Safe and Explicit Composition of Class Properties

Stéphane Ducasse, Nathanael Schärli, Roel Wuyts

Institut für Informatik und Angewandte Mathematik
University of Bern, Switzerland

IAM-03-007

September 11, 2003

Abstract

As object-oriented programmers, we are trained to capture common properties of objects in classes that can be reused. Similarly, we would like to capture common properties of classes in metaclass properties that can be reused. This goal has led researchers to propose models based on explicit metaclasses, but this has opened Pandora's box leading to metaclass composition problems. Numerous approaches have been proposed to fix the problem of metaclass composition, but the composition of conflicting properties was always resolved in an adhoc manner. Our approach uses traits, groups of methods that act as a unit of reuse from which classes are composed, and represent metaclass properties as traits. Metaclasses are then composed from these traits. This solution supports the reuse of metaclass properties, and their safe and automatic composition based on explicit conflict resolution. The paper compares existing models for composing metaclass properties, discusses traits and our solution, and shows some concrete examples implemented in the Smalltalk environment Squeak.

CR Categories and Subject Descriptors: D.3.1 [Programming Languages]: D.3.2 Language Classifications, D.3.3 Language Constructs and Features (E.2), 68N15 Programming languages, 68N19 Other programming techniques (object-oriented, sequential, concurrent, automatic, etc.) [New MSC2000 code]

Safe and Explicit Composition of Class Properties

STÉPHANE DUCASSE
and
NATHANAEL SCHÄRLI
and
ROEL WUYTS
{stephane.ducasse, nathanael.schaerli, roel.wuyts}@iam.unibe.ch
Software Composition Group
University of Berne

As object-oriented programmers, we are trained to capture common properties of objects in classes that can be reused. Similarly, we would like to capture common properties of *classes* in *metaclass properties* that can be reused. This goal has led researchers to propose models based on *explicit* metaclasses, but this has opened Pandora's box leading to metaclass composition problems. Numerous approaches have been proposed to fix the problem of metaclass composition, but the composition of conflicting properties was always resolved in an *ad-hoc* manner. Our approach uses *traits*, groups of methods that act as a unit of reuse from which classes are composed, and represent metaclass properties as traits. Metaclasses are then composed from these traits. This solution supports the reuse of metaclass properties, and their safe and automatic composition based on explicit conflict resolution. The paper compares existing models for composing metaclass properties, discusses traits and our solution, and shows some concrete examples implemented in the Smalltalk environment Squeak.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance—*documentation*; H.4.0 [**Information Systems Applications**]: General

General Terms: Metaclass composition, Inheritance, Mixins, Traits

Additional Key Words and Phrases: Inheritance, Metaclass Composition, Mixins, Multiple Inheritance, Reuse, Smalltalk, Traits

1. REUSING METACLASS PROPERTIES

In class-based object-oriented programming, classes are used as instance generators and to implement the behaviour of objects. In some object-oriented languages that feature classes, like CLOS or Smalltalk, classes themselves are first-class objects, and instances of so-called *metaclasses* [Ingalls 1976; Cointe 1987]. In the same way that classes define the properties for their instances (objects), metaclasses implement the properties for their

Author's address: Stéphane Ducasse.
Software Composition Group, University of Berne, 10 Neubrueckstrasse, CH-3012 Berne. Switzerland.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
©

instances (classes). Properties of metaclasses are called *metaclass properties*. Examples of metaclass properties are *Singleton*, *Final*, *Abstractness* ... [Ledoux and Cointe 1996]

Treating classes as first class-objects and having metaclasses is important for two main reasons:

—*Uniformity and Control.* In a pure object-oriented language it is natural for classes to be instances of metaclasses. The uniformity defines metaclasses as the natural place to *specify* and *control* object creation and other class behavior.

—*Reuse of Class Behavior.* Since a metaclass is just like any other class, class behavior is reused and conventional reuse and decomposition techniques are applied to the metaclasses [Ledoux and Cointe 1996]. Hence the same techniques available for base classes (inheritance and overriding of methods, for example) are applicable at the meta-level.

When a language has metaclasses, those metaclasses can be *implicit* or *explicit*. With *implicit* metaclasses the programmer cannot specify the metaclass for a class [Goldberg and Robson 1989]. As such, implicit metaclasses successfully address the goal of "uniformity and control", but they provide very little help for achieving "reuse of class behavior". *Explicit metaclasses* avoid this limitation because the programmer can explicitly state from which metaclass his or her classes are instances [Ingalls 1976; Cointe 1987; Briot and Cointe 1989; Kiczales et al. 1991; Danforth and Forman 1994].

Languages without explicit metaclasses suffer from the fact that metaclass properties cannot be reused across classes, and that they cannot be combined. For example, every time one needs a class with the Singleton behaviour, the same implementation needs to be done over and over again. With explicit metaclasses, however, the singleton class property can be factored out to a Singleton metaclass, which can then be used to instantiate classes that exhibit the Singleton behavior.

There is a long tradition of languages trying to achieve reuse and composition of metaclass properties by treating classes as first-class objects: from Lisp-based object-oriented extensions such as CLOS [Steele 1990], Flavors, or ObjVlisp [Cointe 1987] via C++-based reflective cores such as SOM [Danforth and Forman 1994; Forman et al. 1994; Forman and Danforth 1999], to pure object-oriented languages such as Smalltalk [Ingalls 1976; Goldberg and Robson 1989], ClassTalk [Briot and Cointe 1989], or NeoClassTalk [Bouraqadi-Saadani et al. 1998; Ducasse 1999]. Related approaches such as CodA [McAffer 1995], Moostrap [Mulet et al. 1995], Iguana/J [Redmond and Cahill 2002], Guarana [Oliva and Buzato 1999], OpenC++ [Chiba and Masuda 1993], or Reflex [Tanter et al. 2001], support the composition and reuse of meta-objects.

While these languages achieve some support for the reuse of metaclass properties, they also suffer from several limitations. First of all some approaches sacrifice the compatibility between the class and the metaclass level [Cointe 1987; Graube 1989]. Secondly there are some approaches that are specifically designed to avoid the compatibility problems from the first point. Their solutions, however, rely on *ad-hoc* composition mechanisms that are based on automatic code generation and dynamically changing the meta-metaclass [Bouraqadi-Saadani et al. 1998]. Not only does this make it hard to understand the resulting code, it also leads to problems in case of conflicting properties and results in hierarchies that are fragile with respect to changes. The solutions are not satisfactory from a conceptual point of view either, because the metalevel (or meta-metalevel) does not employ object-oriented techniques (such as inheritance or instantiation) but *ad-hoc* mechanisms only applicable on the meta-level to do metaclass composition. This breaks the funda-

mental idea of reflective programming that uses the *available* features of a language to define and control the behavior of the language itself [Kiczales et al. 1991]. Last but not least other approaches used in the specific context of meta-objects use chain of responsibility [Mulet et al. 1995] or composite meta-objects [Oliva and Buzato 1999] to compose meta-objects. The first approach does not provide the full control of the composition. The second forces the programmer to develop specific meta-objects to compose others.

Our approach allows one to safely compose and reuse metaclass properties based on a general-purpose object-oriented language technology called *traits* [Schärli et al. 2003]. Traits are composable units of behavior that close the large conceptual gap between a single method and a complete class. Consequently, we model metaclass properties with traits and use trait composition to safely combine and reuse them in our metaclasses. In this way we enjoy all the conceptual benefits of the traits composition model. In particular, composition and conflict resolution are *explicit* and under *control* of the composing entity. This allows explicit control over a composition and resolution of conflicts that occur when two properties are composed that do not quite fit together. Our approach is safe in the sense that it supports upward and downward compatibility [Bouraqadi-Saadani et al. 1998].

The rest of the paper is structured as follows. Section 2 summarizes the criteria to differentiate approaches that solve metaclass compatibility problems. Section 3 then does a detailed analysis of Smalltalk, CLOS, SOM, and NeoClasstalk, comparing their solutions. Section 4 gives an overview of traits, and Sections 5 and 6 show how traits successfully support the definition and composition of metaclasses based on a set of metaclass properties. Section 7 evaluates our approach, in Section 8 we have a look at related work, and Section 9 concludes this paper and gives an outlook on future work.

2. EVALUATION CRITERIA

Offering explicit metaclasses is a way to reuse metaclass properties but it also opens the door for metaclass compatibility problems [Graube 1989]. This section defines criteria by which approaches that solve metaclass composition problems can be characterized and distinguished. We start by listing two criteria that were already identified in [Bouraqadi-Saadani et al. 1998] (*upwards* and *downwards* compatibility), and then introduce four new ones that were not previosuly considered (*per class property*, *property composition*, *property application*, and *control of the composition*).

Upward Compatibility. The fact that classes are instances of other classes which define their behavior introduces hidden dependencies in the inheritance relationships between the classes and their metaclasses. Careless inheritance at one level (be it the class or metaclass level), can break inter-level communication. N. Bouraqadi et al. [Bouraqadi-Saadani et al. 1998] refined the metaclass compatibility problems in two precise cases named *upward* and *downward* compatibility.

Let B be a subclass of A, MetaB the metaclass of B, and MetaA the metaclass of A. Upward compatibility is ensured for MetaB and MetaA iff: every possible message that does not lead to an error for any instance of A, will not lead to an error for any instance of B.

Figure 1 left illustrates upwards compatibility. When an instance of B receives the message i-foo, the message c-bar is sent to MetaB. The composition of MetaA and MetaB is upward compatible, if MetaB understands the message c-bar, *i.e.,* MetaB should implement it or somehow inherit it from MetaA.

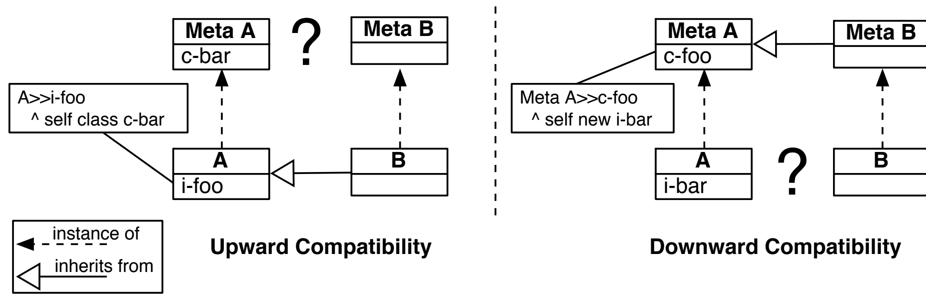

Fig. 1. Left: Upward compatibility - dependencies on the base level need to be addressed at the meta-level. Right: Downward compatibility - dependencies on the meta level need to be addressed at the base level.

Downward Compatibility. *Let MetaB be a subclass of the metaclass MetaA. Downward compatibility is ensured for two classes B, instance of MetaB and A, instance of MetaA iff: every possible message that does not lead to an error for A, will not lead to an error for B.*

Downward compatibility is illustrated in Figure 1 right. When MetaB receives the message c-foo, the message i-bar is sent to a newly created instance of MetaB. The composition of MetaA and MetaB is downward compatible, if that new instance of B understands the message i-bar, *i.e.,* B should implement it or somehow inherit it from A.

Per Class Property. In order to be at all useful, the usage of metaclass properties should not be restricted by the class hierarchy. In particular, it should be possible to assign different metaclass properties to different classes in an inheritance hierarchy [Bouraqadi-Saadani et al. 1998]. As an example, the use of an abstract property would be absolutely pointless if all subclasses of an abstract class had to be abstract as well.

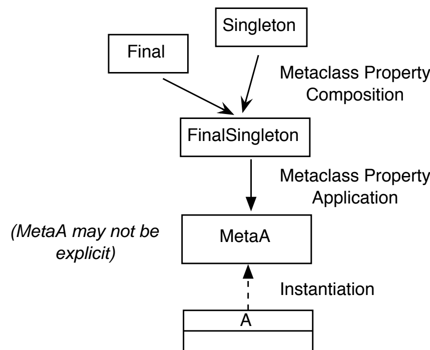

Fig. 2. Property Composition and Property Application: two different periods in the process of reusing metaclass properties.

Property Composition. One of the main goals of having explicit metaclasses is to *combine metaclass properties* as shown by the Figure 2, so that one class can for example be both a Singleton and Final. Hence a mechanism is needed that supports such property

composition. This can be a general-purpose language mechanism such as multiple inheritance [Kiczales et al. 1991; Danforth and Forman 1994], mixin composition [Bouraqadi-Saadani 2003], chain of responsibility [Mulet et al. 1995], or an *ad-hoc* mechanism such as generation of new classes and methods [Bouraqadi-Saadani et al. 1998].

Property Application. Property application is the mechanism by which the composed properties are applied to the metaclasses. As shown by the Figure 2 we distinguish the *composition* of properties from the *application* of a property to a specific metaclass because some approaches employ different techniques for these two purposes. As an example, SOM uses ordinary multiple inheritance to combine metaclass properties but it employs a combination of multiple inheritance and code generation to apply a metaclass properties to a metaclass.

Control. The mechanism used to apply and combine metaclass properties can be *implicit* or *explicit*. We call the mechanism *implicit* if the system automatically combines or applies the metaclass properties and implicitly resolves conflicts in a way that may or may not be what the programmer intends. We call the mechanism *explicit* if the system gives the programmer explicit control over how the properties are combined and applied. In particular, the programmer should have *explicit control* over how conflicts are resolved. For many approaches, this is not the case because the composition of properties is based on a chain of responsibility which does not provide full control of the composition.

3. ANALYSIS OF THE CURRENT SOLUTIONS

This section shows how the main languages that have explicit metaclasses address the problems described in Section 2. We also discuss the solution offered by Smalltalk (although it has implicit metaclasses) since it forms the basis for the NeoClasstalk solution and our own solution. Table I summarizes the comparison of these approaches.

3.1 Smalltalk

In Smalltalk metaclasses are *implicit* and created *automatically* when a class is created [Goldberg and Robson 1989]. Each class is the sole instance of its associated metaclass. This way the two hierarchies are parallel (see Figure 3). Hence the architecture is safe as it addresses compatibility issues but completely prevents metaclass property reuse between several hierarchies.

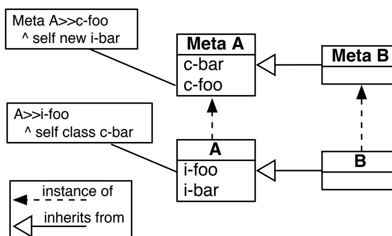

Fig. 3. Smalltalk addresses compatibilities by preventing reuse using implicit metaclasses and parallel hierarchies.

3.2 CLOS

CLOS's approach could be summarized as "do it yourself". Indeed by default in CLOS, a class and its subclasses must be instances of the same metaclass, prohibiting classes in the same hierarchy from having different metaclass properties. For example, in Figure 4, class B has by default the same metaclass as its superclass A, and this cannot be changed. So class B always has the same metaclass properties as class A. Note that since CLOS has explicit metaclasses, multiple inheritance can be used for composing metaclass properties. For example, in the context described by Figure 2 it is possible to use multiple inheritance to explicitly combine the two properties Final and Singleton expressed as metaclasses into a new class SingletonFinal. Note that such an implementation suffers from the same problems as multiple inheritance based on linearization occurring at the base-level [Ducournau et al. 1992].

The general CLOS rule that a class and its subclasses must be instances of the same metaclass can be circumvented using CLOS's meta-object protocol (MOP). Indeed, the generic function validate-superclass [Kiczales et al. 1991] offers a meta-programmer the possibility to specify that a class and its subclasses can be instances of different classes. However, this comes at a very high price because the CLOS MOP does not provide predefined strategies for avoiding compatibility problems or for dealing with possible conflicts. Hence the semantics of the composition has to be implemented manually, a far from trivial undertaking that leads to code duplication and is hard to maintain and debug.

This means that by default CLOS is upward and downward compatible but it prevents usage of different metaclasses within an inheritance hierarchy and reuse of metaclass properties. Both the composition of metaclass properties and the application of properties are done with multiple inheritance. The control of the composition is explicit, because the user has to use multiple inheritance to create a new metaclass. However, since multiple inheritance in CLOS uses implicit linearization, the well-known problems associated with this form of conflict resolution also apply to the meta-level [Schärli et al. 2003].

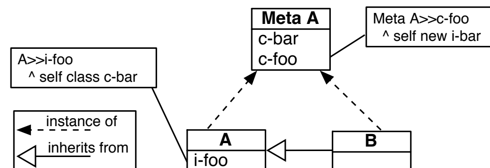

Fig. 4. By default CLOS addresses compatibilities by preventing subclasses to have different metaclasses than their superclasses.

3.3 SOM

The solution proposed by SOM (System Object Model) [Forman and Danforth 1999] is based on the automatic generation of *derived metaclasses*, that multiply inherit from the metaclasses to compose metaclass properties. When at compile time a class is specified to be an instance of a certain metaclass, SOM automatically determines whether upward compatibility is ensured and if necessary creates a derived metaclass. In Figure 5 left, the class B (originally an instance of MetaB), inheriting from class A (instance of MetaA)

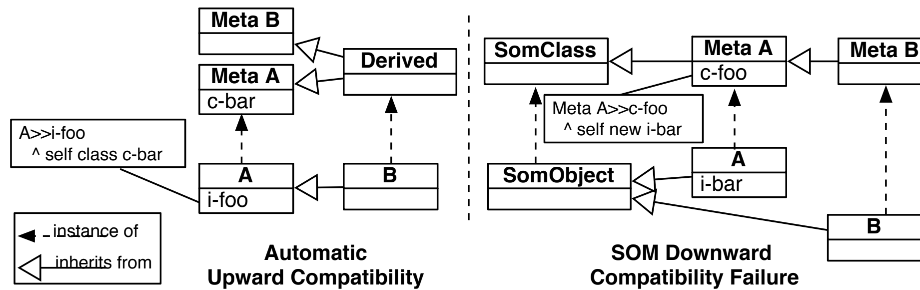

Fig. 5. Left: SOM supports upward compatibility by automatically deriving new metaclasses and changing the class of the inheriting class B. Right: SOM downward compatibility failure example.

finally becomes an instance of a derived metaclass inheriting from MetaA and MetaB. Note that SOM ensures that the existing metaclass MetaB takes precedence over MetaA in case of multiple inheritance ambiguities (since B is a subclass of A).

While SOM supports upward compatibility as shown in Figure 5 left, it does not support downward compatibility [Bouraqadi-Saadani et al. 1998] as shown in Figure 5 right. When the class B receives the c-foo message, a run-time error will occur because its instances do not understand the i-bar message. However, in SOM, contrary to CLOS, two distinct classes need not have the same metaclass. But as in CLOS, the composition of metaclass properties is based on multiple inheritance. The application of a metaclass property is done by a combination of multiple inheritance and automatic class generation. Since this happens at compile time, the programmer has no explicit control over how possible conflicts are resolved.

3.4 NeoClasstalk

NeoClasstalk's approach is interesting since it supports both downward and upward compatibility and enables metaclass property reuse between different hierarchies [Rivard 1997; Bouraqadi-Saadani et al. 1998; Ducasse 1999]. NeoClasstalk uses two techniques to accomplish this: *dynamic change* of classes and the composition of metaclasses by *code generation*. It generalizes the parallel inheritance solution of Smalltalk by enabling metaclass properties reuse, but introduces some problems on its own that we discuss in detail after explaining the basic principles.

NeoClasstalk allows properties to be assigned to classes. Figure 6 shows what happens when assigning a property to Meta B. B inherits from class A, and before the property is assigned it is an instance of the class Meta B. When assigning a property, the system automatically creates a new metaclass Property m + Meta B (called a *property metaclass*), which inherits from the metaclass Meta B and defines the property code. It then changes the class of B to be that newly created metaclass.

In order to be able to reuse the property metaclasses, NeoClasstalk stores the metaclass properties in strings on methods of so-called *meta-metaclasses*. The actual metaclasses are then generated from these strings, as shown for our example in Figure 7. For example, the Property m represented by a meta-metaclass is used to generate a new metaclass named Property m + Meta B from the metaclass Meta B and the Property m.

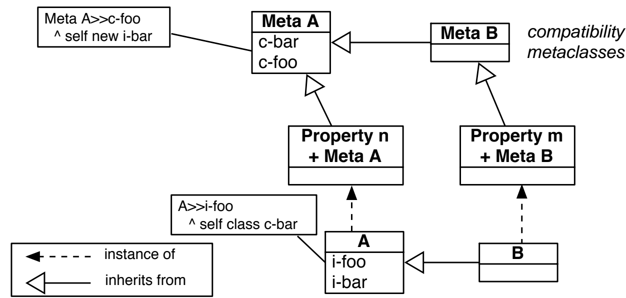

Fig. 6. NeoClasstalk addresses compatibilities and supports reuse using dynamically generated metaclasses and parallel hierarchies.

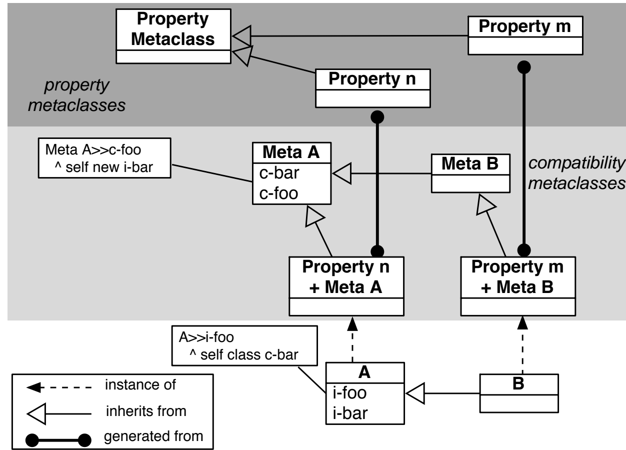

Fig. 7. Assigning the property m to class Meta B and property n to the class Meta A in NeoClasstalk. The light grey area denotes the metaclass area. The dark grey area is the realm of the metaclass properties.

Besides the intrinsic complexity of NeoClasstalk's approach, it has the following drawbacks:

Dynamic class creation and dynamic change of class. The approach relies on the dynamic creation of classes and the dynamic changes of classes. It induces a complex management of meta-metaclass changes that should be propagated to the generated instances. Moreover as programming at the meta-metalevel is based on string manipulation that represents the body of the method of metaclasses, it is not the same as programming at the metaclass or the base level. Basically, despite the name, the property metaclasses are not really at the meta-metaclass level, but merely storage holders for strings. The relation between the meta-metaclass level and the metaclass level is therefore not instantiation, as one would expect, but code generation.

Ad-hoc and Implicit Composition. Property metaclasses are composed by code generation and applied implicitly by defining them in an inheritance chain. The composition

is based on the assumption that a metaclass is designed to be plugged in this inheritance chain and that other composed behavior can be reached via super invocations. The composite metaclass has only limited control over the composed behavior as it can only invoke overridden behavior but does not have the full composition control.

As a summary, NeoClasstalk provides both downward and upward compatibility, and it allows one to assign metaclass properties on a per-class basis. The composition of metaclass properties is implicit and based on code generation and chain of responsibility. The application of metaclass properties is based on dynamic class changes and code generation. As the system automatically generates code and creates new metaclasses, the composition is implicit.

3.5 MetaClasstalk

Metaclasstalk is an extension of NeoClasstalk whose most recent implementation uses mixin composition instead of code generation [Bouraqadi-Saadani 2003]. This experiment makes Metaclasstalk the closest model to our own approach as it supports both downward and upward compatibility while allowing the reuse of metaclass properties.

3.6 Our Approach in a Nutshell

Our approach is based on the Smalltalk concept of parallel hierarchies and as such it provides upward and downward compatibility. However, it circumvents the Smalltalk limits and supports metaclass properties reuse between different hierarchies. This is done by expressing metaclass properties as *traits*: reusable units of behavior that are *explicitly* combined to define classes [Schärli et al. 2003] according to a set of simple rules that give the programmer explicit control over the composition.

Assigning a metaclass property to a specific metaclass therefore comes down to composing this metaclass from the trait that expresses this property. Similarly, composition of metaclass properties is expressed by creating a new trait as the composition of the traits representing the properties that are to be combined. As a result, both property application and composition are explicit and under complete control of the programmer. Section 4 gives an overview of traits, and Section 5 shows in detail how our approach works.

4. TRAITS IN A NUTSHELL

The traits model [Schärli et al. 2003] is an extension of single inheritance with a similar purpose as mixins but avoiding their problems. Traits are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. As such, they allow one to factor out common behaviour and form an intermediate level of abstraction between single methods and complete classes.

A trait consists of *provided methods* that implement its behaviour, and of *required methods* that parameterize the provided behaviour. Traits cannot specify any instance variables, and the methods provided by traits never directly access instance variables. Instead required methods can be mapped to state when the trait is used by a class.

With traits, the behavior of a class is specified as the composition of traits and some *glue methods* that are implemented at the level of the class. These glue methods connect the traits together and can serve as accessors for the necessary state. The semantics of such a class is defined by the following three rules:

—*Class methods take precedence over trait methods.* This allows the glue methods defined in the class to override equally named methods provided by the traits.

—*Flattening property.* A non-overridden method in a trait has the same semantics as the same method implemented in the class.

—*Composition order is irrelevant.* All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Because the composition order is irrelevant, a *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. Traits enforce explicit resolution of conflicts. This can be done by implementing a glue method at the level of the class that overrides the conflicting methods, or by *method exclusion*, which allows one to exclude the conflicting method from all but one trait.

In addition traits allow *method aliasing*; this makes it possible for the programmer to introduce an additional name for a method provided by a trait. The new name can be used to obtain access to a method that would otherwise be unreachable, for example, because it has been overridden.

Traits can also be composed from subtraits; a trait containing subtraits is called a *composite trait*. The composition semantics is the same as explained above with the only difference being that the composite trait plays the role of the class. This means that the order of the subtraits is irrelevant, that methods implemented in the composite trait override equally named methods of the subtraits, and that the semantics of a method defined in a subtrait is identical to the semantics of the same method defined in the composite trait.

To summarize, traits composition is automatic with an explicit resolution of conflicts. It is important to understand that trait composition does not subsume single inheritance; trait composition and inheritance are complementary. Whereas inheritance is used to derive one class from another, traits are used to achieve structure and reusability *within* a class definition. We can summarize this relationship with the equation

$$Class = Superclass + State + Traits + Glue$$

Example: Geometric Objects. Suppose that we want to represent a graphical object such as a circle or square that is drawn on a canvas. Such a graphical object can be decomposed into three reusable aspects — its geometry, its color and the way that it is drawn on a canvas.

Figure 8 shows this for the case of a circle. First of all, the geometry of a circle is expressed with a trait TCircle. Furthermore the color is expressed using a trait TColor, and the behaviour for drawing an object on a canvas is provided by a trait TDrawing:

—TCircle defines the geometry of a circle: it requires the methods center, center:, radius, and radius: and provides methods such as bounds, hash, and =.

—TDrawing requires the methods drawOn: bounds and provides the methods draw, refresh, and refreshOn:.

—TColor requires the methods rgb, rgb: and provides all kind of methods manipulating colors. We only show the methods hash and = as they will be conflicting with others at composition time.

The class Circle is then defined as follows: it specifies three instance variables center, radius, and rgb and their respective accessors methods. It is composed from the three traits TDrawing, TCircle, and TColor. As there is a conflict for the methods hash and = between the traits TCircle and TColor, we alias those methods in both traits to be able to access them in the methods hash and = of the class Circle resolving the conflicts.

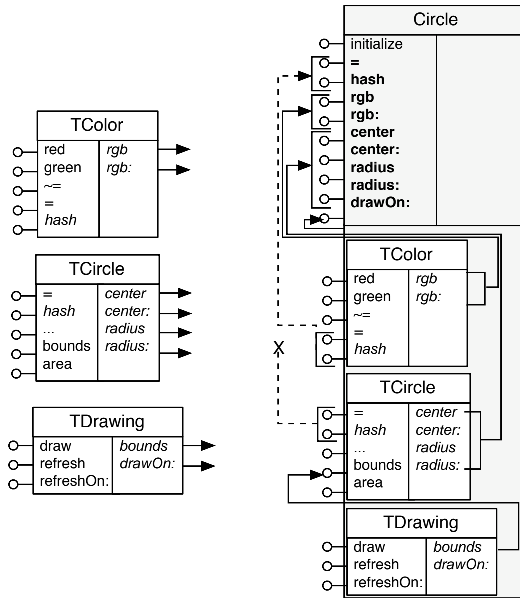

Fig. 8. Left: Three traits TColor,TCircle, and TDrawing. Right: The class Circle composed from these traits. The class circle resolves conflicts by redefining hash and =. Note that the traits figures list their provided methods on the left, and their required methods on the right.

This is expressed in the following piece of Smalltalk code for class Circle, where methods resolving conflicts are shown in bold. Note that the way a class is defined remains largely the same, except for the addition of uses: to specify the composition clause, which defines how the class is composed from traits. Note that we use → for method aliasing and that m1 → m2 means that m1 is an alternative name for the method m2.

```
Object subclass: #Circle
 instanceVariableNames: 'center radius rgb'
 uses: {   TDrawing.
           TCircle @ {#circleHash → #hash .
                 #circleEqual: → #=} .
           TColor @ {#colorHash → #hash .
                 #colorEqual: → #=} }

Circle≫rbg                    Circle≫rgb: aNumber
 ↑rgb                          rgb := aNumber

Circle≫center                 Circle≫center: aNumber
 ↑center                       center := aNumber

Circle≫radius                 Circle≫radius: anInteger
 ↑radius                       radius := anInteger

Circle≫hash
 ↑self circleHash
   bitXor: self colorHash
```

```
Circle≫= anObject
 ↑(self circleEqual: anObject)
   and: [self colorEqual: anObject]
```

5. USING TRAITS TO REUSE AND COMPOSE METACLASS PROPERTIES

Our approach is based on using traits to compose and reuse metaclass properties within the traditional parallel inheritance schema proposed by Smalltalk (See Figure 3). Therefore our approach is safe *i.e.,* it supports downward and upward compatibility still it promotes the reuse of metaclass properties. Composition and application of metaclass properties are based on trait composition, which gives the programmer explicit control.

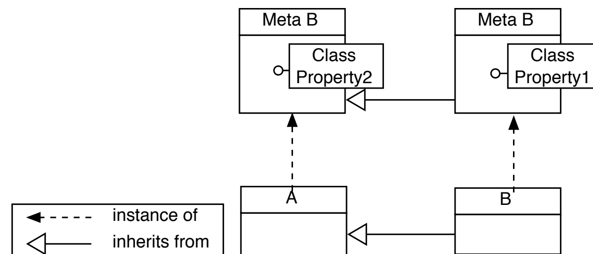

Fig. 9. Metaclasses are composed from traits representing metaclass properties. Traits supports upward and downward compatibility.

We represent metaclass properties as traits, which are then used to compose metaclasses as shown in Figure 9. Since traits have been fully implemented in the open-source Squeak Smalltalk environment [Ingalls et al. 1997], we implemented all the examples shown here in Squeak. During our refactorings of Squeak code we identified the following metaclass properties: TAbstract, TSingleton, TAllInstances, TCreator, and TFinal which we explain below. We start with a simple example illustrating how a class is composed by reusing a metaclass property, then we look how the traditional Boolean hierarchy [Ledoux and Cointe 1996; Bouraqadi-Saadani et al. 1998] is re-expressed with traits and finally Section 6 shows that traits provide a good basis to engineer the meta-level.

5.1 Singleton

To represent the fact that a class is a Singleton, we define the trait TSingleton. This trait defines the following methods: default which returns the default instance, new which raises an error, and reset which invalidates the current Singleton instance. It requires basicNew which returns a newly created instance[1], and the methods uniqueInstance and uniqueInstance:. Note that these accessor methods are needed because traits cannot contain instance variables. Figure 10 left shows the trait TSingleton.

[1]Using basicNew is the traditional way to implement Singleton in Smalltalk when we want to forbid the use of the new method [Alpert et al. 1998]. basicNew allocates objects without initializing them. It is a Smalltalk idiom to never override methods starting with ‘basic’ names.

```
Trait named: #TSingleton
    uses: {} category: 'Traits-Example'


TSingleton≫default
    self uniqueInstance isNil
        ifTrue: [self uniqueInstance: self basicNew].
    ↑ self uniqueInstance
TSingleton≫new
    self error: 'You should use default'
TSingleton≫reset
    self uniqueInstance: nil
```

As an example, suppose that we want to specify that a certain class WebServer is a Singleton. First of all we define the class WebServer in the traditional Smalltalk way as follows:

```
Object subclass: #WebServer
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Traits-Example'
```

Then we specify at the metaclass level *i.e.,* in the class WebServer class, that the class is a Singleton by specifying that the class is composed from the trait TSingleton. The metaclass defines the required state, under the form of the instance variable uniqueInstance, to support the trait definition. It also defines two glue methods uniqueInstance and uniqueInstance: as accessor methods for the instance variable uniqueInstance. Note that the method basicNew is provided by the class Behavior (see Figure 10 right).

```
WebServer class
    uses: {TSingleton }
    instanceVariableNames: 'uniqueInstance'
WebServer class≫uniqueInstance
    ↑ uniqueInstance
WebServer class≫uniqueInstance: anObject
    uniqueInstance := anObject
```

5.2 The Boolean Hierarchy Revisited

The Smalltalk Boolean hierarchy is composed of the class Boolean which is abstract and has two subclasses True and False which are singleton classes. Using traits the boolean hierarchy is refactored as shown in Figure 11. Note that we designed the refactored solution to be backwards compatible with the idioms existing in the current Smalltalk implementation and literature [Alpert et al. 1998]. So we assume a method basicNew defined on the class Behavior that can always be invoked to allocate instances and that should not be overriden.

Boolean. The class Boolean is an abstract class so its class Boolean class is composed from the trait TAbstract.

```
Trait named: #TAbstract
    uses: {} category: 'Traits-Example'
TAbstract≫new
    self error: 'Abstract class. You cannot create instances'
TAbstract≫new: size
```

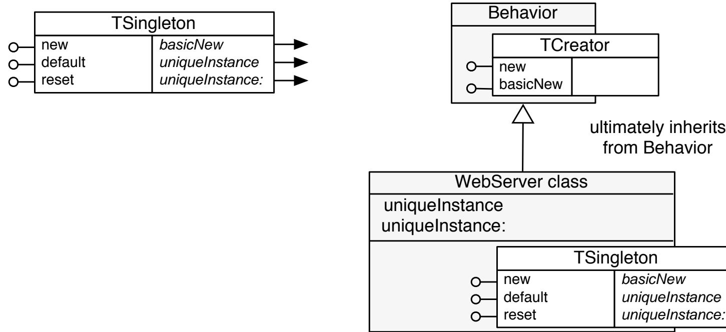

Fig. 10. Left. The trait TSingleton. Right. The class Behavior, the root of metaclasses in Smalltalk, is composed from the trait TCreator and as such provides the method basicNew.

```
    self error: 'Abstract class. You cannot create instances'
```

False and True. The classes False and True are Singletons so their classes False class and True class are composed from the trait TSingleton which is then reused in the two classes.

As mentioned above, the trait TSingleton requires the methods basicNew, uniqueInstance, and uniqueInstance:. Therefore the class False class (resp. True class) has to define an instance variable uniqueInstance and the two associate accessor methods uniqueInstance and uniqueInstance:. Note that the method basicNew does not have to be redefined locally in the class False or True class as it is inherited ultimately from the class Behavior, the inheritance root of the metaclasses [Goldberg and Robson 1989] (see Figure 12).

```
False class
    uses: {TSingleton }
    instanceVariableNames: 'uniqueInstance'
False class≫uniqueInstance
    ↑ uniqueInstance
False class≫uniqueInstance: anObject
    uniqueInstance := anObject
```

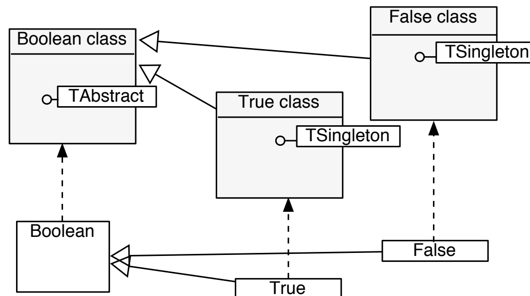

Fig. 11. Boolean hierarchy refactored with traits.

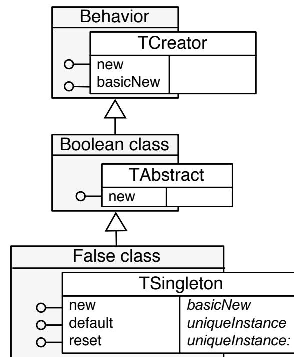

Fig. 12. The complete picture for the Boolean hierarchy solution.

This example shows that metaclass properties are reused over different classes and that metaclasses are composed from different properties.

6. ENGINEERING THE META-LEVEL

So far we presented simple examples that show how traits are well-suited to model metaclass properties, which can then be combined or applied to arbitrary classes. In this section, we show that traits also allow more fine-grained architectures of metaclass properties. We also want to stress that the techniques used here are exactly the same that are used to architect base-level programs and as such traits provide a uniform model.

Since many of these properties are related to instance creation, we first clarify the basic instance creation concept of Smalltalk. In Smalltalk, creation of a new instance involves two different methods, namely basicNew and new [2]. The method basicNew is a low-level primitive which simply allocates a new instance of the receiver class. The method new stands at a higher level and its purpose is to return a usable instance of the receiver class. For most classes, new therefore calls basicNew to obtain a new instance and then initializes it with reasonable default values.

6.1 Metaclass Properties

Figure 13 gives an overview of our metaclass properties. Note that all of these properties are traits, and that they are therefore composed using trait composition.

Allocation. As indicated by its name, the trait TAllocator provides the behavior to allocate new instances. In our case, this is the standard Smalltalk basicNew method, but of course we could also create another trait with an alternative allocation strategy.

Instantiation. The traits TInstantiator and TInitInstantiator are two metaclass properties for instance creation. The trait TInstantiator uses the trait TAllocator and implements the method new in the traditional Smalltalk manner, which means that it does not initialize the newly created instance. The trait TInitInstantiator uses the trait TAllocator. However, as suggested by its name, it actually initializes the newly created instance by calling the method initialize before the instance is returned.

[2]Note that there are also the methods basicNew: and new:, which are used to create objects with indexed fields (*i.e.,* arrays). For sake of simplicity, we do not take these methods into account here.

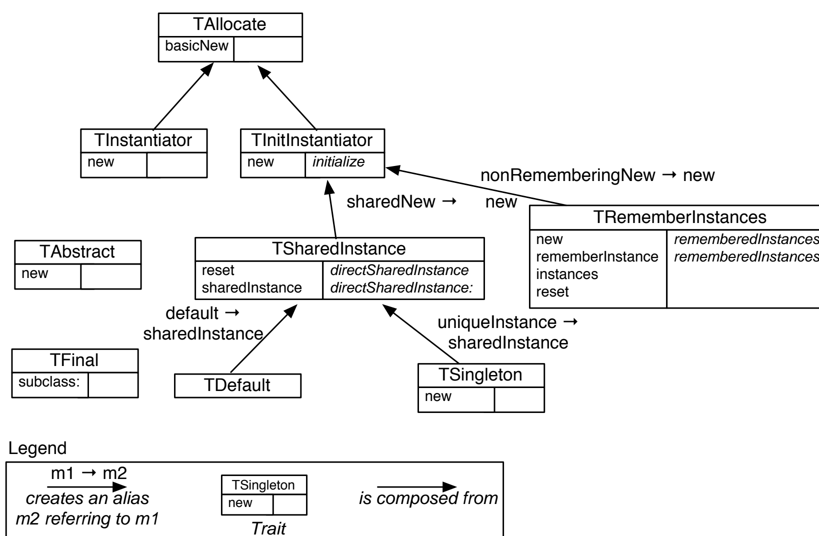

Fig. 13. A fine-grained architecture of metaclass properties based on traits

```
TInstantiator≫new
    ↑self basicNew
TInitInstantiator≫new
    ↑self basicNew initialize
```

Remembering Instances.

The trait TRememberInstances represents an instance creation property ensuring that all the created instances are remembered by the class. It uses the trait TInitInstantiator and aliases the method new of the traits TInitInstantiator which is then available as nonRememberingNew. This aliasing allows one to access the original new method of the trait TInitInstantiator while letting the possibility to override the method new in the trait TInitInstantiator. It requires the methods rememberedInstances and rememberedInstances: that access the collection storing the created instances. Then, it implements the methods new, rememberInstance:, instances, and reset as follows:

```
TRememberInstances≫new
    ↑ self rememberInstance: self nonRememberingNew
TRememberInstances≫rememberInstance: anObject
    ↑ self instances add: anObject
TRememberInstances≫instances
    self rememberedInstances ifNil: [self reset].
    ↑ self rememberedInstances
TRememberInstances≫reset
    self rememberedInstances: IdentitySet new
```

Note that another implementation could be to define the methods reset and rememberedInstances: as trait requirements. This would leave the class with the option to use other implementations for keeping track of the created instances.

Default and Singleton. The traits TDefault and TSingleton implement the metaclass properties corresponding to the Default Instance and Singleton design patterns. Whereas a Singleton can only have one single instance, a class adhering to the Default Instance pattern has one default instance but can also have an arbitrary number of other instances.

Since these two properties are very similar, we factored out the common code into the trait TSharedInstance. In order to get the basic instantiation behavior, this trait uses the property TInitInstantiator and again applies an alias to ensure that the method new is available under the name sharedNew. Then, it implements the methods reset and sharedInstance as follows:

```
TSharedInstance≫reset
    self directSharedInstance: self sharedNew.
TSharedInstance≫sharedInstance
    self directSharedInstance ifNil: [self reset].
    ↑ self directSharedInstance.
```

The property TDefault is then defined as an extension of the trait TSharedInstance that simply introduces the alias default for the method sharedInstance. Similarly, the property TSingleton introduces the alias uniqueInstance for the same method. In addition, TSingleton overrides the method new so that it cannot be used to create a new instance:

```
TSingleton≫new
    self error: 'Cannot create new instances of a Singleton.
        Use uniqueInstance instead'.
```

Another useful metaclass property popularized by Java is the metaclass property TFinal which ensures that the class cannot have subclasses. In Smalltalk, this is achieved by overriding the message subclass: [3]. Note that unlike all the other properties presented in this section, TFinal is not concerned with instance creation and therefore is entirely independent of the other properties. In Section 7.2 we discuss the relevance of the class properties we presented.

6.2 Advantages for the Programmer

Having an architecture of metaclass properties has many advantages for a programmer. Whenever a new class needs to be created, a choice can be made regarding how instances should be created and whether the class should be final or not. Besides the advantage that this avoids a lot of code duplication, it also makes the design much more explicit and therefore facilitates understandability of the class. The level of abstraction of the trait design is at the right level: the traits correspond to the metaclass properties, and the properties can be combined to implemented metaclasses.

In addition, factoring out the properties in such a fine-grained way also gives the user a lot of control about some crucial parts of the system. As an example, consider that we have at first decided to use the trait TInitInstantiator as the basis for all the other instance creation properties. If later on, we would decide to comply to the Smalltalk standard and create uninitialized instances by default, this could be done without modifying any of the involved methods. The only thing we would need to do is to make sure that the traits TRememberInstances and TSharedInstance use the trait TInstantiator instead of TInitInstantiator.

[3] In reality, the method to create a subclass takes more arguments but this is not relevant here

	up	down	per class	application	composition	control
Smalltalk	Yes	Yes	Yes	No	No	No
CLOS	Yes	Yes	No	multiple inheritance	multiple inheritance	explicit + linearization
SOM	Yes	No	Yes	multiple inheritance + code generation	multiple inheritance	implicit
NeoST	Yes	Yes	Yes	inheritance + generation	inheritance + code generation	implicit
MetaST with mixins	Yes	Yes	Yes	inheritance	mixin composition	implicit mixin linearisation
Traits	Yes	Yes	Yes	trait composition	trait composition	explicit

Table I. Comparison of the models from Section 3 on how they handle the composition problems described in Section 2.

Explicit Composition Control Power. By providing several different properties that are all related to instance creation behavior, this example also shows why it is so important to have explicit control over composition and application of metaclass properties. In our example, there are many different properties which essentially introduce variants of the method new, and therefore, combining these properties typically leads to conflicts that can only be resolved in a *semantically* correct manner if the user has explicit control over the composition. In case of traits, this is ensured by allowing partially ordered compositions, exclusions, and aliases.

As an example, imagine that we want to combine the properties TDefault and TRememberInstances in order to get a property that allows both a default instance and also remembers all its instances. With our trait based approach, we do this by creating a new trait TDefaultAndRememberInstances which uses TRememberInstances and TDefault as follows:

```
Trait named: TDefaultAndRememberInstances
    uses: { TDefault @ {#defaultReset → #reset}.
            TRememberInstances − {#new}
                @ {#storeNew → #new.
                    #storeReset → #reset}}

TDefaultAndRememberInstances≫sharedNew
    ↑self storeNew

TDefaultAndRememberInstances≫reset
    self storeReset.
    self defaultReset
```

Since both traits provide a method new, we exclude this method from the trait TRemem-

berInstances when it is composed. As a consequence the trait contains the new method provided by TDefault, which uses sharedNew in order to create a new instance. Since we want to make sure that each new instance is also stored, we override sharedNew so that it calls storeNew, which is an alias for the new method provided by TRememberInstances.

Also the method reset is provided by both traits, and we therefore use aliasing to make sure that we can access the conflicting methods. Then, we resolve the conflict by overriding the method reset so that it first removes the stored instances (by calling storeReset) and then creates a new default instance (by calling defaultReset). Note that the newly created instance will be remembered as the default instance and will also be stored in the collection with all the instances of the class.

7. DISCUSSION

In this section we first discuss the advantages and the limitations of using traits .

7.1 Advantages and Limits of Traits

Advantages. Traits support the decomposition of metaclass properties as reusable units of behavior. Since metaclasses are composed from traits and the model is based on the parallell hierarchy of Smalltalk, it is upward and downward compatible and supports the reuse of metaclass properties across different hierarchies.

In addition the proposed model is uniform with respect to the concepts used at the base and the meta-level (like CLOS). Both levels use the same concepts (traits and inheritance). Furthermore, the model is simple, and there is no need for on-the-fly code generation (as in SOM or NeoClasstalk) or the dynamic change of classes (as in NeoClasstalk).

Metaclass properties can be composed by composing the traits that represent those properties. The application of the properties to an actual metaclass is accomplished by using the approriate composite trait in the construction of the metaclass. The composing metaclass has the *complete* control of the composition, and possible conflicts are resolved *explicitly* when the property is applied on a metaclass.

Having explicit control over the composition is especially important because it allows a programmer to freely adapt the behavior of the composite metaclass and to compose metaclass properties that may not quite fit together. This means that our approach allow the system designers to ship their class hierarchies together with a set of *prefabricated* metaclass properties in the form of traits, which can then be used and combined by the programmers. And in case some metaclass properties built by different vendors will not quite fit together, the traits model will not only indicate the resulting conflicts, but will also provide the programmer with the necessary means to resolve these conflicts in order to achieve the expected semantics.

Limits. Glue methods and state have to be redefined in the metaclass where a property is applied. For example, the instance variable uniqueInstance and the two accessor methods have to be defined in the classes that implement a Singleton.

It may happen that instance variables defined in a superclass are not necessary in the subclasses. For example, if the superclass implements a Singleton and the subclasses do not, then the instance variable that holds the Singleton instance as well as the methods to access it will be inherited by the subclasses. However, this problem is not due to traits by itself but is a result of using the inheritance mechanism in general. Table I compares the approaches.

7.2 About Class Properties

The traits model let us decide if a given functionality is defined as a trait or as a class. When defining a functionality as a trait we automatically offer the possibility to future classes to use the identified behavior. It should be noted that all the functionality of a class *i.e.,* method management, instance memory layout, ancestors and descendant management, method compilation, class description can all defined as traits. However, we do not called them class properties contrary to some authors [Ledoux and Cointe 1996] because we did not reuse them over multiple classes but only within the Class, Metaclass, ClassDescription, Behavior traditional Smalltalk hierarchy. Another point to consider is the role of the classes in the context of a Meta-Object Protocol [Kiczales et al. 1991]; we believe that a lot of class properties identified in [Ledoux and Cointe 1996] are due to the fact that the classes were the single entry point in their MOP, while certain responsibilities are definitively the responsibilities of other meta-entities such as methods.

In this article we present the main *class properties* that we identified during our implementation. It is worth to realize that other important effort building metaclass libraries such as SOM [Forman and Danforth 1999] presents nearly the same kind of class properties.

8. RELATED WORK

In the Section 3 we evaluated how the main languages supporting metaclasses allow one to compose them and reuse metaclass properties. The Table I presents how the different approaches position themselves according to the problems and criteria enounced in Section 2. The only thing we want to add to this analysis is that the table shows the influence of the CLOS approach based on multiple inheritance to support metaclass composition in SOM.

Other approaches such as CodA [McAffer 1995], Moostrap [Mulet et al. 1995], Iguana/J [Redmond and Cahill 2002], OpenC++ [Chiba and Masuda 1993], or Reflex [Tanter et al. 2001], support the composition and reuse of meta-objects. From these approaches the only part that is relevant to the research of this paper is the way meta-objects are composed. Indeed, the application of a meta-object is often done simply by invoking the right MOP entry point on the right object or group of objects. Such a composition is often based on chain of responsibility [Mulet et al. 1995] *i.e.,* a meta-object is designed to be composed in a chain of meta-objects by invoking the overridden functionality. The problem with chain of responsibility is that it forces all the meta-objects to follow a certain architecture but more important the composing meta-object has only a very limited control over the composition, it can invoke the rest or nothing. With traits when there is no conflict, the composition is automatically done. In presence of conflict, the composing metaclass has the complete control over all the composed class properties.

To circumvent the chain of responsibility problem, the authors of Guarana [Oliva and Buzato 1999] proposes the use of composite meta-objects *i.e.,* a meta-objects defined the semantics of the composition of several meta-objects. While this approach works well for coarse-grain composition such as change in the message passing semantics (broadcast, concurrent dispatch, or remote invocations), in the case of metaclass properties it is too heavyweight as it would force the developer to define an explicit composite for any simple conflicts whose reuse is even questionable.

The work developed in CodA [McAffer 1995] is interesting as it structures the meta-level architecture around the life-time of objects. Several meta-objects are responsible for

the different actions (accessing state, receiving, sending, looking up for messages...). However it raises the issue of compatibility between all the meta-objects associated to a given objects. The proposed solution is to manually define a semantically coherent configuration of meta-objects implementing the desired semantics [McAffer 1995].

9. CONCLUSION AND FUTURE WORK

The need to reuse metaclass properties led to the meta-level architectures based on explicit metaclasses *i.e.,* the class of a class may be explicitly specified [Ingalls 1976; Cointe 1987]. While offering the possible reuse of metaclass properties, such models introduced metaclass composition problems [Graube 1989]. Different approaches exist that try to solve metaclass compositions problems, based on multiple inheritance, code generation or automatic change of metaclasses [Danforth and Forman 1994; Forman et al. 1994; Bouraqadi-Saadani et al. 1998]. However, the definition, the composition and the control of the application of the metaclass property were not controllable by the developer or meta-programmer.

In this article we present the traditional problems linked to metaclass property composition, define precisely the life-cycle of a metaclass property (definition, composition, and application) and compare various approaches that tried to solve metaclass property composition problems. Then we show how traits supports the definition, composition and control of metaclass property applications. We model metaclass properties as traits, first class groups of methods and use trait composition to safely combine and reuse them. Using traits to compose metaclass properties solves the metaclass composition problem (upward and downard compatibility is ensured) while supporting the reuse of metaclass properties. In addition, composition and conflict resolution are *explicit* and under *control* of the composing entity.

Whereas the other approaches use special-purpose mechanisms that are invented solely to tackle metaclass composition problems, our approach is based on traits, which are a general-purpose composition mechanism for object-oriented languages. In fact, traits were not designed with meta-programming in mind, and we have already successfully applied them to refactor the Smalltalk collection hierarchy [Black et al. 2003]. The clarity, flexibility, and consistency of the meta-level architecture based on traits is another strong indication that traits are a sound and useful foundation for structuring object-oriented programs.

With our implementation of the Traits model in Squeak we implemented all the examples shown in this article. This proves the power of the traits model to support the definition and composition of metaclass properties. However, we did not introduce traits in the metaclasses of the Squeak kernel. Indeed as Smalltalk is defined in terms of itself this requires a subtle bootstrap when we are modifying its deep kernel. We are currently working on this bootstrapping so that we can introduce traits in the core part of the Squeak reflective architecture.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1). We also like to thank Oscar Nierstrasz, Andrew Black and Noury Bouraqadi for their valuable comments and discussions.

REFERENCES

ALPERT, S. R., BROWN, K., AND WOOLF, B. 1998. *The Design Patterns Smalltalk Companion*. Addison Wesley.

BLACK, A. P., SCHÄRLI, N., AND DUCASSE, S. 2003. Applying traits to the Smalltalk collection hierarchy. In *Proceedings OOPSLA 2003*. To appear.

BOURAQADI-SAADANI, N. M. N. 2003. Mixin-based inheritance for and with explicit metaclasses. In *Proceedings of ESUG'2003 Conference*.

BOURAQADI-SAADANI, N. M. N., LEDOUX, T., AND RIVARD, F. 1998. Safe metaclass programming. In *Proceedings OOPSLA '98*. 84–96.

BRIOT, J.-P. AND COINTE, P. 1989. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*. Vol. 24. 419–432.

CHIBA, S. AND MASUDA, T. 1993. Designing an extensible distributed language with a meta-level architecture. In *Proceedings ECOOP '93*, O. Nierstrasz, Ed. LNCS, vol. 707. Springer-Verlag, Kaiserslautern, Germany, 483–502.

COINTE, P. 1987. Metaclasses are first class: the objvlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*. Vol. 22. 156–167.

DANFORTH, S. AND FORMAN, I. R. 1994. Derived metaclass in SOM. In *Proceedings of TOOLS EUROPE '94*. 63–73.

DUCASSE, S. 1999. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP) 12,* 6 (June), 39–44.

DUCOURNAU, R., HABIB, M., HUCHARD, M., AND MUGNIER, M. 1992. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*. Vol. 27. 16–24.

FORMAN, I. R. AND DANFORTH, S. 1999. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming*. Addison-Wesley.

FORMAN, I. R., DANFORTH, S., AND MADDURI, H. 1994. Composition of before/after metaclasses in SOM. In *Proceedings of OOPSLA '94*, ACM, Ed. ACM Sigplan Notices, vol. 29. ACM, Portland, 427–439.

GOLDBERG, A. AND ROBSON, D. 1989. *Smalltalk-80: The Language*. Addison Wesley. book scglib.

GRAUBE, N. 1989. Metaclass compatibility. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*. Vol. 24. 305–316.

INGALLS, D. 1976. The smalltalk-76 programming system design and implementation. In *POPL'76, Principles of Programming Languages*. ACM Press, 9–16.

INGALLS, D., KAEHLER, T., MALONEY, J., WALLACE, S., AND KAY, A. 1997. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*. 318–326.

KICZALES, G., DES RIVIÈRES, J., AND BOBROW, D. G. 1991. *The Art of the Metaobject Protocol*. MIT Press.

LEDOUX, T. AND COINTE, P. 1996. Explicit metaclasses as a tool for improving the design of class libraries. In *Proceedings of ISOTAS '96, LNCS 1049*. JSSST-JAIST, 38–55.

MCAFFER, J. 1995. Meta-level programming with coda. In *Proceedings ECOOP '95*, W. Olthoff, Ed. LNCS, vol. 952. Springer-Verlag, Aarhus, Denmark, 190–214.

MULET, P., MALENFANT, J., AND COINTE, P. 1995. Towards a methodology for explicit composition of metaobjects. In *Proceedings of OOPSLA '95*. Austin, 316–330.

OLIVA, A. AND BUZATO, L. E. 1999. The design and implementation of guarana. In *USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*.

REDMOND, B. AND CAHILL, V. 2002. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*. Vol. 2374. Springer-Verlag, 205–230.

RIVARD, F. 1997. évolution du comportement des objets dans les langages à classes réflexifs. Ph.D. thesis, Ecole des Mines de Nantes, Université de Nantes, France.

SCHÄRLI, N., DUCASSE, S., NIERSTRASZ, O., AND BLACK, A. 2003. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*. LNCS. Springer Verlag. To appear.

STEELE, G. L. 1990. *Common Lisp The Language*, Second ed. Digital Press. book.

TANTER, E., BOURAQADI, N., AND NOYE, J. 2001. Reflex – towards an open reflective extension of java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. LNCS, vol. 2192. Springer-Verlag, 25–43.