Fully Abstract Operation Contracts*

Richard Bubel, Reiner Hähnle, and Maria Pelevina

Department of Computer Science
Technical University of Darmstadt
bubel|haehnle@cs.tu-darmstadt.de · m.pelevina@gmail.com

Abstract. Proof reuse in formal software verification is crucial in presence of constant evolutionary changes to the verification target. Contract-based verification makes it possible to verify large programs, because each method in a program can be verified against its contract separately. A small change to some contract, however, invalidates all proofs that rely on it, which makes reuse difficult. We introduce fully abstract contracts and class invariants which permit to completely decouple reasoning about programs from the applicability check of contracts. We implemented tool support for abstract contracts as part of the KeY verification system and empirically show the considerable reuse potential of our approach.

1 Introduction

A major problem in deductive verification of software is to keep up with changes of the specification and implementation of the target code. Such changes are inevitable, simply because of bug fixes, but also because in industrial practice requirements are constantly evolving. The situation is exacerbated when, as it is often the case, software systems possess a high degree of variability (e.g., in product line-based development).

To redo most proofs and, in the process of doing this, to re-examine most specifications is expensive and slow, even in the ideal case when no user interaction with the verification tool is required. Therefore, a research question in formal verification that is of the utmost practical relevance is how to minimize the amount of verification effort necessary after changes in the underlying specification and implementation of the target code.

Like most state-of-art formal verification approaches, we assume to work in a *contract-based* [1] setting, where the *granularity* of specification units is at the level of one method. The most important advantage of method contracts is the following notion of *compositionality*: assume a program consists of methods $m \in M$, each specified with a contract C_m. Now we prove that each method m satisfies its contract. During this proof, calls inside m to other methods n are handled by appying their contract C_n instead of inlining the code of n. This works fine as long as there are no recursive method calls.[1] This contract-based

* This work has been partially supported by EC FP7 Project No. 610582 Envisage

[1] There are ways to extend this methodology to recursive method calls, but to keep the presentation simple, we explicitly exclude recursion in this paper. It is an issue that is orthogonal to the techniques discussed here.

verification approach is implemented in many state-of-art verification systems, including KeY [2].

In the context of contract-based verification, the problem of keeping up with target code changes can be formulated 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 C_m in general is no longer valid. Therefore, C_m cannot be used in the proof of p which is accordingly broken and must be redone with the new contract of m. If m occurs in many places in p, this becomes very expensive.

In previous work [3] we presented a novel approach to mitigate the problem by introducing the notion of an *abstract contract*. Abstract contracts do not provide concrete pre- or postconditions, but only placeholders for them. At the same time, abstract contracts can be used in a verification system exactly in the same manner as standard concrete contracts. Even though the proofs resulting from abstract contracts can (most of the times) not be closed, we still end up with partial proofs whose open goals are pure first-order. These open proofs (or just the open goals) can be cached, reused, and then verified for different instantiations for the placeholders. In the present paper we make three contributions:

1. A crucial part of each method contract is its *assignable clause*, a list of references to memory locations whose value might be affected by the execution of that method. Obviously, the assignable clause is necessary for sound usage of contracts. In [3] we imposed the restriction that assignable clauses could not be abstract, they had to be explicitly given. This is a serious restriction, because it assumes that the changeable locations of a method cannot increase when its specification is changed. In the present paper we lift this restriction and, therefore, arrive at a notion of *fully abstract* method contracts.
2. The specification of a software system usually consists not only of contracts, but also of class invariants.[2] We extended our approach to cover *abstract invariants*.
3. The evaluation done in [3] was limited by the fact that we had no native implementation of our approach, but had to simulate the effects with suitably hand-crafted inputs. Now we have native tool support, integrated into the KeY verification system for the full Java fragment supported by KeY. Specifically, we evaluate our approach with two case studies. One of them was taken unaltered from a different project [4] to demonstrate that our approach works well in situations not "tailored" to it.

The paper is structured as follows: Section 2 introduces the necessary notions and theoretical background. Section 3 introduces fully abstract contracts and explains their usage. We evaluate our approach with two case-studies in Section 4 and conclude the paper with a discussion about related work (Section 5) and a conclusion (Section 6).

[2] Invariants can be simulated by contracts, but specifications become much more succinct with them, so it is well worth to support them directly.

2 Background

Here we present the background required to understand our specification and verification approach. Our programming language is sequential Java without floats, garbage collection or dynamic class loading. The specification approach follows the well-known design-by-contract paradigm [1].

Example 1. Our running example is a software fragment that computes whether a student passed a course. The concrete passing criteria give rise to different program versions. The basic version, where a student passes when she has passed the exam and all labs, is shown in Fig. 1.

The program consists of a single class `StudentRecord` which implements two methods. Method `passed()` is the main method determining whether the student has passed. It contains a loop to determine whether all labs were finished successfully and it invokes method `computeGrade()` to access the exam grade.

```
class StudentRecord {
  int exam; // exam result
  int passingGrade; // minimum grade necessary to pass the exam
  boolean[] labs = new boolean[10];  // labs

  //@ public invariant exam >= 0 && passingGrade >= 0 && ;
  //@ public invariant labs.length == 10;

  /*@ public normal_behavior
    @ requires true;
    @ ensures \result == exam;
    @ assignable \nothing; @*/
  int computeGrade(){ return exam; }

  /*@ public normal_behavior
    @ ensures \result ==> exam >= passingGrade;
    @ ensures \result ==> (\forall int x; 0 <= x && x < 10; labs[x]);
    @ assignable \nothing; @*/
  boolean passed() {
    boolean enoughPoints = computeGrade() >= passingGrade;
    boolean allLabsDone = true;
    for (int i = 0; i < 10; i++) {
        allLabsDone = allLabsDone && labs[i];
    }
    return enoughPoints && allLabsDone;
  }
}
```

Fig. 1. Implementation and specification of class `StudentRecord` in its basic version

As specification language we use the Java Modeling Language (JML) [5]. JML allows one to specify instance invariants and method contracts. Instance invariants express properties about objects which are established by the constructor and must be preserved throughout the lifetime of an object. More precisely, if an instance invariant holds at the beginning of a method, then it must hold again when it returns, even though it may be violated during its execution.

Example 2. In JML invariants are specified following the keyword **`invariant`** and a boolean expression. The class `StudentRecord` has two invariants on line 6–7. The first specifies lower bounds for the grade of an exam and the minimum grade required to pass the exam. The second invariant fixes the number of labs to ten. In addition, there are implicit invariants, because JML has a "non-null by default" semantics: for each field with a reference type there is an invariant specifying that it is non-null. All invariants are implicitly conjunctively connected into a single instance invariant (per class).

JML method contracts are specified as comments that appear just before the method they relate to. A JML method contract starts with the declaration of a specification case such as **`normal_behavior`** or **`exceptional_behavior`**. For ease of presentation we consider only method contracts specifying the normal behaviour, i.e., normal termination of the method without throwing an uncaught exception. The generalisation to exceptional behaviour specification cases is straightforward. We introduce JML method contracts by way of example:

Example 3. The contract for method `computeGrade()` has one normal behaviour specification case which consists of a precondition (keyword **`requires`**) and a postcondition (keyword **`ensures`**). The actual condition is a boolean expression following the keyword. JML specification expressions are boolean Java expression plus quantifiers and a few extra operators. For instance, in postconditions it is possible to access the return value of the method using the keyword **`\result`** or to refer to the value of an expression `e` in the prestate of the method by **`\old(e)`**.

Also part of the specification case definition is the **`assignable`** clause (also called modifies clause) which determines the set of locations that may be changed *at most* by the method to which the contract applies.

In the contract of `computeGrade()` the precondition is simply **`true`** which is the default and can be omitted. By default the invariant for the callee object (i.e., the object referred to by **`this`**) is implicitly added as a precondition. The postcondition states that the return value of the method equals the value of the field containing the exam grade and the assignable clause specifies the empty set of locations which expresses that the method is not allowed to modify any object field or array element. A method may have several specification cases of the same kind with the obvious semantics.

Definition 1 (Program location). *A* program location *is a pair* (o, f) *consisting of an object* o *and a field* f*.*

A program location is an access point to a memory location that a program might change. Next we formalize method contracts and invariants:

Definition 2 (Method contracts, invariants). *Let* `B` *denote a Java class. An* (instance) invariant *for* `B` *is a formula* $inv_{\texttt{B}}(\texttt{v\_this}, \texttt{v\_heap})$ *where* `v_this`, `v_heap` *are program variables that refer to the current* **`this`** *object and the heap under which the invariant is evaluated.*

Let $T\ \texttt{m}(S_1\ a_1, \ldots, S_n\ a_n)$ *be a method with return type* T *and parameters* a_i *of type* S_i*. A* method contract

$$C_{\texttt{m}} = (pre, post, mod)$$

for `m` *consists of*

- *a formula pre specifying the method's precondition;*
- *a formula post specifying the method's postcondition;*
- *an assignable clause mod which is a list of terms specifying the set of program locations that might be changed by method* `m`*.*

Each constituent of a contract may refer to method parameters $\bar{a} = (a_1, \ldots, a_n)$*, the callee using the program variable* `v_this`*, and the current Java heap using the program variable* `v_heap`*. In addition, the postcondition can refer to the result value of a method using the program variable* `result` *as well and to the value of a term/formula in the methods prestate by enclosing it using the transformer function old.*

When verifying whether a method satisfies its contract, we ensure that, if a method is invoked in a state where the invariant of the **`this`** object as well as the method's precondition holds, then in the final state after the method execution, both, its postcondition and the instance invariant are satisfied. Stated as a Hoare triple [6] it looks as follows:

$$\{inv \wedge pre\}\ \ o.\texttt{m}(\bar{a})\ \ \{inv \wedge post\}$$

If the method under verification invokes another method `n` of class `B` with contract $(pre, post, mod)$, then, during the deductive verification process, we reach the point where we need to use `n`'s contract and we need to apply the corresponding operation contract rule:

$$\mathsf{useMethodContract}_{\texttt{C}_\texttt{n}}\ \frac{\begin{array}{l}\vdash inv(heap, o) \wedge pre(heap, o, \bar{a}) \\ \vdash P(heap) \wedge post(U(heap), heap, o, \bar{a}) \rightarrow Q(U(heap))\end{array}}{\{P(heap)\}\ \ o.\texttt{n}(\bar{a})\ \ \{Q(heap)\}}\ ,$$

where $U(heap) := U(mod)(heap)$ is a transformer that rewrites the heap representation such that all knowledge about the values of the program locations contained in *mod* is removed and everything else is unchanged.

The method contract rule generates two proof obligations: the first ensures that the precondition of `n` and the invariant of o hold. The second checks whether the information in the poststate is sufficient to prove the desired property.

A few words on practical issues concerning the transformer U. Such a function can be realized in different ways, depending on the underlying program logic.

In our implementation we use an explicit heap model which is formalized as a theory of arrays. The transformer U is then realized with the help of a function $anon(h1, mod, h2)$ which is axiomatized to return a heap that coincides with $h1$ for all program locations not in mod and with $h2$ otherwise. The transformer then introduces a new skolem constant h_{sk} of the heap datatype and returns the new heap $anon(heap, mod, h_{sk})$.

3 Abstract Operation Contracts

Abstract operation contracts have been introduced in [3] for the modeling language ABS [7] by some of the co-authors of this paper. The original version required the assignable clause to be concrete. This section presents a fully abstract version of operation contracts for the Java language including an abstract assignable clause and abstract instance invariants. We introduce first the notion of *placeholders* which are declared and used by abstract operation contracts:

Definition 3 (Placeholder). *Let* `B` *be a class and let* T $\texttt{m}(T_1\ p_1, \ldots, T_n\ p_n)$ *denote a method declared in* `B`*. A* placeholder *is an uninterpreted predicate or function symbol of one of the following four types:*

Requires placeholder *is a predicate* $R(Heap, \texttt{B}, T_1, \ldots, T_n)$ *which depends on the heap at invocation time, the callee (the object represented by* `this` *in* `m`*), and the method arguments.*

Ensures placeholder *is a predicate* $E(Heap, Heap, \texttt{B}, T, T_1, \ldots, T_n)$ *which depends on the heap in the method's final state, the heap at invocation time (to be able to refer to old values), the callee, the result value of* `m`*, and the method arguments.*

Assignable placeholder *is a function* $A(Heap, \texttt{B}, T_1, \ldots, T_n)$ *with return type LocSet representing a set of locations; the set is dynamic and may depend on the heap at invocation time, the callee and the method arguments.*

(Instance) invariant placeholder *is a predicate* $I(Heap,$ `B`$)$ *which may depend on the current heap and the instance (the* `this` *object) for which the invariant should hold.*

Placeholders allow to delay the actual instantiation of a contract or invariant. We extend the notion of invariant and contract from the previous section by introducing placeholders for the various constituents.

Definition 4 (Extended invariant). *An* extended instance invariant $Inv_{\texttt{B}} := (I_{\texttt{B}}, Decls, def_I)$ *of class* `B` *consists of*

- *a unique invariant placeholder* $I_{\texttt{B}}$ *for class* `B`*;*
- *a list Decls declaring the variables* `v_heap` *of type* `Heap` *and* `v_this` *of type* `B`*;*
- *a formula* def_I *which specifies the semantics of* $I_{\texttt{B}}$ *and that can make use of the two variables declared in Decls.*

We refer to formula $I_{\texttt{B}}(\texttt{v\_heap}, \texttt{v\_this})$ *as an* abstract invariant*.*

Definition 5 (Extended operation contract). *An* extended operation contract $C_{\mathtt{m}} = (Decls, C_a, defs)_{\mathtt{m}}$ *for a method* `m` *consists of*

- *Decls a list of variable and placeholder declarations;*
- $C_a := (pre_a, post_a, mod_a)$ *the* abstract operation contract *where the different constituents adhere to Def. 6;*
- *defs a list of pairs* (P, def_P) *with a formula* def_P *for each declared placeholder* $P \in Decls$*.*

Definition 6 (Abstract clauses). *The clauses for* abstract preconditions pre^a, abstract postconditions $post^a$, *and* abstract assignable clauses mod^a *are defined by the following grammar:*

$$\begin{aligned} pre^a &::= R \mid I \wedge pre^a \mid pre^a \wedge pre^a \\ post^a &::= E \wedge I \mid E \wedge pre^a \mid I \wedge pre^a \\ mod^a &::= A \mid mod^a \cup mod^a \end{aligned}$$

where R, E, A *and* I *are atomic formulas with a requires, ensures, assignable or invariant placeholder as top level symbol.*

To render extended invariants and operation contracts within JML we added some keywords which we explain along our running example:

Example 4. The following method contract for `computeGrade()` expresses semantically the same as the one shown in Fig. 1:

```
/*@ public normal_behavior
  @ requires_abs computeGradeR;
  @ ensures_abs  computeGradeE;
  @ assignable_abs computeGradeA;
  @
  @ def computeGradeR = true;
  @ def computeGradeE = \result == exam;
  @ def computeGradeA = \nothing;
  @*/
```

The contract is divided into two sections: the *abstract section* is in the upper section and uses the keywords **`requires_abs`**, **`ensures_abs`**, and **`assignable_abs`**. They are mainly used to declare the placeholders' names. The *concrete section* of the contract consists of the last three lines which provide the definition of the placeholders.

For invariants we do not modify the syntax at all. The placeholder name is generated automatically and the specified **`invariant`** expression is used as its definition. In our logical framework based on KeY, invariants are modelled as JML model fields and the specified invariants have been interpreted as represents clauses. Hence, adding support for abstract invariants was straightforward.

The advantage of having extended invariants and operation contracts is that the abstract contract can be used instead of a concrete contract when applying

the operation contract rule. It is also not necessary to modify the operation contract rule which guarantees the soundness of our approach as long as the original calculus was sound.

By using only abstract contracts, we can construct a proof for the correctness of a method `m` without committing to any concrete instantiation of an abstract contract neither for `m` itself nor for any of the methods it invokes. Once the verification conditions for the whole program are computed (e.g., by symbolic execution, constraint propagation, etc.), possibly followed by additional simplification steps, the open proof (goals) can be saved and cached.

To support abstract assignable clauses the underlying calculus must provide means to represent the modified locations within its logic. As mentioned in the previous section, our formalisation uses the explicit function *anon* which takes a location set as argument. Instead of providing a concrete set we use simply its placeholder.

To finish the proof that a method `m` adheres to its concrete specification we can reuse the cached proof (goals) without the need to redo the program transformation and simplification steps (as long as the implementation of `m` has not changed). It suffices to replace the introduced placeholders by their actual definitions and then continue proof search as usual. Replacing the placeholders by their definitions can be achieved, for example, by translating each placeholder definition into a rewrite rule of the underlying calculus which is added as an axiom.

The saving potential of our approach is particularly strong when verifying software with a high variability. In addition, our approach does not necessarily require to adhere to the Liskov principle [8] for specification but allows for a more flexible approach. We discuss the advantages in detail in the following Section.

4 Evaluation

We evaluate empirically the benefits offered by abstract contracts by measuring the amount of possible proof reuse. The evaluation is based on an implementation of our ideas in KeY. Our conjecture is that using abstract contracts results in considerable savings in terms of proof effort.

We evaluated our approach using two case studies of which each is implemented in several variants. The different variants have a common top-level interface, but differ in the implementation as well as in the concrete specification of called sub-routines. The abstract specification of each sub-routine, however, stays the same.

4.1 Description

The first case study *Student Record* has been already introduced before as a running example. We implemented several variants which differ in the implementation and specification of method `computeGrade()`, but keep method `passed()`

unchanged. The three variants differ in whether and how bonus credits are considered when computing the final exam grade. Version 3 is the version which supports no bonus points at all.

The second case study is taken from [4] and implements two classes, `Account` and `Transaction`. It is implemented in five versions. The most basic variant is shown in (Fig. 2). The class `Account` implements a simple bank account with method `boolean update(int x)` for deposit/withdrawal operations (depending on the sign of parameter `x`). In addition, there is an inverse operation `boolean undoUpdate(int x)`. Both methods signal whether the operation was successful with their return value. In the basic variant these methods perform the update unconditionally (for instance, it is not checked whether the account has enough savings).

The bank account can also be locked to prevent the execution of any operation. Locking is supported by methods `lock()`, `unLock()` and `isLocked()`. The behavior of these methods is not varied among the different variants.

Transactions between accounts are performed by calls to method `transfer` implemented by class `Transaction`. Method `transfer()` performs a number of pre-checks to ensure that the transaction will be successful before actually performing the transfer. For instance, it is checked that the balance of the source account is not negative, none of the accounts is locked, etc.

The versions differ in specification and/or implementation. The second version keeps the implementation of all classes unchanged, but simplifies the contract of method `transfer()` to cover only the case where a transaction is unsuccessful because of a negative amount to be transferred. In the third version an overdraft limit for accounts is supported, i.e., accounts are only allowed to be in the negative until a certain limit. This feature requires changes in the implementation and specification of methods `update()` and `undoUpdate()`. Method `transfer()` needs not to be modified with respect to the basic version. The fourth version extends the third version by adding a daily withdrawal limit. Again changes are necessary for the methods `update()` and `undoUpdate()`. The fifth version is again an extension of the third adding a fee to account operations. The changes affect the implementation and specification for the methods `update()`, `undoUpdate()` and the specification of method `transfer()`.

4.2 Results

For each program in the case studies we conducted three experiments with (i) fully abstract contracts, (ii) partially abstract contracts (the assignable clause is concrete, this correpsonds to the setup in [3]), and (iii) concrete contracts.

In each run where the specification was fully or partially abstract, the first step was to construct a partial proof that eliminates the verified programs and only uses abstract contracts and abstract invariants. This partial proof was then cached and reused to actually verify that the different variants satisfy their specifications. In each experiment with concrete contracts only, we directly ran the verification process on each program version.

```
public class Account {
  int balance = 0;
  boolean lock = false;

 /*@ public normal_behavior
   @ ensures (balance == \old(balance) + x) && \result;
   @ assignable balance;
   @*/
  boolean update(int x) { balance = balance + x; return true; }

 /*@ public normal_behavior
   @ ensures (balance == \old(balance) - x) && \result;
   @ assignable balance;
   @*/
  boolean undoUpdate(int x) { balance = balance - x; return true; }

 /*@ public normal_behavior
   @ ensures \result == this.lock;
   @*/
  boolean /*@ pure @*/ isLocked() { return lock; }
}

public class Transaction {
 /*@ public normal_behavior
   @ requires dest != null && source != null && source != destination;
   @ ensures \result ==> (\old(dest.balance) + amount == dest.balance);
   @ ensures \result ==> (\old(source.balance) - amount == source.balance);
   @ assignable \everything;
   @*/
  public boolean transfer(Account source, Account dest, int amount) {
   if (source.balance < 0) amount = -1;
   if (dest.isLocked()) amount = -1;
   if (source.isLocked()) amount = -1;
   int take, give;
   if (amount != -1) { take = amount * -1; give = amount; }
   if (amount <= 0) { return false; }
   if (!source.update(take)) { return false; }
   if (!dest.update(give)) {
        source.undoUpdate(take);
        return false;
   }
   return true;
}}
```

Fig. 2. Implementation and specification of the classes `Account` and `Transfer` (Ver. 1)

We measured the complexity of the cached proof and the final proofs in terms of nodes and branches (given in parenthesis). The results for the case

Version	Completely abstract			Partially Abstract			Concrete
	Partial proof	Full proof	Savings	Partial proof	Full proof	Savings	Full proof
v1	514 (9)	1191 (25)	43%	519 (9)	1145 (25)	45%	919 (21)
v2		1806 (40)	28%		1782 (40)	29%	1419 (36)
v3		1009 (24)	51%		937 (24)	55%	904 (22)
Total	2978			2826			3242

Table 1. Results for `StudentRecord` example

studies are shown in Table 1 and 2. The column *Savings* shows the amount of proof effort that has been saved (in terms of nodes) by reusing the cached proof. The row *Total* shows the aggregated proof effort (in nodes) for all program versions. For the experiments using concrete contracts the total equals the sum of the number of nodes for the different versions, i.e., $total = \sum_{i=1}^{n} \#nodes_{\mathrm{v}i}$ with n being the number of program versions. In case of the experiments using partially or completely abstract contracts, we compute the total as for concrete contracts, but subtract $n-1$ times the size of the partial proof, i.e., $total = (\sum_{i=1}^{n} \#nodes_{\mathrm{v}i}) - (n-1) \cdot nodes_{partial}$, to account for the proof reuse.

Version	Completely abstract			Partially Abstract			Concrete
	Partial proof	Full proof	Savings	Partial proof	Full proof	Savings	Full proof
v1	1390 (55)	2066 (57)	67%	1408 (55)	1865 (55)	75%	1035 (63)
v2		1882 (57)	74%		1723 (55)	82%	972 (63)
v3		2435 (59)	57%		1896 (55)	74%	1352 (68)
v4		2719 (59)	51%		1975 (55)	71%	1461 (68)
v5		2701 (61)	51%		2073 (57)	68%	1601 (69)
Total	6243			3900			6421

Table 2. Results for `Account` example

The results show that we achieve a reduced overall effort for fully abstract contracts as well as for partially abstract contracts. For the considered examples, the savings are most significant for partially abstract contracts. The reason is that here assignable clause is concrete, so the application of the contract rule keeps more information about the heap, while in the fully abstract case almost all

Version	Symbolic execution	Full proof	Ratio
v1	442 (11)	919 (21)	48%
v2	494 (11)	1419 (36)	35%
v3	410 (11)	904 (22)	45%

Table 3. Analysis of the `StudentRecord` example (specified with concrete contracts)

information about the heap is wiped out. As a consequence, branches checking for `NullPointerExceptions`, etc., cannot be closed in the abstract proofs.

Another reason why fully abstract contracts save perhpas less than expected is that in the case studies the assignable clauses in the concrete case contain only few locations and are mostly stable among the different versions. In the final version of this paper we will include a case study that shows more variability with respect to the set of assignable locations to illustrate the savings potential of fully abstract contracts.

Tables 3 and 4 show the ratio of proof nodes concerned with symbolic execution (i.e., program transformation) as compared to first-order reasoning. It can be directly seen that this ratio and the observed savings are strongly correlated.

Version	Symbolic execution	Full proof	Ratio
v1	842 (53)	1035 (63)	81%
v2	818 (53)	972 (63)	84%
v3	1037 (56)	1352 (68)	77%
v4	1147 (56)	1461 (68)	79%
v5	1141 (56)	1601 (69)	71%

Table 4. Analysis of the `Account` example (specified with concrete contracts)

Fig. 3 illustrates the *amortized* proof complexity of the `Account` example: the proof effort spent for the partial proof is distributed uniformly among the proofs of all five program versions (in case of partial and fully abstract contracts, respectively). We can see that using partial contracts perform best, while using fully abstract contracts leads in most cases to a total proof effort comparable to concrete ones. The reason is that when using abstract assignable clauses most knowledge about the heap after a method invocation is lost, prohibiting certain simplifications and creating more complex first-order problems. Currently, we investigate whether using SMT solvers for first-order goals (instead of the built-in KeY prover) can help. This should definitely be the case, provided that by using SMT solvers one can achieve a considerable speed-up for first-order inference.

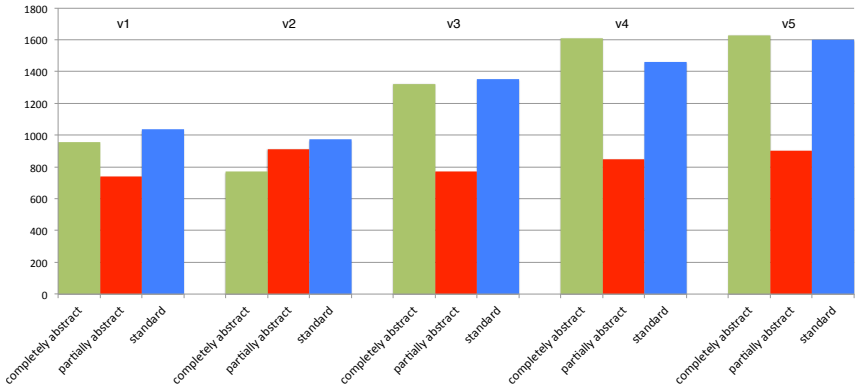

Fig. 3. Amortized proof complexity (in nodes) for the `Account` example

5 Related Work

As our work builds upon previous work [3] of some of the co-authors this is also the closest related work. As stated in the introduction, in the present paper we added abstract assignable clauses, abstract invariants, a proper implementation in KeY, plus an evaluation.

Proof reuse has been studied, for instance, in [9, 10] 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: in [9] a similarity measure is utilized, while in [10] differencing operations are applied. The proof reuse focusses only on the proof structure and does not take the specification into account like our work.

In [11], a set of allowed changes to evolve an OO program is introduced. For verified method contracts, a proof context is constructed which keeps track of 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. Earlier work along the same lines in the context of VCG is [12].

Reasoning by analogy is applied in [13] to reuse problem solving experience in proof planning. Generalization of proofs [14, 15] facilitates to reuse proofs in different contexts.

For formal software development in the large [16], evolving formal specifications are maintained by representing the dependencies between formal specifications and proofs in a development graph. For each modification, the effect in the development graph is computed such that only invalidated proofs have to be re-done. In [17], proofs are evolved together with formal specifications. A set of basic transformation operations for specifications induces the corresponding transformations of the proofs which may include the creation of new proof obligations.

In [18] 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 [19] methods are verified based on a contract which makes assumptions on the contracts of the called methods explicit.

In [20] abstract predicates are defined as abbreviations for specific properties to enforce modular reasoning in the sense of information hiding. Inside a module the definition of the abstract properties can be expanded while outside a module only the abstract predicates can be used. The paper does not use these abstract predicates to decouple symbolic execution from the application check of contracts and an application to proof reuse through caching is not considered.

We are not aware of any specification approach that keeps the used contracts fully abstract by using placeholder functions and predicates, as done here.

6 Conclusion and Future Work

We introduced fully abstract method contracts and class invariants in the context of contract-based verification. We showed that this idea can be simply realized with the help of placeholders. We do not assume any specific specification language, target language, or program logic. Any sufficiently expressive program logic can be used for an implementation. We instantiated and implemented the framework as part of the KeY verification system with Java as target language, JML as specification language, and dynamic logic as the program logic. An experimental evaluation showed that considerable proof reuse in the presence of changes and variations to specifications and code is possible.

In future work, we intend to investigate amount of savings that is achievable when using *mixed* contracts, where abstract contracts are enriched with certain concrete expressions that can be assumed to be stable across different versions of a program. Examples are formal parameters or field values can never be null, system values and boundaries, etc. Using this kind of information should allow us to simplify the cached proofs even further and increase the savings ratio. Further, we will investigate whether an inverse assignable clause (which lists the fields a method must *not* change) makes sense. Such a clause would enable us to retain more information after a method invocation and allow for a better reuse potential.

References

1. Meyer, B.: Applying "design by contract". IEEE Computer **25**(10) (1992) 40–51
2. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer-Verlag (2007)
3. Hähnle, R., Schaefer, I., Bubel, R.: Reuse in software verification by abstract method calls. In Bonacina, M.P., ed.: Proc. 24th Conference on Automated Deduction (CADE), Lake Placid, USA. Volume 7898 of Lecture Notes in Computer Science., Springer-Verlag (2013) 300–314
4. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based deductive verification of software product lines. In: Proceedings of the 11th International Conference on

Generative Programming and Component Engineering. GPCE '12, New York, NY, USA, ACM (2012) 11–20
5. 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. (September 2009) Draft revision 1.235.
6. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (October 1969)
7. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In Aichernig, B., de Boer, F.S., Bonsangue, M.M., eds.: Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Volume 6957 of LNCS., Springer-Verlag (2011) 142–164
8. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6) (1994) 1811–1841
9. Beckert, B., Klebanov, V.: Proof reuse for deductive program verification. In: Third IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society (2004) 77–86
10. Reif, W., Stenzel, K.: Reuse of proofs in software verification. In: Foundations of Software Technology and Theoretical Computer Science. (1993) 284–293
11. Dovland, J., Johnsen, E.B., Yu, I.C.: Tracking behavioral constraints during object-oriented software evolution. In Margaria, T., Steffen, B., eds.: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change — 5th International Symposium, ISoLA 2012, Crete, Greece. Volume 7609 of Lecture Notes in Computer Science., Springer (October 2012) 253–268
12. Grigore, R., Moskal, M.: Edit & verify. In: First-order Theorem Proving Workshop, Liverpool, UK. (2007)
13. Melis, E., Whittle, J.: Analogy in inductive theorem proving. J. Autom. Reasoning **22**(2) (1999) 117–147
14. Walther, C., Kolbe, T.: Proving theorems by reuse. Artificial Intelligence **116**(1-2) (2000) 17–66
15. Felty, A.P., Howe, D.J.: Generalization and reuse of tactic proofs. In: LPAR. (1994) 1–15
16. Hutter, D., Autexier, S.: Formal Software Development in MAYA. In: Mechanizing Mathematical Reasoning. (2005)
17. Schairer, A., Hutter, D.: Proof transformations for evolutionary formal software development. In: AMAST. (2002) 441–456
18. Bruns, D., Klebanov, V., Schaefer, I.: Verification of software product lines with delta-oriented slicing. In Beckert, B., Marché, C., eds.: International Conference on Formel Verification of Object-oriented Software (FoVeOOS 2010), Revised Selected Papers. Volume 6528 of Lecture Notes in Computer Science., Springer-Verlag (2011) 61–75
19. Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E.B., Yu, I.C.: A transformational proof system for delta-oriented programming. In: SPLC (2). (2012) 53–60
20. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '05, New York, NY, USA, ACM (2005) 247–258