

Logical Method Combination

John Lamping, Mike Dixon
Xerox PARC

Abstract

We present an account of method overriding and combination that can be used to richly describe how methods ought to combine and to build method combination facilities based on those descriptions. The idea is that classes are associated with requirements on the behavior of operations, while methods provide code fragments that meet specified requirements under certain conditions. A method is applicable to an operation if it meets all requirements of the classes of the arguments for that operation. We show how to express overriding by having subclasses add additional requirements, and how to express method combination as a composition of code fragments that will meet all requirements. All of this can be captured with first order logic.

1 Introduction

Object oriented programming has been characterized as having three key technical components: object encapsulation, polymorphism via dispatch, and

Category: Research paper

Topic area: Language design and implementation

Contact: John Lamping

Xerox PARC

3333 Coyote Hill Road

Palo Alto CA 94304

415-812-4735 (phone) 412-812-4334 (fax)

lamping@parc.xerox.com

implementation inheritance. The first two are the most widely used. Distributed object models, for example, emphasize encapsulation and polymorphism. But implementation inheritance, with its additional mechanism for cooperation between code is also an important component of many systems, particularly complex ones.

This paper addresses implementation inheritance. In the simplest kind of implementation inheritance, a programmer starts with a specific existing class and builds a subclass which explicitly adds or overrides some methods. This paper focuses on the more complex case where a programmer combines several classes via multiple inheritance, and expects their methods to be combined in a reasonable way. This is a situation where two pieces of code interact, neither of which was necessarily written with knowledge of the function of the other.

Not only is method combination an important tool for some complex object oriented systems, but recent developments in object orientation may make it more important. On the research side, for example, both subjectivity [6] and adaptive software [9] are providing new ways of combining existing class-like information, and thus both require method combination. An example on the commercial side is IBM's SOM [4], which automatically generates multiple inheritance of metaclasses.

But the current mechanisms for method overriding and method combination are fairly ad-hoc, and there is no consensus on how they should behave. For example, if one class multiply inherits from two other classes, each of which provides a method for some operation, C++ [14] will generate an error, while CLOS [13] will choose one of the methods. Similarly, the only kind of method combination that C++ supports is the ability of an overriding method to call on the overridden method, while CLOS supports several different combination mechanisms, including "before" and "after" methods.

In this paper, we present a new account of what methods are, which we then use to address method combination. The idea is to decompose what an operation is supposed to do into a set of requirements, only some of which will apply to any particular invocation of the operation. Which requirements are in force for any particular invocation will depend on the classes of the

arguments, with arguments belonging to subclasses imposing, in general, more requirements. Method declarations do two things: they indicate that certain requirements apply for certain argument classes, and they provide code that meets various requirements. The correct code for an operation will be that code or combination of code that meets all the requirements for the classes of the argument objects. Method combination becomes a matter of identifying a combination of code that satisfies all pertinent requirements.

This approach, basing method combination on a description of what requirements are satisfied by methods, differs from current approaches, which base method combination primarily on the location of methods in the class inheritance structure. We contend that location of methods in a class inheritance structure doesn't carry enough information to do a good job of method combination in the presence of multiple inheritance. Each of the current approaches, for example, give unreasonable results in easy to find cases.

To illustrate our approach, suppose that we have a `FileDrawer` class, intended to support `save` and `retrieve` operations to file things into a file drawer under a name and to get them back out again. A method declaration for the `save` operation would indicate a requirement that `save` operate as expected: when you save something successfully it gets put in the file drawer¹ and it would provide code that obeyed that requirement. Now add a `LockableFileDrawer` subclass, that adds `lock` and `unlock` operations on the file drawer, with the intent that you can't save or retrieve anything from a locked file drawer. A method declaration for `save` on lockable file drawers would add a requirement that no saving happens on a locked file drawer and would provide code that checked the lock state and then called a superior method if the file drawer was unlocked. Next add a different subclass of `FileDrawer`, `SizedFileDrawer`, intended to record the total size of everything in the file drawer. A method declaration for `save` on sized file drawers would add a requirement that the size total be updated and would provide code that updated the size and then called a superior method.

¹This is not an algebraic definition, which would define `save` and `retrieve` in terms of each other, because we want to be able to check individual code against requirements.

Now comes the big moment, and the payoff. Construct a class that inherits from both `LockableFileDrawer` and `SizedFileDrawer`. Clearly, the lock check must be done before updating the size total, eg: first do the lock check, then update the size, then add the thing to the file drawer; any other order would be wrong on locked files. This will be deducible from the kinds of requirements involved. In particular, we will characterize the lock check as a checking requirement, while the size update is an activity requirement. If code that does activity is executed before code that does checking, then the checking requirement is not satisfied by the composition. This lets us conclude that the check must be done first if all the requirements are to be satisfied by the composition. Notice that there is no way to determine the correct order of combination just from the class structure; information about the methods is required, which is what our proposal makes it possible to provide.

This example also illustrates that there is no need to spell out the exact details of the requirements. All that is needed is to know which methods satisfy which requirements and how the satisfaction of requirements is affected by method combination. This kind of information is both easy to reason about and easy to express. It is information that programmers are already thinking about (or should be, if they are anticipating having their methods combined).

Characterizing methods in terms of what kinds of requirements they involve is superficially reminiscent of categorizing methods by qualifiers like `:before` or `:after` under standard CLOS method combination. The focus here, though, is on the requirements and their properties, rather than on combination mechanisms. This leads to a rich and meaningful way of describing methods, and one that is easy to extend. It also allows for flexibility in determining how to combine methods, as well as the possibility of concluding that some methods shouldn't be combined.

This approach leverages object orientation's separation of the task of selecting a method from the task of executing the method code. The former is a formally simple, declaratively driven, approach; while the latter can be almost any computation model, including imperative, functional, or logic

programming. The split allows clean automatic tools, like method combination, to be used for method selection, while leaving the execution model free to embrace the complexities of real computational environments.

The split means that we are free to provide a computationally complex method selection and combination approach without contaminating method execution. Since the split allows the reasoning needed for the method selection process to be done at compile time or at link time, there is no runtime penalty for the added richness: no added runtime computation, and no added constraints on the runtime environment. For example, the approach described here could be implemented using the long form CLOS `define-method-combination`.

Taking advantage of the split is one of the main differences between this approach, and embeddings of object orientation in logic programming [1, 7, 12]. This approach is an extension of the idea of methods as assertions [8] to the domain of method combination. The focus on the intended behavior of operations, analogous the “open semantics” treatment of procedures in [3], allows methods to be more than components of classes. Methods provide free-standing pieces of code that come with information about what they accomplish.

The next section goes into more detail on the new account of methods. Then the paper uses this viewpoint to explore different kinds of requirements and methods. The appendix gives a straightforward expression of the concepts developed into first order logic.

2 Basic Concepts

First, we review some basic concepts that appear in our account.

Objects are just the familiar objects of object oriented programming, like a file drawer; they encapsulate both state and behavior. Objects can be purely functional, or they can have mutable state; that won't matter for this paper, because the issue is how to choose methods, not whether methods mutate state.

Operations are things that one does to objects, like the operation of saving

something in a file drawer. (Operations are called “messages” in Smalltalk [5] and “virtual member functions” in C++.) There is some intended behavior for an operation, which should be consistent across a range of objects. Thus, operations have a meaning independent of any methods that implement them. It is up to the methods, rather, to correctly implement the intent. Of course, some details and fleshing out of the behavior will depend on what object or objects the operation is performed on.

A requirement is a property that code can meet. For example, a requirement could be that a procedure adds an item to a file, unless some preconditions are violated. Requirements, themselves, don’t mention either operations or classes, except as necessary to meaningfully describe the requirement. Another requirement, for example, could be that a procedure does nothing if the first argument is a locked object. A requirement, in other words, is something that you could check that a piece of code satisfies. For the language fragment that we will investigate, the responsibility for actually doing the check will be the programmer’s, which means, in exchange, that the programmer doesn’t actually have to spell out the requirements, only say when they apply and when they are satisfied.

A class is a set of related objects, such as the class of file drawers; and subclasses are subsets. This definition is meant to include both the notion of classes as types, such as all objects satisfying an interface, and the notion of implementation classes, such as all objects implemented a certain way. A class could profitably mix some of both. For example, a method might depend on the availability of some interface. Since this paper is primarily concerned with how methods are selected, the classes mentioned here will all have at least some implementation consequences.

A class is associated with the requirements that pertain to operations on objects in it. More generally, several classes can determine that a requirement applies to an operation when given arguments of the respective classes. The user of objects of the classes can then know that the behavior of the operation will be as specified by the requirements.² Since a class is a subset of its

²Another part of the description of a class could be what operations are supported by the class, the traditional type of the class. Since this paper is not particularly concerned

superclasses, objects in the class belong to the superclasses as well, and must meet all requirements of the superclasses.³

So far, we have talked about things visible to clients of an object. Code descriptions, in contrast, tell how to implement objects. A code description is an assertion that a certain piece of code satisfies certain requirements, under the proper conditions. For example, we could say that a certain piece of code meets the requirement of adding an item to file drawer objects. That would imply that it was appropriate for the `save` operation on basic file drawers. Of course, if a subclass of file drawer adds additional requirements, then this code will not be correct (at least by itself) on the subclass.

Notice that code descriptions don't mention operations, just requirements. Classes and requirements provide the link between code descriptions and operations. A class determines requirements on an operation, and code is applicable to the operation if it satisfy the requirements.

3 Stand Alone Methods

Method declarations are a secondary concept under this account. They are a combination of an assertion that a requirement pertains in certain circumstances and a description of some code that helps meet that requirement. It would be possible to design a programming language consistent with this account that separated these two functions, but we will focus here on how to apply the account to constructs similar to traditional methods.

We will use the account to describe various kinds of methods, and introduce a programming language construct to express each kind. The description in the text will be careful but informal; a translation of each construct into first order logic appears in the appendix.

with typing, these kinds of constraints won't be explored. It does make sense, however to declare methods to be valid only on classes that support certain operations. The treatment in this paper could be extended with the kind of conditions in [8] to accomplish that.

³One could also have a notion of a derived class that was not a subset, and that removed some requirements as well as added others. This might be a useful capability in some circumstances, but we will not address it, because we don't see a coherent interpretation of multiple inheritance, the situation of interest to us, from such classes.

The simplest method is a stand alone method, such as the basic method for saving things in file drawers. Such a method comes with a requirement, in this case the requirement that an activity add a item to a file drawer; specifies that the requirement applies to some operation on some classes, in this case to the `save` operation on file drawers; and describes code that satisfies the requirement if its arguments are of the specified classes, in this case the code actually saves items in file drawers. Since the code description says that the code satisfies the requirement only if the arguments are elements of the specified classes, the method will never apply to superclasses.

We can introduce a syntax for our fragment for such methods as

```
method save (t:any, key:string, f:FileDrawer) begin...end
```

or more schematically

```
method <op> (<v1>:<cl1>, ..., <vn>:<cln>)<body>
```

Notice that the method doesn't actually mention the requirement. There is no need, because, as mentioned already, all that matters about the requirement, as far as the language is concerned, is under what circumstances it applies to operations and what code satisfies it.

Also notice that there is no way to tell from the syntax of the declaration whether or not a method is a multi-method, one that dispatches on the classes of several of its arguments. That can only be determined from the entire collection of methods for an operation; they are multi-methods if they differ on the class of more than one argument.

A method declaration like this does not imply that the code is the right way to carry out the operation on all arguments of the specified classes. It can't guarantee this because there might be other requirements on the operation that are not met by the code. In fact, the code in the method for `save` is not the right code to dispatch to for either of the subtypes of file drawers mentioned in the introduction, like lockable file drawers, because those subtypes have additional requirements that the code does not meet (at least not by itself).

On the other hand, if nothing else has been said about the `save` operation on “vanilla” file drawers, we would like to dispatch to the method’s code for those file drawers. Ordinary object creation naturally yields such “vanilla” objects. Suppose that a programmer asks for a new instance of `FileDrawer`. They would be surprised to get an instance of `LockableFileDrawer`, even though that would be a file drawer object. The point is that programmers who are asking for a new object don’t expect to get more than they ask for. A programmer who asks for a new file drawer object expects a “vanilla” file drawer object, for which the above method is the correct code to dispatch to for save operations. We will call the class that the programmer requested in instantiating an object its *instantiation class*.⁴ While an object will belong to lots of classes, it will have just one instantiation class. Likewise, not all of the objects in a class will have it as their instantiation class. The idea is that an object should be a “vanilla” member of its instantiation class. We can capture the notion of “vanilla” in the following principle:

When an operation is invoked on some objects with instantiation classes C_1, \dots, C_n , code is applicable to the invocation if the code can be proven to satisfy all requirements that can be inferred for the operation when applied to objects in classes C_1, \dots, C_n , under the assumption that the code is applied to objects of class C_1, \dots, C_n .

If all that has been declared about `save` on the class `FileDrawer` is the one method declaration, then only the one requirement can be inferred, and this principle means that the method’s code is applicable to the `save` operation for objects that result from the instantiation of `FileDrawer`. This does not, crucially, allow us to conclude that code is applicable to all objects that are members of `FileDrawer`; it won’t be applicable to lockable file drawers, for example. All we have concluded so far that the code is applicable to objects whose instantiation class is `FileDrawer`, objects that resulted from specifically instantiating the `FileDrawer` class. In other words, it is applicable to “vanilla” file drawers.

⁴This has also been called the runtime class of the object.

This discussion points out that classes are really used in two different ways in programs, one focussing on the set of objects in the class and one focussing on the description associated with the class. When we say that we know that the value of a variable is a member of some class, we know that it is one of the set of objects that belong to the class, which lets us conclude various things about the object, but we don't know what subclasses it might also belong to. But when we ask to create a new object of a particular class, we are asking for a generic, or vanilla, instance of that class, an instance that satisfies the description and no more. The instance will not, for example, be an instance of any strict subclass.⁵

4 Subclasses and Method Overriding

To do overriding, we first need subclasses. A subclass declaration says that the objects in subclass form a subset of the objects in the class. Thus, objects of the subclass must obey not only the subclass's requirements, but also requirements that pertain to superclasses. A syntax for subclassing can look like:

```
declare LockedFileDrawer subclass of FileDrawer
```

or more schematically,

```
declare <subclass> subclass of <class>
```

Since subclasses can add more requirements, the principle for method selection allows for overriding, and other kinds of method combination. First, consider the simplest kind of overriding method, one that doesn't call on the superior method. A method that does all the work for saving things in

⁵The two different kinds of uses here are *not* analogous a type hierarchy vs. implementation hierarchy separation. The requirements that a class imposes, for example, could legitimately be considered part of the exported interface of the class, as in [15, 11], yet they also factor prominently in determining what code implements the operations of the class.

lockable file drawers, would be an example. (Later we'll see a better way to do the same thing using a combinable method). An overriding method brings its own added requirement, in this case the requirement that an activity check the lock before doing anything else; specifies that the requirement applies to some operation on some classes, in this case to the `save` operation on lockable file drawers; and describes code that satisfies that requirement, as well as any requirements on super classes, provided its arguments are of the specified classes, in this case the code checks the lock and then saves items.

A syntax can look like

```
overriding method save(t:any, key:string, f:LockableFileDrawer)
  begin...end
```

or more generically,

```
overriding method <op>(<v1>:<cl1>, ..., <vn>:<cln>) <body>
```

The added requirement that this method specifies on lockable file drawers means that the code from the basic method on file drawers won't work, because it doesn't meet the additional requirement. Meanwhile, the code in this method is described as meeting the new requirement, as well as the original requirement, since this kind of method declaration says that the code meets all requirements of superclasses. Thus, the code is applicable to the save operation for lockable file drawers. Finally, since the code description says that the code meets the requirements only if the file argument is a lockable file drawer, this code doesn't apply to simple file drawers. Notice that if we hadn't said anything about saving in lockable file drawers, then the original method would still apply; it acts as a default, which can be overridden if we say more.

Now it is natural to ask what happens if there is multiple inheritance, where one class inherits from two classes, both of which have an overriding method. Suppose, for example that we have written an analogous overriding method for `SizedFileDrawers`, which will introduce a new condition on `save` and a new method that meets that condition and the basic condition.

Now introduce a class that inherits both from `LockableFileDrawer` and `SizedFileDrawer` and ask what code is applicable for `save`. The answer is that none of the code is applicable, since none meets all three requirements, and that is the correct answer, since none has all the expected behavior.⁶

This is also the answer that C++ gives, in contrast, for example to CLOS, which would pick one of the methods. But before we leap to praise C++, we should note that there is another kind of overriding situation. Suppose that there is a class, `SmallFileDrawer`, which is just like a file drawer, except that we know that it will never be asked to contain more than ten objects. In that case, we might have a special method for `save` on small file drawers that used this fact to get better efficiency. In this case, there are no new requirements for `SmallFileDrawer`; the original method of `save` is still correct. It's just that `SmallFileDrawer` presents an opportunity for more efficiency. In short, we would prefer to use the new method, but it is not necessary.

We might write a syntax for this as

```
preferred method save:any, key:string, f:SmallFileDrawer)
  begin...end
```

or more schematically as

```
preferred method <op><v1>:<cl1>, ..., <vn>:<cln>) <body>
```

The way to make this method have the desired behavior is to not have it introduce any new requirements. It just describes the code as satisfying any requirements from super classes, provided its arguments are of the specified classes. That is, small file drawers are file drawers, with no added requirements on their behavior, and the new code works on them, but nothing invalidates the original code. We introduce a new code selection principle to handle such cases:

Where several pieces of code are applicable, and one has more specific requirements than the other, then the more specific one is chosen.

⁶This is analogous to the situation in Cecil where there may be no applicable multi-method when two class hierarchies are combined [2].

This principle is enough to cause the new method's code to be selected for small file drawers.

While this has the same immediate effect as an overriding method, the difference shows up under multiple inheritance. Consider a subclass of both `SmallFileDrawer` and `LockableFileDrawer`. In this case, the `save` code from `LockableFileDrawer` is correct, but the one from `SmallFileDrawer` isn't. This is what can be deduced from the methods. In the case of multiple inheritance of two preferred methods, either would be applicable.

The point here is that overriding methods come in two flavors, and while both have the same consequence under single inheritance, they are very different under multiple inheritance. Our account makes it possible to talk about the distinction, and to get appropriate behavior in the presence of multiple inheritance.

One might object that programmers are likely to get this distinction wrong in practice, since it only matters under multiple inheritance. But the programmer who must make the distinction is the one who is trying to write a class library that supports multiple inheritance. This is exactly the kind of distinction that they should be thinking about, and is simpler than many issues that come up in designing reusable libraries.

5 Combinable Methods

The code for a straight overriding method, like the examples above, takes on the entire task of meeting all the requirements. A combinable method, on the other hand, is one whose code is designed to combine with other code to jointly satisfy the requirements. The combination typically works via the combinable code being logically wrapped around the combinee code to create the combined code. The place where the combinee code fits in is the site of something like a `send-super` call in the code of the combinable method. For example, the code for a combinable method for checking the lock on a lockable file drawer would check the lock and then do a `send-super` if the file was unlocked.

To know whether some combination of methods is applicable to some op-

eration on some objects, we need to know the requirements satisfied by the combination. Assume that for each combinable method, we know the requirements satisfied by the combined code in terms of the requirements satisfied by the combinee code. For example, if the code for the lock check is wrapped around some other code, the combination satisfies whatever requirements the other code satisfied, plus the additional requirement that nothing happens if the drawer is locked (assuming that the lock check doesn't, itself, violate the other requirements). With this kind of information, we can deduce the requirements satisfied by a sequence of combinations by starting with the requirements satisfied by the innermost code, and work up, layer by layer of the combination.

As in the lock check example, when the code for a combinable method is wrapped around some combinee code, the combination will satisfy some of the requirements satisfied by the combinee code, plus some additional requirements. But the combined code will, in general, violate some of the requirements met by the combinee code. A key to describing the code for combinable methods, then, is to describe which requirements met by the combinee code are still met by the combination, and which are not.

To achieve perspicuous descriptions, we'll classify requirements into interesting categories, and describe which categories of requirements are preserved by the code of different kinds of combination methods. For this paper, we'll need just two categories: activity requirements and checking requirements.

An activity requirement is the kind of requirement introduced by stand-alone methods and by typical overriding methods. An example is "This activity adds an entry to the file, provided all preconditions are satisfied." It typically describes something that an operation is expected to do whenever its arguments are of certain classes. This could include updating data structures, informing other parts of a system, or otherwise doing some piece of the operation.

A checking requirement, on the other hand, is like the one introduced by lockable file drawers, which says that nothing should happen unless some condition is satisfied.

Given this simple categorization, we can describe a variety of methods.

A checking method would be the preferred way to handle the lock check on a lockable file drawer. It adds a new checking requirement to the operation, that nothing happens if the drawer is locked; and describes combinable code where, provided the arguments belong to the expected classes, the combined code satisfies all the activity and checking requirements of the combinee code, plus the new checking requirement, in this case, the code does the check and then does a `send-super` if the file is unlocked. Given this information, it is easy to infer that the combination of the lock check code and the basic save code will be valid for saving in lockable file drawers.

We might write a syntax for this as

```
checking method save:any, key:string, f:LockableFileDrawer)
  begin...send-super end
```

or more schematically as

```
checking method <op>(<v1>:<cl1>, ..., <vn>:<cln>) <body>
```

In order for a combination with a check actually to satisfy the activity requirements, activity requirements are always cast in the form “Does *x*, provided all preconditions are satisfied.” That way the activity requirement is still satisfied, even if the combined code decides to do nothing because of some condition being violated.

The combinable method for saving in sized file drawers, on the other hand, is an activity combinable method. It adds a new activity requirement to the operation, that the size must be updated; and describes combinable code where, provided the arguments belong to the expected classes, the combined code satisfies all the activity requirements of the combinee code (but not the checking requirements) plus the new activity requirement. There is also one more condition: that the combinee code doesn’t also do the activity; the activity shouldn’t be done more than once. In this case, the code updates the size data and does a `send-super`. Again, given this information, it is easy to infer that the combination of the size update code and the basic save code will be valid for saving in sized file drawers.

We might write a syntax for this as

```
activity method save:any, key:string, f:sizedFileDrawer)
  begin...send-super end
```

or more schematically as

```
activity method <op>(<v1>:<cl1>, ..., <vn>:<cln>)<body>
```

Now, we can again consider a class that multiply inherits from both `LockableFileDrawer` and `SizedFileDrawer`. It is now easy to infer that the checking method wrapped around the activity method wrapped around the basic method will meet all three requirements for saving in such file drawers. The other, incorrect, order of combination, where the size is updated before the lock is checked, does not satisfy the requirements, because activity methods don't preserve checking requirements. That is exactly the right reason; it is too late to do a check once you have already done some of the activity.

There isn't room to go into more examples in detail, but it is worth pausing here to note that just the five kinds of methods presented above cover many common method combination situations. Other kinds of requirement might be appropriate to describe other combination situations; that is a topic for further research. But just the distinction between checking vs. activity has more mileage than is obvious from the discussion so far.

A caching method, for example, does both a check (that the argument is not in the cache) and an activity (putting the answer in the cache). If it is described that way, it will be guaranteed to be done just before all other activity methods. This analysis also suggests the idea of splitting a caching method into two separate parts, to give more flexibility for method combination.

It is also possible to describe methods that override combination methods, and to describe combination methods that must wrap around a particular method, with no interposed method. The later can be used in conjunction with mixin classes when the user would like to specify the order of some

method combinations. Another kind of method that can be described is a Beta [10] style method, where a superior calls on an inferior; in this case, it is the superior method that takes on the responsibility of preserving requirements. The point is that since it is the methods, rather than the class hierarchy, that control the combination, many styles of combination can comfortably cohabitate, all under a single combination mechanism.

6 Conclusion

By separating the traditional method declaration into a mandate of requirements for an operation together with a description of code that meets some requirements, method combination becomes more tractable. Multiple inheritance just conjoins requirements, while the methods describe how code can be combined to meet requirements. Only requirements derivable from an object creation request need to be satisfied. This leads to a higher level approach to method combination.

References

- [1] ANDREOLI, J.-M., AND PARESCHI, R. Linear objects: logical processes with built-in inheritance. In *Proceedings of the Seventh International Conference on Logic Programming* (Cambridge, MA, 1990), D. H. D. Warren and P.Szeredi, Eds., MIT Press, pp. 495–510.
- [2] CHAMBERS, C., AND LEAVENS, G. T. Typechecking and modules for multi-methods. In *OOPSLA '94 Conference Proceedings Object-Oriented Programming Systems, Languages, and Applications* (October 1994), pp. 1–15.
- [3] DIXON, M. *Embedded Computation and the Semantics of Programs*. PhD thesis, Stanford University, 1991. Also published as Xerox PARC technical report SSL-91-1.
- [4] FOREMAN, I. R., DANFORTH, S., AND MADDURI, H. Composition of before/after metaclasses in SOM. In *OOPSLA '94 Conference Pro-*

- ceedings Object-Oriented Programming Systems, Languages, and Applications* (October 1994), pp. 427–439.
- [5] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.
 - [6] HARRISON, W., AND OSSHER, H. Subject-oriented programming (a critique of pure objects). In *OOPSLA '93 Conference Proceedings Object-Oriented Programming Systems, Languages, and Applications* (October 1993), A. Paepcke, Ed., ACM/SIGPLAN, ACM Press, pp. 411–428. Volume 28, Number 10.
 - [7] HODAS, J. S., AND MILLER, D. Representing objects in a logic programming language with scoping constructs. In *Proceedings of the Seventh International Conference on Logic Programming* (Cambridge, MA, 1990), D. H. D. Warren and P.Szeredi, Eds., MIT Press, pp. 511–526.
 - [8] LAMPING, J., AND ABADI, M. Methods as assertions. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (1994), Tokoro and Pareschi, Eds., vol. 821 of *LNCS*, Springer-Verlag, pp. 60–80.
 - [9] LIEBERHERR, K. J., SILVA-LEPE, I., AND XAIO, C. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM* 37, 5 (May 1994), 94–101.
 - [10] MADSEN, O. L., AND MOLLER-PEDERSEN, B. Basic principles of the BETA programming language. In *Object-Oriented Programming Systems*, G. Blair, D. Hutchinson, and D. Shepard, Eds. Pitman Publishing, 1989.
 - [11] MEYER, B. Applying “design by contract”. *IEEE Computer* 25, 10 (October 1992), 40–51.
 - [12] SHAPIRO, E., AND TAKEUCHI, A. Object oriented programming in concurrent prolog. *New Generation Computing* 1 (1983), 25–48.
 - [13] STEELE, G. L. *Common Lisp: The Language (second edition)*. Digital Press, 1990.

- [14] STROUSTRUP, B. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991.
- [15] WIRFS-BROCK, R., AND WILKERSON, B. Object oriented design: A responsibility-driven approach. In *OOPSLA '89 Conference Proceedings Object-Oriented Programming Systems, Languages, and Applications* (October 1989), pp. 71–75.

A Translation of the Fragment

This appendix lists each construct in the fragment, followed by its translation into first order logic.⁷ The relation $\text{specified}(r, o, x_1, \dots, x_n)$ is intended to capture that the requirement r is specified for operation o on arguments x_1, \dots, x_n . Similarly, the relation $\text{meets}(R, f, x_1, \dots, x_n)$ is intended to capture that the set of requirements R is met by code f if it is applied to arguments x_1, \dots, x_n . It is necessary to consider the entire set in order to prohibit extraneous activity; meeting a set of requirements entails not performing activities not in the set.

The method selection principle is that code f is applicable to operation o on arguments x_1, \dots, x_n if, for the set R of all requirements r for which it is possible to prove $\text{specified}(r, o, x_1, \dots, x_n)$, it is true that $\text{meets}(R, f, x_1, \dots, x_n)$.

In the following, r_{new} is a new requirement constant for each instantiation of the translation, not equal to any other requirement. f is an abbreviation for the function $\text{lambda}(\langle v1 \rangle, \dots, \langle vn \rangle). \langle body \rangle$ associated with the method body. And \vdash_m means derivable, assuming the information about specifications of requirements.⁸

```
declare <subclass> subclass of <class>
       $\forall x \in \langle subclass \rangle. x \in \langle class \rangle$ 
```

⁷While we use set notation for convenience, we are dealing only with finite, non-nested, sets, so there is no increase in power.

⁸The use of \vdash_m here is well founded, since it is used only to assert information about meeting, not about specifying.

method $\langle op \rangle (\langle v1 \rangle : \langle cl1 \rangle, \dots, \langle vn \rangle : \langle cln \rangle) \langle body \rangle$

activity(r_{new}) \wedge

$\forall x_1 \in \langle cl1 \rangle \dots \forall x_n \in \langle cln \rangle. \text{specified}(r_{new}, \langle op \rangle, x_1, \dots, x_n) \wedge$

$\forall x_1 \in \langle cl1 \rangle \dots \forall x_n \in \langle cln \rangle. \text{meets}(\{r_{new}\}, f, x_1, \dots, x_n)$

overriding method $\langle op \rangle (\langle v1 \rangle : \langle cl1 \rangle, \dots, \langle vn \rangle : \langle cln \rangle) \langle body \rangle$

activity(r_{new}) \wedge

$\forall x_1 \in \langle cl1 \rangle \dots \forall x_n \in \langle cln \rangle. \text{specified}(r_{new}, \langle op \rangle, x_1, \dots, x_n) \wedge$

$\forall x_1 \in \langle cl1 \rangle \dots \forall x_n \in \langle cln \rangle. \text{meets}(R, f, x_1, \dots, x_n)$

where $R = \{r \mid x_1 \in \langle cl1 \rangle \wedge \dots \wedge x_n \in \langle cln \rangle \vdash_m \text{specified}(r, \langle op \rangle, x_1, \dots, x_n)\}$

preferred method $\langle op \rangle (\langle v1 \rangle : \langle cl1 \rangle, \dots, \langle vn \rangle : \langle cln \rangle) \langle body \rangle$

$\forall x_1 \in \langle cl1 \rangle \dots \forall x_n \in \langle cln \rangle. \text{meets}(R, f, x_1, \dots, x_n)$

where $R = \{r \mid x_1 \in \langle cl1 \rangle \wedge \dots \wedge x_n \in \langle cln \rangle \vdash_m \text{specified}(r, \langle op \rangle, x_1, \dots, x_n)\}$

In the following, f is an abbreviation for the function

$\text{lambda}(\text{super}). \text{lambda}(\langle v1 \rangle, \dots, \langle vn \rangle). \langle body \rangle$

associated with the method body, and parameterized on **super**.

checking method $\langle op \rangle (\langle v1 \rangle : \langle cl1 \rangle, \dots, \langle vn \rangle : \langle cln \rangle) \langle body \rangle$

checking(r_{new}) \wedge

$\forall x_1 \in \langle cl1 \rangle \dots \forall x_n \in \langle cln \rangle. \text{specified}(r_{new}, \langle op \rangle, x_1, \dots, x_n) \wedge$

$\forall R. (\forall r \in R. r \neq r_{new} \wedge (\text{checking}(r) \vee \text{activity}(r)))$

$\rightarrow \forall x_1 \in \langle cl1 \rangle \dots \forall x_n \in \langle cln \rangle.$

$(\forall s. \text{meets}(R, s, x_1, \dots, x_n) \rightarrow \text{meets}(R \cup \{r_{new}\}, f(s), x_1, \dots, x_n))$

activity method $\langle op \rangle (\langle v1 \rangle : \langle cl1 \rangle, \dots, \langle vn \rangle : \langle cln \rangle) \langle body \rangle$

activity(r_{new}) \wedge

$\forall x_1 \in \langle cl1 \rangle \dots \forall x_n \in \langle cln \rangle. \text{specified}(r_{new}, \langle op \rangle, x_1, \dots, x_n) \wedge$

$\forall R. (\forall r \in R. r \neq r_{new} \wedge \text{activity}(r))$

$\rightarrow \forall x_1 \in \langle cl1 \rangle \dots \forall x_n \in \langle cln \rangle.$

$(\forall s. \text{meets}(R, s, x_1, \dots, x_n) \rightarrow \text{meets}(R \cup \{r_{new}\}, f(s), x_1, \dots, x_n))$