A Framework for Flexible Program Evolution and Verification of Distributed Systems

Olaf Owe[1], Elahe Fazeldehkordi[1,3], and Jia-Chun Lin[1,2]

[1] Department of Informatics, University of Oslo
[2] Department of Information Security and Communication Technology, NTNU Gjøvik
[3] Department of Technology Systems, University of Oslo

Abstract. Program evolution may reveal bad design decisions, misunderstandings, erroneous code, or erroneous specifications, because problems made early in the design of a system may not be discovered until much later in the life-time of the system. Non-trivial changes of old code may be necessary. Flexibility in making such changes is essential, especially in a distributed setting where the system components are updated independently. In this setting re-verification is challenging. We consider flexibility with respect to what changes can be made as well as what can be efficiently reverified.
In this paper we propose a flexible framework for modeling and evolution of distributed systems. It supports unrestricted modifications in such systems, both in code and specifications, and with support of verification and re-verification. We consider on the setting of concurrent and object-oriented distributed programs, and introduce a core high-level modeling language supporting active objects. We allow *multiple inheritance* because it gives added flexibility during evolution, allowing a wider class of software changes. To avoid undesired effects of multiple inheritance, we apply a *healthy* binding strategy. We prove that the framework supports *Modification Independence* and *Hierarchy Independence*, which requires healthy binding. We demonstrate that our framework can deal with verification of software changes that are not possible in comparable frameworks.

Keywords: Program evolution; Program reasoning; Software changes; Multiple inheritance; Healthy binding; Active objects; Concurrency; Re-verification; Evolution flexibility; Modification independence; Hierarchy independence.

1 Introduction

There is a need for program evolution in modern systems, because of long lifetime and changing environmental needs. System development is a complicated process where many kinds of mistakes can be made over time, including bad design decisions, unclear specifications, misunderstandings, and erroneous code or specifications. Problems or bad design decisions made early may not be discovered until much later. Redesigning or modifying code made at an early stage in the software development may have severe implications on the overall system. Making changes may create new problems that are hard to foresee. These kinds of problems are severe in the setting of concurrent programs where the interaction of the different concurrent units is complicated, and also in the setting of object-oriented programs where inheritance, late binding, and code

reuse cause dependencies between the classes. A systematic approach, in which the consequences of a software change can be formalized, would be advantageous. Formal methods could be helpful in supporting specification and analysis of program properties. However, formal methods are mainly oriented towards developing correct specifications and programs, rather than the process of redoing earlier decisions. It is therefore interesting to look at formal frameworks with support for unrestricted software changes, and such that the framework can detect possible consequences. A trivial approach to reasoning about program changes is to re-verify and reprove all results whenever a change has been made. However this is time-consuming and expensive, especially for large software systems. Ideally we would like to reprove as little as possible, without losing soundness. This is critical in the setting of distributed systems where the system components are updated independently.

We focus on the setting of distributed, concurrent, and object-oriented systems, and introduce a framework for modeling, development, and evolution of such systems – with support of verification. Our framework includes several life cycle aspects such as formal requirement specification, system design, executable modeling, analysis, and maintenance. This means that one can avoid translation between different formalisms. The framework allows unrestricted changes in code and requirements, and includes a theory for reverification of a changed system. We consider programming mechanisms for efficient, imperative style programming in a distributed setting, including non-blocking as well as blocking remote method calls, combined with suspension and scheduling control of processes inside an object. Our goal is *flexibility*, in the sense of support of unrestricted software changes and with simplicity of reverification, more specifically, that the framework makes it possible to do desired changes in software and requirements (*Modification Independence*, theorem 1), and that the effect of changing one class is limited to that class and possibly subclasses inheriting from it (*Hierarchy Independence*, theorem 2). We show that we can deal with software changes that are not possible to verify in comparable frameworks.

A framework that allows the simplest reverification of any given software change, has the best flexibility. Clearly incremental and modular reasoning are preferable, as well as limiting the number of modules to be affected by a given change. It is desirable to avoid reverification of the whole system when possible. Flexibility depends on the choice of programming and specification constructs, their semantics, as well as the reasoning system. In particular flexibility is affected by the choice of abstraction mechanisms. For instance, for shared variable concurrency it is hard to analyze the effect of software changes, even with an advanced reasoning framework. And synchronization by signaling is notoriously hard to reason about. In the setting of behavioral subtyping, a change in a subclass may violate superclasses requirements, thereby limiting flexibility.

Flexibility demands programming languages with a compositional semantics and compositional reasoning frameworks. Compositional reasoning of classes is supported by several approaches. Our framework is based on a programming paradigm with compositional semantics, *cooperative scheduling* to support object-local synchronization control, using *interface abstraction* to reduce dependencies between classes, and the use of *communication histories* to enable compositional specification and reasoning.

In the presence of class inheritance, modularity of each subclass is advantageous, as cross-class dependencies hinder flexibility. The strong dependencies of behavioral subtyping can be reduced with the notion of *lazy behavioral subtyping* [8,9]; however, reasoning requirements to local calls in a superclass are imposed on subclasses, which limits flexibility. A framework for evolution based on this approach is given in [11].

We observe that changing a class *C* in the middle of a class hierarchy may in general affect existing subclasses as well as superclasses. Clearly code inherited from *C* in subclasses could lead to inconsistencies, since *C* is changed. And requirements imposed on *C* from superclasses may also lead to inconsistencies, something which may in general be remediated by changes in these superclasses, thereby affecting other subclasses of these superclasses as well. This makes reasoning about changes of classes difficult. However, the effect on superclasses depends on the semantics of class inheritance. Therefore the choice of class inheritance semantics is essential, in particular when it comes to inheritance of requirements. If a class is changed, it is undesirable that its superclasses also need to be modified, as this will destroy flexibility. This is the case in approaches where requirements are pushed from superclasses to subclasses, as in the case of behavioral subtyping.

In order to avoid this inherent flexibility limitation, we build on an approach with *separation of the reuse of code from the reuse of specifications* to allow unrestricted reuse of code and specifications. In particular we build on the approach of *behavioral interface subtyping* [20] where each class is only required to satisfy its own interface specifications, and any invariant or other local specifications given in the class. This means that a method redefined in a subclass is allowed to break the requirements of the superclass. This opens up for more liberal modifications than earlier work based on lazy behavioral subtyping [8,9]. As no superclass requirements are imposed on a subclass, this allows full control of the inheritance of code and of requirements when a subclass is defined, and when it is modified. In this way we may avoid inconsistent specifications due to inheritance. In our approach we can avoid inconsistencies due to superclass requirements, simply by controlling which requirements to inherit.

The notion of multiple inheritance allows adjustments in the inheritance hierarchy in the middle without removing existing inheritance relationships, simply by adding superclasses (and superinterfaces) as needed. This gives added flexibility during evolution, while allowing backwards compatibility. However, multiple inheritance has been criticized for too much flexibility and ambiguity issues, as exemplified in the diamond problem. We therefore add syntax for resolving ambiguities statically by using class names to limit the binding, and insisting on the healthiness condition suggested in [9], which implies that a local call appearing in a class *C* may only bind to a class below or above *C*, and not to a class in a different branch than that of *C*. Thus program changes in other branches than *C* will not affect the binding of such calls. The addition of superclasses during program evolution makes it possible to adjust the inheritance hierarchy and to reuse code from added superclasses. For instance, a service-oriented system defined by a class *S* defining online purchases of tickets of some kind may be extended with functionality for subscription to newsletters (of the relevant kind) and such that newsletters are sent to the subscribing customers. This extension can be done by adding the subscription class as an additional subclass of *S* and adding the relevant subscription

interface as an interfaces of *S*. Without support of multiple inheritance this extension would not be possible when *S* already has a subclass from before.

Our framework allows unrestricted changes of code and specifications (assuming type correctness). This means that one may write combinations of code and specifications that are inconsistent, for instance when a class does not satisfy the requirements of its interface(s). The framework will detect such inconsistencies so that they may be resolved, by changing code and/or specifications. In order to determine the consequences of changes in a (super)class, the framework needs to keep track of dependencies of local calls. We show that our framework can deal with software changes that are not possible to verify in comparable frameworks, and show how to reason within a hierarchy where some classes are verified and others not. We demonstrate our framework by examples.

Our approach is modular in the sense that the consistency of a class is determined by looking at the class itself, its interface(s), and reused code from superclasses. In order to analyze a software modification, one must first determine the affected code, in particular subclasses, and for each such subclass one must reverify the affected parts (after redesign of any inconsistent parts). Incremental reasoning is achieved by not letting a class impose restrictions on its subclasses. The present work extends the framework of [22] by adding multiple inheritance. As argued, multiple inheritance provides significant improvements in flexibility and simplicity during evolution since it enables added functionality just by adding superclasses and interface support in the middle of a class hierarchy, where needed. Thus multiple inheritance can be more useful in the program evolution phase than in the original program design phase.

Outline. Section 2 gives the programming setting for our framework, and Section 3 gives a summary of history-based specification and reasoning, including an example. Section 4 describes the proof obligations generated by our framework, simplifying [21]. In Section 5, we show how the framework is extended to deal with software changes. Finally, we discuss related work (Section 6) and give a conclusion (Section 7).

2 Language Setting

Our setting is distributed systems, and we focus on asynchronously communicating objects, so-called *active objects*, supporting blocking and non-blocking remote calls, without support of remote field access. In this setting, verification of a system of concurrent objects can be done compositionally, verifying each class separately, letting the specification of each class and interface refer to its local history [21]. The local history of an object reflects the time sequence of communication events such as method calls and returns, involving the object. Each class can be verified in a sequential manner, and a compositional rule states that a global invariant about the global history can be obtained by conjunction of the local invariants on local histories together with a well-formedness predicate relating the local histories to the global history.

We consider *multiple inheritance*, because this gives the freedom to extend the inheritance hierarchy during evolution, which greatly adds to the flexibility of changing programs. A class can then inherit from several superclasses while removing/adding/redefining method definitions, method specifications and invariants. As customary, we require a non-cyclic inheritance graph. And fields *w* may be added (an initial value *r* may

Pr	$::= [In^*\ Cl]^+$	program
In	$::=$ **interface** $F[$**extends** $F^+]^?\{S^*\ I^*\}$	interface declaration
Cl	$::=$ **class** $C[([T\ cp]^+)]^?\ [$**implements** $F^+]^?$	class definition
	$[$**inherits** $[C(\overline{e})]^+]^?\ [$**removing** $m^+]^?$	inheritance mechanisms
	$\{[T\ w\ [:= r]^?]^*\ s^?\ M^*\ S^*\ I^*\}$	class body
M	$::= T\ m([T\ x]^*)\ B\ P^*$	method definition
S	$::= T\ m([T\ x]^*)\ P^*$	method signature
B	$::= \{[[T\ x\ [:= r]^?]^+;]^?\ [s;]^?$ **return** $r\}$	method body
T	$::= F \mid$ Any $\mid$ Void $\mid$ Bool $\mid$ String $\mid$ Int $\mid$ Nat $\mid \ldots$	types
v	$::= x \mid w$	variables (local or field)
e	$::=$ null $\mid$ this $\mid$ caller $\mid v \mid cp \mid f(\overline{e}) \mid (e)$	pure expressions
r	$::= e \mid$ **new** $C(\overline{e}) \mid e.m(\overline{e}) \mid$ this $C.m(\overline{e}) \mid [C]^? : m(\overline{e})$	right-hand-side/call/new
s	$::=$ **skip** $\mid [v :=]^? r \mid s;s$	basic statements
	$\mid$ **await** $v := e.m(\overline{e}) \mid$ **await** e	suspending statements
	$\mid$ **if** e **then** $s\ [$**else** $s]^?$ **fi**	if statement
P	$::= [\ [A,A\ ]\ ]^+\ [$**where** $A^+]^?$	pre-/postcondition pairs
I	$::=$ **inv** $A^+\ [$**where** $A^+]^?$	invariant specification

Fig. 1. Language syntax. Specification elements are written in blue. F denotes an interface name, C a class name, m a method name, cp a formal class parameter, w a field, x a method parameter or local variable. We use $[\ ]$ as meta parentheses and superscipts $*$, $+$, and ? for repetition, non-empty repetition, and optional parts, respectively. Expressions e are side-effect free, and $\overline{e}$ denotes a (possibly empty) expression list. Assertions A are first order Boolean expressions and may refer to the local communication history **h**. A **where** clause defines auxiliary functions used for specification purposes. Other statements, such as while loops, can be added.

be given, otherwise the default value of the type is used). Class parameters are concatenated, and so are fields and initialization code. In case of a diamond-shaped inheritance, where the top superclass is inherited through several superclasses, the top superclass is inherited only once. This is achieved by the binding strategy. Method names (and field and class parameter names) can be qualified by a class name so that the occurrence is unique in the given class. This provides fine-grained control of the inherited names. For local calls we may use a class name to make the name unique, and similarly for fields and class parameters. Dot-notation as in $o.n(\ldots)$ and $this\ C.n(\ldots)$ is reserved for late-bound method calls, while the colon notation $C : n(\ldots)$ is reserved for static local method calls. If a field w is ambiguous due to multiple inheritance, we use the syntax $C{:}w$ for a field as seen in a superclass C. We insist on *healthy binding*, which means that an internal call made by a method defined in class C must bind to a class hereditarily related to C (as defined below).

We consider a core high-level imperative modeling language, given in Figure 1, inspired by the concurrency model of Creol [15] – extended to multiple inheritance. The language is executable with an interpreter in Rewriting logic/Maude [5]. The language is similar to that of [22], which considered only single inheritance. A program consists of a number of interfaces and classes. A class may implement a number of interfaces and inherit a number of (super)classes. The reflexive and transitive extension of the subclass relation is denoted $\leq$. If $A \leq B$, we say that A *is below* B, and B *is above* A; and

we say that A and B are *hereditarily related* if either A is below B or A is above B. Class instances represent concurrent and active objects. Local data structures are defined by (build-in or user-defined) data types. An interface can extend other (super)interfaces and add declarations of methods, behavioral constraints, and invariants.

A variable referring to an object is typed by an interface, not by a class. A variable declared of interface F is called an F variable. Through type checking the language guarantees that for an F variable, the object referred to by the variable at run-time implements F. This is called the *interface substitution principle* [23,15,17]. We distinguish between *public methods*, those exported through an interface of the class, and *private methods*, those that are not exported through any interface of the class. Note that interface abstraction defines the public-ness, rather than keywords such as private and public. Thus a public method in a class may be private in a subclass (and vice versa).

We allow remote calls of public methods with the syntax $v := o.m(\overline{e})$ where $\overline{e}$ the list of actual parameters and o is the callee. The value resulting from the call is assigned to the variable v. (The assignment part may be omitted if this value is not needed). A remote call $v := o.m(\overline{e})$ is type correct if the interface of o supports a method m such that the type of the actual parameters $\overline{e}$ is a subtype of the formal parameters of m and the output type of m is a subtype of the type of v. Since verification is done after type checking, we assume type correct programs, and assume that a class does not offer two declarations of the same method name. (If needed we could index the method name by the input and output type in order to make them distinct).

We allow both late-bound and static-bound local method calls, syntactically indicated by dot-notation and colon-notation, respectively. Local calls have the syntax $v := \mathsf{this}.m(\overline{e})$ and $v := \mathsf{this}\ C.m(\overline{e})$ (where C limits the binding of m) for late-bound calls, or $v := C\!:\!m(\overline{e})$ for static-bound calls, where this refers to the current object. We let this have the enclosing class as its type. Public methods are required to maintain the class invariant. Private methods may only be called locally, and may be called in states violating the invariant. A static local call $:m(\ldots)$ binds to the method m defined in the enclosing class C, if any, and otherwise to the closest inherited m, using a depth-first, then left-first, traversal of the superclasses of C. If neither C nor its superclasses has a method m, the call is statically illegal. The static local call $B:m(\ldots)$ (for $C \leq B$) binds to the method m defined in class B or inherited by B, as defined for a local call $:m(\ldots)$ appearing in class B. The class qualification (B) enables the programmer to select which version of a redefined method is needed. A late-bound local call $this\ B.m(\ldots)$ is legal if $B:m(\ldots)$ is legal and binds to the class closest to that of the executing object, as explained in detail in Sec. 3.2. The local call $this\ C.m(\ldots)$ when occurring in class C, may be abbreviated to $this.m(\ldots)$. Note that all legal calls will have a binding. Type checking ensures that there exists a binding, following [17].

In order to allow non-blocking calls, the language offers a suspension mechanism, programmed by **`await`** statements. An object may perform at most one process at a given time, and suspended processes are placed on a process queue local to the object. When the active process is suspended or completed (by a **`return`** statement), an enabled process from the process queue may be resumed. We consider conditional suspension (by means of a Boolean expression) and call-related suspension, suspending while the return value from a remote call has not arrived. The call **`await`** $v := o.m(\overline{e})$

suspends the current process and places the remaining part of the process in the queue, and it is enabled when the result from the call has arrived to the object. In the meantime the object may execute enables processes or handle incoming calls. Note that the callee o may be this, in which case the call will be done by the object. However, we may not suspend on a local call to a private method (since the syntax **await** $v := C : m(\overline{e})$ is not part of the language) as this would complicate the class invariant reasoning, as explained below.

The behavior of methods may be specified by pre/post specifications. This is needed for reasoning about local calls, for which the invariant may be violated, and is in particular useful for private methods, and other locally called methods. Multiple pre/post specifications of each method are allowed, and a class may implement multiple interfaces. A class without an **implements** list will implement the empty interface Any, which is the superinterface of all interfaces.

When an interface **extends** another (super)interface, all declarations and specifications are inherited. When a class **inherits** another class (the superclass), all code and specifications are inherited unless redefined: A pre/post pair (P) is inherited unless another is stated, an invariant (I) is inherited unless another is stated, the initialization code (s) is inherited unless another is stated, and a method body (B) is inherited unless the method is redefined or removed. Likewise, the implementation clause of the superclass is inherited unless a new implementation clause is provided, in which case the superclass implementation clause is not inherited.

The syntax **removing** $m_1, m_2 \ldots$ expresses that the listed methods should not be inherited, thereby defining "negative" inheritance. By type checking it must be ensured that public methods are not removed and that the remaining methods in C (including inherited ones) do not (directly or indirectly) lead to a call on a removed method. The purpose of a removal is to make a semantically simpler subclass, where irrelevant or problematic code is eliminated. In particular this can be used to make verification easier, and even avoid verification problems for instance when an invariant is redefined. Removal of fields will mainly be a typing issue. For simplicity, we assume read-only access to method and class parameters.

Apart from standard statements, we have included multiple inheritance, both static and late-bound calls, as well as cooperative scheduling and suspension allowing non-blocking calls, something which is useful in a distributed concurrent setting. Recursive calls are allowed, and while statements can easily be added.

3 History-Based Specification

The abstract state of an object is captured by the time sequence of communication events that have occurred so far involving the object. In a given state this sequence is finite. Thus finite communication sequences suffice for safety reasoning, called *histories*. Interface, class, and system specifications are expressed by means of histories. *Global histories* capture all communication events in a distributed system (or subsystem), and *local histories* capture all communication events seen from a given object. The local history $\mathbf{h}$ of an object o is part of the global history H, and these are related

by the equation $\mathbf{h} = H/o$ where H/o denotes the projection of the global history H to all communication events involving o as either the sender or receiver.

The invariant of an interface F may refer to the local history $\mathbf{h}$ and this, but not fields since these are not visible at the interface level. When seen from another class or interface with a larger alphabet, the F invariant must hold on the alphabet of F. The invariant of a class C may refer to fields, the local history $\mathbf{h}$, class parameters, and this. This invariant must be maintained by each public method of the class (possibly inherited), and a class must satisfy each implemented interface using projection on the history to reflect the subset of methods visible through the interface. A method specification may in addition refer to the formal parameters (including the caller) and logical variables (primed variables), and a postcondition may talk about the result (return). When seen from another class with a larger alphabet, a C invariant must hold on the alphabet of C.

The local history $\mathbf{h}$ of a class/interface is the time sequence of communications events seen by this object, considering the following kinds of events:

- a method call made by this object, denoted $\mathsf{this} \rightarrow o.m(\overline{e})$
- a method call received by this object, denoted $o \twoheadrightarrow \mathsf{this}.m(\overline{e})$ (for m in the class)
- a method return made by this object, denoted $o \leftarrow \mathsf{this}.m(\overline{e};e)$ (for m in the class)
- a method return received by this object, denoted $\mathsf{this} \twoheadleftarrow o.m(\overline{e};e)$, as well as
- a creation event made by this object, denoted $\mathsf{this} \rightarrow o.\,\mathbf{new}\,C(\overline{e})$

where o represents the other part in the communication. In practice, specifications using histories will often be concerned about method completions, i.e., $\leftarrow$ and $\twoheadleftarrow$ events, and possibly creation events, since these capture the essential input/output relations. (This is the case for our examples.) For a given method call, the $\leftarrow$ event precedes the $\twoheadleftarrow$ event, which is formalized by a wellformedness predicate (wf) below.

Sequence notation: A sequence is either *empty* or of the form $q;x$ where q is a sequence and x an element. The notation q/s denotes the projection of q restricted to elements in the set s, $q \leq q'$ denotes that q is a prefix (head subsequence) of q', $x\,\mathbf{before}\,x'\,\mathbf{in}\,q$ denotes that x appears before any occurrence of x' in q, i.e., $length(q'/x) \leq length(q'/x')$ for any prefix q' of q. For a global history H, there must be a meaningful ordering of the events, i.e., the history must be *wellformed*, defining $wf(H)$ by the conjunction of:

$$
\begin{array}{l}
(o \rightarrow o'.m(\overline{e}))\ \ \mathbf{before}\,(o \twoheadrightarrow o'.m(\overline{e}))\,\mathbf{in}\ H \\
(o \twoheadrightarrow o'.m(\overline{e}))\ \ \mathbf{before}\,(o \leftarrow o'.m(\overline{e};e))\,\mathbf{in}\ H \\
(o \leftarrow o'.m(\overline{e};e))\,\mathbf{before}\,(o \twoheadleftarrow o'.m(\overline{e};e))\,\mathbf{in}\ H \\
(o' \rightarrow o.\,\mathbf{new}\,C(\overline{e}))\,\mathbf{before}\,(o \rightarrow o''.m(\overline{e}'))\,\mathbf{in}\ H \\
(o' \rightarrow o.\,\mathbf{new}\,C(\overline{e}))\,\mathbf{before}\,(o'' \rightarrow o.m(\overline{e}'))\,\mathbf{in}\ H
\end{array}
$$

expressing that messages are sent before they are received, that method invocation must precede method return, and that a creation event of o must precede other o events. The conjunction of these properties (universally quantified) expresses the wellformedness predicate, used in the compositional rule for global reasoning. The rule for object composition essentially says that the global invariant is the conjunction of the wellformedness predicate and all object interface invariants, each referring to its own alphabet. Since the alphabets of the objects are by definition disjoint, the wellformedness predicate is needed to connect the different object invariants.

We let $\mathbf{h}/F$ denote the projection of a local history $\mathbf{h}$ to the events visible through F, i.e., events of the form $\mathsf{this} \rightarrow o.\,\mathbf{new}\,C(\overline{e})$, $\mathsf{this} \rightarrow o.m(\overline{e})$, and $\mathsf{this} \twoheadleftarrow o.m(\overline{e};e)$, as well as events of the form $o \twoheadrightarrow \mathsf{this}.m(\overline{e})$ and $o \leftarrow \mathsf{this}.m(\overline{e};e)$ for m offered by F. The same notation applies to classes C, projecting to $\mathsf{this} \rightarrow$ and $\mathsf{this} \twoheadleftarrow$ events as well as $\twoheadrightarrow \mathsf{this}.m$ and $\leftarrow \mathsf{this}.m$ for m defined or inherited in the class. An invariant $I(\mathbf{h})$ of an interface F is understood as $I(\mathbf{h}/F)$ in a subinterface or class. We therefore define $I_F(\mathbf{h})$ as $I(\mathbf{h}/F)$, and similarly for classes, defining $I_C(\mathbf{h}) \triangleq I(\mathbf{h}/C)$.

In general history-based invariant specification is more expressive than pre/post conditions since a pre/post pair (P,Q) of a method m with parameters $\overline{x}$ can be formulated as the invariant $(\mathsf{caller} \leftarrow \mathsf{this}.m(\overline{x};\mathsf{return})) \in h \wedge P \Rightarrow Q$ where P and Q may refer to this, caller, and $\overline{x}$, and Q also to return, and $\mathbf{h}$. For instance, one may express that the return event of a method *hire* implies that the object has received the return of a method *check budget* with OK as result. However, a specification expressible as a pre/post specification can be simpler to read and write than the corresponding invariant.

3.1 A Bank Example

Figure 2 shows a minimalistic example defining a class BANK and a subclass BANKPLUS, as well as related interfaces and a possible CLIENT class. The example is taken from [22]. The code illustrates suspension, non-blocking and blocking calls, static and late-bound local calls. Interface and type names are capitalized while class names are written in upper case letters. The keyword **`inv`** identifies invariants and the keyword **`where`** identifies auxiliary function definitions. In assertions, **`inv`** refers to the current invariant, while C : **`inv`** refers to the invariant of class C.

Interface Bank states that the balance (as returned by bal) is the sum of amounts deposited (by add) or withdrawn (by sub) from the bank account, ignoring unsuccessful add and sub calls. In addition it states that add calls always succeed. Interface PerfectBank extends Bank by stating that also sub calls succeed, while interface BankPlus extends Bank by stating that the balance is always non-negative. Interface Client (here omitted) includes methods salary (for receiving salary) and bill (for paying a bill).

The specifications of interface Bank and class CLIENT illustrate history-based specification, with inductive definitions of sum and allpaid. Functions are defined by a set of equations. The left hand sides can be seen as patterns, using underscore (_) to match any expression and letting ***others*** match any other case not covered by the other left hand sides. The auxiliary function *sum* calculates the balance from the local history. Note that only method-return events are used in the specification as other kinds of events are covered by the ***others*** equations. This is a typical situation for objects with "reactive" behavior as illustrated here.

The subclass BANKPLUS inherits the pre/post specifications of bal and add from BANK, but not the ones for upd and sub, which are redefined and therefore not inherited. In fact, the subclass violates the pre/post specifications for upd and sub in BANK. BANKPLUS does not support the BANK interface PerfectBank. Therefore the **`implements`** clause is redefined and not inherited. The await statement in class CLIENT allows the client to be responsive to salary reception calls and bill payment calls in case the *sub* call takes much time. However, it is then possible that two bills with the same kid are both paid. This would not be possible if the *sub* call is made as a blocking call.

```
interface Bank {
  Bool sub(Nat x)
  Bool add(Nat x) [true, return= true]
  Int bal() [true, return= sum(h)]
  where sum(empty) = 0,
        sum(h;(_←this.add(x;true))) = sum(h)+x,
        sum(h;(_←this.sub(x;true))) = sum(h)−x,
        sum(h; others) = sum(h) }

interface PerfectBank extends Bank {
  Bool sub(Nat x) [true, return= true] }

interface BankPlus extends Bank {
  inv sum(h)>=0 }

class BANK implements PerfectBank {
  Int bal:=0; -- a field defining the balance
  Bool upd(Int x) {bal:=bal+x; return true} [true, return= true]
    [inv, bal=sum(h)+x and return=true]
  Bool add(Nat x) {return this.upd(x)} [true, return= true]
  Bool sub(Nat x) {return this.upd(−x)} [true, return= true]
  Int bal() {return bal} [true, return=bal]
  inv bal=sum(h) }

class BANKPLUS implements BankPlus inherits BANK {
  Bool upd(Int x) {Bool ok:=(bal+x>=0);
    if ok then ok:=BANK:upd(x) fi; return ok}
      [inv, bal>=0 and bal=sum(h)+if return then x else 0]
      [b'=bal, return=(b'+x>= 0)]
  Bool sub(Nat x) [b'=bal, return=(b'>= x)]
  inv BANK:inv and bal >=0 }

class CLIENT implements Client {
  Seq[String] paid; -- a field keeping track of paid bills
  Bank acc:= new BANK; -- the banc account of the client
  Bool salary(Nat x) {return acc.add(x)}
  Bool bill(String kid, Nat x, Bank y) { Bool ok:=false;
    if kid ∉ paid then await ok:=acc.sub(x);
      if ok then y.add(x); paid:=(paid;kid) fi
    fi; return ok}
  inv paid=allpaid(h) --- paid corresponds to successful bill payments
  where allpaid(empty) = empty,
        allpaid(h;_←this.bill(k,x,y;true))=(allpaid(h);k),
        allpaid(h; others)=allpaid(h) }
```

Fig. 2. A bank example with a client class [22].

```
class BANK2 implements PerfectBank, BankPlus inherits BANKPLUS {
 Bool upd(Int x)
   {await bal+x>=0; bal:=bal+x; return true}
     [inv, bal=sum(h)+x and bal>=0 and return]
 Bool sub(Nat x) [true, return= true] }
```

Fig. 3. A possible subclass of class BANKPLUS [22].

In this example, the subclass does not obey the requirements imposed by behavioral subtyping, nor by lazy behavioral subtyping. The redefinition of upd in BANKPLUS does not satisfy the BANK postcondition of upd, and therefore the verification of the redefined upd will not succeed when using the framework of lazy behavioral subtyping (since the BANK postcondition of upd is needed for the local upd calls in the verification of BANK and therefore pushed to subclasses). In our framework, the BANK postcondition of upd is not imposed on the subclass, and the example can be verified without problems.

Figure 3 shows a subclass of BANKPLUS that could be meaningful in a distributed setting. A transaction is delayed as long as the balance is insufficient. This is done by means of an await statement, which suspends the sub activation, but does not block the object. Note that sub is inherited but not its specification. Class BANK2 implements the additional interface PerfectBank, and it inherits from BANKPLUS the invariant and all pre/post specifications, except the ones for upd and sub, which are violated. Again reasoning with behavioral subtyping or lazy behavioral subtyping breaks down, because the reasoning about the (late-bound) calls to upd in BANK depends on the postcondition of upd, and therefore it is imposed on all subclasses in the case of lazy behavioral subtyping. Our framework allows flexible reuse of code and specifications, without verification problems, avoiding harmful superclass requirements.

Consider next that class BANKPLUS is changed for instance by redefining sub by

```
Bool sub(Nat x) { return BANK:sub(x+1)}
```

A fee of 1 unit is incorporated in the withdrawal. In this case, class BANKPLUS can still be reverified, but the subclass BANK2 is indirectly affected by this change, and it is no longer a PerfectBank (because of the fee). Thus to avoid this inconsistency, class BANK2 should be modified, say by removing PerfectBank as an interface.

If a subclass of BANK redefines add and sub without using upd, that subclass may remove method upd. And a subclass of BANK implementing an interface with add, but not sub, and with the same class invariant as class BANKPLUS, may remove method sub in order to make invariant reasoning simpler.

3.2 Reasoning about late binding and static binding

Statically bound calls are resolved at compile time, while late bound calls are bound at runtime. In either case the behavior of the call depends on on the class of the object executing the method, called the *actual class*, since the behavior may (possibly indirectly) depend on late bound calls inside the method body. For $C1$ and $C2$ subclasses of C, it

may be that there is a local call $this.n(\bar{x})$ in method m of C, and if n is redefined in both $C1$ and $C2$, an m call will bind n differently depending on the actual class. For instance in the Bank example, a call to *sub* binds to BANK:*sub*, but the $this.upd$ call in the body of BANK:*sub* binds to BANKPLUS:upd or BANK:upd depending on the class of the executing object, BANKPLUS or BANK, respectively.

To formalize the binding of late-bound and static calls, we introduce three functions, $bind(A,m)$, $bind(A,B,m)$, and $bind(A,B,C,m)$, where $A \leq B$ and $B \leq C$. We let the function $bind(A,m)$ return A if it has a definition of m, otherwise the closest class with a definition of m considering the superclasses above A, using a depth-first, left-fist traversal. This is used for binding a static call $:m(...)$ appearing in class A and also for a static call $A:m(...)$ appearing in subclass of A. We let the function $bind(C,B,m)$ return C if C has a definition of m, otherwise the closest superclass of C with a definition of m using a depth-first, left-first search of the superclass hierarchy of C, restricted to classes hereditarily related to B. Similarly, $bind(C,B,A,m)$ returns the closest superclass of C hereditarily related to both B and C, using a depth-first, left-first search of the superclass hierarchy. When C is known, $bind(C,..,m)$ can be calculated, even with an open-ended class hierarchy.

A late-bound local call $this.m(\ldots)$ appearing in a class B binds to $bind(C,B,m)$ where C is the class of the executing object. The late-bound local call $this\ A.m(\ldots)$ appearing in a class B binds to $bind(C,B,A,m)$ with $B \leq A$ and C and B as above. For a late-bound local call the binding can be calculated statically for a given actual class of the executing object, $this$.

In the case of verification based on behavioral interface subtyping, we reconsider each possible actual class of this. Thus for each subclass of C (defined so far), we reconsider the verification of any inherited or reused methods. For each subclass C', the binding can then be done at verification time, binding a call this.m appearing in C to $bind(C',C,m)$ and binding $C:m(..)$ to $bind(C,m)$ as explained.

A complication in reasoning about local calls is that a release point (programmed by an **`await`** statement) should maintain the invariant of the actual class (say D) as opposed to the enclosing class (C). Thus reasoning about a release point occurring in a method $C:m$ must consider the invariant of the actual class, which may be a subclass (D) of C. We therefore index the derivation symbol ($\vdash$) both with the class of the executing object D as well as with the class of the enclosing object C, using the notation $\vdash_{D,C}$. In the setting of behavioral interface subtyping, reasoning is done for each choice of D. For a method inherited from C, we derive properties by means of $\vdash_{D,C}$, thereby letting all relevant proof obligations from C be reconsidered for each subclass D. In reasoning with behavioral subtyping, this is not needed since reasoning about method m of C is made (once) for all actual D. The latter approach makes reasoning simple when it succeeds, at the cost of redefinition flexibility – whereas in our system, based on behavioral interface subtyping, we may differentiate the different versions of an inherited method in the different subclasses. This gives more fine-grained reasoning (and specification) control, which is valuable in the setting of flexible code reuse and program evolution. A pre/post specification of m in C will be based on the invariant of C, which may be different from that of D. Therefore a pre/post specification of m in C cannot in general be guaranteed in a subclass D if $C:m$ has local calls or release points.

For example, consider two executions of a late-bound m call occurring in class A with $C1$ and $C2$ as the actual classes. These can be referred to by $C1:m$ and $C2:m$, respectively. We have that $bind(C1,A,m) = bind(C2,A,m)$ when the closest definition of m (hereditarily related to A) is in a common superclass of $C1$ and $C2$. A call to m with C as the actual class may cause a local call $\mathsf{this}.n(y)$ (directly or indirectly). In the verification, this call will then be re-analyzed with C as the actual class, using the binding $bind(C,A,n)$ where A is the class enclosing the call, and a static call $D:n(y)$ with C as the actual class will be re-analyzed using the binding $bind(D,m)$. The analysis of these call is the same when $bind(C,A,m) = bind(D,A,m)$ and the method body has no local calls to methods redefined below D and no release points.

For partial correctness reasoning, we consider theorems of the form

$$\vdash_{C,A} [P]\, s\, [Q]$$

where C is the actual class and A is the class enclosing s, and the Hoare triple $[P]\, s\, [Q]$ states that if the statement(list) s is executed in a state satisfying the precondition P the final state will satisfy the postcondition Q provided the execution of s terminates (using square brackets rather than curly brackets since the latter are part of the programming language syntax). Figure 4 presents sample proof rules needed for the example, modifying the rules in [22] (using a double-indexed proof symbol). Note that the axiom schema for assignment is as for sequential programs without aliasing. If we had allowed remote field access, this would no longer hold. The notation Q^v_e denotes (capture-free) textual substitution replacing all free occurrences of the variable v by the expression e. Similarly, $Q^{v,v'}_{e,e'}$ denotes simultaneous replacement (v by e and v' by e'). Rules for sequential composition and if-statements are as usual. Rules for while-loops and recursive calls are also standard, but are omitted here for brevity.

For a class C we use $\vdash_{C,A}$ to prove the pre/post verification conditions for objects of that class, for code inherited from A. For code in class C this corresponds to normal class-based reasoning ($\vdash_{C,C}$). For code inherited from A, reasoning about release points and local or self calls depends on C, which reflects the actual class of this object, as well as A. Note that reasoning about late-bound self calls reduces to reasoning about static local calls: According to rule self call, the late-bound self call $v := \mathsf{this}.B.m(\bar{x})$ is equivalent to the static call $v := D:m(\bar{x})$ where D is given by $bind(C,A,B,m)$. Thus the binding depends on the class of the executing object C, restricted by A and B. The binding $bind(C,A,B,m)$ can be calculated at verification time since C, A, and B are known. We have that the self call $v := \mathsf{this}.m(\bar{e})$ abbreviates $v := \mathsf{this}\ A.m(\bar{e})$ where A is the enclosing class, and similarly that the static call $v := \ :m(\bar{e})$ abbreviates $v := A:m(\bar{e})$. Thus rules for these special cases are omitted. For instance, reasoning about the late-bound call $v := \mathsf{this}.m(\bar{x})$ reduces to reasoning about the static $v :=: m(\bar{x})$ if the class of this has a redefinition of m. Rule static call states that reasoning about $v := B:m(\bar{e})$ reduces to reasoning about $body_{bind(B,m):m}$, adding effects on the history, where $body_{C:m}$ denotes the body of the definition of method m in class C.

The *body* of a method definition $m(\bar{x})\{s;\ \mathtt{return}\ e\}$ is given by

$$\begin{array}{l}\mathbf{h} := (\mathbf{h}; caller \twoheadrightarrow \mathsf{this}.m(\bar{x}));\\ s;\ \mathsf{return} := e;\\ \mathbf{h} := (\mathbf{h}; caller \leftarrow \mathsf{this}.m(\bar{x}; \mathsf{return}))\end{array}$$

assign
$$\vdash_{C,A} [Q^{v}_{e}]\, v := e\, [Q]$$

history
$$\vdash_{C,A} [h' = \mathbf{h}]\, s\, [h' \leq \mathbf{h}]$$

await guard
$$\vdash_{C,A} [I_C \wedge L]\ \texttt{await}\ b\, [b \wedge I_C \wedge L]$$

new
$$\vdash_{C,A} [\forall v' .\, \mathit{fresh}(v', \mathbf{h}) \Rightarrow Q^{v,\mathbf{h}}_{v',\mathbf{h};(\mathsf{this} \rightarrow v'.\, \texttt{new}\, C(\overline{e}))}]\, v := \texttt{new}\ C(\overline{e})\, [Q]$$

simple call
$$\vdash_{C,A} [Q^{\mathbf{h}}_{\mathbf{h};(\mathsf{this} \rightarrow o.m(\overline{e}))}]\, o.m(\overline{e})\, [Q]$$

blocking call
$$\vdash_{C,A} [\forall v' .\, o \neq \mathsf{this} \wedge Q^{v,\mathbf{h}}_{v',\mathbf{h};(\mathsf{this} \rightarrow o.m(\overline{e}));(\mathsf{this} \twoheadleftarrow o.m(\overline{e};v'))}]\, v := o.m(\overline{e})\, [Q]$$

non-blocking call
$$\frac{\vdash_{C,A} [P]\ \texttt{await}\ \mathit{true}\, [\forall v' .\, Q^{v,\mathbf{h}}_{v',\mathbf{h};(\mathsf{this} \twoheadleftarrow o.m(\overline{e};v'))}]}{\vdash_{C,A} [P^{\mathbf{h}}_{\mathbf{h};(\mathsf{this} \rightarrow o.m(\overline{e}))}]\ \texttt{await}\ v := o.m(\overline{e})\, [Q]}$$

self call
$$\frac{\vdash_{C,A} [P]\, v := \mathit{bind}(C,A,B,m) : m(\overline{e})\, [Q]}{\vdash_{C,A} [P]\, v := \mathsf{this}\ B.m(\overline{e})\, [Q]}$$

implicit self call
$$\frac{\vdash_{C,A} [P]\, v := \mathit{bind}(C,A,m) : m(\overline{e})\, [Q]}{\vdash_{C,A} [o = \mathsf{this} \wedge P]\, v := o.m(\overline{e})\, [Q]}$$

static call
$$\frac{\vdash_{C,A} [P]\, \mathit{body}_{\mathit{bind}(B,m):m}\, [Q^{\mathbf{h}}_{\mathbf{h};(\mathsf{this} \twoheadleftarrow \mathsf{this}.m(\overline{e};v))}]}{\vdash_{C,A} [P^{\overline{x},\mathsf{caller},\mathbf{h}}_{\overline{e},\mathsf{this},\mathbf{h};(\mathsf{this} \rightarrow \mathsf{this}.m(\overline{e}))} \wedge L]\, v := B : m(\overline{e})\, [Q^{\overline{x},\mathsf{caller},\mathsf{return}}_{\overline{e},\mathsf{this},v} \wedge L]}$$

sequence
$$\frac{\vdash_{C,A} [P]\, s\, [Q] \qquad \vdash_{C,A} [Q]\, s'\, [R]}{\vdash_{C,A} [P]\, s; s'\, [R]}$$

if-then-else
$$\frac{\vdash_{C,A} [P \wedge b]\, s\, [Q] \qquad \vdash_{C,A} [P \wedge \neg b]\, s'\, [Q]}{\vdash_{C,A} [P]\ \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s'\ \texttt{fi}\ [Q]}$$

Fig. 4. Hoare-style rules and axioms. Primed variables represent fresh logical variables, $\mathit{fresh}(v', \mathbf{h})$ expresses that v' does not occur in $\mathbf{h}$, and L denotes a local assertion, i.e., without occurrences of fields. In rules self call and static call, we assume that v does not occur in e (otherwise we would need a primed variable, v'). In rule static call we assume that $\overline{x}$ is the formal parameter list (which is read-only). Note that binding is calculated at verification time.

incorporating the effects on the local history reflecting method call reception and method return. Since each class is analyzed separately, we obtain a modular and incremental verification system suitable for an open-ended class hierarchy, not unlike [7]. In the analysis of a class C we may need to consider superclasses of C, but not subclasses. We may reuse superclass verification results as follows: For code inherited from a superclass B, we may derive $\vdash_{C,B} [P]\, s\, [Q]$ from $\vdash_{B,B} [P]\, s\, [Q]$ when s has no release points and no local calls leading to calls of methods redefined below B. Otherwise $\vdash_{C,B} [P]\, s\, [Q]$ can be established by a new analysis of s and of any locally called methods in s. In par-

ticular $\vdash_{C,B} [P]\, v := B{:}m(\bar{x})\, [Q]$ follows from $\vdash_{B,B} [P]\, v := B{:}m(\bar{x})\, [Q]$ when $B{:}m$ has no release points nor local calls. In contrast to behavioral subtyping and lazy behavioral subtyping, no requirements are imposed on subclasses.

4 Proof Obligations

For each class C we must ensure that it satisfies the stated requirements, i.e., that the **`implements`** clause is satisfied (syntactically and semantically), that the class invariants are maintained by each method (except private ones), and that the stated pre/post specifications are satisfied by the corresponding methods of the class.

In this proof, inherited methods must be considered, while superclass implementation claims, superclass invariants, and superclass pre/post specifications, are not considered unless inherited. Each class is verified in this sense, taking inherited superclass code into consideration. Together with correct typing of object variables, this ensures that each object variable will satisfy its declared interfaces, and each object of run-time class C will satisfy the interfaces of C. This ensures that the compositional rule (Section 5.1) for reasoning about active object systems is sound. Furthermore, each late-bound local call with C as the run-time class of the caller/callee will satisfy the pre/post specification given in C since class C is statically verified. This is reflected in the composition rule, which considers all verified callee classes.

We formalize the proof obligations expressing the correctness of a program, a class, an interface claim, a class invariant, and a method specification. We define the following proof obligations, identifying the actual class and the enclosing class:

Definition 1 (Program and Class Correctness).
A program P is correct*, denoted $\vdash P$ **`ok`**, if each class in the program is correct.*
A class C is correct*, denoted $\vdash C$ **`ok`**, iff*

- $\vdash C$ **`sat`** F *for each interface F specified in the* **`implements`** *clause of C,*
- $\vdash C$ **`inv`** I *for each stated (or explicitly inherited) invariant I of C,*
- $\vdash_{C,C} m(\bar{x})$ **`sat`** $[P,Q]$ *for each method $m(\bar{x})$ defined in C, and each specification $[P,Q]$ stated (or inherited) in C for m,*
- $\vdash_{C,A} m(\bar{x})$ **`sat`** $[P,Q]$ *for each method $m(\bar{x})$ inherited from A, and each specification $[P,Q]$ stated (or inherited) in C for m,*

where

- $\vdash_{C,A} m(\bar{x})$ **`sat`** $[P,Q]$ *is verified by proving $\vdash_{C,A} [P]\, body_{bind(C,A,m):m}\, [Q]$ (as explained above).*
- $\vdash C$ **`inv`** I *is verified by proving $\vdash_{C,C} m(\bar{x})$ **`sat`** $[I,I]$ for each public method $m(\bar{x})$ in C, and by proving $\vdash_{C,A} m(\bar{x})$ **`sat`** $[I,I]$ for each public method $m(\bar{x})$ inherited from A, and by proving that the invariant holds initially, i.e., I holds when $\boldsymbol{h}$ is replaced by the empty history and fields by initial values.*
- $\vdash C$ **`sat`** F *is verified by proving that the conjunction of the invariants $I_i(\boldsymbol{h})$ of C implies the invariant of F, I_F (considering methods visible through F, as explained in Section 3):*

$$\wedge_i I_i(\boldsymbol{h}) \Rightarrow I_F(\boldsymbol{h}/F)$$

Note that type checking ensures that all methods of F are offered in C, with a signature better or equal to that of F (i.e., contravariant parameter types and covariant return types). And it ensures that removed methods are not directly or indirectly called from C, and that private methods of C are not directly or indirectly called with **await**.

For a subclass C' of C, $\vdash_{C,A} m(\bar{x})$ **sat** $[P,Q]$ need not imply $\vdash_{C',A} m(\bar{x})$ **sat** $[P,Q]$ even if m is not redefined, since the binding of local calls appearing in the body of m in C may bind differently in the context of C' (i.e., $bind(C,A,n)$ versus $bind(C',A,n)$, respectively). In general $\vdash_{C,A} m(\bar{x})$ **sat** $[P,Q]$ depends on redefinition of m or locally called methods and possible C invariants in case of suspension (by **await** statements). A redefinition in C' of a locally called method may violate the supertype specification of that method. A suspension point performed on a C' object can only guarantee that the C' invariant is maintained, which could be weaker than the C invariant. We therefore track these dependencies, and we may conclude that $\vdash_{C,A} m(\bar{x})$ **sat** $[P,Q]$ implies $\vdash_{C',A} m(\bar{x})$ **sat** $[P,Q]$ if $\vdash_C m(\bar{x})$ **sat** $[P,Q]$ does not depend on any redefined code and that any invariant used in the verification is respected by C'.

For a method m defined in B without local calls or suspension points, we have that the theorem $\vdash_{C,A} B{:}m(\bar{x})$ **sat** $[P,Q]$ reduces to $\vdash_{C,C} m(\bar{x})$ **sat** $[P,Q]$. This gives a practical way of reusing proofs from superclasses.

4.1 Verification of the Bank example

Let B denote BANK and BP denote BANKPLUS. Let I_B denote the invariant of B and I_{BP} that of BP. According to our definition of class correctness, we get the following verification conditions for class BANK ($\vdash B$ **ok**):

$$\vdash \quad I_B \Rightarrow I_{PerfectBank}(\mathbf{h}/PerfectBank) \tag{1}$$
$$\vdash_{B,B} bal(x) \textbf{ sat } [true, \mathsf{return} = bal] \tag{2}$$
$$\vdash_{B,B} add(x) \textbf{ sat } [true, \mathsf{return} = true] \tag{3}$$
$$\vdash_{B,B} sub(x) \textbf{ sat } [true, \mathsf{return} = true] \tag{4}$$
$$\vdash \quad I_B{}^{\mathbf{h},bal}_{empty,0} \tag{5}$$
$$\vdash_{B,B} bal(x) \textbf{ sat } [I_B, I_B] \tag{6}$$
$$\vdash_{B,B} add(x) \textbf{ sat } [I_B, I_B] \tag{7}$$
$$\vdash_{B,B} sub(x) \textbf{ sat } [I_B, I_B] \tag{8}$$
$$\vdash_{B,B} upd(x) \textbf{ sat } [I_B, bal = sum(\mathbf{h}) + x \wedge \mathsf{return} = true] \tag{9}$$

In addition we must verify the PerfectBank pre/post conditions, which follow by the corresponding BANK pre/post conditions (2,3,4). In particular, the postcondition return $=$ $sum(\mathbf{h}/\mathsf{PerfectBank})$ follows by (2) and I_B. Here (1) represents the entailment of the PerfectBank invariant, which in this case is an empty obligation since PerfectBank has no invariant, (2,3,4) are requirements from BANK, (5) states that I_B holds initially, (6,7,8) state the invariance of I_B, and (9) represents the additional pre/post specifications of upd given in BANK. Verification conditions (2,5,6) and (9) are trivial, (3,4) and (7,8) follow from (9), treating the **return** e of a public method $m(\bar{x})$ as the assignment return $:= e$, followed by $\mathbf{h} := (\mathbf{h}; (\mathsf{caller} \leftarrow \mathsf{this}.m(\bar{x}; \mathsf{return})))$ according to the definition of $body$.

For class BANKPLUS we must verify $\vdash BP$ **ok**, which amounts to the verification

$$\vdash \quad I_{BP} \Rightarrow sum(\mathbf{h}/BankPlus) \geq 0 \tag{10}$$
$$\vdash_{BP,B} bal(x)\ \mathtt{sat}\ [true, \mathsf{return} = bal] \tag{11}$$
$$\vdash_{BP,B} add(x)\ \mathtt{sat}\ [true, \mathsf{return} = true] \tag{12}$$
$$\vdash \quad I_{BP}{}^{\mathbf{h},bal}_{empty,0} \tag{13}$$
$$\vdash_{BP,B} bal(x)\ \mathtt{sat}\ [I_{BP}, I_{BP}] \tag{14}$$
$$\vdash_{BP,B} add(x)\ \mathtt{sat}\ [I_{BP}, I_{BP}] \tag{15}$$
$$\vdash_{BP,B} sub(x)\ \mathtt{sat}\ [I_{BP}, I_{BP}] \tag{16}$$
$$\vdash_{BP,BP} upd(x)\ \mathtt{sat}\ [I_{BP}, bal \geq 0 \wedge bal = sum(\mathbf{h}) + \ \mathtt{if}\ \mathsf{return}\ \mathtt{then}\ x\ \mathtt{else}\ 0] \tag{17}$$
$$\vdash_{BP,BP} upd(x)\ \mathtt{sat}\ [b' = bal, \mathsf{return} = (b' + x \geq 0)] \tag{18}$$

Fig. 5. Verification conditions for class BANKPLUS.

$$\vdash \quad I_{B2} \Rightarrow sum(\mathbf{h}/BankPlus) \geq 0 \tag{19}$$
$$\vdash_{B2,B} bal(x)\ \mathtt{sat}\ [true, \mathsf{return} = bal] \tag{20}$$
$$\vdash_{B2,B} add(x)\ \mathtt{sat}\ [true, \mathsf{return} = true] \tag{21}$$
$$\vdash_{B2,B} sub(x)\ \mathtt{sat}\ [true, \mathsf{return} = true] \tag{22}$$
$$\vdash \quad I_{B2}{}^{\mathbf{h},bal}_{empty,0} \tag{23}$$
$$\vdash_{B2,B} bal(x)\ \mathtt{sat}\ [I_{B2}, I_{B2}] \tag{24}$$
$$\vdash_{B2,B} add(x)\ \mathtt{sat}\ [I_{B2}, I_{B2}] \tag{25}$$
$$\vdash_{B2,B} sub(x)\ \mathtt{sat}\ [I_{B2}, I_{B2}] \tag{26}$$
$$\vdash_{B2,B2} upd(x)\ \mathtt{sat}\ [I_{B2}, bal = sum(\mathbf{h}) + x \wedge bal \geq 0 \wedge \mathsf{return} = true] \tag{27}$$

Fig. 6. Verification conditions for class BANK 2 .

conditions given in Figure 4.1. These represent the entailment of the BankPlus invariant (10), the inherited pre/post specifications of BankPlus (11,12), the initial satisfaction of I_{BP} (13), the invariance of I_{BP} (14-16), and the pre/post specification of upd given in BANKPLUS (17). Here (10,13,14) are trivial and (11) reduces to (2) by observing that $\vdash_{BP,B} bal(x)\ \mathtt{sat}\ [P,Q]$ equals $\vdash_{B,B} bal(x)\ \mathtt{sat}\ [P,Q]$ (for any $[P,Q]$) since there are no local calls nor release points. Then (12,15,16) follows by using (17). For the local call in the redefined upd we observe that proofs about $B{:}upd(x)$ do not depend on the actual class since the body has no local calls. We may therefore reuse the specification of upd from BANK when analyzing the call $\mathsf{BANK}{:}upd(x)$ in class BANKPLUS. Then verification of (17) is straightforward, and verification of (18) reduces to the trivial condition $b' = bal \Rightarrow \ \mathtt{if}\ bal + x \geq 0\ \mathtt{then}\ b' + x \geq 0 = true\ \mathtt{else}\ b' + x \geq 0 = false$. Moreover, the verification above can easily be mechanized.

Consider BANK2, abbreviated $B2$, of Figure 3. We have that I_{B2} is I_{BP}. Figure 4.1 gives the verification obligations for $\vdash B2\ \mathtt{ok}$. Here (19,23) reduce to (10,13) since I_{B2} is the same as I_{BP}. Since reasoning about $bal(x)$ does not depend on the actual class, (20) reduces to (2). Furthermore, (24) is trivial, and (20,22,25,26) follow by (27). For

(27) we use Hoare-style reasoning and must verify that the given pre/post specification is satisfied by the body of *upd*, which is:

$$\textbf{await}\ bal + x \geq 0; bal := bal + x; \mathsf{return} := true; \mathbf{h} := (\mathbf{h}; (caller \leftarrow \mathsf{this}.upd(x; \mathsf{return})))$$

(since we may here ignore all $\rightarrow$ events) which reduces to the condition

$$I_{B2} \wedge bal + x \geq 0 \Rightarrow (bal + x = sum(\mathbf{h}; (caller \leftarrow \mathsf{this}.upd(x; true))) + x \wedge bal + x \geq 0)$$

which is trivial since *upd* events do not affect *sum* due to the ***others*** equation. The example shows that: Verification of a class is done by inspecting the class and its superclasses, and does not impose any proof obligations on subclasses. Reasoning about static and late-bound local calls are handled by the actual class context. Proof obligations can often be reduced to already verified superclass obligations. The verification conditions we have seen are easily verified and thus easily automated.

5 Evolutionary Program Changes

During evolution of a system there may be a series of program changes, including changes of existing classes as well as additions of new classes and interfaces. For instance, an existing class in the middle of a class hierarchy may by augmented by adding a new class as a superclass and by adding new implementation clauses. And one may introduce a new interface to make two independent subsystems interact, adding support of the new interface in one or more existing classes.

In general, an existing class D may be changed by adding methods and fields, replacing methods, changing inheritance clauses, implementation clauses, removal clauses, and/or specifications. This can be understood by replacing the whole class definition by another definition. The updated class D may in general have a number of subclasses (at the time when D is updated) and these are implicitly modified if they inherit or reuse code from D. Thus, we need to reverify the redefined D, but in addition we need to consider the affected subclasses of D.

Definition 2 (System Change). *A system change is given as a sequence of* **`introduce`** *and* **`update`** *definitions. We use the syntax* **`introduce`** *In for adding an interface definition In and the syntax* **`introduce`** *Cl for adding a class definition Cl, with In and Cl as defined in Figure 1. We use the following syntax for defining class updates:*

$$\begin{array}{l}\textbf{update class}\ D\,[([T\ cp]^+)]^?\\ [\,\textbf{implements also}^?\ F^+]^?\\ [\,\textbf{inherits also}^?\ C(\overline{e})]^?\\ [\,\textbf{removing also}^?\ m^+]^?\\ \{[T\ w\ [:= r]^?]^*\ s^?\ M^*\ S^*\ I^*\}\end{array}$$

This class update modifies an existing class D by adding class parameters cp^+ (if present), changing the interface support to F^+ (if present), adding superclasses $[C(\overline{e})]^+$ (if present), removing methods m^+ (if present), adding fields w^+ (if present), adding initialization code s (if present), adding/redefining method definitions M^* (if present),

changing method specifications S^* (if present), and changing the invariant to I^* (if present). For any optional item omitted, there is no change from the original class. This is somewhat similar to the semantics of inheritance, except that the modifications are made on an existing class rather than a new subclass. In order to limit duplication of old code, we use the quasi class name *OLD* to refer to elements of the original version of the class, thus the redefinition of a method *m* may contain the call $OLD : m(...)$ to reuse the old version of *m*. In contrast to static calls, such a call is textually expanded using the original definition (since the original definition may be removed). Similarly, $OLD :$ **inv** expands to the old invariant. We may use the keyword **also** in **implements**, **inherits**, and **removes** clauses, to define added elements. Thus **inherits also** *C* means that the updated class inherits *C* in addition to the classes inherited by the original version of the class.

An example of a class update is given in Fig. 7. Here the transaction-oriented bank version given by BANK 2 is changed so that one can check earlier transactions. This is done by letting the BANK 2 class inherit SAFETRANS in addition to the old superclass, thereby using multiple inheritance. The SAFETRANS class stores transactions in a secure manner, by giving limited read access, through checking if a given transaction has happened or not, and restricting write access to append. (For brevity the class is minimalistic.) The *upd* method of BANK 2 is then updated using the append method of SAFETRANS. The added invariant states that the transactions defined by SAFETRANS corresponds to the history as defined in Bank. Note that $sum(\mathbf{h})$ is here understood as $sum(\mathbf{h}/Bank)$. The example shows the usefulness of multiple inheritance during evolution. The updated BANK 2 class supports the old interfaces (PerfectBank and BankPlus), so any previous usage of BANK 2 objects through these interfaces is not affected by the change. After the update, BANK 2 objects can also be used through the SafeTrans interface. One needs to verify that the updated *upd* method satisfies the (inherited) conditions and the new invariant. This will be quite straight forward in this example.

We consider correctness of the updated code, and avoid complications such as runtime upgrades where new and old versions of the updated code are part of the running system. As before we assume type correctness. In general, the redefined class *D* (let us refer to it as $\hat{D}$) implements some interfaces, which may or may not be the same as for *D*. If $\hat{D}$ includes all interfaces of *D*, all type correct calls to *D* objects will be type correct and supported by $\hat{D}$ objects as well; and if the interface specifications are the same, global reasoning from interface specifications of *D* objects is not violated by replacing *D* objects by $\hat{D}$ objects.

Consider next the case that a class is modified by removing the support for an interface. In this case the statement $v :=$ **new** *D*, becomes illegal when class *D* is modified so that it no longer supports the interface type of variable *v*. We may then change the statement to $v :=$ **new** *B* where *B* supports the interface. In general we may need a sequence of changes in order to obtain a desired resulting program, including changes to *C* and other classes using *D*. (Subclasses that inherit the interface clause of *D* may then explicitly add support for the interface, when desirable.)

The verification obligations caused by the redefinition consist of verifying $\vdash \hat{D}$ **ok** and reverification of the subclasses of *D* since they may be affected by the change. We first mark the obligation $\vdash D$ **ok**, as well as all sub-obligations, as *pending*. And for

```
introduce interface SafeTrans { -- may append and check data,
                                -- but not change data
  Void append(Int x)
  Bool checkTrans(Int x) [true, return= (_←this.append(x)∈h)] }

introduce class SAFETRANS implements SafeTrans {List[Int] trans= empty;
  Void append (Int x) {trans:=(trans;x)}
  Bool checkTrans(Int x) {return x∈trans} }

update class BANK2 implements also SafeTrans inherits also SAFETRANS {
  Bool upd(Int x) {OLD:upd(x); SAFETRANS:append(x); return true}
  inv OLD:inv and sum(h)=add(trans)
  where add(empty)=0
        add(trans;x)=add(trans)+x }
```

Fig. 7. An update of Bank2 causing multiple inheritance.

each subclass D' we mark the obligation $\vdash D'$ **ok**, as well as all sub-obligations, as *pending*. Verification of $\vdash \hat{D}$ **ok** is then done as defined above for the class resulting from the update, and the subclasses of D must be reverified. If an obligation depends on a pending sub-obligation, one should consider the latter first. Since subclasses may depend on classes defined earlier (as substantiated by Theorem 1 below), we reconsider the subclasses in the order defined. For a subclass D', the obligation $\vdash D'$ **ok** should be marked as *pending* if the proof made use of a result from D, say $\vdash_{D,A} m(v)$ **sat** $[P,Q]$. For each such D result, it suffices to prove $\vdash_{\hat{D},A} m(v)$ **sat** $[P,Q]$. If all sub-obligations of $\vdash D'$ **ok** can be reverified in this manner, the obligation $\vdash D'$ **ok** is marked *correct*.

The *state* of a proof obligation indicates whether it has been proved or not. We consider the states: *correct, incorrect, pending*. These express respectively that the obligation is verified, that the (old) proof is no longer valid, that verification remains to be done. As explained above, if a pending obligation can be verified or be reduced to a correct obligation, its state can be reset to *correct*. If a pending obligation cannot be verified, its state can be set to *incorrect*. In some cases it may be possible to reverify the obligation using additional specifications of inherited or called methods, but in general this may require human insight. Otherwise further modifications are needed.

An advantage of our approach is that violations in a class C caused by superclass modifications can be handled without changing the superclasses of C, called *Modification Independence*:

Theorem 1 (Modification Independence). *Assume that a class C is affected by a superclass modification such that some inherited superclass specifications are violated in C. Then C can be modified such that there is no violation.*

Proof. Let $[P,Q]$ be a violated m-specification. If this specification is inherited, we simply change C by not inheriting it and then the specification is no longer required in C. And if $[P,Q]$ is stated in C, we remove the specification. In case $[I,I]$ is then removed

for an invariant I, we also remove the invariant from C, and remove any interface of C depending on the invariant. We repeat this process until all violations are removed. □

This means that any undesired requirements due to modifications in a superclass can be removed. After removal one may add desired requirements and verify these requirements (modifying the class if needed). In this way one may reverify that the stated interfaces are satisfied. This gives full flexibility of properties during evolution, at the cost of reconsidering subclasses in case the modifications require changes in subclasses.

Our framework supports independence between different branches of a hierarchy, a property which is essential for flexible evolution. However, this kind of Hierarchy Independence is non-trivial, especially in presence of multiple inheritance.

Theorem 2 (Hierarchy Independence).
Modification of a class C will only affect C and subclasses of C.

Proof. Let D be a class other than a subclass of C. The case when C is a subclass of D follows by the theorem above. Thus we may assume that C and D are hereditarily unrelated, but in the presence of multiple inheritance they may have common superclasses and common subclasses. By our healthiness condition on the binding strategy, a late-bound call in D cannot bind to a method defined in C because the healthiness condition then requires that C and D are hereditarily related. And it cannot bind to a common subclass of C and D because we may assume that D is the executing object, and thus all late-bound calls will bind to a method defined in D or a superclass. Such a call may bind to a common superclass (of C and D), but by theorem 1, this superclass is not affected by the change in C.

A static-bound call occurring in D may bind to superclass of D which may contain a late-bound call. However, our binding strategy ensures that this is call binds to a class above D due to the healthiness condition. Thus it cannot bind to C. □

5.1 Reasoning in presence of unverified classes.

Our approach may result in some verified classes and some classes that are not yet verified. In this imperfect setting we may still reason about the overall system by using the following formulation of the global system invariant $I(H)$ over the global history H (i.e., the sequence of all events that have occurred so far in the total system):

$$I(H) \triangleq wf(H) \bigwedge_{\to o.\,\mathbf{new}\,C(\_)\in H} \bigwedge_{F\in C} I_F(H/o)^{\mathsf{this}}_o$$

where C is restricted to range over classes that are tagged *correct*, i.e., those satisfying $\vdash C\,\mathbf{ok}$. The last conjunction ranges over all interfaces F implemented by C. Here $wf(H)$ denotes the wellformedness predicate, expressing the **`before`** ordering between events, given in Section 3.

This global invariant captures the partial knowledge of the global history H given by the interface invariants of the objects appearing in the system (possibly dynamically generated) considering only objects of *correct* classes. This global reasoning rule essentially turns off the interface invariants for the non-correct classes.

Limitation: We assume type correctness since reverification of a modified class C will be preceded by type checking of the modified class and other existing classes using C in creation statements. Thus we consider only program changes that result in type correct programs. Removal of declarations of fields, methods, parameters, and variables is therefore only allowed when not in use. Secondly we do not consider changes in an interface I. This can be simulated by adding the new version of I as a separate interface, making changes wherever I (or a subinterface) is used, and then removing the original I when no longer referred to.

5.2 Examples of Software Changes on BANK

Assume now that class BANK is changed so that upd calls checkAvail, which returns true.

```
update class BANK implements PerfectBank {
  Bool checkAvail(Int x){return true} [true, return]
  Bool upd(Int x){Bool ok:=checkAvail(x);
    if ok then bal:=bal+x fi; return ok}
   [inv, bal=sum(h)+x and return=true] }
```

All other aspects of class BANK are kept unchanged, including all BANK specifications. Thus **inv** refers to the original invariant of BANK. Since checkAvail returns true, the verification of upd can be reused, and the other verification conditions of BANK are as before and need not be reverified. And the verification of the added local method checkAvail is trivial. Furthermore, the subclasses are not affected by this change. Thus the verification conditions caused by the class update are straightforward.

However, if class BANKPLUS is changed by redefining checkAvail(x) as in

```
update class BANKPLUS {
  Bool checkAvail(Int x){return bal+x>0} }
```

the local late-bound call to upd$(-x)$ in the inherited method sub results in the value of $bal - x > 0$ to be returned from sub. Again verification conditions are straightforward. In contrast this could not be verified in the frameworks of [10,11].

Adding a side effect in checkAvail such as **if** $x < 0$ **then** $bal := bal - 1$ **fi** would destroy the BANK invariant, but not the BANKPLUS invariant. Then the former should be removed.

Consider next the following update of class BANK with a redefinition of sub:

```
update class BANK { Bool sub(Nat x)
  { bal:=bal−x; return true} }
```

The new version of BANK inherits the old interface (PerfectBank), the methods add, bal, and upd, the old invariant, the old specification of sub (i.e., postcondition return $= true$). The proof obligations amount to first verifying that the redefined sub maintains the invariant and satisfies the postcondition. This is trivial. Secondly it must be verified that each subclass is still **ok**. As subclass BANKPLUS now may allow a negative balance, the BANKPLUS invariant $bal \geq 0$ cannot be verified (because it is incorrect). We may still do (limited) global invariant reasoning about a system containing BANKPLUS objects.

To solve this inconsistency in BANKPLUS, we may update this class by removing the support of interface BankPlus and removing the last conjunct of the invariant and the specification of sub, and then reverify. Alternatively, we may change BANKPLUS by redefining sub so that the old specifications can be reverified.

Finally, the redefinition of sub in Section 3.1 can be handled by removing interface PerfectBank in class BANK2 and checking/adjusting any usage of **new** BANK2 (as PerfectBank) in other classes.

6 Related Work

Formal notions of refinement have been used to reflect software development. A refinement is in general leading from a design with certain properties to a design which preserves these properties while adding more detail. In this way refinement is semantics-preserving [27]. Certain refinement logics support the addition of error values, thereby semantics is preserved as long as no errors appear. Banach et al. have argued for the need of refinement-like steps that go beyond the limitation of semantics-preserving development [2]. But their approach does not support analysis of program properties. Hu and Smith [12,13] consider verification of evolving Z specifications. However, they do not look at changes to classes that may affect global system properties.

In the setting of object-oriented programs with inheritance, behavioral subtyping is the most common reasoning approach, restricting subclasses to obey the super-class specifications [19]. This means that subclasses must preserve behavior. Lazy behavioral subtyping [8,9] relaxes this condition; only behavior that is needed to verify local calls in a superclass must be respected by a subclass redefining the method. This gives added flexibility, allowing a larger class of changes without breaking the requirements.

Interface abstraction allows reasoning about remote calls to rely on the declared interface of the callee. This means that changes in a (super)class implementation may be done as long as the stated interface support is respected, and as long as subclass reasoning is not affected. A calculus allowing changes to methods, (super)classes and interfaces is presented in [11], based on lazy behavioral subtyping. Program properties, represented by Hoare triples, are classified in two categories for each class C, representing the verified ones and the unresolved (unverified) ones, $\mathcal{U}(C)$. The set of verified properties of a given class C and method m is denoted $\mathcal{G}(C,m)$. When the set of unverified program properties is verified (i.e., $\mathcal{U}(C)$ is empty) the class is found to be correct in the sense that all pre/post method specifications are satisfied by the corresponding implementation in a class as well as those in interfaces supported by the class. Changes in code or specifications may affect both categories. However, a program requirement added to $\mathcal{U}(C)$ may be impossible to verify (in case the Hoare triple is not satisfied), and it will then remain in $\mathcal{U}(C)$, and there is no guarantee that this problem is detected.

The approach in [10] addresses transformation of classes and allow classes in the middle of a class hierarchy to be changed. Modifications are archived by means of update operations *modify* and *simplify*. The modify operations extend class definitions, allowing code such as new fields, method definitions, guarantees, and interfaces to be added to classes, and existing methods to be redefined. The simplify operation allows redundant methods to be removed from class definitions. The approach does not classify

classes using $\mathcal{G}$ and $\mathcal{U}$ such as in [11], rather, for each update applied to a class, all verification work is done to methods affected by the update. However, any superclass requirements needed to handle local calls are imposed on subclasses, as in [11].

A number of works on asynchronously communicating concurrent objects, partly by the authors of this paper, consider certain forms of software and/or specification changes: The concept of dynamic software updates allows changes to (super)classes during run-time [16]. A challenge with run-time upgrades of distributed systems is the need to allow updates in a distributed manner, and thereby allowing coexistence of different versions of the software [1,16,25]. In contrast to these works, we are focusing on the reasoning aspects. Bannwart and Müller [3] consider program changes through refactoring, and show how to preserve external behavior for a class of non-trivial refactoring. However, they do not include changes that violate behaviors.

Another line of work considers proof reuse, including partial reuse of proofs of earlier verified properties. This may require some storage of proof outlines or non-trivial verification steps. This means that when a module is corrected, one may try to rerun previous proofs to alleviate the verification burden [24]. The notion of abstract method calls allows reuse of abstract proof outlines, for a fixed method body, while their instances may need further work when other methods or requirements are changed [4,14]. A related approach is the use of symbolic predicates to express requirements to general properties for a given program without knowing what the concrete properties are [6]. These approaches simplify the verification task of evolving programs. The amount of proof reuse can be balanced against the amount of automation. Efficiently automated proofs need not be reused while interactive proofs could benefit from reuse, if possible. Our approach is oriented towards a language with a high degree of automation of verification conditions, and proof reuse is therefore not our focus. A recent work by Ulewicz et al. [26] supports a tight integration of verification of unchanged behavior (regression verification) with that of changed behavior (delta verification); but unrestricted changes are not supported.

We build on results from [21] concerning (single) class inheritance. In contrast to that work, we consider here program changes and evolution, supporting *Modification Independence* (Theorem 1), and give a reasoning rule for partially reverified systems. In addition we provide here more fine-grained control of reused code, and a simplified treatment of (static and late-bound) local calls. Furthermore, while [21] assumes single class inheritance, we here extend the approach to *multiple inheritance*. This greatly improves flexibility since addition of superclasses during evolution allows an outdated inheritance hierarchy to be adjusted with minor class changes. Multiple inheritance has also been considered in [20], but not in the context of program evolution.

The present work is an extension of the framework presented in [22], providing more details and theoretical results supporting *Modification Independence* and *Hierarchy Independence*, and extending that framework and reasoning system to multiple inheritance. This requires the reasoning rules to use a double-indexed proof symbol, without complicating the practical applicability. Multiple inheritance has been criticized due to possible confusion between horizontal and vertical name conflicts. However, our language include qualification of inherited names by a superclass, which provides fine-grained control of used names, solving both horizontal and vertical name conflicts at the

cost of awareness of which superclass inherit the relevant definition. And we insist on a healthy binding strategy, as also argued in [9] for the purpose of program reasoning. This limits undesired vertical name conflicts in the case of late binding. In addition, we allow static binding to allow reuse of code from superclasses in a way not affected by added or changed subclasses.

Moreover healthiness is essential in order to ensure *Hierarchy Independence* (theorem 2) in the presence of multiple inheritance. This ensures that changes in a class will only affect subclasses. Without this property we could no longer claim to have a flexible evolution framework! The particular binding strategy used in our approach is based primarily on depth-first traversal of the superclass hierarchy, and secondarily on following superclasses left to right. This is similar to binding in the Perl language (apart from healthiness). This binding strategy is not the most commonly used, but it is advantageous in the setting of evolution, since adding a superclass at the end of a superclass list (somewhere in a hierarchy) will not affect the binding made in existing code. This simplifies the reverification needed for inherited code. The binding strategy also ensures that calls that have a binding before an added superclass also has a binding after the class change. (If inherited method differ in parameter types and numbers, we would need to index the method name by the parameter types.) These factors are advantageous from a pragmatic point of view.

The considered concurrency model is used by a number of languages supporting active objects, including Creol, ABS, Encore, Rebecca, and ASP/Proactive. The core language used here is avoiding the use of futures, in order to simplify the basic reasoning rules for method calls, as discussed in a recent paper [18].

7 Conclusion

We have introduced a framework for evolution of distributed systems, offering flexibility with respect to both changes of code and specifications. We support reverification of changed classes without requiring changes or reverification of (unchanged) superclasses, captured by *Hierarchy Independence* (Theorem 2). Specification violations in changed classes or affected subclasses can be solved modulo local changes in these classes, captured by *Modification Independence* (Theorem 1). In contrast to earlier work [22], we consider evolution in presence of multiple inheritance. We have argued that multiple inheritance is useful and powerful during evolution, and demonstrated this by an example (given in Fig. 7). In particular, multiple inheritance allows adjustments in the middle of an inheritance hierarchy without removing existing inheritance relationships, simply by adding superclasses as desired. This clearly also makes refactoring easier, by reordering the inheritance relationships. By adopting a healthy binding strategy, we control vertical name conflicts. The semantical value of healthiness is demonstrated by the fact that healthiness is needed for Hierarchy Independence.

We are avoiding inconsistencies that are inherent in frameworks building on behavioral subtyping/lazy behavioral subtyping. Flexibility with respect to reuse and inheritance, beyond the limitations of behavioral subtyping, requires that all objects are seen through an interface (interface abstraction). Our approach builds on the principle of the interface substitution (any object of interface F supports any superinterface of F

as well) and the principle of behavioral interface subtyping, where each class must support its declared interfaces, but need not support interfaces of superclasses. This allows the class hierarchy to be used for code reuse while the interface hierarchy is used for behavioral reuse. In contrast to lazy behavioral subtyping, no superclass requirements are imposed on subclasses by the framework. This gives a more flexible framework for software modifications than those of [10,11] since methods can be redefined without restrictions caused by superclasses. This means that we may deal with software changes that cannot be verified with approaches building on lazy behavioral subtyping. The Bank example demonstrated this.

In our framework, modifications to a class C lead to reverification of that class, and subclasses must be reconsidered when they (directly or indirectly) inherit modified parts of C, but superclasses need not be reverified. Other (i.e., hereditarily unrelated) classes are not affected unless some interfaces are removed from the implementation clause of C, in which case all $v := \texttt{new}\ C$ statements must be reconsidered, ensuring C still supports the interface of o and if not, using another class. During reverification, proofs can be reused as much as possible, and further changes to the class and/or subclasses may be done as needed.

Our framework considers the setting of active, concurrent objects, for which Java code can be generated. We have demonstrated that Hoare-style reasoning is quite simple for this setting, in the sense that reasoning is like sequential reasoning, with sequential effects on the history added. The handling of multiple inheritance implies a double indexing of the proof symbol for Hoare triples, which guides the generation of verification conditions without adding practical complications. Our language supports late-bound remote method calls, as well as static local calls and late-bound local calls. The notion of static local calls is needed in the framework to reduce verification conditions about late-bound local calls to verification conditions about static local calls. Our framework gives fine-grained control of reused code, where the handling of local calls, both late-bound and static ones, as well as suspension, is essential. Static local calls are also useful in programming, avoiding the fragile base class problem since the binding is fixed for such calls.

We have assumed type correct programs. Therefore removal of fields, methods, classes, and interfaces is only allowed when these are superfluous. We have not considered changes in interfaces, other than removal of superfluous interfaces. As mentioned the change of an interface could be simulated by introducing a new version of the interface, and by changing all usage of the old interface, and then removing it.

Our framework may be extended to reason about dynamic (run-time) class upgrades, assuming existing objects are upgraded in invariant states, as in Creol [16], where new calls run renewed code and suspended old calls run old code. The new invariant must imply the old invariant, and it must be verified that old methods maintain the new invariant. This ensures that the interleaving of new code and remaining old code is not harmful. The requirements to an upgraded class are strengthened by these requirements, whereas the requirements to subclasses are as described.

Acknowledgments. This work is supported by the *IoTSec* project, the Norwegian Research Council (No. 248113/O70), and by the *SCOTT* project, the European Leadership Joint Undertaking under EU H2020 (No. 737422).

References

1. Ajmani, S., Liskov, B., Shrira, L.: Modular software upgrades for distributed systems. In: Thomas, D. (ed.) ECOOP'06 – Object-Oriented Programming. pp. 452–476. Springer (2006)
2. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and Theoretical Underpinnings of Retrenchment. Science of Computer Programming **67**(2-3), 301–329 (2007)
3. Bannwart, F., Müller, P.: Changing programs correctly: Refactoring with specifications. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM'06: 14th Intern. Symp. on Formal Methods, Proceedings. LNCS vol. 4085, pp. 492–507. Springer (2006)
4. Bubel, R., Damiani, F., Hähnle, R., Johnsen, E.B., Owe, O., Schaefer, I., Yu, I.C.: Proof repositories for compositional verification of evolving software systems - Managing change when proving software correct. Trans. Foundations for Mastering Change **1**, 130–156 (2016)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude manual (version 2.4) (2008)
6. Din, C.C., Johnsen, E.B., Owe, O., Yu, I.C.: A modular reasoning system using uninterpreted predicates for code reuse. J. Logical and Algebraic Methods in Program. **95**, 82–102 (2018)
7. Din, C.C., Owe, O.: A sound and complete reasoning system for asynchronous communication with shared futures. J. Logical and Algeb. Methods in Program. **83**(5-6), 360–383 (2014).
8. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Lazy behavioral subtyping. Journal of Logic and Algebraic Programming **79**(7), 578–607 (2010)
9. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Incremental reasoning with lazy behavioral subtyping for multiple inheritance. Sc. of Computer Programming **76**(10), 915–941 (2011)
10. Dovland, J., Johnsen, E.B., Owe, O., Yu, I.C.: A Proof System for Adaptable Class Hierarchies. Journal of Logical and Algebraic Methods in Programming **84**(1), 37 – 53 (2015)
11. Dovland, J., Johnsen, E.B., Yu, I.C.: Tracking behavioral constraints during object-oriented software evolution. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. LNCS vol. 7609, pp. 253–268. Springer (2012)
12. Fu, Z., Smith, G.: Towards more flexible development of Z specifications. In: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Eng. pp. 281–288 (June 2008)
13. Fu, Z., Smith, G.: Property transformation under specification change. Frontiers of Computer Science in China **5**(1), 1–13 (Mar 2011)
14. Hähnle, R., Schaefer, I., Bubel, R.: Reuse in Software Verification by Abstract Method Calls. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. LNCS vol. 7898, pp. 300–314. Springer (2013)
15. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software & Systems Modeling **6**(1), 35–58 (2007)
16. Johnsen, E.B., Owe, O., Simplot-Ryl, I.: A dynamic class construct for asynchronous concurrent objects. In: Steffen, M., Zavattaro, G. (eds.) Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05). LNCS vol. 3535, pp. 15–30. Springer (Jun 2005)
17. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science **365**(1–2), 23–66 (2006)
18. Karami, F., Owe, O., Ramezanifarkhani, T.: An evaluation of interaction paradigms for active objects. Journal of Logical and Algebraic Methods in Programming **103**, 154 – 183 (2019)
19. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. on Progr. Lang. and Syst. **16**(6), 1811–1841 (Nov 1994)
20. Owe, O.: Verifiable programming of object-oriented and distributed systems. In: Petre, L., Sekerinski, E. (eds.) From Action System to Distributed Systems: The Refinement Approach, pp. 61–80. Taylor&Francis (2015)

21. Owe, O.: Reasoning about inheritance and unrestricted reuse in object-oriented concurrent systems. In: Ábrahám, E., Huisman, M. (eds.) Proc. 12th International Conference Integrated Formal Methods (IFM'16), Iceland 2016, LNCS vol. 9681, pp. 210–225. Springer (2016)
22. Owe, O., Lin, J.-C., Fazeldehkordi, E.: A flexible framework for program evolution and verification. In: 7th International Conference on Model-Driven Engineering and Software Development (Modelsward'19) (Feb. 2019).
23. Owe, O., Ryl, I.: On combining object orientation, openness and reliability. In: Proc. of the Norwegian Informatics Conference (NIK'99). pp. 187–198. Tapir (Nov. 1999)
24. Reif, W., Stenzel, K.: Reuse of proofs in software verification. In: Shyamasundar, R. (ed.) Foundations of Software Technology and Theoretical Computer Science. LNCS vol. 761, pp. 284–293. Springer (1993)
25. Seifzadeh, H., Abolhassani, H., Moshkenani, M.S.: A survey of dynamic software updating. Journal of Software: Evolution and Process **25**(5), 535–568 (2013)
26. Ulewicz, S., Ulbrich, M., Weigl, A., Kirsten, M., Wiebe, F., Beckert, B., Vogel-Heuser, B.: A verification-supported evolution approach to assist software application engineers in industrial factory automation. In: 2016 IEEE International Symposium on Assembly and Manufacturing (ISAM). pp. 19–25 (Aug 2016).
27. Ward, M.P., Bennett, K.H.: Formal methods to aid the evolution of software. International Journal of Software Engineering and Knowledge Engineering **05**(01), 25–47 (1995)