Proof Reuse for Deductive Program Verification

Bernhard Beckert and Vladimir Klebanov
Institute for Computer Science
University of Koblenz-Landau
www.key-project.org

Abstract

We present a proof reuse mechanism for deductive program verification calculi. It reuses proofs incrementally (one proof step at a time) and is employs a similarity measure for the points (formulas, terms, programs) where a rule is applied.

The method is flexible, as the reuse mechanism does not need knowledge about particularities of the calculus and its rules. It allows to adapt and reuse proof steps even if the situation in the new proof is merely "similar" but not identical to the template. In case reuse has to stop because a changed part in the new program is reached that requires genuinly new proof steps, reuse can be resumed later on when an unaffected part is reached.

Our method has been successfully implemented within the KeY system to reuse correctness proofs for Java programs.

1. Introduction

The need for proof reuse in software verification. Experience shows that the prevalent use case of program verification systems is not a single proof run. It is far more likely that a proof attempt fails, and that the program (and/or the specification) has to be revised. Then, after a small change, it is better to adapt and reuse the existing partial proof than to verify the program again from first principles. This is of particular advantage for deductive verification systems (which we consider here), where proof reuse reduces the number of required user interactions.

The problem setting. In this paper we present a reuse technique that applies to the following general setting. We assume a specification and an "old" program to be given, as well as a corresponding partial correctness proof for the old program (the "proof template") that is done using a sequent-style (or similar) calculus. Moreover, a (corrected) new program is given (for now, we assume the specification to be unchanged), and our goal is to reuse and adapt as much as possible of the template proof for the construction of a "new" proof for the correctness of the new program.

Features. The main features of our reuse method are:

(1) The units of reuse are single rule applications. That is, proofs are reused incrementally, one proof step at a time (alternative approaches are discussed under "related work", see Section 7). This allows to keep our method flexible, avoiding the need to build knowledge about particularities of the calculus, its rules, and the target programming language into the reuse mechnism.

(2) Proof steps can be adapted and reused even if the situation in the new proof is merely "similar" but not identical to the template.

(3) In case reuse has to stop because a changed part in the new program is reached that requires genuinly new proof steps, reuse can be resumed later on when an unaffected part is reached.

Basic ideas. As usual, we assume the rules of the calculus to be represented by rule schemata. Thus, at each proof step, there are three choices that the reuse facility—like every incremental proof construction method—has to make: (a) the rule (schema) to be applied, (b) the goal/position where it is applied (which we call the "focus" of the rule application), and (c) instantiations for schema variables.

Our goal is to make—if possible—the same choices as in the template proof. But that requires us to generalise and extract the essence of the choices in the old proof such that it can be applied to the (similar but different) situation in the new proof.

For finding the rules that are candidates for choice (a), such a generalisation is readily available. The rule *schemata* are natural generalisations of particular rule applications. They are defined by the developer of the verification calculus, who has the required insight to know what the essence of a rule application is. We then adhere to the overall succession of rule schema applications in the template proof. But, since proofs are not linear, at each point in time there

can still be several candidate rules that compete for being used first.

Choice (b), i.e., the point where a candidate rule is to be applied, is more difficult as it is hard to capture the essence of a formula or sequent. To solve this problem, we define a similarity measure on formulas. Fortunately, there is usually only a moderate number of possibilities, because program verification calculi are to a large degree “locally deterministic”. That is, given a partial (new) proof, there is for most rule schemata only a small number of potential application foci.

Then, the combinations of candidate rules and their potential focus points—which we call reuse pairs in the following—are ordered according to the similarity between the potential new and the template focus points. Thus, the similarity measure both implements the generalisation for choice (b) and is used to prioritise the rule candidates left from choice (a).

Finally, to make choice (c), schema variable instantions are computed by matching the rule schema against the chosen focus of application. Schema variables that do not get instantiated that way, e.g. quantifier instantiations, are simply retained from the old proof.

Adaptability. Our approach is very flexible. The only part that is to some extent adapted to the target calculus is the similarity measure on formulas. But even that does not incorporate any knowledge about particular rules but only some limited information about the target programming language (Java in our case) and general properties of the calculus (e.g. that rules are typically applied at the beginning of a program). Nothing has to be changed if new rules are added to the calculus or even new logical operators and the corresponding rules are added. And no knowledge has to be built into the method about what effects a certain program change has on the structure of its correctness proof.

Implementation. Our method has been successfully implemented within the KeY system [8, 2, 1] to reuse correctness proofs for Java programs (see Sect. 2.1). This implementation can handle many different types of changes in the program to be verified, such as adding/changing/deleting statements, changing (sub-)expressions, changing the control structure (e.g. by adding an if-statement), changing the class hierarchy, and overwriting inherited method implementations. It works well in practical everyday use; and only rarely are old proof attempts reused in a less than optimal way.

Structure of this paper. The rest of this paper is organized as follows. Section 2 gives some more details on the background and motivation of our work. In Section 3, we present the main reuse algorithm. Section 4 describes how the similarity of programs and formulas is measured that is used to evaluate reuse candidates. In Section 5, the choice of initial reuse candidates is discussed (the points in the old proof where reuse can be (re-)started), which is based on the difference between the old and the new program to be verified. Section 6 contains an extended example. In Section 7, we discuss other approaches to proof reuse and related work; and, finally, in Section 8, we draw conclusions and discuss future work.

2. Background

2.1. The KeY Project

The KeY system [8, 2, 1] is a comprehensive environment for integrated deductive software design. Software developed with KeY can be formally proven correct, i.e. to be behaving up to the given specification. In the KeY process, the correctness of programs is formally proven by establishing the validity of Java Dynamic Logic formulas (see Sect. 2.2) generated from the specification and the implementation of a program.

The system is built on top of the commercial CASE tool Together ControlCenter, which is an enterprise-grade platform for UML-based software development. KeY augments this modeling foundation with an extension for formal specification, a verification middle-ware, and a deduction component. Formal software specifications are written in Object Constraint Language (OCL), which is part of the UML standard but enjoys only rudimentary support from CASE tool vendors for now. The KeY extension offers facilities for authoring, rendering and analysis of formal specifications. The verification middle-ware is the link between the modeling and the deduction component. It translates the model (UML), the implementation (Java), and the specification (OCL) into Java Dynamic Logic proof goals, which are passed to the deduction component. The verification middle-ware is also responsible for managing proofs during the development and verification process. The deduction component is a novel Java Dynamic Logic theorem prover, which is used to actually construct proofs for the proof goal.

2.2. Java Dynamic Logic

Dynamic Logic (DL) can be seen to be an extension of Hoare logic (see [6] for an overview). It is a first-order modal logic with a modality $\langle p \rangle$ for every program p (we allow p to be any sequence of Java statements with the only restriction that they must not contain threads). In the semantics of these modalities a world w (called state in the DL framework) is accessible from the current world, if the program p terminates in w when started in the current world. The formula $[\mathrm{p}]\phi$ expresses that ϕ holds in *all* final states

of p, and $\langle p\rangle\phi$ expresses that ϕ holds in *some* final state of p. Considering sequential Java programs, there is exactly one such final state for every initial state (if p terminates) or there is no final state (if p does not terminate). The formula $\phi \rightarrow \langle p\rangle\psi$ is valid if, for every state s satisfying precondition ϕ, a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds.

2.3. The KeY Calculus for Java Dynamic Logic

The programs in Java DL formulas are basically executable Java code. The verification of a given program can be thought of as *symbolic code execution*.

The rules of the Java DL calculus [4, 1] operate on the first *active* command p of a program $\pi p\omega$; it is the focus of their application. The non-active prefix π consists of an arbitrary sequence of opening braces "{", labels, beginnings "`try{`" of `try-catch-finally` blocks, etc. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately. The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the following Java block operating on its first active command `i=0;` then the non-active prefix π and the "rest" ω are the marked parts of the block:

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \}finally\{ k=0; \}\}}}_{\omega}$$

Since there is (at least) one rule schema in the Java DL calculus for each Java programming construct, we cannot present all of them in this paper. Instead, we give a simple but typical example, the rule schema for the `if` statement:

$$\frac{\begin{array}{c}\Gamma,\ b = \text{TRUE} \vdash \langle\pi\ \ p\ \ \omega\rangle\phi \\ \Gamma,\ b = \text{FALSE} \vdash \langle\pi\ \ q\ \ \omega\rangle\phi\end{array}}{\Gamma \vdash \langle\pi\ \ \texttt{if(}b\texttt{)}\ \ p\ \ \texttt{else}\ \ q\ \ \omega\rangle\phi}$$

The rule has two premisses, which correspond to the two cases of the `if` statement. The semantics of this rule is that, if the two premisses hold in a state, then the conclusion is true in that state. In particular, if the two premisses are valid, then the conclusion is valid. Note, that this rule is only applicable if the condition b is known (syntactically) to be free of side-effect. Otherwise, if b is a complex expression, other rules have to be applied first to evaluate b.

The KeY calculus is highly locally deterministic (i.e., there are usually few possible foci for a particular rule to extend a given partial proof). The reasons are that (1) symbolic execution rules only apply at the first ("active") statement of the program, and (2) 2there is no split rule, so active statements do not "multiply".

3. The Main Reuse Algorithm

3.1. Basic Definitions

As usual for deductive program verification systems, we use a sequent-style calculus.

Definition 1 *A* sequent *is of the form* $\Gamma \vdash \Delta$*, where* Γ, Δ *are duplicate-free lists of formulas.*

Definition 2 *A* rule R *is a binary relation between (a) the set of all tuples of sequents and (b) the set of all sequents. The elements* $\langle P_1, \ldots, P_k\rangle$ $(k \geq 0)$ *of* R *are called* rule instances*. If* $R(\langle P_1, \ldots, P_k\rangle, C)$ $(k \geq 0)$*, then the* conclusion C *is* derivable *from the* premisses $P_1, \ldots, P_k$ *using rule* R*.*

We assume that rules are, as usual, represented by *rule schemata* (such as the rule for `if` in Section 2.3). In the following, we identify rules and their schema representations.

Most rules (rule instances) have a *focus*, i.e., a single formula, term, or program part (in the conclusion of the rule) that is modified or deleted by applying the rule. The focus of the if-rule in Section 2.3, for example, is the `if-else` statement. An example for a rule that does not have a focus is the cut rule; it can be applied anywhere.

A *proof* for a goal (a sequent) S is an upside-down tree with root S. In practice, rules are applied from bottom to top. That is, proof construction starts with the initial proof obligation at the bottom and ends with axioms (rules with an empty premiss tuple). A *rule application* (in a proof) consists of a rule instance and a node in the proof tree that is derived from its child nodes using that instance.

3.2. The Reuse Scenario

As said in the introduction, we start with two versions of a program: an "old" one, and a (corrected) "new" one. We also have two proofs in the system: the old (template) proof dealing with the old program—it may or may not be a complete proof—and an incomplete new proof dealing with the new program. At the beginning, the new proof only consists of a single node containing the (new) initial proof goal constructed from the new program and the specification. For now we assume that the specification remains unchanged but an extension to allow such changes is possible.

3.3. The Algorithm

The main reuse algorithm is shown in Table 1. It maintains a list C of rule applications in the old (template) proof, which contains the *reuse candidates*. While reuse progresses and the new proof grows, C always contains those rule

input $\mathit{oldProof}$, $\mathit{oldProgram}$, $\mathit{newProgram}$, $\mathit{specification}$;

$\mathit{newProof}$:= initialProofGoal($\mathit{newProgram}$, $\mathit{specification}$);
C_0 := initialCandidateList($\mathit{oldProof}$, $\Delta(\mathit{oldProgram}, \mathit{newProgram})$);
$C := C_0$;
while $\mathit{newProof}$ has open goals **do**
    $\langle \mathit{candidate}, \mathit{newFocus} \rangle$:= chooseReuse(C, $\mathit{oldProof}$, $\mathit{newProof}$);
    if $\langle \mathit{candidate}, \mathit{newFocus} \rangle \neq \bot$ **then**
        $\mathit{newProof}$:= result of applying rule($\mathit{candidate}$) at $\mathit{newFocus}$ in $\mathit{newProof}$;
        if $\mathit{candidate} \notin C_0$ **then** $C := C \setminus \{\mathit{candidate}\}$; **fi**
        $C := C \cup \{c \mid c$ is a child of $\mathit{candidate}$ in $\mathit{oldProof}\}$;
    else
        $\mathit{newProof}$:= applyRuleWithoutReuse($\mathit{newProof}$);
    fi;
od;

output $\mathit{newProof}$;

Table 1. Main reuse and proof construction algorithm.

applications in the old proof that are currently considered available for reuse.

Initially, the new proof consists of the (new) initial proof goal, and C contains the initial candidates computed by the function initialCandidateList (they are also stored in C_0).

A *possible reuse pair* consists of (1) a candidate rule application and (2) a possible new focus, i.e., a position in a goal sequent of the new (target) proof, where the same rule is applicable. At each iteration step, function chooseReuse (Table 2) is invoked to compute all possible reuse pairs and choose the best one (this choice is mainly based on focus similarity).

The rule of the selected reuse pair is then applied at the target focus, extending the new proof. The candidate is removed from the list C, unless it is an initial candidate (i.e., an element of C_0), in which case it is persistent in C. The reason for making the initial candidates persistent is explained in Section 5. Finally, the children of the reused candidate (in the old proof) become new candidates and are added to C. Thus, the candidates form a "wavefront" through the old proof during reuse.

So far, two very important questions have been left open: How is the quality of possible reuse pairs computed (i.e., how does the function score that is used by chooseReuse work)? And where do the initial candidate proof steps come from (i.e., how does function initialCandidateList work)? These questions are answered in Sections 4 and 5, respectively. Note, that our algorithm is "modular" in the sense that different answers would not affect its overall workings.

While performing reuse, the danger is not only to do too little, but also to do "too much". Sometimes, even though there are possible reuse pairs available, it is better to apply none of them. This is not so odd as it seems, since the existence of a reuse pair signifies little more than a *possible* application of a certain rule. Though activating a "wrong" reuse pair never undermines the correctness of the proof under construction (since only correct rule applications are performed), it is poisonous for reuse. To safeguard against bad reuse, we compare the quality scores of possible reuse pairs to a threshold value ε. In case the score of all possible reuse pairs is below ε—which is an indication that we have reached a situation that is either different or not present in the old proof—a completely new proof step has to be chosen by the user or the automated proof search procedure (this choice is symbolized by calling applyRuleWithoutReuse in the algorithm). Reuse can then be restarted using one of the initial candidates.

For some rules, it is not sufficient to know where they are applied (i.e., what their focus is), but additional information is required. For example, (a) the cut formula has to be known for an application of the cut rule, (b) for induction rules, the induction hypothesis has to be known, and (c) for quantifier rules, the appropriate instantiation has to be provided. Since it would be a very hard task to adapt this kind of information from the old rule application to the new one, we either (1) use what is required in the new proof (as computed by a matching algorithm) or, if matching does not give an answer, (2) use the same information as in the old proof.

4. Evaluating the Quality of Reuse Candidates

Recall, that a possible reuse pair consists of a rule application in the old proof and a focus (formula, term, or pro-

```
function chooseReuse(list C of candidates, oldProof, newProof)
possibleReuses := {};
Goals := open goals of newProof;
foreach c ∈ C do
    foreach g ∈ Goals do
        foreach position p in the sequent of g do
            if rule(c) is applicable at p then
                possibleReuses := possibleReuses ∪ ⟨c, p⟩;
            fi;
        od;
    od;
od;
if possibleReuses = {} then return ⊥ fi;
select ⟨c, p⟩ from possibleReuses with score(⟨c, p⟩) maximal;
if score(⟨c, p⟩) > ε then
    return ⟨c, p⟩;
else
    return ⊥;
fi;
```

Table 2. Function for best possible reuse pair.

gram) in the new proof where the same rule is applicable.

To assess the quality of possible reuse pairs based on similarity scoring is a key part of our reuse facility, since it is one of the most crucial and difficult parts in our effort. We have to distinguish between proof parts that are appropriate for reuse and parts that only seem to be so on first sight. In other words, similarity scoring must prevent misapplication of proof steps from the old proof that are not appropriate for reuse.

When all possible reuse pairs have been computed for an iteration step of the reuse algorithm, we are (usually) left with a choice. Several features may influence the quality of a reuse pair. The first and most important one is the similarity between the application foci in the old and the new proof. We distinguish three kinds of rules:

Program logic rules for symbolic execution: Such rules focus on a program part. The similarity score is determined by comparing these focus programs in the old and the new proof (see Sect. 4.1). The logical parts of the focus formulas are not considered, since they rarely provide additional discriminating evidence.

Analytic first-order logic and rewrite rules: Such rules manipulate a (sub-)formula or term without modifying program parts. A logical similarity analysis of these foci is performed (see Sect. 4.2).

Focus-less rules: No similarity score can be computed for such rules. The score of such a reuse candidate is solely based on other features, in particular proof connectivity (see below).

An additional feature that can be used to score possible reuse pairs (besides similarity of rule foci), is the *connectivity* of the new proof (as compared to the old proof). This criterion gives a bias against tearing apart proof steps that are connected in the old proof. Reuse pairs disrupting connectivity are assigned a small penalty (of -0.1). This is enough to tip the scales in case other features do not provide discrimination between several possible reuse pairs.

To get a single numerical quality value for a reuse pair, we sum up the scores computed for different features.

4.1. Similarity Score for Program Parts

We evaluate the appropriateness of symbolic execution proof steps by comparing the programs that these steps focus on. The comparison is not limited to the focus statement, i.e., the statement that is active and being symbolically executed by the rule application. Although an emphasis is put on this focus statement, the other parts of the focus program are considered as well.

A straightforward way to compare two programs is to compute the minimal edit script for turning one program into the other. Successful experiments with the implementation of our reuse approach show that this is not only an easy but also very useful way to compare programs.

Since, for example, the names of variables, methods, etc.

have no effect on the structure of proofs, we use an abstraction of the actual programs for comparison.

Below, (1) the algorithm for computing the minimal edit script, (2) the program abstraction we use for comparison, and (3) the computation of a numerical similarity score from an edit script are explained in more detail.

Computing the minimal edit script. Myers's classical Longest Common Subsequence (LCS) algorithm [11] can be used to efficiently compute the minimal edit script of two sequences. It takes two sequences $A = a_1\,a_2 \cdots a_N$ and $B = b_1\,b_2 \cdots b_M$ as input, where the a_i and b_j are elements of an arbitrary alphabet, and produces a list of A and B's longest common subsequences. From that list, the minimal edit script for turning A into B can easily be extracted.

An edit script is a list of insertion and deletion commands. The delete command "$x\,D$" deletes the symbol a_x from A. The insert command "$x\,I\,b_1\,b_2 \cdots b_t$" inserts the sequence of symbols $b_1\,b_2 \cdots b_t$ immediately after a_x. The script commands refer to symbol positions in A after the preceding commands have been executed. The length of the script is the number of symbols inserted or deleted.

Program abstraction. The computation of a minimal edit script requires as input two sequences of symbols. To construct such sequences from the two programs that are to be compared, we first linearize the programs into a sequence of statements. Then, the statements are abstracted into *statement signatures*.

The first element of the abstraction of a statement S is the name of S (e.g., *If*, *LocalVarDecl*, *Assignment*). In the following cases, more elements are added to the abstraction:

- If the statement S is also an expression, the static type of the expression is added. If, moreover, S is an assignment whose right operand is a boolean literal, then the value of that literal is appended as well.
- If the statement S is a method invocation, the signature of the method and the name of the class containing the referenced implementation are added.

The statement signatures are defined to abstract from names, expressions, literal values, etc. That is, they are designed to abstract from all features that tend to not influence program control flow (and, thus, proof structure). Consequently, our reuse algorithm can easily deal with such program changes as renaming and changes in literal values.

One could devise more elaborate abstraction schemes. Our experience, though, shows that this only leads to a marginal improvement and is in general not necessary.

From edit script to similarity score. To compute a similarity score for two programs α and β, we have computed a minimal edit script between their abstract representations A and B. Now we must condense this edit script into a single numerical value.

Definition 3 *Let $E(A,B) = e_1\,e_2 \cdots e_n$ be the minimal edit script for the abstractions A, B of programs α, β. Then, the* similarity score *of A, B resp. α, β is defined by*

$$\delta(\alpha,\beta) = \delta(A,B) \quad = \quad -\sum_{e_i \in E(A,B)} P(e_i)$$

where the penalty $P(e)$ for an edit command e is

$$P(e) \quad = \quad \begin{cases} \displaystyle\sum_{k=1}^{t} \frac{0.75}{x+k} & \text{if } e = x\,I\,b_1\,b_2 \cdots b_t \\ \displaystyle\frac{1}{x+1} & \text{if } e = x\,D \end{cases}$$

Note that higher values of $\delta(\alpha,\beta)$ mean higher similarity, and that $\delta(\alpha,\beta)$ is always less than or equal to zero. The maximal value 0 is reached for programs with identical signatures.

The function δ is not symmetrical, i.e., $\delta(A,B)$ differs in general from $\delta(B,A)$. The penalty constants are chosen such that statement insertions are penalized less than deletions. The reason for defining δ in that way is that additional statements in the new program are easier to handle for reuse than missing statements. Deleting statements does usually not simply correspond to deleting proof parts but requires more complex changes of the proof.

Program differences are penalized less the farther they are from the active (first) statement, which is the target of symbolic execution.

Example 1 *Consider the following two programs α and β:*

α :
```
int x; int res;
res = x/x;
```

β :
```
int x; int res;
if (x==0) res=1; else res=x/x;
```

The result of abstracting them into sequences A resp. B of signatures is:

A :
```
LocalVarDecl, LocalVarDecl,
Assignment(int)
```

B :
```
LocalVarDecl, LocalVarDecl,
If, Assignment(int), Assignment(int)
```

The underlined parts correspond to the insertions in the minimal edit script. It consists of the two commands $2\,I$ `If` and $4\,I$ `Assignment(int)`.

The similarity score for the two programs is thus:

$$\delta(\alpha, \beta) = \delta(A, B) \quad = \quad -\frac{0.75}{2+1} + -\frac{0.75}{4+1} = -0.4 \ ,$$

which signifies a medium to high similarity. The score is high enough to reuse the application of the local-variable-declaration rule from the old proof in the new one.

4.2. Similarity Score for First-order Logic Parts

Assessing the quality of possible reuse pairs that do not deal with symbolic program execution is a more difficult challenge. This is due to the lower grade of local determinism of the first-order fragment of the calculus and the high "volatility" of first-order formulas in a proof.

We use two different similarity criteria for formulas and terms. (1) A high bonus ($+1.0$) is added if the foci in the old and the new proof are identical up to variable renaming. Otherwise, a small penalty (-0.2) is added. (2) The two formulas that contain the actual rule application foci are compared in a similar manner as programs: First, formulas are linearized, then names of variables, functions, etc. are abstracted to sorts, and finally a minimal edit script is computed. The script is scored uniformly, with every deletion worth a penalty of 0.1 and every insertion a penalty of 0.05. Additionally, the programs in the formulas contribute their similarity scores with a weight of 0.25.

The results of using these criteria are sufficient for a high ratio of correctly reused rule applications but are not as good as for rule applications with a program part in focus.

5. Finding Reusable Subproofs

Our main reuse algorithm requires an initial list of reuse candidates. These initial candidates, which are rule applications in the old proof, can be seen as the points where the old proof is cut into subproofs that are separately reusable. They are the points where reuse is re-started after program changes required the user or the automated proof search mechanism to perform new rule applications not present in the old proof. The choice of the right initial candidates is crucial for reuse performance.

Since program changes may lead to additional case distinctions in the new proof, it may be necessary to reuse old subproofs repeatedly in the new setting. In order to deal with this necessity, we make the initial candidate proof steps persistent. As shown in Table 1, the *initial* candidates (they are the elements of C_0) are not consumed when they are reused. Thus an initial candidate proof step is always available to seed the corresponding old subproof when needed.

The way initial candidates are computed depends on the way the program (and thus the initial proof goal) has changed. For changes affecting single statements (local changes)

```
int x;
int res;
res=x/x;
```

(a)

```
int x;
int res;
if(x==0) {
   res=1;
} else {
   res=x/x;
}
```

(b)

```
-- old
+++ new
@@ -1,3 +1,7 @@
 int x;
 int res;
+if(x==0) {
+  res=1;
+}else {
 res=x/x;
+}
```

(c)

Figure 1. Change detection with GNU diff: (a) old program, (b) new program, and (c) output of "`diff -uw`".

we extract the differences right from the source files, using the GNU diff utility [5, 11]. It is based on the same algorithm by Myers that we use for program similarity scoring. GNU diff is well-known to produce meaningful change sets for modifications of source files. The output of diff is used to identify common sections of code in the old and the new program. The proof fragments dealing with these common parts are good candidates for reuse; thus, their root nodes are marked as initial reuse candidates.

In the KeY system, the differences between program revisions are provided by the integrated source tracking system based on CVS (which in turn uses GNU diff). Based on that information, markers for initial reuse candidates are automatically inserted by our reuse facility into the (old) proof that is to be reused.

Example 2 *An example for the output of GNU diff is shown in Figure 1. The lines starting with "*`+`*" have been added to the old program. Lines starting with a "*`-`*" (not occurring here) have been removed from the old program. Lines starting with a space are common to both programs.*

In this example, the common program parts start with the statements `int x;` *and* `res=x/x;`*. Thus we scan the old proof top-down and look for proof steps with these statements in focus. The result of this procedure yields two initial reuse candidates for our example.*

Figure 2 shows the prover window of the KeY system. The right part of the window shows the current proof goal (the

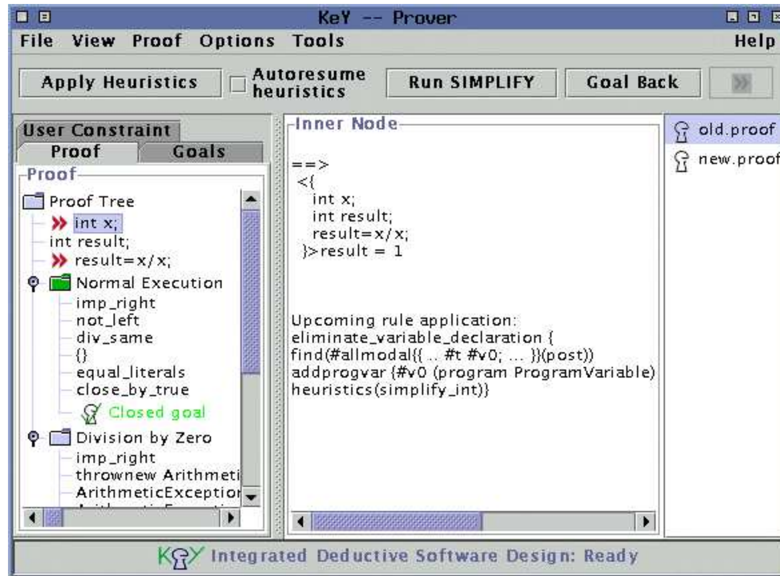

Figure 2. Prover window with markers in the old proof.

initial node of the new proof), which is to show that the new program always terminates with `res` *set to 1. The left part of the window shows a (partial) proof for the old program with the symbol "*`>>`*" marking the initial reuse candidates in the proof tree.*

Our method for detecting initial reuse candidates relies on a certain well-formedness of the programs. Its performance can be impaired, for example, if the programmer puts several statements in one line (which, however, is explicitly discouraged by the official Java Coding Conventions [14]). Given that this is (a) not too common and (b) caused by bad programming style, we did not provide a solution (such as an additional intra-line diff).

Non-local changes, such as renaming of classes or changes in the class hierarchy, cannot be detected in a meaningful way by the standard GNU diff algorithm; other means have to used to signal such changes. In our current implementation, the user indicates these changes to the reuse facility, which then automatically computes the appropriate initial reuse candidates.

6. Extended Example

We will now demonstrate our approach using the two programs from Example 2. For the sake of understandability, we kept the example simple. As a result, these programs do not have a useful purpose. The "old" program is shown in Figure 1 (a). Its intended behavior and specification is that it always terminates with `res` set to 1. The program, however, contains a bug and cannot be proven correct, since an arithmetic exception is thrown on division by zero.[1] Its (unfinished) correctness proof, which in the following will be used as the "old" (template) proof, has one open branch (the "division by zero" branch), corresponding to the case where an exception is thrown. The other branch (the "normal execution" branch) can be closed.

The program is now amended by adding a case distinction, resulting in the "new" program shown in Figure 1 (b). This new program is correct w.r.t. the specification. It always terminates with `res` set to 1. The proof for that consists of a completely "new" branch for the case that `x` is zero, and a "non-zero" subproof that handles the division statement. The latter one contains the two branches from above—here lies the possibility for reuse.

We will trace the first few interesting steps, while slightly simplifying the presentation for clarity (e.g., the connectivity feature is not considered).

As explained in Example 2, our technique for computing initial reuse candidates, in this case gives us two candidates. For now we only consider the first one, namely the rule for variable declarations applied to "`int x;`" in the old proof (the rule of the second initial candidate is not applicable anyway). It has one possible focus in the (new) initial proof goal (it cannot be applied to the second variable declaration, because our Java DL calculus always treats the left-most

[1] In fact, Java initializes the program variable `x` with 0. However, in the remainder of this example, we treat `x` as if it were an input parameter with unknown value.

active statement first):

$$\vdash \langle \texttt{int x; int res;} \\ \texttt{if (x==0) res=1; else res=x/x;} \rangle (\texttt{res} = 1)$$

The similarity score for the single possible reuse pair (see Example 1 for the computation) is -0.4, and reuse is performed. We get the new goal

$$\vdash \langle \texttt{int res;} \\ \texttt{if (x==0) res=1; else res=x/x;} \rangle (\texttt{res} = 1) \quad (1)$$

and a new reuse candidate (the child of the initial candidate in the old proof), which is again an application of the rule for variable declarations, this time applied to "`int res;`". It, also, has one possible focus in the new proof in goal (1). The similarity score for the resulting possible reuse pair is -0.62. That is less than before as there are now less identical parts in the old and the new focus, and the first difference is closer to the active statement. Nevertheless, reuse is still indicated. The resulting new goal sequent is

$$\vdash \langle \texttt{if (x==0) res=1; else res=x/x;} \rangle (\texttt{res} = 1) \quad (2)$$

and the new candidate is the rule handling the assignment "`res=x/x;`" in the old proof (which happens to be identical to the second initial candidate). This candidate, however, is not applicable in (2). We have reached a genuinely new part of the program and, thus, of the proof.

To deal with the new program parts, where no reuse is possible, we manually apply the rules for handling the `if` statement and evaluating its condition (in practice that can be done automatically). The proof tree splits, and we get two subgoals:

$$\texttt{x} = 0 \vdash \langle \texttt{res=1;} \rangle (\texttt{res} = 1) \quad (3)$$

$$\neg(\texttt{x} = 0) \vdash \langle \texttt{res=x/x;} \rangle (\texttt{res} = 1) \quad (4)$$

The single candidate proof step is still the rule application handling "`res=x/x;`" in the old proof. It cannot be applied to (3), as handling an assignments with a literal on the right instead of a division requires the application of a different rule. But the candidate can, of course, be applied to (4). The similarity score for this possible reuse pair is 0.0. The candidate is reused, and (4) is replaced by two new subgoals:

$$\neg(\texttt{x} = 0) \vdash \\ \texttt{x} = 0 \rightarrow \\ (\texttt{res} = div(x, x) \rightarrow \langle\rangle (\texttt{res} = 1)) \quad (5)$$

$$\neg(\texttt{x} = 0) \vdash \\ \texttt{x} = 0 \rightarrow \\ \langle \texttt{throw new} \\ \texttt{ArithmeticException();} \rangle (\texttt{res} = 1) \quad (6)$$

We now have three open goals: (3) is on the "new" branch, (5) is on the "normal execution" branch, and (6) is on the "division by zero" branch. Things get a bit complicated now, as we also obtain two new reuse candidates. Their foci are the sequents that are similar to (5) and (6) in the old proof. Both are applications of the same rule, namely the rule for handling implications (this is a classical logic and not a symbolic program execution rule). The two candidates have possible foci in all three open goals, and we get six possible reuse pairs. Only two of them are really useful (one candidate must be reused at (5) and the other at (6)), and the reuse facility must correctly decide which ones. Fortunately, the two right possibilities have the highest similarity scores and are selected for application (the scores of all six possible reuse pairs range from -0.81 to -0.35).

From here on, reuse continues successfully without further difficulties and the subbranches below (5) and (6) are closed. The "new" branch can then easily be closed with few rule applications.

If this example is done with our implementation in the KeY system (with all details that we left out here), the "old" proof has 191 rule applications, which are all reused.

Note, that the "division by zero" branch with goal (6) could be closed in two steps because, after resolving the implication, the antecedent of the sequent contains the inconsistent formulas $\neg(\texttt{x} = 0)$ and $\texttt{x} = 0$. That is, reuse is not really necessary for this branch, but neither is it harmful.

7. Related Work

Global abstraction methods. An alternative to reusing proofs incrementally, is global proof abstraction. This broad group of methods attempts to capture the overall gist of a given proof and instantiate it for a new problem. Examples are Kolbe and Walther's technique for proving conjectures by induction [9] and the efforts of the Omega Project [7]. To our knowledge, this approach has not been successfully applied to object-oriented software verification.

Constructive methods. A further non-incremental reuse technique is *constructive reuse*. The constructive approach is to analyze the changes made to the proof goal (i.e. the program to be verified) and their effects, and to use this information to identify and reassemble parts of the template proof into a new one. This approach, however, needs to have exact knowledge of all calculus rules and effects of program changes ("when an if-statement is inserted, an application of the if-rule must be added to the proof and, below that, the proof branches ..."). Thus, constructive methods are infeasible for calculi with complex target programming languages (e.g. Java) and a large number of rules.

The software verification system KIV [3], for example, contains a constructive proof reuse facility [13]. It works well as the programs that are verified with KIV are written

in a simple Pascal-like language, and the KIV calculus has only a comparatively small number of program logic rules.

Replay methods. The simplest incremental reuse method is to just replay the (old) proof script. This works well as long as the information in which the new proof must differ from the old proof is not contained in the (linear) script but can be inferred during rule application. An example for such types of information are the instantiations of schema variables, which are computed by a matching algorithm. Significant changes in proof structure, however, cannot be handled by a simple replay mechanism.

A typical example for this kind of reuse is the replay mechanism of the Isabelle theorem prover [12]. It is quite powerful as its proof scripts (usually) contain neither variable instantiations nor the foci of rule applications (which are inferred during rule/tactic application according to simple rules). On the other hand, it cannot automatically cope with changes in proof goal ordering or resume reuse after an intermittent failure.

Similarity guided methods. Melis and Schairer pursue another variation of replay [10]; this time specifically for reuse of subproofs in the verification of invariants of reactive systems, which are specified using first-order logic. Due to symmetries and redundancies in the state space, such proofs give rise to many similar subproofs.

Melis and Schairer's approach identifies a suitable previously solved subproblem via a similarity measure on first-order formulas and replays the stored subproof straight on.

This method is related to our work as it operates under the assumption that similar situations (proof goals) warrant similar actions (rule applications or sub-proofs). The similarity assessment though is performed only once, which is justifiable by a simpler setting.

8. Conclusions and Future Work

We have presented a new reuse method that works well for program verification calculi. It is very flexible as nothing has to be changed if new rules are added to the calculus or even new logical operators and the corresponding rules are added. And no knowledge has to be built into the method about what effects a certain program change has on the structure of its correctness proof.

Our method has been successfully implemented within the KeY system [8, 2, 1] to reuse correctness proofs for Java programs. It requires no modification even as the calculus is constantly evolving.

Future work includes an extension and adaptation to the case where not only the program to be verified but also the specification may change.

It may be useful to use a tree editing distance algorithm for the similarity measure (e.g. [15]). This would allow to take the tree-like structure of programs into account.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling (SoSysM)*, pages 1–42, 2004. To appear. Available at http://www.springerlink.com.

[2] W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In R.-D. Kutsche and H. Weber, editors, *Proceedings, International Conference on Fundamental Approaches to Software Engineering (FASE), Grenoble, France*, LNCS 2306, pages 327–330. Springer, 2002.

[3] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Proc., Fundamental Approaches to Software Engineering (FASE)*, LNCS 1783. Springer, 2000.

[4] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[5] S. D. Gathman. GNU Diff for Java. Available at www.bmsi.com/java/#diff, 2003.

[6] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[7] X. Huang, M. Kerber, J. Richts, and A. Sehn. Planning mathematical proofs with methods. *Journal of Information Processing and Cybernetics (EIK)*, 30, 1994.

[8] KeY Project. Website at www.key-project.org.

[9] T. Kolbe and C. Walther. Reusing proofs. In *European Conference on Artificial Intelligence*, pages 80–84, 1994.

[10] E. Melis and A. Schairer. Similarities and reuse of proofs in formal software verification. In *Proceedings, European Workshop on Advances in Case-Based Reasoning (EWCBR), Dublin, Ireland*, LNCS 1488, pages 76–78, 1998.

[11] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

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

[13] W. Reif and K. Stenzel. Reuse of proofs in software verification. In *Proceedings, Foundations of Software Technology and Theoretical Computer Science*, LNCS 761, pages 284–293. Springer, 1993.

[14] Sun Microsystems, Inc. *Code Conventions for the Java Programming Language*. Available at java.sun.com/docs/codeconv.

[15] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.