A Flexible Framework for Program Evolution and Verification

Olaf Owe, Jia-Chun Lin and Elahe Fazeldehkordi
Department of Informatics, University of Oslo, Norway

Keywords: Program Evolution, Program Reasoning, Software Changes, Active Objects, Re-verification, Flexibility.

Abstract: We propose a flexible framework for modeling of distributed systems, supporting evolution by means of unrestricted modifications in such systems, and with support of verification and re-verification. We focus on the setting of concurrent and object-oriented programs, and consider a core high-level modeling language supporting active, concurrent objects. We show that our framework can deal with verification of software changes that are not possible to verify in comparable frameworks. We demonstrate the approach by variations over a simple example.

1 INTRODUCTION

Program development is in general a complicated process where many kinds of mistakes can be made, including bad design decisions, unclear specifications, misunderstandings, and erroneous code or specifications. Problems made early may not be discovered until much later. Redesigning or modifying code made at an early stage in the software development may affect many parts of the overall system. Making changes in order to correct problematic decisions may create new problems that are hard to foresee. These kinds of problems are severe in the setting of concurrent programs when the interaction of the different concurrent components is complicated, and also in the setting of object-oriented programs, due to inheritance, late binding and code reuse.

We consider the setting of distributed, concurrent, and object-oriented systems, and suggest 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, design, executable modeling, analysis, and maintenance. The framework allows unrestricted changes in code and requirements, and includes a theory for reverification of a changed system. We include 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. 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* (Dovland et al., 2010;

Owe, O., Lin, J. and Fazeldehkordi, E.
A Flexible Framework for Program Evolution and Verification.
DOI: 10.5220/0007690301770189
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 177-189
ISBN: 978-989-758-358-2

Dovland et al., 2011); 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 (Dovland et al., 2012).

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. If a class is changed, it is undesirable that its superclasses also need to be modified, as this can destroy flexibility. This is the case in approaches where requirements are pushed from superclasses to subclasses, as in the case of behavioral subtyping. Therefore the semantics of class inheritance matters, in particular when it comes to inheritance of requirements.

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* (Owe, 2015) 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 – which opens up for more liberal modifications than work based on lazy behavioral subtyping (Dovland et al., 2010; Dovland et al., 2011), 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, which is needed to avoid inconsistent specifications due to inheritance. Thus in our approach we can avoid inconsistencies due to superclass requirements, simply by controlling which requirements to inherit.

We allow 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.

As mentioned, we show that our framework can deal with software changes that are not possible to verify in comparable frameworks. We also show how to reason within a hierarchy where some classes are verified and others not. We demonstrate our framework by examples.

2 LANGUAGE SETTING

We consider the setting of asynchronously communicating objects, so-called *active objects*, supporting blocking and non-blocking remote calls, but not remote field access. In this setting, verification of a system of concurrent objects can be done compositionally, verifying each class separately, letting each class and interface refer to its local history, reflecting the time sequence of communication events such as method calls and returns. Each class can be verified in a sequential manner, and a compositional rule states that a global invariant referring to the global history can be obtained by conjunction of the local invariants (on local histories) together with a wellformedness predicate relating the local histories.

For the purpose of this paper, we consider a core high-level imperative modeling language, given in Figure 1, inspired by the concurrency model of Creol (Johnsen and Owe, 2007). The language is executable with an interpreter in Rewriting logic/Maude (Clavel et al., 2008). A program consists of interfaces and classes. A class may implement a number of interfaces. 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 and invariants.

A class can inherit from a superclass while removing/adding/redefining method definitions, method specifications and invariants. And fields w may be added (an initial value r may be given, otherwise the default value of the type is used). For simplicity, we assume read-only access to method and class parameters, and limit the discussion to single inheritance. 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.

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** $C(\overline{e}) \mid e.m(\overline{e}) \mid C:m(\overline{e})$	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	releasing statements
		$\mid$ **if** e **then** s [**else** $s]^?$ **fi**	if statement
P	::=	$[$ **[** A,A **]** $]^+\ [$**where** $A^+]^?$	pre-/postconditions
I	::=	**inv** $A^+\ [$**where** $A^+]^?$	invariant specification

Figure 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.

An object variable must be typed by an interface, not by a class. A variable declared of interface F is called an F variable. Though 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* (Owe and Ryl, 1999; Johnsen and Owe, 2007; Johnsen et al., 2006). 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 v. For simplicity, we assume type correctness, and assume that a class does not offer two declarations of the same method name.

Dot-notation is reserved for late bound method calls, i.e., for a C object o (an object o of run-time class C) a remote call $o.m(...)$ binds to the definition of method m in C, if any, or the closest superclass with a definition of m. This definition of m is denoted $C:m$, and is referred to as m of C. The notation $C:m$ may also be used in program code, resulting in static binding. Thus $C:m$ binds to the method m of class C if defined in C or the closest superclass. We assume here that the static type checking will replace $C:m$ by $B:m$ if the call binds to m of B.

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). Local calls are possible with the syntax $v :=$ this$.m(\overline{e})$, for late bound calls, or $v := C:m(\overline{e})$ for static calls. (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 not be used in non-blocking calls (since they may be called in states violating the invariant).

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.

We allow "negative" inheritance by the syntax **removing** $m_1, m_2 \ldots$, expressing that the listed methods should not be inherited. 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.)

Apart from standard statements, we include both static and dynamic calls, and we include cooperative scheduling (with the await mechanism) allowing non-blocking calls, something which is useful in a distributed concurrent setting.

2.1 History-based Specification

Histories are used to capture the time sequence of communication events. *Global histories* capture all communication events in a distributed system, and *local histories* capture all communication events seen from a given component. The local history $\mathbf{h}$ of a component 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 a class C may refer to fields, the local history $\mathbf{h}$, class parameters, and this. An 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 post-condition 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 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 local history $\mathbf{h}$ of a class/interface is the time sequence of communications events seen by this object. We define the events below:

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

where o represents the other part in the communication. In the last event, F is the declared interface of the variable receiving the new object, i.e., the interface of v in the statement $v := \mathbf{new}\ C(\overline{e})$. In practice, specifications using histories will often be concerned about method completions, i.e., $\leftarrow$ and $\twoheadleftarrow$ events, since these capture input/output relations. For a given method call, the $\leftarrow$ event precedes the $\twoheadleftarrow$ event, which is formalized by a wellformedness predicate.

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:

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

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.

For a local history $\mathbf{h}$, we let $\mathbf{h}/F$ denote the projection to events visible through F, i.e., events of the form this $\rightarrow o.\,\mathbf{new}\,C(\overline{e})$, this $\rightarrow o.m(\overline{e})$, and this $\twoheadleftarrow o.m(\overline{e};e)$, as well as events of the form $o \twoheadrightarrow$ this$.m(\overline{e})$ and $o \leftarrow$ this$.m(\overline{e};e)$ for m offered by F. The same notation applies to classes C, thus $\twoheadrightarrow$ this$.m$ and $\leftarrow$ this$.m$ events are restricted to methods 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.

2.2 An Example with Reasoning

Figure 2 shows a minimalistic example defining a class BANK and a subclass BANKPLUS and related interfaces. 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 balance is always non-negative.

The code illustrates suspension, non-blocking and blocking calls, static and late bound calls. Interface and type names are capitalized while class names are written in upper case letters. The specification of interface Bank illustrates history-based specification. In the inductive definition of sum, ***others*** is used to match other cases, and underscore (_) is used to match any expression. The keyword **inv** identifies invariants and **where** identifies auxiliary function definitions. In assertions, **inv** refers to the current invariant, while C : **inv** refers to the invariant of class C. The auxiliary function *sum* calculates the balance from the local history. Note that only method-return events are used in the specification.

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. 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 therefore 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.

```
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;
  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{Seq[String] paid;
    Bank acc:= new BANK;
  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)
  where
  allpaid(empty) = empty,
  allpaid(h;_←this.bill(k,x,y;true))=(allpaid(h);k),
  allpaid(h; others)=allpaid(h) }
```

Figure 2: A simple bank example including a possible client class, where paid corresponds to successful bill payments. The call BANK:upd uses static binding.

```
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] }
```

Figure 3: A subclass of class BANKPLUS.

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.

2.3 Dynamic and Static Binding

Dynamic (late) binding implies that a method call to m may behave differently depending on the class of the object executing the method (called the *actual class*), even for calls that bind to the same definition, say the one found in a class C. 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 $\mathsf{BANK}:sub$, but the $this.upd$ call in the body of $\mathsf{BANK}:sub$ binds to $\mathsf{BANKPLUS}:upd$ or $\mathsf{BANK}:upd$ depending on the class of the executing object, BANKPLUS or BANK, respectively.

We let the function $bind(C,m)$ return C if C has a definition of m, otherwise the closest superclass of C with a definition of m. When C is known, $bind(C,m)$ can be calculated, even with an open-ended class hierarchy. For a static call $D:m(\ldots)$, the binding is known statically (given by $bind(D,m)$), and we assume that such a call is replaced by $bind(D,m):m(\ldots)$ during type checking. This makes the binding explicit. This means that $bind(C,D:m)$ equals $bind(D,m)$ when C is a subclass of D. For a late bound local call $this.m(\ldots)$ appearing in a class C, the binding of m can be calculated statically for a given actual class of *this*.

In the approach of verification based on behavioral interface subtyping, we reconsider the call for 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 this.m to $bind(C',m)$ and $C:m$ to $bind(C,m)$.

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 ($\vdash$) with the class of the executing object (as in $\vdash_D$), which is possible with behavioral interface subtyping since all relevant proof obligations are reconsidered for each subclass. 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), 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 control in the reasoning, which is valuable in the setting of 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 can not in general be guaranteed in a subclass D if $C:m$ has local calls or release points.

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

In order to do Hoare-style reasoning, we use a system for deriving theorems of the form

$$\vdash_C [P]\, s\, [Q]$$

where C is the actual class, 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. Figure 4 presents sample proof rules needed for the example. 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 $\vdash_C$ to prove the pre/post verification conditions for objects of that class. For code in class C this corresponds to normal class-based reasoning. For code inherited by C, reasoning about release points and local or self calls depends on C, which reflects the actual class of this object. According to rule self call, the call $v := \mathsf{this}.m(\bar{x})$ is equivalent to the local call $v := C : m(\bar{x})$ where C is the class of this object. Rule local call states that reasoning about $v := B : m(\bar{e})$ reduces to reasoning about $body_{B:m}$, adding effects on the history, and where $body_{B:m}$ denotes the body of the definition of method m in class B. The *body* of a method definition $m(\bar{x})\{s;\ \mathbf{return}\ e\}$ is given by

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

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 (Din and Owe, 2014). 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 $\vdash_C [P]\, s\, [Q]$ from $\vdash_B [P]\, s\, [Q]$ when s has no release points and no local calls leading to calls of methods redefined below B. Otherwise $\vdash_C [P]\, s\, [Q]$ can be established by a new analysis of s and of any locally called methods in s. In particular $\vdash_C [P]\, v := B : m(\bar{x})\, [Q]$ follows from $\vdash_B [P]\, v := B : m(\bar{x})\, [Q]$ when $B : m$ has no release points or local calls. In contrast to behavioral subtyping and lazy behavioral subtyping, no requirements are imposed on subclasses.

3 PROOF OBLIGATIONS

For a given program we must ensure that each class C satisfies the stated requirements of C, i.e., that the **`implements`** clauses are 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 a (run-time) class C will satisfy the interfaces of C. This ensures that the compositional rule for reasoning about concurrent object systems is sound. Furthermore, each late bound local call made at run-time 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, class, interface claim, class invariant, and method specification. We consider the following proof obligations, where each obligation identifies the relevant class C:

Program and Class Correctness
A program P is *correct*, denoted $\vdash P\ \mathbf{ok}$, if each class in the program is correct.
A class C is *correct*, denoted $\vdash C\ \mathbf{ok}$, iff

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

where $\vdash_C m(\bar{x})\ \mathbf{sat}\ [P,Q]$ is verified by proving

$$\vdash_C [P]\, body_{bind(C,m):m}\, [Q]$$

(as above), and $\vdash C\ \mathbf{inv}\ I$ is verified by proving

$$\vdash_C m(\bar{x})\ \mathbf{sat}\ [I,I]$$

for each public method $m(\bar{x})$ (possibly inherited) in C and that the invariant holds initially, i.e., I holds when $\mathbf{h}$ is replaced by the empty history and fields by initial values; and finally, $\vdash C\ \mathbf{sat}\ F$ is verified by

assign $$\vdash_C [Q_e^v]\, v := e\, [Q]$$

history $$\vdash_C [h' = \mathbf{h}]\, s\, [h' \leq \mathbf{h}]$$

await guard $$\vdash_C [I_C \wedge L]\ \texttt{await}\ b\, [b \wedge I_C \wedge L]$$

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

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

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

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

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

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

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

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

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

Figure 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 local call, we assume that v does not occur in e. In rule local call we assume that $\overline{x}$ is the formal parameter list (which is read only). Note that $\mathit{bind}(C,m)$ is calculated at verification time.

proving that the conjunction of the invariants $I_i(\mathbf{h})$ of C implies the invariant of F, I_F (considering methods visible through F, as explained in Section 2.1):

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

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 type checking ensures that removed methods are not directly or indirectly called by methods (possibly inherited) in C, and that private methods of C are not directly or indirectly called with **await**.

For a subclass C' of C, $\vdash_C m(\overline{x})\ \texttt{sat}\ [P,Q]$ need not imply $\vdash_{C'} m(\overline{x})\ \texttt{sat}\ [P,Q]$ 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., $\mathit{bind}(C,n)$ versus $\mathit{bind}(C',n)$, respectively). In general $\vdash_C m(\overline{x})\ \texttt{sat}\ [P,Q]$ 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 $\vdash_C m(\overline{x})\ \texttt{sat}\ [P,Q]$ implies $\vdash_{C'} m(\overline{x})\ \texttt{sat}\ [P,Q]$ if $\vdash_C m(\overline{x})\ \texttt{sat}\ [P,Q]$ does not depend on any redefined code and that any invariant used in the verification is respected by C'.

For a method $m(\overline{x})$ inherited in C from a superclass B, we may write

$$\vdash_C B : m(\overline{x})\ \texttt{sat}\ [P,Q]$$

rather than $\vdash_C m(\overline{x})\ \texttt{sat}\ [P,Q]$ to visualize that the method comes from the superclass B. Since

$bind(C, B : m)$ equals $bind(B, m)$, this is consistent with the definition of $\vdash_C m(\bar{x})$ **sat** $[P, Q]$. For a method $C : m$ without local calls or suspension points, we have that $\vdash_{C'} C : m(\bar{x})$ **sat** $[P, Q]$ reduces to $\vdash_C m(\bar{x})$ **sat** $[P, Q]$. This gives a practical way of reusing proofs from superclasses.

4 SOFTWARE CHANGES

We now consider changing a system, i.e., changing existing classes and adding new classes and interfaces. For instance, one may introduce a new interface to make two independent subsystems interact, adding support of the new interface in one or more existing classes. An existing class D may be changed by replacing methods, changing inheritance clauses, implementation 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/reuse code from D. Thus, we need to reverify the redefined D, but in addition we need to consider the affected subclasses of D.

We use the following syntax for defining class updates: **update** *Cl*, where the meta-symbol *Cl* defines the syntax for class definitions, as in Figure 1. Thus, a class update

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

modifies an existing class D by adding class parameters cp^+ (if present), changing the interface support to F^+ (if present), adding superclasses $[C(\bar{e})]^+$ (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.

We consider correctness of the updated code, and avoid complications such as run-time upgrades where new and old versions of the updated code are part of the running system. As before we assume type correctness. Consider that the redefined 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 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 $\vdash_D m(v)$ **sat** $[P, Q]$. For each such D result, it suffices to prove $\vdash_{\hat{D}} m(v)$ **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 caused by superclass modifications can be handled, 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 may 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 then 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 while doing modifications, at the cost of reconsidering subclasses in case the modifications require changes in subclasses.

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:

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

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 welldefinedness predicate, expressing the **before** ordering between events, given in Section 2.1.

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.

Limitations: We assume type correctness. Thus program changes must result in type correct programs. Removal of declarations of fields, methods, parameters, and variables is 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.

Examples of Software Changes on BANK

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

```
update class BANK implements PerfectBank{
  Bool check(Int x){return true} [true, return]
  Bool upd(Int x){Bool ok:=check(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 check 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 check 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 check(x) as in

```
update class BANKPLUS
      { Bool check(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 (Dovland et al., 2015; Dovland et al., 2012).

Adding a side effect in check 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 spec. 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 2.2 can be handled by removing interface PerfectBank in class BANK2 and checking/adjusting any usage of **new** BANK2 (as PerfectBank) in other classes.

5 RELATED WORK

In formal methods, the notion of refinement is 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 (Ward and Bennett, 1995). 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 (Banach et al., 2007). But their approach does not support analysis of program properties. Hu and Smith (Fu and Smith, 2008; Fu and Smith, 2011) 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 (Liskov and Wing, 1994). This means that subclasses must preserve behavior. Lazy behavioral subtyping (Dovland et al., 2010; Dovland et al., 2011) 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 (Dovland et al., 2012), 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 (Dovland et al., 2015) 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 (Dovland et al., 2012), rather, for each update applied to a class, all verification work is done to methods affected by the update. But, any superclass requirements needed to handle local calls are imposed on subclasses, as in (Dovland et al., 2012).

A number of works on asynchronously communicating concurrent objects consider certain forms of software and/or specification changes: The concept of dynamic software updates allows changes to (super)classes during run-time (Johnsen et al., 2005). 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 (Ajmani et al., 2006; Johnsen et al., 2005; Seifzadeh et al., 2013). In contrast to these works, we are focusing on the reasoning aspects. Bannwart and Müller (Bannwart and Müller, 2006) 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 (Reif and Stenzel, 1993). 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 (Bubel et al., 2016; Hähnle et al., 2013). 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 (Din et al., 2018). 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. (Ulewicz et al., 2016) supports a tight integration of verification of unchanged behavior (regression verification) with that of changed behavior (delta verification); but unrestricted changes are not supported.

The considered concurrency model is used by a number of languages supporting active objects, including ABS and Encore. 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 (Karami et al., 2019).

6 CONCLUSION

We have shown a flexible framework for reasoning about program reuse and evolution, considering an imperative setting of distributed object-oriented components. 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 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 that of (Dovland et al., 2015; Dovland et al., 2012) 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 example demonstrated this. We are avoiding inconsistencies that are inherent in frameworks building on behavioral subtyping/lazy behavioral subtyping.

In our framework, modifications to a class C lead to reverification of that class, and subclasses must be reconsidered when they inherit modified parts of C, but superclasses need not be reverified. 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.

Our language supports late bound method calls. However, for local method calls the language supports static local calls as well as 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 build on results from (Owe, 2016) concerning 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.

Our framework can be generalized to multiple inheritance, which is supported by behavioral interface subtyping. And the framework may be extended to reason about dynamic (run-time) class upgrades, assuming existing objects are upgraded in invariant states, as in Creol (Johnsen et al., 2005), 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 above.

REFERENCES

Ajmani, S., Liskov, B., and Shrira, L. (2006). Modular software upgrades for distributed systems. In Thomas, D., editor, *ECOOP 2006 – Object-Oriented Programming*, pages 452–476. Springer.

Banach, R., Poppleton, M., Jeske, C., and Stepney, S. (2007). Engineering and Theoretical Underpinnings of Retrenchment. *Science of Computer Programming*, 67(2-3):301 – 329.

Bannwart, F. and Müller, P. (2006). Changing programs correctly: Refactoring with specifications. In Misra, J., Nipkow, T., and Sekerinski, E., editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 492–507. Springer.

Bubel, R., Damiani, F., Hähnle, R., Johnsen, E. B., Owe, O., Schaefer, I., and Yu, I. C. (2016). Proof repositories for compositional verification of evolving software systems - managing change when proving software correct. *Transactions on Foundations for Mastering Change*, 1:130–156.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-oliet, N., Meseguer, J., and Talcott, C. (2008). Maude manual (version 2.4).

Din, C. C., Johnsen, E. B., Owe, O., and Yu, I. C. (2018). A modular reasoning system using uninterpreted predicates for code reuse. *Journal of Logical and Algebraic Methods in Programming*, 95:82–102.

Din, C. C. and Owe, O. (2014). A sound and complete reasoning system for asynchronous communication with shared futures. *Journal of Logical and Algebraic Methods in Programming*, 83(5-6):360–383.

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

Dovland, J., Johnsen, E. B., Owe, O., and Steffen, M. (2011). Incremental Reasoning with Lazy Behavioral Subtyping for Multiple Inheritance. *Science of Compututer Programming*, 76(10):915–941.

Dovland, J., Johnsen, E. B., Owe, O., and Yu, I. C. (2015). A Proof System for Adaptable Class Hierarchies. *Journal of Logical and Algebraic Methods in Programming*, 84(1):37 – 53.

Dovland, J., Johnsen, E. B., and Yu, I. C. (2012). Tracking behavioral constraints during object-oriented software evolution. In Margaria, T. and Steffen, B., editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 253–268. Springer.

Fu, Z. and Smith, G. (2008). Towards more flexible development of z specifications. In *2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 281–288.

Fu, Z. and Smith, G. (2011). Property transformation under specification change. *Frontiers of Computer Science in China*, 5(1):1–13.

Hähnle, R., Schaefer, I., and Bubel, R. (2013). Reuse in Software Verification by Abstract Method Calls. In Bonacina, M. P., editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 300–314. Springer.

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

Johnsen, E. B., Owe, O., and Simplot-Ryl, I. (2005). A dynamic class construct for asynchronous concurrent objects. In Steffen, M. and Zavattaro, G., editors, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 15–30. Springer.

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

Karami, F., Owe, O., and Ramezanifarkhani, T. (2019). An evaluation of interaction paradigms for active objects. *Journal of Logical and Algebraic Methods in Programming*, 103:154 – 183.

Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. *ACMTransactions on Programming Languages and Systems*, 16(6):1811–1841.

Owe, O. (2015). Verifiable programming of object-oriented and distributed systems. In Petre, L. and Sekerinski, E., editors, *From Action System to Distributed Systems: The Refinement Approach*, pages 61–80. Taylor&Francis.

Owe, O. (2016). Reasoning about inheritance and unrestricted reuse in object-oriented concurrent systems. In Ábrahám, E. and Huisman, M., editors, *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 210–225. Springer.

Owe, O. and Ryl, I. (1999). On combining object orientation, openness and reliability. In *Proc. of the Norwegian Informatics Conference (NIK'99)*, pages 187–198. Tapir Academic Publisher.

Reif, W. and Stenzel, K. (1993). Reuse of proofs in software verification. In Shyamasundar, R., editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 284–293. Springer-Verlag.

Seifzadeh, H., Abolhassani, H., and Moshkenani, M. S. (2013). A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568.

Ulewicz, S., Ulbrich, M., Weigl, A., Kirsten, M., Wiebe, F., Beckert, B., and Vogel-Heuser, B. (2016). A verification-supported evolution approach to assist software application engineers in industrial factory automation. In *2016 IEEE Internat. Symposium on Assembly and Manufacturing (ISAM)*, pages 19–25.

Ward, M. P. and Bennett, K. H. (1995). Formal methods to aid the evolution of software. *International Journal of Software Engineering and Knowledge Engineering*, 05(01):25–47.