Universität Karlsruhe
Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme

Diploma Thesis

Proof Re-Use in Java Software Verification

Vladimir Klebanov

August 19th, 2003

Responsible Supervisor: Prof. Dr. P.H. Schmitt
Supervisor: Dr. B. Beckert

Contents

1 Context of the Work **7**
- 1.1 The KeY System . . . 7
- 1.2 Java Dynamic Logic . . . 8
- 1.3 The KeY Calculus . . . 9
  - 1.3.1 Sequent Calculus Foundations . . . 9
  - 1.3.2 Taclets as Means of Deduction . . . 10
  - 1.3.3 Reasoning about Programs with Taclets . . . 12
  - 1.3.4 Constructing Proofs with Taclets . . . 13

2 Motivation **15**
- 2.1 Why Re-Use Proofs? . . . 15
- 2.2 Goals of This Work . . . 15
- 2.3 Previous/Related Approaches . . . 17
  - 2.3.1 Analogical Transfer of Verification Proofs . . . 17
  - 2.3.2 KIV: Karlsruher Interactive Verifier . . . 18
  - 2.3.3 Proof Re-Use with Isabelle . . . 19

3 Re-Use Approaches **21**
- 3.1 The Feature-Oriented Approach . . . 21
- 3.2 Feature Overflow . . . 22
- 3.3 The Featureless Approach . . . 24
- 3.4 Feasability of Re-Use . . . 25

4 The Observing Re-Use Technique **28**
- 4.1 Overall System Conception . . . 28
- 4.2 Basic Re-Use Principle . . . 29
- 4.3 The Main Backend Algorithm . . . 31
- 4.4 Re-Use Unit Quality Assessment . . . 34
  - 4.4.1 Quality Assessment Framework . . . 34
  - 4.4.2 1st: “Home-Brewed” Similarity Score . . . 36
  - 4.4.3 2nd: LCS Difference Detection Score . . . 37
  - 4.4.4 3rd: Tree Correction Score . . . 43
  - 4.4.5 Refinement: Connectivity . . . 46

4.4.6 Quality Assessment Of Purely First-Order Re-Use Units . . 50

5 The Frontend **51**
5.1 Local Changes . . . 51
5.1.1 Unified Diff Format . . . 51
5.1.2 Translating Diffs into Markers . . . 53
5.1.3 Additional Remarks on Diff Management . . . 54
5.2 Non-Local Changes . . . 55
5.2.1 Renaming . . . 55
5.2.2 Changes in Class Hierarchy . . . 55

6 Re-Use Scenarios **57**
6.1 Discussion of Possible Program Changes: Local Changes . . . 57
6.1.1 Inserting a Non-Branching Statement . . . 57
6.1.2 Inserting a Statement: Conservative Branching . . . 58
6.1.3 Inserting a Statement: Full Branching . . . 60
6.1.4 Deleting a Statement . . . 62
6.2 Discussion of Possible Program Changes: Non-Local Changes . . . 62
6.2.1 Renaming . . . 62
6.2.2 Model Changes . . . 63
6.2.3 Changes in Specification . . . 64

7 Implementation **67**
7.1 Overview of the Implementation . . . 67
7.2 LCS Similarity Function . . . 67
7.3 Detecting Model Changes . . . 68

8 Conclusion; Perspective **69**

A Selected Taclets of the KeY Calculus **70**

B An Example **73**

References **76**

List of Figures

1.1 Architecture of the KeY system 8

3.1 Object-oriented target language causes complex proof shapes . . . 24

4.1 The re-use facility: components and responsabilities 28
4.2 Assessing program similarity with LCS diff score 39
4.3 Symbolic execution of an `if` statement 43
4.4 Tree correction for programs 44
4.5 Connectivity in proof re-use (in brackets: re-use order) 47

5.1 Change detection with GNU diff 52
5.2 Prover window with markers in the old proof 52

6.1 A program change not leading to branching 58
6.2 The proof branches conservatively for this program change 59
6.3 The proof branches fully with this program change 60
6.4 Model change: adding a method implementation 63
6.5 Method invocation depends on the model 65

Acknowledgment

I would like to thank everyone who encouraged and supported me in creation of this thesis. I thank Prof. Dr. P.H. Schmitt and Dr. Bernhard Beckert for their supervision. I'm very grateful to Dr. Bernhard Beckert for his guidance and many stimulating discussions. Furthermore, I wish to thank all people who shared with me their scientifical opinion, technical advice, and emotional support. My gratitude goes especially to the staff members of the KeY group in Karlsruhe and Göteborg — it was a pleasure working with you.

1 Context of the Work

The work presented in this thesis was carried out in context of the KeY project [KeY]. The goal of the KeY project is to integrate formal methods with object-oriented software development techniques, thus aiding development of high quality programs.

1.1 The KeY System

The KeY system 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 formulae generated from the specification and the implementation of a program.

The system is built on top of a commercial CASE[1] tool Together ControlCenter (TogetherCC)[2], which is an enterprise-grade platform for UML based software development. The overview of KeY's architecture is given in Figure 1.1 on the following page. The system contains the following components:

- A modeling component. The CASE tool TogetherCC provides the functionality for UML modeling and project development — e.g. debugging, refactoring, version control and much more.
- A TogetherCC extension for formal specification. 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 middleware. The verification middleware 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 obligations, which are passed to the deduction component. The

[1] Computer Aided Software Engineering.
[2] Formerly from TogetherSoft, now acquired by Borland.

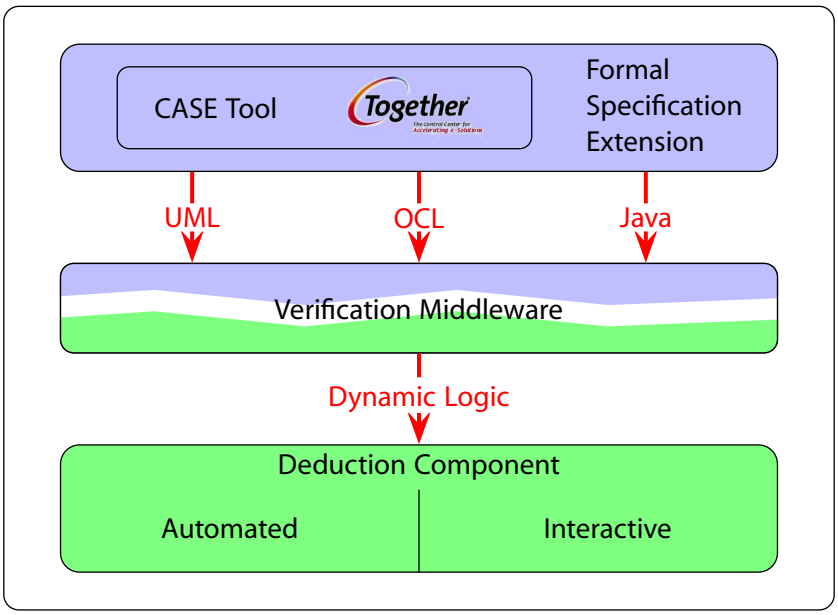


Figure 1.1: Architecture of the KeY system

verification middleware is also responsible for storing and managing proofs during the development process.

- The deduction component. The KeY prover is a Java Dynamic Logic theorem prover, which is used to actually construct proofs for the proof obligations generated by the verification component. Java Dynamic Logic is discussed below.

1.2 Java Dynamic Logic

According to [Bec01], Java Dynamic Logic (Java DL) can be seen as a modal logic with the modalities $\langle p \rangle$ ("diamond") and $[p]$ ("box") for every program p (we specify programs in the next definition). The modality $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program p. In standard DL there can be several such states (worlds) because the programs can be non-deterministic. In Java DL — since Java programs are deterministic — there is exactly one such world (if p terminates) or there is no such world (if p does not terminate). The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state, in which ϕ holds. In contrast, the formula $[p]\phi$ asserts that, if the program p terminates, then in a state satisfying ϕ.

A 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. Thus, the formula $\phi \rightarrow [p]\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$, while $\phi \rightarrow \langle p \rangle \psi$ implies the total correctness of p. We will not give a formal description of the syntax and semantics of Java DL.

Definition 1.2.1 (Program, Context, Java model) If not explicitly stated otherwise, when referring to a *program* we mean a series of Java statements. The non-executable parts of a Java program — class, interface, field and method declarations, etc. — are considered *context*. It is assumed that the proper context embedding is known for every statement. We will call the context in its entirety also a *Java model*. $\triangleleft$

Finally, we state the following naming conventions concerning different types of formulae. We will talk about "purely first-order formulae" when referring to DL formulae lacking modalities. In contrast, a plain "DL formula" is implied to have a non-empty modality, unless explicitly stated otherwise. Besides, since total program correctness is expressed by a "diamond formula" $\langle p \rangle \phi$, where ϕ is a purely first-order formula, the term "diamond" is used synonymously with "modality". Our remarks should be applicable analogously in the case of the box modality.

1.3 The KeY Calculus

1.3.1 Sequent Calculus Foundations

The KeY system's means of reasoning is a sequent calculus for the Java dynamic logic.

Definition 1.3.1 (Sequent) A *sequent* is an enumeration of Java Dynamic Logic formulae of the form

$$\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n \quad (m, n \geq 0)$$

The formulae on the left-hand side of the sequent symbol $\vdash$ are called *antecedent* while the formulae on the right-hand side are called *succedent*. The semantics of a sequent is that of the universal closure of the formula

$$(\phi_1 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n) \quad (m, n \geq 0)$$

$\triangleleft$

Definition 1.3.2 (General sequent calculus rule) A sequent calculus rule is a rule schema of the form

$$\frac{S_1, \ldots, S_n}{S_0}$$

where S_0, S_1, ..., S_n are sequents. S_1, ..., S_n are called the *premises*, S_0 — the *conclusion* of the rule. A rule with no premises is an *axiom*. A rule is *sound* if from the validity of the premises (in a state) follows the validity of the conclusion (in this state). Usually we are only interested in sound rules. ◁

To prove the validity of the formula ϕ, we start with the initial sequent goal $\vdash \phi$. If $\vdash \phi$ is an instance of the conclusion of a calculus rule, we can replace the current goal with the goals that are instances of the premises specified in the rule. If, by repeating the procedure, we can obtain an empty goal list then the formula ϕ holds. This process can also be represented as a tree, which will be the basis for the formal definition of a proof we give below.

1.3.2 Taclets as Means of Deduction

The KeY calculus uses *taclets* as a way of describing rule schemata. Taclets, formerly called schematic theory-specific rules (STSR), have been introduced in [Hab00]. A taclet combines the logical content of a sequent rule with pragmatic information that indicates when and for what it should be used. It may contain *schema variables* that have to be instantiated to obtain a calculus rule. We are not going to present a formal description of taclets — only to the extent required by our needs. For details consult [Hab00].

A taclet consists of an application part, one or several goal description parts and auxiliary control information (which we will disregard here).

Taclet: application part; focus The *application part* describes when a rule is applicable. It has the form

$$\text{find } (\mathit{findexp}) \text{ if } (\mathit{ifseq}) \text{ varcond } (\mathit{varexp})$$

- The "find" clause is the most important part of the taclet as it designates the expression that the taclet "tackles". For the taclet to be applicable, *findexp* must match a concrete focus in a goal of the proof under construction. The focus can be a term, a formula, or a program part depending on the type of the taclet. We present a classification of taclets depending on their focus (and corresponding notations) in Section 1.3.4 on page 14.

- The "if" clause poses conditions on the goal sequent constraining when a taclet is applicable.
- The "varcond" clause describes constraints on variables in instantiations.

Every clause is in principle optional — though, of course, not all clause combinations are meaningful. If the "find" clause is absent from the taclet, the taclet will not focus on any formula or term; instead, it is available by selecting the whole sequent. Typical examples of such "no-find" taclets are CUT and INT_INDUTION. Section A on page 70 offers an overview of important taclets.

Taclet: goal description part The *goal description* part describes how new goal sequents are constructed when the taclet is applied. It consists of several single goal descriptions of the form

$$\text{replacewith } (rwexp) \text{ add } (addseq) \text{ addrules } (tacletlist)$$

It describes the actions performed by the taclet, that is the effects of the rule when it is applied to a sequent. Every single goal description corresponds to one offspring sequent after rule application. The presence of more than one goal description indicates that the proof tree is split and several new goals are generated.

- The $rwexp$ in the "replacewith" clause is the expression that will replace $findexp$.
- The "add" clause describes formulae that will be added to the new sequent.
- The "addrules" clause dynamically generates a new taclet and makes it available from that goal on.

Goal descriptions can contain meta-constructs, which are functions programmed in Java. They can perform complex "computations" on schema variables, and thus make the calculus flexible and powerful.

Example 1.3.1 Consider the taclet

$$\text{find } (b \to c \Rightarrow) \text{ if } (b \Rightarrow) \text{ replacewith } (c \Rightarrow)$$

This means that an implication $b \to c$ on the left side of a sequent may be replaced by c, if the formula b also appears on the left side of that sequent. Apart from this "logical" content in the rule, the find clause indicates that the taclet will focus on the implication. ◁

1.3.3 Reasoning about Programs with Taclets

As the primary goal of the KeY calculus is to prove correctness of programs, taclets may contain, beside logical schema variables (representing terms and formulae), also schematic programs. Thus, taclets perform the "find–replace" operations described above just as well on programs. The applicability of rules manipulating programs, though, is subject to an important limitation: such rules operate only on the *active statement* of the DL formula [Bec01].

Definition 1.3.3 (Active statement) Every DL formula of the form $\langle q \rangle \phi$ can be unambiguously decomposed as $\langle \pi\, p\, \omega \rangle \phi$, where π is the (non-active) prefix consisting of an arbitrary sequence of opening braces, labels, beginnings of try-catch blocks, and beginnings of method invocation blocks ("method frames"). The single statement p is the *active statement*, while the postfix ω denotes the "rest" of the program. ◁

Example 1.3.2 Any taclet working with the following Java block must focus on the if statement. The non-active prefix π and the "rest" ω involved are as indicated:

$$\underbrace{\texttt{l :\{ try \{}}_{\pi}\ \overbrace{\texttt{if (i==0) k=0;}}^{\text{active statement}}\ \underbrace{\texttt{j=0; \} finally \{ k=0; \}\}}}_{\omega}$$

◁

Example 1.3.3 One taclet for dealing with if statements is IF_ELSE_SPLIT[3].

$$\begin{aligned}&\text{find } (\Rightarrow \langle \{ \text{.. if(\#se) \#s0 else \#s1 ...} \} \rangle post)\\ &\text{replacewith } (\Rightarrow \langle \{ \text{.. \#s0 ...} \} \rangle post) \text{ add } (\text{\#se } = \text{TRUE } \Rightarrow);\\ &\text{replacewith } (\Rightarrow \langle \{ \text{.. \#s1 ...} \} \rangle post) \text{ add } (\text{\#se } = \text{FALSE } \Rightarrow)\end{aligned}$$

"{.." denotes the non-active prefix π; "...}" corresponds to the "rest" ω of the diamond. This taclet can only be bound to a formula with an active if statement in the diamond. When the taclet is applied, the proof tree is split, with one new child node for each goal description (the then and the else branch). The new child diamonds have no if but an additional logical condition in the antecedent.

Note that this taclet is only applicable for the small class of conditionals that are known to have no side effects. In practice this simply looking taclet seldomly appears in proofs. We elaborate on this topic in Section 3.2.1 on page 23. ◁

[3] At the moment there are 18 different taclets in the KeY calculus that deal with if statements.

Symbolic execution The pattern shown in the example above is constitutive in our reasoning about programs. This approach is known as *symbolic execution*. Symbolic execution means that our program manipulating rules mimic execution of the program on real hardware. Statements are gradually removed from the program in the diamond, while their effects are recorded by adding purely first-order formulae to the sequent and/or splitting the proof tree. Eventually the program in the diamond(s) will vanish, and the verification obligation will be reduced to a first-order logical problem.

1.3.4 Constructing Proofs with Taclets

Interactive proving Interactively, taclets are applied as follows:

1. The user highlights a part of the goal sequent — the focus of the future taclet application. This can be a formula, a term, a program part, or the whole sequent.
2. The system goes over the registered taclets and picks those taclets whose "find" clause matches the highlighted expression. This already provides a partial to complete instantiation. If, furthermore, the current sequent satisfies the conditions in the "if" clause the taclet is considered *applicable*.
3. The system presents the user with a list of all applicable taclets; the user selects one taclet from this list for execution.
4. If there was not enough information to instantiate the taclet completely, the user is requested to provide the missing instantiations.
5. Finally, the values of the meta-constructs are computed and the rule is applied, giving rise to one or more child nodes.

Automated proving with heuristics There is also a facility for automated proof construction, dubbed "heuristics". Currently, a heuristic is simply a collection of taclets suitable for a certain task. The user decides which heuristics to activate; then the rules are applied exhaustively. There are heuristics for symbolic execution, expression and formula simplification, modal tautologies — over 20 altogether, at the moment.

Definition 1.3.4 (Taclet Application, Proof Step) A *taclet application* is an aggregation of a taclet with corresponding (schema variable) instantiations. The values of the meta-constructs can be computed from instantiations. A taclet application to a given focus in a certain sequent is a *proof step*. ◁

Types of taclets; focus notation We distinguish the following types of taclets in the KeY calculus, and introduce the following notations related to their foci:

Program manipulation taclets The find clause of the taclet contains a (schematic) program part. For a given application of such a taclet, besides the focus F (the concrete program part) we introduce $F_{\langle\rangle}$, which denotes the full program in the diamond containing F.

Analytic first-order and rewrite taclets The find clause of the taclet contains a (schematic) formula or term. For a given application of such a taclet, besides the focus F we introduce F_{ϕ}, which denotes the full formula that F is part of.

Focusless taclets Taclets without a find clause technically do not have a focus ($F = \emptyset$). Interactively, such taclets are applied by selecting the whole sequent.

Definition 1.3.5 (Proof) A *proof* consists of a proof tree and the appropriate *program context*. The inner nodes of the proof tree are proof steps, i.e. sequents and associated taclet applications giving rise to the children nodes. The root node contains the original proof obligation, a single DL formula in the succedent.

A leaf of the tree is called a *goal*. Often this name is used implicitly to denote an open goal — a sequent node without children, to which no taclet has yet been applied. A node without children, associated with an application of an axiom, is a closed goal.

A proof is *closed* when all its goals are. In that case the proof obligation in the root node is valid. ◁

Execution tree Besides the proof tree we will need the notion of an execution tree. An *execution tree* for a given diamond is a tree annontated with diamonds as they originate from the original as a result of symbolic execution along the proof tree. The execution tree shows possible execution paths of a given program fragment. Symbolic execution of an `if` statement (without additional knowledge) causes the execution tree to branch as two diamonds appear. The branching of an execution tree may or may not be accompanied by the branching of the proof tree.

The taclet IF_ELSE_SPLIT_IMP causes a split in the execution tree as one diamond is replaced by two. The proof tree does not branch, since both diamonds are part of the same goal. In contrast, the taclet IF_ELSE_SPLIT has two premises and immediately produces a branch in the proof tree. The taclet IF_ELSE_UPDATE_TRUE causes neither the execution nor the proof tree to split, but only by virtue of making use of additional information in the sequent about the value of the conditional.

2 Motivation

2.1 Why Re-Use Proofs?

Software verification is sometimes represented — mostly by verification solution providers — as a simple add-on step to the software development process. After a piece of software has been finalized, it is "run" through a prover to establish formal correctness (which usually succeeds at first attempt). This scenario seems hardly realistic to us.

We are sure that the prevalent use case of the KeY tool is not a single verification run. It is far more likely that the verification would fail and the program (and/or the specification) has to be revised. Our working hypothesis is that it is more advantageous to adapt the existing (partial) proof for a small amendment, than to prove the program again from first principles. We call this practice proof re-use.

An important pay-off we expect from proof re-use is a saving on user interaction. Human time and expertise are very expensive in comparison to machine time. It is undesirable that the user efforts targeted at proof construction, rule selection and application are lost with every next iteration of the revision process. Proof re-use should protect this investment.

Another potential benefit of proof re-use is due to the fact that re-use provides feedback about the points, at which a program amendment affects the correctness proof. This feedback can be useful to guide the next revision iteration. Making formal development a "round-trip" process is an important goal of the KeY project. Proof re-use can contribute to this goal.

2.2 Goals of This Work

The goal of this thesis is to develop a proof re-use facility for the KeY system. After changing a program the tool should use unaffected proof parts to construct a new proof with a minimum of user interaction.

Functional requirements Similar to [Ste92] we can express a number of functional requirements, which a re-use facility should fulfill:

- Re-use must yield correct proofs. This requirement has absolute priority for a verification solution.
- The re-use facility should re-use proof steps "whenever possible" and not just "cut & paste" in the case of identical sequents.
- The mechanism should stop when a situation not present in the old proof is reached.
- The re-use must integrate seamlessly with user actions. It should neither confuse the user, nor require increased interaction.
- The facility should be sufficiently versatile. A large range of program modifications — insertions, deletions, alterations, renamings, etc. (cf. Section 6 on page 57) — should be treated gracefully.

Performance evaluation The absence of objective criteria makes it difficult to evaluate the re-use performance. It is clear that with increasing differences between revisions the re-use rate *must* deteriorate. We propose estimating re-use performance along the following axes:

1. Ease of use. This includes the amount of user interaction required.
2. Versatility. The range of program modifications that the facility can handle gracefully.
3. Power. How much of the old proof (on average) the system can re-use, as compared to a human appraisal.
4. Robustness. How well the facility behaves in different situations with respect to its own strategy and claims.

Limiting the scope The field of automated proof construction is very broad, and we thus want to state beforehand certain directions, which we do not want to pursue. The re-use facility need not:

- handle genuinely new program parts: neither by internal analogy nor in any other way. This is left to the user.

- "learn from proofs". There is a number of proof-planning, proof-learning and similar efforts (e.g. [OME]), which have been more or less successful in their own fields. We believe that in view of our special situation — given an old proof and a program correction — it is more fruitful to concentrate on a specialized solution.

2.3 Previous/Related Approaches

This section gives an overview of the "state of the art" in verification proof re-use as pertinent to our efforts. Note that the evaluation sections presuppose knowledge of our approach. Thus, you might want to return to them after making yourself familiar with our work.

2.3.1 Analogical Transfer of Verification Proofs for State-Based Specifications

Works of Erica Melis and Axel Schairer [MS98, Sch98] concentrate on re-use of subproofs in the verification of invariants of reactive systems. Reactive systems are described by state-based specifications, where states are explicitly represented by state variables. Such specifications and their properties are expressed in first-order logic (PL1). As means of formal verification a tactical theorem prover with sequent calculus is employed.

Due to symmetries/redundancies in the state space, proof of safety properties of such systems often gives rise to many similar proof obligations. This approach, which sees itself in the case-based reasoning (CBR) tradition, tries to exploit internal analogy between subproblems. The approach consists of four steps:

1. Analysis. The analysis step computes for a given subproof (entity of re-use) a *generalization*, which captures the "essence" of the proof steps performed.
2. Retrieval. Given a new subproblem, the retrieval mechanism would come up — based upon the generalizations of the analysis step — with the appropriate re-use template. Retrieval can be performed manually or by an automated search.
3. Derivational re-play. After establishing the template in the previous step, the tactics associated with it are, essentially, re-played.
4. Completion. Goals not covered by re-use are solved from first principles.

The re-use facility is implemented in the VSE system [VSE].

Evaluation

First, we have to acknowledge a number of striking differences to our approach:

- Target logic: Dynamic logic is a manifest superset of first-order logic. Additional complexity of embedded programs of a very feature-rich, object-oriented language have to be taken into account.

- Internal re-use (a given solution for a subproblem is used to solve a similar subproblem) vs. external re-use (parts of the problem solution are used to solve a modified problem).

- Our efforts require a (much more) complicated superstructure for identifying re-usable units.

Surprisingly, there are also number of parallels:

- Use of similarity to identify the correct re-use template. In our approach similarity measures play the key role in deciding on re-use alternatives. The LCS program similarity measure (Section 4.4.3 on page 37), for instance, also works with intermediate representations, abstractions, or "essence" rather than the immediate objects.

- The general approach described above. Our facility uses almost the same four steps, though we tightly iterate the procedure — not once for every subproof, but once for every proof *step*.

2.3.2 KIV: Karlsruher Interactive Verifier

The KIV system (Karlsruhe Interactive Verifier) is a system for program development and verification. Programs are written in a simple, Pascal-like programming language. The deduction component of KIV is a tactical theorem prover working with a sequent calculus for dynamic logic.

The KIV re-use facility [Ste92,RS93] works by computing structural diffs, i.e. "optimal representations" of the new program in terms of the old one. This information is used to identify unaffected proof fragments.

Evaluation

The KIV re-use facility takes the structural approach by working with whole fragments of the proof tree. This may be feasible, since due to the simplicity of the artificial target language (no object orientation, side effects not a major issue) there is a close correspondence between the features of the program and the shapes of the proof.

The authors of the KIV re-use facility take every measure to abstain from formula and sequent comparison. In [Ste92] we read:[1]

> This poses an important question, as to what extent comparisons between sequents in the old and the new proof are necessary (or desirable). Since the sequents would not, in general, be identical, we would have to come up with similarity measures, which is definitely not an easy task.

KIV still requires elaborate heuristics for efficient fragment selection and arrangement. In our case of a real-life, very feature-rich target language, and, thus, more complicated proof structures, this would be quite difficult.

In KIV an `if` statement (most of the time) translates into a simple branch in the proof tree; in KeY we obtain (most of the time) an "arbitrarily" complex proof part dealing with the condition first, and only then the actual branching. Besides, this branching is often induced not by a rule normally associated with an `if` statement, but by a comparative expression evaluation rule. Section A on page 70 has a roundup on the pertinent taclets.

Thus, our re-use efforts take a *dual* approach by concentrating on formulae and sequents, and to a much lesser degree on structural features of proofs, like identifying "positive" and "negative" branches, etc. We do introduce a structural component in our considerations — the connectivity refinement — but in a less rigid and knowledge-intensive way.

2.3.3 Proof Re-Use with Isabelle

Another theorem prover we want to review as pertinent to our efforts is Isabelle [Pau94] (e.g. with its Java verification calculus Bali [Ohe01]). As such, Isabelle does not have a proof re-use facility; nonetheless users do constantly re-use proof parts between revisions, which is a byproduct of Isabelle's textual interaction style. A user usually maintains a proof record (a command and rule application history) in a text editor of choice. With this, proof re-use boils down to "re-running" the

[1] Translation is ours.

history on the modified proof obligation. As alway with re-use, the question arising is how well this scheme is adequate for coping with change.

Central to this issue is the way of targeting rule applications. With Isabelle, the user usually does not specify the focus of rule application explicitly. Instead, certain conventions allow to infer the target. Rules and more complex tactics are usually applied to the first open goal (others, then, to all open goals) and the first possible occurrence. This device, while clearly more resistant to change than working with absolute term positions, would still cause re-use to go awry if e.g. the number/ ordering of goals changes in an unfavorable way. Our approach does mimic this mode of operation in trying to infer the correct position for a rule application (even for calculi that were not originally designed to do so), but employs target similarity assessment instead of relying on unchanged ordering.

3 Re-Use Approaches

There are clearly a number of possible approaches to proof re-use in software verification. We propose a classification range, with the *feature-oriented* approaches on one side and the *featureless* approaches on the other side. The feature awareness means in this case the presence of engineered knowledge about individual constructs of the target programming language and the forms they induce in proofs about them. We will discuss the approaches in more detail below. Of course, hardly any existing or possible approach could be identified solely with one of these extremes; most applications fall somewhere in the range between.

3.1 The Feature-Oriented Approach

The feature-oriented approach distinguishes itself by its knowledge of how individual program features (programming language constructs) translate into proofs about them. Concerning proof re-use this means that such an approach would not only detect but analyze the changes in the program, precisely identify unaffected proof parts, calculate their arrangement in the new proof and maybe even perform some adaptations on them.

To demonstrate this let's assume a programmer amends the program by "adding a case" — inserting an `if` statement and "sidetracking" a part of the control flow. A feature-oriented re-use facility would

- analyze this change
- identify the proof parts and the way they will have to be "re-parented" according to how the statements are distributed along the `then`/`else` branches

then maybe even:

- split the proof tree at the right point
- add the right provisos to the new branches and complete the proof outside the usual proof construction mode.

This proceeding requires a material upfront investment in structural knowledge for every feature of the target language and the calculus. Otherwise the re-use facility comes out not powerful enough for the general use.

3.2 Feature Overflow

Real-world language The above approach is feasible as long as the programming language we deal with is relatively simple, and there are not too many features to control. This cannot be said of Java. Being a modern, real-world language, Java is very feature-rich. The three basic programming concepts — an assignment, a branching statement and a loop — can appear in many different incarnations: switch statements, exception clauses and at least three different types of loops, just to name a few.

Object-orientation Besides, since Java is an object oriented language, computing in it is accomplished by passing messages between objects, which often modifies their internal state. This is one more reason why a side-effect-free expression — which program verification has for a long time concentrated upon, as it makes reasoning about programs easy — is in Java rather an exception than the rule. Since most statements contain such an expression, working with them in a proof requires first decomposing the expression into its simple parts, which might involve many proof steps and produce a complicated proof structure. An example about how a "simple" method invocation causes extensive proof branching is given later on.

Varying semantics Not only does the KeY calculus consist of many rules, there are also several "calculus modes" for certain aspects of the target language. For instance, the system can assume three different semantics for integer arithmetics [Sch02]:

- Arithmetic semantics ignoring overflow (aka mathematical semantics). The primitive integer datatypes are considered to have an infinite range, and overflow is completely disregarded. Programs proven correct with this semantics may not perform up to the specification when executed on the JVM.
- Native Java semantics. The semantics of primitive integer datatypes and the treatment of overflow corresponds to the semantics defined in the JLS [GJS96]. Of course, many intuitive properties that hold over $\mathbb{Z}$ cannot be shown in this semantics.

- Arithmetic semantics prohibiting overflow. This semantics is designed to combat the disadvantages of the two predecessors. It works with extended arithmetical data types of infinite range, while introducing overflow safeguard provisos for every arithmetical operation.

Accommodating for different treatment of language constructs in different semantics means a multiple effort.

Example 3.2.1 Figure 3.1 on the next page illustrates our point on complex proof shapes. The piece of code shown in part (a) consists of one `if` statement. In case of a simple programming language this would mean a simple case distinction in the proof involved. In our case the proof dealing with this "program" splits 11-fold!

The case explosion is a direct consequence of the object-orientation of our target language. But first, more details on the program: `o` and `u` are variables of object type `BaseClass`. `BaseClass` ("B" in the diagram) has a subclass `SubClass` ("S") that overrides the method `size()` declared in `BaseClass`.

The active statement of the program in Figure 3.1 on the following page is an `if` statement, but it cannot be tackled directly, since the condition may have side effects. The KeY calculus "evaluates" such expressions by decomposing them into simple ones and introducing auxiliary variables. Altogether the process resembles the left to right evaluation of an expression by the JVM. After a few rule applications the program in our diamond is transformed as shown in part (b).

Now the first object variable must be dereferenced: if `o` is `null` (case 1, "$\emptyset$" in the diagram) the execution terminates abruptly with a `NullPointerException`. If `o` is not `null`, an appropriate method implementation must be bound to the call. The object referenced by `o` can be, as we have declared, either an instance of `BaseClass` or of `SubClass`. The real JVM would now perform dynamic binding; we have to simulate it by a case distinction; this gives cases 2–11. Note that the cases 2 and 3 are distinct, since the execution of `o.size()` can have (different) side effects, which are not rolled back by the abrupt termination of `u.size()`.

Other remarkable facts about this example are:

- With *two* subclasses of `BaseClass` providing their own implementations, we would obtain a total of 22 cases.

- Choosing the integer semantics that prohibits overflow, would give an additional branch in every of the cases 4–11, asserting that no arithmetical overflow occurs during addition.

```
if  (o.size()+u.size()<17)
    result=true;
else
    result=false;
```

(a) A simple statement...

```
_var2=o.size();
_var3=u.size();
_var1=_var2+_var3;
_var0=_var1<17;
if  (_var0)
    result=true;
else
    result=false;
```

(b) ...expanded...

	Goals		
Nr.	o	u	<17?
1	∅	*	*
2	S	∅	*
3	B	∅	*
4	S	S	Y
5	S	S	N
6	S	B	Y
7	S	B	N
8	B	S	Y
9	B	S	N
10	B	B	Y
11	B	B	N

(c) ...branches 11-fold

Figure 3.1: Object-oriented target language causes complex proof shapes

- The branching is not caused by the taclet directly associated with an if statement (!) — it is performed by the taclets METHOD_CALL and LESS_THAN_COMPARISON[1] ◁

3.3 The Featureless Approach

The featureless approach, on the other hand, is for the most part oblivious to the way different programming language features are reflected in the proof. In view of the intrinsic complexity of Java programs — which can be estimated by the length of the Java Language Specification alone — and the way it must be reflected in our calculus, the structural proof analysis seems very laborious and knowledge-intensive.

What seems more feasible is to concentrate on the few basic properties of the underlying calculus, and try to accommodate for them in the re-use situation:

[1] We do not provide the complete proof record here, since the rules for treating definedness conditions in KeY have not yet been finalized.

- Similar situations warrant similar proof steps.
- The number of positions where a given rule is applicable is relatively small. Crucial for this is the fact that symbolic execution always deals with the active statement of a Java block and there is no "split" rule for programs like in Hoare Logic.
- Proofs may branch, surprisingly and abundantly. We must be ready to deal with this phenomenon.

We will present a re-use algorithm, which we call "observing". The moniker results from the fact that the system does not have a strong preconception of how re-use must proceed in a particular case, but observes the proof development (mostly the sequents) instead and suggests steps from the old proof for re-use, when it deems so appropriate.

As a largely featureless approach — though, again, we are pragmatics, not extremists — the algorithm stands out through abdication. It includes no claim to exhaustive knowledge of correspondence between programs and proofs. It includes no claim to knowledge what parts of a proof are unaffected by a program change.

What we give up in "hard knowledge" though, we make up in resilience. The algorithm tries hard to overcome small obstacles autonomously. Furthermore, the algorithm will require very little to none modification when calculus support for new constructs and new features is added to the KeY tool (which is happening constantly). The same is valid when the form of existing rules changes, as improvements are made.

3.4 Feasability of Re-Use

In the meantime, we want to make a couple of remarks why we consider our re-use approach feasible at all. Three reasons speak in favor of the possibility of re-use. First, the practice has shown that small alterations in the program (usually) lead only to small alterations in the proof. Of course, a minor change *can* have a very disruptive effect, but such cases are not that common.

The second reason is: limited rule applicability. Given a certain rule and a proof there is a relatively small number of potential application foci for this rule. This makes identifying the right focus for re-use easier. The reason for this condition lies in the following benign property of symbolic execution as defined by the KeY calculus. Program manipulating rules can only be applied to the active statement of a given diamond (see Section 1.3 on page 9). Furthermore, in DL versions for simple artificial programming languages, where no prefixes are needed, any formula of

the form $\langle p\ q\rangle\phi$ can be replaced by $\langle p\rangle\langle q\rangle\phi$. In our calculus, splitting of $\langle\pi\ p\ q\ \omega\rangle\phi$ is not possible.

The third reason has the same justification as the second. It states that global proof structure follows program structure. This can be expressed by the following monotonicity property:

> For any proof reasoning about a program containing the statements α and β, with α appearing before β, the following holds true: for every proof step dealing with β there is a proof step dealing with α above it.

Important for the function of our frontend is that we can cover all proof steps related to the program $\alpha;\beta;\gamma$ by the subtree of the proof, whose root deals with α.

Proofs about statements According to the monotonicity property stated above, we define a natural mapping of program parts to proof parts, which we will call "proofs about a statement".

Definition 3.4.1 (Proof about α: P_α) If α is a (possibly compound) statement then P_α denotes the subforest of the proof where the symbolic execution of α begins. That is, on the path(s) from the root of the proof tree to the root(s) of P_α, there is no proof step bound to (a part of) α. $\lhd$

In most cases P_α will be a single tree, unless the proof is cleaved above P_α by a non-program-related proof step (e.g. CUT).

While the beginning of a proof about α is easy to discern, the further boundaries and, especially, the end are far less so. There are several difficulties that hamper an exact designation at this point:

1. What about the purely first-order formulae that are created during the symbolic execution of α? Do the proof steps working with them belong to the potential proof for α? Unfortunately this matter is complicated even further by the fact that such steps do not only occur after the program has been completely reduced by symbolic execution; the user or the heuristics may interleave purely first-order and program-related proof steps.
2. Potential proofs for different statements can overlap. The user (or the heuristics) may tackle one diamond first, apply some rules to another one, and then switch back to work on the first-mentioned program again.
3. The KIV system would try to compute "proof fragments" responsible for program parts precisely [Ste92]. This may be reasonable given the target language with far less features then Java — KIV, for example, did not have to deal with definedness conditions or integer data type constraints, which

generate additional non-trivial proof complexity. Still, re-use of a computed proof fragment would, according to [Ste92], often have to be "interrupted" for a different effort.

For these reasons we want to leave the exact boundaries of P_α open.

Due to the proof-follows-program property stated above our facility (or more exactly, its Frontend, see Section 5 on page 51) can easily identify several P_α as potential subproofs that are re-usable. It is though not necessary that these subproofs are re-used as a whole — failures to do so should be compensated by an excess of such markers.

4 The Observing Re-Use Technique

4.1 Overall System Conception

The general composition of the re-use facility is presented in Figure 4.1. The facility consists of a *frontend* and a *backend*. The frontend is the part of the system working with the source code; the backend is the part of the system working with the proof(s). Our point of departure for re-use is the following use case. Our frontend has access to two versions of the program — in the future referred to colloquially as the "old" one and the "new" (the amended) one. These may be two full sets of the source files or just one of them accompanied by a list of differences in some form. Additionally, further information about the change between the two may be available: e.g. a log of programmer's actions. The job of the frontend is to analyze the changes between these two versions and to provide results to the backend. It is covered in detail in the Section "Frontend" on page 51.

The backend has access to the "old" proof, which may or may not be complete. The job of the backend is to assist the user in constructing a "new" proof for the "new" program with as little interaction is possible.

We will perform re-use by re-playing certain proof steps from the old proof in the context of the new one. In particular this means:

- only the proof steps present in the old proof are re-used. The re-use facility does not alter or perform proof steps on its own.
- re-use is performed in the normal proof construction mode. The correctness of the new proof is thus guaranteed.

Figure 4.1: The re-use facility: components and responsabilities

Frontend: Source	Backend: Proof
old source	old proof
new source	?

4.2 Basic Re-Use Principle

On the backend side we start re-use with the following premises. We have at least two proofs loaded in the prover: the "old" proof serving as the template — which can be finished or not — and an incomplete "new" proof. At the beginning, the new proof would be just a stub, containing only the initial sequent with the new proof obligation. The prover has been extended to handle several distinct proofs in parallel; the user can switch between proofs with a simple click.

The essence of the proof re-use we are attempting is to try and repeat the rule applications of the old proof in the context of the new one. To illustrate our re-use framework, let's assume for the moment that the old and the new proof obligations are identical, i.e. re-use consists of fabricating an exact copy of the old proof. Our approach is to traverse the old proof tree top-down and copy the proof steps to the new proof. We will keep record of what parts of the old proof tree we have already treated or yet have to treat by setting "markers". We start by marking the root node of the old proof. If there is a marked node, we re-use its proof step, clear the node marker, then mark all of its proof tree children. While iterating this procedure, we can see a "wavefront" of markers expanding through the old proof tree. In the case of 1:1 re-use that we have assumed, this wavefront of markers mirrors exactly the wavefront of goals in the new proof. We also note that we can re-play the whole proof with just the initial marker.

Now we drop the assumption we made above, and consider a scenario where the old and the new proof obligations differ, while sticking to the general approach. Several questions immediately come up:

- What if a rule from the old proof is not applicable in the new one? Since our proof obligations are not identical this could easily happen. In this case the particular marked proof step cannot be re-used, and the wavefront expansion stops at this point. The fact that the marker is "blocked" indicates that we have reached a point where some new/different proof steps are needed, either in addition to the old proof or in replacement. These steps have to be performed by the user or the existing automated proof search facility. Afterwards, it is possible that
  - the marker becomes "un-blocked" (applicable) and we can resume re-use right where we left
  - we have left the particular marker behind; there is no perspective of it ever becoming un-blocked. This means that a certain part of the old proof is not re-usable in the new context. There might be a part of the old proof further on though, which can be re-used after "bridging the

gap". We make provisions for this case by starting re-use with a surplus of markers in the old proof. At the beginning, the frontend introduces markers at all "interesting" proof points, which might be suitable to "restart" re-use if the difference in the problems leads the proofs apart. In the rare case these markers should not suffice, additional markers can be introduced anytime by the user or the recovery (search) procedure.

- How do we know which marker corresponds to which goal? Well, in general, we don't. First, we test what marked proof steps are applicable at all. Then we examine formulae and sequents to determine an appropriate mapping. Similar situations warrant similar proof steps — this is the credo of feature-less re-use. If there is too much dis-similarity we even disregard a re-use possibility, since it's not "sound".

- What about taclet application positions? Our calculus requires to specify a position in the sequent when applying a rule. In the new proof these positions may have shifted. Indeed, we must recur to a search within the goal sequents. If more than one position is susceptible to a taclet application we perform a comparative analysis w.r.t. the old proof.

Terminology

We now state some terminology before we give a formal description of the re-use algorithm.

Candidate proof step A *candidate proof step* is actually just a distinguished proof step in the "old" proof — consisting of a proof node with a sequent and an attached rule application. The attribute "candidate" indicates that the proof step is registered in the candidate list, which the system maintains. There are no further requirements posed though, in particular it is not necessary that the taclet associated with a candidate proof step is applicable in the target goal, or any goal for that matter. Synonymously with calling a proof step candidate we may state that this proof step is *marked* or talk about the *marker*. This circumstance is reflected in the prover GUI by a graphical mark in the proof tree.

The combinations of applicable proof steps and the open goals in the new proof give rise to possible re-use units.

(Possible) re-use unit An entity of re-use is a (possible) re-use unit. A possible re-use unit establishes a relation between a candidate proof step of the old proof and its potential target in the new one. In particular, it consists of:

- A "source" proof step (node) in the old proof tree. This includes a taclet T, a focus F^{source} and other instantiations. During the lifecycle of a re-use unit this node is distinguished as a candidate proof step.
- A focus F^{target} in the new proof, where T is applicable.
- Eventually a numerical "quality score" Ω, which is computed from the above (see Section 4.4 on page 34). The quality score measures how much "'sense"' this possible re-use unit makes.

Note that the applicability of the associated taclet is the only requirement for the creation of a re-use unit. It is not required that $F^{\text{source}} = F^{\text{target}}$, or similar. These facts are accounted for in the quality score. The quality scores decide if and when a possible re-use unit is activated for re-use.

4.3 The Main Backend Algorithm

The Algorithm in Pseudocode

The main re-use algorithm (and its integration into the proof construction loop) is shown in Table 4.1 on the next page. It maintains a list C of proof steps in the old (template) proof, which are marked as *re-use candidates*. While re-use progresses and the new proof grows, C always contains those rule applications in the old proof that are currently considered available for re-use. Initially this list is filled with candidates provided by the frontend (see Section 5 on page 51). The user too can mark nodes in the old proof as additional candidates anytime.

Remarks to the Algorithm

(1) The rationale for making initial candidates persistent in C is presented in Section 6.1.3 on page 61.

(2) The applicability check is performed by the standard taclet matching algorithm (for details of when a taclet is applicable see Section 1.3 on page 9). Usually this mechanism is used to determine taclets that are applicable, whenever the user highlights a part of the sequent. Here though, the mechanism is used in reverse: we iterate over all sequent positions and inspect whether the taclet associated with c

input $oldProof$, $oldProgram$, $newProgram$, $specification$;
$newProof$:= initialProofGoal($newProgram$, $specification$);
C_0 := initialCandidateList($oldProof$, $\Delta(oldProgram, newProgram)$);
$C := C_0$;
while $newProof$ has open goals **do**
  $\langle candidate, newFocus\rangle$:= chooseReuse(C, $oldProof$, $newProof$);
  if $\langle candidate, newFocus\rangle \neq \bot$ **then**
    $newProof$:= result of applying rule($candidate$)
      at $newFocus$ in $newProof$;
    $C := C \setminus \{candidate\}$;
    if ($candidate \in C_0$) and ($candidate \notin C$) **then** (1)
      $C := C \cup \{candidate\}$;
    fi
    $C := C \cup \{c \mid c$ is a child of $candidate$ in $oldProof\}$;
  else
    $newProof$:= applyRuleWithoutReuse($newProof$);
  fi;
od;
output $newProof$;

Table 4.1: Main re-use and proof construction algorithm.

matches. This is, by the way, the rule selection principle used by many other theorem provers, like Isabelle.

The nested loops are presented in this form for clarity. In reality incremetal computation takes place, since only one candidate at a time is involved during re-use.

Re-Use of Taclet Instantiations

As stated in the description of a possible re-use unit, it contains taclet schema variable instantiations originating from the old proof steps. These are partially overridden by binding the taclet to a new locality. The remaining "non-trivial" instantiations that are not forced by the binding (if any) are the ones originally provided by the user. Taclets requiring user-provided instantiations are:

1. CUT, where the right case distinction has to be used
2. INT_INDUCTION and related, where the right induction hypothesis has to be used

```
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 (2) 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 4.2: Choosing the best possible reuse unit.

3. quantifier rules, where the right instantiation has to be found

4. rules that require an introduction of a new variable, e.g. IF_EVAL (this is relatively harmless, since only the name of the variable is at stake here)

Since it would be a very hard task to adapt these instantiations to the new proof automatically, we simply retain them (with an additional option to present the user with the current value and ask for confirmation.)

Fruitful in this context may be application of techniques to make instantiations less "volatile". Induction hypotheses for proving loops, for instance, contain the DL formula with the loop in question. The smallest change in the loop makes thus the hypothesis no longer valid. Employing some kind of indirect reference mechanism instead of explicit formula quotation would make this class of instantiations more change-resistant.

Recovery of Inaccessible Parts

As a part of our resilience strategy, we have built-in a scheme that allows to recover inaccessible parts of the old proof for re-use. This can be helpful when, for instance, not enough markers have been generated beforehand, or if unforeseen obstacles obstruct their progression. The procedure automatically steps into action whenever none of the current re-use units can be activated (re-use is halted). The recovery scheme starts to search the old proof tree in the top-down fashion for applicable proof steps, starting from the current markers. The depth of the search is controllable by the user. The default value is 5 steps and/or, optionally, up to the next symbolic execution step. If a good re-use unit is found then the corresponding marker is really created, and re-use resumes. The user, of course, has the option to request recovery at any time or, on the contrary, turn it completely off.

4.4 Re-Use Unit Quality Assessment

Re-use unit quality assessment, based on similarity scoring is a key part of our re-use facility, since it performs one of the most crucial and difficult parts in our effort: distinction between proof parts that are appropriate for re-use and parts that only seem to be so. In other words, similarity scoring must prevent mis-application of proof steps from the clearly non-related parts of the old proof ("confusion", see Section 4.4.1 on the next page).

4.4.1 Quality Assessment Framework

Relative Quality Assessment

In the function **chooseReuse** of the main re-use algorithm (Table 4.2 on the preceding page), we are usually presented with a choice between a number of possible re-use units of varying quality. At the same time, our algorithm is designed to re-use only one proof step at a time. This means we must perform a relative quality assessment to choose at most one re-use unit from the available selection. We achieve this goal by assigning a certain numerical quality score to *every* possible re-use unit on the list, and subsequently selecting the one with the highest value (a random one if there are several). The computation of these quality scores is described below.

It may seem at this point that our algorithm is greedy: every iteration neither possesses memory of the previous decisions nor performs any lookahead. This is only superficially the case, as the global optimizations are *built-in* into the quality valuation. Optimization w.r.t. past steps is the essence of connectivity analysis (see

Section 4.4.5 on page 46), while the usage of difference detection algorithms for estimating program similarity takes also the potential *future* symbolic execution into account.

Absolute Quality Assessment

While performing re-use, the danger is not only to do to little, but also to do "too much". Sometimes, even though there are re-use units available, it is better to re-use none of them, failing the algorithm. This is not so odd as it seems, since the existence of a possible re-use unit signifies little more than a possible application of a single rule.

If a candidate proof step contains e.g. an assignment rule, then, according to our design, a possible re-use unit is created for every diamond starting with an assignment in every goal available. It's not hard to imagine that this can give us quite a few re-use units. By choosing a "wrong" one, we would be transferring proof steps from clearly non-relevant part(s) of the old proof to the new proof.

We'll call this mis-application of proof steps "confusion". Though confusion — at least — does not undermine the correctness of the proof under construction (since only correct rule applications are performed), it is poisonous for re-use. A situation where re-use performance could be significantly lowered by confusion is shown in Section 4.4.3 on page 42.

To safeguard against confusion we compare the quality scores of possible re-use units to a threshold value ε. In case when a re-use unit is scored under ε it is not considered for re-use. If this is the case for all possible re-use units, the algorithm fails, and control is returned to the user. In most cases this is an indication that we have reached a program part that is either different or not present in the old proof. The optimal value of ε has to be determined empirically in a series of test runs for every given similarity function. It can also be modified by the user at run-time.

No confusion postulate In our further discussions of the re-use algorithm we will assume that the possibility of confusion has been eliminated by the above stated measures. We will refer to this assumption as the "no confusion postulate".

Computing the Quality Score of a Re-Use Unit

The above-mentioned quality score is an integer calculated for every possible re-use unit by the re-use facility. The calculation is based on a variety of criteria, which

we will discuss in the following. One invariant, however, is given: higher number means higher re-use preference in our assessment.

First, we distinguish by the type of the proof step according to the classification in Section 1.3.4 on page 14. A proof step can either manipulate the program in a diamond, a purely first-order formula, or have no focus at all. In these three cases:

- A symbolic execution step; the taclet manipulates the program in one of the diamonds. The quality score is determined by similarity of programs in the diamonds:
$$\Omega = \delta(F^{\text{source}}_{\langle\rangle},\ F^{\text{target}}_{\langle\rangle})$$
Three different program similarity scores δ are presented in the next sections. The logical parts of the formulae are not considered, since they are rarely tackled before the program in the diamond has been completely eliminated by symbolic execution.
- A pure first-order step; the taclet manipulates a formula or a term without modifying modalities. A logical similarity analysis is performed (see Section 4.4.6 on page 50). Programs in formulae contribute to similariy valuation.
- A no-find step. The corresponding taclet contains no "find" clause, and thus the proof step has no focus, which could be compared (e.g. INT_INDUCTION). We obtain one possible re-use unit for every open goal in the new proof, and must solely pick the most "appropriate" one. We do this on the bases of proof connectivity.

Subsequently, further refinements can step into action (e.g. the connectivity refinement, Section 4.4.5 on page 46) and distribute additional penalties or awards. The next sections deal with details of individual scores.

4.4.2 1st: "Home-Brewed" Similarity Score

The first similarity score we tested was "home-brewed" based on simple left-to-right matching. For two programs $\alpha = \alpha_1\,\alpha_2 \cdots \alpha_n$ and $\beta = \beta_1\,\beta_2 \cdots \beta_m$ we computed similarity as

$$\delta(\alpha, \beta) = -\sum_{i=1}^{\min(n,m)} P(\alpha_i, \beta_i)$$

with the penalty $P(\alpha_i, \beta_i)$ being zero if $\alpha_i = \beta_i$, and some positive value otherwise.

A major drawback of this scoring scheme can be demonstrated by the following observation. It is clear that if β differs from α by a single inserted statement β_k then there is hardly a match to be expected at positions after k, and the score drops sharply. To counteract this undesirable effect the penalties $P(\alpha_i, \beta_i)$ had to be weighted with a facotr steeply declining (with i).

Unfortunately, this and other measures (e.g. disregarding blocks), as well as further refinements of the penalty calculation (gradations for type/expression mismatches) could not overcome the basic deficiency. The resolution of this similarity score remained unsatisfactory, failing to provide the desired discrimination even in simple cases. Another approach was necessary.

4.4.3 2nd: LCS Difference Detection Score

Much better performance could be achieved by switching from the score described above to a difference detection measure. To evaluate the similarity of two programs (formulae) these are first transformed into sequences of their statement signatures (see below).

The LCS algorithm

Then the difference algorithm by Eugene Myers [Mye86] is run upon these inputs. This classical dynamic programming algorithm (instance of the single-source shortest path treatment) determines the longest common subsequence (LCS) of the inputs while — dual to this — also producing the minimal edit script, which we use for our goals. According to Myers, the algorithm takes two sequences of symbols $A = a_1\, a_2 \cdots a_N$ and $B = b_1\, b_2 \cdots b_M$ as its input. The symbols can be derived from an arbitrary alphabet; it is merely necessary to discern, whether two symbols are equal or not. An edit script for A and B is a set of insertion and deletion commands that transform A into B. 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. Deviating from [Mye86], we interpret script commands as referring to symbol positions within A *after* the preceding commands have been performed. The length of the script is the number of symbols inserted or deleted.

Statement signatures

To be able to compare programs in DL formulae we first linearize them into a sequence of statements. After that the statements are abstracted into statement sig-

natures to achieve an optimal comparison flexibility. A statement signature is a character string, constructed according to the following rules: the signature of

- a statement is the name of the statement (If, LocalVarDecl, etc. — basically class names of the internal program representation)
- a statement, which is also an expression, is the name of the statement (e.g. Assignment) followed by the static type of the expression.
- an assignment with a boolean literal as the right operand is the signature as described above with the value of this literal appended (see Section 4.4.3 on page 42).
- a method body statement is the signature as described above with the method name and the name of the class containing the referenced implementation appended.

Note that the signatures (mostly) abstract from names, expressions, literal values, etc. This allows us to deal gracefully with such changes as renaming (described in Section 6.2.1 on page 62) and changes in literal values — provided, of course, they do not change the proof structure. The pure structure of the program has proven to be sufficient to avoid confusion in most cases. Pragmatical improvements have been introduced where needed. A possible simple improvement — that has not yet been necessary, though — would be to add certain type information for core expressions in statements (type of operands in comparative `if` conditions, type of the return value in `return`, etc.) to the statement signature.

From edit script to similarity score

To come up with a program similarity score for two programs α and β, we have computed a minimal edit script between their signature representations A and B. Now we must condense this edit script into a single numerical value.

Let $E(A, B) = e_1\, e_2 \cdots e_k$ be the minimal edit script for the sequences A and B. We set

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

while the penalty

```
int x; int result; result = x/x;                          (α)
int x; int result; if (x==0) result=1; else result=x/x;   (β)
```

(a) Programs…

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

(b) …translated into sequences of statement signatures

$2\,I$ `If` (“insert 1 at 2”)
$4\,I$ `Assignment(int)` (“insert 1 at 4”)

(c) The resulting edit script

Figure 4.2: Assessing program similarity with LCS diff score

$$P(e_i) = \begin{cases} \displaystyle\sum_{k=1}^{t} \frac{75}{x+k}, \text{ for } e_i = x\,I\,b_1\,b_2 \cdots b_t \\ \\ \displaystyle\frac{100}{x+1}, \text{ for } e_i = x\,D \end{cases}$$

Properties of the LCS score

1. Higher values mean higher similarity. The peak is achieved between identical programs with the value zero: $\delta(A, A) = 0$
2. The function is not symmetric, $\delta(A, B) \neq \delta(B, A)$. Statement insertions are penalized less than deletions, since additional statements in the target program can potentially be removed by de novo symbolic execution. A missing statement in the target program indicates that the old sequence of symbolic execution would have to be halted and — hopefully — continued with another marker.
3. Differences are penalized less the farther they are from the active statement (i.e. beginning of the formula), where the symbolic execution actually takes place.

Example 4.4.1 The result of translating two programs into sequences of statement signatures is depicted in Figure 4.2, subfigures (a) and (b). The underlined parts

correspond to the insertions detected by the LCS algorithm, which produces for these two sequences the edit script in the subfigure (c). The score of this edit script $\delta(A, B) = -(75/3 + 75/5) = -40$, which signifies a medium to high similarity and a "green light" to re-use the local variable declaration rule application from the old proof in the new one. ◁

Cut-Off Threshold

The threshold value ε has been empirically chosen as -72. While it is hardly possible to give a complete justification of this value, it is possible to provide some guidelines. Note that this value is tightly linked to the penalties assigned to the deletion and insertion operations.

Example 4.4.2 If two diamonds start with an assignment of different types, it is important not to confuse these assignments. The rest of the diamond may be similar enough (this is the case, for instance, when working with exception cascades), and we have to rule out this re-use possibility only based on the mis-match of the first signature.

As re-use signatures we obtain:

```
A=Assignment(NullPointerException), [...]
B=Assignment(boolean), [...]
```

This results in the edit script

0 *D*	("delete 1 at 0")
0 *I* `Assignment(boolean)`	("insert 1 at 0")

This script has a score of -175, so the threshold must lie above this mark. ◁

Example 4.4.3 Another common situation arises from the insertion of an `if` statement, which is tackled by the taclet IF_EVAL (Section 1.3.2 on page 10). If the following common code section starts with an assignment then there is confusion bound to happen. Thus our threshold must prevent an insertion of two statements at the beginning of the block. A pair of typical re-use signatures would be

```
A=[...]
B=Assignment(boolean), If, [...]
```

with the resulting edit script

$0\ I$ `Assignment(boolean), If` ("insert 2 at 0")

This script has a score of -112 — another lower bound for ε. ◁

Note that our selected value goes even further and disallows any insertion at position zero.

Evaluation

This similarity measure works well in practice but still has several issues:

- It does not discriminate by means of names. As you can see, names (of variables, fields, etc.) do not appear in statement signatures and thus play no role in establishing similarity. This means we are indeed disregarding some helpful information, though in most cases the pure structure of the program and the type information is sufficient for our purposes. The additional benefits of this approach are twofold. First, during symbolic execution, the KeY calculus abundantly introduces auxiliary variables in order to resolve expression side-effects. At the moment, these variables are just numbered in the order of their introduction. Avoiding confusion from this naming scheme would be a major effort. Second, this way we can handle renamings (see Section 6.2.1 on page 62) within the same replay framework. A more differentiated approach w.r.t. names might be feasible in the future.
- It relies on types. Type information is embedded in statement signatures. Changing the type of a variable — e.g. from short to int — would force a deletion and an insertion of a signature at this point and thus an (undesirably) high penalty. This issue, just as the one above, is a general result of the "equal–unequal" nature of the LCS algorithm. It does not provide means for assigning "medium" penalties for certain kinds of "soft" mismatches. This deficiency can addressed by the tree correction score (Section 4.4.4 on page 43).
- It uses linear program representation. The sequences of statement signatures reflect the order of statements as they appear in the natural syntactical representation of the program. This causes maladjustment with our similarity function, which penalizes mismatches inversely proportional to their distance from the active statement region. A mismatch in the else branch of an if statement results, for instance, in a lower penalty than the same mismatch in the then branch. This deficiency can likewise be addressed by the tree correction score (Section 4.4.4 on page 43).

Refining the Signatures

In some cases, after extensive testing, we had to selectively amend our re-use signature scheme to achieve adequate performance:

Boolean assignments Consider the Figure 4.3 on the next page, which shows the first steps of a typical symbolic execution of an `if` statement. This is a non-branching (w.r.t. the proof tree) run, where the two possibilities — the condition being true or false — are tackled sequentially. The steps displayed in the figure are usually followed by an ASSIGNMENT and IF_FALSE rule on the left, "false", diamond, as well as an ASSIGNMENT followed by an IF_TRUE rule on the right, or "true", diamond.

Suppose we want to re-play this sequence of proof steps 1:1 within our re-use framework. We have a marker for the `if` statement, and we have replayed the first 3 steps right through the execution tree branching. The proof tree is still linear in this region; we assume that in the old proof the sequence of proof steps ASSIGNMENT, IF_FALSE ("false diamond") would follow from here. The marker points to the assignment `_var1=false;`. This marked candidate proof step gives rise to two re-use units, since the ASSIGNMENT rule is applicable in both diamonds of the new proof.

The problem now is: the two diamonds in Figure 4.3 on the facing page have the same statement signatures! This way, both above-mentioned possible re-use units come up with the same score, although only one of them is the correct one for re-use. Should the system give preference to the re-use unit in the "true" diamond, the ASSIGNMENT can be executed but not the IF_FALSE, and the marker is blocked.

This undesirable situation is very grave, and unfortunately quite common. On the other hand, the boolean literal space is very limited, and the impact of the particular literal value is very high, especially in routing the control flow. Thus we are better off reducing our mismatch tolerance selectively at this point by introducing the following refinement to the signature generation rules:

- In case the statement is an assignment, where the right operand is a boolean literal, the value of that literal is appended to the usual statement signature.

Method body statements Another candidates for confusion are method body statements — pseudostatements of the form `ClassName::m()`, which are short forms for the method implementation from a certain class. The taclet METHOD_BODY_EXPAND replaces these statements with the referenced code. The example in Figure 6.5 on page 65 shows usage of method body statements.

```
<{ if (x==0) result=1; else result=x/x; }> result=1
```

(a) Initial program — in the diamond

```
<{ boolean _var1; _var1=x==0; if (_var1) result=1; else result=x/x; }> result=1
```

(b) One proof step later: evaluating a complex expression

```
( !x = 0                          )   &   (  x = 0                          )
( -> <{                           )       ( -> <{                           )
(       _var1=false;              )       (       _var1=true;               )
(       if (_var1)                )       (       if (_var1)                )
(          result=1;              )       (          result=1;              )
(       else                      )       (       else                      )
(          result=x/x;            )       (          result=x/x;            )
(    }> result = 1                )       (    }> result = 1                )
```

(c) Two proof steps later: the execution tree branches, but not the proof tree

Figure 4.3: Symbolic execution of an if statement

Consider the following scenario. We are about to re-use a proof step that expands the method body statement $\langle \mathsf{BaseClass :: m()} \rangle \phi$, while in the new proof another branch has been tackled, and the potential target formula is $\langle \mathsf{SubClass :: m()} \rangle \phi$. Confusing these two formulae would mean trying to transfer a proof for the code of one method to another method, which is bound to fail. To prevent such incidents, we extend the re-use signature definition as follows:

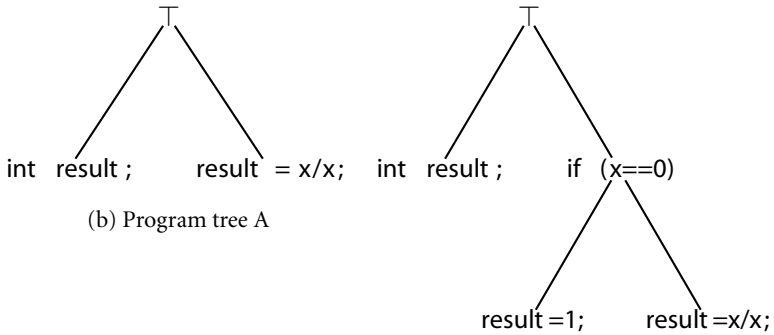
- In case when the statement is a method body statement, the signature of the method and the name of the class containing the referenced implementation is appended to the usual statement signature.

4.4.4 3rd: Tree Correction Score

The LCS score uses linear program representations for assessing similarity. The adverse effects of this circumstance have been discussed in the preceding section. These shortcomings can be alleviated by abandoning the flat sequences of tokens for program trees. In the following, we propose a third program similarity score based on tree correction, which, while being promising, has not yet been put to test in our implementation of the re-use facility.

```
int result ; result = x/x;                                  (A)
int result ; if (x==0) result =1; else result =x/x;         (B)
```

(a) Programs…



(b) Program tree A

(c) Program tree B

Insert node “ if (x==0)” insert node “ result =1;”

(d) Tree correction script

Figure 4.4: Tree correction for programs

A number of algorithms exist for performing difference detection on trees — also known as the “tree edit distance” or the “tree correction” problem. These algorithms show important variations w.r.t. the following criteria:

- exhaustive or heuristic approach
- requirements on input/data structures
- optimizations for certain cases (XML documents with sparse changes, etc.)
- node children ordering sensitivity (then and else branches of an if statement are not interchangeable, while attributes in an XML element might well be)

A tree correction algorithm We consider the “simple fast algorithm for the editing distance between trees” presented by Zhang and Shasha [ZS89] the most appropriate for our purposes. This dynamic programming algorithm defines the following correction operations on trees:

- The modify operation $(a \mapsto b)$ changes the label of a node from a to b. The node labels are assumed to be unambigous.
- The insert operation $(\Lambda \mapsto b)$ inserts a node b as an additional son of node a and makes a number of consecutive children of a become children of node b. Here Λ signifies a phantom "empty" node. Note that the notation $(\Lambda \mapsto b)$ is not unambiguous, unless the context of both trees is known (which is the case).
- The delete operation $(b \mapsto \Lambda)$ deletes the son b of node a and makes the children of node b become additional children of node a.

Each of these operations is assigned a cost factor $\gamma(a \mapsto b)$. Thus, the costs need not be uniform, but for the algorithm to work γ is required to be a distance metric, i.e.:

- $\gamma(a \mapsto b) \geq 0;\ \gamma(a \mapsto a) = 0$
- $\gamma(a \mapsto b) = \gamma(b \mapsto a)$
- $\gamma(a \mapsto c) \leq \gamma(a \mapsto b) + \gamma(b \mapsto c)$

Given two trees, the algorithm is guaranteed to find the sequence of edit operations with the lowest cumulative cost that transforms one tree into the other.

We can directly adopt the tree correction cost as the measure of similarity of two programs α and β:

$$\delta(\alpha, \beta) = \gamma(\text{Tree}_\alpha \mapsto \text{Tree}_\beta)$$

while Tree_α is a "natural" tree representation of a program. An example is presented in Figure 4.4 on the facing page.

Advantages of Tree Correction over LCS Using this algorithm for program similarity scoring has a number of advantages:

- The algorithm calculates the total cost of the correction operation — and thus the similarity score — itself, in one pass. While doing so:
- The algorithm is not limited to "equal–unequal" comparisons. The costs can be assigned dynamically, depending on the particular node content (we give an example later on), as long as the distance metric requirement is satisfied. We estimate that this constraint does not pose a grave restriction for our approach.
- Tree representations of programs do not break the symmetry between equivalent statement parts (most notably within an `if` statement) like linear representations inevitably do.

result =0;	LCS 1	LCS 2	Tree Correction
resultValue =0;	`assign` `assign` no penalty	`assign(int)` `assign(int)` no penalty	name mismatch small penalty

result =true;	`assign`
	`assign`
	no penalty
		`assign(int)`
		`assign(bool)`
		big penalty
			essential type mismatch
			big penalty

result =0;	`assign`
	`assign`
	no penalty
		`assign(int)`
		`assign(short)`
		big penalty
			minor type mismatch
			small penalty

result =1;	`assign` `assign` no penalty	`assign(int)` `assign(int)` big penalty	minor literal mismatch small penalty

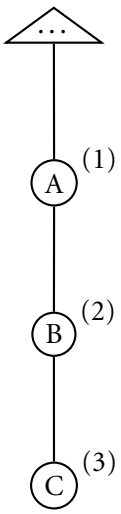
Table 4.3: Penalties for different types of mismatches/algorithms

Example 4.4.4 Table 4.3 shows the performance of different program similarity scores in typical situations. The old program is always result =0;, while the new program varies. For LCS scores the re-use signatures are shown. Either the signatures are equal and there is no penalty, or the signatures are different and there is a big penalty for one insertion and one deletion. The first, very basic, LCS score fails to penalize a radical change of type. This is inadequate for our purposes. The second LCS score includes static expression types into the signatures. This is roughly the score that we use. It still has the deficiency of overpenalizing minor type changes. *Slightly* changing the type of a variable/member — to a related primitive type, or within the object hierarchy — need not always affect the proof. Changes in literals are, however, underpenalized, which enforces a special treatment of boolean literals. The Tree Correction score, in contrast, behaves favorably in all these situations. It performs one "update node" operation, while its cost (and thus the similarity score) can be assessed dynamically, depending on the difference degree of the expressions. ◁

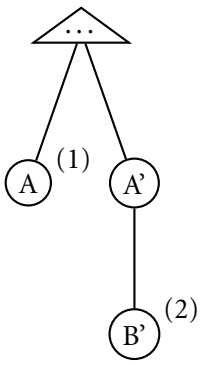
4.4.5 Refinement: Connectivity

Until now we have judged the quality of re-use units by local criteria only. The basis for our decisions at the choicepoints so far was structural similarity of formulae. What we have disregarded is the structure of the overall proof.

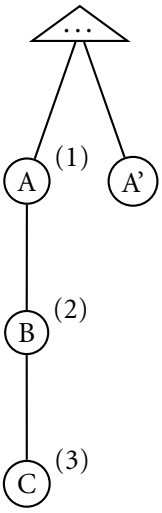
A good way to avoid confusion is to study the connectivity patterns of a proof. While developing a proof it is often the case that "the same" formula or proof part



(a) "Old" proof



(b) Re-use attemt without connectivity: A'$\not\rightarrow$B, B is mis-applied, thus C is blocked



(c) Successful re-use with connectivity

Figure 4.5: Connectivity in proof re-use (in brackets: re-use order)

is tackled several times in succession. Noting this, we want to achieve that related proof steps working in a certain locality are not torn apart during re-use.

First, we formalize the notion that we have — admittedly vaguely — described as "the same formula":

Definition 4.4.1 (Formula connectivity) Let P be a proof step starting with the sequent S and applying taclet T. $S_1, \ldots, S_n$ are the children sequents of P after the application of T. If the rule T has the focus formula $\phi \in S$, then every formula $\psi \in S_i$ that T modifies (via the "replacewith" clause) or adds (via the "add" clause) is *connected* to ϕ. We write $\phi \hookrightarrow \psi$. $\lhd$

Thus, formula connectivity boils down in the natural way to the operational "find–replace with–add" semantics of taclets, as is described in Section 1.3.2 on page 10. Now we will use this relation on formulas as glue for defining "related" proof steps:

Definition 4.4.2 (Proof step connectivity) Two proof steps P_1 and P_2 performed in succession are *connected* if P_1 has the focus formula ϕ, P_2 the focus formula ψ, and furthermore $\phi \hookrightarrow \psi$. $\lhd$

This definition gives us a measure of total connectivity for a proof. The more subsequent proof step pairs are connected in terms of the above definition, the stronger is the total connectivity. High connectivity is a desirable property for proofs for several reasons. Not only is the attention of the human attendants not shifted back and forth unnecessarily, but a high grade of connectivity also makes proofs easier to re-use[1]. We can use connectivity information to distinguish between proof steps appropriate and inappropriate for re-use.

Example 4.4.5 Consider the proof trees in Figure 4.5 on the page before. We have an old proof consisting of three proof steps, which we want to re-use for the new proof. As usual, we will identify the proof steps with the sequent nodes they are applied to. The proof steps in the old proof all tackle the "same" formula — they are connected in the sense of the above definition: $A \hookrightarrow B$ and $B \hookrightarrow C$.

Now suppose we have successfully re-used the proof step A, and our marker in the old proof has advanced to the step B. We could apply B right after A, but there is, unfortunately, another, unrelated goal A' containing a formula, where the step B is applicable too. If our similarity scoring fails — even by a small amount — to give preference to the node A, we would apply the proof step B at the node A', which would be a case of confusion. It is well possible that C cannot be applied at B', and

[1] Of course, we do not demand connectivity at any cost. Sometimes breaking the formula connectivity on purpose — for instance, by introducing a lemma — has its own, higher-ranking merits.

the marker is blocked. This situation could be made far less probable if we took connectivity into account. ◁

Extending quality assessment with connectivity Thus, as motivated so far, we introduce yet another factor in the quality evaluation of a possible re-use unit.

- If a re-use unit would give a proof step connected to its predecessor, increase the quality score of the re-use unit by a small amount (15).

Keeping track of connections A question we have left unanswered so far is how we keep track of the predecessor relation between proof steps, since it is not generally respected by re-use. We do so by storing pointers linking proof steps in the new proof with their templates in the old one. This constitutes no additional computational and a medium storage cost. The storage requirement can be reduced even further by eliminating the pointers after re-use has completed a certain proof region. As an additional benefit, we use this information to inform the user of the next rule that re-use facility expected in the given goal (according to the template proof) after re-use has stopped for some reason.

Connectivity granularity The refinement outlined above uses formulae as glue between related proof steps. This represents a middle degree of granularity; the approach can be accordingly up- or downscaled. It could be further refined to the term level, identifying locality relationships between proof steps w.r.t. subterms of a formula. This constitutes a demanding task. On the other hand, the approach is easily coarsened to the level of semisequents and goals:

- If the potential proof step's focus is not formula-connected to the predecessor but lies in the same semisequent (the antecedent or the succedent) of the same goal, increase the score of the corresponding re-use unit by a lesser amount (10).
- Otherwise: increase the score by an even lesser amount (5), if the potential proof step has its focus in the same goal as its predecessor.

Evaluation The connectivity refinement constitutes a classical feedback loop: it amplifies the impact of good decisions made in the past... unfortunately also the impact of bad decisions. The practice, though, has shown that making a bad decision at the top of the chain would seriously hamper re-use in this locality and make us dependent on recovery measures anyway. Insofar, connectivity is a valuable criterion.

Unfortunately, in the re-use context, we have no means of influencing the already given "old" proof. We can only use as much connectivity as is available in it. At this point our appeal goes to the developers of automated proving procedures to see proof connectivity as one of the goals: connected proofs are easier to follow, easier to re-use, and easier on the cache.

4.4.6 Quality Assessment Of Purely First-Order Re-Use Units

Asessing the quality of possible re-use units that do not deal with symbolic execution is an admittedly difficult challenge. This is due to the high "volatility" of first order formulae in a proof. At the moment we employ two criteria:

- The subterm criterion allots a high bonus (+100) if F^{source} and F^{target} are equal modulo renaming of bound names. Otherwise it allots a small penalty (-20).
- The subterm path criterion compares the locations of the foci within the overall formula. A path is a sequence of positive integers identifying the position of a given subterm/subformula within a term/formula. We match the paths of F^{source} in F_{ϕ}^{source} and F^{target} in F_{ϕ}^{target} by the LCS algorithm, and the edit script is scored uniformly, with every deletion worth a penalty of 10 and every insertion a penalty of 5. This and the previous criterion help discern between several possibilities for a focus within the same formula.
- The overall formula similarity criterion endeavors a difference detection between F_{ϕ}^{source} and F_{ϕ}^{target}. This is achieved again with the LCS algorithm. The formulae are linearized in pre-order, names are abstracted to sorts, and the minimal edit script is computed. The script is scored uniformly, with every deletion worth a penalty of 10 and every insertion a penalty of 5. Additionally, the diamonds in the formulae (as mapped correspondent by the above procedure) contribute their similarity scores to the overall result with a weight of 1/4.

The results of the criteria are summed up.

A further useful refinement would be to employ the technique of Melis and Schairer [Sch98], which uniquely marks different occurences of symbols in logical formulae. This technique is related to our connectivity criterion. Its implementation though has been impeded by the fact that KeY uses immutable shared datastructures for terms, which makes direct annotation impossible.

5 The Frontend

The main re-use algorithm introduced in Section 4.3 on page 31 requires an initial list of candidate proof steps (markers) for its operation. The role of the frontend subsystem is to provide initial candidates that allow the optimal utilization of the re-use potential of the old proof. The way these markers are generated is dependant on the type of program change we are treating.

5.1 Local Changes

For local changes (Section 6.1.1 on page 57) we extract the differences right from the source files. The actual change detection is performed by the GNU diff utility [GNU, Mye86], which is based on the same algorithm by Eugene Myers that we use for program similarity scoring. The GNU diff is well-known to produce in practice meaningful change sets for modifications of source files. We are using the "unified diff" output format, since it provides the most compact and contiguous change representation.

5.1.1 Unified Diff Format

According to [GNU], the unified output format starts with a two-line header, which looks like this:

```
--- FROM-FILE FROM-FILE-MODIFICATION-TIME
+++ TO-FILE TO-FILE-MODIFICATION-TIME
```

Next come one or more hunks of differences; each hunk shows one area where the files differ. Unified format hunks look like this:

```
@@ FROM-FILE-RANGE TO-FILE-RANGE @@
 LINE-FROM-EITHER-FILE
 LINE-FROM-EITHER-FILE...
```

```
int x;
int result;
result = x/x;
```

(a) Old code

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

(b) New code

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

(c) Unified diff

Figure 5.1: Change detection with GNU diff

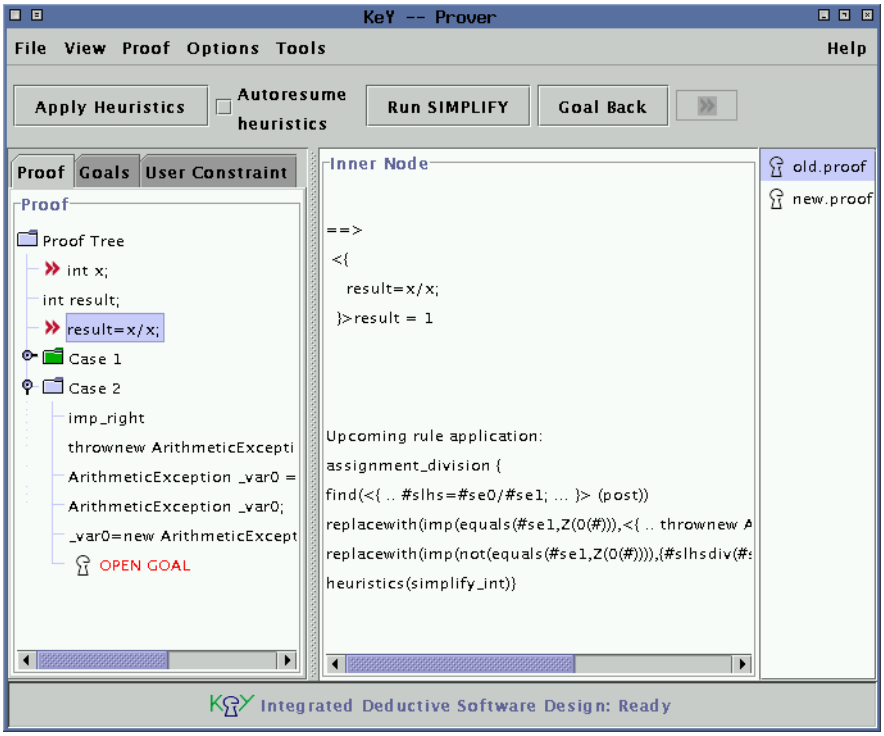


Figure 5.2: Prover window with markers in the old proof

The lines common to both files begin with a space character. The lines that actually differ between the two files have one of the following indicator characters in the left column:

- "+": A line was added here to the first file.
- "-": A line was removed here from the first file.

Surrounding every change hunk are a number of lines (usually three), which are common to both files. These lines provide the *context* of the change.

5.1.2 Translating Diffs into Markers

With the above-shown unified diff we get an overview of identical and changed parts of the programs in question. We want to introduce a marker for every section of identical code. According to our notion of *potential proofs* P_α, introduced in Section 3.4 on page 26, we must identify the first statement of every code section in question.

Technically, the following ingredients provide a glue between source code diffs and proof tree markers:

- From the line ranges stated at the beginning of every difference hunk, exact line numbers of every changed line — and the following context line — can be computed.
- The KeY java parsing process has been modified to embed line count information into the program data structures. Every statement object the parser creates is annotated with the source file name and line number, where the statement begins. The statement objects and the annotations persist throughout the proof.
- In the proof tree object, every node with an associated DL taclet application is tagged with the active statement involved in its proof step (this data is also used to annotate the proof tree in the GUI).

Our approach is thus:

1. Generate initial markers. We create markers for the root node and (if not identical) the top symbolic execution nodes.
2. For every contiguous sequence of change lines in a unified diff determine the immediately following context line. This context line starts a common code section. We are interested in the first relevant statement α in this section (more on relevant statements below). This statement can be uniquely identified by the source file name F and the line number N, where it begins.

3. We search the old proof tree in a top-down manner. Nodes, where symbolic execution takes place, are tagged with active statements. As soon as we encounter a statement tag with the file name F and the line number N we create a marker for that node.

Note that the markers generated by the frontend are "persistent" — they are not consumed during re-use. The unlimited availability of frontend markers is required to control abundant branching (if the programmer inserts such a statement). The detailed discussion of this topic can be found in Section 6.1.3 on page 61.

5.1.3 Additional Remarks on Diff Management

Obtaining the Diffs

The diff file can be provided by the user, or, more systematically, be obtained from CVS. CVS is a very common, Free, version control software. Together ControlCenter integrates CVS for source code management.

When working with CVS, the frontend will assume by default that the old program is the last revision checked in into CVS, while the new program is in the user's working copy. With this setup, obtaining the diff between these two versions is as easy as issuing a `cvs diff` command.

An open issue — for now — is the question of integrity. Re-use can only work, if the old proof loaded into the prover does indeed match the old source copy used in producing the diff. This essential property should be maintained by the proof management subsystem of the KeY tool, which is in development as of now. In the meantime, it is the responsibility of the user to provide consistent inputs.

Remarks on Extracting Relevant Common Statements

Bad programming style Consider the following diff:

```
-if (k>0) { x=17;
+if (k>=0) { x=17;
    y=5;
 }
```

The programmer has violated the Java Coding Conventions [JCC]. The Conventions explicitly discourage putting several statements on the same line. Given this diff, our frontend would produce an erroneous marker with the active statement `y=5;`, instead of the correct `x=17;` Given that this situation is a) uncommon and b) caused by bad programming style, we do not provide a treatment (which could be possible e.g. with an additional intra-line diff).

Statements as expressions There is still one minor problem with our approach for identifying the first common statement after a change. The next statement can be an expression embedded into the current statement, e.g. a parameter in method invocation:

```
-callMethod(11,
+callMethod(17,
    fake++);
 next++;
```

It would be an error to mark `fake++` as the next common statement. We are interested in `next++` instead. We employ heuristics and a bigger context length setting to prevent such mis-attributions from happening.

5.2 Non-Local Changes

5.2.1 Renaming

Renaming is described in Section 6.2.1 on page 62. As a non-local change it cannot be detected meaningfully by the standard diff frontend. The user has to specify explicitly what entity has been renamed in the particular revision. For the same reason (destructive effect on difference detection) renaming must be handled separately from other, local changes. The good news is that handling renaming only requires a single marker, the initial one.

5.2.2 Changes in Class Hierarchy

Every instance method invocation gives rise to an `if` cascade simulating dynamic binding (Section 6.2.2 on page 64). Adding or removing a method implementation to the existing classes, or a whole class altogether, changes the form of these

cascades. This form difference is an obstacle for marker progression, so additional markers below the cascade are required.

Thus, when an implementation for the method m() is added to the program, we must mark such proof steps in the old proof that

- have the taclet METHOD_BODY_EXPAND
- this taclet is bound to a formula $\langle \pi \, \mathsf{AnyClass} :: \mathsf{m}(); \, \omega \rangle \phi$

Where AnyClass :: m() is a method body statement for the method m().From these markers re-use will be restarted after the new if cascade has been processed.

When a new class is added to the model, the above procedure must be performed for all methods, which it implements or inherits. The proof management facility might provide useful upfront information, which of these methods are indeed used in the current proof.

Removal of method implementations or classes requires the same treatment.

6 Re-Use Scenarios

In this section we want to discuss changes to programs, which our re-use facility can treat. We will summarize the change and its impact on the correctness proof. First we will treat "local changes" to programs, that is changes resulting from an insertion or deletion of a few statements. We will represent different change types by "change schemes", which can be read as unified diffs described in Section 5.1.1 on page 51.

We will use diagrams to clarify the arrangement of proof parts before and after the change. Please note that the diagrams are only schematic due to the very elusive nature of proofs about programs as discussed before. In particular, parts shown in dashed style need not be the same or even present.

6.1 Discussion of Possible Program Changes: Local Changes

6.1.1 Inserting a Non-Branching Statement

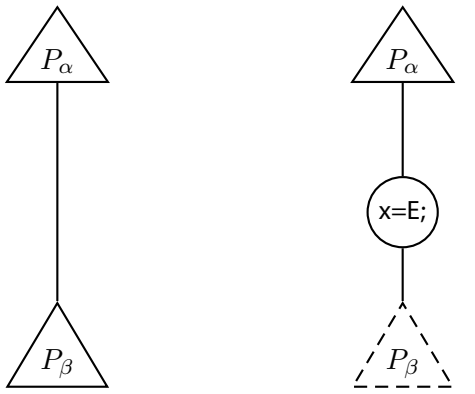
Let's assume that the new program originates from the old by inserting a single very simple statement γ. If γ is simple enough, the proof about γ will not branch. There are a number of non-branching statements, here we will concentrate on the most basic one: a simple assignment[1]. This situation can be described by the following change scheme:

Change scheme

	α
+	`x=E;`
	β

Frontend marks P_α (initial), P_β (post-change)

[1] An assignment is *simple* when evaluating neither the left-hand side nor the right-hand side can have a side effect. E.g. `i=17;` is simple, while `o.a=17;` is not.



(a) Old proof (b) New proof

Figure 6.1: A program change not leading to branching

Change impact Stage 1: Since the α part is identical in both programs, P_α can be re-played in the new proof 1:1 with the initial marker. After α has been executed the re-use will stop due to our "no confusion" postulate.

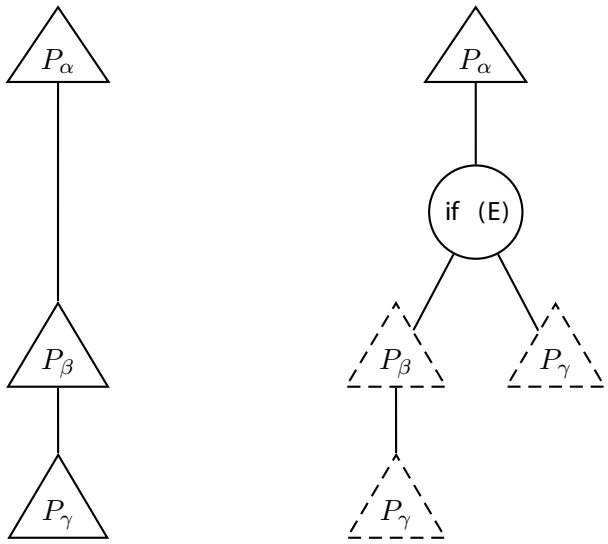
Stage 2: The inserted assignment now has to be executed de novo. As noted above we assume that the assignment is simple and can be executed in one step. An update is thus added to the diamond and the symbolic execution is complete.

Stage 3: Now we are in the situation where the execution of β is probably possible. The corresponding P_β marker is available. Following obstacles can hamper this enterprise:

- The added update changes the term structure, thus rule application positions from the old proof become invalid. This obstacle is overcome by performing a search through all sequent positions for the applicability of a given rule.
- The program change and thus the proof change (in this case an additional assignment/update) might make P_β inapplicable in the new context and a new approach necessary. This is an intrinsical issue with performing amendments. We must solely be *ready* to re-use P_β if appropriate after the change. This circumstance is implied with the dashed style in the diagram.

6.1.2 Inserting a Statement: Conservative Branching

The situation gets more complicated when the new program originates from the old by inserting an `if` statement. A change common in practice is when a part of



(a) Old proof (b) New proof

Figure 6.2: The proof branches conservatively for this program change

the old program is moved to the then branch (or analogically the else branch). This can be summarized by the following change scheme:

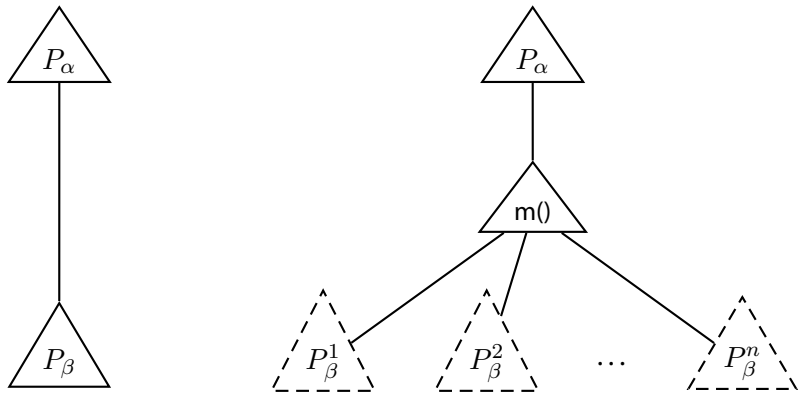
Change scheme

	α
+	if (E) {
	β
+	}
	γ

Frontend marks P_α (initial), P_β, P_γ (post-change)

Change impact Stage 1: Since the α part is identical in both programs, P_α can be re-played in the new proof 1:1 with the initial marker. After α has been executed the re-use will stop due to our "no confusion" postulate.

Stage 2: The inserted assignment now has to be executed de novo. For the purity of the example we assume that the condition E is simple enough to be evaluated without branching the execution tree. Then the execution of the if leads only to a conservative branching (with or without branching of the proof tree).



(a) Old proof (b) New proof

Figure 6.3: The proof branches fully with this program change

Stage 3: We obtain two branches, one can be attacked with the P_β marker and the other with the P_γ marker, both available from the frontend.

6.1.3 Inserting a Statement: Full Branching

The new program originates from the old by inserting a fully branching statement. This could be one of (but by no means limited to):

- a general form of the `if` statement: both `then` and `else` parts are new; evaluation of the condition might lead to even more branching.
- a method call with a high level of overriding
- a statement with an embedded complex boolean expression

We demonstrate this by the following change scheme:

Change scheme

	α
+	m();
	β

Frontend marks P_α (initial), P_β (post-change)

Change impact Stage 1: Since the α part is identical in both programs, P_α can be re-played in the new proof 1:1 with the initial marker. After α has been executed the re-use will stop due to our "no confusion" postulate.

Stage 2: The inserted assignment now has to be executed de novo. This time we assume no restrictions on the nature of the inserted statement(s). This way we have to expect (and deal with) an unknown degree of branching of the execution tree. In our example a method call would generate at least one new branch for every method implementation within the scope of polymorphism.

Stage 3: After the execution of the inserted statement we are left with an (arbitrary) number of execution branches, with a potential continuation P_β awaiting at every one of them. The problem now is that we in general do not have enough markers to attack all of them!

Controlling abundant branching Two possible solutions for the above problem have been evaluated:

- Make frontend markers persistent. When a proof step marked by the frontend is re-used the children nodes are marked but the original marker is not deleted and remains available for future use. This solution has the drawback of increasing the marker space and thus making confusion more probable.
- Attach marker lists to diamonds. Instead of maintaining a global marker list, a separate list is attached to every goal. The re-use procedure is amended as follows:

  1. Whenever a proof step is re-used, the computed children node markers are distributed along the goal lists. This is alway possible in a non-ambiguous way since the same taclet is being applied in the old and in the new proof resulting in the same number of new goals for this application.
  2. Whenever a de novo proof step is performed in the (new) proof, the marker list is copied to the new goal(s). In case of several new goals every one of them receives the parent markers. This duplication guarantees that we have enough markers to attack all subtrees if the inserted statements induce excessive branching.

  This solution has the drawback of increased complexity and encumbered user interaction.

At the moment, the first solution is implemented in the re-use facility.

6.1.4 Deleting a Statement

Change scheme

	α
-	β
	γ

Frontend marks P_α (initial), P_γ (post-change)

Change impact Deletion works along the same lines as insertion; we just read the previous change schemes in reverse. Since the α part is identical in both programs P_α can be re-played in the new proof 1:1 with the initial marker. After α has been executed the re-use will stop due to our "no confusion" postulate. Then there are one or more P_γ markers available (depending on the branching degree of β), which should allow to finish off the proof. Granted, the multiple P_γ subproofs might well differ significantly, and only one of them is possibly productive for re-use in the new context. We conjecture though, that it is the subproof with the highest similarity and quality score of the first proof step that we are interested in. Thus, this subproof will be selected automatically. Otherwise, we still have the recovery procedure at our disposal.

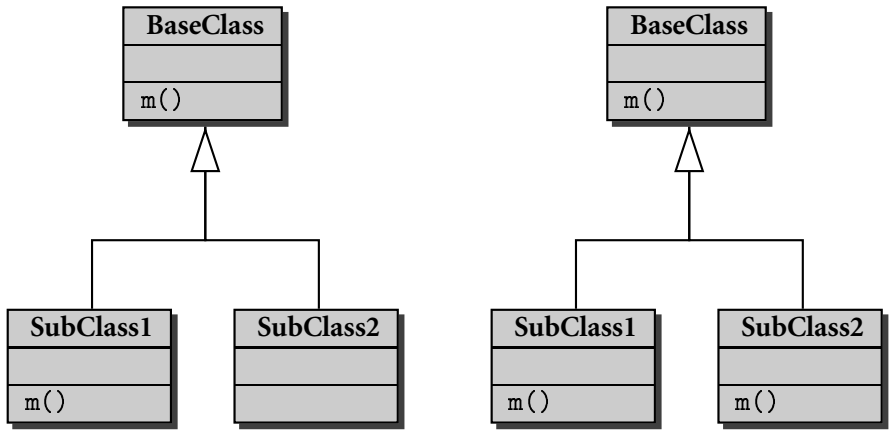
6.2 Discussion of Possible Program Changes: Non-Local Changes

6.2.1 Renaming

The first non-local change is a very common amendment to a program: renaming a variable, a class member or a whole class. It is non-local in nature, because the renamed entity may appear an unlimited number of times throughout the code. On these grounds we cannot summarize a renaming with a change scheme. And furthermore, for the same reasons, we cannot use our usual diff frontend!

We need to traverse the old proof and perform renamings in it — which has to be done very carefully, as not to confuse equally named entities with different scope. Here we can achieve a lot with our general re-use framework after performing some adjustments.

On the backend side, as the sequence of rule applications in the proof remains the same after renaming, the changes must be accommodated within the similarity function. The easiest cases are those, where our similarity function is name-agnostic (our LCS score, for instance, does not discern variable names). Then we



(a) Old model (b) New model

Figure 6.4: Model change: adding a method implementation

can re-play the old proof with the initial marker for the new proof obligation and obtain a valid new proof. In cases where our similarity function is name-pedantic (the LCS score assigns large penalties for type mismatches) we use an equivalence table, which records what class(es) the user has renamed, to equalize the signatures. Renaming tolerance can also be gracefully handled with a tree correction score and its capability of dynamic cost allocation.

6.2.2 Model Changes

As we have have explicated in the section on Dynamic Logic, the statements in a diamond of a formula constitute only a small part of a complete Java program. The rest is hidden from immediate view as context. Since the context is part of the proof, a change of context demands a new proof — a case for proof re-use. We now consider the most important context changes.

Changes In Called Methods

Changes in the implementations of the called methods do not pose a challenge for our re-use scheme. The KeY system supports two modes for treating method in-

vocation: using the specification of verified methods or using the implementation. After a change in the called method, its specification can no longer be used, until the correctness of the method is proven anew. It is the responsibility of the proof management facility to maintain proof integrity in this case.

When using the implementation, this problem does not arise, as the body of the called method is simply "inserted" at the point of the invocation. As both methods — the caller and the callee — live in the common "proof space", this poses no challenge for the backend. The same is true for the frontend, which can even work with different source files, if the caller and the callee are in different classes.

Method Overriding

Since the Java language is based on polymorphism and dynamic binding, the semantics of every (instance) method invocation expression is dependent on the model. Several method implementations can be available for the given method signature in consequence of extension and method overriding. The particular method body to be executed is selected at run-time, depending on the actual object type of the target reference. This process is described in §15.11 of the JLS [GJS96]. In KeY this behavior has to be mimicked by the method invocation rule METHOD_CALL. Due to the exhaustive nature of symbolic execution, a case distinction over all pertinent implementations must be made.

We want to discuss an exemplary change of adding a method implementation to the model, as shown in Figure 6.4 on the page before. Different models lead to different expansions of the instance method invocation expression `o.m()` (`o` has the declared type `BaseClass`). The resulting `if` cascades are shown in Figure 6.5 on the facing page. Since the second cascade requires a different symbolic execution than the first one (the cases are different) the leading marker in the old proof will eventually block. The new cascade has to be expanded manually; subsequently re-use can continue with the method body statement markers introduced by the frontend.

6.2.3 Changes in Specification

As we have mentioned in the Section "Motivation", constant revision of a program is not the only activity in the formal software development process. Deficiencies of the specification may too become apparent and mandate a correction. One could pose the question whether proof re-use is viable in this situation.

We have left this question out of our vision range for the following reasons. Specification is given in a high-level language. Between the specification and the actual proof obligation lies a translation step (performed by the verification middleware

```
{ BaseClass o; o.m() }
```

(a) Simple method invocation...

```
{method−frame(): {
        if (o instanceof SubClass1) SubClass1 :: o.m();
        else
          if (o instanceof SubClass2||d instanceof BaseClass) BaseClass :: o.m();
    }
}
```

(b) ...expanded for the old model...

```
{method−frame(): {
        if (o instanceof SubClass2) SubClass2 :: o.m();
        else
          if (o instanceof SubClass1) SubClass1 :: o.m();
          else
            if (o instanceof BaseClass) BaseClass :: o.m();
    }
}
```

(c) ...or the new model

Figure 6.5: Method invocation depends on the model

component) that is quite complex and hardly "stable". It is thus to be expected that even small changes in the specification would lead to disruptive changes in the proof obligations. However, some success might still be possible with our re-use scheme for really minor changes. The user would simply need to set the initial marker in the old proof.

7 Implementation

7.1 Overview of the Implementation

The actual implementation mirrors in a straightforward way the entities and procedures described in the previous chapters. The heart of the system is a ReuseListener (which is an extension of an InteractiveProofListener). The ReuseListener is notified of all ProofEvents, which are fired when a proof changes, e.g. when a rule has been applied.

The listener contains a vector with candidate proof steps, which are just references to proof nodes. According to the main re-use algorithm, these nodes are analyzed and appropriate ReuseUnit objects are created. A quality score is calculated from the content of a ReuseUnit via pluggable similarity functions, like the LCS function. If the highest quality score in this iteration lies above the cut-off threshold then re-use is possible and the KeYMediator is notified to enable the re-use button in the prover GUI (otherwise the button is disabled). By pressing this button the user starts actual re-use, which will continue automatically as long as possible.

7.2 LCS Similarity Function

The basis for our program similarity score is a difference detection algorithm by Eugene Myers. This algorithm has been serving its purpose well for quite some time inside the standard GNU diff utility. We have used a Free implementation in Java by Stuart Gathman [Gat], which has been derived directly from the GNU diff 1.15 source.

The algorithm can use two optional heuristics for speed-up. The first one activates heuristic path selection, which gives good results on inputs with a constant small density of changes. The second one discards common subsequences at the beginning and the end of the inputs before performing the actual difference detection. We turn both of these heuristics off in our instance, since this guarantees the minimal result, and performance is not an issue with our relatively small inputs and a quite limited alphabet.

The input of the algorithm are two vectors of statement signatures. Statement signatures are strings produced by the methods reuseSignature() embedded into individual statement objects. The base class JavaProgramElement provides a standard implementation of the method, which is overridden in subclasses like Assignment and MethodBodyStatement according to our definition of a re-use signature. To achieve linearization of a complex program data structure into a series of signatures a customized JavaASTWalker is employed.

7.3 Detecting Model Changes

The KeY system uses the CASE tool Together ControlCenter for modeling tasks. Together ControlCenter provides an extensive set of programming bindings (called Open API) that can be used to extend and enrichen the application. As part of the Open API, the package com.togethersoft.openapi.model.elements defines a special ModelChangeListener, which can be added to the Model. The listener is notified in case of a ModelChangeEvent, and receives access to information about added/deleted/modified UML nodes (classes).

Currently (TogetherCC 6.0.1) this package is marked "early access", and the model change event signaling is not functional. For these reasons, model change events have to be signaled manually.

8 Conclusion; Perspective

We have presented a technique for proof re-use in Java software verification. This technique results in a labor-saving when developing software and proving it correct. It allows to "recycle" previous proof construction efforts after a (relatively small) amendment to the program has been made.

We have shown that it is possible to perform efficient proof re-use without resorting to a complicated structural analysis. Instead, we exploit a few general properties of our calculus and concentrate our efforts on the development of formula similarity measures. Thus we predict that our re-use facility will need no to only minimal changes when support for more language features — for instance floating point numbers or concurrency — is introduced into the KeY system.

Our tests with different scenarios have shown the developed re-use facility to be efficient, robust, and perform well in a wide range of situations. The software written for this thesis has been integrated into the KeY system and is available with it at [KeY]. We have also presented directions for further refinement. There are also a number of topics that appear of further interest to us.

An interesting direction for future research would be extending the re-use facility to handle refactoring changes. Refactoring is a technique to restructure code in a disciplined way. It is rapidly gaining importance in developing and maintaining object-oriented software. Together ControlCenter already puts an extensive refactoring toolbox at the disposal of the developer. It would be very advantageous if KeY could automatically re-use old correctness proofs after detecting a refactoring operation.

Another venue worth exploring is an adaptation of external re-use, as we present it, for internal re-use tasks. Already at present, nothing prevents the user from starting re-use with the source and target proof being the same. Re-use markers can be manually set within the same proof that one currently works on, transferring the proof steps from a subproof to another location. When proving programs that work with complex data structures and show a certain degree of redundancy, this may be a good labor saving technique. It remains to be seen how fruitful an approach based on this pronciple would be after more experience with such programs is gathered.

A Selected Taclets of the KeY Calculus

Below we present a commented catalog of taclets that play an important role in this work. The syntax has been slightly adapted for better readability.

IF_EVAL

find ($\langle\{..$ if (#nse) #s0 ...$\}\rangle$ *post*) varcond (boolean #boolv new)
replacewith ($\langle\{..$ boolean #boolv; #boolv = #nse; if (#boolv) #s0 ...$\}\rangle$ *post*)

If the condition of an `if` statement can have side effects — and this is almost always the case — then this taclet must be applied first to "evaluate" this non-simple expression. This taclet does not cleave the execution tree.

IF_ELSE_SPLIT

find ($\Rightarrow \langle\{..$ if (#se) #s0 else #s1 ...$\}\rangle$ *post*)
replacewith ($\Rightarrow \langle\{..$ #s0 ...$\}\rangle$ *post*) add ($\#se =$ TRUE $\Rightarrow$);
replacewith ($\Rightarrow \langle\{..$ #s1 ...$\}\rangle$ *post*) add ($\#se =$ FALSE $\Rightarrow$)

This taclet can be applied when the condition of an `if` statement is simple (i.e. can have no side effects). Applying this taclet causes the proof tree to branch with two child nodes.

IF_ELSE_SPLIT_IMP

find ($\langle\{..$ if (#se) #s0 else #s1 ...$\}\rangle$ *post*)
replacewith (($\#se =$ TRUE $\rightarrow \langle\{..$ #s0 ...$\}\rangle$ *post*) &
($\#se =$ FALSE $\rightarrow \langle\{..$ #s1 ...$\}\rangle$ *post*))

The same as above, but the proof tree does not branch. The execution tree *must* branch nonetheless.

IF_ELSE_UPDATE_TRUE

find ({#se := TRUE}⟨{.. if (#se) #s0 else #s1 ...}⟩ *post*)
replacewith (⟨{.. #s0...⟩ *post*)

A nice version of an if taclet, applicable when the positive valuation of the condition is definitely known (from an "update" attached to the formula, see [Bec01] for more on updates). Due to this additional knowledge, the execution tree does not branch. Also available as IF_ELSE_UPDATE_FALSE.

LESS_THAN_COMPARISION

find (⟨{.. #slhs = #se0 < #se1; ...}⟩ *post*)
replacewith ((#$se0 < se1$) → ⟨{.. #slhs = TRUE; ...}⟩ *post*) &
(!(#$se0 < se1$) → ⟨{.. #slhs = FALSE; ...}⟩ *post*))

If the condition of an if statement contains some kind of comparison, then it's seldomly an if-related taclet that makes the execution tree branch. It's rather a taclet like this one. Commonly followed by an IF_ELSE_UPDATE_TRUE and IF_ELSE_UPDATE_FALSE.

METHOD_CALL

find (⟨{.. #mr ...}⟩ *post*)
replacewith (⟨{.. #method-call(#mr); ...}⟩ *post*)

This is the taclet for symbolic method invocation. The taclet looks so simple, since the complexity is hidden within the #method-call meta-construct.

CUT

add ($b \Rightarrow$);
add ($\Rightarrow b$)

A classical "no-find" taclet. Can be used for a case distinction or to apply a lemma.

INT_INDUCTION

$$\begin{array}{l}\text{add } (\Rightarrow \{nv \leftarrow 0\}(b));\\ \text{add } (\Rightarrow \text{all } nv.((nv \geq 0 \;\&\; b) \rightarrow \{nv \leftarrow \text{succ}(nv)\}b))\\ \text{add } (\text{all } nv.(nv \geq 0 \rightarrow b) \Rightarrow)\end{array}$$

Another “no-find” taclet. Induction is used, among other things, to prove correctness of loops. Consists of a base case, step case, and the conclusion.

B An Example

We demonstrate our approach using the two programs from Figure 5.1 on page 52. The "old" program is shown in Subfigure (a). Its intended behavior and specification is that it always terminates with `result` set to 1. The program, however, contains a bug and cannot be proven correct, since an arithmetic exception can be 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 Subfigure (b). This new program is correct w.r.t. the specification. It always terminates with `result` set to 1. The new proof reflects the case distinction with a completely "new" branch for the case that `x` is zero, and a "non-zero" subproof that handles the division statement. The division rule — oblivious of additional information — causes the proof to split again in the manner similar to the old proof. Here lies the possibility for re-use.

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 Section 5.1.2 on page 53, our technique for computing initial re-use 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 "`result=x/x;`" 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 calculus always treats the left-most statement first):

$$\vdash \langle \texttt{int x; int result;}\ \texttt{if (x==0) result=1; else result=x/x;} \rangle\, (\texttt{result} = 1) \tag{G0}$$

The similarity score for the single possible re-use unit (see Figure 4.2 on page 39 for the computation) is -40, and re-use is performed. We get the new goal

$$\vdash \langle \texttt{int result; if (x==0) result=1; else result=x/x;} \rangle\, (\texttt{result} = 1) \tag{G1}$$

[1] We ignore the fact that Java always initializes the program variable `x` to 0.

and a new re-use 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 result;`". It, also, has one possible focus in the new proof in goal (G1). The similarity score for the resulting re-use unit is -62. That is less than before as there are now less identical parts in the programs containing the focus, and the first difference is closer to the active statement. Nevertheless, re-use is still indicated. The resulting new goal sequent is

$$\vdash \langle \texttt{if (x==0) result=1; else result=x/x;} \rangle (\texttt{result} = 1) \qquad \text{(G2)}$$

and the new candidate is the rule handling the assignment "`result=x/x;`" in the old proof (which happens to be identical to the second initial candidate). This candidate, however, is not applicable in (G2). We have reached a genuinely new part of the program and, thus, of the proof.

To deal with the new program parts, where no re-use is possible, we manually apply the rules for handling the `if` statement and evaluating its condition. The proof tree splits, and we get two subgoals:

$$\vdash \texttt{x} = 0 \rightarrow \langle \texttt{result=1;} \rangle (\texttt{result} = 1) \qquad \text{(G3.1)}$$

$$\vdash \neg(\texttt{x} = 0) \rightarrow \langle \texttt{result=x/x;} \rangle (\texttt{result} = 1) \qquad \text{(G3.2)}$$

The single candidate proof step is still the rule application handling "`result=x/x;`" in the old proof. It cannot be applied to (G3.1), 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 (G3.2). The similarity score for this possible re-use unit is 0. The candidate is re-used, and (G3.2) is replaced by two new subgoals:

$$\vdash \neg(\texttt{x} = 0) \rightarrow \neg(\texttt{x} = 0) \rightarrow (\{\texttt{result} = \mathit{div}(x, x)\} \langle\rangle (\texttt{result} = 1)) \qquad \text{(G3.2.1)}$$

$$\vdash \neg(\texttt{x} = 0) \rightarrow \texttt{x} = 0 \rightarrow \langle \texttt{throw new ArithmeticException();} \rangle (\texttt{result} = 1) \qquad \text{(G3.2.2)}$$

We now have three open goals: (G3.1) is on the "new" branch (no re-use should occur here), (G3.2.1) is on the "normal execution" branch, and (G3.2.2) is on the "division by zero" branch. Things get a bit complicated now, as we also obtain two new re-use candidates. Both are applications of the same rule, namely the rule for handling implications; their foci are:

$$\neg(\texttt{x} = 0) \rightarrow (\{\texttt{result} = \mathit{div}(x, x)\} \langle\rangle (\texttt{result} = 1)) \qquad \text{(C-N)}$$

$$\texttt{x} = 0 \rightarrow \langle \texttt{throw new ArithmeticException();} \rangle (\texttt{result} = 1) \qquad \text{(C-Z)}$$

The two candidates have possible foci in all three open goals: (G3.1), (G3.2.1), (G3.2.2). Thus we obtain six possible re-use units, of which in fact only two are

useful — (C-N) must be re-used at (G3.2.1) and (C-Z) at (G3.2.2). The re-use facility computes the following quality scores for these units:

	(C-N)	(C-Z)
(G3.1)	-53	-81
(G3.2.1)	-35	-77
(G3.2.2)	-58	-35

Following our principle of best re-use score first, the correct combinations are identified and applied. Subsequently the candidate markers move on, and the other 4 possible re-use units become obsolete.

Altogether re-use can be continued to the successfull completion of the proof. If we immediately close the branch that is futile in the new situation (the one under (G3.2.2)), the new proof consists of 45 proof steps, of which 27 has been re-used. This amounts to the optimal reuse performance for the given correction. If we decided to wait with closing this branch we could have even re-used all 191 old proof steps first (most of which are related to the creation and initialization of a new `ArithmeticException` object).

References

[Bec01] Bernhard Beckert, *A dynamic logic for the formal verification of Java Card programs*, Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France (I. Attali and T. Jensen, eds.), LNCS 2041, Springer, 2001, pp. 6–24.

[Gat] Stuart D. Gathman, *GNU Diff for Java*, website at `http://www.bmsi.com/java/#diff`.

[GJS96] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Java Series, Sun Microsystems, 1996.

[GNU] *GNU Diffutils*, website at `http://www.gnu.org/software/diffutils/`.

[Hab00] Elmar Habermalz, *Ein dynamisches automatisierbares interaktives Kalkül für schematische theoriespezifische Regeln*, Ph.D. thesis, Universität Karlsruhe, 2000.

[JCC] Sun Microsystems, Inc., *Code Conventions for the Java Programming Language*, available at `http://java.sun.com/docs/codeconv/`.

[KeY] *KeY Project*, website at `http://www.key-project.org`.

[MS98] Erica Melis and Axel Schairer, *Similarities and reuse of proofs in formal software verification*, EWCBR '98: Proceedings of the 4th European Workshop on Advances in Case-Based Reasoning (London, UK), Springer-Verlag, 1998, pp. 76–87.

[Mye86] Eugene W. Myers, *An O(ND) difference algorithm and its variations.*, Algorithmica **1(2)** (1986), 251–266.

[Ohe01] David von Oheimb, *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*, Ph.D. thesis, Technische Universität München, 2001, `http://www4.in.tum.de/~oheimb/diss/`.

[OME] The Omega Group, Universität des Saarlandes, *Proof planning*, website at `http://www.ags.uni-sb.de/~omega/research/ProofPlanning/`.

[Pau94] Lawrence C. Paulson, *Isabelle: A generic theorem prover*, LNCS 828, Springer, 1994.

[RS93] W. Reif and K. Stenzel, *Reuse of Proofs in Software Verification*, Foundation of Software Technology and Theoretical Computer Science. Proceedings, Lecture Notes in Computer Science, vol. 761, Springer, 1993, pp. 284–293.

[Sch98] A. Schairer, *A technique for reusing proofs in software verification*, Diplomarbeit, Universität des Saarlandes and Universität Stuttgart, 1998.

[Sch02] Steffen Schlager, *Behandlung von Integer Arithmetik bei der Verifikation von Java-Programmen*, Diplomarbeit, Universität Karlsruhe, 2002, available at `http://i12www.ira.uka.de/~key/doc/2002/DA-Schlager.ps.gz`.

[Ste92] Kurt Stenzel, *Wiederverwendung fehlgeschlagener Beweise bei der Verifikation von Programmen*, Diplomarbeit, Universität Karlsruhe, 1992.

[VSE] *Verification Support Environment*, website at `http://www.dfki.de/vse/`.

[ZS89] K. Zhang and D. Shasha, *Simple fast algorithms for the editing distance between trees and related problems*, SIAM Journal on Computing **18** (1989), no. 6, 1245–1262.