Modular Reasoning in the Presence of Subclassing

Raymie Stata John V. Guttag

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
raymie@larch.lcs.mit.edu

Abstract

Considerable progress has been made in understanding how to use subtyping in a way that facilitates modular reasoning. However, using subclassing in a way that facilitates modular reasoning is not well understood. Often methods must be overriden as a group because of dependencies on instance variables, and the programmers of subclasses cannot tell which methods are grouped without looking at the code of superclasses. Also, the programmers of subclasses must look at the code of superclasses to tell what assumptions inherited methods make about the behavior of overriden methods.

We present a systematic way to use subclassing that facilitates formal and informal modular reasoning. Separate specifications are given to programmers writing code that manipulates instances of a class and to programmers writing subclasses of the class. The specifications given to programmers of subclasses are divided, by *division of labor specifications*, into multiple parts. Subclasses may inherit or override entire parts, but not sub-parts. Reasoning about the implementation of each part is done independently of other parts.

1 Introduction

Subclassing fosters code reuse by allowing old classes to be specialized into new ones. However, programming with subclasses is not without its difficulties. Two important problems are:

- *Grouping dependencies.* Two or more methods often share responsibility for maintaining a set of instance variables. If a subclass overrides one member of such a group, it must override all members. Programmers of subclasses need to know the grouping dependencies of superclasses.

- *Behavior dependencies.* When subclasses override some methods and inherit others, dependencies exist among the behaviors of subclass and superclass methods. Inherited methods make assumptions about the behavior of overriden methods, assumptions programmers of subclasses need to meet.

Programmers of subclasses often solve these problems by guessing or by looking at the source code of superclasses. These solutions are prone to errors. Also, they tend to incorporate undocumented assumptions about superclasses into subclasses, making it difficult to know what changes to a superclass will affect subclasses.

This paper presents an alternative solution based on a new kind of specification for specialization interfaces. The specialization interface of a class is the interface used by programmers who write subclasses of the class. This can be quite different from the client interface, the interface used by programmers who write code that manipulates instances of the class. Our specifications partition the specialization interface into multiple parts. Subclasses override entire parts rather than individual methods. Reasoning about the

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014–92–J–1795, and in part by the National Science Foundation under grant CCR–9115797.

OOPSLA '95 Austin, TX, USA
© 1995 ACM 0-89791-703-0/95/0010...$3.50

implementation of each part is done without considering the implementations of other parts; this allows programmers to build and change classes without looking at the source code of superclasses or of subclasses.

Sec. 2 defines our language assumptions and terminology. Sec. 3 illustrates grouping and behavior dependencies using an example; we refer to this example throughout the paper. Sec. 4 presents our specifications for the specialization interface. Sec. 5 describes how to verify the specialization interface; Sec. 6 describes how verify the client interface. Sec. 7 discusses some related work, and Sec. 8 gives some concluding remarks, focusing on how our results can be applied informally to existing projects.

2 Language model

We assume a model of object-oriented programming based on objects, object types, and classes. Our model is a fairly standard, single-inheritance model, except we separate object types from classes.

An object is state along with methods that manipulate the state. As is usual, an object is implemented as a set of instance variables and code for its methods.

An object type describes the behavior of objects. Most languages support only signature specifications of object types that define the names and signatures of a type's methods. We assume behavioral specifications of object types that define the effects of a type's methods on the program's state. We use the behavioral specifications describe in [Liskov94]. Such specifications consist of an abstract description of the value space and method behavior of objects subsumed by the type:

$$\textit{object-type} \quad \rightarrow \quad \Sigma\ I\ C\ M^*$$

Σ, I, and C together described the value space of the type. The sort Σ is the underlying set of values that objects subsumed by the type can take on; the invariant I is a static constraint on Σ that objects subsumed by the type must satisfy; the constraint C is a dynamic constraint that describes how the values of objects subsumed by the type evolve over time. For example, the sort of a counter type might be the set of integers, the invariant might be that counters are never negative, and the constraint might be that counters always increase in value. The set of method specifications M^* define the behavior of each method in terms of pre- and post-conditions.

A class is a template that defines a set of instance variables and methods. Instantiating a class creates a new object with the instance variables and method code defined by the class. We assume a standard single inheritance model. A subclass inherits instance variables and methods definitions from a superclass. Instance variables are encapsulated: methods of a subclass cannot access instance variables of its superclass. A class overrides a method by defining locally a method with the same name as one of its superclasses' methods. When a class overrides a method, the signature of the new method must conform to the signature of the old one. As in C++, a protected method of a class is a method that can be called and overriden by the classes' subclasses but cannot be called by clients of its instances.

The client interface of a class consists of the non-protected methods defined by the class. Client interfaces are specified using object types. The specialization interface of a class consists of all the methods defined by the class, including protected methods. We specify specialization interfaces with a new kind of specification described in Sec. 4.

3 An example

Fig. 1 contains the class `IntSetClass`. This class is designed to both implement integer sets and to be a superclass for other unordered collection classes. The code for **`member`** is contrived to illustrate a number of problems in a single page of code, but the overall pattern of calling methods on **self** that are overriden by subclasses is a fundamental aspect of subclassing.

Subclasses of `IntSetClass` can inherit a working **`member`** operation when they override **`add`**, **`remove`**, and **`size`**. `IntMultiSetClass` (Fig. 2), a class implementing multisets, is one such class.

Fig. 1 includes a specification for `IntSetClass`.

```
class IntSetClass
% state [ s:IntSet, c:IntSet ]
%   s holds the contents of the set
%   c caches membership hits
% invariant self.c ⊆ self.s

els:Array[Int] % used by add, remove, and size
valid:Bool % used by member and invalidate
cache:Int  % used by member and invalidate

method add(el:Int)
  % Modifies self.s
  % Ensures self_post.s = self_pre.s ∪ {el}
  index:Int := self.els.find(el)
    except when not_found: self.els.addh(el)
    end
  end add

method remove(el:Int)
  % Modifies self
  % Ensures self_post.s = self_pre.s − {el}
  index:Int := self.els.find(el)
    except when not_found: return end
  self.invalidate() % Empty cache
  self.els[index] := self.els.top()
  self.els.remh()
  end remove

method size() returns(Int)
  % Ensures result = |self_pre.s|
  return(self.els.size())
  end size

method member(el:Int) returns(Bool)
  % Modifies self.c
  % Ensures result = el ∈ self_pre.s
  if ~self.valid or self.cache ~= el then
    beforeSize:Int := self.size()
    self.remove(el)
    if self.size() = beforeSize then
      return(false)
      end
% Line 1:
    self.add(el)
    self.valid := true
    self.cache := el
    end
  return(true)
  end member

protected method invalidate()
  % Modifies self.c
  % Ensures self_post.c = {}
  self.valid := false
  end invalidate
end IntSetClass
```

Figure 1: IntSetClass class.

Although more formal than usual, this specification is typical of current practice in attempting to specify both the client and specialization interfaces in a single specification. For the client interface, the specification is expressed in terms of sets rather than in terms of something better suited to potential subclasses. For the specialization interface, the specification exposes the cache used by **member** to cache successful membership tests, even though the cache is irrelevant to clients.

3.1 Grouping dependencies

Multiple methods often share responsibility for maintaining a set of instance variables. Members of such groups must be overriden together ([Lamping93, Taligent94]). The methods **add**, **remove** and **size** together maintain the instance variable **els** used to represent the set. If a subclass of **IntSetClass** were to use a tree representation for sets, it would have to override all three methods. Similarly, **member** and **invalidate** together maintain **valid** and **cache** and must also be overriden together.

Method grouping is important to programmers of superclasses as well as of subclasses. For example, in the code for **member**, on the line marked (1), the choice to call the method **add** rather than directly manipulating the instance variable **els** is driven by the method grouping of the class. This choice ensures that subclasses will be able to inherit **member** when they override **add**, **remove**, and **size**. The same is true for the choice to call **invalidate** in **remove** rather than clearing the **valid** bit directly.

3.2 Behavior dependencies

Subclasses such as **IntMultiSetClass** override some methods and inherit others. Programmers of subclasses need to understand what assumptions inherited methods make about overriden methods. Consider, for example, the method **size**, which is called by **member**. The version of **size** given in Fig. 2 happens to be what is required by **member**, while an equally plausible version that implements the specification:

```
method size() returns(Int)
  % Ensures result = |toSet(self_pre.m)|
class IntMultiSetClass
% state [ m:IntMultiSet, c:IntSet ]
% invariant self.c ⊆ toSet(self.m)

superclass IntSetClass
els:Table[Int,Int]

method add(el:Int)
  % Modifies self.m
  % Ensures self_post.m = addOnce(self_pre.m, el)
  self.els[el] := self.els[el] + 1
    except when not_found:
      self.els[el] := 1
      end
    end add

method remove(el:Int)
  % Modifies self
  % Ensures self_post.m = removeOnce(self_pre.m, el)
  count:Int := self.els[el]
    except when not_found: return end
  if count > 0 then
    self.els[el] := count - 1
    self.invalidate()
    end
  end remove

method size() returns(Int)
  % Ensures result = Σ_x count(self_pre.m, x)
  result:Int := 0
  for key:Int in self.els.keys() do
    result := result + self.els[key]
    end
  return(result)
  end size

% method member(el:Int) returns(Bool)
%   Modifies self.c
%   Ensures result = el ∈ toSet(self_pre.m)
%   Code inherited from IntSetClass

% protected method invalidate()
%   Modifies self.c
%   Ensures self_post.c = {}
%   Code inherited from IntSetClass
end IntMultiSetClass
```

Figure 2: IntMultiSetClass class.

would cause member to function incorrectly.

Inheriting superclass specifications into subclasses is not a solution. As illustrated by IntMultiSet, the subclass is often specified in terms of a different value space than the superclass, so the superclasses' specification does not make sense in the subclasses' context. Even when the value spaces are the same, inheriting specifications is too restrictive: subclasses must have the freedom to behave differently from their superclasses. Instead, we use the specification of the superclass as a constraint on the specification of the subclass without requiring that the two specifications be identical. This constraint is described in Sec. 5.3.

4 Specialization specifications

In our model, object types give both a signature and a behavioral specification of objects. Object types are a good way to specify client interfaces, but they are not sufficient for specialization interfaces. We specify specialization interfaces with an object type together with a division of labor specification:

$$\begin{aligned} \textit{special-spec} &\rightarrow \textit{object-type labor-div} \\ \textit{labor-div} &\rightarrow \{[\textit{substate method-name}^*]^*\} \end{aligned}$$

We call this combination a *specialization specification.*

A division of labor partitions the state and methods of a class into groups. These groups form abstraction barriers within classes. The value spaces associated with each method group are called *substates.* The methods of a method group are responsible for maintaining the substate of the group. Only the methods of a group directly manipulate the representation of the group's substate; other methods manipulate the substate indirectly by calling methods in the substate's group. In a subclass, if any method of a method group is overriden, then all methods in the group must be overriden, and the new code becomes responsible for implementing the group's substate.

A specialization specification for IntSetClass is given in Fig. 3 (the **existential** clause will be explained later). We describe the sorts of object types as tuples with named fields. Each field of the entire object type's sort is assigned to the substate of exactly one method group. The state of IntSetClass is described as an integer-container field h associated with a method group

```
specialization specification IntSetClass

state [ h:IntHolder, c:IntSet ]
invariant e ∈ self.c ⇒ mem(self.h, e)

method group
  substate [ h:IntHolder ]
  existential ins, del, measure

  method add(el:Int)
    Modifies self.h
    Ensures self_post.h = ins(self_pre.h, el)

  method remove(el:Int)
     Modifies self
     Ensures self_post.h = del(self_pre.h, el)

  method size() returns(Int)
     Ensures result = measure(self_pre.h)

method group
  substate [ c:IntSet ]

  method member(el:Int) returns(Bool)
     Modifies self.c
     Ensures result = mem(self_pre.h, el)

  method invalidate()
     Modifies self.c
     Ensures self_post.c = {}

end IntSetClass
```

Figure 3: Specialization specification.

that contains the methods `add`, `remove`, and `size`, and a cache field c associated with a method group that contains the methods `member` and `invalidate`.

The values of h are modeled using a space of unordered integer containers called `IntHolder`. While the integer sets used in Fig. 1 are an appropriate sort for clients of `IntSetClass`, they are too specific for subclasses: a more general value space allows more subclasses. Fig. 4 describes `IntHolder` values using an LSL trait ([Guttag93]). A trait defines properties of function symbols that can be used in specifications; these functions define sorts. In this case, `IntHolderTrait` defines `IntHolder` in terms of the functions *new*, *ins*, *del*, *mem*, and *measure*.

```
IntHolderTrait: trait

introduces
 new: → IntHolder
 ins: IntHolder, Int → IntHolder
 del: IntHolder, Int → IntHolder
 mem: IntHolder, Int → Bool
 measure: IntHolder → Int

asserts
 IntHolder generated by new, ins
 ∀ h:IntHolder, i, i1, i2:Int
   ins(ins(h, i1), i2) = ins(ins(h, i2), i1)
   ¬mem(new, i)
   mem(ins(h, i1), i2) = (i1 = i2 ∨ mem(h, i2))
   mem(h, i) ⇒ ins(del(h, i), i) = h
   ¬mem(h, i) ⇒ del(h, i) = h
   mem(h, i) ⇒ measure(del(h, i)) < measure(h)
```

Figure 4: `IntHolder` trait.

After giving signatures for these functions, the trait asserts their properties. The **generated by** property states that all `IntHolder` values can be generated using just the *new* and *ins* functions. The next assertion states that applications of *ins* commute (thus, `IntHolder` is unordered). The next two assertions define *mem* in terms of *new* and *ins*. The final assertions are properties of *del* and *measure* assumed by the code for `member`.

Informally, we define correctness for specialization specifications as follows:

> **Def**: A class implements its specialization specification if, for each method group, the methods of that group will implement their specialization specifications for all implementations of other method groups that meet their own specifications.

By this definition, the correctness of a method group is not allowed to depend on a particular implementation of another method group. Instead, every method group must be correct for any implementation of other method groups. This way, if a subclass overrides a method group, inherited method groups will still be correct.

To maximize the freedom designers of subclasses have in defining the behavior of subclasses, the semantics of specialization specifications must be different from those of client specifications. In client specifications, function symbols are treated as if they were universally quantified over the entire program. In specialization specifications, we treat function symbols using a combination of universal and existential quantification.

Consider the trait:

```
FnTrait: trait
introduces
  fn:Int -> Int
```

that introduces a function symbol *fn* used in the following class:

```
class Fixpoint

% state [ ]

% method group
  % existential fn
    deferred method do_fn(x:Int) returns(Int)
      % Ensures result = fn(x)

% method group
    method is_fixpoint(x:Int) returns(Bool)
      % Ensures result = (x = fn(x))
      return(x = self.do_fn(x))
      end is_fixpoint

end Fixpoint
```

The constraints on *fn* are weak: all we know is that it is a function from integers to integers. In client specifications, function symbols like *fn* are quantified over the entire program, *i.e.*, they must denote the same function throughout the program, even if that function is not completely defined. This approach supports modularity by making it impossible for independent authors of two different modules to make local assumptions about function symbols that contradict each other. The only properties one can assume about a symbol are those properties listed in traits, which are shared globally.

In specialization specifications, function symbols are quantified over classes, *i.e.*, they must denote the same function within a class, but they can denote different functions in different classes. To see the difference, consider the following code fragment:

```
if o1.do_fn(0) = o2.do_fn(0)
  then x := true
  else x := false
```

where `o1` and `o2` are instances of different subclasses of `Fixpoint`. With client semantics, `x` must be true after the **if** statement because *fn* would denote the same function over the entire program and thus `do_fn` of `o1` and `o2` would have to compute the same function. With specialization semantics, `x` could be true or false after the **if** statement because *fn* could denote different functions in different subclasses so `do_fn` of `o1` and `o2` could be different.

Further, in specialization specifications, function symbols are interpreted using a combination of existential and universal quantification. Ordinarily, a method group interprets a specification symbol using universal quantification: the code of the group must be correct for all functions that satisfy the constraints put on the symbol by traits. However, a specification symbol can be assigned to (at most) one method group for existential interpretation using optional **existential** clauses. Inside that group, the symbol is interpreted using existential quantification: the code of the group is correct as long as it is correct for some but not necessarily all functions that satisfy the constraints put on the symbol by traits.

Existential interpretation facilitates specialization. For example, the `do_fn` group of `Fixpoint` uses existential quantification for *fn* and thus can be specialized for different values of *fn*. For example, the class:

```
class Zerofixpoint
superclass Fixpoint
method do_fn(x:Int) returns(Int)
   return(0)
   end do_fn
end Zerofixpoint
```

assumes that *fn* is the constant function zero; other subclasses of `Fixpoint` can assume different functions. The code for `is_fixpoint`, in contrast, uses universal quantification for *fn* and thus must be correct for any *fn*. As a result, `is_fixpoint` can be inherited into all subclasses of `Fixpoint` even though they can make different, possibly contradictory, assumptions about *fn*.

We can now offer a more detailed but still informal definition of correctness for specialization specifications:

> **Def:** A class implements its specialization specification if, for each method group G, the code of methods in G is correct for some values of the specification symbols assigned (via **existential** clauses) to G and for all values of other specification symbols.

5 Specialization verification

Our approach to verifying that a class meets its specialization specification is based on standard simulation techniques ([Hoare72]). However, verifying specialization interfaces raises issues not found in the traditional context. This section discusses three central ones:

- *Specializing method groups.* When verifying the correctness of methods whose specifications existentially quantify some function symbols, one can choose any interpretation of those symbols. Choosing appropriate interpretations of existentially quantified function symbols is the first step in verifying a group.

- *Verifying method groups independently.* After the existential function symbols of a group have been specialized, the code of the group is verified. The group is verified in terms of the specifications of the other groups to ensure that it will work with all implementations of those groups. This includes treating the substate of other groups abstractly in addition to treating their methods abstractly.

- *Constraining specifications of subclasses.* As mentioned in Sec. 3.2, specialization specifications of subclasses must be constrained by the specialization specifications of their superclasses. These constraints ensure that inherited code meets its specification and that overriden code will satisfy assumptions made about it by inherited code.

A wide range of techniques are available for dealing with each of these issues. The rest of this section discusses each point in turn, presenting basic techniques to handle the common cases.

5.1 Specializing method groups

A method group can be specialized as long as (a) the code implements at least the properties in its specification and (b) the code for other groups does not depend on the specialization. Code for a specialized method group is verified by specializing its specification and then verifying the code as discussed in Sec. 5.2.

The specification of a method group can be specialized in two ways. First, the group can be specialized by strengthening the specifications of its methods following the rules of behavioral subtyping ([Liskov94]).

Second, additional properties can be asserted about specification symbols that have been assigned to the group for existential interpretation. The additional properties must conservatively extend the old properties, *i.e.*, they must not contradict the old ones. These properties provide auxiliary information used in reasoning about the implementation of a group, and are similar to abstraction functions and representation invariants in this regard. Like abstraction functions and representation invariants, they can serve to document implementations.

In `IntSetClass`, the method group containing `add`, `remove` and `size` uses three properties to specialize `IntHolder` into integer sets:

$$\begin{aligned} ins(h,i) &= ins(ins(h,i),i) \\ mem(del(h,i),i') &= (i \neq i' \wedge mem(h,i')) \\ measure(h) &= \textstyle\sum_i (mem(h,i)\,?\,1:0)^{1} \end{aligned}$$

The first property asserts that *ins* is idempotent; this is a central property distinguishing sets from other unordered integer containers. The second and third properties define *del* and *measure* in terms of *mem*.

5.2 Verifying a group

Once the specification of a method group has been specialized, the code of the group can be

[1] The value of the conditional term "$t\,?\,c : a$" is c when t is true and a otherwise.

verified. A method group is verified independently of the implementation of other method groups; this ensures a method group will work regardless of whether other method groups it depends on are inherited from superclasses, implemented locally, or overriden by subclasses. Achieving this independence requires overcoming two difficulties:

- The verifier of a group does not have access to the code of other groups. The verifier knows the code of methods internal to the method group, but these methods call methods external to the group and the verifier does not know that code.

- The verifier does not have access to the full representation of **self**. The verifier only knows the locally-defined instance variables and not any instance variables defined by superclasses or subclasses.

Both of these difficulties are overcome using specifications. Although the verifier does not know the code of external methods, the verifier does know the specifications of external methods and reason about them in terms of these specifications. And although the verifier does not know the external instance variables, the verifier does know the abstract substate associated with those instance variables and can reason about them in terms of this substate.

Verification of specialization interfaces is done via simulation, *i.e.*, establishing an abstraction function from the concrete state of **self** to the abstract state and showing that methods simulate their specifications under this abstraction function ([Hoare72]). In traditional simulation proofs, a single abstraction function defines the entire state of **self**. When verifying specialization interfaces, the state of **self** is defined by multiple *subabstraction functions*. There is one subabstraction function for each method group, and these functions define only the substate assigned to that group. Each method group is verified with its own subabstraction function without knowledge of other groups' subabstraction functions. Overall correctness of the class is ensured by guaranteeing that a single abstraction function for the entire state of **self** can be constructed by taking the product of the subabstraction functions of each method group.

The concrete state of **self** is a set of instance variables. We assume that each instance variable is assigned to exactly one method group and that only the methods of that group read or write it. This assumption can be relaxed by adding an extra verification step, but we omit this extension for simplicity. Although languages do not enforce such assignments, they can be enforced by convention. Let R_i denote the instance variables assigned to the ith method group. We assume R_i is a tuple with named fields, one field per instance variable. Table 1 summarizes symbols defined in this section and gives particular values needed for verifying the `add` method group of `IntSetClass`.

In specialization specifications, the abstract state of **self** is partitioned into substates assigned to method groups. Let S denote the entire abstract state of **self** and S_i denote the substate assigned to the ith method group. As with R_i, we assume S and S_i are tuples with named fields, and we assume that $S = \prod_i S_i$, where tuple products take tuples with distinct field-names and return a new tuple with the fields of both.

For each method group, the verifier defines a subabstraction function A_i:

$$A_i : R_i \rightarrow S_i$$

This function defines how the group's instance variables are used to represent the group's substate.

We cannot verify the methods of group i directly in terms of A_i because A_i does not fully define **self**. In `IntSetClass`, for example, `remove` needs to manipulate the c part of **self** as well as the h part. Our solution is for group i to treat **self** as if it already includes the substates defined by other groups. That is, inside group i we assume that **self** is described by:

$$\textbf{self} : R_i \times \prod_{j \neq i} S_j$$

Under this assumption, we verify group i using an abstraction function V_i that has the signature:

$$V_i(s) : (R_i \times \prod_{j \neq i} S_j) \rightarrow S$$

Because **self** already contains $\prod_{j \neq i} S_j$ and S_i is defined by A_i, V_i is not freely chosen but rather is defined by the equation:

$$V_i(s) = (s \downarrow R_i) \times A_i(s \uparrow R_i)$$

where $s \downarrow R_i$ is s without the R_i fields and $s \uparrow R_i$ is s with only the R_i fields.

We verify the methods of group i using V_i according to standard simulation techniques (see, *e.g.*, [Liskov86, Dahl92]). Where these methods make calls to methods in other method groups, the specifications of the external methods are used to reason about the calls.

We illustrate our approach on the add method of IntSetClass. We do not perform a full verification here, but rather show pieces that illustrate the use of V_i to verify method groups. Recall that we have specialized this group by assuming integer set properties for IntHolder. To verify add, we need $A_{\texttt{ars}}$, the subabstraction function for its group. This function maps els (the instance variable assigned to the group) to h (the substate maintained by the group):

$$A_{\texttt{ars}}(s) = [h := toH(s.\texttt{els})]^2$$

where:

$$\begin{aligned} toH(empty) &= new \\ toH(addh(rest, i)) &= ins(toH(rest), i) \end{aligned}$$

and *empty* and *addh* are functions for building Array[Int] values. In other words, $A_{\texttt{ars}}$ constructs an IntHolder by inserting each element of **self**.els into an initially empty set.

From $A_{\texttt{ars}}$ follows the abstraction function $V_{\texttt{ars}}$ with which we verify the code in the add method group:

$$\begin{aligned} V_{\texttt{ars}}(s) &= [h := A_{\texttt{ars}}(s).h, c := s.c] \\ &= [h := toH(s.\texttt{els}), c := s.c] \end{aligned}$$

[2]The tuple constructor "$[f_1 := v_1, ..., f_n := v_n]$" denotes a tuple value where field f_i has value v_i.

Table 1: Symbols for verifying method groups. Symbols are parameterized by method group i. Also given are particular values for i = ars, the method group of IntSetClass containing add, remove, and size.

R_i	=	Instance variables assigned to group i
	=	[els : Array[Int]]
S_i	=	Sort of substate of group i
	=	[h : IntHolder]
S	=	Sort of entire state of **self**
	=	$\prod_i S_i$
	=	[h : IntHolder, c : IntSet]
A_i	=	Subabstraction function for group i
	:	$R_i \rightarrow S_i$
	=	$\lambda s \cdot [h := toH(s.\texttt{els})]$
V_i	=	Function for verifying group i
	:	$(R_i \times \prod_{j \neq i} S_j) \rightarrow S$
	=	$\lambda s \cdot (s \downarrow R_i) \times A_i(s \uparrow R_i)$
	=	$\lambda s \cdot [h := toH(s.\texttt{els}), c := s.c]$

To verify a method in the add group, we compose $V_{\texttt{ars}}$ with the specification of the method to transform the specification into the domain of the implementation. For example, composing $V_{\texttt{ars}}$ with the **ensures** clause of add yields:

$$V_{\texttt{ars}}(\textbf{self}_{post}).h = ins(V_{\texttt{ars}}(\textbf{self}_{pre}).h, \texttt{el})$$

which expands to:

$$toH(\textbf{self}_{post}.\texttt{els}) = ins(toH(\textbf{self}_{pre}.\texttt{els}), \texttt{el})$$

We then use the proof rules of the language to show that the code of the method meets its transformed specification.

In the code for add, the addh method of **self**.els is called only when el is not already in $V_{\texttt{ars}}(\textbf{self}_{pre}).h$. Thus, the verification of add proceeds in two cases:

1. When el is in **self**$_{pre}$.els, we can conclude that

   $$mem(V_{\texttt{ars}}(\textbf{self}_{pre}).h)$$

   Also, because addh is not called to change **self** in this case, we can conclude:

   $$V_{\texttt{ars}}(\textbf{self}_{post}).h = V_{\texttt{ars}}(\textbf{self}_{pre}).h$$

Putting these two conclusions together with the lemma

$$mem(h, i) \Rightarrow h = ins(h, i)$$

which follows from idempotence of *ins*, we can show that:

$$Vars(\mathbf{self}_{post}).h = ins(Vars(\mathbf{self}_{pre}).h, \mathtt{el})$$

So the **ensures** clause is met in this case.

2. When el is not in $\mathbf{self}_{pre}$.els, we need to reason about the effects of the addh invocation. It follows from the definition of *toH* that:

$$toH(addh(array, e)) = ins(toH(array), e)$$

From this we can show that after the addh method:

$$toH(\mathbf{self}_{post}.\mathtt{els}) = ins(toH(\mathbf{self}_{pre}.\mathtt{els}), \mathtt{el})$$

So the **ensures** clause is met in this case too.

A complete verification of add also involves verifying the **modifies** clause and verifying that the code preserves the invariant. The **modifies** clause requires that only the *h* field of **self** is modified. The invariant requires that *c* is a subset of *h* after the call if it is a subset before the call. We do not present these verifications here.

Verification of member is interesting because it depends on the invariant of the specialization specification of IntSetClass. $A_{\mathtt{mi}}$, the subabstraction function for the method group containing member and invalidate, maps the instance variables valid and cache to the substate field *c*:

$$A_{\mathtt{mi}}(s) = [c := (s.\mathtt{valid}\,?\,\{s.\mathtt{cache}\} : \{\})]$$

From this we can define an abstraction function $V_{\mathtt{mi}}$ for member:

$$V_{\mathtt{mi}}(s) = [h := s.h, c := A_{\mathtt{mi}}(s).c]$$

Composing this abstraction function with the invariant of IntSetClass yields:

$$e \in (\mathbf{self}.\mathtt{valid}\,?\,\{\mathbf{self}.\mathtt{cache}\} : \{\}) \Rightarrow mem(\mathbf{self}.h, e)$$

which simplifies to:

$$\mathbf{self}.\mathtt{valid} \Rightarrow mem(\mathbf{self}.h, \mathbf{self}.\mathtt{cache})$$

Given this, one can conclude that it is correct for member to return true if valid is true and the argument to member equals cache.

5.3 Constraining subclasses

A class with no superclasses (*e.g.*, IntSetClass) can be verified by explicitly verifying each of its method groups as described above. However, we cannot explicitly verify method groups inherited from superclasses because we do not have access to their code. Instead, we verify inherited methods by requiring that the specialization specification of a class is properly related to that of its superclass.

Assume that the sub- and superclass specifications have the same value sort, invariant, and constraints, and the same division of labor specification (we will relax this assumption in a moment). In this case, the two specifications are properly related if:

1. The subclass specifications of all methods imply their superclass specifications.

2. The subclass specifications of inherited methods are implied their superclass specifications.

By our definition of correctness for specialization specifications, inherited methods will continue to behave as specified in the superclass if overriden methods meet the assumptions stated about them in their superclass specifications. The first rule ensures that methods overriden by the subclass do meet the assumptions stated about them in their superclass specifications. Because of the first rule, we can assume that inherited methods behave as specified in the superclass. Rule two ensures that we assume only the behavior specified in the superclass and not something stronger.

It might seem that rule one need not apply to inherited methods because inherited methods already meet assumptions made about them by the superclass. However, a subclass of the subclass might override a method group that is inherited by the subclass. By requiring that the subclass specifications of inherited methods imply their superclass specifications, assumptions made about inherited methods are passed along to subsubclasses that may override them.

The above two rules are the basic requirements for soundness. Additional rules allow the sub- and superclass specifications to vary more broadly. For example:

- The invariant and constraint in the subclass can be stronger than in the superclass.
- The sort defining the value space of the subclass can be different from the sort of the superclass if the two specifications can be related by an abstraction function.
- The subclass can merge two or more method groups of the superclass into a single group.
- The subclass can add new methods to methods groups it is overriding and can add entirely new method groups.

Even more differences are possible.

6 Client verification

At run-time, a program manipulates objects, not classes. Objects are described by client specifications, not specialization specifications. Thus, the ultimate goal of our methodology must be to ensure that classes meet their client specifications.

A client specification for `IntSetClass` is given in Fig. 5. As an aside, comparing this specification to the one in Fig. 3 illustrates an important difference in the content of client and specialization specifications. The client specification of `IntSetClass` is specified in terms of the more specific `IntSet` rather than the more general `IntHolder`, yet the client specification hides the internal cache exposed by the specialization specification. In general, client specifications tend to be more specific yet less detailed than specialization specifications.

To validate the client interface of `IntSetClass`, we could directly verify the code in Fig. 1 against the client specification in Fig. 5. However, this would not work for classes with superclasses because the code for inherited methods is not available for verification.

```
client specification IntSetClass
state IntSet

method add(el:Int)
  Modifies self
  Ensures self_post = self_pre ∪ {el}
  end add

method remove(el:Int)
  Modifies self
  Ensures self_post = self_pre − {el}
  end remove

method size() returns(Int)
  Ensures result = |toSet(self_pre)|
  end size

method member(el:Int) returns(Bool)
  Ensures result = el ∈ self_pre
  end member
end IntSetClass
```

Figure 5: Client specification of `IntSetClass`.

An alternative approach is to structure the verification of the specialization interface such that the correctness of the client interface follows as well. In fact, the verification process described in the previous section is already so structured. In that process, the specification of a method group is specialized before the code of that group is verified. When verifying the code of one method group, the unspecialized specifications of the other groups are used. To verify the client specification, the specializations of each method group are combined to form a single, specialized specification; if the specializations for each group are chosen correctly, this specialized specification will imply the client specification.

After all groups are specialized, the specialization specification must imply the client specification, but it need not be equal to the client specification. The implication can be checked in many ways, *e.g.*, by using the simulation rules in [Liskov94]. For example, the specialized specialization specification for `IntSetClass` is the specification in Fig. 3 specialized with new properties for *ins*, *del*, and *measure*; clearly this does not equal the specification in Fig. 5. However, using the abstraction function:

$$\mathcal{A} : \texttt{IntHolder} \rightarrow \texttt{IntSet}$$

where:

$$\begin{aligned} \mathcal{A}(new) &= \{\} \\ \mathcal{A}(ins(h, i)) &= \mathcal{A}(h) \cup \{i\} \end{aligned}$$

the specialized specification can be shown to imply the desired client specification.

Client invariants can be used to help show that a specialization specification implies a client specification. Client invariants arise because public methods often preserve properties that are not preserved by protected methods. Because clients cannot break these properties, they are invariant to clients and can be hidden from them. For example, consider the class:

```
class C
  x:Int

  protected method add(d:Int)
    self.x := self.x + d
    end add

  method inc2() returns(Int)
    self.add(2)
    return(self.x)
    end inc2
  end C
```

The client interface can incorporate the invariant that x is always even (assuming the initial value of x is even). However, the specialization interface cannot because of subclasses like the following:

```
class D
  superclass C

  method inc() returns(Int)
    self.add(1)
    return(x)
    end inc
  end D
```

Client invariants can be incorporated as explicit invariants in the client specification or they can be incorporated into the abstraction function that maps the specialization specification to the client specification.

7 Related work

[Lamping93] and [Lamping94] are related to our work in their emphasis on subclassing and the specialization interface. Both [Lamping93] and [Lamping94] are type-system oriented, where our work is specification oriented; thus, *e.g.*, they do not help one reason about the behavior of inherited methods. [Lamping93] suggests partitioning the specialization interface into method groups as we do; however, it does not suggest associating substate with method groups, which we feel is important to using subclassing in a modular manner.

[Kiczales92] presents design and documentation guidelines for class libraries. The documentation guidelines are analogous to our specialization specifications. [Kiczales92] uses informal specifications where we use formal specifications. Also, the specifications proposed by [Kiczales92] are operational, *i.e.*, they explain the behavior of a method in terms of invocations of other methods. For example, their specification of `member` would explain that `member` calls `remove`, `add`, and `size` to test membership, and their specification of `remove` would explain that it calls `invalidate`.

We use declarative specifications instead. By allowing only the methods of a method group access to the representation of the group's substate, our declarative specifications can force methods like `member` and `remove` to invoke other methods. At the same time, declarative specifications are likely to be more abstract than operational ones, leading to specifications that are easier to understand and are less likely to capture accidental implementation details such as invocation order.

Behavioral subtyping is another area of related work ([Leavens90, America91, Liskov94]). Work on behavioral subtyping consists of specification and reasoning techniques to facilitate modular use of subtyping, while our work consists of specification and reasoning techniques to facilitate modular use of subclassing. However, the work is different in many details. The specifications needed for subclassing are different from those needed for subtyping. Subclassing requires division of labor specifications and mixed universal and existential quantification of specification symbols over a single class, while subtyping requires universal quantification over the entire program. Also, subclassing allows a subclass to specialize the interpretation of function symbols, while subtyping only allows a subtype to strengthen the specifications of methods.

Our work also relates to the separation of subtyping from subclassing. If D is a subclass of C, does D always implement a subtype of C? Although many languages implicitly assume that the answer is "yes," researchers working on object-calculi and type systems argue that the answer is "no" (see, *e.g.*, [Cook90, Bruce93]). However, according to these arguments, subtyping and subclassing differ only when the type system has the type expression **MyType** (the type of **self**). Many object-oriented languages do not support **MyType**, and for these languages the object-calculi argument for answering "no" is not by itself compelling.

Our work provides reasons to separate subtyping and subclassing even when **MyType** is not part of the type system. Sec. 6 shows that the client specification of a class can differ significantly from the specialization specification because of either specialization or client invariants. As a result, the client specification of a class need not be a subtype of the client specification of its superclass.

8 Summary and conclusions

We have identified two obstacles to using subclassing in a modular manner: grouping dependencies and behavior dependencies. To handle these difficulties, we define a new kind of specification and associated reasoning techniques for the specialization interface of classes. Grouping dependencies are handled by partitioning the method and abstract state of a class into method groups. Behavior dependencies are handled by requiring a particular relation between the specifications of a subclass and its superclass.

Our methodology has some limitations. Although we are extendeding it to handle multiple inheritance, it is not clear that it can be extended to handle multiple dispatch. Also, our verification methodology verifies partial correctness only. Separate termination arguments are needed for total correctness, and we currently have no systematic approach to such arguments.

Our results should be useful for designers, programmers, and users of class libraries. Division of labor specifications give designers a systematic framework for thinking about design. Designers should design classes intended to be subclassed such that part-way between the representation and the client interface there is a semi-abstract interface that reveals the implementation strategy without revealing implementation details. The methods and abstract state of the class should be partitioned into method groups that can be understood independently and overriden as a whole. Although existing languages do not support method groupings, method groupings can be supported using simple programming conventions.

While programmers usually do not formally verify their code, they do reason about it informally. Our methodology gives some rules for doing so. First, programmers should assign instance variables to method groups and avoid reading or writing an instance variable assigned to one group in a method assigned to another. The instance variables of a group should be used to implement only the abstract substate assigned to the group and not the substates of other groups. A group can implement behavior that is more specific than what the specialization specification calls for, but the implementation of one group should not depend on the specialized behavior of another.

Perhaps the most useful application of our work is in improving the documentation of class libraries. It has been suggested that vendors of object-oriented libraries such as Microsoft's Foundation Classes and Borland's Object Windows Library distribute the source code for those libraries because the documentation of specialization interfaces is inadequate ([Atkinson92]). This claim is consistent with our own claim that programmers currently have to either guess or look at source code to deal with grouping and behavior dependencies.

Our results provide an approach to informal documentation of classes that should eliminate the need to use source code as documentation. The first step in improving documentation is to separate the specifications of the client and specialization interfaces: the specifications of client interfaces should be more specific yet less detailed than specialization specifications. The second step is to use division of labor specifications for specialization interfaces.

Acknowledgements

This work arose from combining the ideas in [Liskov93] with those in [Lamping93]; we thank John Lamping, Barbara Liskov, and Jeannette Wing for patiently answering our e-mail regarding their papers. Our thinking benefited from discussions with Luca Cardelli, Steve Garland, Nate Osgood, Mark Reinhold, and Mark Vandevoorde. We thank David Evans and Deepak Kapur for lengthy and helpful discussions and comments on earlier drafts of this paper, and Sheryl Risacher and Jeff Schachter for proof reading.

References

[America91] P. America. Designing an object-oriented programming language with behavioural subtyping. *Foundations of Obj.-Orien. Lang.* (Noordwijkerhout, The Netherlands, May/June 1990). Published as *LNCS 489*, pages 60–90. Springer-Verlag, 1991.

[Atkinson92] B. Atkinson. Panel: reuse—truth or fiction. *OOPSLA '92 Conf. Proceedings* (Vancouver, Oct. 1992). Published as *SIGPLAN Notices*, **27**(10):41–2. ACM, Oct. 1992.

[Bruce93] K. B. Bruce. Safe type-checking in a statically-typed, object-oriented programming language. *Proc. 20th Annual Symp. on Princ. of Prog. Lang.* (Charleston, SC, Jan. 1993), pages 285–98. ACM, Jan. 1993.

[Cook90] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. *Proc. 17th Annual Symp. on Princ. of Prog. Lang.* (San Francisco, CA, Jan. 1990), pages 125–35. ACM, Jan. 1990.

[Dahl92] O.-J. Dahl. *Verifiable Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Guttag93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[Hoare72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, **1**(4):273–81. Springer-Verlag, 1972.

[Kiczales92] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. *OOPSLA '92 Conf. Proceedings* (Vancouver, Oct. 1992). Published as *SIGPLAN Notices*, **27**(10):435–51. ACM, Oct. 1992.

[Lamping93] J. Lamping. Typing the specialization interface. *OOPSLA '93 Conf. Proceedings* (Washington, DC. Oct. 1993). Published as *SIGPLAN Notices*, **28**(10):201–14. ACM, Oct. 1993.

[Lamping94] J. Lamping and M. Abadi. Methods as assertions. *ECOOP '94 Proceedings* (Bologna, Italy, July 1994). Published as *LNCS 821*, pages 60–80. Springer-Verlag, 1994.

[Leavens90] G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes. *ECOOP/OOPSLA '90 Conf. Proceedings* (Ottawa, Canada, Oct. 1990). Published as *SIGPLAN Notices*, **25**(10):212–23. ACM, Oct. 1990.

[Liskov86] B. Liskov and J. Guttag. *Abstraction and specification in program development.* MIT Press/McGraw-Hill Book Co., 1977.

[Liskov93] B. Liskov and J. M. Wing. Specifications and their use in defining subtypes. *OOPSLA '93 Conf. Proceedings* (Washington, DC, Oct. 1993). Published as *ACM*

SIGPLAN Notices, **28**(10):16–28. ACM, Oct. 1993.

[Liskov94] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Sys.*, **16**(6):1811–41. ACM, Nov. 1994.

[Taligent94] Taligent. *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*. Addison-Wesley, Reading, MA and London, UK, 1994.