Propagating Class and Method Combination

Erik Ernst

DEVISE, Department of Computer Science
University of Aarhus, Denmark
eernst@daimi.au.dk

Abstract. This paper presents a mixin based class and method combination mechanism with block structure propagation. Traditionally, mixins can be composed to form new classes, possibly merging the implementations of methods (as in CLOS). In our approach, a class or method combination operation may cause any number of implicit combinations. For example, it is possible to specify separate aspects of a family of classes, and then combine several aspects into a full-fledged class family. The combination expressions would explicitly combine whole-family aspects, and by propagation implicitly combine the aspects for each member of the class family, and again by propagation implicitly compose each method from its aspects. As opposed to CLOS, this is type-checked statically; and as opposed to other systems for advanced class combination/merging/weaving, it is integrated directly in the language, ensuring a clear semantics and a seamless interaction with the type system. Moreover, the basic mechanism used in the combination, linearization, is formalized and generalized compared to previous presentions.

1 Introduction

In recent years the management of concerns involving multiple classes has been a very active area of research. Subject orientation [10], aspect oriented programming [13], and object collaborations [20] are all examples of such efforts. This paper presents a language integrated approach to the achievement of similar goals. A seamless integration into a statically typed general purpose programming language opens the possibility for type checking at the level of multi-class constructs, separate type-checking and compilation, and avoidance of the "impedance" problems associated with the use of several different mind-sets, languages, and tools.

The approach described in this paper is based on the use of general block structure (i.e., multi-level nesting, like inner classes in Java) to enable a natural expression of groups of mutually dependent classes, along with a very flexible inheritance and class/method combination mechanism which interacts with the block structure to support propagation of class combinations. The word *propagation* refers to the following feature: An expression, like `a&b` which denotes the combination of the classes or methods `a` and `b`, can imply a number of implicit class or method combinations. Since the explicit combination operation

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 67–91, 1999.
© Springer-Verlag Berlin Heidelberg 1999

causes the implicit ones, it is natural to think of the combination process as starting with an explicit operation and propagating to implicit ones. The benefit is similar to that of all other abstraction mechanisms—a complex but regular result is achieved from a simple expression, ensuring readability, consistency, maintainability, and so on.

The contributions of this work are the design and implementation of the propagating class and method combination mechanism and its static analysis, and a clarification and generalization of the formal foundations of inheritance graph linearization. Linearization is the low-level mechanism on which the combination mechanism builds; it does not support propagation itself, but linearization together with block structure and virtual (class and method) attributes supports it. Virtual methods are well-known; virtual classes [18, 24, 11] are attributes which may vary with the enclosing class, e.g., if `NodeType` is a virtual class attribute of the class `Graph`, then `Graph.NodeType` could be the class `Node` and `ColoredGraph.NodeType` could be the class `ColoredNode`. Virtual classes are similar to, but not equivalent with, type parameters of parameterized classes (such as template classes in C++ or generic classes in Eiffel). See [5] for examples, but also note [25].

The result of implementing this is the language `gbeta`, which is a generalization of the language BETA. BETA already offers the combination of general block structure, virtual classes, and a kind of method combination—the `INNER` mechanism. `INNER` is similar to resending a message to '`super`' in Smalltalk in that it calls the implementation of a method in another class, but `INNER` calls to the subclass (if any) where `super` calls to the superclass. Consequently, all methods in BETA (and `gbeta`) are invoked at the most *general* level, and `INNER` determines where to *insert* the next more specialized method implementation.

`gbeta` generalizes BETA in several ways in order to support the propagation mechanism. The subclass relation between classes in BETA is based on name equivalence; in `gbeta` it is based on a coarse grained, mixin based structural equivalence which has the BETA relation as a subset. BETA has single inheritance, but both for classes and methods [15]; in `gbeta` classes and methods can be combined, again with BETA semantics as a special case. In `gbeta` it is also possible to inherit from a virtual class, and to do several other things not supported in BETA, but these features are not essential for the topic of this paper.

The approach may seem tied to a particular language, but that is only because so few object-oriented languages offer general block structure and virtual classes. With the introduction of inner classes in Java and the possible addition of virtual types [24] or a similar mechanism, the same techniques would apply here.

Since CLOS [12] programmers introduced "mixin" classes as a particular programming style in context of linearization based multiple inheritance, a related but separate concept of *mixins* and mixin-based inheritance has emerged [2, 3, 23, 8]. Inheritance allows for the creation of one new class, based on zero or more superclasses and a specification of an increment (often with syntax like `{..}` containing a list of declarations). Mixins liberate the increment such that

it can be applied to different superclasses. The semantic denotation of an increment may be a function from classes to classes [8], a class-like entity which can be composed with others using special mixin combination operators [2, 3], or even a method which enhances the structure of the enclosing object [23]. In any case, the application of a mixin to an actual superclass resembles inheritance. This motivates the alternative term for mixins: *abstract subclasses*.

In a statically typed language the not-yet-known superclass of a mixin must be characterized somehow before the usage of inherited attributes in the mixin can be type-checked. In [8]—which deals with a subset of Java, enhanced with mixin support—this is achieved by requiring that mixins specify an *inheritance interface*. This is an interface which is assumed about the formal superclass during checking of the mixin, and required of the actual superclass at mixin application. In gbeta, the inheritance interface is the statically known prefix (superclass or supermethod) of the mixin. The mixin is type checked on the assumption that the prefix is available, and the run-time semantics ensure that the actual prefix always is the statically known one or a descendant of it.

In summary, the essense of typed mixins is the support for standalone, class- and method-like entities which can be applied as abstract subclasses/methods to several different actual superclasses/methods, such that the interaction between the two is precisely specified. This description matches the mechanism in gbeta, hence the use of the word 'mixin'. However, since mixins in gbeta are managed and composed not individually but in lists and using linearization, CLOS might be a closer reference point than *Jigsaw* [3], Agora [23] or MixedJava [8].

The remainder of this paper is organized as follows: Section 2 describes the usage of the propagating combination mechanism and argues for its usefulness via a number of examples. Section 3 gives a precise characterization of the linearization algorithm and the propagation mechanism. The current implementation is briefly described in Sect. 4. Related work is covered in Sect. 5, future work in Sect. 6, and Sect. 7 concludes.

2 Usage

This section gives a survey of significant usages of the propagating combination mechanism, thus illustrating its semantics and motivating its usefulness.

The concrete syntax used throughout is a modification of gbeta syntax (which is Beta syntax enhanced with a few new constructs); the differences are that (1) certain special characters which indicate kinds of declarations have been replaced with keywords, and (2) the "`->`" which unifies the syntax for assignment, method argument transfer, and function return has been transformed to a more mainstream notation. Specifically, assignment is designated with "`:=`" with the source on the right hand side and the destination on the left hand side; formal arguments to methods are declared outside the method body, and actual arguments are given in parentheses after the method name. This makes the ex-

amples more verbose but hopefully improves the readability for those who are not accustomed to the BETA style of syntax.[1]

Note that method signatures (argument types and names, and return types) are inherited (not repeated) in BETA/gbeta.

2.1 Class Combination, Propagating to Methods

The first example illustrates the use of propagation in only one level; this special case works similarly to CLOS method combination using '`before`' and '`after`' methods, illustrating the combination mechanism by showing how it achieves a recognizable goal.

Consider an abstract class `Stack` which specifies a stack data structure, along with a subclass `StackImpl` which contributes an implementation of the stack using a list (whose type constraint on contained objects, `Element`, is specified to be the same as the constraint given for the enclosing `StackImpl`):

```
class Stack: [1](#
  virtual class Element: Object;
  virtual method init: [2](# do INNER #);
  virtual method push: [3](elm: Element)(# do INNER #);
  virtual method pop: [4](# do INNER #): Element
#);

class StackImpl: Stack [5](#
  extended method init: [6](# do storage.init #);
  extended method push: [7](# do storage.insert(elm) #);
  extended method pop: [8](# return storage.deleteFirst #);
  object storage: list
    [9](# extended class Element: this(StackImpl).Element #)
#)
```

For brevity, this example uses minimal error handling, e.g., `pop` on an empty `StackImpl` fails because `deleteFirst` fails.

In `gbeta` (and BETA) a virtual method cannot be overridden, but it can be *extended*. For example, `init` in `StackImpl` is [2]`(# do INNER #)` extended with [6]`(# do storage.init #)`. The keyword `do` marks the beginning of the statements of the method. The small superscript numbers (2 and 6 in this case) are not part of the syntax; they are added to the syntax here in order to identify declaration blocks (including argument list and return type, if present). As Fig. 6 in Sect. 3.2 will show, a mixin in `gbeta` is a pair of a declaration block and an environment, but for simplicity we will ignore the environment at this point. The notation for a mixin is m_i where i is the identification number. This means that `init` in `StackImpl` can be concisely specified as the mixin list $[m_2, m_6]$.

As mentioned in Sect. 1, `INNER` is a call to the next more specific contribution to a method, so the effect of `init` in `StackImpl` is just to invoke `storage.init`.

[1] The examples in original `gbeta` syntax are available at `http://www.daimi.au.dk/~eernst/gbeta/examples/PropComb/`

The method body `do INNER` is the BETA/gbeta idiom for a deferred implementation, since it simply calls the subclass extensions of the method.

Returning to the concrete example, the `StackImpl` class can be used directly, but it does not protect itself from shared access in a multi-threaded context. To add concurrency control we write another subclass of `Stack`:

```
class StackConc: Stack [10](#
  extended method init: [11](# do mutex.init; INNER #);
  extended method push: protect;
  extended method pop: protect;
  method protect: [12](# do mutex.P; INNER; mutex.V #);
  object mutex: Semaphore;
#)
```

In `StackConc`, the virtual methods `push` and `pop` are extended with `protect`. As explained below, this equips `push` and `pop` with concurrency control.

We now have two aspects, `StackConc` and `StackImpl`. They can be combined using the combination operator '`&`':

```
class ThreadSafeStack:  StackConc & StackImpl;
```

The resulting class, `ThreadSafeStack`, is actually a thread safe stack, because the combination of the two classes propagates to the methods such that each method contains contributions from both `StackConc` and `StackImpl`.

Figure 1 shows the four classes. Each box is a mixin, so each class is a list of mixins (all classes and methods are). Moreover, the combination operator '`&`' is defined in terms of mixin lists, as described in Sect. 3.1. The subclass relation is the superlist relation, so we have a subclass ordering as specified below the mixin lists in the figure.

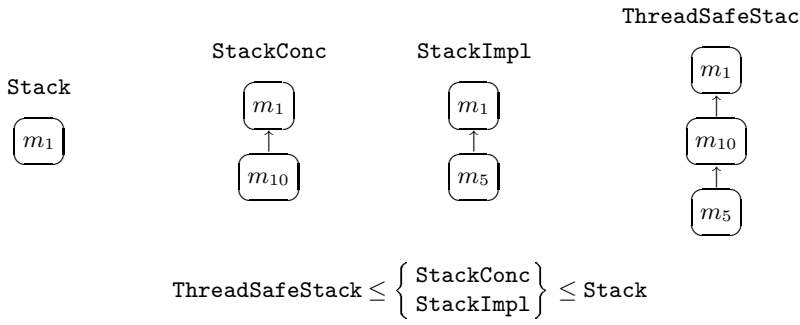

Fig. 1. The `Stack` classes, and their subclass relations

In Fig. 2 the contributions to `push` in `ThreadSafeStack` are listed. The 'combined result' in Fig. 2 is similar to `push` in that it has the same signature and will

execute the same statements in the same order, so it presents an overview of the semantics of `push`. As desired, the implementation (in m_7) is enclosed between the acquisition and the relinquishment of the `semaphore` (the `P` and `V` operations in m_{12}). Consequently, at most one thread can execute the m_7 part of `push` (or similarly for `pop`) on a `ThreadSafeStack` at the same time, i.e., `storage` has been put under concurrency control.

```
From m1: 3(elm: Element)(# do INNER #)
From m10: 12(# do mutex.P; INNER; mutex.V #)
From m5: 7(# do storage.insert(elm) #)
Combined result:
  (elm: Element)(#
  do mutex.P; storage.insert(elm); mutex.V
  #)
```

Fig. 2. The contributions to `push` in `ThreadSafeStack`

For the explanation of how `push` was determined to be $[m_3, m_{12}, m_7]$, we must consider the enclosing class, `ThreadSafeStack`. Note that it is not necessary, or even relevant, to know if or how the enclosing entity was constructed by combining other entities.

A virtual method or class is computed by combining the contributions to it, from the most general mixin of the enclosing class or method, and towards more specific ones. In this example the enclosing class is `ThreadSafeStack`, or $[m_1, m_{10}, m_5]$, so `push` is computed as follows:

$$[m_3]\&[m_{12}]\&[m_7] = [m_3, m_{12}, m_7]$$

The contributions to a virtual need not be singleton lists in general, but for the rather common case that they are singletons, we can simply collect them in order from the enclosing entity.

Note that we could have combined the concurrency control with another implementation, and the implementation with zero or more auxiliary aspects such as concurrency control. In general, we can build classes from aspects such that the implementation of methods contains contributions from any or all of those aspects.

In CLOS, a similar result could have been obtained (not in a type safe manner, though) by putting statements before `INNER` into a '`before`' method and statements after `INNER` into an '`after`' method, and letting `ThreadSafeStack` inherit from `StackConc` and `StackImpl`, in that order. This illustrates that propagation in one level from classes to enclosed methods amounts to a similar mechanism as CLOS standard method combination. The relation to BETA method combination is obvious (both BETA and gbeta use `INNER` for this), but there is much more freedom to build mixin lists in gbeta. In fact, the list of mixins in

any class or method in BETA is fully determined when the most specific mixin is known. In other words, mixins in BETA are *not* liberated from the superclass—so they are not really mixins, but just ordinary class increments. That is the most fundamental difference between BETA and gbeta.

2.2 Combination of Methods, Propagating to Classes

In this section we consider an example where the combination operator '&' is applied to a group of classes used in a method, and by propagation combines aspects of each of the members of the group. Assume that we have defined some auxiliary classes supporting payments and delivery of things:

```
class Person: [13](# object name: String #);
class Payer: Person(# method pay: (amnt:Integer)(#..#):Integer#);
class Paid: Person[14](# method accept: (amnt:Integer)(# .. #)#);
class Receiver: Person(# method receive: (t:Thing)(# .. #)#);
class Deliverer: Person[15](# method deliver: (# .. #):Thing #);
```

A Person has a name; a Payer can pay an amount of money, and a Paid person can accept amounts of money. Moreover, a Receiver can receive a Thing and a Deliverer can deliver a Thing.

Collaborations for these classes arise naturally. An example could be the activity "to pay" (the argument list of pay is inherited from collaboration):

```
method collaboration: (fst:First, snd:Second) (#
   virtual class First: Person;
   virtual class Second: Person;
do INNER
#);
method pay: collaboration(#
   extended class First: Payer;
   extended class Second: Paid;
   virtual method price: (# do INNER #): Integer;
do snd.accept(fst.pay(price));
   INNER
#)
```

In collaboration it may be confusing that the types of the arguments are the virtual classes First and Second declared inside collaboration, but that is actually the case. The problem only arises here because of the adjustments of the syntax to a more mainstream style; in BETA/gbeta the method arguments are declared inside the block, so the usage of other attributes inside the block is quite natural.

The collaboration method introduces two *roles*, played by fst and snd, and specified by First and Second. The pay method enhances the collaboration method by extending the role classes First and Second, and by adding one statement to the behavior in which snd accepts the payment from fst, as specified by price.

We can create a similar activity for the transfer of possession of some item, where the `snd` role player delivers a `Thing` which is then `received` by the `fst` role player:

```
method deliver: collaboration(#
   extended class First: Receiver;
   extended class Second: Deliverer;
do fst.receive(snd.deliver);
   INNER
#)
```

With these activities in place we can create a combination method which supports the combination of the activities—both transferring an amount of money and transferring an entity in exchange for the money:

```
(# method doTrade: pay & deliver;
   object diamond: Thing;
   object walrus: Paid & Receiver & Deliverer;
   object lucy: Payer & Receiver;
do
   walrus.receive(diamond);
   doTrade(lucy,walrus)
#)
```

In this piece of code we create the above mentioned combined method `doTrade`, thus by propagation combining the nested virtual classes `First` and `Second` from the contributions in `pay` and `deliver`. For example, `Second` in `doTrade` is `Paid&Deliverer`, i.e.:

$$[m_{13}, m_{14}] \& [m_{13}, m_{15}] = [m_{13}, m_{14}, m_{15}]$$

This ensures that an object which is to play the `snd` role has both an `accept` and a `deliver` method.

The behavior of `pay` and `deliver` is combined, due to the `INNER` mechanism, such that both the transfer of money and the transfer of things will occur. The object `diamond` is created, so we have something to transfer. Two role player objects, `walrus` and `lucy`, are created with the necessary mixins. Since the `walrus` must first `receive` the `diamond` in order to be able to `deliver` it to `lucy`, there is both a `Deliverer` and a `Receiver` aspect of `walrus`; `lucy` could have been a `Deliverer`, too, but she probably won't.

2.3 Growing a Family of Classes

The last example seems to be almost compulsory in papers about advanced languages and type systems recently [5, 24], but in this case we emphasize the possibility of distributing the implementation over several levels of specialization, in order to deal with various concerns as "soon" as possible—that is, at the most general level where the necessary information is available. This is an example

of combining classes which contain other classes which again contain methods, propagating the combination operations as usual.

Figure 3 specifies a class `ObserverDesignPattern` which can be used to support the Observer design pattern [9]. It contains two nested mutually recursive classes `Subject` and `Observer`. An instance of `Observer` may `attach` to an instance of `Subject`. Once inserted into the `Set` of `observers` for that `Subject` it will be a target for notifications. The `scan` method on `Set` is the standard device in BETA/gbeta for iteration over all elements in a collection.

`scan` contains a declaration of a reference `current`; it executes `INNER` once for each element in the collection, with `current` referring to the current element. Hence, `notify` invokes `update` on each member of `observers` with `this(Subject)` (the enclosing instance of `Subject`) as an argument. The observer may then update its own state according to the changes in the given subject.

```
class ObserverDesignPattern: (#
   virtual class Subject: (#
      method attach: (o:Observer)(# do observers.insert(o) #);
      method detach: (o:Observer)(# do observers.delete(o) #);
      method notify: observers.scan
         (# do current.update(this(Subject)) #);
      object observers: Set
         (# extended class Element: Observer #)
   #);
   virtual class Observer: (#
      virtual method update: (s:Subject)(# do INNER #)
   #)
#)
```

Fig. 3. Support for the Observer design pattern

Each (significant) change in the state of the `Subject` should be followed by an invocation of `notify` (it is a programmer responsibility to remember to invoke `notify` at the right places).

To use this we need a couple of auxiliary classes, for instance a `TextBuffer` to be observed by a `Window` which could be an instance of the subclass `ColorIcon`:

```
class TextBuffer: (#
   object name: String;
   virtual method setFileName: (n:String)(# do name := n #);
   virtual method getFileName: (# do return name #): String;
#);
class Window: (# method refresh: (# .. #); .. #);
class ColorIcon: Window(#
   method setIconTitle: (s:String)(# .. #);
#)
```

Now we can create a subclass of the `ObserverDesignPattern` which extends the virtual classes and thereby lets `Windows` observe a `TextBuffer`:

```
class WindowAndTextODP: ObserverDesignPattern(#
   extended class Subject: TextBuffer(#
      (* ensure that 'notify' is called after changes *)
      extended method setFileName: (# do INNER; notify #)
   #);
   extended class Observer: Window(#
      extended method update: (#~do~getState(s);refresh~#);
      virtual method getState: (s:Subject)(# do INNER #)
   #)
#)
```

We could now use the class `WindowAndTextODP` as a class family aspect, combining it with some other class family aspects that contribute something else to `Subject` and `Observer`; this would be an example of a two-level propagation: From combined class families to member classes to methods of each member class. But for brevity we will use `WindowAndTextODP` directly in the following.

At the level of `WindowAndTextODP` we can do part of the notification work: When an `Observer` learns that the `Subject` has changed (i.e., when `notify` invokes `update` with that observer as `current`) then we can get the state and `refresh` the `Window`. We do not yet know *how* to get the state, but that's a deferred virtual method so we can put it in later. Finally we can create an instance of the design pattern, `myODP`, and populate it with a subject `myBuffer` and an observer `myIcon`:

```
object myODP: WindowAndTextODP;
object myBuffer: myODP.Subject;

object myIcon: ColorIcon & myODP.Observer (#
   extended method getState:
      (# do setIconTitle(s.getFileName)#)
#)
```

The class of `myIcon` has two super-classes, `ColorIcon` and `myODP.Observer`. The first is a standard GUI support class, and the second contributes the design pattern related aspect, and the block provides the implementation of `getState`. This implementation depends on the following information from the static analysis:

- `s` is (a descendent of) a `TextBuffer` because `myODP` is a `WindowAndTextODP` which declares `extended class Subject: TextBuffer(#..#)`, so it has a `getFileName` method which takes no arguments and delivers a `String`
- `myIcon` is a `ColorIcon`, so it has a `setIconTitle` method which takes a `String` argument

This could not be type checked if `s` in the body of `getState` only had the the type declared in the original `ObserverDesignPattern`. However, in both BETA

and gbeta, a virtual class/method attribute *is* recognized by the type system as denoting a subclass/submethod when looked up in context of a more specialized enclosing class or method.

Compared to Beta, this example uses the support for extending virtual classes with *unrelated* classes—in Beta a virtual class must always be extended with a descendant of the value of that virtual class in the superclass (Beta has single inheritance). This means that in Beta we cannot use classes like `Window` and `TextBuffer` which have been written independently of the design pattern classes, only classes created especially for use in `ObserverDesignPattern` can be used.

With the approach in [5] and all other approaches with virtual *types* (or interfaces) as opposed to virtual *classes*, there is no support for creating instances of virtuals, or for putting state or implementation into virtuals, such as the `observers` and `notify` in `Subject`. As it is known from Java, this property can lead to duplication of code or manual delegation, to compensate for the fact that state and implementation cannot be inherited from interfaces.

By means of a mechanism called code *weaving*, AspectJ [21] is capable of adding statements to methods, and more. As the Subject/Observer example[2] shows, this approach can do similar things as gbeta, in some respects even more flexibly. There are of course many differences in the details, but one profound difference is that the types of fields and methods cannot be modified by aspects in AspectJ; for instance, there is a method `getData` in the example which has a return type of `Object`, and an dynamic cast is needed when that method is used. The gbeta example above is slightly different, but a method `getData` could easily be defined, returning a `Subject`, and that method would actually have a return type which is affected by the aspects being used. This means that no dynamic casts are required in gbeta, avoiding the potential run-time type error. Another example of this phenomenon was the `doTrade` method, where the types of the formal arguments were constructed by propagated combination. In short, AspectJ handles aspects of implementation, but not aspects of types.

3 Linearization and Propagation

In the previous section the usage of the complete language was in focus, with combination and propagation intertwined, and by example. This section presents the basic mechanisms separately and in more detail. The basic, propagationless mechanism behind class and method combination is a linearization algorithm, and propagation emerges with the semantics of virtuals. The linearization is presented and formally defined in Sect. 3.1, and the propagation mechanism is presented in Sect. 3.2 via a small functional language whose semantics is the core of gbeta.

[2] Available from [21], via 'AspectJ Primer'

3.1 Linearization

The class and method combination mechanism in gbeta builds on a simple graph linearization algorithm which is presented in this section. The linearization is characterized as conceptually wholesome (previously [1] it has been described as a 'hybrid'); it is specified precisely what the linearization is; the linearization is generalized to handle cases which would otherwise be rejected; and some results about its properties are proved.

A graph linearization is an algorithm which constructs a list from a given directed graph, such that the list is a topological sorting of the nodes. Obviously a cyclic graph does not allow this, hence some graphs cannot be linearized. Since there are many possibilities for typical graphs, some systematic choices must be made in order to arrive at a well-defined result. Existing linearizations [6, 7, 1] are described in terms of such systematic choices of "what node to take next", and this makes it hard to understand their outcome, and to reason about their properties.

Luckily, the linearization which is used in gbeta, 'C3' [1], can be characterized in a much more declarative way, and it can even be generalized in a way that makes it a proper operation on a suitable set M:

$$\forall x, y \in M.\ x \,\&\, y \in M$$

The name C3 reflects three consistencies exhibited by this linearization, namely consistency with the local precedence order,[3] consistency with the extended precedence graph,[4] and monotonicity.[5] The other known linearizations (including the ones used in LOOPS, CLOS, and Dylan) do not have all three consistencies. The remaining problem (to be solved below) is that *none* of the linearizations can handle all inheritance hierarchies, some are rejected as inconsistent.

We will present the concrete algorithm first, and then proceed with the declarative characterization and the generalization. The concrete algorithm takes two (argument) lists and merges them into one (result) list by repeatedly choosing one element from the argument lists and transferring it to the result list, until all elements have been transferred. Figure 4 shows the rule for choosing and transferring one element. The linearization is then defined in terms of sequences of such steps:

$$A \,\&\, B = R \quad \text{iff} \quad (A, B, []) \ \rhd^* \ ([], [], R)$$

The process requires and preserves the property that the elements in each of the lists are distinct. As an example, we can demonstrate that $[a, b, c] \,\&\, [x, b] = [a, x, b, c]$ by the following process:

$$\begin{aligned}([a, b, c], [x, b], []) \ &\rhd \ ([a, b], [x, b], [c]) \ \rhd \ ([a], [x], [b, c]) \\ &\rhd \ ([a], [], [x, b, c]) \ \rhd \ ([], [], [a, x, b, c])\end{aligned}$$

[3] The programmer-chosen ordering of direct superclasses.

[4] Which additionally orders classes according to the local precedence order from the most general common subclass.

[5] Avoidance of the phenomenon that an inherited feature is looked up in a class that none of the direct superclasses would have chosen.

$$([a_n \dots a_1], [b_m \dots b_1], [r_k \dots r_1]) \rhd ([a_n \dots a_1], [b_m \dots b_2], [b_1, r_k \dots r_1]),$$
$$\text{if } b_1 \notin \{a_1 \dots a_n\}$$

$$([a_n \dots a_1], [b_m \dots b_1], [r_k \dots r_1]) \rhd ([a_n \dots a_2], [b_m \dots b_1], [a_1, r_k \dots r_1]),$$
$$\text{if } b_1 \in \{a_1 \dots a_n\} \wedge a_1 \notin \{b_1 \dots b_m\}$$

$$([a_n \dots a_1], [b_m \dots b_1], [r_k \dots r_1]) \rhd ([a_n \dots a_2], [b_m \dots b_2], [a_1, r_k \dots r_1]),$$
$$\text{if } a_1 = b_1$$

Fig. 4. Defining ' $\rhd$ ': How to take one step in a C3 linearization

The process may fail, as with $([a, b], [b, a], [])$ where no step can be taken and the configuration is not on the final form $([][], R)$. This is a problem because it means that some classes or methods cannot be combined; gbeta generates a 'bad merge' error message and rejects the program. As we shall see below, this problem can be overcome. It is, however, still future work to implement the generalized version of C3 in gbeta.

To reach a declarative characterization we must make a shift in mindset and terminology. If we regard the edges in a given acyclic oriented graph as a relation and take the reflexive and transitive closure of that, we get a partial order. Similarly, a list determines a total order. Hence, a linearization is a construction of a total order by adding elements to a partial order. Note that '$m_i < m_j$' in this context means 'm_i is closer to the rightmost end of a list of mixins than m_j'. This ordering is an entirely different concept than subclass ordering, which is concerned with the comparison of mixin lists as a whole.

C3 actually constructs a total order from a number of given total orders, namely the linearizations of the superclass hierarchies. The C3 principle can now be given for two order relations:

The C3 principle:
The linearization of two orders A and B, $A \,\&\, B$, is
the union of A and B together with
all non-contradictory edges from B to A

In other words, $A \,\&\, B$ just adds a default rule to A and B, namely that elements from B by default are smaller than elements from A. This is formalized straightforwardly below, but first we will have to establish a few facts.

Total preorders. We need to consider total preorders:

Definition 1. *A* total preorder *is a relation which is reflexive, transitive, and total. A* total order *is a total preorder which is also anti-symmetric.*

It is easy to prove that:

Lemma 1. *Assume $\preceq$ is a total preorder. The relation $\sim$ defined by $a \sim b \Leftrightarrow a \preceq b \wedge b \preceq a$ is an equivalence relation, and the relation $\leq$ on equivalence classes defined by $a^\sim \leq b^\sim$ iff $a \preceq b$ is well-defined and a total order.*[6]

Conversely, given an equivalence relation $\sim$ and a total order on the equivalence classes $\leq$, then the relation $\preceq$ defined by $a \preceq b \Leftrightarrow a^\sim \leq b^\sim$ is a total preorder.

In other words, a total preorder corresponds to a list of equivalence classes of elements, rather than a list of individual elements.

This is the desired generalization: to construct a *list of groups* of mixins, each group consisting of mixins considered equally specific.

In such a setting, clashing names are not always disambiguated. This might at first seem to be a step backwards; it is in fact an improvement. When the ordinary C3 linearization would succeed, the generalization delivers the same result (all groups have size one). When the hierarchy would be rejected by ordinary C3, the resulting non-trivial groups from generalized C3 would in many cases work quite well. For example, as long as a name is only declared in one of the mixins in a given group, there will be no clashes on that name. In fact, a number of inheritance hierarchies would be *better* described by making certain mixins equally specific, since the commitment to one order causes unnecessary restrictions on future usage.

Formalization of C3. We need a couple of tools before C3 can be formalized:

Definition 2. *When R is a relation, its* domain *is* $\mathrm{dom}(R) \triangleq \{y | (\exists z.\ (y,z) \in R) \vee (\exists x.\ (x,y) \in R)\}$*, its* inversion *is* $\overline{R} \triangleq \{(y,x) | (x,y) \in R\}$*, its* one-step transitive closure *is* $R^{+1} \triangleq R \cup \{(x,z) | \exists y.(x,y),(y,z) \in R\}$*, and its* transitive closure *is* $R^* \triangleq \bigcup_{i \in \omega} R_i$*, where* $R_0 \triangleq R$*,* $\forall i \in \omega.\ R_{i+1} \triangleq R_i^{+1}$*.*

The following lemma is immediate from the definitions:

Lemma 2. *Let R and S be relations. Then R^* is transitive. The domain is additive:* $\mathrm{dom}(R \cup S) = \mathrm{dom}(R) \cup \mathrm{dom}(S)$*. The domain is preserved by transitive closure and inversion:* $\mathrm{dom}(R^*) = \mathrm{dom}(\overline{R}) = \mathrm{dom}(R)$*. Reflexivity is preserved by transitive closure, inversion, and union: if* $\forall x \in \mathrm{dom}(R).\ x \preceq_R x$ *then* $\forall x \in \mathrm{dom}(S).\ x \preceq_S x$*,* $S \in \{R^*, \overline{R}\}$*, and if* $\forall x \in \mathrm{dom}(T).\ x \preceq_T x$*,* $T \in \{R, S\}$ *then* $\forall x \in \mathrm{dom}(R \cup S).\ x \preceq_{R \cup S} x$*.*

[6] $a^\sim$ denotes the equivalence class wrt. $\sim$ containing a

The formalization of C3 is:

Definition 3 (C3 Linearization). *Let R_1 and R_2 be relations. The* C3 linearization *of R_1 and R_2 is $R_1 \,\&\, R_2 \triangleq R \cup (\mathrm{dom}(R_2) \times \mathrm{dom}(R_1) \setminus \overline{R})$, where $R \triangleq (R_1 \cup R_2)^*$.*

Intuitively, the linearization combines the two given relations R_1 and R_2 into $(R_1 \cup R_2)$ which is then "repaired" to be a transitive relation R by taking the transitive closure. R is complemented with everything from $\mathrm{dom}(R_2) \times \mathrm{dom}(R_1)$ which does not contradict R. In other words, R_2 elements are smaller than R_1 elements, unless something is known to the contrary.

Now we can state the closure property that makes total preorders interesting:

Proposition 1. *Assume R_1 and R_2 are total preorders. Then $R_1 \,\&\, R_2$ is a total preorder.*

Proof. Let $R \triangleq (R_1 \cup R_2)^*$ as in the definition, and let $S \triangleq \mathrm{dom}(R_2) \times \mathrm{dom}(R_1) \setminus \overline{R}$. Observe that R and $\overline{R}$ are reflexive because union, transitive closure, and inversion preserve reflexivity. Moreover, since $\mathrm{dom}(S) \subseteq \mathrm{dom}(R_1) \cup \mathrm{dom}(R_2) = \mathrm{dom}(R)$, also $R_1 \,\&\, R_2$ is reflexive—"S does not touch any new elements compared to R."

For transitivity, note that $R_1 \,\&\, R_2 = R \cup S$, and $(x, y) \in S \Rightarrow (y, x) \notin R$. Assume $(x, y), (y, z) \in R_1 \,\&\, R_2$. We show that $(x, z) \in R_1 \,\&\, R_2$:

- If $(x, y), (y, z) \in R$, then $(x, z) \in R \subseteq R_1 \,\&\, R_2$ because R is transitive.
- If $(x, y) \in R$ and $(y, z) \in S$ then $(z, y) \notin R$ by definition of S. If $(z, x) \in R$ then by transitivity of R we get $(z, y) \in R$, contradiction, hence $(z, x) \notin R$. Observe that $y \in \mathrm{dom}(R_2)$ and $z \in \mathrm{dom}(R_1)$ because $(y, z) \in S$. If $x \in \mathrm{dom}(R_2)$ then $(x, z) \in \mathrm{dom}(R_2) \times \mathrm{dom}(R_1) \setminus \overline{R} = S \subseteq R_1 \,\&\, R_2$. Otherwise $x \in \mathrm{dom}(R_1)$, but then $(z, x) \in \mathrm{dom}(R_1)^2$, and by totality of R_1, $(x, z) \in R_1 \vee (z, x) \in R_1$. Since $(z, x) \in R_1$ contradicts $(z, x) \notin R$ we must have $(x, z) \in R_1 \subseteq R_1 \,\&\, R_2$.
- The case $(x, y) \in S$ and $(y, z) \in R$ is similar.
- If $(x, y), (y, z) \in S$ then $x \in \mathrm{dom}(R_2)$, $y \in \mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)$, and $z \in \mathrm{dom}(R_1)$. Moreover $(y, x), (z, y) \notin R$, by definition of S. Then $(x, y) \in R_2$ by totality of R_2, and $(y, z) \in R_1$ by totality of R_1, hence $(x, y), (y, z) \in R$ and by transitivity of R finally $(x, z) \in R \subseteq R_1 \,\&\, R_2$.

For totality of $R_1 \,\&\, R_2$ we choose arbitrary $x, y \in \mathrm{dom}(R_1 \,\&\, R_2)$, and show that either $(x, y) \in R_1 \,\&\, R_2$ or $(y, x) \in R_1 \,\&\, R_2$:

- If $x, y \in \mathrm{dom}(R_1)$ then $(x, y) \in R_1 \vee (y, x) \in R_1$ by totality of R_1.
- If $x \in \mathrm{dom}(R_1)$ and $y \in \mathrm{dom}(R_2)$ then either $(y, x) \in R$ or $(x, y) \in S$.
- The remaining two cases are similar.

This proves that $R_1 \,\&\, R_2$ is reflexive, transitive, and total, i.e. it is a total preorder. □

The ordinary C3 fails precisely when the generalized C3 produces a total preorder which is not a total order. A total preorder is a total order if and only if there are no cycles, so we need to consider them:

Definition 4. *Let R be a relation. A sequence of distinct elements $d_1 \ldots d_n \in \mathrm{dom}(R)$, $n \geq 2$, is a* cycle *in R iff $(\forall i \in 1 \ldots n{-}1.\ (d_i, d_{i+1}) \in R) \wedge (d_n, d_1) \in R$. R is* acyclic *iff there are no cycles in R.*

Lemma 3. *Let R be a reflexive, acyclic relation. Then R^* is reflexive and acyclic.*

Proof. Given a reflexive, acyclic relation R. With $R_0 \triangleq R$, $\forall i \in \omega.\ R_{i+1} \triangleq R_i^{+1}$ we have $R^* = \bigcup_{i\in\omega} R_i$. Assume that R^* has a cycle and let $k \in \omega$ be the least number such that R_k has a cycle, say $d_1 \ldots d_n$; then $k > 0$ because R is acyclic. Since

$$\{(d_i, d_{i+1}) | i \in 1 \ldots n{-}1\} \cup \{(d_n, d_1)\} \subseteq R_k$$

and $R_k = R_{k-1}^{+1}$ we can choose $c_1 \ldots c_n \in \mathrm{dom}(R)$ such that

$$\begin{array}{l} \quad \forall i \in \{1 \ldots n{-}1\}.\ (\{(d_i, c_i), (c_i, d_{i+1})\} \subseteq R_{k-1}) \vee ((d_i, d_{i+1}) \in R_{k-1})) \\ \wedge \\ \quad (\{(d_n, c_n), (c_n, d_1)\} \subseteq R_{k-1}) \vee ((d_n, d_1) \in R_{k-1}) \end{array}$$

which provides us with a cycle in R_{k-1}, contradicting the minimality of k. □

We can now state and prove the "non-pre" equivalent of proposition 1:

Proposition 2. *Assume R_1 and R_2 are total orders and $R_1 \cup R_2$ does not have cycles. Then $R_1 \,\&\, R_2$ is a total order.*

Proof. Since R_1 and R_2 are total preorders we get reflexivity, transitivity, and totality directly from proposition 1. Only anti-symmetri remains to be proved. Assume that $(x, y), (y, x) \in R_1 \,\&\, R_2$; we must prove that $x = y$: Let $R \triangleq (R_1 \cup R_2)^*$ and $S \triangleq \mathrm{dom}(R_2) \times \mathrm{dom}(R_1) \setminus R$ such that $R_1 \,\&\, R_2 = R \cup S$ and $(x, y) \in S \Rightarrow (y, x) \notin R$.

- If $(x, y), (y, x) \in R$ then $x = y$ since R is acyclic, by lemma 3.
- Both $(x, y) \in R \wedge (y, x) \in S$ and $(y, x) \in R \wedge (x, y) \in S$ are impossible by definition of S.
- Similarly, if $(x, y), (y, x) \in S$ then $(y, x), (x, y) \notin R$. This is a contradiction since $x, y \in \mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)$ and in particular by totality of R_1, $(x, y) \in R_1 \vee (y, x) \in R_1 \subseteq R$.

□

We have seen that C3 can be formalized in a rather obvious manner and proved that the formalization has the nice stability property of proposition 1 and the incomplete stability property of proposition 2. It seems to be worthwhile to try to develop the strict linearization of various languages into the more wholesome total preorder model, going from class precedence lists to class *group* precedence lists. As mentioned, this has not yet been implemented in gbeta; the main problem is that the existence of unordered mixins in methods would conflict with the usage of `INNER` to determine the combined behavior.

It is not hard to see that the algorithmic version of C3 actually implements the formalization presented here—the algorithm each time selects the least remaining element according to our formalization of C3.

As an aside it is interesting to note that gbeta actually used a quite different algorithm for linearization during a period of more than a year. Only after the above formalization was created did it become clear that the C3 algoritm (which was simpler and therefore attractive) solved the exact same problem, because both algorithms clearly implement the formal characterization of the linearization. Algorithms *are* generally harder to compare and reason about than declarative formalizations like Def. 3.

3.2 The Propagation Mechanism

This section presents the propagation mechanism in gbeta indirectly, by describing an untyped functional core of gbeta, **gb**. This core expresses the essence of the semantics of object creation and attribute lookup in gbeta, including the semantics of virtuals and the combination mechanism. There is syntax for specifying a program, a rule for building a class from such a program, a rule for creating an object as an instance of a given class, and a rule for looking up a name in a given object.

The abstract syntax for **gb** programs is given in Fig. 5. It includes blocks (corresponding to `(# .. #)` blocks in gbeta), descriptors (blocks with indication of a superclass), and specifications (the right hand side of declarations). The l denotes labels, i.e., a predefined set of identifiers. The only label with a predefined semantics is '`object`' which is the class with no mixins. The syntax includes only one kind of attribute declarations, corresponding to virtual declarations in gbeta. So all attributes are virtual classes (or virtual methods—there is no semantic difference).

$$
\begin{array}{ll}
b = (\#\ l_i : s_i\ ^{i \in I}\ \#) & \text{(block)} \\
d = l\ b & \text{(descriptor)} \\
s = l \mid d & \text{(specification)} \\
l & \text{(label)}
\end{array}
$$

Fig. 5. The abstract syntax of **gb**

There is no statement syntax, but the rules for creating instances and looking up names can be applied repeatedly, so objects and classes at any level of nesting in the program can be created.

The semantic entities are shown in Fig.6. They include the syntax as Block, Descriptor, and Spec. The central concept of mixin is represented by Mixin which is a block in an environment. A class is simply a list of mixins.[7] An environment, Env, is not only the enclosing object but the list of all enclosing objects, ending in the outermost object, which contains everything in the program execution. An Object is a set of attributes, and an Attribute is a pair of a label and its value. The value of an attribute is a list of specifications, each in its own environment.

Since the result of looking up a label in **gb** is always a Class, it would have been natural to use the definition Attribute = Label $*$ Class, but that definition

[7] Note that the notation used for mixin lists elsewhere in this paper has the most specific mixin *rightmost*, since that yields the most natural notation for '`&`' expressions. In this section we put the most specific mixin at the *left* end (reverting the lists), because that is necessary for the standard notation, '$h :: t$', which names the head, h, and the tail, t, of a list by pattern matching (as in SML and other functional languages).

$$
\begin{aligned}
\mathsf{Block} &= (\mathsf{Label} * \mathsf{Spec})\ \mathsf{set} \\
\mathsf{Descriptor} &= \mathsf{Label} * \mathsf{Block} \\
\mathsf{Spec} &= \mathsf{Label} \mid \mathsf{Descriptor} \\
\mathsf{Class} &= \mathsf{Mixin}\ \mathsf{list} \\
\mathsf{Mixin} &= \mathsf{Env} * \mathsf{Block}
\end{aligned}
\qquad
\begin{aligned}
\mathsf{Env} &= \mathsf{Object}\ \mathsf{list} \\
\mathsf{Object} &= \mathsf{Attribute}\ \mathsf{set} \\
\mathsf{Attribute} &= \mathsf{Label} * \mathsf{EnvSpec}\ \mathsf{list} \\
\mathsf{EnvSpec} &= \mathsf{Env} * \mathsf{Spec}
\end{aligned}
$$

Fig. 6. Semantic Entities

conflicts with the dynamic semantics for objects which contain self references. The definition of Attribute in Fig. 6 is one way to handle recursive objects, namely by evaluating specifications lazily.

From Program to Class to Object. A **gb** program is a block. For a given program b we construct the initial class $[([\,], b)]$, which contains one mixin which places b in the empty environment. This class can then be instantiated like any other class, and that initiates the **gb** 'execution'—which is a chain of evaluations of $\textsc{New}(\cdot)$ and $\textsc{Lookup}(\cdot,\cdot)$.

Any given class can be instantiated using the function $\textsc{New}(\cdot)$ which takes a class and yields an object. It is defined in terms of the auxiliary functions $\textsc{Labels}(\cdot)$ and $\textsc{Val}(\cdot,\cdot)$. See Fig. 7.

$$
\begin{aligned}
\textsc{New}(C : \mathsf{Class}) &= \{\ (l, \textsc{Val}(l, C))\ \mid\ l \in \textsc{Labels}(C)\} \\
\forall j \in I.\ \ \textsc{Val}(l_j, (\#\ l_i : s_i\ ^{i \in I}\ \#)) &= s_j \\
\textsc{Val}(l, (e, b) : \mathsf{Mixin}) &= \begin{cases} [(e, \textsc{Val}(l, b))] & \text{if } l \in \textsc{Labels}(b) \\ [\,], & \text{otherwise} \end{cases} \\
\textsc{Val}(l, [\,] : \mathsf{Class}) &= [\,] \\
\textsc{Val}(l, (h :: t) : \mathsf{Class}) &= \textsc{Val}(l, h) +\!\!+ \textsc{Val}(l, t) \\
\textsc{Labels}((\#\ l_i : s_i\ ^{i \in I}\ \#)) &= \{l_i \mid i \in I\} \\
\textsc{Labels}((e, b) : \mathsf{Mixin}) &= \textsc{Labels}(b) \\
\textsc{Labels}([\,] : \mathsf{Class}) &= \emptyset \\
\textsc{Labels}((h :: t) : \mathsf{Class}) &= \textsc{Labels}(h) \cup \textsc{Labels}(t)
\end{aligned}
$$

Fig. 7. Creation of objects ($+\!\!+$ concatenates lists)

Figure 8 presents the semantics attribute lookup. Given an object O and a label l, $\textsc{Lookup}(O, l)$ delivers the result of looking up l in O. It yields a class if l is defined in O, and raises an error otherwise. To lookup l in O we search the labels of O using $\mathrm{L}_{\mathrm{object}}(O, \cdot, l)$. If we find l then we have an EnvSpec list, *ess*, which is then looked up in O using $\mathrm{L}_{\mathrm{envspecs}}(O, ess)$. Note that *ess* is the result

of collecting all contributions to a given attribute—**gb** has virtual attributes, only.

The next step is crucial. The use of C3$(\cdot,\cdot)$ in the definition of $\mathrm{L}_{\mathrm{envspecs}}(\cdot,\cdot)$ constructs the virtual by C3 linearizing all the contributions. A similar core language for BETA would not linearize at this point; it would *replace* the definition in the less specific enclosing class with the definition in the more specific one. Moreover, the static analysis ensures that this always replaces the virtual class with a descendant. Since $A \,\&\, B \leq X$ for $X \in \{A, B\}$ and $A \,\&\, B = B \,\&\, A = B$ whenever $B \leq A$, the BETA semantics comes out as a special case of the gbeta semantics. Finally, $\mathrm{L}_{\mathrm{env}}(\cdot,\cdot)$ is used to look up labels in the given environment e, enhanced with the current object to $O :: e$; this (very late) enhancement of the environment to include the current object is actually the essence of the lazy evaluation that makes it possible to handle recursion. This ends the brief presentation of **gb**.

$$
\begin{aligned}
\textsc{Lookup}(O : \mathsf{Object}, l : \mathsf{Label}) &= \mathrm{L}_{\mathrm{object}}(O, O, l) \\
\mathrm{L}_{\mathrm{object}}(O, [\,] : \mathsf{Object}, l) &= \text{raise Undefined} \\
\mathrm{L}_{\mathrm{object}}(O, ((l', ess) :: t) : \mathsf{Object}, l) &= \begin{cases} \mathrm{L}_{\mathrm{envspecs}}(O, ess), & \text{if } l = l' \\ \mathrm{L}_{\mathrm{object}}(O, t, l), & \text{otherwise} \end{cases} \\
\mathrm{L}_{\mathrm{envspecs}}(O, [\,] : \mathsf{EnvSpec\ list}) &= [\,] \\
\mathrm{L}_{\mathrm{envspecs}}(O, (h :: t) : \mathsf{EnvSpec\ list}) &= \mathrm{C3}(\mathrm{L}_{\mathrm{envspec}}(O, h), \mathrm{L}_{\mathrm{envspecs}}(O, t)) \\
\mathrm{L}_{\mathrm{envspec}}(O, (e : \mathsf{Env}, l : \mathsf{Label})) &= \mathrm{L}_{\mathrm{env}}(O :: e, l) \\
\mathrm{L}_{\mathrm{envspec}}(O, (e : \mathsf{Env}, (l, b) : \mathsf{Descriptor})) &= (O :: e, b) :: (\mathrm{L}_{\mathrm{env}}(O :: e, l)) \\
\mathrm{L}_{\mathrm{env}}([\,] : \mathsf{Env}, l) &= \begin{cases} [\,], & \text{if } l = \text{"}\texttt{object}\text{"} \\ \text{raise Undefined}, & \text{otherwise} \end{cases} \\
\mathrm{L}_{\mathrm{env}}((h :: t) : \mathsf{Env}, l) &= \begin{cases} \textsc{Lookup}(h, l), & \text{if } l \in \textsc{Labels}(h) \\ \mathrm{L}_{\mathrm{env}}(t, l), & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 8. Looking up a label in an object

The Relation to gbeta. The core language **gb** described in the previous section is of course very different from gbeta. It is purely functional, so the **gb** objects (in environments) are replaced with store locations ("pointers") in gbeta. In **gb**, names are matched according to their spelling. Since gbeta uses static name-binding, the identification of names in **gb** is much more inclusive (**gb** considers two declarations related in many cases where gbeta considers them unrelated). To obtain the effect of static name binding in **gb** we would need to rename identifiers in a given program, but since the static analysis of gbeta determines exactly what names are equivalent, it is certainly a tractable problem to choose new names such that only the gbeta-equivalent names are spelled identically.

In **gb**, the immanent recursion of objects is handled using lazy evaluation of attributes. In gbeta, the exact mixins contributing to a given declaration are determined at compile-time,[8] and cycles (e.g., a class which indirectly inherits from itself) are detected using a graph coloring algorithm: Whenever the type of a declaration depends on itself, the program is rejected with a 'cyclic dependency' error message. The run-time context is represented *relative* to a current object in the gbeta static analysis, since the actual objects are of course not available before run-time.

However, **gb** accurately reflects the semantics of looking up names in gbeta, starting with declarations in the currently selected mixin (e.g., the method being executed) and continuing through all enclosing objects until the outermost "universe" object is reached. Similarly, the semantics of virtuals is the same in **gb** and gbeta (apart from the name binding issue which was mentioned above). Each attribute includes the full context (potentially many objects) in **gb**, but this has been reduced to one pointer shared by several attributes in gbeta. The semantics comes out clearer and simpler with the complete environment attached to each attribute. Note that **gb** does not need to include the explicit linearization operator '`&`' since the semantics of that operator can be obtained using a couple of auxiliary classes and virtuals. This is because virtual class contributions are linearized with C3, just like '`&`' expressions.

4 Implementation

Our implementation of gbeta has been available on the Internet (with source code under GPL) since August 1997.[9] It is implemented in BETA. We wrote about 70 KLOC specifically to implement gbeta. Since gbeta generalizes the semantics of BETA at such a fundamental level the implementation essentially had to start from scratch, but of course the techniques used in the implementation of BETA, e.g. in the type checker, have been an important starting point for the more general approach in gbeta.

However, the language and its implementation are still under development, and in particular there are many possibilities for improving the performance. The primary goal so far has been to develop the language design, and to provide an implementation which would allow hands-on experience with the consequences of design decisions. The practical experience has been an invaluable source of feed-back, both when small test programs revealed new ways to use a given construct or when they demonstrated problems with design decisions, but also when a particular design proved hard to type check and consequently turned out to be be ill defined.

As mentioned above, the source code is freely available. So far the project has been more cathedral than bazaar [22], but collaboration on the further development of gbeta is indeed welcome!

[8] Unless we use the support for dynamic inheritance, not covered in this paper

[9] See `http://www.daimi.au.dk/~eernst/gbeta/`

5 Related Work and Discussion

The basis on which this work has been built is the Scandinavian tradition of object-orientation, in particular the BETA community [19, 14, 4]. In relation to this basis, gbeta represents a generalization which is intended to further develop rather than revolutionize. However, as mentioned in Sect. 4, the fundamental mechanisms of gbeta—the new semantics of virtuals and the support for class combination—are so different that a complete reimplementation was required.

To our knowledge there are no existing systems which support a mechanism like our propagation.

CLOS [12] was an important source of inspiration, and gbeta does not have an equivalent of the CLOS meta-object protocol; it is probably necessary for static type checking to lay down a firm foundation for the language and not let *everything* be user-definable. Thus, in addition to the propagation mechanism, gbeta offers static type checking and the more general combination with "`&`" which applies both to classes and methods; standard CLOS method combination only applies to generic functions, and only as in the special case of one-level propagation from classes to enclosed methods.

Several efforts aim at supporting advanced combination of separate entities (subjects, aspects, ...), in order to allow for a better separation of concerns which otherwise appear to be scattered globally. We believe that such efforts and gbeta basically attack the same problem from two sides, and generally a seamless language integration is a desirable end-point for any of these approaches.

In subject oriented programming [10] each subject is a separate "universe" consisting of fragments of classes in the system. This makes it possible for one (complete) class to participate in several different subjects (universes) with different interfaces in each subject, hence allowing designers to concentrate on one perspective at a time and later combine the subjects to complete systems. This allows for large-scale combination strategies. However, the interaction between the different subjects in any given class must be resolved explicitly, e.g., whether a given name denotes a shared entity in two subjects or two separate entities. In gbeta, the propagating combination of large systems (groups of groups of families of classes etc.) allows for implicit resolution of those matters in a modularized fashion: E.g., two entities are shared iff they are statically recognized as being the same, because they are declared in the same declaration of a shared superclass.

AspectJ [13, 21] supports combination of aspects in Java which amounts to the *weaving* of method implementations and class members from separate *aspects*. Otherwise globally scattered but logically connected pieces of code can then be expressed in separate (aspect) modules and combined in flexible ways. Certain features in AspectJ have no parallel in gbeta, such as selecting the methods in which to put a given aspectual piece of code by an expression which may contain wildcards ("add this statement to the beginning of all methods matching the expression `Point.*(*)`"). AspectJ is currently implemented as a textual pre-processor which generates Java-code. Even though some type checking can be carried out on pre-weaved source code there is always a *transition* which must

be dealt with, in type-checking and error reporting, and even in compilation and optimization. We believe that full language integration is a natural goal to strive for. The challenge is to reach this goal without sacrificing the flexibility; `gbeta` demonstrates one possible approach, entirely language integrated and with strict type analysis.

In [20] the notion of Adaptive Plug-and-Play Components (APPCs) is introduced. An APPC is a construct which aims to capture collaboration between classes at a level of granularity between traditional classes (too small for reuse) and modules (too large, inflexible). An *interface class graph* is used to specify the bindings between an APPC and a concrete class graph, such that a given APPC applies to different contexts regardless of such details as choice of names of classes and methods. Moreover, results from the work on Adaptive Programming [17, 16] are applied, ensuring that attributes can be looked up across a varying path of part objects (e.g., both `anObj.target` and `anObj.xyz.target` can be accessed from a given `anObj` using the same specification in an APPC). In `gbeta` there is no parallel to the adaptive programming path abstraction. It would probably be possible to add such a mechanism to `gbeta`, but that would be orthogonal to the rest of the language—a separate concern. However, the binding of a virtual class to a concrete class in `gbeta` supports a similar detachment of the collaboration (entity with nested virtual classes) from concrete choice of names as the class interface graph does with APPCs.

The other line of related work is about mixins. They were introduced as a particular usage of multiple inheritance in CLOS and later formalized and developed as an independent notion which can subsume the concepts of inheritance and class. The ground-breaking article [2] presents the notion of mixins and mixin combination as a generalization of several known inheritance models, clearly influencing this work. Linearizing multiple inheritance is described as a mechanism which supports mixin style inheritance, but adds too much complexity. In `gbeta`, linearization was introduced in order to have a mechanism for large-scale mixin combination, and for implicit propagation of mixin combination. The problems usually associated with linearization are greatly reduced in `gbeta`: firstly, the linearization in use has the three consistency properties mentioned in section 3.1. Secondly, the static name binding enables programmers to inspect at compile-time exactly what declaration any given name application refers to.

In [3] a small and beautifully formalized set of operations on modules (mixins) is defined, supporting many different notions of inheritance and class combination, although with quite low-level operations. Since there is no particular description of a coherent high-level system of usage of those primitives, this is a work which mostly serves to solidify the foundation of more high-level approaches. The notion of mixins is different in [23] where each mixin (method) is nested in a class; by executing a mixin method the object changes class and, e.g., obtains some new instance variables. In this approach each class must foresee every possible extension, and every generally applicable class must be a member of the top class "`object`". This does not seem very manageable in a large-scale

development project, but a very good module system might help. Finally, [8] presents an extension of a subset of Java with mixins, called MixedJava. Like in our approach, the interaction between a mixin and the actual superclass is known at the declaration of the mixin (using the *inheritance interface* in MixedJava and the declared superclass in gbeta), such that static type-checking of the mixin implementation can be carried out once, independently of applications. In MixedJava an interface is the only option for specifying the assumable properties of the actual superclass, whereas gbeta offers the more flexible option of using anything from a completely abstract class to a fully concrete class as the formal superclass. Of course, none of these systems have a mechanism which is similar to the propagation support offered in gbeta.

6 Future Work

The language is stabilizing, though multi-methods might still be added, so an obvious target for future work is to clarify whether (and how much) the chosen language design adversely affects the performance of compiled code, or of compilation. The current state of affairs just demonstrates that it is not trivial to reach an implementation with good performance. The fact that attributes cannot have fixed offsets (as they can in Beta) implies that simple data member lookup will be slightly more expensive in gbeta.

Another interesting topic is the linearization algorithm. As mentioned in Sect. 3.1, the linearization generalizes from lists of individual mixins to lists of *groups* of mixins, and this provides the nice closure property that *all* class combinations are well-formed. However, is does not blend well with method combination, and some solution to that problem must be found first.

Finally, several partial specifications of the formal semantics of gbeta and its type system have been created, and those efforts should be combined into a full formalization of the language.

7 Conclusion

The language gbeta was presented, with special focus on the support for propagation of class and method combination, according to static dependencies between virtuals and their context. Such propagation enables the specification of, e.g., mutually recursive families of classes as seen from different perspectives. These class family aspects can then be combined with simple top-level expressions, implicitly leading to the combination of the aspects of members of the family and by propagation also the individual methods of those members. The integration of the propagation mechanism into a statically typed language ensures seamless high-level support for the constructs in the type-system and elsewhere. We believe that language integration must be a goal for all similar efforts to support the clean separation of currently globally scattered concerns.

Acknowledgements

The design and implementation of gbeta has taken years, and lots of discussions at the University of Århus have been part of the process. In particular, Mads Torgersen and Kresten Krab Thorup and also Henry Michael Lassen and Søren Brandt gave valuable input. Recently a stay at the University of Washington has given lots of food for thought, especially from Vassily Litvinov and Craig Chambers. Dave Ungar from Sun Microsystems Laboratories has also given valuable advice. The anonymous reviewers gave well-informed and detailed advice, greatly improving the paper. Finally, the whole project would be unthinkable without the BETA tradition and community at large which includes many more people at Århus University and nearby, first of all my advisor Ole Lehrmann Madsen.

References

[1] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, Andrew L. M. Shalit, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 69–82, New York, October6–10 1996. ACM Press.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, pages 303–311, October 1990. Published as Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, volume 25, number 10.

[3] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, April 1992. IEEE Computer Society.

[4] Søren Brandt and Jørgen Lindskov Knudsen. Generalising the BETA type system. In P. Cointe, editor, *Proceedings ECOOP '96*, LNCS 1098, pages 421–448, Linz, Austria, July 1996. Springer-Verlag.

[5] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98*, LNCS 1445, pages 523–549, Brussels, July 1998. Springer-Verlag.

[6] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA'92*, volume 27, no 10, pages 16–24, Vancouver, USA, October 1992. ACM SIGPLAN Notices.

[7] R. Ducournau, M. Habib, and M.L. Mugnier M. Huchard. *Proposal for a Monotonic Multiple Inheritance Linearization*, volume 29, no 10, pages 164–175. ACM SIGPLAN Notices, Oregon, USA, October 1994.

[8] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[10] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *OOPSLA 1993 Conference Proceedings*, volume 28/10 of *ACM SIGPLAN Notices*, pages 411–428. ACM Press, October 1993.

[11] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *6th Workshop on Foundations of Object-Oriented Languages (FOOL)*, at `http://www.cs.williams.edu/~kim/FOOL/sched6.html`, January 1999.

[12] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA, 1989.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.

[14] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson. *Object-Oriented Environments – The Mjølner Approach*. Prentice Hall, Hertfordshire, GB, 1993.

[15] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Classification of actions, or inheritance also for methods. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP'87*, LNCS 276, pages 98–107, Paris, France, June 15-17 1987. Springer-Verlag.

[16] K. J. Lieberherr, I. Silva-Lepe, and C. Xaio. Adaptive object-oriented programming using graph-based customizations. *Communications of the ACM*, 37(5):94–101, May 1994.

[17] Karl J. Lieberherr and Cun Xiao. Minimizing dependency on class structures with adaptive programs. *Lecture Notes in Computer Science*, 742:424–441, 1993.

[18] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 397–406, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.

[19] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.

[20] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. *ACM SIGPLAN Notices*, 33(10):97–116, October 1998.

[21] The AspectJ project group. Aspectj home page. `http://www.parc.xerox.com/spl/projects/aop/aspectj`, March 1999.

[22] Eric S. Raymond. *The Cathedral and the Bazaar*. at `http://www.tuxedo.org/~esr/writings/cathedral-bazaar/`, 1998.

[23] Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested Mixin-Methods in Agora. In Oscar M. Nierstrasz, editor, *Proceedings of the 7^{th} European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 197–219. Springer-Verlag, 1993.

[24] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97*, LNCS 1241, pages 444–471, Jyväskylä, June 1997. Springer-Verlag.

[25] Mads Torgersen. Virtual types *are* statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, at `http://pauillac.inria.fr/~remy/fool/program.html`, January 1998.