Reusing Proofs when Program Verification Systems are Modified

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

Abstract

In this position paper, we describe ongoing work on reusing deductive proofs for program correctness when the verification system itself is modified (including its logic, its calculus, and its proof construction mechanism).

We build upon a method for reusing proofs when the program to be verified is changed, which has been implemented within the KeY program verification system and is successfully applied to reuse correctness proofs for Java programs.

1. Motivation

Proof reuse in program verification is mostly thought of as a means to more easily construct a proof for the correctness of some program p in cases where a proof for a similar program p' (or the same program p with a slightly different specification) is already available.

If proofs are used as certificates for the correctness of programs, however, there is an even more important reason for reuse. One has to be able to reuse proofs in case the proof *system* is modified. While changing the program that is to be verified has only local effects in that only the proofs for that particular program are invalidated, modifying the verification system globally affects and potentially invalidates *all* proofs done so far. Since proofs that serve as certificates for program correctness need to be maintained over a longer period, possibly over many years, modifications to the proof system are to be expected over the lifetime of proofs. Thus, being able to reuse proofs when the system is modified is an indispensable feature of any program verification infrastructure that is put to serious practical use.

In [5], we have presented a method for reusing correctness proofs when the program changes. That method has been successfully implemented within the KeY system [7, 1] (see Sect. 2) to reuse correctness proofs for Java programs. It 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.

In this position paper, we describe ongoing work on the extension of our method to handle modifications of the verification environment instead of the programs to be verified. Possible modifications we consider include changes to the program logic used in the system, changes to the rules of the verification calculus, changes to the language used to represent these rules, and changes to the deductive engine of the proof system.

Note that we are not formalizing our method in a metalogic, as it's not helpful to achieve a working solution. Our subject is a particular kind of proof search procedure, and only valid proof objects can be constructed in any given prover version anyway (this also applies to loading a proof from a file). In a sense, the problem we are looking at is not one of logics.

2. Background

The KeY Project. The KeY system [7, 1] is a comprehensive environment for integrated deductive software design. Software developed with KeY can be formally proven correct, i.e., 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 generated from the specification and the implementation of a program. This correctness is asserted by an explicit proof object.

The system is built on top of the CASE tool Borland Together ControlCenter, which is an enterprise-grade platform for UML-based software development. A version integrated with the popular open IDE Eclipse is also available. KeY augments this modeling foundation with an extension for formal specification, a verification middleware,

and a deduction component. Formal software specifications are written either in Object Constraint Language (OCL), which is part of the UML standard, or Java Modeling Language (JML). The KeY extension offers facilities for authoring, rendering and analysis of formal specifications. The verification middleware is the link between the modeling and the deduction component. It translates the model (the class diagram), the implementation (Java), and the specification (OCL/JML) into Java Dynamic Logic proof goals, which are passed to the deduction component. The verification middleware is also responsible for managing proofs during the development and verification process. The deduction component is a novel Java Dynamic Logic theorem prover that is used to actually construct proofs for the proof goal.

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 $\langle p \rangle \phi$ 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.

The KeY Calculus for Java Dynamic Logic. As usual for deductive program verification systems, we use a sequent-style calculus. 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 [3, 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, 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 like `throw` and `return` can be handled appropriately. The postfix ω denotes the "rest" of the program, i.e., everything except the prefix and the part of the program the rule operates on.

Since there is (at least) one rule schema in the Java DL calculus for each Java programming construct, we can here only give a simple but typical example, the rule schema for the `if` statement:

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

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-effects. Otherwise, if b is a complex expression, other rules have to be applied first to evaluate b.

The Taclet Mechanism. The KeY system provides a formalism for implementing rules (resp. rule schemata) called *taclets* [4]. As the name suggests taclets can be considered as lightweight, stand-alone tactics. They have a simple syntax and semantics and have means to represent explicitly (i) the pure logical content of a rule; (ii) restrictions or *guards* on the expected context and position of a rule application; (iii) heuristic information on whether and when a rule is applied automatically/interactively. Here is the same rule as above formulated as a taclet:

find $\langle \pi\ \ \texttt{if(}b\texttt{)}\ \ p\ \ \texttt{else}\ \ q\ \ \omega \rangle \phi$
replacewith $(\langle \pi\ \ p\ \ \omega \rangle \phi)$ *add* $(b = \text{TRUE} \vdash\ )$;
replacewith $(\langle \pi\ \ q\ \ \omega \rangle \phi)$ *add* $(b = \text{FALSE} \vdash\ )$

The KeY Proof Format. The KeY prover stores its proofs as a proof script consisting of a stream of rule names, application positions (as index into the sequent) and explicit schema variable instantiations if these cannot be inferred from the sequent. This format avoids excessive inclusion of formulas in the file, since these include programs and can be quite lengthy. On the other hand, this design does not perform gracefully if—for some reason—the currently constructed proof object does not match the form expected in the script.

3. Proof Reuse for Program Changes

In this section, we briefly describe our method presented in [5], which allows to reuse proofs when the program to be verified changes. It forms the basis for our work on reusing proofs when the verification system is modified.

The Need For Proof Reuse Upon Changes in Programs. 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, see Section 5.2) 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.

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.

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 mechanism.

(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.

Basic Ideas. The rules of the calculus are represented by rule schemata (taclets). 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 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 finding the rules that are candidates for choice (a), such a generalization is readily available. The rule *schemata* (i.e., the schematic representations of the rules) are natural generalizations 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 use a syntactical 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.

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.

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.

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) we extract the differences right from the source files, using the GNU diff utility. Non-local changes, such as renaming of classes or changes in the class hierarchy, cannot be detected in a meaningful way by the standard diff algorithm; the user has to announce the changes separately. We are also investigating application of the recently emerged techniques for difference detection in object-oriented programs [2].

Adaptability. Our reuse 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).

4. Proof Reuse for Modifications of the Verification System

4.1. Recertification Strategy Catalogue

In this section, we discuss different modification scenarios that we have encountered during the six years of development of the KeY system. All verification systems are subject to evolution, and the ones that persistently store proof-relevant information (proof scripts, lemmas, abstractions, program invariants, etc.) have to deal with similar problems as we did. We believe that developers of other deductive verification systems can profit from our experiences.

We classify the recertification strategies that are possible responses to modifications of the verification environment as follows:

A No action necessary. The old proof can be loaded without modification.

B Automated recertification with the help of additional information that has been provided at time of change.

C Machine-supported recertification while necessary additional information is inferred during the process. In this case we assume that the old proof can be loaded into the system with the corresponding set of rules.

D Same as C, but in case the old proof cannot be loaded into the system. Then, just the information available in the stored proof file is available.

E No action possible/intended.

$$\frac{\begin{array}{c}\Gamma,\ a = null\ \vdash\ \langle\pi\ \texttt{NPE;}\ \omega\rangle\phi \\ \Gamma,\ a \neq null \wedge (i < 0 \vee i \geq a.length)\ \vdash\ \langle\pi\ \texttt{AOBE;}\ \omega\rangle\phi \\ \Gamma,\ a \neq null \wedge i \geq 0 \wedge i < a.length\ \vdash\ \{a[i] := val\}\langle\pi\ \omega\rangle\phi\end{array}}{\Gamma\ \vdash\ \langle\pi\ \ a\texttt{[}i\texttt{]=}val\ \ \omega\rangle\phi}$$

$$\frac{\begin{array}{c}\Gamma,\ a = null\ \vdash\ \langle\pi\ \texttt{NPE;}\ \omega\rangle\phi \\ \Gamma,\ a \neq null \wedge (i < 0 \vee i \geq a.length)\ \vdash\ \langle\pi\ \texttt{AOBE;}\ \omega\rangle\phi \\ \boxed{\Gamma,\ a \neq null \wedge i \geq 0 \wedge i < a.length \wedge \neg storable(val, a)\ \vdash\ \langle\pi\ \texttt{ASE;}\ \omega\rangle\phi} \\ \Gamma,\ a \neq null \wedge i \geq 0 \wedge i < a.length \boxed{\wedge storable(val, a)}\ \vdash\ \{a[i] := val\}\langle\pi\ \omega\rangle\phi\end{array}}{\Gamma\ \vdash\ \langle\pi\ \ a\texttt{[}i\texttt{]=}val\ \ \omega\rangle\phi}$$

Figure 1. A rule for array assignment: initial and revised version. Differences are boxed.

In the following discussion of different types of modifications, we indicate what we believe is the right kind of strategy to handle the respective modification. We currently work on developing methods for strategies B, C, and D.

4.2. Changes of the Logic Syntax

The presence of a rich program and logic vocabulary within the same formula makes designing a usable and at the same time parseable logic syntax quite a challenge. Several iterations were necessary to obtain a satisfactory solution.

The quantifier notation of `exists x:int.prop(x)` was changed in order to allow fully qualified sort names, as in `\exists java.lang.Object o; prop(o)`. The diamond modality notation had to be modified from the simple `<program>formula` to `\<program\>formula` in order to allow `a<b` in place of `lt(a,b)`. Proposed recertification strategy: B, as one could parse old proof versions with the associated old parser and then transform the abstract representation into the new format. Furthermore, stored KeY proofs rarely mentions formulas explicitly.

4.3. Changes of the Taclet Language

The taclet language used to define the rules of the KeY prover is also subject to change. As clashes between taclet declaration keywords and JAVA identifiers became apparent, an escaping mechanism was put in place (`find` $\rightsquigarrow$ `\find`). Altogether this kind of change is transparent in the stored proofs, as these only reference taclet names. Recertification strategy: A. The semantics of the taclet language has turned out to be exceedingly stable.

4.4. Changes in Parser/Disambiguation

Between the levels of syntax and semantics are changes in parsing and disambiguation of logical expressions. An example is a modification of the associativity of logic operators. The interpretation of the expression $A \wedge B \wedge C$ has changed from $(A \wedge (B \wedge C))$ to $((A \wedge B) \wedge C)$. In addition, the precedence between the state update operator and arithmetic operators were changed in favor of the update so that $\{update\}a + b$ evaluates to $(\{update\}a) + b$ instead of the former meaning $\{update\}(a + b)$. Recertification strategy: B. The old parser can be used to produce an AST, from which an equivalent linearization for the new parser can be generated using explicit brackets.

4.5. Changes in Formalization of the JAVA Language Semantics

Sometimes minor errors in the symbolic execution rules of the KeY calculus have to be fixed. This cannot be ruled out, since one can never arrive from an informal specification at a formal one by formal means. The KeY project on regular bases performs the only measure suitable to mitigate this: cross-checking our rules with other formalizations of JAVA. A recent check of this kind [8] has discovered a missing case in our array assignment rule. The erroneous rule and its correction are presented in Figure 1. Recertification strategy: C. Small local changes allow a similarity-guided proof reuse.

4.6. Changes in the Logical Structure of the Rules

At one point all rules containing a potential case distinction have been reformulated from the form (here's an example)

$$\frac{\Gamma\ \vdash\ ((a > b) \rightarrow \langle\pi\ \ l\ \texttt{=}\ \texttt{true;}\ \omega\rangle\phi) \wedge (\neg(a > b) \rightarrow \langle\pi\ \ l\ \texttt{=}\ \texttt{false;}\ \omega\rangle\phi)}{\Gamma\ \vdash\ \langle\pi\ \ l\ \texttt{=}\ a\ \texttt{>}\ b\ \ \omega\rangle\phi}$$

to a form employing a conditional formula

$$\frac{\Gamma \vdash \mathit{if}\,(a > b) \quad \langle \pi\ l = \texttt{true;}\ \omega\rangle\phi\ \mathit{else}\ \langle \pi\ l = \texttt{false;}\ \omega\rangle\phi\ \mathit{fi}}{\Gamma \vdash \langle \pi\ l = a > b\ \omega\rangle\phi}$$

which has the advantage that one has to reason about the condition only once. Recertification strategy: C, same as above.

4.7. Changes in the Execution Engine

As noted in Section 2, it is important that the expected shape of the proof object implicit in the proof script matches the actual construction. This concordance can be disrupted if the execution engine of the prover is either not deterministic or is purposefully changed.

Ordering of Proof Branches/Formulas. One degree of freedom left by the calculus is the way (i.e., position of) formulas are added to the sequent, and the ordering of the newly generated subgoals whenever a rule has several premisses.

Recertification strategy: D, as it's not possible to load the old proof with a changed system.

The Link to the Program Model. The method call rule of the KeY calculus simulates dynamic binding by a case distinction over all possible classes that offer a suitable implementation of the called method. The ordering of branches is a potential nondeterminism source, which has been eliminated recently by applying alphabetical sorting. Recertification strategy in case of change: D.

Changes in Logic Data Structures. Since stored proofs contain numerical indices into the internal representation of logical entities (formulas, terms), changes in this data structure affect the loading of proofs. Except one such change, the representation has remained stable so far. Recertification strategy: D as this problem class is similar to the one dealing with the ordering of proof branches and formulas.

5. Further Related Issues

5.1. Changes of the JAVA Platform

In spite of Sun's policy of upward source compatability, new versions of the JAVA platform may bring changes to the semantics of existing programs. We want to mention here the introduction of new keywords and APIs (code has to be rewritten to avoid clashes), bugfixes and updates to the JAVA libraries (e.g., fixing the method `StringBuffer.append(StringBuffer)` to be thread-safe or method `BigInteger.isProbablePrime(int)` not to report false for certain primes), or revisions of the JAVA Memory Model (as proposed in JSR 133). The range of necessary recertification actions stretches from A to E, depending on the particular case (e.g., whether the specification or the implementation of library methods was used in proofs, etc.).

5.2. Changes in Program Specification

The complementary case of a change in the program is a changing specification. While the same techniques of proof reuse should be applicable to some extent, a specification is usually a higher-level description, and small changes are likely to lead to bigger disruptions in proofs. We will not pursue this issue here, as it affects individual problems only.

6. Conclusion

We have discussed possible modifications occurring during the development and evolution of software verification systems. A similarity-based method for proof reuse, which has already been successfully implemented in the KeY system, has been presented that handles program changes. Currently we extend and adapt this method to implement recertification strategies of types B, C, and D.

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)*, 4:32–54, 2005.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 2–13, Linz, Austria, September 2004.

[3] 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.

[4] B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer, and S. Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.

[5] B. Beckert and V. Klebanov. Proof reuse for deductive program verification. In J. Cuellar and Z. Liu, editors, *Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China*. IEEE CS Press, 2004.

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

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

[8] K. Trentelman. Proving correctness of JAVACARD DL taclets using Bali. In B. Aichernig and B. Beckert, editors, *Proceedings, 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE CS Press, 2005.