Proof Reuse for Program Verification Calculi

Bernhard Beckert and Vladimir Klebanov

University of Koblenz-Landau
Institute for Computer Science
beckert@uni-koblenz.de, vladimir@uni-koblenz.de

Abstract. We present a similarity-based proof reuse mechanism for a certain group of program verification calculi that requires very little knowledge about individual rules of the calculus at hand. This makes our technique well-suited for calculi with complex target languages, a large number of rules, and/or evolving rule sets.

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, which we do not consider here) has to be revised. Then, for a small amendment, it is often more advantageous to adapt and reuse the existing (partial) proof than to verify the program again from first principles. In this case proof reuse pays off through saved user interaction. We present such a technique.

The reuse scenario. 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 a stub of new proof dealing with the new program. The proof obligation are, in general, constructed from the program and the specification, which we assume unchanged.

Features of our technique. (1) The units of reuse are single rule applications. That is, proofs are reused incrementally, one proof step at a time. (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 genuinely new proof steps, reuse can be resumed later on when an unaffected part is reached.

Implementation and adaptability. Our technique is at work within the KeY system [4, 1], which is an environment for formal design and verification of Java software. The KeY theorem prover employs a sequent calculus for Java Dynamic Logic [3, 1]. JavaDL is a modal logic, with programs embedded directly into formulas. The verification of a given program can be thought of as *symbolic code execution* plus induction. The calculus contains several hundred rule schemata and is constantly evolving, as new language features are covered. Our reuse scheme has the advantage of being agnostic of this.

Important is merely the fact that most rules have a *focus* (i.e. a single formula, term, or program part that is modified or deleted by applying the rule), and that the calculus is highly *locally deterministic* (i.e. there are usually few possible foci for a particular rule to extend a given partial proof). The KeY calculus has a high degree of local determinism because (1) symbolic execution rules only apply at the first (*active*) statement of the program, and (2) there is no split rule, so active statements do not "multiply". Our technique should be applicable to any calculus with similar properties, e.g. a Hoare logic with a restricted application of the split rule.

Basic idea. 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 (the *focus*), 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 generalize 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 choice (a), we try to 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 schema is to be applied, is more difficult as it is hard to capture the essence of a term, 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. Finally, to make choice (c), schema variable instantiations 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 copied verbatim from the old proof.

The main algorithm. The main reuse algorithm is shown in Figure 1. It maintains a list C of rule applications in the old (template) proof, which contains the *reuse candidates*, i.e. those rule applications in the old proof that are currently considered available for reuse. A *possible reuse pair* consists, in contrast, of (1) such 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 schema is applicable. At each iteration step, the function chooseReuse (Figure 2) is invoked to compute all possible reuse pairs and choose the best one. This accomplishes choice (b) and finalizes choice (a). The selection is mainly based on rule focus similarity and described below.

The rule of the best 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. Finally, the

```
input oldProof, oldProgram, newProgram, specification;
newProof := initialProofGoal(newProgram, specification);
C_0 := initialCandidateList(oldProof, Δ(oldProgram, newProgram));
C := C_0;
while newProof has open goals do
    ⟨candidate, newFocus⟩ := chooseReuse(C, oldProof, newProof);
    if ⟨candidate, newFocus⟩ ≠ ⊥ then
        newProof := result of applying rule(candidate) at newFocus in newProof;
        if candidate ∉ C_0 then C := C \ {candidate}; fi
        C := C ∪ {c | c is a child of candidate in oldProof};
    else
        newProof := applyRuleWithoutReuse(newProof);
    fi;
od;
output newProof;
```

Fig. 1. Main reuse and proof construction algorithm

children proof steps of the reused candidate (in the old proof) become new candidates and are added to C. Thus, the candidates form a "wavefront" expanding through the old proof during reuse.

So far, the following question has been left open: where do the initial candidate proof steps come from (i.e. how does the function initialCandidateList work)? The initial candidates are computed from the results of syntactic difference detection between the old and the new program. Our calculus makes it easy to identify starting points for the subproofs, which correspond to the common code sections of the two versions. The availability of several such initial candidates is the key to the power of our reuse technique. When the reuse of a subproof fails (due to differences in programs), they allow to restart reuse of unaffected subproofs.

Evaluating the quality of reuse pairs. We evaluate the appropriateness of symbolic execution proof steps by comparing the programs in the formulas that these steps focus on. After a series of experiments we have settled on the minimal edit script (for a certain program representation) as the basis of similarity. The minimal edit script can be efficiently computed with the classical Longest Common Subsequence (LCS) algorithm by Eugene Myers [6].

An important part of similarity computation is the abstraction stage, which transforms programs into sequences of symbols beyond just characters. Statements are condensed into *statement signatures*, which are defined to abstract from most names, expressions, literal values — as far as they presumably do not influence program control flow and, thus, proof structure. Consequently, our reuse algorithm can easily deal with such program changes as renamings and changes in literal values. The signature of a statement S is the name of S (e.g.

```
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;
```

Fig. 2. Function for best possible reuse pair

If, *LocalVarDecl*, *Assignment*). If S is also an expression, the static type of the expression is appended to the signature. If, moreover, S is an assignment whose right operand is a boolean literal, then the value of that literal is added as well.

Now let $E(A, B) = e_1\, e_2 \cdots e_n$ be the minimal edit script for the abstractions A, B of program α, β. Then, the *similarity score* of A, B resp. α, β is defined by $\delta(\alpha, \beta) = \delta(A, B) = -\sum_{e_i \in E(A,B)} P(e_i)$ where the penalty $P(e)$ for an edit script command e is $P(e) = \sum_{k=1}^{t} \frac{0.75}{x+k}$, if e inserts symbols $b_1\, b_2 \cdots b_t$ after the one at position x, and $P(e) = \frac{1}{x+1}$, if e deletes the symbol at position x.

Higher values of $\delta(\alpha, \beta)$ mean higher similarity; the maximal value 0 is reached for programs with identical signatures. Penalty constants are chosen so that statement insertions are penalized less than deletions, since deleting statements does not usually simply correspond to deleting proof parts but requires more complex changes of the proof. Furthermore, differences are penalized less the farther they are from the active (i.e. first) statement.

Since the existence of a reuse pair signifies little more than a mere *possibility* of applying a rule, we also install a quality threshold ε to safeguard against misapplication. Though activating a "wrong" reuse pair never undermines the correctness of the proof under construction (as only correct rule applications are performed), it is poisonous for orderly reuse. In case the score of all possible reuse pairs is below ε—which is an indication that we have a situation that is either significantly different or not present in the old proof—a completely new proof step has to be chosen. Reuse can be restarted later on when appropriate.

The procedure for evaluating appropriateness of analytic first-order and rewrite rule candidates is more complicated (since this calculus fragment is less locally deterministic), but goes in the same key as the one described above. We also make use of an additional criterion of *proof connectivity*, which is a side measure against accidentally tearing apart related proof steps.

Related work. The simplest reuse method is to just replay the old (linear) 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 script but can be inferred during rule application. A typical example for this kind of reuse is the replay mechanism of the Isabelle theorem prover [7]. It is simple and powerful, but cannot automatically cope with changes in proof goal ordering or resume reuse after an intermittent failure.

The software verification system KIV [2] contains a proof reuse facility [8] that has knowledge on how program changes affect proof structure. This allows it to identify and rearrange whole proof parts. Engineering such knowledge for a very complex target language like Java — KIV deals with a simple Pascal-like language — would be quite expensive.

Melis and Schairer [5] use a similarity measure on first-order formulas to reuse subproofs in verification of invariants of reactive systems, which are specified using first-order logic. Due to symmetries/redundancy in the state space, this problem setting gives rise to many similar subproblems. For each, the similarity measure is applied once to identify a suitable, previously solved subproblem and replay the stored subproof.

There is also a number of proof planning and further proof abstraction methods, which, to our knowledge, have not been successfully applied to object-oriented software verification.

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*, 3, 2004. To appear.
2. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *Proceedings, FASE*, LNCS 1783, 2000.
3. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In *Java on Smart Cards: Programming and Security*, LNCS 2041, 2001.
4. KeY Project. Website at `www.key-project.org`.
5. E. Melis and A. Schairer. Similarities and reuse of proofs in formal software verification. In *Proceedings, EWCBR*, LNCS 1488, 1998.
6. E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2), 1986.
7. L. C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. 1994.
8. W. Reif and K. Stenzel. Reuse of proofs in software verification. In *Proceedings, FST & TCS*, LNCS 761, 1993.