Reuse in Software Verification by Abstract Method Calls *

Reiner Hähnle[1], Ina Schaefer[2], and Richard Bubel[1]

[1] Department of Computer Science, Technische Universität Darmstadt
haehnle|bubel@cs.tu-darmstadt.de
[2] Institute for Software Engineering, Technical University of Braunschweig
i.schaefer@tu-braunschweig.de

Abstract. A major obstacle against adoption of formal software verification is the difficulty to track changes in the target code and to accomodate them in specifications and in verification arguments. We introduce *abstract method calls*, a new verification rule for method calls that can be used in most contract-based verification settings. By combining abstract method calls, structured reuse in specification contracts, and caching of verification conditions, it is possible to detect reusability of contracts automatically via first-order reasoning. This is the basis for a verification framework that is able to deal with code undergoing frequent changes.

1 Introduction

Why is formal verification of software so much slower in industrial uptake than hardware verification? After all, it is expensive and requires special expertise for the hardware side, too. And on many occasions, software is safety-critical or at least errors are very expensive to fix, so there should be, a business case for formal software verification. But there is one decisive difference between soft- and hardware: software is, to a much greater extent than hardware, a moving target. Most application software is constantly *changed* to accomodate bug fixes or feature requests and to realize ever shorter time-to-market cycles. As compilation and deployment is cheap, this is no problem. But our current approaches to formal specification and verification do not support fast-paced changes: specification and verification effort is largely wasted when changes occur, systematic reuse is not possible (see Sect. 6 for a discussion). To improve the situation, two things are required: (i) formal specifications and verification proofs must be equipped with a systematic reuse principle; (ii) the reuse principle employed in the targeted code must match the one in specifications and proofs, so it is possible to reflect code changes when reasoning formally about them.

The standard composition principle of modern programming languages are procedure or method calls. To render formal verification scalable with the size

* Partly funded by the EU project FP7-231620 HATS (http://www.hats-project.eu) and by the German Science Foundation (SCHA1635/2-1)

of target programs, most approaches use a formalization of Meyer's *design-by-contract* principle [14], where the behavior of a method is captured in the form of a contract between caller and callee. Contract-based specification has been realized for industrial target languages such as JAVA by specification languages such as JML [12] or in specific program logics, e.g., [2, Ch. 3]. The central idea of contract-based formal verification is to substitute a method call in the target code with a declarative specification of the effect of the call, obtained from the obligation that the callee ensures in its contract. For this to work, two things are necessary: first, the called method must have been successfully verified against its contract; second, the application requirements of the contract must be fulfilled in the call context. The problem of keeping up with target code changes during verification can be formulated in this framework as follows: Assume we have successfully verified a given piece of code p. Now, one of the methods m called in p is changed, i.e., m's contract in general is no longer valid. Therefore, this contract cannot be used in our proof of p which is accordingly broken and must be redone with the new contract of m. If p contains loops, then new invariants must be found, which is time consuming and expensive.

A rather restrictive approach to the change problem is *Liskov's principle* [13]: here, the new contracts must be substitutable for the old ones or, equivalently, only code changes that respect the existing contracts are permitted. But, even with optimizations [8, 9], this is too restrictive in practice, because already very simple code modifications tend to break existing contracts. A more fundamental solution is called for, and this is the contribution of our paper.

Our approach consists of two elements: the first is a *structured reuse principle* for both code *and* contracts. Changes to target code and to contracts are expressed as "deltas" that describe explicitly the difference to the most recent version. Deltas in contracts foster reuse of specifications and make reused parts syntactically explicit. The details are in Sect. 3. But the problem remains that modified contracts are in general no longer applicable in a proof. It is impossible to figure out at method-call time whether a modified contract (or parts of it) might still be applicable or useful for the proof at hand. Therefore, the second ingredient of our approach is to *disentangle* in proofs the analysis of program code and the application of method contracts. This is achieved with *abstract contracts* in Sect. 4. In Sect. 5 we show that by combining abstract contracts, structured reuse of contract-based specification, and caching of first-order goals, it is possible to establish reusability of contracts by first-order reasoning. The result is a verification framework for programs under change, where reusable verification tasks are detected automatically.

2 Verification Framework

As mentioned above, we work in a contract-based [14] verification setting. Our approach is largely agnostic of the target language. Let us assume an imperative, class-based language with calls of methods whose implementation is known. (Resolving, for example, dynamic dispatch is an orthogonal problem to our concerns

and not considered here.) In the examples, we use a subset of JAVA. Our terminology follows closely that of KeY and JML [2, 12]. We use an obvious notation to access classes $\mathtt{C}$ and methods $\mathtt{m}$ within a program $\mathcal{P}$: $\mathcal{P}.\mathtt{C}$, $\mathcal{P}.\mathtt{C}.\mathtt{m}$, etc.

Definition 1. *A* program location *is an expression referring to an updatable heap location (variable, formal parameter, field access, array access). A* contract *for a method* $\mathtt{m}$ *consists of:*

1. *a first-order formula* $\mathtt{r}$ *called* precondition *or* requires clause*;*
2. *a first-order formula* $\mathtt{e}$ *called* postcondition *or* ensures clause*;*
3. *a set of program locations* $\mathtt{a}$ *(called* assignable clause*) that occur in the body of* $\mathtt{m}$ *and whose value can potentially be changed during execution.*

We extend our notation for accessing class members to cover the constituents of contracts: $\mathtt{C.m.r}$ is the requires clause of method $\mathtt{m}$ in class $\mathtt{C}$, etc.

Definition 2. *Let* $\mathtt{m}(\overline{\mathtt{p}})$ *be a call of method* $\mathtt{m}$ *with parameters* $\overline{\mathtt{p}}$*. A* total correctness expression *has the form* $\langle \mathtt{m}(\overline{\mathtt{p}}) \rangle \Phi$ *and means that whenever* $\mathtt{m}$ *is called then it terminates and in the final state* Φ *holds where* Φ *is either again a correctness expression or it is a first-order formula. (Partial correctness adds nothing to our discussion: we omit it for brevity.)*

In first-order dynamic logic [2] correctness expressions are just formulas with modalities. One may also encode correctness expressions as weakest precondition predicates and use first-order logic as a meta language, as typically done in verification condition generators (VCGs). Either way, we assume that we can build first-order formulas over correctness expressions, so we can state the intended semantics of contracts: Validity of the formula $\mathtt{r} \to \langle \mathtt{m}(\overline{\mathtt{p}}) \rangle \mathtt{e}$ expresses the correctness of $\mathtt{m}$ with respect to the pre- and postcondition of its contract. In addition we must state correctness of $\mathtt{m}$ with respect to its assignable clause: one can assume [10] there is a formula $A(\mathtt{a}, \mathtt{m})$ whose validity implies that $\mathtt{m}$ can change at most the value of program locations in $\mathtt{a}$. To summarize:

Definition 3. *A method* $\mathtt{m}$ *of class* $\mathtt{C}$ satisfies *its contract if the following holds:*

$$\models \mathtt{C.m.r} \to \langle \mathtt{m}(\overline{\mathtt{p}}) \rangle \mathtt{C.m.e} \quad \wedge \quad A(\mathtt{C.m.a}, \mathtt{C.m}) \tag{1}$$

The presence of contracts makes formal verification of complex programs possible, because each method can be verified separately against its contract and called methods can be approximated by their contracts (see method contract rule below). The assignable clause of a method limits the program locations a method call can have side effects on.[3]

[3] We are aware that this basic technique is insufficient to achieve modular verification. Advanced techniques for modular verification, e.g. [1], would obfuscate the fundamental questions considered in this paper and can be superimposed.

```
interface IAccount {
  Unit deposit(Int x);
  Unit withdraw(Int x);
  Unit move(Int x, IAccount a, IAccount b);
}
class Account implements IAccount {
  Int balance = 0;

  @requires x > 0;
  @ensures balance == \old(balance) + x;
  @assignable balance;
  Unit deposit(Int x) { balance = balance + x; }

  @requires a != b ∧ amount > 0;
  @ensures a.balance+b.balance <= \old(a.balance)+\old(b.balance)
  Unit move(Int amount, IAccount a, IAccount b) {
      a.withdraw(amount); // dual of deposit
      b.deposit(amount); } }
```

Fig. 1: Specification of a Bank Account Class

Example 1. Fig. 1 shows a simple bank account interface and its implementation. Contract elements appear before the method they refer to and start with a `@`. The method `deposit(x)` is specified with a contract whose precondition in the `@requires` clause says that the balance should be positive. The postcondition in the `@ensures` clause expresses that the balance after the method call is equal to the balance before the method call plus the value of parameter `x`. For simplicity, we use the JML keyword `\old` to access prestate values, but saving old values in renamed locations is equally possible. The obvious dual method `withdraw(x)` is not shown. Method `move(amount,a,b)` moves an amount between accounts. This should not increase the overall balance as stated in its `@ensures` clause.

In a calculus for verification of a method like `move(amount,a,b)` against its contract, methods calls (here, `deposit(x)` and `withdraw(x)`) must be replaced by their contracts using the *method contract rule* to achieve scalability:

$$\mathsf{methodContract}\ \frac{\Gamma \vdash \mathtt{m.r} \qquad \Gamma \vdash \mathcal{U}_{\mathtt{m.a}}(\mathtt{m.e} \rightarrow \langle\omega\rangle\Phi)}{\Gamma \vdash \langle \mathtt{m}(\overline{\mathtt{p}});\omega\rangle\Phi} \tag{2}$$

The rule is applied to the conclusion below the line: in a proof context Γ (a set of formulas) we need to establish correctness of a program starting with a method call $\mathtt{m}(\overline{\mathtt{p}})$ with respect to a postcondition Φ (typically, an ensures clause). The rule uses the contract of `m` and reduces the problem to two subgoals. The first premise establishes that the requires clause is fulfilled, i.e., the contract is honoured by the callee. That is exploited in the second premise, where the ensures clause can now be used to prove the remaining program ω correct. But one must be careful with the possible side effects the call might have had on the

values of locations listed in the assignable clause of $\mathtt{m}$'s contract. As we cannot know these, we use a substitution $\mathcal{U}_{\mathtt{m.a}}$ to set all locations occurring in $\mathtt{m.a}$ to fresh Skolem symbols not occurring elsewhere. It is intentional that we do not commit to a specific calculus or program logic. Soundness of the method contract rules is formally stated here:

Theorem 1. *If the implementation of m satisfies its contract, then rule* (2) *is sound.*

Proof. The method contract rule is fairly standard except for the use of the substitution $U_{\mathtt{m.a}}$ which encodes the assignable clause of the contract. In [4] a theorem is shown from which the correctness of (2) follows as a special case.

Remark 1. In general, a JML contract may involve several *specification cases*, connected by the keyword **also**. It is possible to reduce this by propositional reasoning to proof obligations of the form that occur in the premisses of rule (2). So we assume wlog to deal with a single specification case at verification time. Similarly, each JML specification case may have multiple requires and ensures clauses, implicitly connected by conjunction. For simplicity, we assume wlog the presence of at most one requires/ensures clause per specification case.

The question we focus on in the following is: What happens with a correctness proof when the implementation of a called method changes? Liskov's well-known substitutability principle [13], rephrased in terms of contracts, gives one answer.

Definition 4 (*Substitutable Contract*). *For two methods* $\mathtt{m}$, $\mathtt{m}'$, *with contracts* $\mathtt{m.r}$, $\mathtt{m}'.\mathtt{r}'$, $\mathtt{m.e}$, $\mathtt{m}'.\mathtt{e}'$, $\mathtt{m.a}$, $\mathtt{m}'.\mathtt{a}'$, *the second method's contract is* substitutable *for the first if the following holds:*

$$(\mathtt{m.r} \rightarrow \mathtt{m}'.\mathtt{r}') \wedge (\mathtt{m}'.\mathtt{e}' \rightarrow \mathtt{m.e}) \wedge (\mathtt{m}'.\mathtt{a}' \subseteq \mathtt{m.a}) \tag{3}$$

The next lemma is immediate by the definition of contract satisfaction (Def. 3), propositional reasoning over (3), and monotonicity of postconditions in total correctness formulas.

Lemma 1. *If a method* $\mathtt{m}'$ *satisfies its contract, then it satisfies as well any contract substitutable for it.*

This justifies Liskov's principle and guarantees that a proof stays valid, whenever we replace a method by one whose contract is substitutable for it. As we shall soon see, substitutability is much too restrictive to be practically useful.

3 Delta-Oriented Reuse of Programs and Contracts

If a program evolves due to bug fixes, newly added features or other modifications, the program code itself and also its specification in form of method

```
delta DFee(Int fee);
modifies class Account {
  modifies Unit deposit(Int x) { if (x>=fee) original(x-fee); } }
```

Fig. 2: Delta for introducing a fee to the bank account

contracts changes. To enable systematic reuse for verification, we need to represent changes explicitly. To this end we use the ideas from *delta-oriented programming* [16, 6] (DOP), a code reuse technique originally developed for implementing static variability in software product line engineering. A *delta* is simply a container of descriptions of modifications applied to a program or a specification. It can be thought of as the "diff" between subsequent versions of a program and its associated contracts. Hence, deltas are a flexible concept that can be used to represent anticipated and unanticipated changes of programs and specifications *in a structured manner*. The case for DOP is made elsewhere [6, 16]. Let it suffice to say that deltas achieve a separation of concerns between the modelling of variability and the modelling of data and code design, which are conflated in standard OO languages.

3.1 Program Deltas

Different flavours of DOP can be found in the literature, but they have in common that deltas may add, remove, and modify elements of the target program. At the class level, we use the keywords **modifies**, **adds**, and **removes** preceding the changed class declaration. With **adds**, new classes can be created and with **removes** a whole class can be removed. A modified class contains further directives that may change its method and field declarations, for which the same keywords are used (for fields, only **adds** and **removes** are permitted). In general, the **modifies** directive is a mere convenience, as it can be replaced by a suitable combination of **removes** and **adds**.

A modified method declaration can be completely replaced or it can be a *wrapper* using the **original** call. The keyword **original** stands for a call to the most recent version of the currently modified method and it must match its type signature. The **original** construct makes code reuse possible and is reminiscent of **super** calls in standard OO languages. A main difference between **original** and **super** is that the former, as all other changes contained in a delta, are resolved at compile time, when applying a delta to an existing program. The same method can be modified and wrapped in several subsequent delta applications capturing individual changes.

Example 2. We want to extend the account class of Ex. 1 with a feature to charge a fee for each deposit. This is realized in the delta in Fig. 2. Delta declarations begin with the keyword **delta**, followed by a name and optional parameters, here, the amount of the transaction fee. The delta modifies class `Account` and its method `deposit(x)` by wrapping and reusing the previous version with an

```
product AccountWithFee(Base, DFee(2));
// results in:
interface IAccount { Unit deposit(Int x); }
class Account implements IAccount {
  Int balance = 0 ;
  Unit deposit(Int x) { if (x>=2) balance = balance + (x-2); } }
```

Fig. 3: Result of applying the *DFee* delta to the bank account

original call. The conditional around the call to **original** ensures that the deposited amount is not lower than the fee to avoid counter-intuitive results.

Obviously, the application of a delta to a given program may fail. The **modifies** and **removes** directives implicitly assume the existence of program elements that may be missing, the type signature of an **original** call may not match, etc. Therefore, compilation in DOP is a two-stage process: in a first step, deltas are applied in a user-specified sequence, where the well-definedness of each delta application is checked, and the result is a flattened, delta-free program to be processed further with a standard compiler. To specify products resulting from delta applications, we use the keyword **product**, followed by a name and a sequence of the deltas that are to be applied.

Example 3. Fig. 3 shows the declaration of a bank account product with deposit fee derived from Ex. 1 (for which the name `Base` is used by convention) and subsequent application of the `DFee` delta of Ex. 2, where the parameter is instantiated with 2 units. The declaration of class `Account` is generated automatically by the delta compiler.

3.2 Contract Deltas

Changing program code typically requires to change method contracts since a change might intentionally cause a different functionality that has to be reflected in the contract. Hence, it is natural to extend the concept of deltas to contracts. Following [5, 11], we permit the keywords **adds**, **modifies**, and **removes** in front of specification cases and requires/ensures/assignable clauses. If there is more than one specification case we use names to distinguish them. A name clause is of the form "`@case <name>;`" and this name can be used to qualify a **modifies** or **removes** directive.

Going beyond [5], and in analogy to program deltas, we allow reuse of specifications in modified and added contract clauses using the keyword **original**. As with program deltas, this means that the most recent version of the contract clause is replaced for the **original** keyword when the delta is applied. If there is more than one specification case, **original** can be qualified with a name.

Example 4. For the contract of the modified `deposit(x)` method in `DFee` (Fig. 2), we want to reuse the contract of the **original** method shown in Fig. 1. This can

```
delta DFee(Int fee);

modifies class Account {
  // modify the only existing specification case
  modifies @requires \original ∧ x >= fee;
  modifies @ensures balance == \old(balance) + x - fee;
  // add a new specification case
  adds also @case TrivialAmount
           @requires \original ∧ x < fee;
           @assignable \nothing;
  modifies Unit deposit(Int x) { if (x>=fee) original(x-fee); }

  modifies @requires \original ∧ amount>=fee
  of Unit move(Int amount, IAccount a, IAccount b) }
```

Fig. 4: Delta changing the specification of the deposit method

```
@requires x > 0 ∧ x >= 2;
@ensures balance == \old(balance) + x - 2;
@assignable balance;
also
@case TrivialAmount
@requires x > 0 ∧ x < 2;
@assignable \nothing;
Unit deposit(Int x) { if (x>=2) balance = balance + (x-2); }
```

Fig. 5: Result of applying the DFee program and contract delta

look as in Fig. 4. Note that we need two specification cases: one when the fee does not exceed the deposited amount and one when it does. The first is obtained as a modification of the existing contract: in the requires clause, a suitable precondition is added to the **original** requires clause. The previous version of the ensures clause is replaced by a new version which takes the deduction of the fee into account. The assignable clause is untouched. The second case is obtained by **adds**. Again, the **original** precondition is reused. In this case, the balance of the account remains unchanged, which is implied by the new assignable clause. While this is sufficient, it is hard to detect and, therefore, to exploit.

Example 5. If we compute the product shown in Fig. 3 by delta application *including* the base contract and the contract of the delta we obtain the contract of the modified `deposit(x)` method shown in Fig. 5.

3.3 Verification of Deltas

The main question in a formal verification context is: *how to prove that a program delta satisfies its contract delta?* In general, this is not possible for a delta in

isolation, for two reasons: the first is, before actually applying a delta, its code and its contract are partially unknown. In a previous paper [11] we addressed this issue by imposing a number of constraints: occurrences of **original** in requires clauses must be of the form **original** $\vee$ `r'`, in ensures clauses of the form **original** $\wedge$ `e'`, and in assignable clauses of the form **original**$\backslash$`a'`. This ensures substitutability of reused contracts and makes Lemma 1 applicable. The second problem is that a method satisfying its contract might cease to do so after modification of either code or contract. In [11] we imposed two conditions relative to a given partial order $\prec$ of delta applications: (i) modified contracts in $\prec$-larger deltas must be substitutable, and (ii) calls to methods that might have been modified are replaced with the $\prec$-minimal contract of that method. Under these conditions, satisfaction of each contract in the base and in all deltas implies that all contracts in any $\prec$-compatible product are satisfied [11, Thm. 1]. Unfortunately, these restrictions are too severe in practice:

Example 6. Consider the contracts in Figs. 1, 4. The ensures clause of the modified contract introduces the parameter `fee` and bears no logical relation to the clause it replaces. In the added specification case, the implicitly given default ensures clause "`@ensures true;`" is *weaker*, not stronger as required.

4 Abstract Method Calls

Ex. 6 shows that already minor changes to programs violate Liskov's principle. This makes reuse of verification effort problematic in general:

Example 7. Consider method `move()` from Fig. 1 which transfers money between two accounts. Its contract states that money might be lost, because of fees or similar, but it strictly excludes generation of money. Its implementation calls `deposit()` to credit the receiving account. To prove that `move()` satisfies its contract requires to apply the method contract rule (2) to `deposit()`. Changing the contract of `deposit()` to the version in Fig. 5, entails that neither of the two specification cases satisfies Liskov's principle (see Ex. 6). Consequently, in the verification of `move()` no reuse is possible.

The previous example highlights an unfortunate property of the usual verification setup: assuming we use a complete[4] verification method, intuitively, the logical information in the proof with the call to the original `deposit()` should be sufficient to justify the contract of `move()` even for the version with a fee. But this cannot be detected easily, because the proof of `move()`'s contract uses the ensures clause of `deposit()`. Now, when `deposit()`'s contract is changed, it is impossible to disentangle the information from `deposit()`'s contract and the steps used to prove `move()`. To achieve such a separation we need a technique that splits method invocation from the actual contract application. This is the central contribution of this paper and explained now.

[4] Relatively complete, with respect to Peano arithmetic, of course.

```
// abstract contract
@requires R;
@ensures l1 == \def(l1) && ... && ln == \def(ln) && E;
@assignable l1, ..., ln;
// definitions allowing to " instantiate " the contract
@def R = R', \def(l1) = E1, ..., \def(ln) = En, E = E';
```

Fig. 6: Shape of an *abstract method contract*

```
@requires R;
@ensures balance == \def(balance);
@assignable balance;
@def R = x > 0, \def(balance) = \old(getBalance()) + x;
Unit deposit(Int x) {...}
```

Fig. 7: Abstract method specification for `deposit` without fee

The main technical idea is to introduce a level of indirection into a method contract that allows to delay the substitution of its concrete requires and ensures clause. We call this an *abstract method contract*. It has the shape shown in Fig. 6. Its *abstract* section consists of the standard requires, ensures, and assignable clauses. As before, the assignable clause specifies all locations that might be changed by the specified method. The requires and ensures clauses, however, now merely contain placeholders `R`, `E`, and $\texttt{\def(l}_\texttt{i}\texttt{)}$, which are defined by concrete formulas and terms in the *definitions* section, which must contain a definition for each placeholder in the abstract section. The ensures clause is a conjunction of equations specifying the post value of each possibly changed location in the assignable clause and additional properties `E` on the post state. Please note that Fig. 6 is merely a convenient notation. The formal definition of an abstract method contract is given in Def. 5 below.

The main restriction inherent to abstract method contracts is that the assignable clause explicitly lists updatable locations (i.e., it is not abstract). Nevertheless, it is part of the abstract section, so that it is shared by all clients of the contract. This is necessary to ensure that applying any abstract contract for a method has the same result before definitions are unfolded. Another, minor restriction is the equational form, which enforces that the post value for any assignable location is well-defined after contract application. Field accesses occurring in definitions are expressed using getter methods. This ensures that their correct value is used when definitions are unfolded.

Example 8. Fig. 7 reformulates the contract of method `deposit()` in terms of an abstract method contract.

Abstract method contracts are *fully compatible* with contract deltas, with the restriction that assignable clauses may not be changed. The only difference is that all changes specified in a delta are acted upon in the *definition* section

```
@assignable balance;
@requires R;
@ensures balance == \def(balance);
@def R = \original_Base(R) && x>=2, \def(balance) = \old(getBalance())+x-2;
Unit deposit(Int x) {...}
```

Fig. 8: Abstract method specification case for a successful `deposit()` with fee

of an abstract contract—the abstract section remains completely unchanged. In our example, the application of the delta in Fig. 4 results in an abstract contract with two specification cases, one of which is shown in Fig. 8.[5]

It is perhaps surprising that **original** still occurs after delta application. The explanation is that the abstract shape of contracts does not force us anymore to unfold **original** references immediately. As we shall see in Sect. 5, it can have advantages not to do so. To indicate that the **original** has been in fact resolved, we add a reference (here: `Base`) to its container. Now we can define abstract method contracts formally:

Definition 5 (Abstract method contract). *An* abstract method contract C_m *for a method* m *is a quadruple* $(r, e, U, defs)$ *where*

- r, e *are logic formulas representing the contract's pre- and postcondition,*
- U *is an explicit substitution representing the assignable clause, and*
- $defs$ *is a list of pairs* $(defSym, \xi_{defSym})$ *where* $defSym$ *are non-rigid (i.e., state dependent) function or predicate symbols used as placeholders in* r, e, *and* ξ *their defining term or formula. For each* $\texttt{\textbackslash def}(\texttt{l}_\texttt{i})$ *there is a unique function symbol in* $defs$*. For simplicity, we refer to both with* $\texttt{\textbackslash def}(\texttt{l}_\texttt{i})$*, as long as no ambiguity arises.*

Placeholders must be non-rigid to prevent the program logic calculus to perform simplifications over them that are invalid in some program states. To ensure soundness of the abstract setup we add the definitions of the placeholders (i.e., the contents of the definition section of each abstract contract) as a *theory* to the logic, just like other theories, such as arithmetic, etc. This means that the notion of contract satisfaction (Def. 3) is now able to consider defined symbols in abstract contracts. Additionally the rule on the right that substitutes placeholders by their definitions is now obviously sound. The advantage of this setup is that we can still use the old method contract rule (2), which simply ignores the definition section. As we changed neither the satisfaction of contracts nor the method contract rule, Thm. 1 still holds.

expandDef

$$\frac{\xi_{defSym}}{defSym}$$

[5] There is a technicality here about represening multiple specification cases for abstract method contracts. This is inessential and distracting, so we don't give details.

5 Application Scenarios

5.1 Abstract Verification

Rule (2) now uses only the abstract section of an abstract method contract. Hence, its application yields the same result for all method contracts that share the same abstract section. This allows to define the following general verification process: Assume we want to establish that a method m satisfies a contract C. In the first phase the rules of the underlying calculus are applied until only first-order proof goals are remaining. This can be done, for example, with symbolic execution or with a VCG and typically involves manual annotation of the target program with suitable loop invariants. During this phase all calculus rules, including SMT and first-order solvers, except for expandDef may be used to close subgoals. If the implementation of m contains a call to another method n we use an abstract contract C_n for the latter. Because of this, some first-order subgoals will usually remain open. Let us call this partial proof p.

There are two things one can do at this stage: first, we can use the expandDef rules of the definition section of C_n and first-order reasoning on the open subgoals of p. If m satisfies C and suitable invariants were chosen in p, this complete the proof by first-order reasoning. Second, assume now we made changes to n and modified the definition section of its contract, let's call it C'_n. As long as C_n and C'_n have the same abstract section, we can *reuse* proof p completely. To test whether p still satisfies C after the change, it is sufficient to use the expandDef rules of the definition section of C'_n on p. Again, this is a first-order problem. This is significant, because coming up with the right invariants is usually much more expensive than first-order reasoning.

Example 9. Applying the verification rules of KeY [2] to show that `move()` satisfies its contract (Fig. 1), while using abstract contracts of `deposit()` and `withdraw()`, results in a partial proof p with the open first-order goal

$$\texttt{\bsl def(a.balance)} + \texttt{\bsl def(b.balance)} \leq \texttt{\bsl old(a.balance)} + \texttt{\bsl old(b.balance)}$$

If we use the expandDef rules gained from the definition sections (cf. Fig. 7) we obtain the goal

$$\texttt{\bsl old(a.getBalance())} - \texttt{x} + \texttt{\bsl old(b.getBalance())} + \texttt{x} \leq \texttt{\bsl old(a.balance)} + \texttt{\bsl old(b.balance)}$$

which is trivial for an SMT solver. In addition, we can *reuse* p after applying the `DFee` delta to `deposit()`, because the abstract contracts in Figs. 7, 8 have identical abstract sections. With the rules gained from the definition section of Fig. 8 the resulting subgoal is still first-order provable.

5.2 Liskov for Free

A nice feature of our approach is that preservation of changed contracts as justified by Liskov's substitutability principle (Sect. 3.3) is detected automatically.

Let n' be a method whose contract $C_{n'}$ is substitutable (3) for n's contract C_n and assume m invokes n. Abstract verification first constructs a partial proof p_m for m and its contract that has open first-order verification conditions V_m. These contain placeholders from C_n. Assume we can finish p_m by expanding their definitions plus first-order reasoning.

To verify that m still satisfies its contract when n is replaced with n', we proceed as follows: in V_m substitute each placeholder from C_n with its corresponding placeholder from $C_{n'}$. This is possible, because both contracts have identical abstract sections. After expanding the definitions, by substitutability (3), one first-order subsumption step is enough to obtain the definitions from C_n, which have been proven already. Therefore, (complete) first-order reasoning will automatically detect such a situation.

5.3 Experiments

We performed preliminary experiments using the KeY verification system[6]. The necessary abstract method contracts and definition expansion rules have been provided manually. Manual steps for saving and loading of the partial proof were also necessary, but will be fully automatic once proper support for abstract method calls is implemented.

The example used in this paper showed only modest gains by abstract method calls, which is not surprising considering the low complexity of the involved methods. Verifying the more complex method `requestTransaction()` which calls `deposit()` and contains a loop, we achieved savings of 90%. The following table summarizes our results (proof complexity in number of nodes (branches)):

Example	Cached Proof	Proof (`Base`)	Savings	Proof (`DFee`)	Savings
move	100 (6)	477 (10)	21%	517 (10)	19%
reqTrans	887 (20)	976 (23)	91%	979 (23)	91%

5.4 Program Evolution

For verifying a program that evolves by changing methods via delta operations, we can proceed as follows: For each contract C_m of each method m contained in the initial program, we construct a proof (e.g., by VCG or symbolic execution) using the abstract method contracts. We store the proof and also the open subgoals in a cache for future reference. Then, we unfold the definitions in the abstract method contract C_m and use, e.g., an SMT solver to close the open subgoals to verify the method.

If the program evolves by delta application, we consider several cases: If the implementation of a method m and its contract C_m as well as all abstract sections of contracts C_n for methods n called by m remain unchanged, we can completely reuse the stored partial proof for C_m as in Sect. 5.1. For contracts of called methods n that are unchanged, we can reuse previous proof goals stored in

[6] The experiments are available at `http://www.key-project.org/cade13/tud/`

the cache. If the contracts of the called methods n are changed, but substitutable for C_n, we obtain the proof as described in Sect. 5.2. If the contracts of called methods are changed in other ways, we unfold their definitions and obtain the proof by first-order reasoning and store the new proof goals in the cache. If the implementation of method m or its contract changed, we need to construct a new proof for C'_m as for the initial program. Here, we do not unfold the definitions of **original** when the partial proof is stored in the cache in order to be able to reuse partial proofs also for different instantiations of original. If, in the newly constructed proof, contracts for methods n called by m did not change or their contracts are substitutable wrt. previous contracts, we can reuse proof goals stored in the cache or apply the principle of Sect. 5.2. If a method is newly added, it has to be verified from scratch and the proof is stored in the cache.

6 Related Work

Previous approaches to deductive verification of evolving programs [3, 15] propose proof replay techniques to ameliorate verification effort. The old proof is replayed as long as possible, and at a mismatch a new proof rule is guessed. The paper [3] uses a similarity function to determine which parts of a previous proof can be reused, while [15] uses differencing operations. Unlike our work, proof replay is unrelated to the program or specification reuse principle.

In [9], a set of allowed changes to evolve an OO program is introduced which is similar to delta operations. For verified method contracts, a proof context is constructed which keeps track of the shown proof obligations. Program changes cause the proof context to be adapted so that the proof obligations that are still valid are preserved and new proof goals are created. The proof context is similar to the proof cache proposed in Sect. 5.4, but reuse only happens at the level of contracts, not on the level of (partial) proofs as in our work.

Several approaches have been proposed recently that target efficient verification of delta-oriented programs. These works are set in the context of software product line engineering where *static* program variability is considered in contrast to program evolution which is considered in this work. In [5], it is assumed that one program variant has been fully verified. From the structure of a delta to generate another program variant it is analyzed which proof obligations remain valid in the new product variant and need not be reestablished. The main result of [11], and its restrictions, are discussed in Sect. 3.3. In [7], methods in a delta are verified based on a contract which makes assumptions on the contracts of the called methods explicit. The main difference to our presented work is that reuse in the above approaches only happens at the level of the proof obligations limiting their reuse potential.

7 Conclusion

We presented a framework for systematic reuse of verification effort for programs and specification under change. Its distinctive feature is that reuse takes place

at the level of code, specification, and proofs with a matching reuse principle. Our work highlights the importance of automated first-order reasoning in verification of programs that undergo frequent changes. Detaching the *usage* from the *validation* of contracts turns the test for reusability of previously cached results from a specific verification problem into a general first-order problem. A logical next step is to investigate the nature and the relative difficulty of the resulting first-order problems.

References

1. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *J. of Object Techn.*, 3(6):27–56, 2004.
2. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.
3. B. Beckert and V. Klebanov. Proof reuse for deductive program verification. In *SEFM*, pages 77–86. IEEE Computer Society, 2004.
4. B. Beckert and P. H. Schmitt. Program verification using change information. In *Proc. SEFM, Brisbane, Australia*, pages 91–99. IEEE Press, 2003.
5. D. Bruns, V. Klebanov, and I. Schaefer. Verification of software product lines with delta-oriented slicing. In B. Beckert and C. Marché, editors, *FoVeOOS 2010, Revised Selected Papers*, volume 6528 of *LNCS*, pages 61–75. Springer-Verlag, 2011.
6. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457. Springer-Verlag, 2011.
7. F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A transformational proof system for delta-oriented programming. In *SPLC (2)*, pages 53–60, 2012.
8. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.
9. J. Dovland, E. B. Johnsen, and I. C. Yu. Tracking behavioral constraints during object-oriented software evolution. In *ISoLA (1)*, pages 253–268, 2012.
10. C. Engel, A. Roth, P. H. Schmitt, and B. Weiß. Verification of modifies clauses in dynamic logic with non-rigid functions. Technical Report 2009-9, University of Karlsruhe, 2009.
11. R. Hähnle and I. Schaefer. A Liskov principle for delta-oriented programming. In T. Margaria and B. Steffen, editors, *5th International Symposium, ISoLA 2012*, volume 7609 of *LNCS*, pages 32–46. Springer, Oct. 2012.
12. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. *JML Reference Manual*, Sept. 2009.
13. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
14. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10), Oct. 1992.
15. W. Reif and K. Stenzel. Reuse of proofs in software verification. In *FSTTCS*, pages 284–293, 1993.
16. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th SPLC*, Sept. 2010.

A Experiments

In this section we describe briefly the experiments mentioned in Sect. 5.3. All files and the used KeY version are at `http://www.key-project.org/cade13/tud/`.

The version of KeY we used for our experiments has not been modified, i.e., no support for abstract method call contracts was available. Instead we simulated our approach by

- specifying contracts not in JML, but in JAVA dynamic logic,
- manually specifying the definition expansion rules in KeY's rule language, and by
- manually saving proofs (playing the role of cached proofs/verification conditions) and loading them as reuse mechanism.

All of this can be easily automatized.

After unpacking the files there are two directories, `AcountWithoutFee` and `AccountWithFee`, containing the source files. The specified abstract contracts and expansion rules are contained in a file named `contracts.key`. The file is available in both directories and differs only in the expansion rule definitions, the abstract contracts are identical in both files.

The abstract JAVA dynamic logic contract for `deposit()` as used in our experiments looks as follows:

```
deposit {
  \programVariables {
     Account self;
     int x;
   }
   R_deposit(self, x) ->
        \<{ self.deposit(x)@Account; }\> E_deposit(self, x)
   \modifies { self.balance }
};
```

The contract uses the non-rigid (i.e., state dependent) predicates `R_deposit` and `E_deposit` as placeholders for the method's precondition and postcondition. The rewrite rule expanding the definition of `R_deposit` is shown below:

```
expandDef_R_deposit {
  \schemaVar \term int param;
  \schemaVar \term Account account;

  \find(R_deposit(account, param)) \replacewith (param > 0)
  \heuristics(userTaclets1)
};
```

The rule can be applied to any formula matching its find expression (here: `R_deposit(account, param)`). If applied it replaces the matched formula by the formula specified in the replacewith section of the rule [2].

We explain the steps we performed to verify method `move()` for both variants:

1. We started the KeY system and loaded the file `move.key` which can be found in the directory `AccountWithoutFee`. It specifies the property to be proven for `move()` .
2. We configured the proof strategy in the proof strategy tab as follows:
   - Max. Rule Applications: 2000
   - Logical Splitting: Delayed
   - Loop treatment: Invariant
   - Method treatment: Contract
   - Query treatment: Expand
   - Arithmetic treatment: DefOps
   - Quantifier treatment: Unrestricted
   - User-specific taclets: all off (disables expansion of definitions)
3. Pressing then the run button (green triangle) starts the proof search strategy. The strategies stop with a partial proof whose open goals consist only of first-order sequents. The first-order goals contain occurrences of the used placeholders which have not yet been expanded.
4. We saved the partial proof in directory `AccountWithoutFee` as a file named `moveCachedProof.proof` and copied it to directory `AccountWithFee`.
5. The saved proof allows us to simulate proof reuse with the following steps:
   (a) Load the saved proof from `AccountWithoutFee` or `AccountWithFee`
   (b) Set the proof strategy setting option `User-specific taclets 1` to "on", which enables the definition expansion of the placeholders
   (c) Continue proof search by pressing the run button: this closes the proofs for `AccountWithoutFee` and `AccountWithFee` without any further interaction.

These steps are identical for the second example, with the exception that the file `requestTransaction.key` has to be used instead of file `move.key`.