Noname manuscript No.
(will be inserted by the editor)

Proof Repositories for Compositional Verification of Evolving Software Systems

Managing Change When Proving Software Correct

Richard Bubel · Ferruccio Damiani · Reiner Hähnle · Einar Broch Johnsen · Olaf Owe · Ina Schaefer · Ingrid Chieh Yu

Received: date / Accepted: date

Abstract We propose a new and systematic framework for proof reuse in the context of deductive software verification. The framework generalizes abstract contracts into incremental proof repositories. Abstract contracts enable a separation of concerns between called methods and their implementations, facilitating proof reuse. Proof repositories allow the systematic caching of partial proofs that can be adapted to different method implementations. The framework provides flexible support for compositional verification in the context of, e.g., partly developed programs, evolution of programs and contracts, and product variability.

Partly funded by the EU project H2020-644298 HyVar: Scalable Hybrid Variability for Distributed Evolving Software Systems (`www.hyvar-project.eu`), the EU project FP7-610582 Envisage: Engineering Virtualized Services (`www.envisage-project.eu`), the Ateneo/CSP project RunVar, and the ICT COST Actions IC1402 ARVI (`www.cost-arvi.eu`) and IC1201 BETTY (`www.behavioural-types.eu`).

R. Bubel, R. Hähnle
Department of Computer Science, Technische Universität Darmstadt
E-mail: {bubel, haehnle}@cs.tu-darmstadt.de

F. Damiani
Department of Computer Science, University of Torino
E-mail: ferruccio.damiani@unito.it

E. B. Johnsen, O. Owe, I. Chieh Yu
Department of Informatics, University of Oslo
E-mail: {einarj,olaf,ingridcy}@ifi.uio.no

I. Schaefer
Institute for Software Engineering, Technische Universität Braunschweig
E-mail: i.schaefer@tu-braunschweig.de

1 Introduction

Deductive software verification [3] made significant advances in recent years, primarily through the development and improvement of verification tools [28] such as Dafny [24], KeY [4], Why [18] or Verifast [21], and through novel techniques and formalisms for *verification-in-the-small*, such as code contracts [17], dynamic frames [22, 34], and separation logic [30]. However, similar advances have not been achieved for *verification-in-the-large*. In particular, verification systems typically rely on strong assumptions about how modules compose and interact. These assumptions come in two flavors. Approaches based on a closed-world assumption require that all code is developed before verification can start, making software verification a *post-hoc* activity. Approaches based on an open-world assumption require that the behavior of the modules complies with a priori fixed contracts, typically up to behavioral subtyping [26]. In both cases, breaking with the chosen assumption has severe consequences: the verification process for the "infected" part of the software needs to restart from scratch, and may in the worst case cascade through the entire program!

Established software development methodologies interleave development and testing activities and they do not enforce behavioral subtyping. Hence, they go against both approaches discussed above and make formal verification prohibitively expensive. Two very common aspects of software development in particular break current formal verification approaches: *program evolution* and *product variability* [32]. In the first, method implementations change frequently; at each change, the code that relies on the modified implementations must be re-verified. In the second, the sheer number of possible implementations for each method call leads to a correspondingly large number of proof obligations for the code depending on these calls or to very weak contracts.

To better support verification-in-the-large and to increase the degree of reuse during the software verification effort, we believe that proof systems used in formal verification need to improve the separation of concerns between the client and server side of behavioral contracts. Recent progress on this issue was made in [9, 19] with the concept of *abstract contracts* that permits to separate between when a contract is called and when it is instantiated. In the current paper, we generalize abstract contracts using ideas from [11] to a new and systematic framework for verification proof reuse. In the context of object-oriented software development we address both program evolution and product variability by

1. disentangling the verification of a given piece of code from the implementation of its called methods, and
2. systematically caching partial, abstract proofs that can be instantiated with different method implementations.

Our framework is not restricted to a specific binding strategy for method calls or a specific verification logic. Particular binding strategies can be superimposed to express different ways of composing modules (in the object-oriented setting, classes) such as late binding, feature composition, preprocessing, etc.

Paper organization. In Sect. 2 we introduce the programming and verification model of this paper, as well as our running example. Sect. 3 presents abstract contracts and Sect. 4 explains the use of proof repositories. Sect. 5 illustrates how different structuring concepts are expressed in terms of binding strategies. Sect. 6 presents first evaluation results of our approach using an early prototype. Sect. 7 discusses related work and Sect. 8 concludes the paper.

2 A Framework for Contract-Based Verification

2.1 The Programming Model

The pivotal idea of our approach is to carefully distinguish between a method call in the code and an actual method invocation. More precisely, when we encounter a method call during the verification of a program, we do not want to make a commitment as to which method implementation is actually invoked. This is similar to the setting of programming languages that allow late binding (such as JAVA), but we do not want to commit to a particular binding strategy: this is *deliberately left open* to allow our approach to be usable for different method binding strategies, including late binding, feature composition, preprocessing, etc. In particular, we want to be able to *revise* a binding to the invocation of a a different method implementation at any time. Consequently, we use a programming model where the binding of method calls to method implementations is recorded explicitly: a method may be bound to no, to one, or to more than one implementation.

We work in a contract-based [27] verification setting (being the most common approach to deductive verification [3]): every method is specified by a contract. That contract may be trivial, but it must always be present. Defs. 1 and 2 below define programs and contracts simultaneously.

Definition 1 (Program, Class, Signature, Method) A *program* consists of a finite set $\mathcal{C}$ of classes and a finite set $\mathcal{B}$ of method call bindings.

A *class definition* $\mathtt{C} = (\mathcal{F}, \mathcal{I})$ in $\mathcal{C}$ has a name $\mathtt{C}$ and finite sets $\mathcal{F}$ of fields and $\mathcal{I}$ of method implementations. For simplicity, assume all fields are public, there are only default constructors, and no static members. Each class defines a type, but class types are unordered.

A *method implementation* consists of a method declaration, i.e., a method signature and method body, plus a contract for this method declaration. The *method signature* contains the method's name and the types of the formal argument and return parameters. Each method implementation has a unique label, by which it can be referred to (prefixing it by the class name $\mathtt{C}$). There can be more than one method implementation for a given method signature. These are distinguished by their labels. *Method bodies* contain standard statements: local variable declarations, assignments, conditionals, loops, method calls, and return statements.

The *method call bindings* $\mathcal{B}$ are a set of pairs, where the first element is the code position of a method call within some method body and the second

element is the label corresponding to a method implementation in some class of $\mathcal{C}$.

With this definition the binding of method calls need not be deterministic, nor statically determined. This has the advantage that different method selection schemas (e.g., by inheritance, by features, etc.) can be superimposed on the idea of a proof repository. In a *well-formed program*, the body of each method implementation typechecks w.r.t. all the bindings of the method calls that occur in it—a method call that has no bindings typechecks if all its actual parameters typecheck.

Remark 1 Our approach is agnostic of the target programming language, but to be concrete, we write example programs in a JAVA-like syntax. These are assumed to be well-formed, such that, e.g., used fields are properly defined.

The terminology for method contracts in this paper follows closely that of KeY [4] and JML [23]. We use the following notation to access classes $\texttt{C} \in \mathcal{C}$, method implementations `i`, fields `f`, and method declarations `m` within a program $\mathcal{P} = (\mathcal{C}, \mathcal{B})$: $\mathcal{P}.\texttt{C}$, $\mathcal{P}.\texttt{C.f}$, $\mathcal{P}.\texttt{C.i}$, $\mathcal{P}.\texttt{C.i.m}$, etc. When $\mathcal{P}$ or `C` are clear from the context, we can omit them.

Definition 2 (Location, Contract) A *program location* is an expression referring to an updatable heap location (variable, formal parameter, field access, array access). A *contract* for a method declaration `m` consists of:

1. a first-order formula `r` over program locations called *precondition* or *requires clause*;
2. a first-order formula `e` over program locations called *postcondition* or *ensures clause*;
3. a set of program locations `a` (called *assignable clause*) whose value can potentially be changed during execution of `m`.

The notation for accessing class members is extended to contract constituents: `C.i.r` is the requires clause of method implementation `i` in class `C`, etc.

Contract elements appear in the code before the method declaration they refer to and start with an `@`, followed by a keyword (`requires`, `ensures`, `assignable`) that identifies each element of the contract. Analogous notation (`label`) is used to specify the label associated with the method declaration. The JML keyword `\old` is used to access prestate values. We permit JML's `*`-notation in assignable clauses to describe unbounded sets of program locations.

Remark 2 There are several ways to implement access to prestate values. For our purposes it is easiest to assume that for each location `l` that occurs in the assignable clause of a contract, there is an implicitly declared variable `\old(l)` that is set to the value that `l` had at the point when the contract was invoked.

Example 1 Consider a simple class `Bank` through which customers can update the balance on an array `acc` of accounts.

```
1  class Bank {
2    Account[] acc;
3
4  /*@ label    update0
5    @ requires interest >= 0;
6    @ ensures  (\forall int i; 0 <= i < acc.length;
7                acc[i].balance >= \old(acc[i].balance));
8    @ assignable acc[*].balance;
9    @*/
10   void update(int interest) {
11     for (int i = 0; i < acc.length; i++) {
12       acc[i].deposit(interest)
13     }
14   }
15 }
```

Fig. 1: The `Bank` class

1. Fig. 1 shows the class `Bank` with one method implementation labeled with $\texttt{update}_0$. This is a well-formed program, even though the call to `deposit()` is not bound: Programs are *not necessarily executable*, but *always analyzable* by a verifier.
2. Let us add a class `Account` and its method implementation $\texttt{deposit}_0$ (see Fig. 2). The call to `deposit()` in `update()` is still not bound. When to bind and which binding to make depends on the *program composition discipline* which is left open in our approach.
3. Now add an explicit binding from the method call `deposit(interest)` in line 12 to $\texttt{deposit}_0$. This makes $\texttt{update}_0\texttt{.update()}$ executable.
4. Next add a second implementation of `deposit()` labeled $\texttt{deposit}_1$ (see Fig. 3). The binding is not changed, so $\texttt{deposit}_1$ is not called from anywhere.
5. Finally, add a second binding from the method call `deposit(interest)` in line 12 to $\texttt{deposit}_1$. The binding of the call became *non-deterministic*: during execution of `update()` either $\texttt{deposit}_0$ or $\texttt{deposit}_1$ may be invoked.

Example 2 The method implementation $\texttt{Account.deposit}_0$ in Fig. 2 is specified by a contract whose precondition in the `@requires` clause says that the deposited amount should be non-negative. 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`.

In our programming model, we never delete any method implementation or any binding. Which of the existing implementations and bindings are actually used in a concrete execution is outside the programming model. For example, one could superimpose a class hierarchy and a corresponding dynamic binding rule, or one could view a subset of the existing method implementations and bindings as a particular program version evolving in a development process, but this is deliberately left open to render our results general.

```
1  class Account {
2    int balance = 0;
3
4  /*@ label    deposit_0
5    @ requires x >= 0;
6    @ ensures  balance == \old(balance) + x;
7    @ assignable balance;
8    @*/
9    void deposit(int x) { balance += x }
10 }
```

Fig. 2: The `Account` class

```
1  class Account {
2    int balance = 0;
3    final int fee = 2;
4
5    ... implementation of deposit_0 ...
6
7  /*@ label    deposit_1
8    @ requires x >= 0;
9    @ ensures  balance >= \old(balance);
10   @ assignable balance;
11   @*/
12   void deposit(int x) {
13     if (x >= fee) {balance += x - fee}  }
14 }
```

Fig. 3: The `Account` class with a second implementation of `deposit()`

Definition 3 (Subprogram, Complete program) Given program $(\mathcal{C}, \mathcal{B})$, a *subprogram* is a well-formed program $(\mathcal{C}', \mathcal{B}')$ such that $\mathcal{B}' \subseteq \mathcal{B}$ and for each class $\mathtt{C}' = (\mathcal{F}', \mathcal{I}') \in \mathcal{C}'$ there is a class $\mathtt{C} = (\mathcal{F}, \mathcal{I}) \in \mathcal{C}$ with the same name as $\mathtt{C}'$, such that $\mathcal{F}' \subseteq \mathcal{F}$ and $\mathcal{I}' \subseteq \mathcal{I}$.

A subprogram $(\mathcal{C}', \mathcal{B}')$ is *complete* if every method call occurring in a method implementation in $\mathcal{C}'$ appears on the left-hand side of at least one binding in $\mathcal{B}'$.

Each program is a subprogram of itself. A complete subprogram is executable, even though the implementation of a method call need not to be uniquely determined and not every method implementation needs to be reachable.

Example 3 The programs resulting from Steps 3, 4, and 5 in Example 1 are complete.

2.2 Contract-Based Verification

Verification is about proving the correctness of programs. We use the notion of contract-based specification of methods as introduced in the previous section.

This approach was proposed by Meyer in the context of design-by-contract [27] and subsequently adopted by a number of programming languages and verification tools, including the Eiffel programming language, SPEC# [2] or Microsoft's Code Contracts for the .NET platform.

We define partial correctness in the setting of first-order dynamic logic [4]; we omit total correctness and class invariants as neither adds anything essential to our discussion.

Definition 4 Let $\mathtt{m}(\overline{\mathtt{p}})$ be a call to method $\mathtt{m}$ with parameters $\overline{\mathtt{p}}$. A *partial correctness expression* has the form $[\mathtt{m}(\overline{\mathtt{p}})]\,\Phi$ and means that whenever $\mathtt{m}$ is called and terminates, then Φ holds in its final state; the formula Φ is either another correctness expression or it is a first-order formula.

In first-order dynamic logic, 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, which is 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 $\mathtt{i.r} \to [\mathtt{i.m}(\overline{\mathtt{p}})]\,\mathtt{i.e}$ expresses the correctness of a method implementation $\mathtt{i}$ with respect to the pre- and postcondition of its contract. In addition, we must capture the correctness of $\mathtt{i.m}$ with respect to its assignable clause: for the latter, one can assume that there is a formula $A(\mathtt{i.a}, \mathtt{i.m})$ whose validity implies that $\mathtt{i.m}$ can change at most the value of program locations in $\mathtt{i.a}$ (following [16]). Formally, we define:

Definition 5 (Contract satisfaction) A method implementation $\mathtt{i}$ of class $\mathtt{C}$ *satisfies* its contract if the following formula is valid :

$$\mathtt{C.i.r} \to [\mathtt{m}(\overline{\mathtt{x}})]\,\mathtt{C.i.e} \quad \wedge \quad A(\mathtt{C.i.a}, \mathtt{C.i.m}) \tag{1}$$

Here, $\mathtt{m}(\overline{\mathtt{x}})$ is a call to the method declared by $\mathtt{i}$ with formal parameters $\overline{\mathtt{x}}$, which may be referenced in $\mathtt{C.i.r}$.

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 the method contract rule (2) below. The assignable clause of a method limits the program locations on which a method call can have side effects.[1] To keep the treatment simple (and also in line with most implementations of verification systems), we do not allow metavariables to occur in first-order formulas.

To verify a method implementation (such as `update()` in Fig. 1) against its contract in a verification calculus, method calls that may occur in the body (here, `deposit()`) are not inlined, but replaced by their contract using a *method contract rule* to achieve scalability:

[1] We are aware that this basic technique is insufficient to achieve modular verification. Advanced techniques for modular verification, e.g. [1,22,34], would obfuscate the fundamental questions considered in this paper and can be superimposed.

$$\mathsf{methodContract}\ \frac{\Gamma \Longrightarrow \mathtt{i.r} \qquad \Gamma \Longrightarrow \mathcal{U}_{\mathtt{i.a}}(\mathtt{i.e} \rightarrow \Phi)}{\Gamma \Longrightarrow [\mathtt{m}(\overline{\mathtt{p}})]\Phi} \tag{2}$$

The rule is applied to the conclusion below the horizontal line: given a proof context with a set of formulas Γ, we need to establish the correctness of a program starting with a method call $\mathtt{m}(\overline{\mathtt{p}})$ with respect to a postcondition Φ. The latter could contain either the continuation of the program or, if $\mathtt{m}(\overline{\mathtt{p}})$ is the final statement, an ensures clause.

Note that we here assume that a binding of the method call to the method implementation `i` has been added. We also assume that the underlying verification calculus has associated the formal parameters of `m` with the actual parameters $\overline{\mathtt{p}}$. Rule (2) uses the contract of `i` to reduce verification of the method call to two subgoals. The first of these (left premise) establishes that the requires clause is fulfilled, i.e., the contract is honored by the callee. This justifies that it is sufficient to prove the second goal (right premise), where the ensures clause may be used to prove the desired postcondition Φ correct. Here, one needs to account for the possible side effects of the call on the values of locations listed in the assignable clause of `i`'s contract. As we cannot know these, the substitution $\mathcal{U}_{\mathtt{i.a}}$ is used to set all locations occurring in `i.a` to fresh Skolem symbols (see [4, Sect. 3.8] for details). Soundness of the method contract rule is formally stated as follows:

Theorem 1 *If method implementation* `i` *satisfies its contract, then rule* (2) *is sound.*

Proof The method contract rule is fairly standard except for the use of the substitution $U_{\mathtt{i.a}}$ which encodes the assignable clause of the contract. In [6], a theorem is shown from which the correctness of (2) follows as a special case. □

3 Abstract Method Calls

In the method body of $\mathtt{Bank.update}_0$ in Ex. 1, there are two possible implementations for the call to `deposit()`. Clearly, it is inefficient to redo the correctness proof for $\mathtt{Bank.update}_0$ for each of these implementations. For example, the proof of $\mathtt{Bank.update}_0$ might have involved expensive user interaction. In addition, `deposit()` might be called from many other methods.

Intuitively, the arguments in the proof where $\mathtt{Account.deposit}_0$ is used should be sufficient to justify the proof even when $\mathtt{Account.deposit}_1$ is used. After all, the difference between the proofs occurs only at the first-order level and it should be easy to close the gap with the help of automated first-order reasoning, such as SMT solving.

However, the method contract rule (2) does not permit to detect the similarity between both proof obligations easily, because it works with a fixed binding of method call to method implementation. The proof of $\mathtt{Bank.update}_0$

```
1  /*@
2    @ requires R;
3    @ ensures (∀ l ∈ LS; l == \def(l)) && E;
4    @ assignable LS;
5    @ def R == r;
6    @ def LS == {l1,...,ln};
7    @ def \def(l1) == e1, ..., \def(ln) == en;
8    @ def E == e;
9    @*/
```

Fig. 4: Shape of an *abstract method contract*

```
1  /*@ label    deposit0
2    @ requires R;
3    @ ensures (∀ l ∈ LS; l == \def(l)) && E;
4    @ assignable LS;
5    @ def R == x > 0;
6    @ def LS == {balance};
7    @ def \def(balance) == \old(balance) + x;
8    @ def E == true;
9    @*/
10   void deposit(int x) { balance += x }
```

Fig. 5: Abstract method specification for `deposit` from Fig. 2

uses the ensures clause $\texttt{Account.deposit}_0\texttt{.e}$. When $\texttt{deposit}_0$ is changed to $\texttt{deposit}_1$, it is impossible to disentangle the new ensures clause from the steps used to prove $\texttt{Bank.update}_0$.

We want to achieve a separation of method call and actual contract application, as suggested by the programming model in Sect. 2.1. This can be achieved technically by means of *abstract contracts*, as proposed in [9, 19].

The main technical idea is to introduce a level of indirection into a method contract that permits the substitution of the concrete requires, assignable, and ensures clauses to be delayed. We call this an *abstract method contract*. It has the shape shown in Fig. 4 and comprises an abstract section and a definition section. Its *abstract* section (lines 2–4) consists of the standard requires, ensures, and assignable clauses. But these clauses are now mere placeholders, where `R` and `E` are *abstract predicates*, `LS` is an abstract function that returns a set of assignable locations, and the `\def`'s are abstract functions that specify the precise post value of these locations.[2] The *definition section* of the contract (lines 5–8) provides concrete expressions for each placeholder in the abstract section. Fig. 4 merely suggests a convenient notation. The formal definition of an abstract method contract is given in Def. 6 below.

The equational form of the ensures clause is a minor restriction, 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, e.g., `getBalance()` is used to access the `balance` field.

[2] Not all locations in `LS` need to appear in the *def*s. About the ones who do not, nothing is known except what is stated in *E*.

This ensures that their correct value is used at the time when definitions are unfolded.

Example 4 Fig. 5 reformulates the contract of $\texttt{Account.deposit}_0$ in Fig. 2 as an abstract method contract.

The abstract section of an abstract method contract is completely generic and indeed *the same for each method*. Therefore, an abstract contract is completely specified by its definition section and the signature of the method it relates to. This is reflected in the following definition.

Definition 6 (Abstract method contract) An *abstract method contract* for a method declaration m is a quadruple $(r, e, ls, defs)$ where

- r, e are logic formulas representing the contract's pre- and postcondition,[3]
- ls is a set of heap locations representing the assignable locations,
- *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 definitions ξ. For r, e, and ls, as well as for the defined subset of the $\texttt{\bsl def(l}_\texttt{i}\texttt{)}$ with $l_i \in ls$, there is a unique function symbol in *defSym*. For simplicity, we use $\texttt{\bsl def(l}_\texttt{i}\texttt{)}$ as well to refer to that function symbol, as long as no ambiguity arises.

Placeholders must be non-rigid signature symbols to prevent the program logic calculus to perform simplifications over them that are invalid in certain program states.

$$\mathsf{expandDef}\ \frac{\xi_{defSym}}{defSym} \tag{3}$$

To ensure completeness 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, arrays, etc. This means that the notion of contract satisfaction (Def. 5) includes symbols with definitions in abstract contracts. Additionally, Rule (3) above substitutes placeholders by their definitions (by a slight abuse of notation, but with obvious meaning for function symbols), is obviously sound. The advantage of this setup is that we can use the standard method contract rule as follows: Applying the method contract for a method invocation at position `pc`, we instantiate the method contract rule as follows. As precondition `i.r` we use the placeholder predicate $\texttt{R}_\texttt{i\_pc}$ for the method implementation `i` using the program counter `pc` as a unique marker for the specific method call. This placeholder predicate may depend on the heap, method parameters and depending on the programming language other program locations (e.g., a parameter used to pass the this-reference in Java). The postcondition `i.e` is of the shape $\texttt{l} \doteq \texttt{\bsl def(l)}_\texttt{pc} \land \texttt{E}_\texttt{i\_pc}$ where $\texttt{E}_\texttt{i\_pc}$ is a placeholder predicate which depends on the

[3] This implies the limitation that no (not even pure) method calls can occur in pre- and postconditions. This could be lifted or worked around in various ways.

same program locations as the precondition placeholder as well as the method result and the prestate of the method.

The anonymization of changed variables uses a placeholder function $\mathtt{LS}_{\mathtt{i\_pc}}$ representing a set of program location (those which may be changed by the method). The update $\mathcal{U}_{\mathtt{i.a}}$ becomes the update $\mathcal{U}_{\mathtt{LS}_{\mathtt{i\_pc}}}$ which is a generic substitution that sets exactly the heap locations in $\mathtt{LS}_{\mathtt{i\_pc}}$ to fresh Skolem symbols. This is expressible provided that quantification over heap locations is permitted in the underlying program logic. Apart from that, the abstract rule is exactly like the old method contract rule, but it ignores the definition section at the time when it is applied.

$$\mathsf{abstractMethodContract}\ \frac{\Gamma \Longrightarrow \mathtt{i.r} \qquad \Gamma \Longrightarrow \mathcal{U}_{\mathtt{LS}_{\mathtt{i\_pc}}}(\mathtt{i.e} \rightarrow \Phi)}{\Gamma \Longrightarrow [\mathtt{m}(\overline{\mathtt{p}})]\Phi} \tag{4}$$

As we neither changed the satisfaction of contracts nor the method contract rule, Thm. 1 still holds.

Example 5 We illustrate the application of rule (4) with the call to `deposit()` at line 12 of Fig. 1 (`pc : 112`) during verification of $\mathtt{update}_0$ in Fig. 1: Applying the contract to the sequent $\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{acc[i].deposit(interest);}]\phi, \Delta$ splits the proof in two branches:

1. $\Gamma \Longrightarrow \{\mathcal{U}\}R_{\mathtt{deposit}_0\_\mathtt{112}}, \Delta$ and
2. $\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathcal{U}_{LS_{\mathtt{deposit}_0\_\mathtt{112}}}\}(\mathtt{balance} \doteq \backslash\mathtt{def}(\mathtt{balance})_{\mathtt{112}} \wedge E_{\mathtt{deposit}_0\_\mathtt{112}}) \rightarrow []\phi, \Delta$

Successive applications of Rule (3) (expandDef) first replace the placeholder by the respective counterpart of the abstract method contract specifications and successively the therein used placeholders such that for instance $R_{\mathtt{deposit}_0\_\mathtt{112}}$ is finally expanded to the concrete precondition $\mathtt{interest} > 0$.

4 A Proof Repository

The idea of a proof repository is that it faithfully records which method implementations have been proven correct for which possible method call bindings of a given program in the sense of Def. 1. Each program change and each new binding gives rise to new proof obligations, which are then added to the proof repository so that it reflects the changed program. Like for our notion of program, we never delete any information in the proof repository.

For simplicity, with each method implementation we associate a single proof obligation of the form (1) of Def. 5. Assume that we have a program logic that reduces such proof obligations to first-order subgoals. This is possible even for incomplete programs (in the sense of Def. 3) using the abstract contracts introduced in the previous section. Of course, there will in general be unprovable first-order subgoals that contain abstract symbols from the abstract contracts. Constructing such partial proofs might involve considerable work; for example, it is generally necessary to supply loop invariants manually. Therefore, it makes sense to store these partial, abstract proofs in a repository for later reuse.

Definition 7 (Proof repository) A *proof repository* for a well-formed program $(\mathcal{C}, \mathcal{B})$ is a finite set of triples (i, σ, ϕ), where:

- i is the label of a method implementation in a class of $\mathcal{C}$ (to simplify the notation, in the following we use the same metavariable i also to refer to the associated method implementation);
- $\sigma \subseteq \mathcal{B}$ is a set of bindings such that only method call locations inside i occur in its domain (σ can be empty, which is denoted by ϵ);
- ϕ is a first-order formula representing a verification condition; it may contain symbols from abstract contracts that originate from applications of rule (2) to method calls in i.

Intuitively, an element (i, σ, ϕ) of a proof repository S expresses that ϕ is a proof obligation that needs to be established for the correctness of i provided that methods called inside i are bound at most to implementations occurring in σ. Querying a proof repository is done by means of obvious projection functions:

Definition 8 (Repository projection functions) Let S be a proof repository for a well-formed program $(\mathcal{C}, \mathcal{B})$, i a method implementation in a class of $\mathcal{C}$ and $\sigma \subseteq \mathcal{B}$. Let s_j be the j'th component of a triple $s \in S$. Then we define:

- $S \downarrow i = \{s \in S \mid s_1 = i\}$
- $S \Downarrow i, \sigma = \{s \in S \downarrow i \mid \ s_2 \supseteq \sigma \text{ and } \not\models \sigma(s_3)\}$, and $\models$ is first-order validity.

Let $b \sqsubset \sigma$ denote that the left-hand side of the binding b is in the domain of the set of bindings σ.

The set $S \Downarrow i, \sigma$ characterizes those proof obligations that are not valid (and thereby not first-order provable) for a method implementation i with local calls bound by the given set of bindings σ. In the next definition we connect proof repositories with the notion of contract satisfaction in our framework.

Definition 9 (Sound proof repository) *Soundness* of a proof repository S for a well-formed program P is defined inductively:

- The empty proof repository $S = \emptyset$ is sound for the empty program $P = (\emptyset, \emptyset)$.
- Assume S is sound for program $P = (\mathcal{C}, \mathcal{B})$. We distinguish three cases:
  Case 0: Extend P to a well-formed program P' by adding a new empty class definition to $\mathcal{C}$ or a new field to a class definition $\mathtt{C} \in \mathcal{C}$. Then S is sound for P'.
  Case 1: Extend P to a well-formed program P' by adding a new method implementation i to a class in $\mathcal{C}$.
  Then we use the underlying program logic to reduce the satisfaction of i's contract (Def. 5) to a (possibly empty) set Φ of first-order proof obligations. Each of these may or may not be provable. In general they contain abstract symbols originating from abstract contracts. Let $S' = S \cup \{(i, \epsilon, \phi) \mid \phi \in \Phi\}$, then S' is sound for P'.

Case 2: Create a new well-formed program P' by adding a new method binding $b = (\text{pos}, i_b)$ to $\mathcal{B}$.
As P' is well-formed, the method call at pos on the left hand-side of b occurs in some method implementation i in a class of P. As S is sound for P and we never remove anything from S, the elements in $S \downarrow i$ must contain all first-order proof obligations for i. We choose those not yet containing a binding for the left-hand side of b and extend them. Thus, let $S' = S \cup S''$, where

$$S'' = \{(i, \{b\} \cup \mathcal{B}, b(\phi)) \mid (\mathcal{F}, \mathcal{I}) \in \mathcal{C},\, i \in \mathcal{I},\, (i, \mathcal{B}, \phi) \in S \downarrow i,\;\; b \not\sqsubseteq \mathcal{B}\}$$

and with $b(\phi)$ we denote replacement of all abstract symbols in ϕ created by the method call at pos with the concrete expressions given by the contract of i_b. Then S' is sound for P'.

The extension of S in Step 2 is a copy-and-substitute operation which does not involve any reproving. Only the addition of new contracts makes it necessary to prove new facts about programs.

Given a sound proof repository for a program, it is possible to query in a simple manner whether a method implementation satisfies its contract:

Theorem 2 *Let S be a sound proof repository for a program $P = (\mathcal{C}, \mathcal{B})$, i a method implementation in a class of $\mathcal{C}$, and $\sigma \subseteq \mathcal{B}$ a set of bindings whose domain are method calls inside i. If $S \Downarrow i, \sigma = \emptyset$ then i satisfies its contract for any possible implementation of its called methods given by σ.*

The correctness of a complete subprogram can be checked by querying the status of each method implementation and bindings of its method calls.

Example 6 We build a sound proof repository S for the program developed in Ex. 2 step by step. S and P are initially empty.

1. Following Case 0 of Def. 9, we add class `Account` with field `balance` and without any method implementation so far.
2. Following Case 1 of Def. 9, we extend `Account` with method implementation $\texttt{deposit}_0$. Then we create a proof that $\texttt{deposit}_0$ satisfies its contract and insert the resulting proof obligations into S. A typical entry is $(\texttt{deposit}_0, \epsilon, \phi)$, where ϕ is a provable first-order formula without abstract contract symbols (because `deposit()` calls no other methods and the contract is obviously satisfied). This will entail query results such as $S \Downarrow \texttt{deposit}_0, \epsilon = \emptyset$.
3. Following Case 0 of Def. 9, we add class `Bank` with field `acc`.
4. Again, following Case 1 of Def. 9, we create a partial proof that $\texttt{update}_0$ satisfies its contract and insert the resulting subgoals into S. A typical entry is: $(\texttt{update}_0, \epsilon, \phi)$. Now several of the ϕ's will be unprovable and contain abstract symbols from the method call to `deposit()`. The query

$S \Downarrow \texttt{update}_0, \epsilon$ will return these entries, so we know that the contract of $\texttt{update}_0$ is not satisfied.[4]

5. Now we follow Case 2 of Def. 9 and $b =$ (line 12 in $\texttt{update}_0, \texttt{deposit}_0$) to $\mathcal{B}$. Looking for entries in S with the method implementation $\texttt{update}_0$ in the first component, we find entries of the form $(\texttt{update}_0, \epsilon, \phi)$. For each of these entries, we add a new entry of the form $(\texttt{update}_0, \{b\}, b(\phi))$. The resulting first-order subgoals turn out to be provable, because the contract of $\texttt{deposit}_0$ is sufficient to prove that the contract of $\texttt{update}_0$ is satisfied. Hence, $S \Downarrow \texttt{update}_0, \{b\} = \emptyset$.
6. We add field `fee` and then method implementation $\texttt{deposit}_1$ to class `Account`. Similar as in Step 2, new proof obligations are added to S.
7. We add $b' =$ (line 12 in $\texttt{update}_0, \texttt{deposit}_1$) to $\mathcal{B}$. Similar as in Step 5, new entries of the form $(\texttt{update}_0, \{b'\}, b'(\phi))$ are created. As $b' \sqsubset \{b\}$, only the entries containing ϵ in the second component are copied. Even these new entries are automatically first-order provable and no new verification effort is necessary.

Remark 3 It would be sound to delete all entries with first-order provable subgoals from the repository, because contract satisfaction queries ask for *unprovable* subgoals. In this case, the proof repository would be unchanged for Steps 2, 5, and 7 of Ex. 6. This could lead to substantially smaller repositories, but also preclude optimizations based on caching previous results. An obvious compromise would be to replace an entry (i, σ, ϕ) with a first-order provable constraint ϕ with (i, σ, true) to enable caching of first-order provable subgoals. To determine such trade-offs requires further implementation and experimentation, which is planned for future work.

In the evolution of programs, it is typically desirable to work towards correct and complete repositories. However, there could be (older or newer) method implementations that cause problems due to faults in code, contract, or usage. To cope with this in our framework we may remove problematic implementations and bindings, forming a subprogram of the given program. This will not destroy well-formedness nor soundness, and we may regain correctness of the subprogram. In order to obtain completeness, we may then add new implementations (of removed methods and additional ones) and add corresponding bindings, taking care to maintain correctness. This means that our framework supports replacement of methods in this way; allowing a quite flexible program evolution process. Since repositories record information for each implementation and each binding it is easy to form repositories of subprograms, as well as of extended programs.

[4] If i is the label of a method implementation that contains at least one method call, then $S \Downarrow i, \epsilon$ will always return a non-empty set. More generally, if i is the label of a method implementation and the domain of $\mathcal{B}$ does not contain all the method calls in i, then $S \Downarrow i, \mathcal{B}$ will always return a non-empty set.

5 Examples: Integration with Structuring Concepts

5.1 Class Inheritance and Behavioral Subtyping

We show how class inheritance and late binding can be integrated into the programming model presented in Sect. 2.1. With single inheritance one may build tree-like class hierarchies, where a class may have several (direct or indirect) subclasses and several (direct and indirect) superclasses, but at most one direct superclass, called the *parent* class. In general a subclass C' will extend an existing parent class C by introducing new fields and new method implementations, possibly including re-implementation of methods (i.e., implementation of methods with a name found in C or a superclass of C). The class inherits all fields and method implementation of its parent class. Inside the subclass C' a method name m refers to the method implementation of C' if any, otherwise that inherited from C. The syntax *super.m* refers to the method m of C (possibly inherited in C).

Objects of the new class C' will contain an instance of all fields declared or inherited. An object variable o declared of class C may at run-time refer to an object of class C or C' or any other subclass of C. Late binding means that the binding of a method call $o.m$ depends on the class of the object that o refers to at run-time. Similarly, a local call m is bound to that of the class of the executing object. For example the call `acc[i].deposit(interest)` in class `Bank` may bind to the deposit of class `Account` or a subclass `FeeAccount`. At verification time the actual binding of method calls cannot in general be decided, and the binding of a call $o.m$ is treated as non-deterministic, potentially binding to type-correct implementations of any m of the declared class of o (possibly inherited) or any type-correct re-implementation in a subclass. A call is classified as *static* if it contains **`super`**, otherwise *late-bound*.

In order to extend the framework of proof repositories, we let (the label of) each call $o.m$ be indexed by the declared class of o, and let each local call m be indexed by the name of the enclosing class, i.e., treating a local call m as $this.m$ where *this* refers to the current object. As above we may allow a class to define alternative implementations of the same method m, even though such alternatives are typically controlled by inheritance in object-oriented programs. In this case we must restrict bindings to those that result in type-correct calls considering the type of the actual parameters and that of the result value: A binding of a call $v = o.m(e_1, e_2, ..., e_n)$ to a method implemented in a class C is *type-correct* if C is the declared class of o, or a subclass, and the assignments $x_i = e_i$ (for each i) and $v = w$ are type-correct when x_i has the type of the ith formal parameter and variable w has the type of the declared method result.

The proof repository for a subclass C' with C as parent class is built according to definition 9, using case 1 for methods implemented or re-implemented in C', resulting in new partial proofs, and case 2 for re-implemented methods, such that for each late-bound call of m occurring in C' or a superclass of C' (i.e., indexed by C' or a superclass of C'), we add a binding associat-

```
1  class Account {
2    int balance = 0;
3
4  /*@ label    deposit0
5    @ def R  == x >= 0;
6    @ def LS == {balance};
7    @ def \def(balance) == \old(balance) + x;
8    @ def E  == true;
9    @*/
10   void deposit(int x) { ··· }
11
12 /*@ label    transfer0
13   @ def R  == x >= 0 ∧ x <= balance;
14   @ def LS == {balance};
15   @ def \def(balance) == \old(balance) - x;
16   @ def E  == true;
17   @*/
18  void transfer(Account a, int x) { ··· }
19 }
```

Fig. 6: The `Account` superclass

ing each (type-correct) re-implemented m with the label of the call. A static call $super.m_{C'}$ is bound to the (possibly inherited) implementation of m in C (which gives deterministic binding if C has only one type-correct method m).

Thus the framework for proof repositories is well suited for object-oriented inheritance and late binding. Also static binding by means of *super* can be accommodated. Since subclassing implies a kind of subtyping, the notion of type-correct program is specialized to cover (a version of) the standard contra/co-variance for methods signatures, allowing also multiple definitions of methods in a class without any type restrictions. The binding is type-safe since it exploits the type of the actual parameters and return value to select a subset of method definitions yielding well-formed and type-correct programs.

As mentioned it might happen in our framework that a program gives rise to unprovable proof obligations. Even if each class has provable proof obligations when seen in isolation, an unsatisfiable proof obligations may arise for instance when a method defined and used in a class is redefined in a subclass with a conflicting contract specification, and the usage of the method in the superclass depends on the original specification. To resolve the situation one can avoid the call in the superclass or modify the redefined method and its contract, and re-verify the resulting program. But this could give substantial re-verification. In our setting, a resolution can be found by means of a subprogram obtained by removing method definitions, and corresponding bindings, such that the subprogram has a subset of the original bindings and results in provable proof obligations. In this case one can reuse the (corresponding subset of) proof obligations, and re-verification is avoided (apart from applying the verification logic). The subprogram can then be augmented by new versions of problematic methods, and re-verification is limited to these additions.

Consider our example, where class `Account` in Fig. 6 has one `deposit` method and one `transfer` method. We here ignore the standard requires, en-

```
1  class FeeAccount extends Account {
2    final int fee = 2;
3
4  /*@ label    deposit0
5    @ def R  == x >= 0;
6    @ def LS == {balance};
7    @ def E  == balance >= \old(balance);
8    @*/
9    void deposit(int x) {  if (x >= fee) {balance += x - fee } }
10
11 /*@ label transfer0}
12 ... contract of transfer0 ... */
13   void transfer(Account a, int x) { ··· }
14
15 /*@ label    transfer1
16   @ def R  == x >= 0;
17   @ def LS == {balance};  // or super.assignable
18   @ def E  ==  balance <= \old(balance);
19   @*/
20   void transfer(FeeAccount a, int x) { ··· }
21 }
```

Fig. 7: The `FeeAccount` class extending the `Account` class

sures, and assignable clauses. In Fig. 7 we add a subclass `FeeAccount` which implements the extension of the bank account deposit with a fee (as in Fig. 3). The subclass adds a new implementation of `deposit`, which we label with `FeeAccount.deposit`$_0$. Given the proof repository S for classes `Bank` and `Account` of Ex. 6, adding the subclass is realized by two steps. First the new implementation gives a partial proof that is added. Second, the implementation `FeeAccount.deposit` of the subclass results in a new binding for the `deposit` call in the `update` method of the `Bank` class. Thus, a call to `deposit` in a `Bank` object may non-deterministically bind to `deposit`$_0$ in `Account` or `deposit`$_0$ in `FeeAccount`. Consider multiple definitions of methods where the subclass `FeeAccount` implements the extension of the bank account money transfer with a fee. `FeeAccount` has two implementations of `transfer`, one labeled `transfer`$_0$ with input parameter `a` of type `Account` and the other labeled `transfer`$_1$ with `a` of type `FeeAccount`. In this case a call `o.transfer(a)` in `Bank` gives the following possible bindings depending on the typing information: (i) If `o` evaluates to type `Account`, we have a deterministic binding to `transfer`$_0$ in class `Account`, (ii) if `o` evaluates to type `FeeAccount` and `a` evaluates to type `FeeAccount`, the call binds to `transfer`$_1$ in `FeeAccount` using the narrowest type, otherwise (iii) the call binds to `transfer`$_1$ in `FeeAccount`.

At the level of the proof system, the generation of proof obligations corresponds to lazy behavioral subtyping [12, 13], since bindings are added to the repository of an implementation only for method calls that occur in the implementation. In particular we do not insist that a redefined method satisfies the original contract of the superclass as in behavioral subtyping [26]. For instance in the example, the contract of `deposit`$_0$ in `Account` is not satisfied by the re-implemented `deposit` in `FeeAccount`.

```
1  /*@
2    @ delta def R == dr;
3    @ delta def LS == [\original(LS) - {l'_1,...,l'_p} +] {l''_1,...,l''_q};
4    @ delta def \def(l'''_1) == de_1, ..., \def(l'''_m) == de_m;
5    @ delta def E == de;
6    @*/
```

Fig. 8: Shape of a *delta abstract method contract* (in line 3, the part enclosed in square brackets is optional, and the symbols "`-`" and "`+`" denote set-theoretic difference and union, respectively)

5.2 Delta-oriented Programming

Delta-oriented programming (DOP) [7, 31] is a transformational approach for Software Product Line (SPL) development [32]. It supports developing an SPL by starting from at least one complete product, called the *core product*. To this core product one applies program transformations (the delta modules) that specify changes to implement other products. The alterations inside a delta module act both at the class level, by adding or removing classes, and at the class structure level by modifying the internal structure of existing classes (i.e., changing the super class and adding, removing, or modifying fields, and methods). Modifying a method means either replacing the method body or wrapping the existing body using the `original` construct. The call `original(...)` calls a method with the same name as before the modifications, and is bound when the product is generated. This call may only occur in the body of the method provided by a method-modify operation. A method-modify operation that uses the `original` construct adds a new method with a fresh name that is used (instead of `original`) in the body of the modified method in the generated product—the name of the new method is denoted by `m`δ, where `m` is the name of the modified method and δ is the name of the delta module that contains the method-modify operation.

A delta module may not only change the code of a program, but also its specification. In the setting of this paper, a delta on a method contract may replace the requires or ensures clause or modify it by referring to the previous version of the respective clause using the original construct. These modifications can be expressed by chaining the definition section of an abstract method contract. A *delta abstract method contract* describes how to modify the definition section of an abstract method contract following the delta (the abstract section, which is the same for each method, cannot be modified). It has the shape shown in Fig. 8, where:

- any of the `delta def` clauses can be omitted (meaning that the corresponding `def` clause in the contract to be modified is unchanged),
- `dr`, `de` may contain occurrences of `\original(R)`, `\original(E)`, resp.
- if `\original(LS)`$==\{l_1,\ldots,l_n\}$, then
  - $\{l'_1,\ldots,l'_p\} \subseteq \{l_1,\ldots,l_n\}$ and $\{l''_1,\ldots,l''_q\} \cap \{l_1,\ldots,l_n\} = \emptyset$,
  - $\{l'''_1,\ldots,l'''_m\} \subseteq (\{l_1,\ldots,l_n\} - \{l'_1,\ldots,l'_p\}) \cup \{l''_1,\ldots,l''_q\}$,

– for all $\mathtt{l}'''_i \in \{\mathtt{l}_1,\ldots,\mathtt{l}_n\} - \{\mathtt{l}'_1,\ldots,\mathtt{l}'_p\}$, the term $\mathtt{de}_i$ may contain occurrences of `\original(\def(`$\mathtt{l}_i$`))`.

We consider the DOP scenario of SPLs of JAVA programs described in [7], where method overloading is not used.[5] Therefore each delta module contains at most one class addition/modification/removal clause for each class name and, inside a class clause, at most one method addition/modification/removal clause for each method name. Whenever a delta module is applied to a (possibly incomplete) program a possibly incomplete program is generated by updating the bindings according to the JAVA binding strategy. We adopt the following convention for the labels of the methods:

– The label of each method added or modified by a delta module contains the name of the delta module.
– The label of each method occurring in a product (or intermediate program) contains the ordered sequence of the names of the delta modules that have been applied to generate the product and that effectively contributed to the generation of the code of the method. In particular, if a method `m` has been added to a product (or intermediate program) $\mathcal{P}$ by a delta module δ and has not been affected by subsequent delta modules used for generating $\mathcal{P}$, then the label of `m` in $\mathcal{P}$ is the same as in δ.

Example 7 Consider a simple product line, that we call the Bank PL. The Bank PL has two features, Base (mandatory) and Fee (optional), and two products:

– A core product p_1 corresponding to the feature configuration {Base}, which provides the basic functionalities described in steps 1–3 of Ex. 1.
– Another product p_2, corresponding to the feature configuration {Base, Fee}, which additionally charges each deposit with a fee.

The code base contains:

– The core product, consisting of the class `Bank` in Fig. 1, the class `Account` in Fig. 2, and the binding from the method call `deposit(interest)` in line 12 of Fig. 1 to $\mathtt{deposit}_0$.
– The delta module `DFee`, in Fig. 9, which modifies the class `Account` to implement the feature Fee and illustrates the concept of a delta abstract method contract.

The program for product p_2 consists of the classes illustrated in Fig. 10 and of the bindings

– from the method call `deposit(interest)` (line 12, Fig. 1) to $\mathtt{deposit}_{DFee}$, and
– from the method call `deposit$DFee(x-fee)` (line 27, Fig. 10) to $\mathtt{deposit}_0$.

It is obtained by applying the delta module `DFee` to the core product.

[5] This is not a restriction since, in JAVA, method overloading is resolved statically.

```
1  delta DFee {
2    modifies class Account {
3      adds final int fee = 2;
4
5    /*@ label deposit_DFee
6      @ delta def \def(balance) ==
7      @   (x >= fee) ? \original(\def(\balance)) - fee : \old(balance);
8      @*/
9      modifies void deposit(int x) {
10        if (x >= fee) { original(x - fee) }
11 }  }  }
```

Fig. 9: The DFee delta module providing the functionality for the Fee feature (line 7 uses the JAVA syntax for conditional expressions, "... ? ... : ...")

```
1    class Bank {
2      // body of class Bank is the same as in the core product
3    }
4
5    class Account {
6      int balance = 0;
7      final int fee = 2;
8
9    /*@ label    deposit_0
10     @ def R == x > 0;
11     @ def LS == {balance};
12     @ def \def(balance) == \old(getBalance()) + x;
13     @ def E == true;
14     @*/
15     void deposit$DBase(int x) { balance += x }
16
17
18   /*@ label    deposit_DFee
19     @ def R == x > 0;
20     @ def LS == {balance};
21     @ delta def \def(balance) == (x >= \old(getFee())) ?
22     @                 (\old(getBalance()) + x) - fee : \old(getBalance());
23     @ def E == true;
24     @*/
25
26     void deposit(int x) {
27       if (x >= fee) { deposit$DBase(x-fee) }
28     }
29   }
```

Fig. 10: The product with features Base and Fee

In order to verify an SPL by exploiting our proof repositories we identify each method name of the form `m`$\$\delta$ with the name `m` and we do not replace the occurrences of `m` in the body of the methods introduced by the method modify operations in the delta modules (i.e., the method renaming introduced, during product generation, for dealing with the `original` construct is ignored). So, the explicit binding management mechanism can be exploited to maximize verification proof reuse across the products of the SPL.

The *declaration* of deltas will add implementations of methods with potentially changed contracts with partial proofs. In addition to abstract symbols the resulting proof obligations also contain unresolved `original` references. The *application* of deltas to a program and its proof repository in order to generate a program variant results in new bindings, whereby abstract symbols are replaced and original calls are bound. This also includes the original constructs used in the definition part of the abstract method contracts. Those are bound to the parts of the contracts belonging to the respectively bound methods.

To summarize, for a core product and a set of delta modules changing code and contracts, we construct the following:

- a program $(\mathcal{C}, \mathcal{B})$, that we call the *family program*, such that
  - each class in $\mathcal{C}$ contains all the method implementations that occur in at least one product—there are the method implementations and corresponding contracts of the core product or the method implementations and corresponding contracts that are added or modified by a delta module;
  - the set of method call bindings in $\mathcal{C}$ contains all the bindings in at least one product; and
- a sound proof repository for $(\mathcal{C}, \mathcal{B})$.

Note that, if all the products of the SPL are well-formed programs then the family program $(\mathcal{C}, \mathcal{B})$ is well-formed.

The family program and the associated proof repository can be built incrementally, by iterating over the set of valid feature configurations, as follows.

1. The program (classes and bindings) representing the core product (product p_1, for the Bank PL example) are added.
2. For each other valid feature configuration (product p_2, for the Bank PL example): the methods introduced (either by an add or by a modify operation) by each delta module associated to the configuration are added (if they are not already present)[6] and the associated bindings are added together with the (partial) proofs.

Example 8 The family program for the Bank PL of Ex. 7 consists of the classes illustrated in Fig. 11 and of the bindings

- from the method call `deposit(interest)` (line 12, Fig. 1) to $\texttt{deposit}_0$ (as in product p_1),
- from the method call `deposit(interest)` (line 12, Fig. 1) to $\texttt{deposit}_{DFee}$ (as in product p_2), and
- from the method call `original(x-fee)` (line 27, Fig. 11) to $\texttt{deposit}_0$ (as in product p_2).

[6] This can be checked straightforwardly by comparing the labels.

```
1   class Bank {
2     // body of class Bank is the same as in the core product
3   }
4
5   class Account {
6     int balance = 0;
7     final int fee = 2;
8
9   /*@ label    deposit_0
10    @ def R == x > 0;
11    @ def LS == {balance};
12    @ def \def(balance) == \old(getBalance()) + x;
13    @ def E == true;
14    @*/
15    void deposit(int x) { balance += x }
16
17
18  /*@ label    deposit_ODFee
19    @ def R == x > 0;
20    @ def LS == {balance};
21    @ delta def \def(balance) == (x >= \old(getFee())) ?
22    @               (\old(getBalance()) + x) - fee : \old(getBalance());
23    @ def E == true;
24    @*/
25
26    void deposit(int x) {
27      if (x >= fee) { original(x-fee) }
28    }
29  }
```

Fig. 11: The family program for the Bank PL

6 Initial Experiments

We used a modified version of the KeY verification system for Java with support for abstract contracts to emulate our proof repository approach. We did the experiment along the lines of the running example and report here our preliminary findings. The proof effort for the experiment outlined below is summarized in Table 1.

We started to populate the proof repository S by adding class `Account` including its field `balance` and added then method `deposit(int)` (without fees). This first step created a proof repository with proof obligations that required to verify the correctness of `deposit(int)` w.r.t. its own contract. This resulted in six first-order proof obligations with placeholders from its own abstract contract only. In other words, the resulting proof repository contained only entries with empty binding sets, because no method is called from within `deposit(int)`. Two of the six proof obligations are immediately closeable without expanding the placeholder definitions of the abstract contracts. The remaining four first-order proof obligations are provable by expanding the placeholders, hence, the proof repository did not contain any unprovable proof obligations for method `deposit(int)` (i.e., $S \Downarrow \texttt{deposit}_0, \epsilon = \emptyset$).

`Account deposit`	w/o fee (`deposit`$_0$)			with fee (`deposit`$_1$)		
	po (abstract)	po (own)	po (bindings)	po (own)	po (bindings)	
po open	4 (of 6)	0	—	0	—	
size	92	9	—	37	—	**Σ : 138**

`Bank update`					
	po (abstract)	po (own)	po (`deposit`$_0$)	po (`deposit`$_1$)	
po open	21 (of 29)	8	0	0	
size	679	930	1195	998	**Σ : 2965**

(size is measured in length of proof derivation and an indicator of the required proof effort)

Table 1: Proof Effort

The program was then extended by adding class `Bank` and its method `update()`. Method `update()` invokes `deposit(int)` (inside the loop) and causes for the first time in our scenario the addition of method bindings into our proof repository. Using abstract contracts 29 proof obligations were generated of which 21 were not first-order provable. Expanding the definitions of the abstract contract of method `update` only eight unprovable first-order proof obligations (PO_1) were left containing the abstract symbols for the method contract of `deposit(int)`. Interestingly the open proof obligations were all concerned with the verification that the loop invariant is preserved. They were not necessary for proving the initially valid case or use case (loop invariant is used to prove the actual method contract) part of the loop invariant calculus rule.

Adding entries for method binding `deposit(int)` and instantiating the eight open proof obligations for the specific contracts, all first-order obligations could be closed (i.e., $S \Downarrow \texttt{update}_0, \{\text{call to } \texttt{deposit}_0\} = \emptyset$).

Two variants for the introduction of an account with fees were simulated: The first version was added as a solution where both kinds of accounts could be present in a system at the same time. This solution uses subclassing and makes use of the flexibility of our approach, where behavioral subtyping is not a prerequisite. This version requires only the verification of the method contract for `deposit(int)` of the new subclass of `Account` and the re-verification of the eight proof obligations PO_1 where the abstract symbols are instantiated with the new method contract (`deposit`$_1$). The second variant simulated software evolution, where the method implementation of the original class changes and is extended to support fees. In this scenario it turns out that the verification task is identical with the first one and both give rise to the same proof obligations that need to be re-verified.

Table 1 measures the proof effort in terms of length of the derivation (more precise: number of rule applications). The total effort is the sum of all proof sizes. How does our proposed approach compare with a traditional approach based on behavioral subtyping? In our scenario the traditional approach re-

quires a total proof effort of 4846 for the subclassing scenario (sum of derivation length of the correctness proofs for all methods including those required to ensure that subclasses satisfy the contract of their superclass) and 9292 for the evolution scenario (basically no reuse possible, all proofs had to be redone). In summary, our approach (total effort: $3103 = 138 + 2965$) saves us 36% resp. 66% of verification work. In the current stage, our prototype does not yet include optimizations like identification of identical first-order proof obligations. For instance, in case of the `Bank` class, the proof effort could be reduced further by ca. 200 by identifying that the addition of a new binding to $\texttt{deposit}_1$ does not require to reprove that (i) the precondition of `deposit` is established (identical first-order proof obligations to the present proof for the precondition of $\texttt{deposit}_0$), (ii) the loop terminates and that (iii) the assignable clause of `update` is correct.

Please note also that in case of the subclassing scenario, the numbers do not reflect the refactoring effort that is necessary to achieve a code base compatible with Liskov's substitution principle. This means, the introduction of a suitable abstract class and an abstract `deposit(int)` method with a contract strong enough to verify method `update()` of class `Bank`.

A more thorough and general comparison of our approach is future work. It will require more profound changes to the used verification system, which on the specification level is based on JML, and hence, behavioral subtyping by specification inheritance. In addition, a fair comparison should also measure the flexibility of different approaches and their applicability to a variety of structuring paradigms—an area where our proposed solution seems to be promising.

7 Related Work

Proof reuse was studied in [5, 29] where proof replay is proposed to reduce the verification effort. The old proof is replayed and when this is no longer possible, a new proof rule is chosen heuristically. The proof reuse focuses only on the proof structure and does not take the specification into account like our work. In [8], it is assumed that one program variant has been fully verified. By analyzing the differences to another program variant, one obtains those proof obligations that remain valid in the new product variant and that need not be reestablished. In [20], evolving formal specifications are maintained by representing the dependencies between formal specifications and proofs in a development graph. The effect of each modification is computed so that only invalidated proofs have to be re-done. In [33], proofs are evolved together with formal specifications. The main limitation is that the composition of proofs is not aligned to the composition of programs as in our framework. The paper [25] studies fine-grained caching of verification conditions that arise during a proof. It is optimized for a highly interactive scenario where each keystroke of the user in an IDE potentially leads to new verification tasks. The labels that identify method implementations in our framework are called "checksums"

there. Their approach is specific to call structures of the Boogie language. It uses a simple abstraction mechanism called "assumption variables" that can be seen as conditional requires clauses.

Previous work by the authors explored proof systems with an explicit proof environment (similar to a typing environment) to improve flexibility and reuse for the open world assumption in the context of class inheritance [12, 13], traits [10], and (dynamic) software updates [14, 15]. One exploits the binding mechanism of a particular code structuring concept to define the proof environment and formalizes the proof environment as a cache for the software verification process. In the context of the proof repositories presented in the current paper, such approaches can still be made use of by mapping them into partial proofs and bindings (details are the subject of future work). This line of work, however, assumes that at least a part of the program beyond the module under analysis is known in order to perform the analysis; e.g., the superclasses, the subtraits, etc. However, even this assumption does not hold in the general case of program evolution and variability, such as DOP. To address this challenge, the authors proposed a proof system which transforms placeholders for assertions for software product lines [11], and explored abstract contracts as a mechanism for verification reuse [9, 19]. Our proof repositories generalize and make use of this work while being compatible with such refinements as mentioned above.

8 Conclusion

To increase the applicability of deductive software verification, ongoing efforts on improving verification-in-the-small need to be complemented by better integration in software development processes. The underlying assumptions about module composition and development in our verification systems must be aligned with those of the development processes. For this reason, it is important to investigate more flexible approaches to compositionality in software verification.

This paper has proposed a novel, systematic framework for verification reuse which makes use of abstract method contracts to realize an incremental proof repository aimed for verification-in-the-large. Abstract method contracts provide a separation of concerns between the usage of a method and its (changing) implementations. The proof repository keeps track of the verification effort in terms of abstract proofs which can be reused and completed later. The approach is meaningful for partial programs, so it allows the developer to start the verification effort while the program is being developed. We believe the approach can be combined with many software structuring concepts. To support this claim, we showed how to realize behavioral subtyping for class inheritance as well as delta-oriented variability for software product lines.

References

1. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3**(6), 27–56 (2004)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: G. Barthe, L. Burdy, M. Huisman, J.L. Lanet, T. Muntean (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004), *LNCS*, vol. 3362, pp. 49–69. Springer-Verlag, New York, NY (2005)
3. Beckert, B., Hähnle, R.: Reasoning and verification. IEEE Intelligent Systems **29**(1), 20–29 (2014)
4. Beckert, B., Hähnle, R., Schmitt, P. (eds.): Verification of Object-Oriented Software: The KeY Approach, *LNCS*, vol. 4334. Springer-Verlag (2007)
5. Beckert, B., Klebanov, V.: Proof reuse for deductive program verification. In: Third IEEE International Conference on Software Engineering and Formal Methods, pp. 77–86. IEEE Computer Society (2004). DOI http://doi.ieeecomputersociety.org/10.1109/SEFM.2004.10013
6. Beckert, B., Schmitt, P.H.: Program verification using change information. In: Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia, pp. 91–99. IEEE Press (2003)
7. Bettini, L., Damiani, F., Schaefer, I.: Compositional type checking of delta-oriented software product lines. Acta Informatica **50**(2), 77–122 (2013). DOI 10.1007/s00236-012-0173-z. URL `http://dx.doi.org/10.1007/s00236-012-0173-z`
8. Bruns, D., Klebanov, V., Schaefer, I.: Verification of software product lines with delta-oriented slicing. In: B. Beckert, C. Marché (eds.) International Conference on Formel Verification of Object-oriented Software (FoVeOOS 2010), Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 6528, pp. 61–75. Springer-Verlag (2011)
9. Bubel, R., Hähnle, R., Pelevina, M.: Fully abstract operation contracts. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation, 6th International Symposium, ISoLA 2014, Corfu, Greece, *Lecture Notes in Computer Science*, vol. 8803, pp. 120–134. Springer-Verlag (2014)
10. Damiani, F., Dovland, J., Johnsen, E.B., Schaefer, I.: Verifying traits: an incremental proof system for fine-grained reuse. Formal Aspects of Computing **26**(4), 761–793 (2014)
11. Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E.B., Yu, I.C.: A transformational proof system for delta-oriented programming. In: Proc. 16th Intl. Software Product Line Conference (SPLC), Volume 2, pp. 53–60. ACM (2012)
12. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Lazy behavioral subtyping. Journal of Logic and Algebraic Programming **79**(7), 578–607 (2010)
13. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Incremental reasoning with lazy behavioral subtyping for multiple inheritance. Science of Computer Programming **76**(10), 915–941 (2011)
14. Dovland, J., Johnsen, E.B., Owe, O., Yu, I.C.: A proof system for adaptable class hierarchies. Journal of Logical and Algebraic Methods in Programming **84**(1), 37–53 (2015)
15. Dovland, J., Johnsen, E.B., Yu, I.C.: Tracking behavioral constraints during object-oriented software evolution. In: T. Margaria, B. Steffen (eds.) 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change (ISoLA 2012), *Lecture Notes in Computer Science*, vol. 7609, pp. 253–268. Springer-Verlag (2012)
16. Engel, C., Roth, A., Schmitt, P.H., Weiß, B.: Verification of Modifies Clauses in Dynamic Logic with Non-rigid Functions. Tech. Rep. 2009-9, Department of Computer Science, University of Karlsruhe (2009)
17. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: B. Beckert, C. Marché (eds.) International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010), Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 6528, pp. 10–30. Springer-Verlag (2011)
18. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: W. Damm, H. Hermanns (eds.) Computer Aided Verification,

19th International Conference, Berlin, Germany, *Lecture Notes in Computer Science*, vol. 4590, pp. 173–177. Springer-Verlag (2007)
19. Hähnle, R., Schaefer, I., Bubel, R.: Reuse in software verification by abstract method calls. In: M.P. Bonacina (ed.) Proc. 24th Conference on Automated Deduction (CADE), Lake Placid, USA, *Lecture Notes in Computer Science*, vol. 7898, pp. 300–314. Springer-Verlag (2013)
20. Hutter, D., Autexier, S.: Formal Software Development in MAYA. In: Mechanizing Mathematical Reasoning (2005)
21. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: M.G. Bobaru, K. Havelund, G.J. Holzmann, R. Joshi (eds.) NASA Formal Methods, Third International Symposium, Pasadena, CA, USA, *Lecture Notes in Computer Science*, vol. 6617, pp. 41–55. Springer-Verlag (2011)
22. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: J. Misra, T. Nipkow, E. Sekerinski (eds.) 14th International Symposium on Formal Methods (FM 2006), *Lecture Notes in Computer Science*, vol. 4085, pp. 268–283. Springer-Verlag (2006)
23. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M.: JML Reference Manual (2009). URL `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmlrefman.pdf`. Draft revision 1.235
24. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: E.M. Clarke, A. Voronkov (eds.) 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16), *Lecture Notes in Computer Science*, vol. 6355, pp. 348–370. Springer-Verlag (2010)
25. Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: D. Kroening, C.S. Pasareanu (eds.) Computer Aided Verification, 27th Intl. Conf. CAV, San Francisco, CA, USA, Part I, *LNCS*, vol. 9206, pp. 380–397. Springer (2015)
26. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems **16**(6), 1811–1841 (1994)
27. Meyer, B.: Applying "design by contract". IEEE Computer **25**(10), 40–51 (1992)
28. Müller, P., Shankar, N., Leavens, G.T., Ridge, T., Tuerk, T., Klebanov, V., Ulbrich, M., Weiß, B., Leino, K.R.M., Chapman, R., Monahan, R., Polikarpova, N., Bronish, D., Arthan, R., Alkassar, E., Cohen, E., Hillebrand, M., Tobies, S., Jacobs, B., Piessens, F., Smans, J.: The 1st verified software competition: Extended experience report. In: Formal Methods (FM'11), *Lecture Notes in Computer Science*, vol. 6664. Springer-Verlag (2011)
29. Reif, W., Stenzel, K.: Reuse of proofs in software verification. In: Foundations of Software Technology and Theoretical Computer Science, pp. 284–293 (1993)
30. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 55–74. IEEE Computer Society (2002)
31. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Proc. of SPLC 2010, *Lecture Notes in Computer Science*, vol. 6287, pp. 77–91. Springer-Verlag (2010)
32. Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K.: Software diversity: state of the art and perspectives. International Journal on Software Tools for Technology Transfer **14**(5), 477–495 (2012). DOI 10.1007/s10009-012-0253-y. URL `http://dx.doi.org/10.1007/s10009-012-0253-y`
33. Schairer, A., Hutter, D.: Proof transformations for evolutionary formal software development. In: H. Kirchner, C. Ringeissen (eds.) Algebraic Methodology and Software Technology, *Lecture Notes in Computer Science*, vol. 2422, pp. 441–456. Springer-Verlag (2002). DOI 10.1007/3-540-45719-4_30. URL `http://dx.doi.org/10.1007/3-540-45719-4_30`
34. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic frames in Java dynamic logic. In: B. Beckert, C. Marché (eds.) International Conference on Formel Verification of Object-oriented Software (FoVeOOS 2010), Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 6528, pp. 138–152. Springer-Verlag (2011)