or Hoare-like languages known from the literature are designed only for one special proof strategy. We have therefore developed a technique for axiomatizing programming languages, see for example [HRS 89], which is flexible enough to serve as a basis for a universal verification system. This technique incorporates the basic ideas of *uninterpreted reasoning* as described for example in the book of Goldblatt, [Go 82]. Although proofs in this system, if carried out on paper, are completely intractable, it has turned out that in combination with tactical theorem proving our axiomatization can be used as a basis for all sorts of verificaction and development strategies.

The paper is organized as follows. In section two we outline the basic properties of our calculus and explain why this sort of axiomatization is needed for our purposes. In section three we introduce some technical details of our metalanguage PPL which are relevant for the rest of the paper. In section four and five we present two specific aspects of our system, the role of validations and the use of metavariables for the development of programs. We finish with a discussion of strategies, which are programs controlling the use of the basic logical building blocks of a method.

2 Dynamic Logic

2.1 Syntax, Semantics and Expressibility

Dynamic logic extends ordinary predicate logic (PL) by formulas $[\alpha]\varphi$ ("box α φ") and $\langle\alpha\rangle\varphi$ ("diamond α φ"), where α is a program, and φ is again a DL-formula. The intuitive meaning of

$[\alpha]\varphi$ is: "if α terminates, φ holds after execution of α ". The formula $\langle\alpha\rangle\varphi$ has to be read as: "α terminates and φ holds after execution of α ".

The programs that may occur in such program formulas are taken from the following PASCAL-like programming language. We consider skip, abort, assignments, conditionals, while loops, local variables and mutually recursive procedures. For local variables, procedure declarations, and procedure calls we use the following syntax:

$$\textbf{var } x_1{=}\tau_1 \ ; \ldots ; x_n{=}\tau_n \textbf{ in } \alpha$$

$$\textbf{proc } p_1 \Leftarrow \lambda(x_1{:}y_1).\alpha_1 \ ; \ldots ; p_n \Leftarrow \lambda(x_n{:}y_n).\alpha_n \textbf{ in } \alpha \qquad\qquad p(\tau{:}z).$$

In the *declaration of local variables* x_i is initialized to the value of τ_i. In the *procedure declaration* $\mathbf{x}_i$ and $\mathbf{y}_i$ are the lists of the *formal* value and variable parameters respectively and α_i is the procedure body. The construct $\lambda(\mathbf{x}_i{:}\mathbf{y}_i).\alpha_i$ is called an *abstraction*. In the *procedure call* $\boldsymbol{\tau}$ and $\mathbf{z}$ are the lists of the *actual* parameters. The occurrences of variables following the symbols "λ" and "**var**" are *binding occurrences*. The *scope* of these binding occurrences is defined in the usual way. An occurrence of a variable inside the scope of a binding occurrence of that variable is called *bound*.

In DL many interesting properties of programs are expressible. For PL-formulas φ and ψ, the formula $\varphi \rightarrow [\alpha]\psi$ corresponds to the well known partial correctness assertion $\varphi\{\alpha\}\psi$. Total correctness assertions, $\varphi \rightarrow \langle\alpha\rangle\psi$, are obtained by using diamond formulas instead of box formulas, and the equivalence of two programs α and β with respect to a program variable x is expressed by an equivalence between two diamond formulas: $\langle\alpha\rangle x{=}x' \leftrightarrow \langle\beta\rangle x{=}x'$. More general relations between programs can be expressed by implications between diamond formulas: $\langle\alpha\rangle\varphi \rightarrow \langle\beta\rangle\psi$. Such formulas are used, for instance, to express the correctness of the implementation of abstract data types.

2.2 Interpreted versus Uninterpreted Reasoning

For our aim, namely to provide a tool for implementing arbitrary verification and synthesis methods, the choice of the basic calculus plays a crucial role. It should be flexible enough to simulate many different "special purpose" calculi for specific verification methods. In the sequel we take a look at the calculi available from the literature and motivate our choice in the KIV system.

The difference between the known axiomatizations is what David Harel [Ha 79] calls *interpreted* versus *uninterpreted* reasoning. In the case of interpreted reasoning we have in mind a fixed interpretation of the data structures the programs operate upon. There exists a surprisingly simple axiomatization for interpreted DL due to Harel. The central idea is to reduce statements about programs to statements about the data. Harel proves this calculus to be arithmetically complete, i.e. the formulas valid in an arithmetical structure are exactly those derivable in the calculus from the first-order theory of that structure.

Besides the axioms for propositional logic, in Harel's calculus there are the substitution axiom for assignments and the axioms for compounds and conditionals:

$$[x{:=}\tau]\varphi \leftrightarrow \varphi_x^{\tau} \quad ,$$

where φ is a PL-formula such that x does not occur free in the scope of a quantifier which binds variables from τ. φ_x^{τ} denotes the formula obtained from φ by the substitution of τ for all free occurrences of x.

$[\alpha;\beta]\varphi \leftrightarrow [\alpha][\beta]\varphi$ $\quad$ $[\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta \textbf{ fi}]\varphi \leftrightarrow (\varepsilon \rightarrow [\alpha]\varphi) \wedge (\neg\varepsilon \rightarrow [\beta]\varphi)$

The inference rules are modus ponens, modal quantification, existential quantification, the invariance rule and the convergence rule:

$$\frac{\varphi(n+1) \rightarrow (\varepsilon \wedge \langle\alpha\rangle\varphi(n)) \;\;|\;\; \varphi(0) \rightarrow \neg\varepsilon}{\varphi(n) \rightarrow \langle \textbf{while } \varepsilon \textbf{ do } \alpha\rangle\, \varphi(0)}$$

, where $\varphi \in$ PL with free n, $n \notin \mathrm{Var}(\alpha)$.

The convergence rule relates the control structure to the data structure. The user has to supply a *convergence formula* $\varphi(n)$. Roughly, $\varphi(n)$ is a first-order description of what happens to the program variables if the loop continues for n further iterations. The convergence rule makes the calculus what it is, namely one for interpreted reasoning. It is the only rule which is sound only in arithmetical structures.

Although the calculus is simple and has nice theoretical properties, it is not suitable as the basis of a verification system because of some severe practical drawbacks. These are illustrated by the following example. Consider the two programs prog_1 and prog_2, which both operate on the integers:

prog_1 :
h:= x;
{while$_1$} **while** h≠0
do {bdy$_1$} y:= y+h; h:= h-1 **od**
{end of while$_1$}
h:= x;
{while$_2$} **while** h≠0
do {bdy$_2$} z:= h*z; h:= h-1 **od** {end of while$_2$}

prog_2 :
h:= x;
{while$_3$} **while** h≠0
do {bdy$_3$} y:= y+h; z:= h*z; h:= h-1 **od**
{end of while$_3$}

The first loop in prog_1 counts down h from x to 0, summing up the intermediate values of h in the variable y. Then the same is done in the second loop with multiplication and the variable z. Since the two loops do not interfere with each other we can merge them into one, which is slightly more efficient.

At this point probably every programmer would agree that the two programs are equivalent despite the sparse explanations given so far. This is obviously due to a programmer's familiarity with control structures. Formally, the equivalence can be expressed by $\langle \mathrm{prog}_1\rangle E \leftrightarrow \langle \mathrm{prog}_2\rangle E$, where $E \equiv h=h_0 \wedge x=x_0 \wedge y=y_0 \wedge z=z_0$.

A formal proof in the above system requires several applications of the convergence rule. The user has to supply the corresponding convergence formulas $\varphi(n)$. One of them is

$$\varphi(n) \equiv (\, h_0=0 \;\wedge\; h=n \wedge y_0=y+(\, n*(n+1)\,)/2 \;\wedge\; z_0=z*n! \;\wedge\; x=x_0\,) \;.$$

This application of the convergence rule leads to the following two verification conditions in the proof, which are both true in the integers:

$$h=n+1 \;\rightarrow\; (\,(n+1)*(n+2)\,)/2 = h + (\,n*(n+1)\,)/2\,)$$

$$h=n+1 \;\rightarrow\; (\,h*n! = (n+1)!\,) \;.$$

So far everything seems to be completely fine, but there is something special with this proof. First we left the original signature and introduced the factorial function and the integer division

by 2 in order to express the convergence formula. Furthermore, the verification conditions are very misleading. They suggest that the "deeper" reason for the equivalence of $prog_1$ and $prog_2$ is somehow related to the above properties of the factorial function and the sum of the first n integers. This impression is definitely wrong. The equivalence does not rely on any arithmetical property and even holds in any data structure. This means, that during the proof, verification conditions (proof obligations) arise, which stem from the proof technique and not from the theorem. From a practical point of view, this situation is very unsatisfactory. The calculus is based on one hard-wired proof technique, using invariants and convergents, and ignores knowledge about control structures.

To avoid this drawback, we have adopted a calculus for uninterpreted reasoning. In this case all the axioms and inference rules are sound in any data structure. The effects of programs are expressed in terms of other programs. The basic guideline for the following axiomatization is Goldblatt's system for uninterpreted reasoning [Go 82] and can be found in [Re 84], [St 85] and [HRS 89].

2.3 The Logic Underlying the KIV System

In addition to the above substitution axiom for assignments and the decomposition axioms for compounds and conditionals which are the same as in Harel's axiomatization, we have algebraic computation rules for assignments, like

$[x:=\tau][x:=\sigma]\varphi \rightarrow [x:=\sigma_x^{\tau}]\varphi$ and $[x:=\tau][y:=\sigma]\varphi \rightarrow [y:=\sigma_x^{\tau}][x:=\tau]\varphi$, where y not in $x:=\sigma$.

The first one, for instance, states that two assignments to x are the same as a single one with x replaced by τ in σ. In order to capture the while loop, we use an auxiliary counter structure which is not at the programmer's disposal. This counter structure with a constant *zero* and a successor function *next* is used to talk about the number of iterations of a while loop. The i-th approximation of a while loop, (**while** ε **do** α **od**$\downarrow$i), is a program which simulates the loop iff the loop terminates after exactly i iterations. A statement about a loop then holds iff it holds for every such approximation. Consequently, statements about loops are proved by induction on the approximations. The axiom for the while-loop is:

$$[\textbf{while } \varepsilon \textbf{ do } \alpha \textbf{ od}]\varphi \leftrightarrow \forall i.\, [\textbf{while } \varepsilon \textbf{ do } \alpha \textbf{ od}\downarrow i]\varphi$$

For the i-th approximation of a while loop we have the following schemes:

$$[\textbf{while } \varepsilon \textbf{ do } \alpha \textbf{ od}\downarrow \text{zero}]\varphi \leftrightarrow \neg\varepsilon \rightarrow \varphi$$

$$[\textbf{while } \varepsilon \textbf{ do } \alpha \textbf{ od}\downarrow \text{next}(i)]\varphi \leftrightarrow [\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else abort fi}][\textbf{while } \varepsilon \textbf{ do } \alpha \textbf{ od}\downarrow i]\varphi$$

For recursive procedures, we also add an auxiliary structure to the logic, namely environments, which are tree-like objects. The actual axiomatization captures higher-order procedures. Here we only consider the simpler case where there are only call-by-value and call-by-reference parameters. In this case environments are only used to store procedure definitions. Now, commands are always considered together with an environment (e | α) (e omitted if irrelevant). Environments are needed to talk about the fixed-point environment e' with respect to some current environment e and a set of new mutually recursive declarations δ. The n-th approximation of the fixed-point environment with respect to e and δ is an environment term, able to simulate the behaviour of the new definitions iff during execution there are no more than n recursive calls of the procedures which are declared in δ. The predicate $Ap_\delta(e,e')$ is true if e' is an approximation of the fixed-point environment with respect to e and δ. A statement about a procedure declaration evaluated in some environment e holds iff it holds for the main program in every ap-

proximation of the fixed-point environment. Consequently, statements about declarations are proved by induction on environments. This is reflected by the following axioms for procedure declarations and calls :

$$[(e \mid \textbf{proc}\ \delta\ \textbf{in}\ \alpha\ )]\varphi \ \leftrightarrow\ \forall e'.\ (\ Ap_{\delta}\ (e, e') \ \rightarrow\ [(e' \mid \alpha)]\varphi\ )$$

$$[(e[p/(e_1 \mid \lambda(\mathbf{x}{:}\mathbf{y}).\alpha)] \mid p(\tau{:}\mathbf{z}))]\varphi \ \leftrightarrow\ [u_1{:=}\tau_1] \ldots [u_n{:=}\tau_n][(e_1 \mid \alpha'^{\mathbf{u}}_{\mathbf{x}\,\mathbf{y}\,\mathbf{z}})]\varphi\ ,$$

where $\mathbf{u} \equiv (u_1,\ldots,u_n)$ is a vector of pairwise distinct fresh variables and α' is obtained from α by a suitable renaming of the bound variables in α in order to avoid a binding conflict when the elements of $\mathbf{z}$ are substituted for all free occurrences of the elements of $\mathbf{y}$. Note that we use the inner environment e_1 for α'. The expression $e[p/(e_1 \mid \lambda(\mathbf{x}{:}\mathbf{y}).\alpha)]$ denotes an environment which is like e except that p is bound to (the closure) $(e_1 \mid \lambda(\mathbf{x}{:}\mathbf{y}).\alpha)$. If there is no entry for p in the given environment a call of p leads to abortion.

2.4 The Example Revisited

In the rest of the section we sketch the top level steps of a proof of the above example. We write $\alpha =_{\{x_1,\ldots,x_n\}} \beta$ if α and β terminate on the same inputs and produce the same outputs with respect to the variables $\{x_1,\ldots,x_n\}$. The notation is an abbreviation for the DL formula $\langle\alpha\rangle E \leftrightarrow \langle\beta\rangle E$, where $E \equiv (x_1{=}x_1') \wedge \ldots \wedge (x_n{=}x_n')$. In this notation the claim takes the form $prog_1 =_{\{x,y,h,z\}} prog_2$. Suppose we already knew the following lemmata:

1) $while_1 =_{\{x,y\}} while_3$ and 2) $while_2 =_{\{x,h,z\}} while_3$.

Then we may argue as follows: The first program does the same as its first half with respect to x and y. This yields $prog_1 =_{\{x,y\}} (h{:=}x\ ;\ while_1)$. We apply lemma 1 and replace $while_1$ by $while_3$ getting $prog_1 =_{\{x,y\}} (h{:=}x\ ;\ while_3)$. The right-hand side is already $prog_2$, giving $prog_1 =_{\{x,y\}} prog_2$. Analogously the first program does the same as its second half with respect to the variables x, h and z. This yields $prog_1 =_{\{x,h,z\}} (h{:=}x\ ;\ while_2)$ We apply lemma 2 and replace $while_2$ by $while_3$ getting $prog_1 =_{\{x,h,z\}} (h{:=}x\ ;\ while_3)$. Again the right-hand side is already $prog_2$ and hence $prog_1 =_{\{x,h,z\}} prog_2$. Together we have $prog_1 =_{\{x,y,h,z\}} prog_2$. Finally, we prove the two lemmata by induction on the number of iterations.

Of course, behind every proof step above, there is hidden a number of other proof steps. But neither the ones shown nor those that have been omitted involve any property of the integers.

2.5 Simulation of Other Logics

Besides the possibility of reasoning on the level of control structures, like in our example above, the main advantage of a calculus for uninterpreted reasoning is its flexibility and its deductive strength. By adding nonlogical DL-axioms, many known calculi for specific verification methods can be simulated with our logic. Among these calculi are, for instance, Harel's calculus for Deterministic Dynamic Logic mentioned above, Hoare's logic, Manna's system for total correctness and Burstall's intermittent assertion method (see [HRS 89] for more details). This flexibility is an important premise for building a verification tool like the KIV system.

3 PPL

The KIV system is based on a *sequent calculus*. The most important data structure of PPL are *proof trees* whose nodes consist of sequents. The proven sequent is the root of the tree and

the leaves are either axioms or premises. Thus, proof trees represent *partial proofs*.

It is an important feature of PPL that we allow schematic objects: There are *metavariables* for each syntactic class. Metavariables will be denoted by names prefixed with "$", while we continue to use greek letters for arbitrary syntactical objects which may or may not contain metavariables. So, for example, we have the following axiom scheme for the sequential composition of programs

$$\vdash \quad [\$C1 ; \$C2]\$F \leftrightarrow [\$C1]\,[\$C2]\$F \quad .$$

From the context, the system knows that $C1 and $C2 are metavariables for commands while $F is a metavariable for a formula.

Apart from the axioms for the programming language, which are all of the form as the one given above, we have the usual rule schemes, like

$$\frac{\$G \vdash \$F1 , \$D \qquad | \qquad \$G \vdash \$F2 , \$D}{\$G \vdash \$F1 \wedge \$F2 , \$D}$$

If a *basic rule* has an associated variable condition (quantifiers and modalities) this is checked by built-in programs. Rules are proof trees of height one. New proof trees can be generated by operations called *infer* $(\tau, [i_1,..,i_n], [\tau_1,.., \tau_n])$ and *refine* (σ_1, i, σ_2). The infer operation performs a forward proof step by using the conclusions of n proof trees $\tau_1,\ldots,\tau_n$ as premises of a proof tree τ yielding a new conclusion. The refine operation performs a backward proof step by replacing one premise of a proof tree σ_1 by a proof tree σ_2 yielding the premises of σ_2, as new subgoals. Both operations use matching: the proof trees τ and σ_2 can be considered as generalized inference rules and thus are instantiated by applying a matcher Θ to some of the metavariables. Θ satisfies the following conditions: the i_k-th premise of $\Theta(\tau)$ is equal to the conclusion of τ_k for k=1,...,n (infer), or the conclusion of $\Theta(\sigma_2)$ is equal to the i-th premise of σ_1 (refine). We use matching instead of unification to avoid undesirable instantiations of metavariables, for instance in equivalence preserving proof steps. PPL allows an arbitrary combination of forward and backward proof steps.

In addition to the basic rules the user may define *derived rules*. Derived rules have to be supplied with a *validation* which is either a proof tree which fits the rule or a PPL-program. This program, when started by the command validate_rule, takes as arguments the premises and conclusion of the rule, which may have been instantiated in the meantime, and tries to generate a fitting proof tree. Usually, to prevent the validation from *failing*, it will be necessary to instantiate (by infer or refine) at least some of the metavariables occurring in the rule. Even if there are no open premises, a proof tree does not represent a proof as long as there are validations which have not yet been executed. The use of derived rules has turned out to be a good means to structure proofs. For our backtracking facilities see section 6.

4 Validations

As mentioned above, derived (or user-defined) rules which are the basic building blocks of our tactics have to be supplied with a validation. In many cases, the rule schemes are not supposed to be valid for all instantiations. Validations are therefore used to check various *additional constraints*.

Since programs are part of our formulas, we are faced with a much richer syntax than in most other formalisms. Especially in the case of rules for recursive procedures it has turned out that we have to handle very complex variable conditions which can be shown to be sufficient only by formal proofs. Indeed, many of our first guesses about these conditions were wrong. The same holds for some rules we adopted from the literature. Recall also that we are talking about a system where the design (or redesign) of proof-rules is a frequent activity. Our solution is to abandon the explicit formulation of variable conditions (except for expositional purposes) and instead to write validations that establish these constraints a posteriori. If a validation terminates without failure it generates a proof for the instance in question.

It has turned out that validations for these purposes can be implemented in a relatively short time and that they work with reasonable efficiency as compared to standard test programs, written for example in LISP, which however are subject to logical errors. What one needs to obtain well structured solutions is a whole hierarchy of derived rules together with their validations.

A characteristic feature of uninterpreted reasoning is that we manipulate programs in the same way as we use algebraic laws to transform expressions. The implication

$$\vdash \quad [x:=\tau][\alpha]\,\varphi \;\rightarrow\; [\alpha_x{}^{\tau}][x:=\tau]\,\varphi$$

is needed in many validations that check variable conditions. For commands α without declarations, $\alpha_x{}^{\tau}$ denotes the command which results from α by substituting τ for all occurrences of x in expressions in α. In addition in α there must be no assignments to variables from $x:=\tau$. In the following we describe the implementation of a function *commute*, that, given (concrete) x, τ, α, and φ, generates a sound instance of this lemma (rule without premises) together with its validation. If the condition is not satisfied, *commute* fails. We outline the proof generated by the validation.

This validation can be structured like a metalevel proof which uses *induction on the structure of commands*, as it is the case with many others. Instead of using the inductive hypothesis we apply the rule recursively to subcommands. Each layer in the proof tree generated by the validation then corresponds to the inductive step according to the shape of the given command.

We start with the axiom

$$[x:=\tau][\alpha]\,\varphi \;\vdash\; [x:=\tau][\alpha]\,\varphi \quad .$$

In the case where $\alpha \equiv (\beta_0 ; \beta_1)$ we transform the composition axiom into

$$\frac{\$G \;\vdash\; [\$C1;\$C2]\;\$F}{\$G \;\vdash\; [\$C1][\$C2]\;\$F}$$

by the function *mk_srule*. From

$[x:=\tau][\beta_0;\beta_1]\,\varphi \;\vdash\; [x:=\tau][\beta_0][\beta_1]\,\varphi$ we get $[x:=\tau][\beta_0;\beta_1]\,\varphi \;\vdash\; [\beta_{0\,x}{}^{\tau}][x:=\tau][\beta_1]\,\varphi$

by using *mk_srule*(*commute*(x, τ, β_0,φ)) . The function *prefix*, given a list of commands $(\alpha_0 , .. , \alpha_n)$ as a first argument, transforms an implication

$$\vdash \quad \varphi \rightarrow \psi \qquad \text{into} \qquad \vdash \quad [\alpha_0] \ldots [\alpha_n]\,\varphi \;\rightarrow\; [\alpha_0] \ldots [\alpha_n]\,\psi \quad .$$

An application of the rule *mk_srule*(*prefix*($[\beta_{0\,x}{}^{\tau}]$, (*commute*(x, τ, β_1, φ))) yields

$$[x:=\tau][\beta_0;\beta_1]\,\varphi \;\vdash\; [\beta_{0\,x}{}^{\tau};\, \beta_{1\,x}{}^{\tau}][x:=\tau]\,\varphi \quad .$$

In the case where $\alpha \equiv$ **while** ε **do** β **od** we have to use *induction on counters*. In the inductive step we have to derive the sequent

$[x:=\tau][\textbf{while } \varepsilon \textbf{ do } \beta \textbf{ od}\downarrow next(i)]\ \varphi$, Hyp $\vdash$ $[\textbf{while } \varepsilon_x^{\tau} \textbf{ do } \beta_x^{\tau} \textbf{ od}\downarrow next(i)][x:=\tau]\ \varphi$,

where for $\{y_1, .. ,y_n\} = Var([x:=\tau][\textbf{while } \varepsilon \textbf{ do } \beta \textbf{ od}]\ \varphi)$, Hyp is the formula

$\forall(y_1, .. ,y_n).\ ([x:=\tau][\textbf{while } \varepsilon \textbf{ do } \beta \textbf{ od}\downarrow i]\ \varphi \rightarrow [\textbf{while } \varepsilon_x^{\tau} \textbf{ do } \beta_x^{\tau} \textbf{ od}\downarrow i][x:=\tau]\ \varphi)$.

Using the scheme for approximations (see section 2) we immediately get

$[x:=\tau][\textbf{while } \varepsilon \textbf{ do } \beta \textbf{ od}\downarrow next(i)]\ \varphi$, Hyp $\vdash$

$[x:=\tau][\textbf{if } \varepsilon \textbf{ then } \beta \textbf{ else abort fi}]\ [\textbf{while } \varepsilon \textbf{ do } \beta \textbf{ od}\downarrow i][x:=\tau]\ \varphi$.

By the rules for the conditional and *mk_srule*(*commute*(x, τ, β,φ)) we get

$[x:=\tau][\textbf{while } \varepsilon \textbf{ do } \beta \textbf{ od}\downarrow next(i)]\ \varphi$, Hyp $\vdash$

$[\textbf{if } \varepsilon_x^{\tau} \textbf{ then } \beta_x^{\tau} \textbf{ else abort fi}][x:=\tau][\textbf{while } \varepsilon \textbf{ do } \beta \textbf{ od}\downarrow i][x:=\tau]\ \varphi$.

By a rather special rule for applying the inductive hypothesis we then get the desired result.

The condition that there must be no assignments to variables in $x:=\tau$ is checked in the proof for the case where $\alpha \equiv y:=\sigma$. Here we use the second axiom given in section 2.3. Variable conditions for basic rules are implemented directly in LISP.

5 Metavariables

A very special feature of the KIV system are the *metavariables*. Sequents may contain metavariables for any syntactic category like formulas, terms, programs and so forth. They act as placeholders which may become instantiated later. This feature allows to specify and to prove schematic statements. For instance, the basic rules schemes of our calculus are represented as schematic proof trees of height one, and, conversely, each schematic proof tree can be used as an inference rule.

In the KIV system metavariables are used for several purposes: Statements can be generalized by inserting metavariables for subexpressions which are irrelevant for the proof. Thus the structure of the statement becomes simpler and the search space of the proof is reduced. Metavariables are also used to postpone decisions. For example, an invariant required in Dijkstra's method for program construction (see below) need not be given as a whole beforehand. A part of it is given immediately when the corresponding rule is applied, but the yet unknown rest of it is postponed by inserting a metavariable instead (cf. the metavariable $rest below). Finally, metavariables are one of the important concepts in the KIV system that enable the integration of program verification and program development without any changes to the logic or to the system itself. With the help of a metavariable we may express the specification of a program as a total correctness assertion $\Gamma \vdash \langle \$C \rangle \psi$ with known preconditions Γ and post-condition ψ but a yet unknown program $C. In the rest of this section we will concentrate on this use of metavariables.

5.1 Developing Programs by Instantiating Metavariables

Program development in the KIV system means to construct a particular proof tree: We start with a specification like the one above containing a metavariable for the yet unknown program. The application of proof refinements gradually leads to a series of new proof trees resulting in a

stepwise instantiation of the metavariables. We stop if there are no further uninstantiated metavariables in the proof. Then the conclusion of the proof tree is the total correctness assertion with the original pre- and postconditions but with the generated program instead of the metavariable. The premises of the proof tree are first-order verification conditions entailing the conclusion.

There are two types of proof refinements applied to the specification and the intermediate proof trees: (i) operations to instantiate a metavariable with a new code fragment possibly containing other metavariables and (ii) operations which reduce a goal containing such a code fragment to sufficient subgoals (subspecifications for the remaining metavariables in it). The central operation of type (i) is the *specialization of proofs* which takes a proof tree, a metavariable and an expression of the same syntactic category and yields an instance of the given proof where the metavariable is replaced by the expression throughout the proof. The operations of type (ii) are refinements with verification rules for the various PASCAL-like instructions.

5.2 An Example

Due to space limitations we will not display the applied rules. They are standard, and a rough understanding of what they do is enough for the purpose of this example.

Suppose we wanted to develop a function computing the length of a list. The initial specification is

$$\vdash \langle \textbf{proc}\ \mathrm{lg} \Leftarrow \lambda(\mathrm{l} : \mathrm{n}).\ \$C\ \textbf{in}\ \mathrm{lg}(\mathrm{l} : \mathrm{n})\rangle\ \mathrm{n} = \mathrm{length}(\mathrm{l})$$

where the implementation of the procedure body is left open. We decide to develop a recursive version of lg and apply the *induction rule* (with respect to the input list l) although we do not know all the details by now. Applying the rule yields a proof tree of height one with the above specification at the root and the following subgoal as the only premise

$$\mathrm{Ind}[\$C] \vdash \langle \textbf{proc}\ \mathrm{lg} \Leftarrow \lambda(\mathrm{l} : \mathrm{n}).\ \$C\ \textbf{in}\ \mathrm{lg}(\mathrm{l} : \mathrm{n})\rangle\ \mathrm{n} = \mathrm{length}(\mathrm{l}).$$

Ind[$C] is the induction hypothesis and the notation displays the metavariables occurring in it.

$$\mathrm{Ind}[\$C] \equiv \forall(\mathrm{l}', \mathrm{n}).(\mathrm{l}' < \mathrm{l} \rightarrow \langle \textbf{proc}\ \mathrm{lg} \Leftarrow \lambda(\mathrm{l} : \mathrm{n}).\ \$C\ \textbf{in}\ \mathrm{lg}(\mathrm{l}' : \mathrm{n})\rangle\ \mathrm{n} = \mathrm{length}(\mathrm{l}'))$$

Now we apply the verification *rule for procedure calls* as a reduction rule and get a proof tree of height two with the original specification at the root and the following premise

$$\mathrm{Ind}[\$C] \vdash \langle \textbf{proc}\ \mathrm{lg} \Leftarrow \lambda(\mathrm{l} : \mathrm{n}).\ \$C\ \textbf{in}\ \$C\rangle\ \mathrm{n} = \mathrm{length}(\mathrm{l}).$$

It is noteworthy that with most of the rules there is associated a variable check or a validation. However, these checks cannot be performed immediately, since the sequents still contain unspecified parts. Here we make use of the possibility to postpone the validations until the end of the proof. In our example we decide to *specialize the current proof tree* by inserting a conditional for $C. All the occurrences of $C in the current proof tree are replaced by **if** l = nil **then** $B **else** $G **fi**. This yields

$$\mathrm{Ind}[\$B, \$G] \vdash \langle \textbf{proc}\ \mathrm{lg} \Leftarrow \lambda(\mathrm{l} : \mathrm{n}).\ \textbf{if}\ \mathrm{l} = \mathrm{nil}\ \textbf{then}\ \$B\ \textbf{else}\ \$G\ \textbf{fi}\ \textbf{in}\ \textbf{if}\ \mathrm{l} = \mathrm{nil}\ \textbf{then}\ \$B\ \textbf{else}\ \$G\ \textbf{fi}\rangle\ \mathrm{n} = \mathrm{length}(\mathrm{l}).$$

This goal can be reduced by the *splitting rule for conditionals*. The proof tree branches and we get two premises (the above procedure body is abbreviated by bdy)

$$\mathrm{Ind}[\$B, \$G], \mathrm{l} = \mathrm{nil} \vdash \langle \textbf{proc}\ \mathrm{lg} \Leftarrow \lambda(\mathrm{l} : \mathrm{n}).\mathrm{bdy}\ \textbf{in}\ \$B\rangle\ \mathrm{n} = \mathrm{length}(\mathrm{l})$$

$$\mathrm{Ind}[\$B, \$G], \mathrm{l} \neq \mathrm{nil} \vdash \langle \textbf{proc}\ \mathrm{lg} \Leftarrow \lambda(\mathrm{l} : \mathrm{n}).\mathrm{bdy}\ \textbf{in}\ \$G\rangle\ \mathrm{n} = \mathrm{length}(\mathrm{l}).$$

We now deal with the first one. We specialize the proof by inserting n:=0 for $B and apply the *assignment reduction* rule to the first premise. This results in the verification condition

$$l = nil \vdash \quad 0 = length(l).$$

Note that replacing $B also affects the second subgoal since $B occurs on its left hand side and in bdy. For the second subgoal we specialize the proof by inserting (lg(cdr(l) : n); $D) for $G and apply the decomposition *rule for compound statements*. The result is

$$Ind[\$D], l \neq nil \vdash \langle \textbf{proc} \ldots \textbf{in } lg(cdr(l) : n)\rangle\langle \textbf{proc} \ldots \textbf{in } \$D\rangle\ n = length(l).$$

Applying the *induction hypothesis* yields

$$Ind[\$D], l \neq nil, n = length(cdr(l)) \vdash \langle \textbf{proc} \ldots \textbf{in } \$D\rangle\ n = length(l).$$

Finally we specialize the proof once more and insert n:= n+1 for $D. Reducing the assignment yields

$$l \neq nil, n = length(cdr(l)) \vdash \quad n+1 = length(l).$$

The two remaining subgoals are first-order verification conditions and do not contain any meta-variables. Therefore we are done and the root of the tree contains the generated program

$$\vdash \langle \textbf{proc } lg \Leftarrow \lambda(l : n).\ \textbf{if } l=nil \textbf{ then } n:=0 \textbf{ else } lg(cdr(l) : n)\ ;\ n:=n+1 \textbf{ fi}$$
$$\textbf{in } lg(l : n)\rangle\ n = length(l).$$

Since every metavariable is instantiated now, we may execute the validations postponed so far. In our example the validations are successful. If they were not, we had to backtrack and revise our program decisions that led to the failure.

6 Strategies

Strategies are top-level programs that provide the control structure of a proof method by calling substrategies and tactics in a certain order. They guide an uninformed user, i.e. a user without specific knowledge about the system and its logic, through proofs according to the special verification or development method. Strategies embody the interaction with the user and heuristics for the selection of rules and tactics as well as for backtracking.

Typically the user is required to provide loop invariants, induction hypotheses, or convergence formulas. Depending on the strategy it may also happen that the system needs help to decide what to do next. Especially in the case of program development, where no (complete) program is available, this situation occurs frequently.

Most verification strategies are syntax-directed in that they rely heavily on the form of the given program. Although the search space is typically much smaller than in other areas, conflicts arise. We have to decide what goal to tackle next and which rule or tactic to apply to it.

In addition to methods for the selection of goals and tactics, strategies contain specific ways of doing backtracking and proof search.

6.1 Implemented Strategies

In the following, we give a list of the proof methods implemented in the KIV system, where we also mention the special heuristics involved, if there are any.

* A (complete) proof procedure for propositional logic is implemented, which does not make use of any structural rules.
* There are several versions of a prover for predicate logic, for instance with or without equality, complete or terminating. Heuristics are taken from [OS 88].
* There is a comfortable interactive implementation of Hoare's calculus: user guidance and backtracking facilities provided by the system enable the interactive development of proofs. For example, this has the advantage that the user does not need to know all the loop invariants in

advance, but can experiment with several possibilities.

* We have implemented Bergstra's and Klop's method for proving program inclusion and equivalence, which is published in [BK 84]. Given a first-order axiomatization Ax of a data structure and two programs α und β the method proves that α is equivalent to β in every model of Ax by showing:

$$Ax' \vdash_{\text{HOARE}} \varphi \rightarrow [\alpha]\psi \;\Leftrightarrow\; Ax' \vdash_{\text{HOARE}} \varphi \rightarrow [\beta]\psi$$

for all conservative refinements Ax' of Ax and all formulas φ and ψ.

* An implementation of Burstall's method [Bur 74], [HRS 87] allows to verify programs that implement inherently recursive functions by loops. Given the goal $\Gamma \vdash \langle\alpha;\beta\rangle\varphi$, the strategy eliminates α and computes a new goal of the shape $\Gamma' \vdash \langle\beta\rangle\varphi$. But in contrast to Hoare's calculus, loops are not eliminated by proving an invariance relation, but by structural induction on the data structure involved. Heuristics for this method have been developed in [Hä 87].

* As an example for a program development method we implemented Gries's method [GR 81], [He 89]. The program works like a structural editor for programs: Given a precondition Γ and a postcondition φ, a goal $\Gamma \vdash \langle\$C\rangle\varphi$ is created, where \$C is a metavariable standing for the program to be developed. This metavariable gets more and more instantiated until it is a concrete program. The heuristics involved in this strategy can also be found in [GR 81].

* Based on the implementation of Gries's method, there is the implementation of a strategy (introduced by an example by Dijkstra [Di 88]) which is capable of generating certain programs completely automatically. This strategy is treated below in more detail.

6.2 General Control Problems

Although the strategies mentioned above are very different, we can state some control problems they have in common.

How to Build up Proof Trees

There are two possibilities to build up proof trees: depth-first or breadth-first. Depth-first means that one branch of the proof tree has be be built up completely before another branch is worked on. This approach is useful for program development, since it corresponds to top-down program construction. In program verification, however, one might want to switch the branches to work on in order to be able to prove goals that are considered as critical first. In this case the proof tree is built up in layers. These two different approaches are also reflected in the backtracking structure of the respective strategy.

Backtracking

In all strategies that are interactive or that involve heuristics, the possibility of backtracking should be included. In PPL, there are two backtracking constructs, **or** and **org** (global **or**). e_1 **or** e_2 is evaluated as follows: First evaluate e_1. If this was successful, then this is the value of e_1 **or** e_2. Otherwise the value of e_1 **or** e_2 is the value of e_2. The **or**-construct is equivalent to **?** in ML [GMW 79]. It is sufficient if the strategy builds up proof trees in a depth-first manner. The advantage of this backtracking construct is that it can be implemented rather efficiently, because only the state before evaluation of e_1 has to be stored. Also, the use of **or** leads to readable programs.

However, the **or**-construct has one disadvantage: once the evaluation of e_1 was successful, there is no possibility to evaluate e_2 any more. This may be required when the proof tree is built

up breadth-first, giving the user the possibility to experiment, e.g. with different rules or different invariants or induction hypotheses. In such a situation the user might want to undo some steps and return to a former state by explicitly invoking a failure. Thus a backtracking construct having a "memory" is needed. This is the **org**-construct, which works as follows: first evaluate e_1. If this fails, the value of e_1 **org** e_2 is the value of e_2. Otherwise return the value of e_1, but put the current environment together with e_2 onto a stack of open alternatives. If a failure occurs, then the dynamically last alternative is tried, as opposed to the syntactically last alternative, which is tried by **or**. There is no ML-equivalent to **org**.

Proofs as a Means for Resolving Conflicts

In the implementation of every strategy one has to decide how to treat goals of predicate logic. One possibility is to leave them as verification conditions and prove them after the strategy has terminated. This approach has been taken e.g. in the implementation of Gries's method. However, if one wants to try several ways to proceed, it can be useful to apply a rule, try to prove some of the simpler goals created, and undo the rule application if the subgoals in question could not be proved. One example where this is the case is the verification of a loop in Hoare's logic. Before the invariance of a suggested formula is shown, it should be proven that the formula is not too weak, i.e. that the formula together with the negation of the loop condition implies the postcondition.

6.3 A Completely Automatic Strategy

In the case of development strategies it is sometimes possible to fix the overall structure of the proof. In this situation, the program constructs are selected such that the subgoals yielded by that strategy become provable. As an example we present a strategy which completely automatically generates programs for specifications of the following shape:

$$\Phi(0) = b \qquad \Phi(k+1) = \vartheta(\Phi(k), \Psi(k)),$$

where ϑ is an elementary operation and Ψ does not depend on Φ.

The basic ideas of this strategy go back to an example presented by E. W. Dijkstra [Di 88]. The task was to develop a program that computes n^3 for a given $n \geq 0$ without using multiplication. The idea is to successively compute $1^3, 2^3, \ldots, n^3$, such that $(i+1)^3$ is computed from i^3 using only addition (the preliminary loop invariant is $0 \leq i \leq n \wedge x_0 = i^3$). Since $(i+1)^3 - i^3 = 3i^2 + 3i + 1$ cannot be computed without multiplication, a new variable a is introduced with invariance property $x_1 = 3i^2 + 3i + 1$. The invariant is augmented by this conjunct. The new variable has to be initialized with 1 (i=0). To keep the invariance property when i is increased, x_1 has to be increased by $(3(i+1)^2 + 3(i+1) + 1) - (3i^2 + 3i + 1) = 6i + 6$. This also cannot be done directly; thus, again a new variable is introduced with invariance property $x_2 = 6i + 6$; it has to be initialized with 6. To maintain the invariance property, x_2 has to be increased by $(6(i+1) + 6) - (6i + 6) = 6$, which can be done without a new variable. Thus, the developed program is:

$i := 0, x_0 := 0; x_1 := 1; x_2 := 6;$
while $i \neq n$ **do**
$x_0 := x_0 + x_1;$
$x_1 := x_1 + x_2;$
$x_2 := x_2 + 6;$
$i := i + 1$ **od** .

Abstracting from the example, when supplied with a specification of the shape $n \geq 0 \vdash \langle \$C \rangle x_0 = \Phi(n)$, we generate the following schema for $C:

i:= 0, x_0:= $\Phi(0)$; $\{0 \leq i \leq n \wedge x_0 = \Phi(i)\}$
while $i \neq n$ **do**
$x_0 := x_0 + \tau_1(i)$;
$body;
i:= i+1 **od** $\{0 \leq i \leq n \wedge x_0 = \Phi(i)\}$

where $\tau_1(i) := \Phi(i+1) - \Phi(i)$. If $\tau_1(i)$ does not contain nonelemenatry operations, the strategy terminates. Otherwise a new variable x_1 is introduced with invariance property $x_1 = \tau_1(i)$, and the schema is augmented by the initialization $x_1 := \tau_1(0)$ and the assignment $x_1 := x_1 + \tau_2(i)$ in the loop body, where $\tau_2(i) := \tau_1(i+1) - \tau_1(i)$. The invariant is augmented by the conjunction $x_1 = \tau_1(i)$. This process is repeated until a τ_i is found that contains no nonelementary operations.

Implementation in the KIV System

1. Generate a proof tree with premises
   (1) $\Gamma \vdash \langle \$C1 \rangle (\$rest \wedge \psi)$
   (2) $\$rest, \psi, \$b \vdash \langle \$C \rangle (\$rest \wedge \psi)$
   (3) $\$rest, \psi, \neg \$b \vdash x_0 = \Phi(n)$
   (4) $\psi, \$b, \$rest \vdash bf > 0$ (bf is the bound function)
   (5) $\psi, \$b, \$rest, bf = t \vdash \langle \$C \rangle bf < t$

   and conclusion $\Gamma \vdash \langle \$C1; \textbf{while } \$b \textbf{ do } \$C \textbf{ od} \rangle x_0 = \Phi(n)$.
2. Instantiate $C1 with i:=0; $init , $b with $i \neq n$, and $C with $body; i:= i+1 .
3. Split the second premise into
   $\$rest, \psi, i \neq n \vdash \langle \$body; i:= i+1 \rangle \$rest$ (new second premise) and
   $\$rest, \psi, i \neq n \vdash \langle \$body; i:= i+1 \rangle \psi$ (new third premise),

where ψ has the shape $x_j = \tau_j(i)$. x_j is the j-th new variable to be introduced. We have $\tau_0(i) = \Phi(i)$, and $\tau_{j+1}(i) = \tau_j(i+1) - \tau_j(i)$. The next steps only deal with the third premise. Thus, it has to be shown

$$\Delta \vdash \langle \$body; i:= i+1 \rangle x_j = \tau_j(i) ,$$

where $\Delta \equiv \$rest, x_j = \tau_j(i), i \neq n$.

4. Generate the specification for the assignment in the loop body, i.e. eliminate the assignment i:= i+1; the new goal is

$$\Delta \vdash \langle \$body \rangle wp(i:= i+1, x_j = \tau_j(i)) .$$

5. compute initialization for variable x_j; $x_j := \tau_j(0)$.

6. compute $\tau_{j+1}(i) := \tau_j(i+1) - \tau_j(i)$. If $\tau_{j+1}(i)$ contains no nonelementary operations, instantiate $body with $x_j := x_j + \tau_{j+1}(i)$, instantiate $rest with true and terminate the strategy. Otherwise instantiate $body with $x_j := x_j + x_{j+1}$; $body , where x_{j+1} is a new variable; instantiate $rest with

$rest $\wedge x_{j+1}=\tau_{j+1}(i)$; and repeat steps 3-6, after suitable preparation of the goals.

As already mentioned, the advantage of this method is that in those cases where it is applicable, it runs completely automatically.

References

[BK 84] Bergstra, J. A., Klop, J. W. Proving Program Inclusion Using HOARE's Logic, Theoretical Computer Science 30 (1984), pp. 1 - 48

[BM 79] Boyer, R.S./ Moore, J.S. A Computational Logic. Academic Press, New York 1979

[Bur 74] Burstall, R.M. Program Proving as Hand Simulation with a little Induction. Information Processing 74, North-Holland Publishing Company (1974)

[Bun 89] Bundy, A. Automatic Guidance of Program Synthesis Proofs. Proc. Workshop on Automating Software Design, IJCAI 89. Kestrel Institute, Palo Alto (1989), pp. 57-59

[Co 86] Constable, R. et al. Implementing Mathematics with the Nuprl Proof Development System. Prentice Hall, Englewood Cliffs (1986)

[Di 88] E.W. Dijkstra. Lecture held on 8/4/1988, International Summer School "Constructive Methods in Computing Science", Marktoberdorf 1988.

[FM 88] Felty,A./Miller, D. Specifying Theorem Provers in a Higher-Oder Logic Programming Language. Proc. 9-th International Conference on Automated Deduction, E. Lusk, R. Overbeek (eds), Springer LNCS 310 (1988), pp. 61-80

[GMW 79] Gordon,M/Milner,R./Wadsworth,C. Edinburgh LCF. Springer LNCS 78 (1979)

[Go 82] Goldblatt, R. Axiomatising the Logic of Computer Programming. Springer LNCS 130 (1982)

[Gr 81] Gries, D. The Science of Programming, Springer-Verlag (1981)

[Ha 79] Harel, D. First Order Dynamic Logic. Springer LNCS 68 (1979)

[Hä 87] Hähnle, R. Programmverifikation durch symbolische Ausführung und Induktion. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (1987)

[He 89] Heisel, M. A Formalization and Implementation of Gries's Program Development Method within the KIV Environment. Int. Bericht 3/89, Fak. für Informatik, Univ. Karlsruhe

[HRS 87] Heisel,M./Reif, W./Stephan, W. Program Verification by Symbolic Execution and Induction. Proc. 11-th German Workshop on Artificial Intelligence, K. Morik (ed.), Informatik Fachberichte 152, Springer-Verlag (1987)

[HRS 89] Heisel,M./Reif, W./Stephan, W. A Dynamic Logic for Program Verification. Proc. Logic at Botik 89, A. Meyer, M. Taitslin (eds), Springer LNCS 363 (1989), pp. 134-145

[Mu 84] Mulmuly, K. Mechanization of existence proofs of recursive functions. Proc. 7-th International Conference on Automated Deduction, Springer LNCS 170 (1984), pp. 460-475

[OS 88] Oppacher, F./Suen, E. HARP: A Tableau-Based Theorem Prover. Journal of Automated Reasoning, Vol. 4 (1988), pp. 69-100.

[Pa 87] Paulson, L. Logic and computation. Cambridge University Press, Cambridge (1987)

[Re 84] Reif, W. Vollständigkeit einer modifizierten Goldblatt-Logik und Approximation der Omegaregel durch Induktion. Diplomarbeit, Fak. für Informatik, Univ. Karlsruhe (1984)

[St 85] Stephan, W. A Logic for Recursive Programs. Interner Bericht 5/85, Fak. für Informatik, Universität Karlsruhe (1985)