A Proof System for Adaptable Class Hierarchies ☆

Johan Dovland, Einar Broch Johnsen, Olaf Owe, Ingrid Chieh Yu

Department of Informatics, University of Oslo, Norway

Abstract

The code base of a software system undergoes changes during its life time. For object-oriented languages, classes are adapted, e.g., to meet new requirements, customize the software to specific user functionalities, or refactor the code to reduce its complexity. However, the adaptation of class hierarchies makes reasoning about program behavior challenging; even classes in the middle of a class hierarchy can be modified. This paper develops a proof system for analyzing the effect of operations to adapt classes, in the context of method overriding and late bound method calls. The proof system is *incremental* in the sense that reverification is avoided for methods that are not explicitly changed by adaptations. Furthermore, the possible adaptations are not unduly restricted; i.e., flexibility is retained without compromising on reasoning control. To achieve this balance, we extend the mechanism of *lazy behavioral subtyping*, originally proposed for reasoning about inheritance when subclasses are added to a class hierarchy, to deal with the more general situation of adaptable class hierarchies and changing specifications. The reasoning system distinguishes *guaranteed* method behavior from *requirements* toward methods, and achieves incremental reasoning by tracking guarantees and requirements in adaptable class hierarchies. We show soundness of the proposed proof system.

Key words: Software evolution; object orientation; verification; proof systems; class updates; dynamic code modification; adaptable class hierarchies.

1. Introduction

An intrinsic property of software in the real world is that it needs to evolve. This can be during the initial *development* phase, as *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]. As the code is enhanced and modified, it becomes more complex and drifts away from its

☆This work was done in the context of the EU project FP7-610582 *ENVISAGE: Engineering Virtualized Services* (`http://www.envisage-project.eu`).

Email addresses: `johand@ifi.uio.no` (Johan Dovland), `einarj@ifi.uio.no` (Einar Broch Johnsen), `olaf@ifi.uio.no` (Olaf Owe), `ingridcy@ifi.uio.no` (Ingrid Chieh Yu)

original design [2]. 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 and cheaper to modify without changing its observable behavior [3]. In this paper, the term *adaptable class hierarchies* covers transformations of classes during the development, improvement, customization, and refactoring of class hierarchies. Adaptations are achieved by means of update operations for adding code to a class such as new fields, method definitions, and implements clauses, by modifying method definitions, and by removing methods (under certain conditions).

Reasoning about properties of object-oriented systems is in general non-trivial due to complications including class inheritance and late binding of method calls. Object-oriented software development is based on an open world assumption; i.e., class hierarchies are typically extendable. In order 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 a *modular reasoning* framework, a new subclass can be analysed in the context of its superclasses, such that the properties of the superclasses 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 reasoning framework for class hierarchies is *behavioral subtyping* [4]. However, behavioral subtyping has been criticized for being overly restrictive and is often violated in practice [5]. For these reasons it is interesting to explore alternative approaches which allow more flexibility, although this may lead to more proof work

Incremental reasoning frameworks generalize modular reasoning by possibly generating new verification conditions for superclasses in order to guarantee established properties. Additional properties may be established in the superclasses after the initial analysis, but old properties remain valid. Observe that these frameworks subsume modularity: if the initial properties of the classes are sufficiently strong (for example by adhering to a behavioral contract), it will never be necessary to add new properties later. *Lazy behavioral subtyping* is a formal framework for such incremental reasoning, which allows more flexible code reuse than modular frameworks. The basic idea underlying lazy behavioral subtyping is a separation of concerns between the behavioral *guarantees* of method definitions and the behavioral *requirements* to method calls. Both guarantees and requirements are manipulated through an explicit proof-context: a book-keeping framework controls the analysis and 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 while still ensuring soundness.

Lazy behavioral subtyping 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 [6], but has later been extended to multiple inheritance

$$
\begin{array}{ll}
T & ::= I \mid \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Nat} \mid \mathsf{Double} \\
M & ::= MS\ \{\overline{T}\ \overline{x}; t; \textbf{return}\ e\} \\
v & ::= f \mid x
\end{array}
\qquad
\begin{array}{ll}
K & ::= \textbf{interface}\ I\ \textbf{extends}\ \overline{I}\ \{\overline{MS}\} \\
L & ::= \textbf{class}\ C\ \textbf{extends}\ C\ \textbf{implements}\ \overline{I}\ \{\overline{T}\ \overline{f}; \overline{M}\ \overline{MS}\} \\
MS & ::= [T \mid \mathsf{Void}]\ m\ (\overline{T}\ \overline{x}) : (a, a) \\
t & ::= t; t \mid v := rhs \mid e.m(\overline{e}) \mid \textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ t\ \textbf{fi} \mid \textbf{skip} \\
rhs & ::= \textbf{new}\ C(\,) \mid e.m(\overline{e}) \mid m(\overline{e}) \mid e
\end{array}
$$

Figure 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 a is an assertion. Vector notation denotes collections, as in the expression list $\overline{e}$, interface list $\overline{I}$ (with a slight abuse of notation, we write $\overline{T}\ \overline{x}$ to denote a list of variable declarations), and in sets such as $\overline{K}, \overline{L}, \overline{MS}$ and $\overline{M}$. Let *nil* denote the empty list. To distinguish assignments from equations in specifications and expressions, we use $:=$ and $=$ respectively.

[7] and to trait-based code reuse [8].

Adaptable class hierarchies add a level of complexity to proof systems for object-oriented programs, as superclasses in the middle of a class 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 in the case of extending a class hierarchy at the bottom with new subclasses. This paper extends the approach of lazy behavioral subtyping to allow incremental reasoning about adaptable class hierarchies.

The approach is presented using a small object-oriented language and a number of *basic update operations* for adapting class definitions. We consider a series of "snapshots" of a class hierarchy during a development and adaptation process. The developer applies basic update operations and analysis steps to the class hierarchy between these snapshots. Based on the lazy behavioral subtyping framework, the specified and required properties of method definitions and method calls are tracked through these adaptation steps.

The paper is structured as follows: Section 2 presents the language considered and provides a motivating example for update operations. Section 3 presents our program logic for classes, based on proof outlines. Section 4 presents lazy behavioral subtyping and the verification environments needed for analyzing class adaptations. We explain the proof system for the basic update operations in Section 5 and finally, related work is discussed in Section 6 before we conclude the paper in Section 7.

2. The Programming Language

We consider an object-oriented kernel language akin to Featherweight Java [9], in which pointers are typed by behavioral interfaces and methods are annotated with pre/postconditions, The syntax for classes and interfaces in the language is given in Figure 1. A program consists of a set of interfaces and class definitions, followed by a method body. A behavioral interface I extends a list $\overline{I}$ of superinterfaces and declares a set $\overline{MS}$ of method signatures with behavioral annotations. We refer to the behavioral annotation occurring in a method

signature as a *guarantee* for the method. These guarantees reflect semantic constraints on the use of the declared methods, and are given as pairs (p, q) of pre- and postconditions to the signatures. An interface may introduce additional constraints to methods already declared in superinterfaces. A class may inherit from a single superclass, adding fields $\overline{f}$, methods $\overline{M}$, and method guarantees $\overline{MS}$.

Method parameters are read-only, and we assume that programs are well-typed. A method body consists of a sequence of standard statements followed by **return** e, where e is the value returned by the method. Methods of type Void need no return statements and are called without assignment to actual parameters.

The language is based on the notion of *programming to interfaces* [10] and distinguishes interfaces, which describe the usage of objects, from classes, which describe their implementation. We let both objects and pointers (references) to objects be typed by interfaces. If a class C implements an interface I then its instances may be typed by I; we say that instances of C *support* I. However, a subclass of C need not implement I. Remark that both subclasses and subinterfaces may also declare additional guarantees for inherited methods. Different classes may provide different implementations of the same interface. The language does not support remote access of variables since this would break with the interface encapsulation. A class C may implement several interfaces, so variables typed by different interfaces may refer to the same object.

The language provides substitutability at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subinterface of I* in a context depending on I. This substitutability is reflected in the semantics by the fact that late binding applies to all method calls, as the runtime class of an object reference is not in general statically known.

Adaptable Class Hierarchies. Given a program in the above language, we consider a class adaptation mechanism inspired by invasive composition operators such as the runtime class upgrades [11] of Creol [12] and delta-oriented programming [13]. A program can evolve by either adding new elements to the structure of the program (i.e., interfaces or classes) or by modifying the existing structural elements. For simplicity we will not consider the modification of interfaces, and focus on a class adaptation mechanism which may introduce new functionality and interfaces to classes, modify and relocate methods, and remove legacy code. The adaptation mechanism is based on the following *basic update operations* U:

$$U \quad ::= \quad K \mid L \mid \quad \textbf{modify}\ C\ \textbf{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M\ MS}\} \mid \textbf{simplify}\ C\ \{\overline{m}\}.$$

The first two operations express the open environment in which our reasoning occurs and allow an existing program to be extended with new interfaces and classes. The **modify** operation extends an existing class with new functionality by allowing the class to implement additional interfaces. In addition, new fields, methods, and guarantees can be added, and existing methods can be redefined. The **simplify** operation removes redundant methods from an existing class. Observe that in contrast to a class addition, the *class modification*

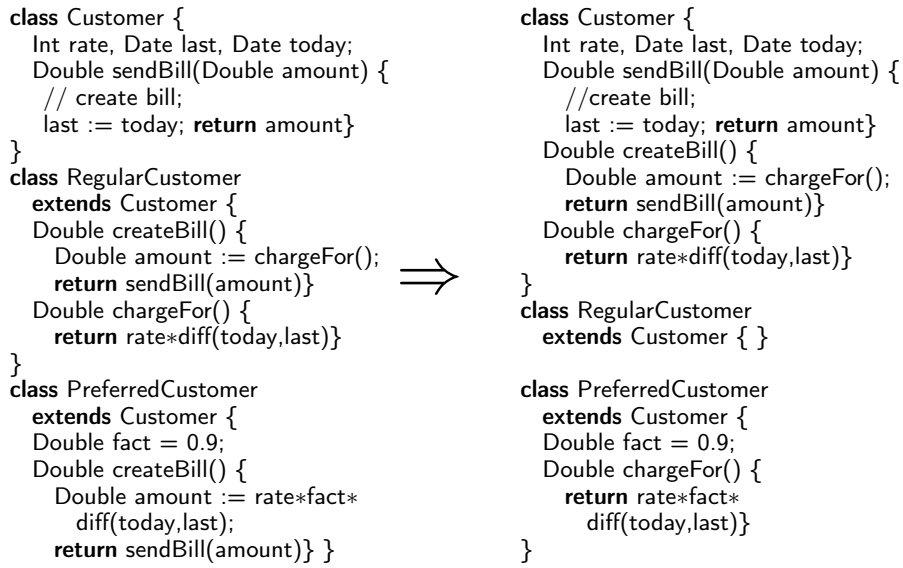

Figure 2: Refactoring Customer code. The original code is shown to the left, and the code after pulling up createBill is shown to the right. For two dates d1 and d2, the operation diff(d1,d2) is assumed to return the number of days between the two dates.

operations **modify** and **simplify** may be applied to classes in the middle of an existing class hierarchy. In this paper we assume that basic update operations are type safe; the type system of Creol [14, 11] guarantees that runtime type errors do not occur for well-typed basic update operations and, in particular, that method binding always succeeds.

We illustrate how basic update operations can be used by an example of *Pull Up Method* refactoring [3], where common subclass functionality is moved to a shared superclass method.

Example 1. Consider a Customer class with two subclasses: regular customer and preferred customer. The code is given on the left hand side of Figure 2; both subclasses feature similar methods createBill. We consider adaptations of the class hierarchy such that createBill becomes a method of the Customer class. For type safety, we also need to pull up chargeFor. Observe that it is not possible to refactor all the subclass behavior into the superclass since the charged amount of PreferredCustomer should be reduced by the factor fact. The result of these modifications is shown on the right hand side of Figure 2, which can be derived by applying the following *sequence* of basic update operations to the original code:

```
modify Customer {
  Double createBill() { Double amount := chargeFor(); return sendBill(amount)}
  Double chargeFor() { return rate∗diff(today,last)}}
simplify RegularCustomer {createBill, chargeFor}
```

modify PreferredCustomer {Double chargeFor() {**return** rate∗fact∗diff(today,last)}}
simplify PreferredCustomer {createBill}

For the original code (on the left hand side of Figure 2), we may consider the following guarantees for the two versions of createBill. For the implementation in class RegularCustomer, we have the following pair of pre- and postconditions:

$$(\mathsf{last} = l_0,\ \mathsf{result} = \mathsf{diff}(\mathsf{today}, l_0) * \mathsf{rate} \wedge \mathsf{last} = \mathsf{today}) \tag{1}$$

and for the implementation in PreferredCustomer we have:

$$(\mathsf{last} = l_0, \mathsf{result} = \mathsf{diff}(\mathsf{today}, l_0) * \mathsf{rate} * \mathsf{fact} \wedge \mathsf{last} = \mathsf{today}) \tag{2}$$

Here l_0 is a logical variable which is used to freeze the initial value of last and result is used as the name of the logical variable storing the return value. Assuming that these pre- and postconditions have been verified for the original code, we return to them in Example 2 which illustrates how they are maintained by the proof system, showing that these guarantees will still hold after the refactoring.

3. A Program Logic for Classes

Consider a program logic for classes in this language based on proof outlines for method definitions. The purpose of the program logic is to ensure that a class behaves according to the semantic constraints of its behavioral interfaces. This is done in two steps: first, it is shown that the methods in a class behave according to their guarantees and, second, it is shown that the method guarantees in the class imply those of its declared interfaces. Typically, the guarantees of the public methods stem from the interfaces, and start the verification process. Each proof outline asserts the correctness of a method guarantee in the class. A characteristic feature of our approach is that a method may have *several* guarantees; i.e., a method has a set of associated behavioral guarantees and associated proof outlines. In this section we do not consider how classes behaviorally relate to each other; the extension from individual classes to a class hierarchy is considered in Section 4.

Apart from the treatment of method calls, the program logic uses a proof system which follows standard proof rules [15, 16] for partial correctness, adapted to the object-oriented setting; in particular, de Boer's technique using sequences in the assertion language addresses the issue of object creation [17]. We present the proof system using Hoare triples $\{p\}\ t\ \{q\}$ [18], for assertions p and q, and a statement sequence t. Here, p is the precondition and q is the postcondition to t. A triple $\{p\}\ t\ \{q\}$ is given a standard partial correctness interpretation: if t is executed in a state where p holds and the execution terminates, then q holds 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.

Assertions. We consider an expression language defined by

$$a ::= \mathsf{this} \mid \mathsf{result} \mid \mathsf{null} \mid f \mid x \mid z \mid op(\overline{a}).$$

Here, this denotes the current object, result the current method's return value, f a field, x a local variable (including formal parameters), z a logical variable, and op an operation on data types. In particular, equality of x and y is denoted $x = y$. *Assertions* are the Boolean typed subset of the given expression language. Thus, an *assertion pair* (p, q) (of type APair) is a pair of Boolean expressions such that p is a precondition and q a postcondition (for some sequence of program statements).

Proof outlines. In a *proof outline* [19] for a triple $\{p\}\ t\ \{q\}$, the statement sequence t is decorated with assertions. The main idea is to decorate different program points in t with assertions such that the analysis between the different program points can be done mechanically, ensuring $\vdash_{PL} \{p\}\ t\ \{q\}$. A classical example is to decorate loops with loop invariants. For the purposes of this paper, we are mainly interested in decorating method calls with pre- and postconditions; i.e., the proof outline assumes a certain behavior from the called methods in order to ensure the guarantee of the considered method. By standard techniques, a decorated method call $\{r\}\ v := n(\overline{e})\ \{s\}$ may be formulated as a proof obligation $\{r'\}\ n(\overline{x})\ \{s'\}$ towards the called method n, where $r' = r[\overline{z}_0/\overline{z}] \wedge \overline{x} = \overline{e}[\overline{z}_0/\overline{z}]$ and $s' = s[\mathsf{result}/v][\overline{z}_0/\overline{z}]$, where $\overline{z}$ are the variables local to the caller (including formal parameters), and $\overline{z}_0$ is a list of fresh logical variables. For remote calls, we also need to replace this by a fresh variable name. Following the idea of programming to interfaces, the validity of these proof obligation for remote calls to an object typed by interface I can be checked by inspecting the guarantees of the methods in I. However, this is not the case for internal calls, which are late bound according to the class hierarchy.

Let the notation $O \vdash_{PL} t : (p, q)$ denote that O is a (valid) proof outline proving that the *guarantee* (p, q) holds for a body t; i.e., $S \vdash_{PL} \{p\}\ O\ \{q\}$ holds when assuming that the set S consists of annotations for all the internal method calls in O. Pre- and postconditions for internal late bound calls are called *requirements*, and we assume that these are of the form $\{r'\}\ n(\overline{x})\ \{s'\}$ as described above. Assuming no method overloading, requirements are written $\{r'\}\ n\ \{s'\}$ of type Req.

Method specifications. A method specification is a tuple $\langle m, (p, q), O\rangle$ of type MSpec, where m : Mid, (p, q) : APair is a guarantee and O is a proof outline for $t : (p, q)$ where t is the body of method m. Method specifications may be decomposed by the functions *mid* : MSpec $\to$ Mid, *guar* : MSpec $\to$ APair, and *reqs* : MSpec $\times$ Mid $\to$ Set[APair], such that $mid(\langle m, (p, q), O\rangle) \triangleq m$, $guar(\langle m, (p, q), O\rangle) \triangleq (p, q)$, and $reqs(\langle m, (p, q), O\rangle, n)$ returns the set of all requirements (r, s) for $\{r\}\ n\ \{s\} \in O$. These functions are straightforwardly lifted to sets of specifications. Specification projection: $\_/\_$: Set[MSpec] $\times$ Set[Mid] $\to$ Set[MSpec] is defined in Figure 3, such that $\overline{MSP}/\overline{m}$ restricts $\overline{MSP}$ to specifications of methods $\overline{m}$.

Entailment. By the notion of lazy behavioral subtyping, a method may be given several specifications. Especially, inherited superclass methods may be given additional specifications in subclasses which override internally called methods. Dealing with sets of assertion pairs, the standard consequence rule of Hoare Logic [15] is insufficient. We therefore define an entailment relation which allows us to combine information from several assertion pairs. Let p^o denote an expression p with all occurrences of program fields f substituted by the corresponding variables f^o, avoiding name capture. The assertion pair (p, q) is understood as an input/output relation $\forall \overline{z} \;.\; p \Rightarrow q^o$, where f and f^o denotes the input and output values of f, respectively, and $\overline{z}$ are the logical variables in p and q. The entailment relation is defined for assertion pairs and for sets of assertion pairs:

Definition 1. ***(Entailment.)*** *Let (p, q) and (r, s) be assertion pairs and let $\mathcal{X}$ and $\mathcal{Y}$ denote the sets $\{(p_i, q_i) \,|\, 1 \leq i \leq n\}$ and $\{(r_i, s_i) \,|\, 1 \leq i \leq m\}$, respectively.* Entailment *is defined by*

1. $(p, q) \twoheadrightarrow (r, s) \triangleq (\forall \overline{z}_1 \;.\; p \Rightarrow q^o) \Rightarrow (\forall \overline{z}_2 \;.\; r \Rightarrow s^o)$,
   where $\overline{z}_1$ and $\overline{z}_2$ are the logical variables in (p, q) and (r, s), respectively.
2. $\mathcal{X} \twoheadrightarrow (r, s) \triangleq (\bigwedge_{1 \leq i \leq n} (\forall \overline{z}_i \;.\; p_i \Rightarrow q_i^o)) \Rightarrow (\forall \overline{z} \;.\; r \Rightarrow s^o)$.
3. $\mathcal{X} \twoheadrightarrow \mathcal{Y} \triangleq \bigwedge_{1 \leq i \leq m} \mathcal{X} \twoheadrightarrow (r_i, s_i)$.

The relation $\mathcal{X} \twoheadrightarrow (r, s)$ corresponds to classic Hoare style reasoning, proving $\{r\}\ t\ \{s\}$ from $\{p_i\}\ t\ \{q_i\}$ for all $1 \leq i \leq n$, by means of the consequence and conjunction rules [15]. Note that when proving entailment, program fields (input and output) are implicitly universally quantified. Furthermore, entailment is reflexive and transitive, and $\mathcal{Y} \subseteq \mathcal{X}$ implies $\mathcal{X} \twoheadrightarrow \mathcal{Y}$.

4. A Lazy Behavioral Subtyping Framework for Basic Update Operations

Lazy behavioral subtyping [6, 7] is an extension of the behavioral subtyping approach [4] to reason about programs with late bound method calls in a more flexible way. Thus, these approaches are concerned with how the set of assumptions in a valid proof outline can be discharged when the classes are assembled into of a class hierarchy. Both approaches share the goal to avoid reverification and thus to support incremental reasoning about classes under an open-word assumption: when analyzing a class, it suffices to consider that class and its superclasses. In contrast to traditional behavioral subtyping, lazy behavioral subtyping avoids reverification without restricting method overriding to fully behavior-preserving redefinition. The formalizations in [6, 7] ensure that all method specifications required by late bound calls remain satisfied when new classes *extend* a class hierarchy. In this section, we extend the lazy behavioral subtyping framework to handle basic update operations.

In order to avoid reverification, the essential idea behind behavioral subtyping is that a method overriding must adhere to the specification of the overridden

method, as declared by, e.g., the pre- and postconditions to the method body. In contrast, lazy behavioral subtyping relaxes this idea: a method overriding must adhere to the *required* behavior of the method. Given a method m specified by a precondition p and a postcondition q, there is no need to restrict the behavior of methods overriding m and require that these adhere to that specification. Instead it suffices to preserve the "part" of p and q that is actually *used to verify* the program at the current stage. Specifically, if m is used in the program in the form of a method call $\{r\}\ m(\ldots)\ \{s\}$, the pre- and postconditions r and s at that call-site constitute m's *required* behavior. It is these weaker conditions that need to be preserved in order to avoid reverification. For a method of some class, both guarantees and requirements are given as sets of pre- and post condition pairs. The approach ensures that the required behavior will follow from the guarantees (by the entailment relation of Definition 1). Lazy behavioral subtyping formalizes this idea as a calculus which lazily imposes context-dependent subtyping constraints on method definitions and stores these constraints in an analysis environment which reflects the analyzed part of the class hierarchy. The presentation is based on a Hoare style proof system.

When analyzing basic update operations, it is no longer sufficient to consider only one class and its superclasses, since updates may apply to classes which already have (analyzed) subclasses present in the hierarchy. Consider some method m, originally equipped with a pre/post specification. As the system evolves, methods that call m may be modified such that m is called with weaker requirements, and calls to m may even disappear. If m is later changed, the new version need not adhere to the original specification of m; the new version only needs to respect the behavior required by call-sites *at the time* when the modification of m is performed.

In order to reason incrementally about basic update operations, we introduce two verification environments. The *proof* environment is used to maintain verified properties of the current class hierarchy. Soundness is defined for proof environments such that for each class, the verified guarantees are strong enough to satisfy the constraints of the implemented interfaces. Furthermore, the requirements imposed by the analysis of each guarantee follow from the guarantees of the implementations to which each call can bind. Performing an update operation may break soundness; e.g., method implementations may be modified, implementations may disappear, or new implementations may appear. Consequently, there are certain conditions that must be checked after an update operation in order to re-establish soundness. The purpose of the *update* environment is to keep track of these potentially unsound properties.

4.1. The Proof Environment

The proof environment captures the class hierarchy, and maintains verified method specifications. Intuitively, the proof environment consists of three parts: the class definitions, the interface definitions, and the (verified) method specifications per class. Formally, proof environments are defined as follows:

Definition 2. ***(Proof environments.)*** *A* proof environment $\mathcal{E}$ *of type* Env *is*

$$
\begin{array}{rcl}
body_{\mathcal{E}}(C,m) & \triangleq & \begin{cases} t & \text{if } m(\overline{x})\{t\} \in C.mtds \\ body_{\mathcal{E}}(C.inh,m) & \text{if } m \notin C.mtds \end{cases} \\
\emptyset/\overline{m} & \triangleq & \emptyset \\
(\langle m',(p,q),O\rangle \cup \overline{MSP})/\overline{m} & \triangleq & \begin{cases} \langle m',(p,q),O\rangle \cup \overline{MSP}/\overline{m} & \text{if } m' \in \overline{m} \\ \overline{MSP}/\overline{m} & \text{otherwise} \end{cases} \\
G_{\mathcal{E}}(C,m) & \triangleq & guar(S_{\mathcal{E}}(C)/m) \\
G{\uparrow}_{\mathcal{E}}(C,m) & \triangleq & \begin{cases} G_{\mathcal{E}}(C,m) & \text{if } m \in C.mtds \\ G_{\mathcal{E}}(C,m) \cup G{\uparrow}_{\mathcal{E}}(C.inh,m) & \text{otherwise} \end{cases} \\
R_{\mathcal{E}}(C,m) & \triangleq & reqs(S_{\mathcal{E}}(C),m) \\
R{\uparrow}_{\mathcal{E}}(nil,m) & \triangleq & \emptyset \\
R{\uparrow}_{\mathcal{E}}(C,m) & \triangleq & R_{\mathcal{E}}(C,m) \cup R{\uparrow}_{\mathcal{E}}(C.inh,m) \quad \text{for } C \neq nil \\
mids(\emptyset) & \triangleq & \emptyset \\
mids(m(\overline{x})\{t\} \cup \overline{M}) & \triangleq & m \cup mids(\overline{M}) \\
spec_{\mathcal{E}}(\mathrm{nil},m) & \triangleq & \emptyset \\
spec_{\mathcal{E}}((I;\overline{I}),m) & \triangleq & spec_{\mathcal{E}}(I,m) \cup spec_{\mathcal{E}}(\overline{I},m) \\
spec_{\mathcal{E}}(I,m) & \triangleq & spec_{\mathcal{E}}(\overline{MS},m) \cup spec_{\mathcal{E}}(\overline{I},m) \\
 & & \text{where } K_{\mathcal{E}}(I) = \langle \overline{I},\overline{MS}\rangle \\
spec_{\mathcal{E}}((m'\colon(p,q);\overline{MS}),m) & \triangleq & \begin{cases} (p,q) \cup spec_{\mathcal{E}}(\overline{MS},m) & \text{if } m = m' \\ spec_{\mathcal{E}}(\overline{MS},m) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 3: Auxiliary functions. Here, $\overline{MSP} : \mathsf{Set}[\mathsf{MSpec}]$.

a tuple $\langle L_{\mathcal{E}}, K_{\mathcal{E}}, S_{\mathcal{E}}\rangle$*, where* $L_{\mathcal{E}} : \mathsf{Cid} \to \mathsf{Class}$ *and* $K_{\mathcal{E}} : \mathsf{Iid} \to \mathsf{Interface}$ *are partial mappings and* $S_{\mathcal{E}} : \mathsf{Cid} \to \mathsf{Set}[\mathsf{MSpec}]$ *is a total mapping.*

For a class definition **class** C **extends** D **implements** $\overline{I}$ $\{\overline{f};\ \overline{M}\ \overline{MS}\}$ (ignoring types), the name C maps to the tuple $\langle D, \overline{I}, \overline{f}, \overline{M}\rangle$ of type Class by the mapping L. A class tuple may be decomposed by observer functions *inh*, *impl*, *fields*, and *mtds*; e.g., $L_{\mathcal{E}}(C).impl = \overline{I}$ for the class above. For brevity, we often write $C.impl$ instead of $L_{\mathcal{E}}(C).impl$ if it is clear from the context that C is in $\mathcal{E}$. Furthermore, we assume a function $sub : \mathsf{Cid} \to \mathsf{Set}[\mathsf{Cid}]$, such that $C.sub$ returns the set of names for all classes D such that $D.inh = C$; i.e., the names of all the direct subclasses of C (and $\emptyset$ if C has no subclasses). We let $C.sub{\downarrow}$ extend this function such that $D \in C.sub{\downarrow}$ if D directly or indirectly inherits C.

The mapping K maps an interface name to a tuple $\langle \overline{I}, \overline{MS}\rangle$ of type Interface. The function $spec : \mathsf{List}[\mathsf{Iid}] \times \mathsf{Mid} \to \mathsf{Set}[\mathsf{APair}]$ is defined in Figure 3 such that $spec(\overline{I}, m)$ returns the declared constraints of m in $\overline{I}$. We assume a function $public : \mathsf{List}[\mathsf{Iid}] \to \mathsf{Set}[\mathsf{Mid}]$ such that $m \in public(I)$ if I (or a superinterface of I) declares m. Additional functions are given in Figure 3. The function $body(C, m)$ returns the implementation of m as found in class C, or inherited from a superclass of C. The mapping S maps a class name to a set of method specifications $\langle m, (p, q), O\rangle$. The function $G_{\mathcal{E}}(C, m)$ returns the guarantees for

m found in $S_{\mathcal{E}}(C)$, and $G{\uparrow}_{\mathcal{E}}(C,m)$ returns the guarantees for m up to the first class above C where m is defined. Similarly, the function $R_{\mathcal{E}}(C,m)$ returns the requirements towards m imposed by proof outlines in $S_{\mathcal{E}}(C)$, and $R{\uparrow}_{\mathcal{E}}(C,m)$ returns the union of all $R_{\mathcal{E}}(B,m)$ for B above C. For brevity, we let $m \in C.mtds$ abbreviate $m \in mids(C.mtds)$.

Extending the proof environment.. Proof environment modifications are represented by the operator $\_ \star \_ : \mathsf{Env} \times \mathsf{Mod} \to \mathsf{Env}$, where the first argument is the current proof environment and the second argument is the modification. The modifications $extL$ and $extS$ extend the environment with a class definition and specification, respectively:

$$\begin{array}{rcl} \mathcal{E} \star extL(C, D, \overline{I}, \overline{f}, \overline{M}) & \triangleq & \langle L_{\mathcal{E}}[C \mapsto \langle D, \overline{I}, \overline{f}, \overline{M}\rangle], K_{\mathcal{E}}, S_{\mathcal{E}}\rangle \\ \mathcal{E} \star extS(C, m, (p,q), O) & \triangleq & \langle L_{\mathcal{E}}, K_{\mathcal{E}}, S_{\mathcal{E}}[C \mapsto S_{\mathcal{E}}(C) \cup \langle m, (p,q), O\rangle]\rangle \end{array}$$

Next we define sound classes and sound environments.

Definition 3. ***(Sound environments.)*** *Given a proof environment $\mathcal{E} : \mathsf{Env}$ and $C : \mathsf{Cid}$ such that $C \in \mathcal{E}$, we say that C* is sound in $\mathcal{E}$ *if the following conditions are satisfied:*

1. $\forall \langle m, (p,q), O\rangle \in S_{\mathcal{E}}(C)\ .\ O \vdash_{PL} body_{\mathcal{E}}(C,m) : (p,q)$
   $\wedge\ \forall \{r\}\ e.n\ \{s\} \in O\ .\ \forall I\ .\ e : I \Rightarrow spec_{\mathcal{E}}(I,n) \twoheadrightarrow (r,s)$
2. $\forall m\ .\ G{\uparrow}_{\mathcal{E}}(C,m) \twoheadrightarrow R{\uparrow}_{\mathcal{E}}(C,m)$
3. $\forall \overline{I}\ .\ \overline{I} = L_{\mathcal{E}}(C).impl\ .\ \forall n \in public_{\mathcal{E}}(\overline{I})\ .\ G{\uparrow}_{\mathcal{E}}(C,n) \twoheadrightarrow spec_{\mathcal{E}}(\overline{I},n)$

We say that an environment $\mathcal{E}$ is sound if C is sound in $\mathcal{E}$ for all classes $C \in \mathcal{E}$.

The first condition expresses that all recorded specifications have verified proof outlines, and the second condition ensures that all inherited requirements follow from the specifications of each implemented method. The third condition ensures that, for each class, the constraints of the implemented interfaces follow from the verified guarantees. The definition ensures that whenever $body_{\mathcal{E}}(C,m)$ is executed on an instance of a subclass of C, all guarantees contained in $G{\uparrow}_{\mathcal{E}}(C,m)$ hold. Especially, the requirements imposed by these guarantees are satisfied for the implementations to which the calls will bind.

The top part of Figure 4 defines operations that manipulate the proof environment for the analysis of class modifications. For a proof environment $\mathcal{E}$, we let $\mathcal{E} \star mod$ denote the update of $\mathcal{E}$ by the modification mod (thus, $\star$ is an infix update-operator). The modification $addL$ extends an existing class with new interfaces, fields, and method definitions. The definition of $addL$ accounts for method redefinitions: if a method m is defined in both the old version of the class and in the modified version, the old definition is removed, and the one from the update operation survives. The modification $remL$ removes the given methods from the considered class. For each method $m \in \overline{m}$ the modification $remS(C, \overline{m})$ removes the specifications of m in $S_{\mathcal{E}}(C)$. In addition, specifications of m in a subclass D of C are removed if D inherits m from C without redefinition. Specification removal $\_ \setminus \_ : \mathsf{Set}[\mathsf{MSpec}] \times \mathsf{Set}[\mathsf{Mid}] \to \mathsf{Set}[\mathsf{MSpec}]$ removes

the specifications for the given methods, and is defined in terms of projection and ordinary set minus.

4.2. The Update Environment

Basic update operations may introduce new requirements when redefining some methods, these requirements must propagate downwards and be verified in the subclasses. Similarly, subclasses may contain requirements toward redefined methods. We introduce an *update environment* $\mathcal{U}$ to capture new specifications and requirements accumulated during the analysis of the basic update operations which remain to be verified by subclasses of the modified class.

Definition 4. ***(Update environments.)*** *An* update environment $\mathcal{U}$ *is a pair* $\langle M_{\mathcal{U}}, R_{\mathcal{U}} \rangle$*, where* $M_{\mathcal{U}} : \mathsf{Cid} \rightarrow \mathsf{Set}[\mathsf{Mid}]$ *and* $R_{\mathcal{U}} : \mathsf{Cid} \rightarrow \mathsf{Set}[\mathsf{Req}]$ *are partial mappings.*

Let $\mathcal{U} = \emptyset$ denote that $M_{\mathcal{U}}(C) = R_{\mathcal{U}}(C) = \perp$ for all $C : \mathsf{Cid}$, where $\perp$ represents that the mapping is undefined. Furthermore, we let $C \in \mathcal{U}$ denote that $M_{\mathcal{U}}(C)$ and $R_{\mathcal{U}}(C)$ are well-defined. Definitions for updating $\mathcal{U}$ can be found in Figure 4.

5. The Proof System for Adaptable Class Hierarchies

In this section we explain the calculus for the basic update operations. First we introduce the inference rules for analyzing new classes, then we address operations for transforming existing class definitions. The introduction of interfaces does not generate proof obligations, and the rules for analyzing interface declarations are omitted for brevity; needed interfaces are always assumed to be included in the environment.

Judgements in the calculus are on the form $\mathcal{U}, \mathcal{E} \vdash \mathcal{M}$, where $\mathcal{U}$ is an *update environment*, $\mathcal{E}$ is a *proof environment*, and $\mathcal{M}$ is a sequence of *analysis operations*. The syntax for the analysis operations is given in Figure 5.

5.1. Extending Class Hierarchies

The calculus for analyzing class addition is given in Figure 6. A new class extends a previously analyzed class hierarchy and can only be added to the bottom of the class hierarchy. Rule newClass extends the proof environment with the new class definition and generates an $anClass(C, \overline{MS}, \overline{M}, \overline{m})$ operation, which will be analysed in the context of the extended proof environment. Given an environment $\mathcal{E}$ and interfaces $\overline{I}$, the auxiliary function $public_{\mathcal{E}}(\overline{I})$ returns the set of names for the methods in $\overline{I}$. The rule for class analysis anClass generates an operation of the form $\langle C : \mathcal{O} \rangle$, where $\mathcal{O}$ is a sequence of analysis operations to be performed for a class C. Rule anClass generates three initial operations *anSpec*, *anReq*, and *newIntSpec*. Operation *anSpec* is analyzed by newSpec, initiating a verification of the user-given specifications with regard to their respective implementations. For each method m defined in C, *anReq* collects the inherited requirements toward m which are analyzed in newMtd.

$$
\begin{array}{rcl}
\mathcal{E} \star addL(C, \overline{I}, \overline{f}, \overline{M}) & \triangleq & \langle L_{\mathcal{E}}[C \mapsto \langle D, \overline{I}'\ \overline{I}, \overline{f}'\ \overline{f}, del(mids(\overline{M}), \overline{M}')\ \overline{M}\rangle], \\
& & \quad K_{\mathcal{E}}, S_{\mathcal{E}}\rangle \quad \text{where } L_{\mathcal{E}}(C) = \langle D, \overline{I}', \overline{f}', \overline{M}'\rangle \\
\mathcal{E} \star remL(C, \overline{m}) & \triangleq & \langle L_{\mathcal{E}}[C \mapsto \langle D, \overline{I}, \overline{f}, del(\overline{m}, \overline{M})\rangle, K_{\mathcal{E}}, S_{\mathcal{E}}\rangle \\
& & \text{where } L_{\mathcal{E}}(C) = \langle D, \overline{I}, \overline{f}, \overline{M}\rangle \\
\mathcal{E} \star remS(C, \overline{m}) & \triangleq & \mathcal{E} \star rem(C, \overline{m}) \star subRemS(C.sub, \overline{m}) \\
\mathcal{E} \star rem(C, \overline{m}) & \triangleq & \langle L_{\mathcal{E}}, K_{\mathcal{E}}, S_{\mathcal{E}}[C \mapsto (S_{\mathcal{E}}(C) \setminus \overline{m})]\rangle \\
\mathcal{E} \star subRemS(C\ \overline{C}, \overline{m}) & \triangleq & \mathcal{E} \star subRemS(C, \overline{m}) \star subRemS(\overline{C}, \overline{m}) \\
\mathcal{E} \star subRemS(C, m\ \overline{m}) & \triangleq & \begin{cases} \mathcal{E} \star subRemS(C, \overline{m}) & \text{if } m \in C.mtds \\ \mathcal{E} \star rem(C, m) & \\ \quad \star\ subRemS(C.sub, m) & \\ \quad \star\ subRemS(C, \overline{m}) & \text{otherwise} \end{cases} \\
\mathcal{E} \star subRemS(\emptyset, \overline{m}) & \triangleq & \mathcal{E} \star subRemS(\overline{C}, \emptyset) \triangleq \mathcal{E} \\
del(\overline{m}, \emptyset) & \triangleq & \emptyset \\
del(\overline{m}, (m(\overline{x})\{t\} \cup \overline{M})) & \triangleq & \begin{cases} del(\overline{m}, \overline{M}) & \text{if } m \in \overline{m} \\ m(\overline{x})\{t\} \cup del(\overline{m}, \overline{M}) & \text{otherwise} \end{cases} \\
\overline{MSP} \setminus \overline{m} & \triangleq & \overline{MSP} \setminus (\overline{MSP}/\overline{m})
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{U} \star init(\emptyset) & \triangleq & \mathcal{U} \\
\mathcal{U} \star init(C\ \overline{C}) & \triangleq & \langle M_{\mathcal{U}}[C \mapsto \emptyset], R_{\mathcal{U}}[C \mapsto \emptyset]\rangle \star init(\overline{C}) \\
\mathcal{U} \star empty(C) & \triangleq & \langle M_{\mathcal{U}}[C \mapsto \bot], R_{\mathcal{U}}[C \mapsto \bot]\rangle \\
\mathcal{U} \star newReq(\emptyset, \{r\}n\{s\}) & \triangleq & \mathcal{U} \\
\mathcal{U} \star newReq(C\ \overline{C}, \{r\}n\{s\}) & \triangleq & \langle M_{\mathcal{U}}, R_{\mathcal{U}}[C \mapsto R_{\mathcal{U}}(C) \cup \{r\}n\{s\}]\rangle \\
& & \quad \star\ newReq(\overline{C}, \{r\}n\{s\}) \\
\mathcal{U} \star newSpec(\overline{C}, m\ \overline{m}) & \triangleq & \mathcal{U} \star newSpec(\overline{C}, m) \star newSpec(\overline{C}, \overline{m}) \\
\mathcal{U} \star newSpec(\emptyset, m) & \triangleq & \mathcal{U} \\
\mathcal{U} \star newSpec(C\ \overline{C}, m) & \triangleq & \begin{cases} \mathcal{U} \star newSpec(\overline{C}, m) & \text{if } m \in C.mtds \\ \langle M_{\mathcal{U}}[C \mapsto M_{\mathcal{U}}(C) \cup m], R_{\mathcal{U}}\rangle & \\ \quad \star\ newSpec(C.sub\ \overline{C}, m) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4: Environment update definitions.

Finally, newIntSpec1 and newIntSpec2 ensure that the implementation of C satisfies the behavioral constraints given in the interfaces $\overline{I}$ of C. Specifically, if m is in a public interface of C, intSpec verifies that the specifications of m in C satisfy the specifications of m in $\overline{I}$, as given by the auxiliary function $spec_{\mathcal{E}}(\overline{I}, m)$. For this verification, we allow users to provide additional guarantees (p, q).

Specifications are verified by the rules reqDer or reqNotDer. The specification of m is discarded by reqDer if it follows from the previously proven specifications of m. Otherwise, reqNotDer leads to the analysis of a new proof outline for m. In such proof outlines, internal calls are handled by call, which ensures that the method definitions to which the call may be bound satisfy

$$
\begin{array}{l}
\mathcal{M} ::= \mathcal{P} \mid \mathcal{C} \cdot \mathcal{P} \\
\mathcal{P} ::= U \mid \mathcal{P} \cdot \mathcal{P} \\
\mathcal{C} ::= \mathit{anClass}(C, \overline{MS}, \overline{m}, \overline{n}) \mid \langle C : \mathcal{O} \rangle \\
\mathcal{O} ::= O \mid O \cdot \mathit{upd}(\overline{m}, \overline{T}) \\
O ::= \epsilon \mid \mathit{anReq}(\overline{m}) \mid \mathit{anSpec}(\overline{MS}) \mid \mathit{newSpec}(m, (p, q)) \mid \mathit{anCalls}(t) \\
\qquad \mid \mathit{verify}(m, \overline{R}) \mid \mathit{intSpec}(\overline{m}) \mid \mathit{newIntSpec}(\overline{m}) \mid O \cdot O \\
L ::= \textsf{class}\ C\ \textsf{extends}\ D\ \textsf{implements}\ \overline{I}\ \{\overline{f}; \overline{M}\ \overline{MS}\} \\
U ::= L \mid \textsf{modify}\ C\ \textsf{implements}\ \overline{I}\ \{\overline{f}; \overline{M}\ \overline{MS}\} \mid \textsf{simplify}\ C\ \{\overline{m}\}
\end{array}
$$

Figure 5: Syntax for analysis operations. Here, $\overline{T}$ denotes a set of requirements $\{r\}\ n\ \{s\}$.

the requirement (r, s). Remark that a requirement $\{r\}\ n\ \{s\}$ is added to the proof environment by Rule reqNotDer as a part of the method specification. Thus, the requirement is included in the set of requirements that is analyzed by an *anReq* operation whenever n is overridden by a subclass. The extension $\mathit{newReq}(C.\mathit{sub}\downarrow, \{r\}n\{s\})$ of $\mathcal{U}$ in Rule call is oriented towards the analysis of class modifications. When analyzing class modifications, requirements for n may be introduced in the middle of the class hierarchy To ensure soundness, these must propagate down the class hierarchy and be verified for those subclasses that override n. Remark that the extension of $\mathcal{U}$ plays no role when analyzing new classes; if C is a leaf class, then $C.\mathit{sub}\downarrow = \emptyset$.

5.2. Modifying Class Definitions

In this section we extend the inference system in Figure 6 with basic update operations **modify** and **simplify** such that any class in a class hierarchy may be subjected to the update operations. The corresponding rules are given in Figure 7. The analysis tracks specified and required properties and limits reverification to methods that are explicitly changed by the basic update operations.

Class extensions are analysed by class-Extend. The proof environment of C is extended with the new interfaces $\overline{I}$, fields $\overline{f}$, and new and redefined methods $\overline{M}$. Moreover, old specifications and requirements associated with the redefined methods are removed from the specification and requirement sets. As subclasses may contain requirements toward methods redefined by the modification of C, the class hierarchy below C needs to be traversed. Consequently, the update environment is extended with method names $\overline{m}$ and subclasses of C. The modified class is then analysed in the context of the new environments $\mathcal{E}'$ and $\mathcal{U}'$. After the analysis of the modified class C, the subclasses of C are traversed by rule newUpd. For each subclass, the operation *upd* is generated with input values given by the accumulated update environment for the subclass. Rule subSpec checks that the requirements are still valid after the class transformation. Similarly, a redefined method may introduce new requirements into $\mathcal{U}$ (in call). Consequently, generated requirements are propagated down to subclasses. If the called method is overridden by a subclass, the requirements is analyzed by subReq. Otherwise, the requirement is discarded by noSubReq.

$$\frac{\overline{I} \in \mathcal{E} \quad C \notin \mathcal{E} \quad D \neq nil \Rightarrow D \in \mathcal{E} \quad \mathcal{U} = \emptyset \quad \mathcal{U}, \mathcal{E} \star extL(C, D, \overline{I}, \overline{f}, \overline{M}) \vdash anClass(C, \overline{MS}, mids(\overline{M}), public_{\mathcal{E}}(\overline{I})) \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \textbf{class } C \textbf{ extends } D \textbf{ implements } \overline{I}\ \{\overline{f}; \overline{M}\ \overline{MS}\} \cdot \mathcal{P}}\ (\textsc{newClass})$$

$$\frac{\mathcal{U}, \mathcal{E} \vdash \langle C :\ anSpec(\overline{MS}) \cdot anReq(\overline{m}) \cdot newIntSpec(\overline{n}) \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash anClass(C, \overline{MS}, \overline{m}, \overline{n}) \cdot \mathcal{P}}\ (\textsc{anClass}) \qquad \frac{\mathcal{U}, \mathcal{E} \vdash \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ \epsilon \rangle \cdot \mathcal{P}}\ (\textsc{empClass})$$

$$\frac{m \in public_{\mathcal{E}}(C.impl) \quad \mathcal{U}, \mathcal{E} \vdash \langle C :\ intSpec(m) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ newIntSpec(m) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{newIntSpec1})$$

$$\frac{m \notin public_{\mathcal{E}}(C.impl) \quad \mathcal{U}, \mathcal{E} \vdash \langle C :\ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ newIntSpec(m) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{newIntSpec2})$$

$$\frac{(G\!\uparrow_{\mathcal{E}}(C, m) \cup \{(p, q)\}) \twoheadrightarrow spec_{\mathcal{E}}(C.impl, m) \quad \mathcal{U}, \mathcal{E} \vdash \langle C :\ verify(m, (p, q)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ intSpec(m) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{intSpec})$$

$$\frac{O \vdash_{PL} body_{\mathcal{E}}(C, m) : (p, q) \quad \mathcal{U}, \mathcal{E} \star extS(C, m, (p, q), O) \vdash \langle C :\ anCalls(O) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ verify(m, (p, q)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{reqNotDer})$$

$$\frac{G\!\uparrow_{\mathcal{E}}(C, m) \twoheadrightarrow (p, q) \quad \mathcal{U}, \mathcal{E} \vdash \langle C :\ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ verify(m, (p, q)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{reqDer}) \qquad \frac{\mathcal{U}, \mathcal{E} \vdash \langle C :\ verify(m, (p, q)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ anSpec(m(\overline{x}) : (p, q)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{newSpec})$$

$$\frac{\mathcal{U}, \mathcal{E} \vdash \langle C :\ verify(m, R\!\uparrow_{\mathcal{E}}(C, m)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ anReq(m) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{newMtd})$$

$$\frac{\mathcal{U} \star newReq(C.sub\!\downarrow, \{r\} n \{s\}), \mathcal{E} \vdash \langle C :\ verify(n, (r, s)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ anCalls(\{r\}\ n\ \{s\}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{call})$$

$$\frac{e : I \quad I \in \mathcal{E} \quad spec_{\mathcal{E}}(I, n) \twoheadrightarrow (r, s) \quad \mathcal{U}, \mathcal{E} \vdash \langle C :\ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C :\ anCalls(\{r\}\ e.n\ \{s\}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{extCall})$$

Figure 6: Inference rules for classes and the generated operations. The rules cover extensions of class hierarchies with new subclasses. We here ignore rules for flattening operations with set arguments, e.g., $anReq(m\ \overline{m})$ is assumed flattened to $anReq(m) \cdot anReq(\overline{m})$, and rules for discarding operations with empty arguments, e.g., an operation $anReq(\emptyset)$ is ignored.

$$\frac{\begin{array}{c}C \in \mathcal{E} \qquad \overline{I} \in \mathcal{E} \qquad \overline{m} = mids(\overline{M}) \qquad \mathcal{U} = \emptyset \\ \mathcal{E}' = \mathcal{E} \star addL(C, \overline{I}, \overline{f}, \overline{M}) \star remS(C, \overline{m}) \quad \mathcal{U}' = \mathcal{U} \star init(C.sub\downarrow) \star newSpec(C.sub, \overline{m}) \\ \mathcal{U}', \mathcal{E}' \vdash anClass(C, \overline{MS}, \overline{m}, \overline{m}\ public_{\mathcal{E}}(\overline{I})) \cdot \mathcal{P}\end{array}}{\mathcal{U}, \mathcal{E} \vdash \textbf{modify}\ C\ \textbf{implements}\ \overline{I}\ \{\overline{f}; \overline{M}\ \overline{MS}\} \cdot \mathcal{P}}\ (\textsc{Class-Extend})$$

$$\frac{\begin{array}{c}\mathcal{U} = \emptyset \qquad \mathcal{U}' = \mathcal{U} \star init(C.sub\downarrow) \star newSpec(C.sub, \overline{m}) \\ C \in \mathcal{E} \qquad \mathcal{E}' = \mathcal{E} \star remS(C, \overline{m}) \star remL(C, \overline{m}) \\ \mathcal{U}', \mathcal{E}' \vdash \langle C : upd(\overline{m}, \emptyset)\rangle \cdot \mathcal{P}\end{array}}{\mathcal{U}, \mathcal{E} \vdash \textbf{simplify}\ C\ \{\overline{m}\} \cdot \mathcal{P}}\ (\textsc{Class-Simplify})$$

$$\frac{\begin{array}{c}m \notin C.mtds \\ \mathcal{U}, \mathcal{E} \vdash \langle C : verify(m, R(C, m)) \cdot newIntSpec(m) \cdot upd(\overline{m}, \overline{T})\rangle \cdot \mathcal{P}\end{array}}{\mathcal{U}, \mathcal{E} \vdash \langle C : upd(m\ \overline{m}, \overline{T})\rangle \cdot \mathcal{P}}\ (\textsc{subSpec})$$

$$\frac{\begin{array}{c}n \notin C.mtds \\ \mathcal{U}, \mathcal{E} \vdash \langle C : upd(\overline{m}, \overline{T})\rangle \cdot \mathcal{P}\end{array}}{\mathcal{U}, \mathcal{E} \vdash \langle C : upd(\overline{m}, (\{r\}\ n\ \{s\})\ \overline{T})\rangle \cdot \mathcal{P}}\ (\textsc{noSubReq}) \qquad \frac{\begin{array}{c}n \in C.mtds \\ \mathcal{U}, \mathcal{E} \vdash \langle C : verify(n, (r, s)) \cdot upd(\overline{m}, \overline{T})\rangle \cdot \mathcal{P}\end{array}}{\mathcal{U}, \mathcal{E} \vdash \langle C : upd(\overline{m}, (\{r\}\ n\ \{s\})\ \overline{T})\rangle \cdot \mathcal{P}}\ (\textsc{subReq})$$

$$\frac{\mathcal{U} \star empty(C), \mathcal{E} \vdash \mathcal{P}}{\mathcal{U}, \mathcal{E} \vdash \langle C : upd(\emptyset, \emptyset)\rangle \cdot \mathcal{P}}\ (\textsc{empUpd}) \qquad \frac{\begin{array}{c}C \in \mathcal{U} \quad C.inh \notin \mathcal{U} \quad M_{\mathcal{U}}(C) = \overline{m} \\ \mathcal{U}, \mathcal{E} \vdash \langle C : upd(\overline{m}, \overline{T})\rangle \cdot \mathcal{P} \quad R_{\mathcal{U}}(C) = \overline{T}\end{array}}{\mathcal{U}, \mathcal{E} \vdash \mathcal{P}}\ (\textsc{newUpd})$$

Figure 7: Inference rules for the basic update operations **modify** and **simplify**.

Rule class-Simplify removes methods from a class. As above, method specifications and definitions are removed. The proof and update environments are updated accordingly before analyzing the generated *upd* operation. Since class-Simplify only removes methods, there is no need to reanalyze the class. Note that the update environment $\mathcal{U}(C)$ of a class C is checked after the analysis of its superclass (in newUpd), and for all analysed classes C, Rule empUpd removes C from $\mathcal{U}$. Consequently, for a successful analysis of an update operation, the analysis reaches some judgement $\mathcal{U}_r, \mathcal{E}_r \vdash$, where $\mathcal{U}_r = \emptyset$. We refer to $\mathcal{E}_r$ as the *resulting environment* of the update analysis.

The *upd* operation generated by the rules is essential in order to ensure the soundness of the requirements that are introduced below the modified class C. Consider Figure 8, where B_m denotes that m is defined in B, and correspondingly for C_m and D_m, and assume that the definition of m in C is removed. Since m is defined in B, there may still be calls to m in C and C'; i.e., the sets $R(C, m)$ and $R(C', m)$ may be non-empty. The analysis of $\langle C : upd(m, \emptyset)\rangle$ and $\langle C' : upd(m, \overline{T})\rangle$ will then analyse B_m with

Figure 8: Requirement analysis.

respect to these requirements. Furthermore, if m is public in C or C', the operation ensures that B_m satisfies the external behavior.

Soundness. For the operations in the calculus, we have the following soundness theorem, the proof of which can be found in Appendix A.

Theorem 1 (Soundness). *Let $\mathcal{E}$: Env be a sound environment, and let $\mathcal{P}$ be one of the basic update operations*

class C ***extends*** B ***implements*** $\overline{I}$ $\{\overline{f}; \overline{M}\ \overline{MS}\}$
simplify C $\{\overline{m}\}$
modify C ***implements*** $\overline{I}$ $\{\overline{f}; \overline{M}\ \overline{MS}\}$

Assume that the judgement $\mathcal{U}, \mathcal{E} \vdash \mathcal{P}$ is successfully analyzed leading to the resulting environment $\mathcal{E}_r$. Then $\mathcal{E}_r$ is sound.

Example 2. Consider the guarantees 1 and 2 for createBill given in Example 1. After basic update operations, each of these may still be verified for their respective subclasses, imposing different requirements to the method chargeFor. Guarantee 1 applies to RegularCustomer. For this class, the call to chargeFor binds to the implementation inherited from Customer which satisfies the imposed requirement $\{\mathsf{last}{=}l_0\}$ chargeFor $\{\textbf{result}{=}\mathsf{diff}(\mathsf{today}, l_0) * \mathsf{rate}\}$. Guarantee 2 applies to PreferredCustomer. Method chargeFor is overridden in PreferredCustomer, and this implementation satisfies

$$\{\mathsf{last}{=}l_0\}\ \mathsf{chargeFor}\ \{\textbf{result}{=}\mathsf{diff}(\mathsf{today}, l_0) * \mathsf{rate} * \mathsf{fact}\}$$

as imposed by the analysis of Guarantee 2.

Example 3. Consider the implementation of two bank accounts Account and TransAccount, given in Figure 9 (top). The class Account implements an interface Account with methods in and out for depositing and withdrawing funds, letting the result value indicate success. Additionally, the method avail returns available funds. We use invariant notation **inv** I to abbreviate a user-defined guarantee (I, I) for each public method, and auxiliary functions are defined in the interfaces by means of a **where** clause. Method guarantees are defined in terms of *local communication histories* H [20, 21, 22]. In this example, it suffices to consider communication histories consisting of the events corresponding to method completions generated by **this** object. The completion of a method m has the implicit effect $H := H; m(x; r)$ where x is the list of input parameters and r is the result value, and where $\_;\_$ denotes the sequence append operation. Only public methods are considered on H. Remark that invariants are not inherited by subclasses.

In the Account interface, the avail method is specified by a postcondition stating that the result value equals the bal function over H. In the class, this method is given the guarantee (true, result=bal) which implies the guarantee of the interface given the class invariant bal=bal(H). For the methods in and out,

```
interface Account {
  Int avail() : (true, result = bal(H))
  Bool in(Nat x)
  Bool out(Nat x)
where
  bal(empty) = 0
  bal(H;avail(;r)) = bal(H)
  bal(H; in(x;r)) = if r then bal(H)+x else bal(H) fi
  bal(H;out(x;r)) = if r then bal(H)−x else bal(H) fi
}

class Account implements Account {
  Int bal=0;
  Int avail(): (true, result = bal) {return bal}
  Bool upd(Int x) {Bool r; r:= bal+x ≥ 0; if r then bal:= bal+x fi; return r}
  Bool in (Nat x) {Bool r; r:=upd( x); return r}
  Bool out(Nat x) {Bool r; r:=upd(−x); return r}
  inv bal=bal(H)
}

interface Transfer {
  Bool trans(Nat x, Account dest)
where
  bal(H;trans(x,d;r)) = if ¬r ∨ (d = this) then bal(H) else bal(H) − x fi
}

class TransAccount extends Account implements Transfer, Account {
  Bool trans(Nat x, Account dest) {
    Bool r; r:=out(x); if r then dest.in(x) fi; return r}
  inv bal=bal(H)
}
```

```
interface Overdraft {
  Void setoverdraft(Nat x)
where bal(H;setoverdraft(x;)) = bal(H)
}
modify Account implements Overdraft {
  Nat overdraft=0;
  Void setoverdraft(Nat x) {overdraft:=x}
  Bool out(Nat x) {Bool r; r:= bal+x+overdraft ≥ 0; if r then upd(−x) fi; return r}
  Bool upd(Int x) {bal:= bal+x; return true}}
```

Figure 9: The interfaces and classes of the bank accounts (top) and the modification of the bank account (bottom).

the invariant gives the guarantee (bal=bal(H), bal=bal(H)), and these guarantees are satisfied by imposing the requirement

$$\{\text{bal=bal(H)}\}\ \text{upd(x)}\ \{\text{bal=bal(H)+}\textbf{if}\ \text{result}\ \textbf{then}\ \text{x}\ \textbf{else}\ 0\ \textbf{fi}\}.$$

This requirement is easily verified for the definition of upd in Account. The

method trans in TransAccount maintains the invariant by a similar argument.

Now consider a modification of the class Account to provide Overdraft functionality, redefining upd and out. The modification is shown in Figure 9 (bottom). The new version of out imposes the requirement

$$\{\mathsf{bal}{=}\mathsf{bal}(\mathsf{H})\}\ \mathsf{upd}(\mathsf{x})\ \{\mathsf{bal}{=}\mathsf{bal}(\mathsf{H}){+}\mathsf{x}\}$$

and the class modification can be verified using this requirement.

This example illustrates differences between *extending* a class with subclasses and *modifying* the class itself. The redefinition of upd would not be possible in a subclass since the original requirement then remains. On the other hand, introducing fees for out could be made in a subclass (not implementing the Account interface), but not in a class modification since the Account interface must be respected by modifications.

6. Related Work

To integrate software verification techniques into object-oriented design, proof systems should be constructed for incremental (or modular [23]) reasoning. Most prominent in that context are approaches based on *behavioral subtyping* [4], but these have been criticized for overly restricting code reuse and are often violated in practice [5]. *Lazy behavioral subtyping* [6, 7] facilitates incremental reasoning while allowing more flexible code reuse than traditional behavioral subtyping. We refer to [6, 7] for a more comprehensive discussion on incremental reasoning about class extensions. Whereas modular and incremental techniques are attractive from a verification perspective, they do impose restrictions on how code can be structured. In [24], a reasoning framework is developed which does not favor incremental reasoning over flexible code reuse, in order to compare possible operations on an object-oriented program during program development with respect to their verification cost.

Whereas lazy behavioral subtyping extends a class hierarchies, this paper broadens the approach to better reflect an object-oriented development process in terms of a few basic update operations which allow changes to classes inside the hierarchy. These operations were inspired by work on invasive software composition: the typing of dynamic class evolution in Creol [11] and the assemblage and verification of software product lines [25, 26]. Software product lines assemble features to create a range of different products; some approaches have been proposed to the deductive verification of such systems. In [27], a verification approach for behavioral program properties of software product lines is proposed where each feature has an associated partial proof script. These proof scripts are composed and checked when a product is assembled. A product-based analysis approach is proposed in [28] for delta-oriented programming [13]. Assuming one product variant has been fully verified, from the structure of a delta to generate another program variant, it is analyzed which proof obligations remain valid in the new product variant and need not be reestablished. In [29], a verification approach for delta-oriented programming is presented which applies a behavioral

subtyping discipline to delta-oriented programming; i.e., specifications of methods introduced by deltas must be more specific than previous versions of these methods. Then, each delta can be verified by approximating called methods defined in other deltas by the specification of their first introduction.

A recent paper by the authors [30] proposes a proof system for delta-oriented programs by means of transformational reasoning, which relaxes the restrictions of [29]. After verifying deltas in isolation, the actual products are verified based on the specifications already established for the used deltas. The transformational approach of [30] uses symbolic assumptions on called methods and thus separates the specifications of method implementations from the requirements to method calls in a way inspired by lazy behavioral subtyping [6]. Compared to the present work, that paper does not consider class hierarchies and late bound method calls nor operations which support, e.g., refactoring operations.

We have found few systems which address 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., [31, 32, 33], discuss how to preserve behavioral consistency between different model versions when refinement or refactoring is applied. Program transformations, such as *verification refactoring* [34], may be applied in order 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 [35], formalizing standard refactoring steps. Not limited to behaviorally preserving transformations, *slicing techniques* [36] may be used to describe the effect of update operations, to determine which properties are preserved or potentially invalidated in the new version.

7. Conclusion

Ideally, verified software could be designed from specifications and verified a posteriori. However, in practice both the software and its specifications *evolve*, both during the initial software design and during maintenance. Likewise, the requirements to a piece of software will also change over time. For this reason we cannot always expect that specifications are made before the code is written and that the verification efforts happen afterwards. Modularity is one answer to this problem, which isolates the part of the software that needs to be redesigned and re-verified. However, inside the module, large chunks of code may still have valid specifications and proofs, even if the other parts of the specifications or program have changed.

In the setting of object-oriented programming, reuse of code is essential. However, classes made early in a class hierarchy influence subclasses introduced later. Early design decisions may later turn out undesirable, and hinder flexible code reuse. Thus, there is a need to redesign classes high up in a hierarchy. This need for redesign concerns specifications as well as code, and in particular requirements, which are inherited by existing and future subclasses. A mechanism for adaptation of classes in the middle of a class hierarchy is therefore valuable with respect to code reuse and program reasoning.

We believe that in order to facilitate the development and maintenance of verified programs, it is an advantage that programming and verification activities go hand in hand. For this purpose, programming environments need to let the program developer alternate between programming and verification tasks in a flexible way. This paper has presented a calculus which tracks verification conditions when the program changes, based on an *incremental* rather than a modular reasoning framework. Changes are formalized through basic update operations which include addition, removal, and redefinition of code. These operations may be composed to form more complex refactorings, as shown in our examples. In contrast to previous work on lazy behavioral subtyping, the approach in this paper allows the removal and modification of guarantees and requirements when methods are removed or modified. As requirements put restrictions on code reuse in subclasses, our framework allows the revision of inconvenient restrictions, allowing flexible reuse of code, in the sense that restrictions on method redefinitions and local calls can be adjusted.

The framework presented here is oriented towards tool implementation. Our formalism can be used to generate proof outlines for each class adaption made by an update operation, supporting incremental generation of verification conditions. We are planning an integration in the KeY tool [37], which has the underlying support for dealing with verification conditions. A complementary line of work is *proof adaptation*, which has been studied in the automated theorem prover communities. Proof adaptation considers how to adapt a proof when a specification changes. Such techniques would fit naturally into an implementation of the reasoning framework proposed in this paper, to alleviate the overhead of additional proof activity resulting from the flexibility of the proposed framework.

References

[1] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, P. Y. H. Wong, Modeling spatial and temporal variability with the HATS abstract behavioral modeling language, in: M. Bernardo, V. Issarny (Eds.), Proceedings of the 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011), Vol. 6659 of Lecture Notes in Computer Science, Springer-Verlag, 2011, pp. 417–457.

[2] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2) (2004) 126–139.

[3] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[4] B. H. Liskov, J. M. Wing, A behavioral notion of subtyping, ACM Transactions on Programming Languages and Systems 16 (6) (1994) 1811–1841.

[5] N. Soundarajan, S. Fridella, Inheritance: From code reuse to reasoning reuse, in: P. Devanbu, J. Poulin (Eds.), Proceedings of the Fifth International Conference on Software Reuse (ICSR5), IEEE Computer Society Press, 1998, pp. 206–215.

[6] J. Dovland, E. B. Johnsen, O. Owe, M. Steffen, Lazy behavioral subtyping, Journal of Logic and Algebraic Programming 79 (7) (2010) 578–607.

[7] J. Dovland, E. B. Johnsen, O. Owe, M. Steffen, Incremental reasoning with lazy behavioral subtyping for multiple inheritance, Science of Computer Programming 76 (10) (2011) 915–941.

[8] F. Damiani, J. Dovland, E. B. Johnsen, I. Schaefer, Verifying traits: an incremental proof system for fine-grained reuse, Formal Aspects of Computing 26 (4) (2014) 761–793.

[9] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Mass., 1995.

[11] E. B. Johnsen, M. Kyas, I. C. Yu, Dynamic classes: Modular asynchronous evolution of distributed concurrent objects, in: A. Cavalcanti, D. Dams (Eds.), Proc. 16th International Symposium on Formal Methods (FM'09), Vol. 5850 of Lecture Notes in Computer Science, Springer-Verlag, 2009, pp. 596–611.

[12] E. B. Johnsen, O. Owe, An asynchronous communication model for distributed concurrent objects, Software and Systems Modeling 6 (1) (2007) 35–58.

[13] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-oriented programming of software product lines, in: Proceedings of the 14th International Conferenc on Software Product Lines (SPLC 2010), Vol. 6287 of Lecture Notes in Computer Science, Springer-Verlag, 2010, pp. 77–91.

[14] E. B. Johnsen, O. Owe, I. C. Yu, Creol: A type-safe object-oriented model for distributed concurrent systems, Theoretical Computer Science 365 (1–2) (2006) 23–66.

[15] K. R. Apt, Ten years of Hoare's logic: A survey — Part I, ACM Transactions on Programming Languages and Systems 3 (4) (1981) 431–483.

[16] K. R. Apt, F. S. de Boer, E.-R. Olderog, Verification of Sequential and Concurrent Systems, 3rd Edition, Texts and Monographs in Computer Science, Springer-Verlag, 2009.

[17] F. S. de Boer, A WP-calculus for OO, in: W. Thomas (Ed.), Proceedings of Foundations of Software Science and Computation Structure, (FOSSACS'99), Vol. 1578 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 135–149.

[18] C. A. R. Hoare, An Axiomatic Basis of Computer Programming, Communications of the ACM 12 (1969) 576–580.

[19] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs I, Acta Informatica 6 (4) (1976) 319–340.

[20] O.-J. Dahl, O. Owe, Formal development with ABEL, in: S. Prehn, H. Toetenel (Eds.), Proc. Formal Software Development Methods (VDM'91), Vol. 552 of Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 320–362.

[21] J. Dovland, E. B. Johnsen, O. Owe, Verification of concurrent objects with asynchronous method calls, in: Proceedings of the IEEE International Conference on Software - Science, Technology & Engineering (SwSTE'05), IEEE Computer Society Press, 2005, pp. 141–150.

[22] C. C. Din, J. Dovland, E. B. Johnsen, O. Owe, Observable behavior of distributed systems: Component reasoning for concurrent objects, Journal of Logic and Algebraic Programming 81 (3) (2012) 227–256.

[23] K. K. Dhara, G. T. Leavens, Forcing behavioural subtyping through specification inheritance, in: Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, 1996, pp. 258–267, also as Iowa State University Tech. Rep. TR-95-20c.

[24] J. Dovland, E. B. Johnsen, I. C. Yu, Tracking behavioral constraints during object-oriented software evolution, in: Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'12), Part I, Vol. 7609 of Lecture Notes in Computer Science, Springer-Verlag, 2012, pp. 253–268.

[25] K. Pohl, G. Böckle, F. Van Der Linden, Software Product Line Engineering: Foundations, Principles, and Techniques, Springer-Verlag, 2005.

[26] I. Schaefer, R. Hähnle, Formal methods in software product line engineering, IEEE Computer 44 (2) (2011) 82–85.

[27] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, Proof composition for deductive verification of software product lines, in: Proceedings of the International Workshop Variability-intensive Systems Testing, Validation and Verification (VAST), IEEE Computer Society Press, 2011, pp. 270–277.

[28] D. Bruns, V. Klebanov, I. Schaefer, Verification of software product lines with delta-oriented slicing, in: Proceedings of the International Conference on Formel Verification of Object-oriented Software (FoVeOOS 2010), Vol. 6528 of Lecture Notes in Computer Science, 2011, pp. 61–75.

[29] R. Hähnle, I. Schaefer, A Liskov principle for delta-oriented programming, in: Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), Part I, Vol. 7609 of Lecture Notes in Computer Science, Springer-Verlag, 2012, pp. 32–46.

[30] F. Damiani, J. Dovland, E. B. Johnsen, O. Owe, I. Schaefer, I. C. Yu, A transformational proof system for delta-oriented programming, in: E. S. de Almeida, C. Schwanninger, D. Benavides (Eds.), Proceedings of the 16th International Software Product Line Conference (SPLC), Volume 2, ACM, 2012, pp. 53–60, proceedings of the 3rd International Workshop on Formal Methods for Software Product Lines (FMSPLE'12).

[31] R. Van Der Straeten, V. Jonckers, T. Mens, A formal approach to model refactoring and model refinement, Software and Systems Modeling 6 (2007) 139–162.

[32] T. Massoni, R. Gheyi, P. Borba, Synchronizing model and program refactoring, in: J. Davies, L. Silva, A. Simao (Eds.), Formal Methods: Foundations and Applications, Vol. 6527 of Lecture Notes in Computer Science, Springer-Verlag, 2011, pp. 96–111.

[33] S. Marković, T. Baar, Refactoring OCL annotated UML class diagrams, Software and Systems Modeling 7 (2008) 25–47.

[34] X. Yin, J. C. Knight, W. Weimer, Exploiting refactoring in formal verification, in: Proceedings Dependable Systems and Networks (DSN'09), IEEE Computer Society Press, 2009, pp. 53–62.

[35] N. Ubayashi, J. Piao, S. Shinotsuka, T. Tamai, Contract-based verification for aspect-oriented refactoring, in: Proceedings of the International Conference on Software Testing, Verification, and Validation, IEEE Computer Society Press, 2008, pp. 180–189.

[36] H. Wehrheim, Slicing techniques for verification re-use, Theoretical Computer Science 343 (3) (2005) 509–528.

[37] B. Beckert, R. Hähnle, P. H. Schmitt (Eds.), Verification of Object-Oriented Software. The KeY Approach, Vol. 4334 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2007.

A. Proof of the Soundness Theorem

This appendix details the proof of Theorem 1.

Theorem 1 (Soundness). *Let $\mathcal{E} : \mathsf{Env}$ be a sound environment, $\mathcal{P}$ be one of the three operations*

$$\begin{array}{l}\textbf{\textit{class}}\ C\ \textbf{\textit{extends}}\ B\ \textbf{\textit{implements}}\ \overline{I}\ \{\overline{f}; \overline{M}\ \overline{MS}\}\\ \textbf{\textit{simplify}}\ C\ \{\overline{m}\}\\ \textbf{\textit{modify}}\ C\ \textbf{\textit{implements}}\ \overline{I}\ \{\overline{f}; \overline{M}\ \overline{MS}\}\end{array}$$

Assume that the judgement $\mathcal{U}, \mathcal{E} \vdash \mathcal{P}$ is successfully analyzed leading to the resulting environment $\mathcal{E}_r$. Then $\mathcal{E}_r$ is sound.

Proof. Soundness of the different operations are treated by Lemma 1, Lemma 2, and Lemma 3 below. The theorem then follows from these lemmas.

The remainder of this appendix is structured as follows: Auxiliary definitions are introduced in Appendix A.1. Appendix A.2, A.3, and A.4 treats the soundness of simplify, modify, and class operations, respectively. Auxiliary lemmas can be found in Appendix A.5.

A.1. Auxiliary Definitions

Let $cls(\mathcal{U})$ return the set of names for classes well-defined in $\mathcal{U}$ such that $cls(\mathcal{U}) = \emptyset$ if $\mathcal{U} = \emptyset$, $cls(\mathcal{U} \star init(C)) = cls(\mathcal{U}) \cup C$, and $cls(\mathcal{U} \star empty(C)) = cls(\mathcal{U}) \setminus C$. Let $\#(\mathcal{U})$ denote the size of $cls(\mathcal{U})$.

Definition 5. ***(Partially sound classes.)*** *For environments $\mathcal{E}$ and $\mathcal{U}$ and $C : \mathsf{Cid}$ such that $C \in \mathcal{E}$ and $\mathcal{U}(C) \neq \bot$, we say that C* is partially sound in *$\mathcal{U}, \mathcal{E}$ if the following conditions hold:*

1. $\forall \langle m, (p,q), O\rangle \in S_{\mathcal{E}}(C)\ .\ O \vdash_{PL} body_{\mathcal{E}}(C,m) : (p,q)$
   $\wedge\ \forall \{r\}\ e.n\ \{s\} \in O\ .\ \forall I\ .\ e : I \Rightarrow spec_{\mathcal{E}}(I,n) \twoheadrightarrow (r,s)$
2. $\forall m\ .\ m \notin M_{\mathcal{U}}(C) \Rightarrow$
   $\forall (r,s) \in R_{\mathcal{E}}(C,m)\ .\ G\!\uparrow_{\mathcal{E}}(C,m) \twoheadrightarrow (r,s)\ \wedge$
   $\forall (r,s) \in R\!\uparrow_{\mathcal{E}}(C.inh, m)\ .\ G\!\uparrow_{\mathcal{E}}(C,m) \twoheadrightarrow (r,s) \vee \{r\}\ m\ \{s\} \in R_{\mathcal{U}}(C)$
3. $\forall \overline{I}\ .\ \overline{I} = L_{\mathcal{E}}(C).impl\ .$
   $\forall n \in public_{\mathcal{E}}(\overline{I})\ .\ n \notin M_{\mathcal{U}}(C) \Rightarrow G\!\uparrow_{\mathcal{E}}(C,n) \twoheadrightarrow spec_{\mathcal{E}}(\overline{I}, n)$

Definition 6. ***(Partially sound environments.)*** *Given environments $\mathcal{E}$ and $\mathcal{U}$, we say that $\mathcal{U}, \mathcal{E}$ are partially sound if and only if, for all $C \in \mathcal{E}$, the following holds:*

- *If $C \notin \mathcal{U}$ then C is sound in $\mathcal{E}$.*
- *If $C \in \mathcal{U}$ then C is partially sound in $\mathcal{U}, \mathcal{E}$ and $C.sub\!\downarrow \subseteq cls(\mathcal{U})$.*

A.2. Lemma 1: Class Simplify

Lemma 1 (Class simplify). *Let $\mathcal{E} : \mathsf{Env}$ be a sound environment, $U : \mathsf{Cid}$, $\mathcal{U} = \emptyset$, and assume that the judgement $\mathcal{U}, \mathcal{E} \vdash$ **simplify** $U\ \{\overline{u}\}$ is successfully analyzed, leading to the resulting judgement $\mathcal{U}_r, \mathcal{E}_r \vdash$. Then $\mathcal{E}_r$ is sound.*

Proof. The initial judgement is analyzed by Rule CLASS-SIMPLIFY, leading to some judgement $\mathcal{U}', \mathcal{E}' \vdash \langle U : upd(\overline{u}, \emptyset)\rangle$. By rule premise, we have the following for $\mathcal{U}', \mathcal{E}'$:

- $\forall C < U\ .\ C \in \mathcal{U}'$ and C is partially sound in $\mathcal{U}', \mathcal{E}'$.
- For all classes $C \in \mathcal{E}'$ such that $C \notin \mathcal{U}'$ and $C \neq U$, we know that C is sound in $\mathcal{E}'$ and C is not below U.
- For class U we have the following (the conditions refer to sound classes, Definition 3):
  - Condition 1 holds.
  - Condition 2 holds for all m except for $m \in \overline{u}$.
  - Condition 3 holds for all public methods n except for $n \in \overline{u}$.
  - $\forall u \in \overline{u}\,.\, u \notin U.mtds$ and $U \notin \mathcal{U}'$.

Since the analysis of the initial **simplify** operation succeeds, the analysis of $\langle U : upd(\overline{u}, \emptyset)\rangle$ must succeed. Let $\mathcal{E}'', \mathcal{U}''$ be the environments resulting from the analysis of *upd*, i.e., the judgement $\mathcal{U}'', \mathcal{E}'' \vdash \langle U : upd(\emptyset, \emptyset)\rangle$ is reached. We next prove that U must be sound in $\mathcal{E}''$, according to Definition 3. Condition 1 holds by the initial assumption on $\mathcal{E}'$ and since reqNotDer, the only rule providing a specification, ensures the existence of proof outlines. For condition 2, we consider the operation $verify(u, R_{\mathcal{E}'}(U, u))$ that is generated for each $u \in \overline{u}$. Successful analysis of this operation ensures $G{\uparrow}_{\mathcal{E}''}(U, u) \twoheadrightarrow R_{\mathcal{E}'}(U, u)$. We may also prove that all requirements added to $R(U, u)$ during the analysis of U follows from the guarantees of u, this proof corresponds to the one given in case **(c)** in the proof of Lemma 5. We then have $G{\uparrow}_{\mathcal{E}''}(U, u) \twoheadrightarrow R_{\mathcal{E}''}(U, u)$. The desired conclusion $G{\uparrow}_{\mathcal{E}''}(U, u) \twoheadrightarrow R{\uparrow}_{\mathcal{E}''}(U, u)$ then follows since $u \notin U.mtds$. Especially, if $R{\uparrow}_{\mathcal{E}''}(U, u) \neq \emptyset$, type-safety guarantees that there exists some implementation of u above U. Then $G{\uparrow}_{\mathcal{E}''}(U, u) = G_{\mathcal{E}''}(U, u) \cup G{\uparrow}_{\mathcal{E}''}(B, u)$ for $B = U.inh$, and $G{\uparrow}_{\mathcal{E}''}(B, u) \twoheadrightarrow R{\uparrow}_{\mathcal{E}''}(B, u)$ since B is sound. For condition 3, the analysis of the generated operation $newIntSpec(u)$ ensures the condition for all public methods $u \in \overline{u}$. We also need to prove that sound classes in $\mathcal{E}'$ are also sound in $\mathcal{E}''$ and that partially sound classes in $\mathcal{U}', \mathcal{E}'$ are partially sound in $\mathcal{U}'', \mathcal{E}''$. These proofs follow a corresponding argument as for cases **(a)** and **(b)** in the proof of Lemma 5.

We thereby have that $\mathcal{U}'', \mathcal{E}''$ are partially sound. The resulting judgement $\mathcal{U}'', \mathcal{E}'' \vdash \langle U : upd(\emptyset, \emptyset)\rangle$ is analyzed by Rule empUpd, which leads to $\mathcal{U}'', \mathcal{E}'' \vdash$ since U is not in $\mathcal{U}''$. Soundness of the resulting environment $\mathcal{E}_r$ then follows by Lemma 6.

A.3. Lemma 2: Class modify

Lemma 2 (Class modify). *Let $\mathcal{E}$: Env be a sound environment, and assume that the judgement $\mathcal{U}, \mathcal{E} \vdash$ **modify** U **implements** $\overline{I}$ $\{\overline{f}; \overline{M}\ \overline{MS}\}$ is successfully analyzed, leading to the resulting environments $\mathcal{U}_r$ and $\mathcal{E}_r$. Then $\mathcal{E}_r$ is sound.*

Proof. Application of Rule class-Extend leads to the judgement $\mathcal{U}', \mathcal{E}' \vdash anClass(U, \overline{MS}, \overline{u}, \overline{u}\ public_{\mathcal{E}}(\overline{I}))$, where $\overline{u} = mids(\overline{M})$. The proof follows the same outline as for Lemma 1. In order to apply Lemma 6, we need to ensure

that the analysis of U succeeds in some environments $\mathcal{U}', \mathcal{E}'$ such that U is sound in $\mathcal{E}'$. For each $u \in \overline{u}$ an operation $anReq(u)$ is generated, which ensures that each method in $\overline{u}$ satisfies its inherited requirements. Furthermore, if m is a public method in $\overline{u}$ or in $\overline{I}$, an operation $intSpec(m)$ is generated, ensuring that the guarantees of m satisfies the interface constraints.

A.4. Lemma 3: Class Definition

Lemma 3 (Class definition). *Let $\mathcal{E}$: Env be a sound environment, and assume that the judgement $\mathcal{U}, \mathcal{E} \vdash$ **class** C **extends** B **implements** $\overline{I}$ $\{\overline{f}; \overline{M}\ \overline{MS}\}$ is successfully analyzed, leading to the resulting environments $\mathcal{U}_r$ and $\mathcal{E}_r$. Then $\mathcal{E}_r$ is sound.*

Proof. The premises of Rule newClass ensures that $\mathcal{U} = \emptyset$, and only the rules in Figure 6 are applied during the analysis of C. Remark that during analysis of C, the environment is only extended with specifications for C, i.e. for all classes $B \in \mathcal{E}$, we have $S_{\mathcal{E}_r}(B) = S_{\mathcal{E}}(B)$. These rules maintain an empty update environment, which means that $\mathcal{U}_r = \emptyset$. Especially, since $C.sub\downarrow$ is empty (C is a new class at the bottom of the class hierarchy) it follows that $\mathcal{U}$ is empty after application of call. The application of Rule newClass leads to a judgement $\mathcal{U}, \mathcal{E}' \vdash anClass(C, \overline{MS}, \overline{m}, public_{\mathcal{E}}(\overline{I}))$, where $\overline{m} = mids(\overline{M})$ and $\mathcal{E}'$ is $\mathcal{E}$ extended with the definition of C. It follows directly by induction over the inference rules that for any sound class B in the initial environment $\mathcal{E}$, then B is also sound in $\mathcal{E}_r$. In order to prove soundness of $\mathcal{E}_r$ it therefore suffices to prove that C is sound in $\mathcal{E}_r$. Assume that C is not sound in $\mathcal{E}_r$, then one of the conditions in Definition 3 must be violated. *Condition 1*: The only Rule that extends $S(C)$ with a new specification is reqNotDer. This rule ensures that there is a valid proof outline O for the new specification which is analyzed by $anCalls(O)$. For each external call $\{r\}\ e.n\ \{s\}$ in O, Rule extCall ensures that the requirement follows from the interface constraints of e. Thus, this condition is not violated in $\mathcal{E}_r$. *Condition 2*: Let $B = C.inh$. In order to violate the condition, there must be some requirement (r, s) either in $R_{\mathcal{E}_r}(C, m)$ or in $R{\uparrow}_{\mathcal{E}_r}(B, m)$ such that $G{\uparrow}_{\mathcal{E}_r}(C, m) \twoheadrightarrow (r, s)$ does not hold. *Case* $(r, s) \in R_{\mathcal{E}_r}(C, m)$: Then (r, s) must occur in a proof outline of some specification included in $S(C)$ by Rule reqNotDer. However, the proof outline is then analyzed by a *anCalls* operation which lead to an operation $verify(m, (r, s))$ by Rule call. This operation succeeds by either reqDer or reqNotDer, ensuring $G{\uparrow}_{\mathcal{E}_r}(C, m) \twoheadrightarrow (r, s)$. *Case* $(r, s) \in R{\uparrow}_{\mathcal{E}_r}(B, m)$: If $m \notin C.mtds$, the condition cannot be violated since $G{\uparrow}_{\mathcal{E}_r}(C, m) = G_{\mathcal{E}_r}(C, m) \cup G{\uparrow}_{\mathcal{E}_r}(B, m)$ and B is sound, i.e., $G{\uparrow}_{\mathcal{E}_r}(B, m) \twoheadrightarrow R{\uparrow}_{\mathcal{E}_r}(B, m)$. Otherwise, if $m \in C.mtds$, the analysis of the initial *anClass* operation leads to a *anReq* operation for each method defined in C. By Rule newMtd, the operation $verify(m, R{\uparrow}_{\mathcal{E}''}(C, m))$ where $\mathcal{E}''$ is an extension of $\mathcal{E}$. The analysis of each of these requirements succeeds by either Rule reqDer or reqNotDer. Since $R{\uparrow}_{\mathcal{E}_r}(B, m) \subseteq R{\uparrow}_{\mathcal{E}''}(C, m)$ the conclusion $G{\uparrow}_{\mathcal{E}_r}(C, m) \twoheadrightarrow (r, s)$ then follows. *Condition 3*: In order to violate this condition, there must be some interface constraint $\{p\}\ n\ \{q\}$ for some public method n in $\overline{I}$ such that the relation $G{\uparrow}_{\mathcal{E}_r}(C, n) \twoheadrightarrow (r, s)$ does not hold. However, for each

public method n in $\overline{M}$, an operation $intSpec(n)$ is generated during the analysis of C. Since this operation succeeds, Rule intSpec ensures the condition.

A.5. Additional Lemmas

Lemma 4. *Given partially sound environments $\mathcal{U}, \mathcal{E}$ where $\mathcal{U} = \emptyset$. Then $\mathcal{E}$ is a sound environment.*

Proof. For each class, the obligations of Definition 3 follow directly from Definition 6 when $\mathcal{U} = \emptyset$. Especially, for each $C \in \mathcal{E}$ we have $C \notin \mathcal{U}$ which means that $m \notin M_{\mathcal{U}}(C)$ and $R_{\mathcal{U}}(C) = \emptyset$ for all m.

Lemma 5. *Let $\mathcal{U}, \mathcal{E}$ be partially sound, $\mathcal{U} \neq \emptyset$, and consider the judgement $\mathcal{U}, \mathcal{E} \vdash$. Let C be a class such that $C \in \mathcal{U}$, $C.inh \notin \mathcal{U}$, $C.inh \in \mathcal{E}$, and let $\mathcal{U}, \mathcal{E} \vdash \langle C : upd(\overline{m}, \overline{T}) \rangle$ be the judgement resulting from application of Rule* newUpd. *Assume that the analysis of this operation succeeds with the judgement $\mathcal{U}', \mathcal{E}' \vdash \langle C : upd(\emptyset, \emptyset) \rangle$. Then*

- *cls*$(\mathcal{U}') =$ *cls*$(\mathcal{U})$
- *C is sound in $\mathcal{E}'$.*
- *$\mathcal{U}', \mathcal{E}'$ are partially sound.*

Proof. The equality *cls*$(\mathcal{U}) =$ *cls*$(\mathcal{U}')$ follows by induction over the rules manipulating $\langle C : \mathcal{O} \rangle$ operations. Especially, Rule empUpd is not applied during this analysis.

For the two remaining clauses, we prove the following:

(a) *For any class $A \notin \mathcal{U}$, we have that A is sound in $\mathcal{E}'$.* The analysis of $\langle C : \mathcal{O} \rangle$ modifies only the environment $\mathcal{E}$ of class C. Since $\mathcal{U}, \mathcal{E}$ are partially sound, we know that for any class $A \notin \mathcal{U}$ that A is sound in $\mathcal{E}$ and that A is not below C. We then have that A is sound also in $\mathcal{E}'$. Especially, the requirements inherited by A cannot be extended during the analysis of C.

(b) *For any class $D \in \mathcal{U}$ where $D \neq C$, we have that D is partially sound in $\mathcal{U}'$.* Assume that D is not partially sound in $\mathcal{U}'$. Thus, at least one of the conditions of Definition 5 is violated for D. Recall that D is partially sound in the initial update environment $\mathcal{U}$. Guarantees and requirements of D (contained in environment $\mathcal{E}$), in addition to the set $M_{\mathcal{U}}(D)$ are not modified during the analysis of C. Thus, the only way to introduce unsoundness is to violate Condition 2 of Definition 5. This can only happen if D is a subclass of C and the set $R(C, n)$ has been extended with some requirement (r, s) such that the condition is violated. However, this is impossible since reqNotDer is the only rule that extends $R(C, n)$, which means that an $anCalls(\{r\}\ n\ \{s\})$ operation is analyzed by call. Thus rule ensures that $\{r\}\ n\ \{s\}$ is added to $R_{\mathcal{U}}(D)$ through the extension of $\mathcal{U}$.

(c) *Class C is sound in $\mathcal{E}'$.* Let $B = C.inh$. Since C is selected for analysis by NEWUPD, we have $B \notin \mathcal{U}$, which means that B is sound in $\mathcal{E}$ and $\mathcal{E}'$. Condition 1 of sound classes (Definition 3) follows from partial soundness of $\mathcal{U}$ and the premises of Rule REQNOTDER. For Condition 2, we first consider the set $R_{\mathcal{E}'}(C, m)$ for some method m. Let (r, s) be some requirement in this set. If (r, s) is introduced in this set during the analysis of C, i.e., $(r, s) \notin R_{\mathcal{E}}(C, m)$, the requirement must appear in some specification added by Rule REQNOTDER. Further analysis of the proof outline of this specification will then ensure $G{\uparrow_{\mathcal{E}'}}(C, m) \twoheadrightarrow (r, s)$. Otherwise, if $(r, s) \in R_{\mathcal{E}}(C, m)$, we consider two cases. *Case* $m \in M_{\mathcal{U}}(C)$, i.e., $m \in \overline{m}$: During the analysis of C, Rule SUBSPEC is applied to this method, leading to the operation $\mathit{verify}(m, R_{\mathcal{E}''}(C, m))$ for some environment $\mathcal{E}''$ that is an extension of $\mathcal{E}$. The successful analysis of this operation ensures $G{\uparrow_{\mathcal{E}'}}(C, m) \twoheadrightarrow (r, s)$. *Case* $m \notin M_{\mathcal{U}}(C)$, i.e., $m \notin \overline{m}$: The condition $G{\uparrow_{\mathcal{E}'}}(C, m) \twoheadrightarrow (r, s)$ then follows directly from partial soundness of C.

We thereby have $G{\uparrow_{\mathcal{E}'}}(C, m) \twoheadrightarrow R_{\mathcal{E}'}(C, m)$, In order to conclude Condition 2, it then remains to prove $G{\uparrow_{\mathcal{E}'}}(C, m) \twoheadrightarrow R{\uparrow_{\mathcal{E}'}}(B, m)$. We consider two cases. *Case* $m \notin C.mtds$: By definition, we then have $G{\uparrow_{\mathcal{E}'}}(C, m) = G_{\mathcal{E}'}(C, m) \cup G{\uparrow_{\mathcal{E}'}}(B, m)$. Since B is sound in $\mathcal{E}'$ we have $G{\uparrow_{\mathcal{E}'}}(B, m) \twoheadrightarrow R{\uparrow_{\mathcal{E}'}}(B, m)$ which ensures the proof obligation. *Case* $m \in C.mtds$: By partial soundness of C in $\mathcal{U}, \mathcal{E}$, we know that $G{\uparrow_{\mathcal{E}}}(C, m) \twoheadrightarrow (r, s) \vee \{r\}\ m\ \{s\} \in \overline{T}$ for all $(r, s) \in R{\uparrow_{\mathcal{E}}}(B, m)$. For all $\{r\}\ m\ \{s\} \in \overline{T}$, Rule SUBREQ applies, ensuring $G{\uparrow_{\mathcal{E}'}}(C, m) \twoheadrightarrow (r, s)$. The proof obligation then follows since $R{\uparrow_{\mathcal{E}}}(B, m) = R{\uparrow_{\mathcal{E}'}}(B, m)$ and $G{\uparrow_{\mathcal{E}}}(C, m) \subseteq G{\uparrow_{\mathcal{E}'}}(C, m)$.

Consider next Condition 3 of sound classes, Definition 3. For any public method m, the condition follows directly from partial soundness of C if $m \notin \overline{m}$. Otherwise, if $m \in \overline{m}$, and operation $\mathit{newIntSpec}(m)$ is generated during the analysis of C, which leads to an operation $\mathit{intSpec}(m)$ by NEWINTSPEC1 since m is public. Since the analysis of this operation succeeds, Condition 3 is satisfied also in this case.

It follows from **(a)**, **(b)**, and **(c)** that C is sound in $\mathcal{E}'$ and that $\mathcal{U}', \mathcal{E}'$ are partially sound.

Lemma 6. *Let $\mathcal{U}, \mathcal{E}$ be partially sound environments, and assume that $\mathcal{U}, \mathcal{E} \vdash$ is successfully analysed, leading to the resulting judgement $\mathcal{U}_r, \mathcal{E}_r \vdash$. Then $\mathcal{E}_r$ is sound.*

Proof. Let $n = \#(\mathcal{U})$. If $n = 0$, the conclusion follows directly from Lemma 4. Otherwise, we prove the conclusion by n applications of Lemma 5.

Considering the initial environments $\mathcal{U}, \mathcal{E}$ with $n > 0$. Since the analysis succeeds, there must exists some class $C \in \mathcal{U}$ to which Lemma 5 can be applied. We then arrive at some judgement $\mathcal{U}', \mathcal{E}' \vdash \langle C : \mathit{upd}(\emptyset, \emptyset)\rangle$ such that $\mathit{cls}(\mathcal{U}) = \mathit{cls}(\mathcal{U}')$, C is sound in $\mathcal{E}'$ and $\mathcal{U}', \mathcal{E}'$ are partially sound. This judgement is analyzed by Rule EMPUPD, which removes C from $\mathcal{U}'$. Since C is sound in $\mathcal{E}'$,

the removal preserves partial soundness, i.e., the environments $\mathcal{U}'', \mathcal{E}'$, where $\mathcal{U}'' = \mathcal{U}' \star \mathit{empty}(C)$, are partially sound. We then have $\#(\mathcal{U}'') = \#(\mathcal{U}') - 1$, and performing a corresponding argument $n-1$ more times, we finally arrive at some resulting environments $\mathcal{U}_r, \mathcal{E}_r$ that are partially sound with $\mathcal{U}_r = \emptyset$. The conclusion then follows by Lemma 4.