Sādhanā, Vol. 21, Part 2, April 1996, pp. 229–244. © Printed in India.

Reuse of proofs in software verification

WOLFGANG REIF and KURT STENZEL

Abt. Programmiermethodik, Fakultät für Informatik, Universität Ulm, D-89069 Ulm, Germany
E-mail: {reif, stenzel}@informatik.uni-ulm.de

Abstract. This paper presents a method for automated reuse of proofs in software verification. Proofs about programs as well as proof attempts are used to guide the verification of modified programs, particularly of program corrections. We illustrate the phenomenon of reusability, present an evolutionary verification process model and discuss theoretical and technical aspects. Finally, we report on case studies with an implementation of this method in the Karlsruhe Interactive Verifier (KIV).

Keywords. Automated reuse of proofs; software verification; Karlsruhe interactive verifier.

1. Introduction

Currently, the technological frontier for developing correct software is somewhere between 1000 and 2000 lines of verified code per year (Moore 1988,Re92b). This productivity can be achieved with advanced verification systems, such as the Boyer & Moore Prover (Boyer & Moore 1979), or the KIV system (Reif 1992). Combined with a hierarchical and strictly decompositional software design discipline, these systems can be used to verify fairly large systems. Nevertheless, the development of verified software is still a time and money consuming activity.

Most verification systems make the tacit assumption that the major problem to be solved is to verify (affirmatively) a software system. Experience shows, however, that in practical applications this assumption is not realistic and must be relaxed: a proof attempt is more likely to reveal errors than to prove their absence, and the programs under consideration might undergo several modifications until correct versions are obtained. Most verification systems ignore this evolutionary aspect of programming. In KIV this defect is overcome. KIV pursues an evolutionary verification model and offers tool support for a tight integration of error correction and verification.

The main problem to be solved is to preserve or to re-establish proofs that have become invalid by modifications. This is illustrated by the following scenario. Let $\varphi_1(\alpha)$ and $\varphi_2(\alpha)$ be two proof obligations with a program α occurring in both formulas. Suppose, $\varphi_1(\alpha)$ has been successfully proved, although the program α is erroneous. This means that the

errors in α do not affect the truth of $\varphi_1(\alpha)$. Assume further, that a first proof attempt for $\varphi_2(\alpha)$ failed, because of the errors in α. After correction of α this proof has to be repeated. Moreover, since the program α has changed to β, $\varphi_1(\alpha)$ has changed to $\varphi_1(\beta)$. Therefore the original proof for $\varphi_1(\alpha)$ (although successful) becomes obsolete. This means that for every program correction all proof obligations involving the corrected program have to be proved again. Conventional verification systems repeat these proofs over and over again without exploiting the experience accumulated during earlier proof attempts.

This paper presents a method which extracts this experience from previous proofs in order to guide the proofs for the corrected programs. Case studies with the KIV system (Heisel *et al* 1990; Reif 1992) have shown that large parts of earlier proofs can actually be reused, saving a lot of proof search and user assistance.

The phenomenon of reusability is illustrated by an example in § 2. Section 3 investigates the basic assumptions of the approach, and sketches an evolutionary verification methodology based on reuse of proofs. In § 4 we present the theoretical and technical aspects. In § 5 we report on our experiences with an implementation in the KIV system, and give an evaluation of the experimental results. Section 6 comments on related work, and in § 7 we draw some conclusions.

2. An example

2.1 *Binary arithmetic and dynamic logic*

Consider binary words with two constants *zero* and *one*, as well as two unary constructors s_0 and s_1. The constructors s_0 and s_1 add *zero* or *one*, respectively, at the end of a given binary word ($s_1(zero)$ stands for the word '01'). The selector *top* selects the last bit of a word, and *pop* cuts off the last bit. The unary predicate $nlz(a)$ is true if a has no leading zeros. Binary words without leading zeros are used as representations of natural numbers. The two procedures $succ(a : b)$ and $pred(a : b)$ with input parameter a and output parameter b are intended to implement the arithmetical functions successor and predecessor on binary words, respectively. Finally, the statement to be proved, asserts that *pred* is the inverse operation of *succ* at least for inputs satisfying *nlz*. In the KIV system this proof obligation is expressed as a sequent of Dynamic Logic (DL, see Harel 1979; Heisel *et al* 1989):

$$nlz(a) \vdash \langle succ(a : b); pred(b : c)\rangle c = a \tag{1}$$

In a sequent $\Gamma \vdash \Delta$, Γ and Δ are lists of formulas. A sequent holds if the conjunction of the formulas of Γ implies the disjunction of the formulas of Δ. A formula $\langle\alpha\rangle\varphi$ is true if the program α terminates and φ holds afterwards. Hence the above sequent can be read as: if the input a has no leading zeros, *succ* terminates for a and yields a result stored in b. Then *pred* terminates with input b and yields a result stored in c equal to a.

The truth of (1) depends, of course, on the implementations of *succ* and *pred* (figure 1). The program *succ* is given in two versions: a faulty one on the left and a correct one on the right. In the erroneous version the case for $top(x) = zero$ is missing. If we adopt the implementation of *pred* and the second version of *succ*, the proof obligation (1) can be

succ $(x : y)$
if $x = zero$ **then** $y := one$ **else**
 if $x = one$ **then** $y := s_0(one)$ **else**
 $succ(pop(x):y)$; $y := s_0(y)$ **fi fi**

succ $(x : y)$
if $x = zero$ **then** $y := one$ **else**
 if $x = one$ **then** $y := s_0(one)$ **else**
 if $top(x) = zero$ **then** $y := s_1(pop(x))$ **else**
 $succ(pop(x):y)$; $y := s_0(y)$ **fi fi fi**

pred $(x : y)$
if $x = zero \vee x = one$ **then** $y := zero$ **else**
 if $x = s_0(one)$ **then** $y := one$ **else**
 if $top(x) = one$ **then** $y := s_0(pop(x))$ **else**
 $pred(pop(x):y)$; $y := s_1(y)$ **fi fi fi**

Figure 1. the programs: two versions of *succ*, and *pred*.

proved. With the first version of *succ*, (1) is not provable. Let us compare the attempts to prove both versions of (1).

2.2 *The proofs for the two versions of* succ

Starting out from the original proof obligation, a proof tree (like those in figure 2) is constructed in a goal-directed manner. Proof rules are applied, reducing a goal to sufficient subgoals which themselves are reduced and so forth. The original proof obligation is the root of the tree (conclusion), the yet unproved subgoals are its premises (light circles). A proof tree stands for the assertion that the conclusion holds if the premises do. A proof is completed if no premises remain.

KIV provides a proof strategy for DL sequents. Applying it to (1) with the erroneous version of *succ*, we obtain the proof tree on the left side in figure 2. With regard to the intended reuse of this proof, we will call it the "old" proof and the other one the "new" proof. One open premise (59) remains: $a \neq zero, a \neq one \vdash top(a) = one$ states that the last bit of every binary word, different from *zero* and *one*, is *one*. Since this is wrong, the proof attempt fails.

Applying the proof strategy to (1) with the correct version of *succ* (the second one in figure 1) it yields the proof tree on the right side of figure 2. This proof is successful.

The fragments A, B, C, D of the old proof correspond directly to the fragments A, B, C, D of the new proof. Although the corresponding sequents in the two proofs are different, the same rules can be applied in the same order within the fragments. However, there are intermediate proof fragments F and G in the new proof without counterpart in the old proof. They deal with the correction of *succ*, an additional conditional with a new **then**-branch. Although in the example the relative order of A, B, C, D is preserved, in general this is not the case.

Furthermore, the fragment E of the old proof corresponds to the fragment E' in the new proof. However, the rule applied in the new proof differs from the one applied in the old, because the fragment E' is carried out in a different proof context. This change of the proof context is due to the intermediate fragment F in the new proof. The fragment H has no counterpart in the new proof.

In total 95% of the old proof can be reused, which amounts to 85% of the new proof. Only 15% of the proof steps of the new proof are actually new. The example illustrates the

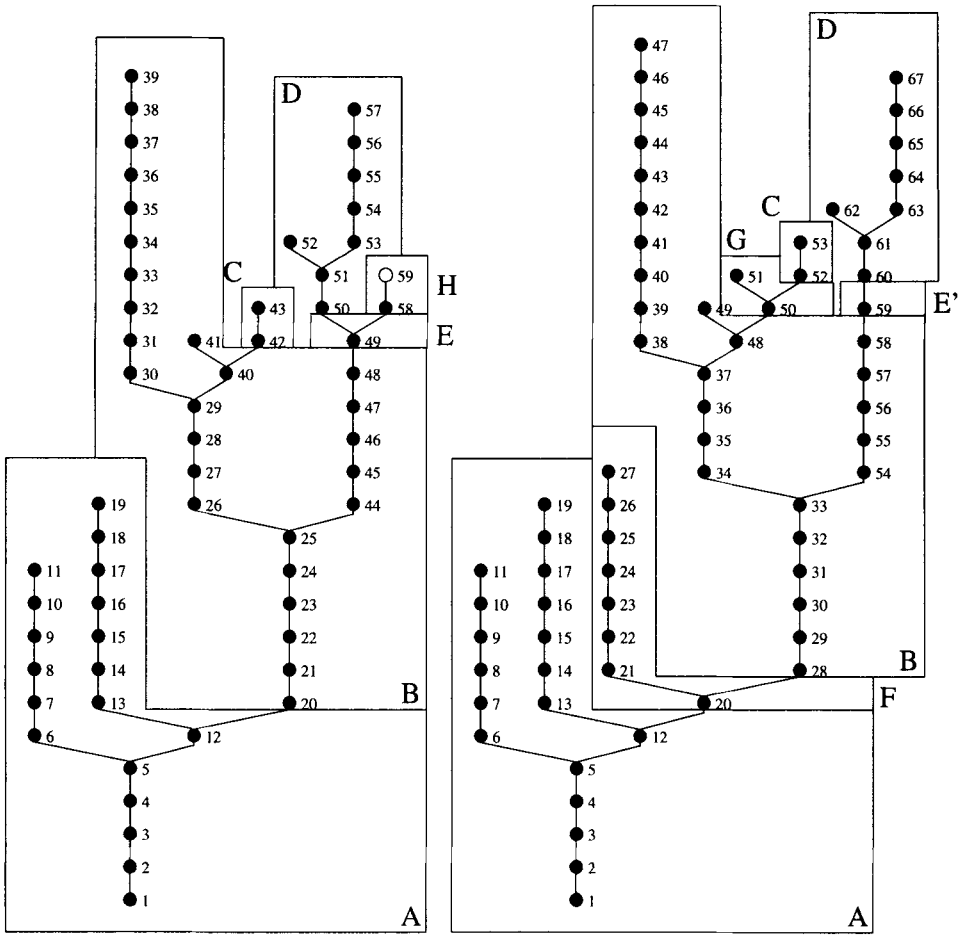


Figure 2. Comparison of the two proofs: old vs. new proof.

potential that can be exploited with an automated technique for reuse of proofs. However, a careful analysis of the corrected program and the old proof is required to detect the reusable fragments, to determine the order in which to apply them, and to cope with the influence of modified proof contexts. Experiences confirm that the above example is not an exception but reflects a general phenomenon.

3. An evolutionary approach to verification

Before presenting the technique for reuse in § 4, we describe how it can be incorporated into a conventional verification process model. The result is an *evolutionary approach* to verification. It is called *evolutionary* because proof attempts often reveal errors instead of showing their absence, and the programs are subject to several modifications until correct versions are obtained.

The evolutionary approach to verification is built on top of the conventional proof strategy used in § 2 to generate the proof on the left side in figure 2. It is called the *basic*

strategy and is based on symbolic execution of programs and induction. To some extent the approach is generic, and applies to other strategies as well, provided they meet two basic requirements. First, verification with a particular strategy must be "continuous" in the informal sense that small changes in the program text usually lead to small changes in the proof. Second, there must be a correspondence between positions in programs and positions in proofs generated by the strategy. Whether the strategy fulfills these requirements, depends on the underlying proof rules. For the one adopted in this paper they are satisfied. Other examples are the methods due to Boyer & Moore (1979) and Burstall (1974).

Verification with reuse of proofs proceeds as follows: The first proof attempt for a statement $\varphi(prog)$ about a procedure $prog$ with a body α is carried out with the basic strategy. If this attempt leads to a subgoal g which seems to be unprovable, the proof is interrupted. Let t be the proof constructed so far. If the user provides a ground substitution s for the free variables of g as a candidate for a counterexample for g, the system tries to verify that s is indeed a counterexample, hence that g is not provable.

Example 1. In our example from § 2 $prog$ is the procedure *succ* with the faulty implementation (left hand side of figure 1), and $\varphi(prog)$ is

$$nlz(a) \vdash \langle succ(a : b); pred(b : c)\rangle c = a \tag{2}$$

The proof attempt leads to a subgoal (No. 59 on the left side in Fig. 2)

$$a \neq zero, a \neq one \vdash top(a) = one \tag{3}$$

The user guesses $a = s_0(one)$ and KIV proves that $s_0(one)$ is indeed a counter example for the goal, i.e. that the negation of the goal is true for $a = s_0(one)$:

$$a = s_0(one) \vdash \neg(a \neq zero \wedge a \neq one \rightarrow top(a) = one) \tag{4}$$

Now the goal g is known to be unprovable. What does this mean? In general, there are two possible reasons why a proof attempt may lead to an unprovable goal. Either a wrong decision was made during the proof (in this case the faulty proof decision must be found and withdrawn), or the original goal $\varphi(prog)$ is not provable. To find out the accurate reason the system tries to construct a counterexample s' for $\varphi(prog)$ from s by inspecting t. Inspecting the proof tree basically means to collect and simplify all first order formulas on the relevant branch of the proof tree from g backwards to the conclusion $\varphi(prog)$.

Example 2. In our example, KIV computes the candidate counter example $a = s_0(one)$. KIV proves automatically that $a = s_0(one)$ is indeed a counter example for (2). Inspection of the counter example proof reveals that $succ(s_0(one) : b)$ produces the output $b = s_0(s_0(one))$, i.e. the successor of two is four. Consequently, an error is located in the implementation of the procedure *succ*.

Now the original goal $\varphi(prog)$ is known to be unprovable because of an error in the procedure body α of $prog$. Therefore, the user must provide a corrected version of $prog$ with body β instead of α, and $\varphi(prog)$ has to be proved again.

In order to enable the reuse of t in the new proof, the system computes a presentation of β as a combination of fragments of α and new fragments. Then t is analysed and a

correspondence is set up between the fragments of α and proof fragments of t. Finally, the new proof attempt is guided by these proof fragments. The reuse of proofs is explained more precisely in the next section.

The quality of the technique depends on the quality of the first proof attempt. If the proof idea was correct, and the proof failed due to errors in the implementation, then reuse usually yields good results. If the first attempt followed a wrong idea, reuse is unlikely to succeed in a second attempt.

4. The technique for reuse

4.1 *Presentation of program corrections*

We represent program corrections by presenting the corrected program as a combination of fragments of the old program and new fragments. Therefore, we need the formal notions (program) *skeletons* and (program) *fragments*. Roughly skeletons are programs with "holes", in which other skeletons may be inserted. A fragment of a program is a skeleton together with the position (*occurrence*), where in the program it occurs. If a fragment is a common part of several programs, the skeleton is associated with several occurrences. In the following definitions we view programs as terms and use some notation from Huet & Oppen (1980).

DEFINITION 1
Skeletons, occurrences.

- *Skeletons* are the smallest set containing $\Box$ (the "hole"), the statements **skip**, **abort**, assignment $x := \tau$, procedure call $f(\underline{\sigma} : \underline{y})$, and which is closed under conditional (**if** ε **then** α **else** β), definition of local variables (**var** x = τ **in** α) (where α is the scope of the definition of x with initialization τ) and compound (α; β).
- A *program* is a skeleton without $\Box$. We call a skeleton α *connected* iff for each compound γ; δ occurring in α, γ is a program.
- Let $O(\alpha)$ be the *set of occurrences* of α, $p \in O(\alpha)$ a *position* in α. Here, positions are sequences over $\{1, 2\}$.
- α/p denotes the subskeleton at position p in α, $\alpha[p \leftarrow \gamma]$ the *replacement in* α *with* γ *at* p (for $p \in O(\alpha)$).
- $\prec_1$ is a partial order on skeletons: $\alpha \prec_1 \beta$ iff there exist a skeleton $\gamma \neq \Box$ and a position $p \in O(\beta)$, such that $\beta/p = \Box$ and $\alpha = \beta[p \leftarrow \gamma]$. $\preceq$ is the reflexive, transitive closure of $\prec_1$.

  Intuitively, $\alpha \prec_1 \beta$ means that α is more concrete than β: β can be derived from α by replacing a subskeleton by $\Box$, or, conversely, α can be derived from β by replacing $\Box$ by a skeleton (this replacement can be viewed as an instantiation). If $\alpha \preceq \beta$ holds we call β a *pattern* for α.
- For two sequences p and q, pq is the concatenation of p and q. For a sequence p and a set S of sequences, $pS = \{pq | q \in S\}$. () denotes the empty sequence.

Example 3. An example for a skeleton α is

if $x = zero$ **then** $y := one$ **else**
if $x = one$ **then** $y := s_0(one)$ **else** $\square$

This skeleton is derived from the procedure *succ* (figure 1) by replacing the **else**-part of the second conditional by $\square$. The set of occurrences $O(\alpha)$ contains

- $(), \alpha/() = \alpha$ itself,
- $(1), \alpha/(1)$ is the **then**-branch of the first conditional $y := one$,
- $(2), \alpha/(2)$ is the **else**-branch of the first conditional, i.e. the second conditional,
- $(2, 1), \alpha/(2, 1)$ is the **then**-branch of the second conditional $y := s_0(one)$,
- $(2, 2), \alpha/(2, 2)$ is the **else**-branch of the second conditional $\square$.

The replacement in α with $\square$ at position (2), $\alpha[(2) \leftarrow \square]$, yields

if $x = zero$ **then** $y := one$ **else** $\square$

α is smaller than $\alpha[(2) \leftarrow \square]$ with respect to $\prec_1$, since α can be constructed from $\alpha[(2) \leftarrow \square]$ by replacing $\square$ with the second conditional of α.

DEFINITION 2
Program fragments. Let $\gamma, \beta_1, \ldots, \beta_n$ be skeletons, $p_1, \ldots, p_n$ positions. $(\gamma, p_1, \ldots, p_n)$ is a *program fragment* of $(\beta_1, \ldots, \beta_n)$ iff $p_i \in O(\beta_i)$ and γ is a pattern for β_i/p_i for $i = 1 \ldots n$. The fragment is called *connected* iff its skeleton γ is connected.

In a program fragment $(\gamma, p_1, \ldots, p_n)$, γ is a pattern for a subskeleton of each β_i at the position p_i, i.e. it is possible to instantiate γ (with different skeletons for each i) to γ_i such that $\gamma_i = \beta_i/p_i$. Our aim is to compare an incorrect "old" program α with a correct "new" program β. Therefore we will consider only fragments (γ, p_1, p_2) of (β, α) and (γ, p_1) of (β), and call them *old* and *new* fragments, respectively. Basically, an old fragment describes a part of a program that can be found in both the old and the new program. The demand for connected skeletons is important in the analysis of the old proof (§ 4.2). Now we are prepared to define the representation of program corrections by presenting the corrected program as a sequence of old and new fragments.

DEFINITION 3
Presentation of β using α. Let β and α be skeletons, $P = (f_1, \ldots, f_n)$ a sequence with f_i a connected fragment of (β, α) or a fragment of (β). P is a *presentation of β using α* iff $\square \bullet P$ is defined and yields β:

$$\gamma \bullet P = \begin{cases} \gamma & \text{if } P = () \\ \gamma[p_1 \leftarrow \delta] \bullet P' & \text{if } P = (\delta, p_1, \ldots, p_n)P', \gamma/p_1 = \square, n \in \{1, 2\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The function $\bullet$ tells us how to construct the new program β from the sequence of proof fragments: Begin with $\square$ and replace $\square$ by the skeleton γ_1 of the first fragment, then replace

one occurrence of $\square$ in γ_1 by the skeleton of the second fragment, and so forth. If $\bullet$ is defined, then each position in a fragment corresponds to a $\square$ during the construction, and, if β is a program, eventually all occurrences of $\square$ are eliminated. Only the first position of each fragment (denoting a position in the new program) is used in this process, the second position (denoting a position in the old program if it exists) is only used in the analysis of the old proof (§ 4.2).

Example 4. The following is a presentation P of the body of the correct procedure *succ* (from Fig. 1) using the erroneous version.

$$P = \left(\begin{array}{l} \left(\begin{array}{l} \textbf{if } x = zero \textbf{ then } y := one \textbf{ else} \\ \quad \textbf{if } x = one \textbf{ then } y := s_0(one) \textbf{ else } \square \end{array}, (), () \right) \\ (\textbf{if } top(x) = zero \textbf{ then } y := s_1(pop(x)) \textbf{ else } \square, (2, 2)) \\ (succ(pop(x)\text{:}y);\ y := s_0(y), (2, 2, 2), (2, 2)) \end{array} \right)$$

The first and third fragments are fragments of both versions of the body, hence are *old* fragments, the second one is a fragment of the corrected body only, hence a *new* fragment. The new program can be constructed by replacing $\square$ in the first skeleton by the second skeleton, and then by replacing $\square$ from the second skeleton by the third skeleton.

Every program β can be trivially presented using α by $P = ((\beta, ()))$. However, our intention is to use as many fragments of the old program α as possible. Therefore we need the notion of optimal presentations.

DEFINITION 4
Optimal presentation. A presentation P of β using α is called *optimal* iff P fulfills the following three conditions:

1. It is not possible to extract from a new fragment an old fragment, i.e. to find a pattern for a subskeleton of the new fragment that is also a pattern for a subskeleton of the old program. Formally:

   There are no new fragment $(\gamma, p) \in P$, positions $q \in O(\gamma)$ and $q' \in O(\alpha)$ and connected skeleton $\delta \neq \square$ such that δ is a pattern for γ/q and α/q'.

   Since δ is a pattern for a part of the old program, δ should belong to an *old* fragment, even if the number of fragments is increased.

2. It is not possible to merge two new fragments into one. Formally:

   There are no fragments $(\gamma_1, p_1), (\gamma_2, p_2) \in P$ and position $p_3 \in O(\gamma_1)$ such that $p_1 p_3 = p_2$.

   If P is a presentation and $p_1 p_3 = p_2$, then $\gamma_1/p_3 = \square$ holds, and it is possible to instantiate γ_1 at p_3 with γ_2 and to discard the second fragment (γ_2, p_2), thereby reducing the number of fragments.

3. It is not possible to merge two old fragments into one. Formally:

   There are no fragments $(\gamma_1, p_1, q_1), (\gamma_2, p_2, q_2) \in P$, positions $p_3 \in O(\gamma_1)$ and $q_3 \in O(\alpha)$ such that $p_1 p_3 = p_2$ and $\gamma_1[p_3 \leftarrow \gamma_2]$ is a pattern for α/q_3.

The definition is slightly more complex than for new fragments, since it is possible that one old fragment is instantiated with another old fragment when constructing the new program, but the second fragment stems from another position in the old program. However, if the skeleton of the combined fragment is also a pattern for part of the old program, both old fragments can be replaced by the combined fragment.

Condition 1 is the most important one. It guarantees that the old program is reused as much as possible. The two other conditions just reduce the number of fragments. Example 4 shows an optimal presentation that is also unique. However, this is not always the case, since a program may contain one skeleton at different positions, which leads to different optimal presentations.

The following theorem states that *any* program correction can be expressed as an optimal presentation.

Theorem 1. *For two programs α and β there exists an optimal presentation of β using α.*

Proof sketch. For two programs α and β there always exists a presentation of β using α — the trivial presentation $((\beta, ()))$. Starting from this presentation it is possible to construct an optimal presentation. Assume that one fragment violates condition 1. Then this fragment can be replaced by one or more other fragments that do not violate 1., such that the new sequence of fragments still is a presentation of β using α. If condition 2. or 3. is violated by two fragments, they can be replaced by one fragment (see Def. 4.2 and 4.3). By iteration we get an optimal presentation.

4.2 *Analysis of the old proof*

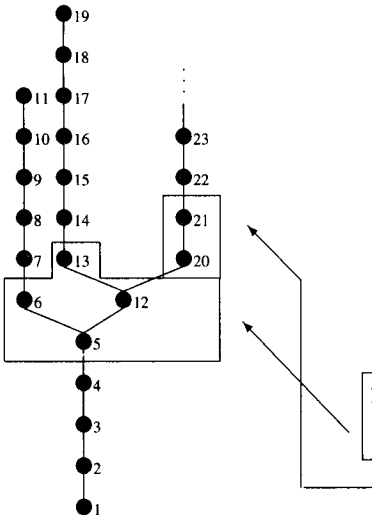
The next step is the analysis of the old proof with a given presentation P of the new program β using α. Our aim is to identify, for each program fragment f_i of β and α, corresponding fragments of the old proof.

In the first phase of the analysis of the old proof we assign to a goal g a position q, if the goal contains the subskeleton α/q of the old program α, no position otherwise. This assignment can be computed since each rule of the calculus dealing with programs is extended by a description how programs and occurrences are modified.

Here are some example rules: the symbolic execution rules *conditional* and *assign*. The rules are reduction rules and have to be read bottom up:

$$\frac{\Gamma, \varepsilon \vdash \langle \alpha_1 \rangle \varphi, \Delta \qquad \Gamma, \neg\varepsilon \vdash \langle \alpha_2 \rangle \varphi, \Delta}{\Gamma \vdash \langle \textbf{if } \varepsilon \textbf{ then } \alpha_1 \textbf{ else } \alpha_2 \rangle \varphi, \Delta} \qquad \frac{\Gamma \vdash \varphi_x^\tau, \Delta}{\Gamma \vdash \langle x := \tau \rangle \varphi, \Delta}\, x' \text{ new}$$

The first rule (*conditional*) has two premises, one for a positive test ε and one for a negative test $\neg\varepsilon$. This rule is the proof theoretical counterpart to a symbolic execution of a conditional. If q is the assigned position of the conclusion, then α/q = **if** ε **then** α_1 **else** α_2. Since α_1 is the next statement in the first premise, we assign to this goal the new position $q(1)$, and to the second premise $q(2)$. The second rule corresponds to the symbolic execution of the assignment $x := \tau$. Since the application of this rule discards the statement, no position is assigned to the new goal. In both cases, the conclusion of the rule corresponds to exactly one statement of a program.



1-4. $*$
5. **if** $x = zero$ **then** $y := one$ **else**
if $x = one$ **then** $y := s_0(one)$ **else**
$succ(pop(x):y);\ y := s_0(y)$
6. $y := one$
7-11. $*$
12. **if** $x = one$ **then** $y := s_0(one)$ **else**
$succ(pop(x):y);\ y := s_0(y)$
13. $y := s_0(one)$
14-19. $*$
20. $succ(pop(x):y);\ y := s_0(y)$
21. $y := s_0(y)$
22-... $*$
etc.

if $x = zero$ **then** $y := one$ **else**
if $x = one$ **then** $y := s_0(one)$ **else**

$succ(pop(x):y);\ y := s_0(y)$

Figure 3. The analysed proof.

In the next phase we identify the set of proof fragments T corresponding to program fragments. For each fragment $f = (\gamma, p, q)$ in P and each goal g with an assigned position q, the corresponding proof fragment is computed recursively over the subtree beginning with g. A proof step with conclusion g' belongs to the proof fragment for f if the position q' assigned to g' belongs to $qO(\gamma)$. The process ends if $q' \notin qO(\gamma)$ or no position is assigned to g'. This approach guarantees that the resulting proof fragment forms a proof tree in itself. The demand for connected fragments (in Def. 3) assures that every goal g' with an assigned position $q' \in qO(\gamma)$ is member of one $t_i \in T$. Result of the analysis are T and the set T' of proof fragments where α does not appear in any goal.

Example 5. We demonstrate the analysis for the example presented in § 2 (figure 3). On the left side the lower half of the proof is shown. On the upper right side the corresponding subskeletons α/q for each goal are shown (following the numbering of goals in the proof). $*$ marks goals with no assigned position. The lower right side shows the correspondence between proof fragments and program fragments.

4.3 *The new proof*

The last step in the reuse of proofs is the proof of the original goal with the corrected program β instead of α. Since this is the only difference between the old and the new goal the proof is likely to differ only in those parts where the corrected program is involved somehow. In general, however, β also introduces new fragments without any counterpart in the old proof. Then, the basic proof strategy with all its heuristics is invoked. This means that the new proof is done partly by the basic proof strategy and partly by using fragments

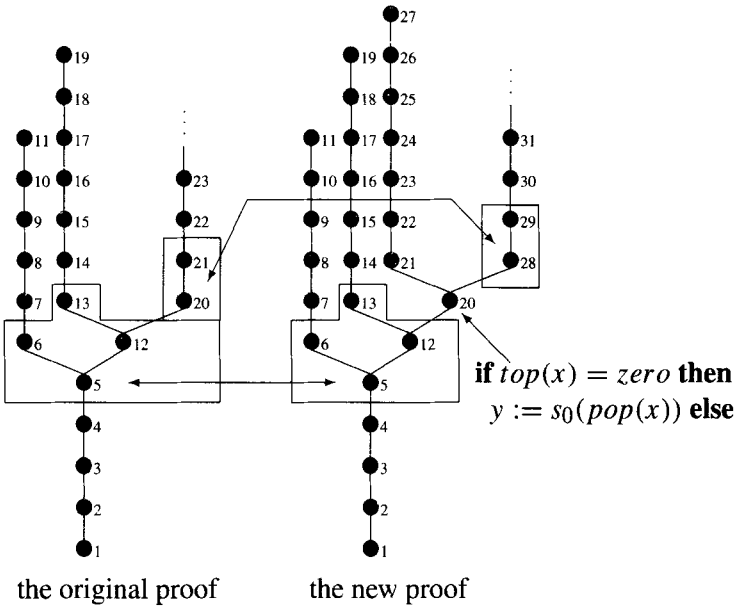


Figure 4. The new proof.

of the old proof. The procedure starts out with the initial goal as the *current goal* and proceeds recursively according to the following decision table:

- If the corrected program does not occur in the current goal of the new proof, a proof fragment from T' is selected and reused.
- If the current goal refers to an old program fragment (γ, p, q) of β and α, a corresponding proof fragment from T for (γ, p, q) is selected and reused.
- If the goal refers to a new program fragment (γ, p), the basic proof strategy is invoked.

Reusing a proof fragment from T or T' basically means to copy each rule application of this fragment and to check whether it is actually applicable. Otherwise it is checked whether this proof step can be replaced by a different one without giving up the rest of the fragment. This flexibility is achieved by a number of elaborate heuristics. Heuristics are also involved in the selection of suitable proof fragments from T or T'. It may happen that the program fragment referred to in the current goal is associated with more than one proof fragment in T, or there may be several possible selections for proof fragments in T'. If no further reuse is possible for a goal the basic strategy is invoked.

Example 6. The procedure is explained by the example. Goals are numbered in both proofs. To distinguish between old goals and new goals, numbers for old goals are indexed with "o", numbers for new goals with "n".

The reuse strategy starts with goal 1_n of the new proof and selects the old proof fragment 1_o–5_o, yielding 1_n–5_n. Goal 5_n refers to an old program fragment $f = (\gamma, p, q)$ in the presentation of β. The corresponding proof fragment for f begins at 5_o. Application of the rules of this fragment leads to a new proof with premises 7_n, 14_n, 20_n. Goal 20_n refers to a new program fragment (γ', p') in the presentation of β. This means that no reuse is

possible. Therefore the reuse strategy calls the basic proof strategy, which performs the proofs steps 20_n–27_n. The steps 7_n–11_n, 14_n–19_n result from reusing fragments of T'.

At 28_n the old program fragment 20_o–21_o is reused, even though the proof context has changed: $top(a) \neq zero$ is an additional precondition. This modification avoids the dead end of the old proof, but has no other influence on the proof. The remaining details are omitted here. The resulting tree is identical to the new proof in figure 2.

5. Results

Now we report on our experiences with the reuse strategy implemented in the KIV system. First of all, the algorithms for proof analysis and the strategy for reuse are efficient enough to be integrated into KIV's tactic collection for interactive theorem proving. The analysis algorithm typically takes only one to ten percent of the time needed to carry out the new proof. The reuse strategy is faster than the basic proof strategy (even if there are no interactions) since there is less proof search in the reuse strategy. To illustrate its applicability we present a real example in the sense that the errors have not been introduced a posteriori, but have been discovered during the verification of a module for binary arithmetic.

The procedure *divmod* (figure 5) computes the quotient and the remainder in binary arithmetic. It is recursive and uses two other procedures *le* (less or equal) and *sub* (subtraction). This procedure contains three errors, marked with ^{1} to ^{3}. The first (^{1}) can be considered as a typographical error and is corrected by replacing a variable, the second (^{2}) is corrected by replacing a part of the program and the third (^{3}) is corrected by deleting one statement and changing the structure of the program. Especially the last correction shows that a strategy for reuse of proofs must be able to cope with complex program corrections. We show how these errors have been found and corrected during the attempt to prove four properties of this procedure. They are named *divmod-r* (termination of *divmod*), *mod-lemma* (property of the remainder), *div-lemma* (property of the quotient), and *mod-ls* (remainder less than divisor).

The verification protocol is given below. The interesting statistical results are the number of proof steps, interactions with the user and the total time needed for the proof. The basic proof strategy is partly interactive; e.g. 7 interactions for a proof with 169 proof steps means that 7 proof steps were applied by the user, and that the remaining 162 steps were performed automatically. We measure the degree of reusability by two numbers that indicate *how many of the old proof steps are reused?* and *how many proof steps of the new proof stem from reuse?*. These numbers will be given in the form *Reuse: 97% : 95%*.

The verification starts by proving the first goal divmod-r using the basic strategy:

1. proof of divmod-r: successful with 169 proof steps, 7 interactions, 6 min. (even though the procedure contains three errors, it still terminates)
2. proof of mod-lemma: failure after 571 proof steps, 66 interactions, 3 h 45 min. This failure is a result of the first error (marked above by ^{1}).
3. correction of the error. New proof of mod-lemma: 566 proof steps, 0 interactions, 18 min. till the situation of 2. *Reuse: 95% : 99%* (without reuse, i.e. using the basic strategy to prove mod-lemma again till 2., 66 interactions and 3 h 45 min. are needed)

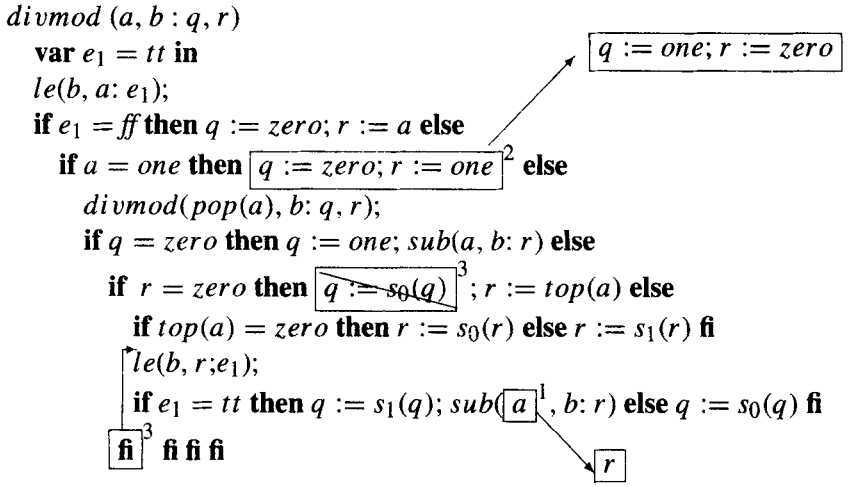


Figure 5. The procedure *divmod*.

4. continue the proof of mod-lemma. mod-lemma and div-lemma can be proved. The remaining two errors become apparent during the proof of mod-ls. After their correction mod-ls is proved.

Now the following situation is reached: All four proof obligations have been successfully proved. However, *divmod* has been modified three times and the proofs refer to different versions of *divmod*. Only the proof for mod-ls refers to the correct *divmod*. Therefore the first three goals divmod-r, mod-lemma and div-lemma have to be proved again. The reuse strategy is applied again:

5. new proof of divmod-r: 220 proof steps, 3 interactions, 5 min *Reuse: 116% : 89%*. 116% result from the reuse of several proof steps twice. (without reuse 220 proof steps, 9 interactions, 9 min)
6. new proof of mod-lemma: 1389 proof steps, 24 interactions, 58 min *Reuse: 96% : 76%* (without reuse 1389 proof steps, 147 interactions, 6 h 45 min)
7. new proof of div-lemma: 140 proof steps, 0 interactions, 4 min *Reuse: 98% : 96%* (without reuse 140 proof steps, 42 interactions, 1 h 45 min)

This concludes the proof of the four goals. Table 1 accumulates the results. If the procedure had been correct from the beginning, we would have obtained the figures for the *correct program*. In this case each goal would have been proved exactly once. Due to the errors it was necessary to repeat some proofs several times. This extra work is referred to as *additional with reuse* vs. *additional without reuse* in the table.

The example shows that reusability of proofs is a significant phenomenon: An average of 94% of the old proofs have been reused, and 92% of the new proofs are reused proof steps. With the strategy for reuse the verification effort (the overall time including specification, implementation, interactive and automatic proof) is only half the time needed without it. The additional verification effort due to error corrections is 11% compared to

Table 1. Comparison of the results with or without reuse.

	Time	Interactions	Steps
Correct program	9 h	227	1994
Additional with reuse	1 h	9	2192
Additional without reuse	12 h	255	2142
Total with reuse	10 h	236	4186
Total without reuse	21 h	482	4136

the verification effort for a correct version. Without a strategy for reuse the additional verification effort is approximately 130%. Furthermore, with the strategy for reuse the degree of automation improved significantly compared to a verification without it. The example demonstrates that the technique can handle complicated proofs and considerable program corrections. A number of case studies have confirmed these experiences. Reuse rates of more than 90% are typical, and the advantages of reuse grow with the size of the programs and proofs.

6. Related work

A certain amount of proof reuse capability is standard in most tactical theorem provers. Proof scripts and a replay mechanism can be found e.g. in NUPRL (Constable *et al* 1986), ISABELLE (Paulson 1994), HOL (Gordon 1988), PVS (Owre *et al* 1993) and others. The above reuse mechanism however, goes far beyond that. A machine learning approach to proof reuse is taken in Kolbe & Walther (1994). Example proofs are generalized and applied to similar goals by pattern matching. This approach aims at generalizing successful proofs but is not adequate to handle program corrections.

Proof reuse in the context of program corrections can be seen as a special case of analogical reasoning in the sense of Owen (1990). Let P_1 be a problem (goal) with the known solution S_1 (proof), and P_2 a new problem (corrected goal) with the yet unknown solution S_2 (proof). The problem to construct S_2 by analogy is divided into four subproblems:

- **base filtering.** For P_2 find a similar problem (here P_1) which is already solved.
- **analogy matching.** Find a mapping between P_1 and P_2.
- **plan construction.** Get one or more candidate solutions for S_2 by transforming the known solution S_1 using the analogy match.
- **plan validation.** Check whether the candidates for S_2 are indeed solutions for P_2. Otherwise modify the candidates appropriately.

In the context of program corrections the base filtering and the analogy matching problems are void, since P_2 is constructed from P_1. There is no search involved. The plan construction is realized by computing the presentation of the new program using the old one (§ 4.1) and analyzing the old proof (§ 4.2). Plan validation is realized by the construction of the new proof (§ 4.3).

7. Conclusion

We have presented an evolutionary approach to software verification based on reuse of proofs. It was shown how reuse of proofs can be integrated into a conventional verification process model. Proof attempts for erroneous programs are used to guide the verification of corrected versions.

Program corrections are formally described by presenting the corrected version of a program as a combination of fragments of the "old" program and new fragments. This presentation can be computed in such a way that the original program is reused optimally.

Based on the presentation of program corrections, the unsuccessful proof attempt is analysed and a correspondence is set up between old program fragments and corresponding fragments of the old proof.

This analysis is used to guide the verification of the corrected program. Since more than 90% of the old proofs can actually be reused, this approach saves a lot of proof search and user interaction. However, the quality of the technique depends on the quality of the proofs to be reused.

The technique is completely automated, and tested with complicated examples. We have presented one of them. The results show that reusing proofs improves current verification technology significantly.

This research was partly sponsored by the BMFT project KORSO.

References

Boyer R S, Moore J S 1979 *A computational logic* (New York: Academic Press)

Burstall R M 1974 Program proving as hand simulation with a little induction. *Information Processing* 74: 309–312

Constable R L, Allen S F, Bromley H M, Cleaveland W R, Cremer J F, Harper R W, Howe D J, Knoblock T B, Mendler N P, Panagaden P, Sasaki J T, Smith S F 1986 *Implementing mathematics with the Nuprl proof development system* (Englewood Cliffs, NJ: Prentice Hall)

Gordon M J 1988 HOL: A proof generating system for higher-order logic. In *VLSI specification and synthesis* (eds) G Birtwistle, P A Subrahmanyam (Amsterdam: Kluwer Academic Publishers)

Harel D 1979 *First order dynamic logic*, LNCS (Berlin: Springer)

Heisel M, Reif W, Stephan W 1989 A dynamic logic for program verification. In *Logic at Botik* LNCS (eds) A R Meyer, M A Taitslin, (Berlin: Springer) p. 89

Heisel M, Reif W, Stephan W 1990 Tactical theorem proving in program verification. *10th International Conference on Automated Deduction*, Kaiserslautern. LNCS (Berlin: Springer)

Huet G, Oppen D C 1980 Equations and rewrite rules: a survey. In *Formal languages: Perspectives and open problems* (ed.) R Book (New York: Academic Press)

Kolbe Th, Walther C 1994 Reuse of proofs. In *11th European Conference on Artificial Intelligence* (ed.) N L Cohn (Amsterdam: John Wiley & Sons)

Moore J S 1988 Piton, A verified assembly level language. Technical Report 22, Computational Logic, Inc., Austin, Texas

Owen S 1990 Analogy for automated reasoning. *Perspectives in artificial intelligence* (New York: Academic Press)

Owre S, Rushby J, Shankar N 1993 User guide for the PVS specification and verification system, language, and proof checker (Beta Release), Computer Science Laboratory, SRI International, Menlo Park, CA

Paulson L C 1994 *Isabelle: A generic theorem prover.* LNCS 828 (Berlin: Springer)

Reif W 1992 The KIV-system: Systematic construction of verified software. *11th Conference on Automated Deduction*, Albany, NY USA LNCS (ed.) D Kapur (Berlin: Springer)

Reif W 1992 Verification of large software systems. *Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi LNCS (ed.) R K Shyamasundar (Berlin: Springer)