Tracking Behavioral Constraints during Object-Oriented Software Evolution*

Johan Dovland, Einar Broch Johnsen, and Ingrid Chieh Yu

Department of Informatics, University of Oslo, Norway
{johand,einarj,ingridcy}@ifi.uio.no

Abstract. An intrinsic property of real world software is that it needs to evolve. The software is continuously changed during the initial development phase, and existing software may need modifications to meet new requirements. to facilitate the development and maintenance of programs, it is an advantage to have *programming environments* which allow the developer to alternate between programming and verification tasks in a flexible manner and which ensures correctness of the final program with respect to specified behavioral properties.
This paper proposes a *formal framework* for the flexible development of object-oriented programs, which supports an interleaving of programming and verification steps. The motivation for this framework is to avoid imposing restrictions on the programming steps to facilitate the verification steps, but rather to track *unresolved proof obligations* and *specified properties* of a program which evolves. A *proof environment* connects unresolved proof obligations and specified properties by means of a *soundness invariant* which is maintained by both programming and verification steps. Once the set of unresolved obligations is empty, the invariant ensures the soundness of the overall program verification.

1 Introduction

An intrinsic property of software in the real world is that it needs to evolve. This can be as part of the initial *development* phase, *improvements* to meet new requirements, or as part of a software *customization* process such as, e.g., feature selection in software product lines or delta-oriented programming [1,14]. Requirements to a piece of software also change over time. For this reason we cannot always expect that the specifications are written before the code is developed, and that the verification efforts happen afterwards. As the code is enhanced and modified, it becomes increasingly complex and *drifts away from its original design* [11]. For this reason, it may be desirable to redesign the code base to improve its structure, thereby reducing software complexity. For example, the process of *refactoring* in object-oriented software development describes changes to the internal structure of software to make the software easier to understand

* Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

and cheaper to modify without changing its observable behavior [6]. In this paper, the term *adaptable class hierarchies* captures class transformations which occur during object-oriented software evolution, including the development, improvement, customization, and refactoring of class hierarchies.

This paper proposes a formal framework for tracking behavioral constraints during such class transformations to allow incremental reasoning about adaptable class hierarchies, by extending the approach taken by *lazy behavioral subtyping* [4]. We consider a version of Featherweight Java [7] extended with behavioral interfaces, in which methods are annotated with pre/postconditions, and a number of *basic adaptation operations* for manipulating classes and interfaces, reflecting the level of basic program modifications. We consider a series of "snapshots" of a program during software development and evolution, in which the developer applies adaptation and analysis steps. A *proof environment* records both unresolved proof obligations and verified properties, and is manipulated by the different adaptation and analysis steps. Unresolved obligations reflect constraints that are imposed by the analysis, but it remains to ensure that they are satisfied. The purpose of analysis steps is to ensure that unresolved obligations are satisfied, whereas adaptation steps may spawn a number of unresolved obligations, reflecting that the program has changed. The spawned obligations depend on the actual adaptation, and may be inferred from the proof environment.

Paper overview: Sect. 2 motivates our approach, Sect. 3 introduces proof outlines, and Sect. 4 a kernel object-oriented language. Sect. 5 defines a soundness invariant for incremental reasoning, and Sect. 6 explains basic programming and verification tasks in our framework. Sect. 7 presents an example, Sect. 8 discusses related work, and Sect. 9 concludes the paper.

2 Motivation

There is a conflict of interest between the development and verification processes for software: the easier it is to flexibly develop and maintain programs in a language, the harder it is to verify programs in this language. Object-oriented programming, the de facto industry standard for software development, is a case in point: software verification projects have made significant progress in the last decade to support the verification of object-oriented programs, but features such as concurrency, class inheritance, and late binding still pose challenges.

Object orientation offers flexible ways of structuring and restructuring code by means of class inheritance and late binding, but reasoning about the behavior of object-oriented systems is in general non-trivial due to complications which arise from these code structuring mechanisms. Object-oriented software development is based on an open world assumption; i.e., class hierarchies are typically extendable. To have reasoning control under such an open world assumption, it is advantageous to have a framework which controls the properties required of method redefinitions. With *modular reasoning*, a new class can be analysed in the context of its superclasses, such that superclass' properties are guaranteed to be maintained. This has the significant advantage that each class can be fully

verified at once, independent of subclasses which may be designed later. The best known modular framework for class hierarchies is *behavioral subtyping* [8], but this framework has been criticized for being overly restrictive and is often violated in practice [15]. It is therefore of interest to investigate approaches which are better aligned with the flexibility expected by the software developer, even if these may have a higher price in terms of verification effort.

Incremental reasoning generalizes modular reasoning by possibly generating new verification conditions for superclasses to guarantee new properties. Additional properties may be established in superclasses after the initial analysis, but old properties remain valid. Incremental reasoning subsumes modularity: if the initial properties of a classes are sufficiently strong (e.g., by adhering to a behavioral contract), it never becomes necessary to add new properties. *Lazy behavioral subtyping* (LBS) is a formal framework for such incremental reasoning, which allows more flexible code reuse than modular frameworks. LBS is based on a separation of concerns between the behavioral *specifications* of method definitions and the behavioral *requirements* to method calls. Both specifications and requirements are manipulated through a bookkeeping framework which controls the analysis and the proof obligations in the context of a given class. Properties are only inherited by need. Inherited requirements on method redefinition are as weak as possible for ensuring soundness. LBS seems well-suited for the incremental reasoning style desirable for object-oriented software development, and can be adjusted to different mechanisms for code reuse. It was originally developed for single inheritance class hierarchies [4], but has later been extended to multiple inheritance [5] and to trait-based code reuse [2].

Adaptable class hierarchies add a level of complexity to proof systems for object-oriented programs, as classes in the middle of a hierarchy can change. Unrestricted, such changes may easily violate previously verified properties in both sub- and superclasses. The management of verification conditions becomes more complicated than for a class hierarchy which is only extended at the bottom. To facilitate program development and maintenance, it is an advantage that programming and verification activities go hand in hand. For this purpose, we need programming environments for flexible alternation between programming and verification tasks. The proposed analysis does not assume that class hierarchies are build top-down; the internal class structures may be revealed during implementation. The proposed analysis technique is developed with the intention to better integrate formal verification with software engineering processes.

3 Proof Outlines and Soundness

The reasoning framework is presented in terms of *proof outlines* [12], which can be explained in terms of Hoare triples. A Hoare triple $\{p\}\, t\, \{q\}$ defines the effect on a state described by the *precondition* p when a statement t executes, leading to a state described by the *postcondition* q (where p and q are assertions). The meaning $\models \{p\}\, t\, \{q\}$ of a triple $\{p\}\, t\, \{q\}$ is here given by a standard partial correctness interpretation: if t is executed in a state where p holds and the execution

terminates, then q holds in the state after t has terminated. The derivation of triples can be done in any suitable program logic. Let PL be such a program logic and let $\vdash_{PL} \{p\}\, t\, \{q\}$ denote that $\{p\}\, t\, \{q\}$ is derivable in PL.

A proof outline for t is obtained by decorating t with assertions at selected program points such that the analysis between these program points can be done mechanically. A classical example is to decorate loops in the program with loop invariants. For the purposes of this paper, we are interested in decorating method calls with pre- and postconditions, and we assume that all method calls in the considered proof outlines are decorated. Let $O \vdash_{PL} t : (p, q)$ denote that O is a proof outline for t such that $\vdash_{PL} \{p\}\, t\, \{q\}$ holds, assuming that the decorated statements O are correct. The assertion pair (p, q) is called a *guarantee* for t, and to the decorated call statements in O as *requirements* for the called methods, and we say that these requirements are *imposed* by t. Thus, for a decorated method call $\{r\}\, n()\, \{s\}$ in O, we say that (r, s) is a requirement for n. This terminology can be lifted to method definitions $m(\overline{x})\{t\}$ as follows: If the proof outline O is such that $O \vdash_{PL} t : (p, q)$, we say that m guarantees (p, q) by imposing the requirements in O on the methods that are called by the method body t.

Given a set of methods, proof outlines allow a "divide and conquer" technique in the overall program analysis. For each method we may establish a guarantee by providing a proof outline for the method body. For the overall *soundness* of the program analysis, we need to ensure that each requirement in a proof outline follows from the guarantee of the called method. Let (p, q) be the guarantee for m, and assume that the requirement (r, s) is imposed on m by some proof outline. We essentially need to check the implications $r \Rightarrow p$ and $q \Rightarrow s$ which can be captured by an *entailment relation* $\twoheadrightarrow$ over assertion pairs, defined as follows [5]:

$$(p, q) \;\twoheadrightarrow\; (r, s) \triangleq (\forall \overline{z}_0 \,.\, p \Rightarrow q') \Rightarrow (\forall \overline{z}_1 \,.\, r \Rightarrow s')$$

Here, $\overline{z}_0$ and $\overline{z}_1$ denote the logical variables in (p, q) and (r, s), respectively, and the primed assertions q' and s' replace all occurrences of the fields f in q and s by some fresh name f'. This entailment relation may be lifted to sets of assertion pairs [5], e.g., to prove that (r, s) follows from a set of assertion pairs.

Given a closed set of methods (i.e., each method called from the set is defined in the set) and a proof outline establishing a guarantee for each method, the set of proof outlines is *sound* if each requirement follows from the method guarantee in the set. For a proof outline $O \vdash_{PL} t : (p, q)$ in the set, we have $\models \{p\}\, t\, \{q\}$.

4 Proof Outlines for Object-Oriented Programs

In this section the soundness notion for proof outlines is extended to an object-oriented context where the methods are organized in classes in a class hierarchy.

4.1 An Object-Oriented Kernel Language

We consider a kernel object-oriented language with the syntax given in Fig. 1. A program P defines interface and class. An interface I extends superinterfaces

$$
\begin{array}{lll}
P & ::= \overline{K}\ \overline{L} & K \quad ::= \textsf{interface}\ I\ \textsf{extends}\ \overline{I}\ \{\overline{MA}\} \\
T & ::= I \mid \mathsf{Bool} \mid \mathsf{Int} & L \quad ::= \textsf{class}\ C\ \textsf{extends}\ C\ \textsf{implements}\ \overline{I}\ \{\overline{F}; \overline{M}\} \\
F & ::= T\ f & MA ::= [T \mid \mathsf{Void}]\ m\ (\overline{T\ x}) : (p, q) \\
M & ::= MA\ \{\overline{T\ x}; t; \textsf{return}\ e\} & t \quad ::= t; t \mid v := rhs \mid v.m(\overline{e}) \mid \textsf{if}\ b\ \{t\} \mid \textsf{skip} \\
v & ::= f \mid x & rhs\ ::= \textsf{new}\ C(\,) \mid v.m(\overline{e}) \mid m(\overline{e}) \mid e
\end{array}
$$

Fig. 1. The language syntax. C is a class name, and I an interface name. Variables v are fields (f) or local variables (x), and e denotes side-effect free expressions over the variables, b expressions of Boolean type, and p and q are assertions. Vector notation denotes lists, as in the expression list $\overline{e}$, interface list $\overline{I}$, and in the variable declaration list $\overline{T\ v}$, otherwise vectors denote sets, as in $\overline{K}, \overline{L}, \overline{MA}$ and $\overline{M}$. To distinguish assignments from equations in specifications and expressions, we use := and = respectively.

$\overline{I}$ and declares a set of *method constraints* $\overline{MA}$, where a constraint is given by a method signature with pre/post assertions. An interface may extend its superinterfaces with declarations of new methods and with additional constraints for methods already declared in the superinterfaces. We say that I *provides* the methods declared in I or in a superinterface of I. For interfaces I and J, we say that I is *below* J and J is *above* I if I equals J or if J is a superinterface of I.

A class C may inherit from at most one direct superclass B, implement a list $\overline{I}$ of interfaces, and define fields $\overline{F}$ and methods $\overline{M}$. The class may override superclass methods, but we assume no method overloading and no field shadowing (fields with the same name in different classes may be qualified by class names). We say that a method m is *available* in C if m is defined in $\overline{M}$ or a definition is inherited from the superclass B, i.e., the method is available in B. To implement an interface I, each method provided by I must be available in C and the interface constraints must be satisfied. Class C may in addition define *auxiliary* methods for internal purposes. For flexibility, interfaces are not inherited at the class level: The class C may implement different interfaces than those of its superclass B, which leads to a separation of class hierarchies and type hierarchies. Remark that the situation where interfaces are inherited at class level may be considered as the special case where C must implement at least the interfaces of B, leading to behavioral subtyping constraints on class inheritance. *Local calls* in C are late-bound in a standard bottom-up manner following the superclass relation: when a local call $m()$ is executed on an instance of C, the binding is resolved by starting the bottom-up search in C. For classes C and D, we say that C is *above* D and D is *below* C if C equals D or if C is a superclass of D.

Object references are typed by interfaces. Let $v : I$ denote that v is a variable of type I, so v may refer to an instance of any class D, implementing an interface below I. For *external calls* $v.m()$, m must be provided by I, and can bind to any object to which v may refer. Statements t and expressions e are standard.

4.2 Proof Outlines and Inheritance

The notion of proof outlines extends naturally to object-oriented programs; to specify and reason about a program, proof outlines may be provided for the

methods implemented in the classes of the program. Each proof outline gives a method guarantee and a set of method call requirements. However, in contrast to the presentation in Sect. 3, there is not a one-to-one correspondence between a call statement and the implementations to which the call may bind. For a requirement $\{r\}\ v.m()\ \{s\}$ with v typed by interface I, we need to ensure that (r, s) follows from each implementation to which the call may bind, i.e, for each class that implements some interface below I. However, proving this directly for each class requires global knowledge about all classes, and contradicts the open world assumption by which classes may be incrementally added.

To enhance the modularity of the reasoning system, we therefore assume that interface constraints are sufficiently strong to analyze external calls. For $\{r\}\ v.m()\ \{s\}$, this means that (r, s) must follow from the constraints for m in I. If a class D implements I or a subinterface of I, the constraints for m in I must be satisfied by the implementation. By transitivity, we then know that the external call requirement is satisfied by all implementations to which the external call can be bound. This reasoning approach is feasible in an open environment where the programmer does not control all parts of the program: Calls to external objects may be done without knowing the detailed implementation of those objects. If the interfaces are fixed, this means that the classes of external objects may be implemented independently from the current class. Also, the approach facilitates e.g., calls to library methods without consulting the library implementation. Remark that interface encapsulation generally leads to incomplete reasoning systems, since an interface represents an abstraction of the actual implementations. This is illustrated by the following example:

Example 1. Consider an interface I with a method m and a class C which implements I. The class D makes a call to m of a newly created instance of C.

```
interface I { Int m() : (true, return ≥0)}
class C implements I {Int m() {return 2} : (true, return = 2)}
class D implements J {Int n() {I x := new C; Int v := x.m(); return v}}
```

The guarantee for m in C satisfies the interface constraint. The question is what we know about the value returned by method n in class D. By inspecting the code, we see that the method will always return the value 2. A proof outline for for n with guarantee $(\mathsf{true}, \mathbf{return} = 2)$ results in a call requirement $\{\mathsf{true}\}\mathsf{x.m()}\{\mathbf{return} = 2\}$ but the constraint given by I only promises $\mathbf{return} \geq 0$, which means that the requirement cannot be verified. Remark that a proof outline for n with guarantee $(\mathsf{true}, \mathbf{return} \geq 0)$ can be verified.

For a language with interface encapsulation, a natural goal for the program analysis is to verify that classes satisfy the constraints of the implemented interfaces. Let $\mathit{body}(C, m)$ denote the implementation of a method m that is available in C. If I gives the constraint (p, q) for m and C implements I, we must ensure $\models_C \{p\}\ \mathit{body}(C, m)\ \{q\}$, where the subscript denotes that the triple must be true when executed on an instance of C. We emphasize that the constraint originates from the fact that C implements I. Especially, the constraint need not be satisfied if $\mathit{body}(C, m)$ is executed on an instance of some subclass D of C, if D does

```
interface I { Int m() : (true, return=0)}
interface J { Int m() : (true, return>0)}
interface K { Int m() : (true, return<0)}

class C implements I {Int x;
  Int m() {x:=n(); return x}
  Int n() {return 0}}
class D extends C implements J {Int n() {return 1}}
class E extends C implements K {Int n() {return −1}}
```

Fig. 2. A small class hierarchy with method overriding.

not implement I. Related to C, the binding of a local call to some method n is uniquely determined by the available implementation in C. Given the above constraint (p, q) and a proof outline O such that $O \vdash_{PL} \mathit{body}(C, m) : (p, q)$ and $\{r\}\, n()\, \{s\}$ is a requirement in O, it is sufficient to analyze the requirement with respect to the implementation available in C. If the analysis of all requirements succeed, we may then conclude $\models_C \{p\}\ \mathit{body}(C, m)\ \{q\}$.

Since locally called methods may be overridden differently in different subclasses, it is natural to allow more than one guarantee for each method. The guarantees may possibly be in conflict as illustrated by Example 2.

Example 2. Consider the code in Fig. 2. For class C, we supply a proof outline for m with the guarantee $(\mathsf{true}, \mathbf{return} = 0)$ to ensure that C satisfies the constraints of interface I. This proof outline imposes the requirement $\{\mathsf{true}\}\mathsf{n}()\{\mathbf{return} = 0\}$, which can be verified for n as defined in C. For class D, the method m must satisfy the interface constraint $(\mathsf{true}, \mathbf{return} > 0)$. A proof outline with this guarantee yields the requirement $\{\mathsf{true}\}\ \mathsf{n}()\ \{\mathbf{return} > 0\}$, which can be verified for the overriding version of n in D. The verification of class E follows the same pattern as for D. Combined, this leads to three different proof outlines for m, according to the different behavior of the called method. When analyzing each class, we may select the proof outline that fits with the actual interface constraint.

Allowing many proof outlines provides flexibility when analyzing independent properties. For instance, if C implements two interfaces I and J, I declares a constraint (p_I, q_I) for m, and J declares a different constraint (p_J, q_J) for m. The constraints may be verified independently by providing two proof outlines.

Assuming type safety, we formulate the following *soundness conditions* for a class C implementing the interfaces $\overline{I}$: *A set of proof outlines (with guarantees and requirements) for the methods available in C are given such that:*

- *For each method m provided by $\overline{I}$, the guarantees for m ensure all interface constraints.*
- *For external calls $\{r\}\, v.m()\, \{s\}$ in some proof outline, the requirement (r, s) follows from the constraints for m in I (where $v : I$).*
- *For each local call $\{r\}\, m()\, \{s\}$ in some proof outline, the requirement (r, s) follows from the guarantees for m.*

If a proof outline for m in C with guarantee (p, q) is verified, the soundness constraints ensure $\models_C \{p\}\ body(C, m)\ \{q\}$ (provided that PL is also sound).

5 A Soundness Invariant for the Open World

In this section we generalize the above soundness conditions for an open world assumption for software evolution, by accommodating an interleaving of software evolution and analysis tasks. The generalization is formulated as a *soundness invariant* which is maintained by each individual task. Software evolution tasks are represented as *basic adaptations* which are applied to the classes of an existing program. These adaptations affect the class hierarchy of the program, and are discussed in detail in Sect. 6. Analysis tasks are performed at user request, and the user may select which program properties to analyze.

We assume given a set $\mathcal{U}(C)$ of *unresolved obligations* associated with each class C. This set contains requirements and constraints imposed by the analysis so far, but for which we have not checked that they are satisfied. The set $\mathcal{U}(C)$ is in general extended by the different adaptation tasks, reflecting that proof obligations are spawned if the program is changed. Program analysis removes obligations from $\mathcal{U}(C)$ and perform the actions necessary to maintain the soundness invariant. We identify three kinds of obligations that occur in $\mathcal{U}(C)$:

- $I \rightsquigarrow m : (p, q)$. Here, $m : (p, q)$ is a constraint imposed by I. When this obligation appears in $\mathcal{U}(C)$, the class C is declared to implement I, but it remains to ensure that the implementation actually satisfies the constraint.
- $m \rightsquigarrow n : (r, s)$. Method m is here available in C and the obligation reflects the requirement $\{r\}\, n\, \{s\}$ of a local call statement in a verified proof outline for m. When this obligation appears in $\mathcal{U}(C)$, it remains to ensure that the requirement is satisfied by the called method.
- $m \rightsquigarrow I : n : (r, s)$. This obligation reflects that the requirement $\{r\}\ v.n\ \{s\}$, with $v : I$, is imposed by an external call statement in a verified proof outline for m. When this obligation appears in $\mathcal{U}(C)$, it remains to check that (r, s) follows from the constraints of I.

Remark that the last two obligations include the name of the method which imposes the requirement. As explained in more detail in Sect. 6 this allows us to discard obligations if m changes before the requirements have been analyzed. The soundness invariant is formulated by weakening the above soundness conditions:

Definition 1 (Soundness invariant). *For each class C implementing interfaces $\overline{I}$, the set of proof outlines for the methods available in C are such that:*

- *For each m provided by some I in $\overline{I}$ with constraint (p, q), either (p, q) follows from the guarantees for m, or $I \rightsquigarrow m : (p, q)$ is in $\mathcal{U}(C)$.*
- *For each external call $\{r\}\ v.m()\ \{s\}$ in a proof outline for method n, where $v : I$, either (r, s) follows from the constraints for m in I, or $n \rightsquigarrow I : m : (r, s)$ is in $\mathcal{U}(C)$.*

$$
\begin{aligned}
\textit{Class adaptation} ::= {} & \mathit{newCls}(C) \mid \mathit{remCls}(C) \mid \mathit{newFld}(C, F) \\
& \mid \mathit{remFld}(C, F) \mid \mathit{newMtd}(C, M) \mid \mathit{remMtd}(C, M) \\
& \mid \mathit{setSup}(C, B) \mid \mathit{newImpl}(C, I) \mid \mathit{remImpl}(C, I) \\
\textit{Interface adaptation} ::= {} & \mathit{newInt}(I) \mid \mathit{remInt}(I) \mid \mathit{newConstr}(I, m : (p, q)) \\
& \mid \mathit{remConstr}(I, m : (p, q)) \mid \mathit{newSup}(I, J) \\
& \mid \mathit{remSup}(I, J)
\end{aligned}
$$

Fig. 3. Basic class and interface adaptations.

– *For each local call* $\{r\}\ m()\ \{s\}$ *in a proof outline for method* n*, either* (r, s) *follows from the guarantees for* m*, or* $n \rightsquigarrow m : (p, q)$ *is in* $\mathcal{U}(C)$*.*

If all unresolved requirements have been verified, i.e., the set $\mathcal{U}(C)$ is empty, the soundness invariant reduces to the soundness conditions given in Sect. 4. The soundness of a class C depends only on declared interfaces and classes above C: i.e., the soundness of C is not affected by modifications of subclasses C.

6 Evolution through Adaptable Class Hierarchies

Object-oriented software evolution can be perceived as a sequence of adaptations to a class hierarchy. The framework allows an interleaving of adaptation and analysis tasks, initiated by the user. A *program environment* $\mathcal{P}$ keeps track of the current definitions of classes and interfaces. In addition to the *unresolved obligations* sets $\mathcal{U}$, each class C and method m available in C will maintain a set $\mathcal{G}(C, m)$ of *verified specifications*. Elements in $\mathcal{G}(C, m)$ are tuples consisting of an assertion pair (the guarantee) and a proof outline (capturing the requirements).

6.1 Basic Program Adaptations

Consider a suite of basic program adaptations as given in Fig. 3, each of which reflects evolution at the level of a single structuring artefact (i.e., a method, class, or interface in the kernel language). We focus on behavioral analysis, so certain behavioral preserving modifications are not considered, such as the consequent renaming of fields or methods within a program. Such renaming is captured by the basic adaptations, but would produce a number of trivial proof obligations. We explain how each adaptation maintains the soundness invariant by recording unresolved obligations, and discuss the verification complexity. Complex adaptations may be constructed by combining basic adaptations (see Sect. 6.2).

Class adaptations. We consider the following basic adaptations for classes:

– ***newCls***(C) **and** ***remCls***(C). The adaptation $\mathit{newCls}(C)$ inserts a new class with name C in $\mathcal{P}$. The new class is initially empty (i.e., without inheritance and implements clauses and method definitions), so the soundness invariant is maintained without modifying the sets $\mathcal{U}$ and $\mathcal{G}$. The adaptation $\mathit{remCls}(C)$ removes the class C from $\mathcal{P}$. The set $\mathcal{U}(C)$ is erased and all verified proof outlines for C in $\mathcal{G}$ are removed. For this adaptation to be safe,

we assume that C has no subclasses and that C does not appear in any **new** statements, which requires global knowledge about the classes.

- ***newFld***(C, F) **and** ***remFld***(C, F). The adaptation $newFld(C, F)$ includes the field F in C, and $remFld(C, F)$ removes field F from C. Since there are no field shadowing, we may assume that a new field is not used in classes above or below C. To preserve type safety, a removed field cannot be used in classes below C.
- ***newMtd***(C, M). The definition of C is here extended with a method M, or M replaces the old version if a method with the same name was previously defined in C. Let m be the name of M. The soundness invariant is maintained as follows: If m is redefined, the verified specifications for the old version no longer apply, thus the set $\mathcal{G}(C, m)$ is emptied. Consequently, unresolved obligations in $\mathcal{U}(C)$ that are imposed by these specifications are removed. Such obligations are of the forms $m \rightsquigarrow I : n : (r, s)$ and $m \rightsquigarrow n : (r, s)$ for some I, n, and (r, s). We furthermore perform the following steps: *a)* If m is public, the constraints for m, imposed by the implemented interfaces, are added to $\mathcal{U}(C)$ as obligations of the form $I \rightsquigarrow m : (p, q)$ for some I and (p, q); *b)* Verified specifications for other methods in C may impose requirements on m. These requirements are included in $\mathcal{U}(C)$ as obligations of the form $n \rightsquigarrow m : (r, s)$ for some n and (r, s); and *c)* For each subclass D of C which inherits m from C, we perform the corresponding modifications of $\mathcal{U}(D)$ and $\mathcal{G}(D, m)$ as explained for C above.
- ***remMtd***(C, M). Removing a method with name m is similar to method redefinition. We empty $\mathcal{G}(C, m)$ and remove unresolved obligations imposed by m from $\mathcal{U}(C)$. However, there may still be calls to m in C if the method is inherited from some superclass B of C. Therefore, $\mathcal{U}(C)$ is extended by interface constraints if m is public, and with requirements imposed by the verified specifications of C. All of these modifications are repeated for each subclass D of C which inherits m from B.
- ***setSup***(C, B). This adaptation sets B to be the immediate superclass of C. After the operation, C extends B in $\mathcal{P}$. To maintain the soundness invariant, we consider each available method definition M in B that is not overridden by C. The operations performed to $\mathcal{U}$ and $\mathcal{G}$ correspond to the operations needed by a $remMtd(C, M)$ adaptation, which means that unresolved obligations may also be added to subclasses of C. Remark that the soundness invariant is maintained for the old direct superclass of C, since the soundness of that class does not depend on its subclass C.
- ***newImpl***(C, I). This adaptation extends the implements clause of C with interface I. The soundness invariant is maintained by extending $\mathcal{U}(C)$ with all constraints of I. The adaptation has only local effects on C.
- ***remImpl***(C, I). This adaptation removes I from the implements clause of C. Locally, this means that we can remove unresolved obligations from $\mathcal{U}(C)$ that are imposed by this interface. However, this is a complicated adaptation to implement, since it requires global knowledge of the system. To be safe, no references to C objects can be typed by an interface above I. Especially, for

each statement $v :=$ **new** $C(\,)$, the declared type of v must be implemented by C after reducing the implements clause.

Interface adaptations. We consider these basic adaptations for interfaces:

- ***newInt***(I). This adaptation introduces a new empty interface in $\mathcal{P}$, which trivially maintains the soundness invariant.
- ***remInt***(I). This adaptation removes the interface I from $\mathcal{P}$. Global concerns must be taken for the adaptation to be safe, ensuring that no other interface is inheriting I and that no class implements I. Thus to perform this adaptation, it may be necessary to first perform a sequence of *remSup* (explained below) and *remImpl* adaptations, which is quite expensive. Remark that as an effect of such a sequence, all references typed by I are removed from the global system, which makes it safe to remove I.
- ***newConstr***$(I, m : (p, q))$. The definition of I is here extended by the constraint $m : (p, q)$. The adaptation may be used to add a new constraint to an already provided method, or to extend I such that it provides m. To preserve the soundness invariant, we must find each class C which implements an interface below I, and add $I \rightsquigarrow m : (p, q)$ to $\mathcal{U}(C)$.
- ***remConstr***$(I, m : (p, q))$. By removing the constraint $m : (p, q)$ from I, the behavior that can be assumed for external calls made via I is reduced. Thus, to maintain the soundness invariant, a global check must be performed on all requirements imposed on m via I. For each class C with method n, we consider $\mathcal{G}(C, n)$. For any call $\{r\}\, v.m()\, \{s\}$ with $v : I'$ in these proof outlines, $\mathcal{U}(C)$ is extended by $n \rightsquigarrow I' : m : (r, s)$, if I' is below I.
- ***newSup***(I, J). This adaptation includes interface J in the extends clause of interface I. As a result, we need to check that all constraints above J are satisfied for classes which implement an interface below I. For each such class C and constraint $m : (p, q)$ of J, we extend $\mathcal{U}(C)$ with $J \rightsquigarrow m : (p, q)$
- ***remSup***(I, J). This adaptation removes J from the extends clause if I. As for the *remConstr* adaptation, this means that the behavior assumed by external calls via I is reduced. For each constraint $m : (p, q)$ of J, we need to check all classes which make calls to m via I or a subinterface of I, in the same manner as for a *remConstr*$(I, m : (p, q))$ adaptation.

6.2 Combining Adaptations

High-level operations can be defined from the basic adaptations. Adaptations may be lifted to take more than one element as the second argument by flattening the adaptation to a *sequence* of basic adaptations; e.g., $\mathit{newMtd}(C, M \cup \overline{M}) \triangleq \mathit{newMtd}(C, M) \cdot \mathit{newMtd}(C, \overline{M})$ (where $\cdot$ is the sequence append constructor). Extending the program with a *new class definition* may then be defined by:

$$\mathsf{class\ C\ extends\ B\ implements}\ \overline{I}\ \{\overline{F};\ \overline{M}\} \triangleq \mathit{newCls}(C) \cdot \mathit{setSup}(C, B) \cdot \mathit{newImpl}(C, \overline{I}) \cdot \mathit{newFld}(C, \overline{F}) \cdot \mathit{newMtd}(C, \overline{M})$$

The following adaptation *modifies an existing class definition*, adding support for new interfaces, defining new fields, and (re)defining methods:

$$\mathsf{modify\ C\ implements}\ \overline{I}\ \{\overline{F};\overline{M}\} \triangleq \mathit{newImpl}(C,\overline{I}) \cdot \mathit{newFld}(C,\overline{F}) \cdot \mathit{newMtd}(C,\overline{M})$$

The basic adaptations may be further combined to cover common refactoring patterns [6]. For instance, *moving a method* M from class C to class B is captured by the sequence $\mathit{remMtd}(C,M) \cdot \mathit{newMtd}(B,M)$. By applying predefined refactorings defined as sequences of basic adaptations, the user may ensure that program analysis is postponed as appropriate. For instance, if $\mathit{remMtd}(C,M) \cdot \mathit{newMtd}(B,M)$ denotes a simple *pull up method refactoring* from C to a superclass B, program analysis will probably fail between the two adaptations as the proof obligations cannot in general be resolved at that stage.

6.3 Analysis Tasks

An analysis task removes an obligation from $\mathcal{U}(C)$ which is analyzed depending on its structure. This analysis may spawn new proof obligations which are included in $\mathcal{U}(C)$; the size of $\mathcal{U}(C)$ does not necessarily shrink by each task. However, each analysis task maintains the soundness invariant, and the soundness conditions are ensured if all unresolved obligations are successfully analyzed.

- *Obligation* $I \rightsquigarrow m : (p,q) \in \mathcal{U}(C)$. This obligation is resolved if (p,q) follows by entailment from the guarantees in $\mathcal{G}(C,m)$. To ensure entailment, it may be necessary to first extend $\mathcal{G}(C,m)$. Let O be a proof outline such that $O \vdash_{PL} \mathit{body}(C,m) : (p',q')$ for some (p',q'). Extending $\mathcal{G}(C,m)$ by $\langle (p',q'), O\rangle$ means that $\mathcal{U}(C)$ must be extended to maintain the soundness invariant: For each requirement of the form $\{r\}\, n()\, \{s\}$ in O, the obligation $m \rightsquigarrow n : (r,s)$ is included in $\mathcal{U}(C)$, and for each requirement of the form $\{r\}\, v.n()\, \{s\}$ with $v : I$, the obligation $m \rightsquigarrow I : n : (r,s)$ is included in $\mathcal{U}(C)$. Remark that the supplied guarantee (p',q') may be identical to (p,q).
- *Obligation* $m \rightsquigarrow n : (r,s) \in \mathcal{U}(C)$. This obligation is resolved if (r,s) follows by entailment from the guarantees in $\mathcal{G}(C,n)$. Similar to above, it may be necessary to first extend $\mathcal{G}(C,n)$.
- *Obligation* $m \rightsquigarrow I : n : (r,s) \in \mathcal{U}(C)$. This obligation is resolved if (r,s) follows from the constraints for m in I. One may need to extend the constraints of I before removal, e.g., by the adaptation $\mathit{newConstr}(I, n : (r,s))$.

7 Example

Figure 4 presents a snapshot of a small bank account system under development. A class Account provides basic operations for depositing and withdrawing amounts of money. The balance of the account is stored in a field bal. A class Customer has references to two Account objects; a regular account reg and a savings account sav. The method save implements functionality for transferring an

```
interface IAccount {
  Bool deposit(Int x)  : (bal = b0 ∧ x > 0, bal = b0 + x ∧ return)
  Bool withdraw(Int x) : (bal = b0 ∧ x > 0 ∧ bal − x ≥ 0, bal = b0 − x ∧ return)
                         (bal = b0 ∧ x > 0 ∧ bal − x < 0, bal = b0 ∧ ¬return)
}
class Account implements IAccount { Int bal := 0;
  Bool deposit(Int x) { Bool val := false ;
    if (x>0) {bal := bal+x; val := true}; return val
  }
  Bool withdraw(Int x) {Bool val := false ;
    if (x>0 ∧ bal−x≥0) {bal := bal−x; val := true}; return val}}
class Customer { IAccount reg, sav;
  Bool save (Int amt) {
    Bool res := reg.withdraw(amt); if (res) {sav.deposit(amt)}; return res}}
```

Fig. 4. The initial bank account system, with Account and Customer classes.

amount from the regular account to the savings account. We consider different adaptations of the code in Fig. 4 and explain the essential parts of the analysis.

Internal modifications of Account. Assume that the programmer extracts the manipulation of bal in Account into one method, defined by the adaptation

$$newMtd(\mathsf{Account}, \mathsf{Bool\ update(Int\ y)} : (bal = b_0, bal = b_0 + y \wedge \mathbf{return})\ \{ \mathsf{bal := bal+y;\ return\ true}\}).$$

The guarantee may be verified by a proof outline which imposes no requirements. In the sequel, we assume that this specification is in $\mathcal{G}(\mathsf{Account}, \mathsf{update})$.

Next we consider removing the assignments to bal from deposit and withdraw. For deposit, we may apply the following adaptation:

$$newMtd(\mathsf{Account}, \mathsf{Bool\ deposit(Int\ x)\ \{\ Bool\ val:=false\ if\ (x>0)\ \{\ val := update(x)\}; return\ val}\}).$$

Since the method definition has changed, we cannot rely on a previous verification of this method; i.e., all elements must be removed from $\mathcal{G}(\mathsf{Account}, \mathsf{deposit})$. However, since deposit is public, the interface constraint for this method is currently unresolved. This means that the following element is added to $\mathcal{U}(\mathsf{Account})$:

$$\mathsf{IAccount} \rightsquigarrow \mathsf{deposit} : (bal = b_0 \wedge x > 0, bal = b_0 + x \wedge \mathbf{return}).$$

We may consider this constraint as a guarantee for the new version of deposit, and verify a proof outline O with $\{x = y \wedge bal = b_0\}\,\mathsf{update}\,\{bal = b_0 + x \wedge \mathbf{return}\}$ as requirement. The set $\mathcal{G}(\mathsf{Account}, \mathsf{deposit})$ is then extended by

$$\langle (bal = b_0 \wedge x > 0, bal = b_0 + x \wedge \mathbf{return}), O \rangle$$

and the obligation $\mathsf{deposit} \rightsquigarrow \mathsf{update} : (y = x \wedge bal = b_0, bal = b_0 + x \wedge \mathbf{return})$ is included in $\mathcal{U}(\mathsf{Account})$. Now, both elements of $\mathcal{U}(\mathsf{Account})$ follow directly from

the specifications of their respective methods. Removing the assignment to bal in withdraw follows the same pattern by applying the adaptation

$$\begin{aligned} &\mathit{newMtd}(\mathsf{Account}, \mathsf{Bool\ withdraw(Int\ x)\ \{\ Bool\ val{:=}false} \\ &\qquad\qquad \textbf{if}\ (\mathsf{x{>}0} \wedge \mathsf{bal\text{-}x} \geq 0)\ \{\ \mathsf{val} := \mathsf{update(\text{-}x)}\};\ \textbf{return}\ \mathsf{val}\}). \end{aligned}$$

The interface constraints listed in Fig. 4 are added to $\mathcal{U}(\mathsf{Account})$. These can be analyzed by proof outlines which rely on the verified specification of update.

Adding new functionality. In Customer, the developer adds functionality to save money if the balance of regular account has reached a given value:

$$\begin{aligned} &\mathit{newMtd}(\mathsf{Customer}, \mathsf{Bool\ saveLimit()\ \{\ Bool\ res := false;\ Int\ rbal := reg.getBal();} \\ &\qquad\qquad \textbf{if}\ (\mathsf{rbal{>}limit})\ \{\mathsf{res{:=}save(rbal\text{-}limit)}\};\ \textbf{return}\ \mathsf{res}\}). \end{aligned}$$

The new method calls a method getBal via IAccount, but getBal has not yet been defined. However, the developer of Customer may assume that the method is there by postponing the adaptation of IAccount. Especially, the new method in Customer may be analyzed before adapting IAccount; new requirements will then be available at the time the interface is extended. We illustrate how such a requirement is tracked, ensuring that both IAccount and the class implementing this interface satisfy the requirement. Assume that the analysis of saveLimit imposes the following requirement on getBal, which is included in $\mathcal{U}(\mathsf{Customer})$:

$$\mathsf{saveLimit} \rightsquigarrow \mathsf{IAccount{:}getBal} : (bal = b_0, bal = b_0 \wedge \textbf{return} = b_0).$$

To resolve this proof obligation, the interface IAccount must be extended, for example by the following adaptation:

$$\mathit{newConstr}(\mathsf{IAccount}, \mathsf{Int\ getBal} : (bal = b_0, bal = b_0 \wedge \textbf{return} = b_0)).$$

To preserve the soundness invariant, the framework adds the following obligation to all classes that implement IAccount:

$$\mathsf{IAccount} \rightsquigarrow \mathsf{getBal} : (bal = b_0, bal = b_0 \wedge \textbf{return} = b_0).$$

In this case the obligation is added to $\mathcal{U}(\mathsf{Account})$. This obligation is resolved by a straightforward proof outline for the following method addition to Account:

$$\mathit{newMtd}(\mathsf{Account}, \mathsf{Int\ getBal()\{}\textbf{return}\ \mathsf{bal}\}).$$

8 Related Work

Pierik and de Boer [13] present a sound and complete proof outline logic for object-oriented programs. This work is based on a closed world assumption, meaning that the class hierarchy is not open for incremental extensions. To support object-oriented design, proof systems should be constructed for incremental (or modular [3]) reasoning. Most prominent in that context are approaches

Modularity level	Adaptations
1. Class local	*newCls newImpl*
2. Below class	*newFld*, *remFld*, *newMtd*, *remMtd*, *setSup*
3. Global – implements clause	*remInt*, *newConstr*, *newSup*
4. Global – implementation	*remCls*, *remImpl*, *remConstr*, *remSup*

Fig. 5. Basic adaptations classified by modularity. Category 1 needs access to a single class; category 2 to the hierarchy below the adapted class; category 3 needs global access to all implements clauses; and category 4 needs global access to all implementations.

based on *behavioral subtyping* [8]. Relaxing this approach, *lazy behavioral subtyping* [4,5] facilitates incremental reasoning while allowing more flexible code reuse than traditional behavioral subtyping. We refer to [4,5] for a more comprehensive discussion on incremental reasoning about class extensions.

We have found few systems for the analysis of general class modifications. Two widely discussed topics within model transformations in the context of model-driven development are refactoring and refinement. Different approaches, e.g., [9, 10, 17], discuss how to preserve behavioral consistency between different model versions when refinement or refactoring is applied. Program transformations, such as *verification refactoring* [19], may be applied in to reduce program complexity and facilitate verification, e.g., to reduce the size of verification conditions. Contract-based software evolution of aspect-oriented programs is considered in [16], formalizing standard refactoring steps.

Going beyond behavior preserving transformations, *slicing techniques* [18] may be used to describe the effect of updates to determine which properties are preserved and which are potentially invalidated in the new version.

9 Conclusion

As programs evolve, reasoning frameworks for behavioral analysis need to handle shifting proof obligations for different program units. This paper has presented the building blocks of such a framework for object-oriented software evolution. The framework is able to handle program evolution by separating the guarantees of a method from the requirements imposed by calls to the method. The paper formulates a soundness invariant which enables a flexible interleaving of software evolution and reasoning actions by tracking unresolved proof obligations for each class. Software evolution is captured by basic adaptations on existing programs. We describe how each adaptation maintains the soundness invariant and unresolved obligations are tracked automatically. Fig. 5 summarizes the level of modularity supported by the different basic adaptations, a high degree of required global knowledge indicates that the operations may be complex to support in practice. For instance, removing interface constraints may have severe impact on software systems. Proof obligations are resolved by program analysis, and the soundness invariant ensures the overall soundness of the performed analysis when no unresolved proof obligations remain. Whereas many behavioral

restrictions to software evolution make sense for manual proof, it is interesting to see to what extent advanced verification systems can be used to alleviate these restrictions by tracking constraints in a more general way. A full formalization of the framework, implementation, and soundness proofs are future work.

Acknowledgment. We thank Olaf Owe for valuable discussions of this work.

References

1. D. Clarke, *et al.*, Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In *Formal Methods for the Design of Computer, Communication and Software Systems (SFM'11)*, *LNCS* 6659, Springer, 2011.
2. F. Damiani, J. Dovland, E. B. Johnsen, and I. Schaefer. Verifying traits: A proof system for fine-grained reuse. In *Proc 13th Workshop on Formal Techniques for Java-like Programs (FTfJP'11)*, pages 8:1–8:6. ACM, 2011.
3. K. K. Dhara and G. T. Leavens. Forcing behavioural subtyping through specification inheritance. In *18th Conf. on Software Engineering*. IEEE Press, 1996.
4. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.
5. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. *Science of Computer Programming*, 76(10):915–941, 2011.
6. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.
7. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
8. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, Nov. 1994.
9. S. Marković and T. Baar. Refactoring ocl annotated uml class diagrams. *Software and Systems Modeling*, 7:25–47, 2008.
10. T. Massoni, R. Gheyi, and P. Borba. Synchronizing model and program refactoring. In *Formal Methods: Foundations and Applications*, *LNCS* 6527. Springer, 2011.
11. T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
12. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.
13. C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.
14. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPL*, LNCS 6287. Springer, 2010.
15. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In *5th Intl. Conf. on Software Reuse (ICSR5)*, pages 206–215. IEEE Press, 1998.
16. N. Ubayashi, J. Piao, S. Shinotsuka, and T. Tamai. Contract-based verification for aspect-oriented refactoring. In *Proc. Intl. Conf. on Software Testing, Verification, and Validation*, pages 180–189. IEEE Press, 2008.
17. R. Van Der Straeten, V. Jonckers, and T. Mens. A formal approach to model refactoring and model refinement. *Software and Sys. Modeling*, 6:139–162, 2007.
18. H. Wehrheim. Slicing techniques for verification re-use. *Theoretical Computer Science*, 343(3):509 – 528, 2005.
19. X. Yin, J. Knight, and W. Weimer. Exploiting refactoring in formal verification. In *Proc. Dependable Systems and Networks (DSN'09)*, IEEE Press, 2009.