Dynamic Object Evolution without Name Collisions

Mira Mezini

Department of Electrical Engineering and Computer Science (FB 12),
University of Siegen, D-57068 Siegen, Germany
mira@informatik.uni-siegen.de

Abstract. Support for modular evolution of objects is required in many application areas. However, existing mechanisms for incremental behavior composition either do not support evolving objects at all, or do not satisfactorily solve the encapsulation and name collision problems associated with them. In this paper, a new approach to behavior composition in a class-based environment is presented. It is based on the encapsulated object model of class-based inheritance, but introduces an additional abstraction layer between objects and classes. By being responsible for the compositional aspects of the behavior of objects, this layer provides support for the evolution of behavior while at the same time solving the name conflicts that may occur. A formal description of the approach is provided and its feasibility is demonstrated by implementing it as a metalevel extension of Smalltalk-80.

1 Introduction

In several object-oriented application areas, support for dynamic and incremental evolution of the behavior of objects is required. For example, in an application containing entities that model persons, an entity may acquire and abandon several roles, dynamically and independently of each other (e.g. being a student, an employee, a project manager, or a sportsman). These different roles may share some common structure and behavior (e.g. the name of the person), but they may exhibit role-specific behavior which is visible only within a particular context of the respective role (e.g. the identification numbers in the student and sportsman roles of a person). The need for evolving objects in order to adequately model roles has been pointed out in [11].

The subject-oriented approach to software development [12] goes even further: it requires the same objects to be shared by suites of cooperating applications called *subjects*. The goal is to enable the incremental enhancement of already existing systems through new functionality such that the present persistent objects not only remain valid, but are also automatically extended to support the new functionality. Similar to roles, some state and behavior may be shared between different subjects, while others may only be visible in the context of specific subjects. Although supporting subjectivity is a more general problem that is beyond the scope of this paper, support for evolvable objects is definitely one of the basic requirements to be satisfied.

A third area where support for evolvable objects is required are systems that adapt their behavior to run-time conditions and/or application requirements for achieving better performance and/or effectiveness [2, 3, 16]. The components of such systems should ideally have a range of implementations from which the most appropriate one is chosen in a particular situation. Furthermore, for better reusability, the range of supported implementations should be extensible. For example, a toolkit for multimedia application design should be able to support clients in computing environments ranging from hand-held PDA devices to powerful workstations, and communication environments ranging from telephone lines to high-speed and wireless networks. It should also be able to handle a variety of compression standards, rather than become obsolete as new ones emerge. Since multimedia processing is sensitive to variations in resource availability, replaceable presentation processing algorithms that can be dynamically configured according to the run-time environment should be supported.

As indicated by the application areas presented above and illustrated by an example in the next section, two main issues need to be considered by an incremental composition mechanism providing support for evolving objects. First, *dynamic composition* should be supported in the sense that (a) behavior modifications may happen after the object has become operational, and (b) once accomplished, a modification may remain valid only under certain conditions. Second, the mechanism should provide for *internal encapsulation*. In general, objects being considered here are composed of several aspects (roles, subjects, implementations). These aspects may at the same time be coupled to and disjoint from each other in the sense that they may (a) cooperate by sharing state or by jointly performing operations, but (b) nevertheless need to hide parts of their implementation from the others. An incremental composition mechanism that provides for internal encapsulation is able to cope with these conflicting interests. The internal encapsulation issue also manifests itself as the name collisions issue in multiple inheritance hierarchies, as indicated by Knudsen in [18]. However, in the case of evolving objects the situation gets worse, since the conflicts cannot be detected at compile time.

Unfortunately, none of the existing object-oriented incremental composition mechanisms provides a general framework to handle these issues in a uniform manner. The class-based inheritance models do not support dynamic composition, as argued in [11, 22, 24, 30, 32, 35]. The object-based [38] and hybrid [33] inheritance models do provide support for dynamic composition of object behavior, but at the price of violating the encapsulation of the client interface [35] and losing valuable sharing and structuring mechanisms [32]. The approaches that model real-world entities by a set of role [11] or exemplar [19] objects lack object identity semantics and therefore do not support a general form of object evolution. The mixin-methods approach [34] rectifies the encapsulation problem of the object-based model by enclosing possible modifications within an object itself, but the problem is that an object is required to know all its possible modifications in advance [35]. Moreover, neither the mixin-methods approach, nor more advanced approaches to behavior composition, such as *composition filters*

[1] or the *context relation* [30], adequately support internal encapsulation.

With respect to internal encapsulation, most of existing inheritance models either declare name collisions as invalid, i.e. they do not support internal encapsulation at all (like single inheritance and linear approaches to multiple inheritance [14, 26]), or exploit ad-hoc solutions mostly based on renaming and class qualification of message selectors (like graph-based approaches [4, 21, 31]). The latter are criticized in [6, 7, 39] because they violate late binding and consequently restrain reusability, even in a static environment. While being considered as remedies for the static case, more advanced solutions proposed in [6, 7, 39] still exhibit flexibility problems especially in dynamic environments.

We claim that the inability of the existing approaches to uniformly handle dynamic composition and internal encapsulation is due to the lack of sufficient abstraction levels in their design, which are needed to distinguish between defining, composing and supplying behavior. Based on this observation, a mechanism for incremental composition of object behavior is presented in this paper which uniformly deals with both issues, by an explicit combination layer, introduced between an object and the software components that define its behavior(s). A *combiner-metaobject* is associated with each evolving object, responsible for the compositional aspects of the object's behavior. The behavior definition of an evolving object is dispersed between a class that provides the standard behavior of the object and a set of mixin-like [5] software modules, called *adjustments*, that provide different modifications of the standard behavior in specific contexts. However, there is no static inheritance relationship between the class and the adjustments involved in the definition of an object. The information about how these cooperate with each other to yield a full behavior is managed by the combiner-metaobject in a way that provides internal encapsulation. From this information the combiner-metaobject derives the environment where to evaluate the messages sent to the object. Modifying the behavior of the object is a matter of requesting the combiner-metaobject to update the information it encapsulates, and can thus be performed dynamically.

Instead of designing a new language from scratch for demonstrating the viability of our proposal, a prototype of the mechanism has been implemented as an extension of Smalltalk-80 [10]. Smalltalk-80 has been chosen because of its existing metalevel facilities that enable the implementation of the mechanism as an extension of the standard image, without changes in the virtual machine. Other metalevel facilities of Smalltalk-80 also allow us to keep the overhead related to the additional metalevel indirection to a minimum.

The paper is organized as follows. In section 2 we illustrate the issues associated with dynamic object composition and internal encapsulation. In section 3, the proposed model for solving them is presented. A formal description of the model follows in section 4, and implementation issues are briefly discussed in section 5. A discussion of the features of the proposed model and related work follows in section 6. Section 7 concludes the paper and outlines topics for future research.

2 An Example

In this section, the issues that need to be solved by a behavior composition mechanism are illustrated by means of an example. Consider an atomic data type [40] representing a bank account within a database system. Its simplified functional aspect, i.e. the externally observable functionality it provides, is specified in the *Account* module in Fig. 1. The non-functional aspects of the *Account* object, concerned with local synchronization and recovery such as *begin-transaction*, *commit-transaction* and *abort-transaction*, are specified in the *AtomicObject* module in Fig. 1. Suppose that the database system supports object-based concurrency control [37], i.e. the concurrency semantics are defined at the level of object operations and not at the level of elementary read and write operations, taking into account the application semantics in order to increase the level of concurrency[1]. Then, the application semantics to be used by the concurrency controller are represented by the *conflictRelationStructure* data structure in *AtomicObject*. In the presence of an object-based concurrency control, when an operation on an account object is called, information about the transaction making the call should be recorded in order to later use it at transaction commit time for deciding whether to make the effects of a transaction permanent or not. The default information recording functionality is implemented by *recordTransactionInfo* in *AtomicObject* (Fig. 1) which exploits the *infoStruct* for storing the information. The definition of an account object, as seen by the concurrency controller is given by *RecordAccount* in Fig. 1.

Furthermore, suppose that our hypothetical database system is adaptive in terms of both the functional and non-functional aspects. For example, the *debit* functionality of an account may dynamically change depending on some conditions, e.g. depending on who attempts to withdraw how much money if the account is shared by a group of people, or the amount of money already withdrawn from an automatic teller machine within the last 24 hours. These two special withdrawing behaviors are schematically modeled by the *SharedAccount*, respectively *ATMAccount* modules in Fig. 1. A special action, represented by the *specialAction* method, should be performed in both cases before the withdrawal takes place. Additionally, assume that the system supports adaptable concurrency control [2], i.e. a multiplicity of optimistic, pessimistic, or hybrid concurrency control strategies are provided, each valid under certain conditions. For example, the strategy may be switched between pessimistic and optimistic depending on the conflict level in the system, or from one optimistic strategy to another depending on some heuristics based on the state of an object. This implies the presence of multiple modules implementing variations of *AtomicObject*. The behavioral landscape becomes much more complicated when replication and persistency issues are concerned.

The above example system illustrates the issues that a mechanism for incremental object composition should deal with. Obviously, an atomic account

[1] An object-based strategy allows, for example, that two credit operations from two different transactions may be scheduled concurrently.

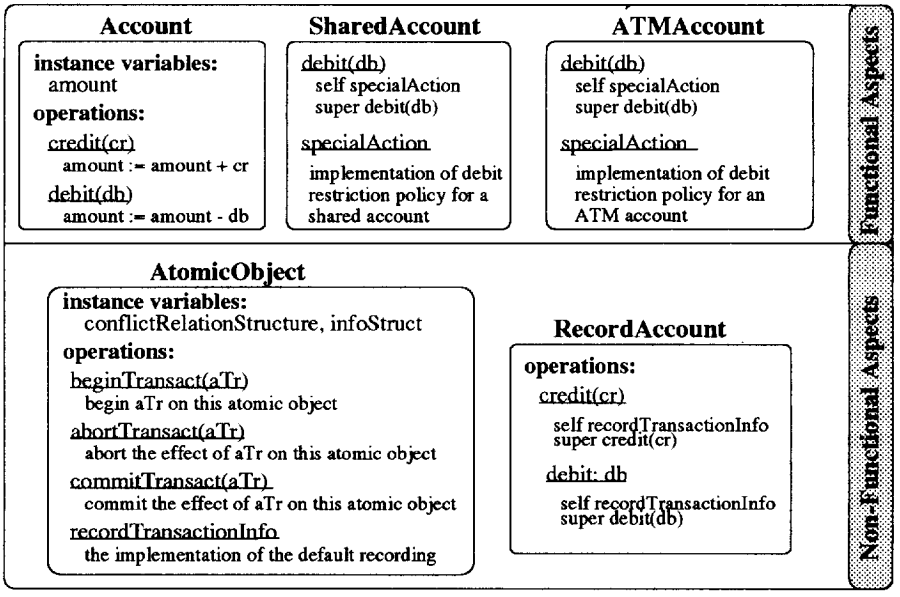

Fig. 1. The specification of an account object

object in this system should be able to change its behavior, becoming a shared account when required by its owner, changing to a shared and ATM-account, when a withdrawal attempt from an automatic teller machine occurs, and then switching to an optimistic atomic account due to a low level of conflicts detected. All of the above behavior alterations may happen after the account object has become operational. Additionally, a particular acquired behavior might remain valid only under certain conditions.

For illustrating the internal encapsulation issue, consider a shared account and its *debit* operation. The complete definition of *debit* is a combination of the definition in *SharedAccount* with that in *Account*, where the former incrementally modifies the latter, i.e. the *super* parameter in *SharedAccount* is the method in *Account*. Now suppose that the account becomes additionally an *ATMAccount*. The definition of *debit* should be further refined by the *debit* operation in *ATMAccount* to perform both special actions before really accomplishing the withdrawal, i.e. there is no conflict here. The situation is, however, different with regard to the *specialAction* operations. Each of the special *debit* operations has it own *specialAction*, which should be invisible to the other. The mechanism for incremental composition of object behavior should provide some kind of internal encapsulation between different subparts of the shared and ATM-account object, in addition to enabling them to jointly contribute to the full debit behavior by being incremental modifications of each other.

3 General Overview of the Model

There are trade-offs related to both issues illustrated above. As pointed out in [35], there is a trade-off between dynamic composition and the encapsulation of the client interface. Taking the object-based approach would allow dynamic modification, but by merging specialization and usage entities into a single unit, all encapsulation problems inherent to specialization [31] now concern the object's client interface as well. Thus, the choice of a class-based approach is obvious, but even the two-layered design of the latter seems to be insufficient for two reasons. First, while the objects are relieved from the burden of generating behavior, i.e. the client and composition interfaces are separated, no abstraction is provided for behavior alteration at the object level. While it is possible to have explicit conditionals in an object's code to alter its behavior to reflect changes in it state, the idea of object-oriented programming is to lift such dispatch at the language level. Avoiding the "case"-like style of procedural programming is an essential factor for the qualitative progress in reusability attributed to the object-oriented paradigm. Furthermore, state-related alterations do not cover all kinds of behavior alterations desired.

Second, there is a trade-off between incremental behavior modification, dynamic or not, and internal encapsulation between the subparts of an object, also called class encapsulation in [27], as identified by [33, 39]. The implementations of different aspects involved in the behavior of an object should be mutually visible to enable incremental modification, and at the same time certain parts of them should be hidden. This trade-off is the source of the name collisions problem. As pointed out by Nierstrasz et al. [27] and Bracha et al. [6], the class-based approach to incremental behavior composition fails to properly solve this trade-off because classes are overloaded to serve both as templates for defining object behavior and as software components responsible for behavior composition by means of inheritance. The separation of these roles by providing explicit components for each of the two different kinds of interfaces supported by classes is the key to uniformly solve the issues of dynamic composition and internal encapsulation in the model proposed in this paper. The model, presented in Fig. 2, has a three layered design consisting of the *behavior definition*, *behavior combination*, and *behavior provision* layer.

The behavior definition layer is responsible for the specification of behavior as a set of independent software modules, and the provision layer is responsible to provide clients with functionality. The known abstractions existing in the standard class-based model, classes and objects, are elements of the definition and the provision layer, respectively. The behavior combination layer represents the additional abstraction between an object and its behavior definition. Its elements, called *metaCombiners*, are responsible for the structural aspects of the behavior of the underlying object and its evolution. Since objects remain the same as in the class-based model, only the first two layers will be the subject of the informal discussion in this section.

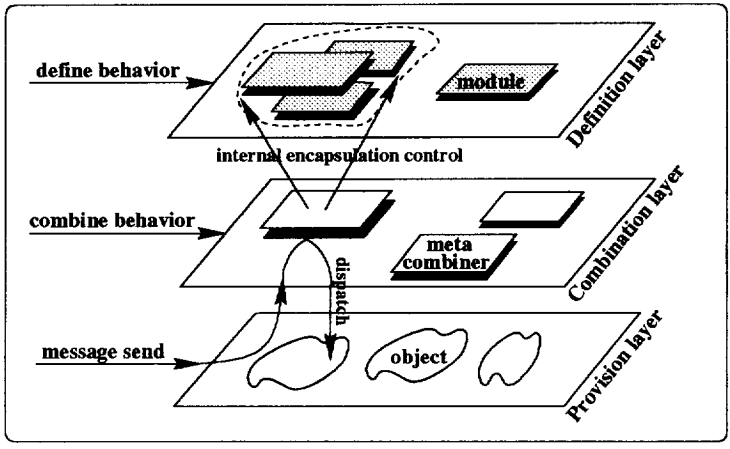

Fig. 2. The structure of the model

3.1 Behavior Definition Layer

Because of the highly complex and dynamic nature of the behavioral landscape of the objects we are concerned with, an approach to behavior definition should provide total expressiveness of the object's behavioral structure. In the proposed model the behavior definition of an object is in general dispersed among the class the object is initially generated from and a set of independent software modules, called *adjustments*. The class specifies the default (intrinsic) functionality of an object. In contrast, adjustments specify modifications of the default definition of an object, valid under special conditions. While there is always a class involved in the behavior definition of an object, the set of involved adjustments may be empty.

Adjustments are similar to mixins [5], in that the methods modeling the special behavior may refer to a parent behavior, which need not be known at the adjustment definition time and can be bound later. In order to support the special behavior, an adjustment may also declare state that is accessible only within the adjustment methods. Generally, there are three kinds of adjustments. *Specialization adjustments* specialize a class, or generally a full combined behavior definition, in a similar way to mixins. *Connection adjustments* connect two other modules by specifying how their methods are interrelated. *Class-like adjustments* are very similar to classes in that they provide full functionality (do not refer to any *super* parameter), which however does not make sense when standing alone. For illustration let us consider the following definitions for the account example presented above, written in the language *DOORS* being currently under development as a higher-level language on top of the *metaCombiner* mechanism for supporting the construction of adaptable object-oriented systems [25].

class Account *inherits* Object *def* {amount} *in*
{ credit(cr){amount := amount+cr}; debit(db){amount := amount-db; ...; }
adjustment SharedAccount *def* {} *in*
{ debit(db){self specialAction(cr); super debit(cr)}; specialAction(db){ ...}; }
adjustment ATMAccount *def* {} *in*
{ debit(db){self specialAction(cr); super debit(cr)}; specialAction(db){ ...}; }
adjustment AtomicObject *def* {conflictRelationStructure, infoStruct} *in*
{ beginTransaction(aTr){ ... }; abortTransaction(aTr){ ... } ...; }
adjustment RecordAccount *def* {} *in*
{ debit(db) {self recordTransactionInfo; super debit(db);}; ...; }
ATMAccount *specializes* Account *when* {"ATM account condition"}
SharedAccount *specializes* Account *when* {"shared account condition"}
replicas {{SharedAccount, ATMAccount} ↦ {specialAction}}
RecordAccount *connects* {Account, AtomicObject} *when* {"atomicity condition"}
alsoActivate AtomicObject

In addition to the syntactic constructs for specifying classes and adjustments, the language also provides the following expressions.

- The *specializes* and *connects* expressions specify the type of adjustments. In the example above, *SharedAccount* and *ATMAccount* are specified to be specialization adjustments for the class *Account*. *AccountRecord* is a connection adjustment for *Account* and *AtomicObject*: its redefinition e.g. for *debit* simply specifies the interconnection between *debit* in *Account* and *recordTransactionInfo* in *AtomicObject*. Finally, *AtomicObject* is a class-like adjustment: it defines full functionality in the sense that no *super* call is made. However, the synchronization and recovery functionality it provides is meaningful only in conjunction with some application functionality, e.g. that defined in *Account*.
- The *when* expression specifies the event that causes the activation of an adjustment. An adjustment is said to be activated when it becomes part of the behavior definition of an object. The activation event may be a conditional on an object's state, a certain application-level declaration, or another external event, e.g. the activation of another adjustment.
- The *alsoActivate* expression specifies adjustments to be simultaneously activated. In the example above, the activation of *RecordAccount* should be accompanied by the activation of *AtomicObject*. Thus, the behavior definition of an atomic account object is always a combination of *Account*, *RecordAccount* and *AtomicObject*.
- The *replicas* expression specifies the interaction between methods for messages with the same name in different modules. When two methods are defined for the same message by two modules that may get involved in the definition of the same object, either one incrementally modifies the other, or they represent two different meanings of the message with different validity scopes and should be handled independently. The latter are explicitly annotated as *replicas*. A method that is not explicitly specified as being a

replica but contains no *super* call, overwrites an existing method for the same message.

As the result of translating the class and adjustment specifications, class and adjustment metaobjects are created to internally represent the respective definitions. Additionally, so-called *Manager* metaobjects are automatically created and associated with each class for internally managing the information provided by the expressions above. The resulting infrastructure of the definition layer of *DOORS* is presented in detail in [24]. Since the emphasis in this paper is on the composition semantics of *DOORS* realized by the *metaCombiner* mechanism, further details about the definition layer are left out of this paper. This is also reflected by the denotational semantics in the following section, where *metaCombiner* operations are exposed at the syntactic level as metaexpressions. The following discussion assumes that class and adjustment modules, corresponding entities that contain specifications of replica-methods, and other higher-level constructs responsible for adjustment activation that avoid exposing the *metaCombiner* mechanism to the user exist. The discussion focuses only on how this mechanism handles requests for dynamic behavior combination.

3.2 Combination Layer

Despite the similarity between adjustments and mixins, there is an important semantic difference in the way the behavior they define is "bound" to that of their parent: there is no "physical" inheritance relationship between the class of an object and the adjustment that may get involved in its behavior definition over time. Dynamically "assembling" together default and special behavior is the responsibility of *metaCombiners* belonging to the combination layer. A metaCombiner is associated with each evolvable object, at instantiation time, taking responsibility for the compositional aspects of the object's behavior. Staying between an object and its dispersed behavior definition, a metaCombiner realizes some kind of connecting bridge between both, by taking over two responsibilities. First, it manages the information about how behavior definitions from different modules cooperate to yield a full behavior, in a way that provides internal encapsulation. Second, from this information it derives the environment where to evaluate the messages sent to the object. Modifying the behavior of the object is a matter of requesting the metaCombiner to update the information it encapsulates, and can thus be performed dynamically. This double role is also reflected by the definition of a *metaCombiner*, schematically shown in Fig. 3.

The *methodEnvironment* data structure encapsulated by a metacombiner plays a central role in the realization of its double functionality. Similar to method dictionaries in Smalltalk, or virtual tables in C++, this has the structure of a table with an entry for each message supported by the object. But in contrast to a class, a *metaCombiner* does not provide any behavior definition. Instead of containing the code corresponding to a message name, the method environment of the metaCombiner simply contains information about the behavioral structure of the message, as follows.

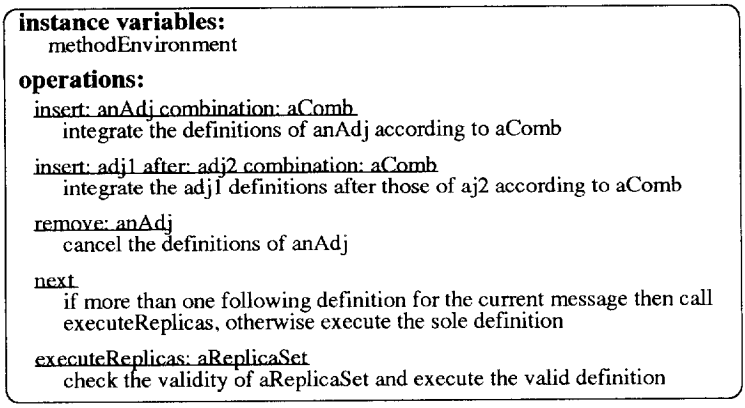

Fig. 3. The definition of a metaCombiner

In order to support internal encapsulation, the modules involved in the definition of an object are virtually – through the information stored in the method environment – grouped in *visibility scopes* individually for each message. Each replica definition, *rd*, of a message *m* has its individual visibility scope, defined as the set of modules having visibility for it, i.e. the set of modules for which *x* has the definition *rd*. Each scope has a unique identifier that is constructed successively along the alterations of the object's behavior, as discussed later. For messages with several scope-specific definitions the corresponding entry in the method environment contains one sub-entry for each scope-specific definition. This sub-entry is indexed by the corresponding scope identifier and encodes information about the set of modules jointly contributing to this scope-specific method definition, in the order in which these contributions should be executed. Messages with a single definition are a special case of those with multiple scope-specific definitions: their corresponding entry in *methodEnvironment* has a single sub-entry.

Consider, for example, the double restricted atomic account, *anAccount*, in Fig. 4. There are two sub-entries for the *specialAction* message, indexed by the scope identifiers *B*, respectively *C*. The sole elements of the order structures associated with these scope identifiers point to *SharedAccount* and *ATMAccount*, respectively. These adjustments have been marked with the labels *B*, respectively *C* in order to identify them within scope identifiers. The distribution of marks and the construction of scope identifiers will be discussed below. The scope identifier of the first sub-entry implies that only methods in *SharedAccount* have visibility for the *specialAction* definition encoded in the corresponding order structure: only the label assigned to *SharedAccount*, *B*, is included in the scope identifier. In contrast, the entries for *debit* and *recordTransactionInfo* have only one sub-entry, i.e. for both these messages there is a unique definition visible within the

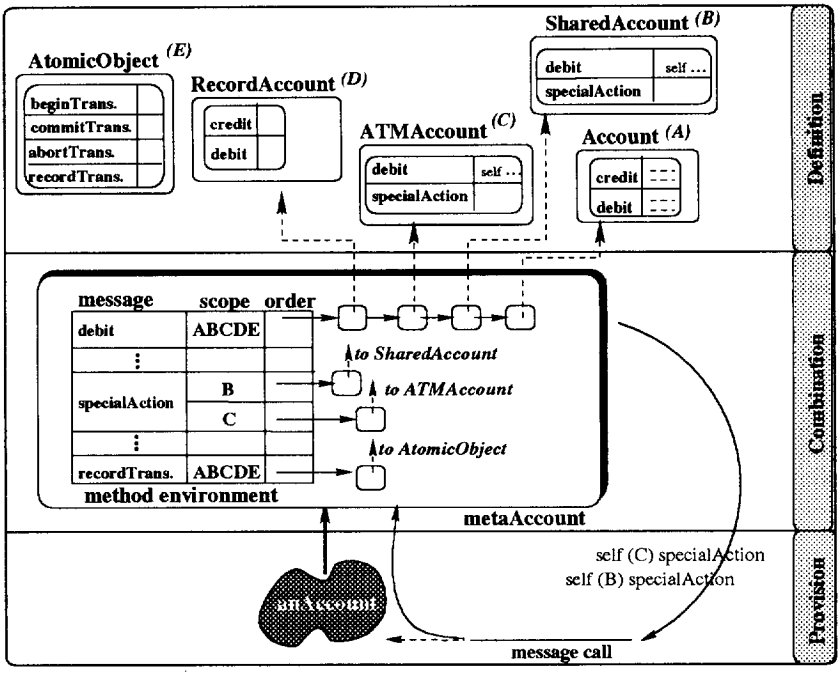

Fig. 4. The definition of an evolvable account object

scope of the entire object, as it is also reflected in the content of the respective scope identifiers.

The information in the method environment is exploited by the *dispatching functionality* (the *next* and *executeReplicas:* operations in Fig. 3). According to the information in the order structure of the *debit* entry, the information needed for the transaction validation time is gathered first by executing the implementation of *debit* in *RecordAccount*. Then, the *debit* implementations of *ATMAccount*, *SharedAccount*, and *Account* are successively executed. Additionally, *methodEnvironment* provides the information needed for deciding which of the multiple definitions of a message is valid in the context where the invocation of the message happens. When *self* invocations from the code of the adjustments already involved in the behavior definition of the object happen, only those definitions are valid whose associated scope identifier is "compatible" with (includes) the label assigned to the caller adjustment.

For illustration, let us follow the execution of *debit:* sent to *anAccount* in Fig. 4, after the implementation in *RecordAccount* has been executed. During the execution of the *SharedAccount* implementation, *specialAction* gets invoked. Although there are two different definitions for this message, only the one found

in *SharedAccount* is valid in the context of the current *self* invocation. The adjustment *ATMAccount* is not in the same scope identifier as *SharedAccount*, which is the origin of the invocation. The execution continues with the implementation for *debit* provided by *ATMAccount*, which in turn calls the *specialAction* message again. This time, the implementation of *ATMAccount* will be executed. The process ends with the execution of the implementation for *debit* in *Account*.

Intuitively, this selective execution is equivalent to equipping each *self* invocation within an evolvable object with an implicit parameter informing the dispatching process about the specific flavor, i.e. the internally encapsulated subpart of the object which the current *self* invocation stems from. In Fig. 4, this is illustrated by putting the label of the caller adjustment in brackets after the *self* variable in calls from within the *debit:* method. A precise description of how these internal encapsulation boundaries are established and taken into consideration during message dispatching is given in the denotational semantics in the next section. Currently, only methods that are not visible to clients are allowed to have multiple scope-specific definitions. However, this is not because the mechanism is principally unable to deal with other cases. It simply reflects the fact that the mechanism presented here should be considered as a framework for supporting evolving objects in a range of areas, which may have different semantics for dealing with multiple scope-specific definitions. For example, the subject making the invocation may be used as a criteria for chosing the valid scope-specific definition(s) of an interface message in a subjective environment, while "as-"constructs may be used for modeling role-specific behavior. Being modeled in an object-oriented way, the mechanism can be specialized to serve several concrete areas.

The second responsibility of a *metaCombiner* is the initialization and maintenance of the methodEnvironment information. This is initialized when the object is initially created. All entries have a single scope identifier: a special label marking class definitions, and all order structures have a single element pointing to the class. For more details about the initialization consider the instantiation semantics in the next section. Passing an adjustment component together with the request to integrate or cancel its definitions from the current behavior definition of the object corresponds to an object evolution. This is supported by *insert* and *remove* operations on a metaCombiner (the *insert:combination:*, *insert: after: combination:*, and *remove:* operations in Fig. 3), which appropriately update the method environment. Thus, the evaluation of future messages to the underlying object, accomplished by the dispatching functionality, will occur in the context of the updated behavior definition.

When an adjustment has to be integrated, an additional parameter, called a *combination rule*, is passed to the metaCombiner. A combination rule contains the messages whose corresponding methods in the adjustment should be considered as *replicas*[1]. For example, suppose that the modules in Fig. 4 are inserted

[1] While currently supporting only method combination, the mechanism can be extended to support more complex combination rules used for composing subjects [12].

in the following order: *Account*, *RecordAccount*, *AtomicObject*, *SharedAccount*. Now, in order to insert *ATMAccount*, the *insertAdj: after: combination:* message should be sent to *metaAccount* with the following parameters: *ATMAccount*, *SharedAccount* and a combination rule that annotates *specialAction* as *replica*. Based on the combination rules, the metaCombiner successively constructs the visibility scope identifiers, such that the internal encapsulation of the replica-methods is ensured, as described in the following summary of the functionality of the *insert:combination:* operation.

- First, the adjustment to be inserted is marked with a label that identifies the behavior alteration related to the current insertion within the structures of the method environment. An adjustment which is inserted more than once will have as many different labels as the number of its insertions. This allows to achieve a similar effect as the so-called *repeated inheritance* [11]. Labels are released by the remove functionality. A new element of the order structure is created for the adjustment.
- For each replica-message defined in the adjustment to be inserted, a new scope is created with the label of the adjustment as its identifier. A new sub-entry is created for the message in the method environment, associating the new scope identifier with the new order element.
- For each *refinement* definition, i.e. a non-replica definition for a message that already exists in the method environment, the label assigned to the adjustment is added into all currently existing scope identifiers which makes the method visible within all currently existing scopes. In this way the late binding of *self* is emulated. The new ordering element is added at the beginning of the order structure, thus incrementally modifying the current combined definition of the message.
- For each new method definition, i.e. a definition for a message that did not exist in the method environment, a new scope identifier is created, containing all currently distributed labels – the definition is visible within the scope of the entire object. A new entry is added into the method environment for the new message associating the created scope identifier to its definition.

The *insert: combination:* operation models the modification of the entire combined behavior of an object. The *insert: after: combination:* operation serves to modify a subpart of the combined behavior. Its functionality and that of the *remove:* operation are defined similarly. For a precise description of these operations, consider their denotational semantics in the next section.

4 Formal Description

In this section a formal description of the model will be presented, as an extension of the denotational semantics of an object-oriented language with state proposed by Hense in [13]. The language defined in [13], called *O'Small*, can be considered as a simplified Smalltalk, in that complicated constructs are eliminated, while preserving the intuitiveness of its inheritance mechanism. We prefer

to formally describe the result of integrating the *metaCombiner* mechanism into *O'Small* rather than describe its prototype implementation as an extension of Smalltalk-80, which is outlined in the next section. In this way, the comprehensive denotational semantics available for *O'Small* can be partly reused. However, concerning the composition semantics, the extended *O'Small* and the prototype implementation in the next section are essentially equivalent to each other and to the informal semantics described in the previous section.

For the definition of syntactic and semantic domains the following notation is used. The name of the domain is given on the left side of a line. The metavariable on the right side designates a member of the domain, while the text between the name and the metavariable informally describes the domain. The structure of compound domains is given between the name and the informal description, as a composition of operations on other domains. For the sake of simplicity, the following formal description focuses on the new syntactic constructs and those that have a modified semantics in the extension. The semantic of the syntactic clauses that are not affected by the extension, such as the program clause (P), the clauses for compound expressions (C), variable definitions (V), as well as some of the expression clauses (E), is left out of this description. However, since these are standard constructs that are supported by (almost) any object-oriented language, we assume that the intuitive understanding of their semantics should suffice to read the semantics of the new and modified constructs; refer to [13] for an exact definition.

Syntactic domains

Ide	the domain of identifiers	I
Bas	the domain of basic constants	B
Pro	the domain of programs	P
Exp	the domain of expressions	E
CExp	the domain of compound expressions (commands)	C
MExp	the domain of metaexpressions	ME
Var	the domain of variable declarations	V
Meth	the domain of method declarations	M
Cls	the domain of class definitions	K
Adj	the domain of adjustment definitions	A
BDef	the domain of behavior definition modules	BD

Syntactic clauses

P	::=	BD C
BD	::=	K | A | BD_1 BD_2 | ϵ
K	::=	class I_1 inherits I_2 def V in M | I_1 combine I_2 I_3 rep I_4
A	::=	adjustment I def V in M
ME	::=	evolve E | I_1 insert A rep I_2 | I_1 insert A_1 after A_2 rep I_2 | I remove A
E	::=	ME | B | true | false | I | E.I(E_1,...,E_n) | new E | newEvol E
C	::=	E | I := E | if E then C_1 else C_2 | while E do C | C_1;C_2
V	::=	var I := E | V_1 V_2 | ϵ
M	::=	meth I(I_1,...,I_n) C | M_1M_2 | ϵ

Semantic domains

Lab		adjustment labels	lb
Unit		the one-point-domain	u
Bool		the domain of booleans	b
Loc		the domain of locations	l
Bv		the domain of basic values	e
Scope	$= \mathcal{P}(\text{Lab})$	method scopes	sc
$\text{Record}_{\alpha,\beta}$	$= \alpha \mapsto [\beta + \bot]$	records	
Env	$= \text{Record}_{Ide,Dv}$	environments	r
CombEnv	$= \text{Record}_{\{se,gen\},Dv}$	combiner environments	r_{mc}
Object	$= \text{Record}_{Ide,Dv}$	objects	o
ST_α	$= \text{Store} \to [\alpha \times \text{Store}]$	state transformer values	x
Store	$= \text{Record}_{Loc,Sv}$	stores	s
Dv	$= \text{Loc} + \text{Rv} + \text{Method}_n + \text{Class}$ $+ \text{GenList} + \text{SMethod}_n + \text{Adjust}$	denotable values	d
Sv	$= \text{File} + \text{Rv}$	storable values	v
Rv	$= \text{Unit} + \text{Bool} + \text{Bv} + \text{Object}$	R-values	v
File	$= \text{Rv}^*$	files	i
Method_n	$= \text{Dv}^n \mapsto \text{ST}_{Dv}$	method values	m
SMethod_n	$= \text{Dv} \to [\text{Dv}^n \to \text{Record}_{Scope,ST_{Dv}}]$	scoped method values	sm
Class	$= \text{ST}_{Object} \to \text{ST}_{Object}$	class values	c
Adjust	$= \text{ST}_{Object} \to \text{Class}$	adjustment values	a
Comb	$= \text{ST}_{CombEnv} \to \text{ST}_{CombEnv}$	metacombiner values	mc
GenList	$= \text{Adjust}^* \times \text{Class}$	generator lists	g

Semantic functions

$[\![]\!]_K$:	$\text{Cls} \to \text{Env} \to \text{ST}_{Env}$
$[\![]\!]_A$:	$\text{Adj} \to \text{Env} \to \text{ST}_{Env}$
$[\![]\!]_{SM}$, $[\![]\!]_M$:	$\text{Meth} \to \text{Env} \to \text{Env}$
$[\![]\!]_E$ $[\![]\!]_R$:	$\text{Exp} \to \text{Env} \to \text{ST}_{Dv}$
$[\![]\!]_{ME}$:	$\text{MExp} \to \text{Env} \to \text{ST}_{Dv}$

Consider the class declaration function in Fig. 5. Similar to the standard semantics as defined in [13], the x_{super} parameter is bound at the class declaration time. However, in contrast to the standard semantics, the evaluation of a class declaration is performed by two different functions in Fig. 5, depending on the value of b. This parameter is bound at the object creation time, as indicated by the functions for instance creation in Fig. 9. It is true when evolvable objects have to be created, and false otherwise. When b is false, the semantic definition remains the same as the standard one in [13] (the second choice of the conditional). The x_{self} parameter is the state transformer returned by applying the fix-point operator to the class, as indicated by the function for *"new E"* in Fig. 9. Applying x_{self} to the creation time store, s_{create}, returns the method environment to which the *self* parameter of the object being created (referred to within M) is definitively bound, leaving the store unchanged.

If b is true, however, the state transformer passed as parameter, x_{mc}, returns a combiner-environment after being applied to the store (r_{mc}). As indicated by

$[\![\text{class } I_1 \text{ inherits } I_2 \text{ defines V in M}]\!]_K r = [\![I_2]\!]_E$ * Class? * $\lambda c.\text{result}[I_1 \mapsto w \boxed{\triangleright} c]$,

$$w = \lambda x_{super}.\lambda b.\text{cond}(\lambda x_{mc}.\lambda s_{create}.[\![M]\!]_M(\begin{bmatrix} self & \mapsto (r_{mc}\ se) \\ super & \mapsto r_{super} \end{bmatrix} \oplus r_{local} \oplus r),$$

$$\lambda x_{self}.\lambda s_{create}.[\![M]\!]_M(\begin{bmatrix} self & \mapsto r_{self} \\ super & \mapsto r_{super} \end{bmatrix} \oplus r_{local} \oplus r))$$

$(r_{super}, s_{super}) = (x_{super}\ s_{create})$ $\quad (r_{self}, -) = (x_{self}\ s_{create})$
$(r_{local}, s_{new}) = ([\![V]\!]_V r\ s_{super})$ $\quad (r_{mc}, -) = (x_{mc}\ s_{create})$

The auxiliary functions used above are defined as follows:

The generic function * is defined by Hense for the composition of commands and declarations, as follows. Let f and g be two functions with the following types:
$f : \left\langle \begin{matrix} \text{Store} \\ D_1 \mapsto \text{Store} \end{matrix} \right\rangle \to [D_2 \times \text{Store}]$, $\quad g : D_2 \to \text{Store} \to [D_3 \times \text{Store}]$
The lines in braces represent alternatives. The alternatives below depend on the choices of the alternatives above. If the upper/lower alternative of a brace above has been chosen, the upper/lower alternative in every brace below has to be chosen, as well. The composition of f and g is defined, as follows:
$f*g : \left\langle \begin{matrix} \text{Store} \\ D_1 \mapsto \text{Store} \end{matrix} \right\rangle \to [D_3 \times \text{Store}]$, $\quad f*g = \left\langle \begin{matrix} \lambda s_1 \\ \lambda d_1.\lambda s_1 \end{matrix} \right\rangle \cdot \begin{cases} (\bot, s_2), \text{ if } s_2 \text{ err} \\ g\ d_2\ s_2, \text{ otherwise} \end{cases}$
where $(d_2, s_2) = \left\langle \begin{matrix} f\ s_1 \\ f\ d_1 s_1 \end{matrix} \right\rangle$

result: $D \to \text{Store} \to [D \times \text{Store}]$, $\quad$ result $d = \lambda s.(d, s)$

cond: $[D \times D] \to \text{Bool} \to D$, cond(d1, d2) $= \lambda b.b \mapsto d1,d2$

D? : D' $\to$ Store $\to$ [D' $\times$ Store], $D \subseteq D'$, $D? = \lambda d.\begin{cases} result\ d\ d \in D \\ error \quad \text{otherwise} \end{cases}$

$\oplus$, $\boxed{\triangleright}$ are the left-preferential record combination operator, respectively the inheritance operator as defined in [9]

Fig. 5. Class declaration

the semantic function for creating evolvable objects (*"newEvol E"* in Fig. 9), x_{mc} is the result of evaluating an implicit *"evolve E"* metaexpression. The function for this metaexpression, given in Fig. 6, creates a new combiner-environment as follows. First, the fix-point operator is applied to the non-evolvable version of the class to be instantiated, (c') and a binding for the pseudo-variable *flav* is appended to the result. The method environment gained in this way is bound as the value of *se* in the combiner-environment being created. When present in a method environment, the pseudo-variable *flav* designates the module which methods in the environment belong to. As it can be also noticed in the adjustment declaration function in Fig. 7, all environments used to bind *self* within

methods of an evolvable object do contain a binding for *flav*. This models the implicit parametrization of *self*-calls used to indicate the internal "encapsulated subobject" which a certain (*self*) call stems from. As discussed in the previous section and indicated by the semantic function for method invocation below, this information is used to dispatch the invocation. The *gen* entry of the created combiner-environment is bound to a transformation of *se* accomplished by the *scoped* function. This function stamps all methods in *se* as being visible within the default scope (the special label *def*).

$$[\![\text{evolve E}]\!]_{ME}\text{r} = [\![\text{E}]\!]_{E}\text{r} * \lambda\text{c.(c false)} * \text{Class?} * \lambda\text{c'.(result comb)},$$

$$\text{comb} = \lambda\text{x.}\lambda\text{s.}\begin{bmatrix} gen \mapsto \lambda x.\text{result(scoped(se, } def\text{))} \\ se \mapsto [flav \mapsto def] \oplus \text{r}_{self} \end{bmatrix}, \ (\text{r}_{self}, \text{-}) = \text{Fix(c')s}$$

$$\text{scoped: Env} \times \text{Lab} \to \text{Env}, \quad \text{scoped(r, lb)(x)} = [\text{lb} \mapsto (\text{r x})]$$

def is a special label used to mark the default definition

Fig. 6. Metaexpressions (1)

The combiner-environment constructed in this way is passed (indirectly through *"newEvol"*) as a parameter to the class declaration function (r_{mc} in Fig. 5). As it can be seen in Fig. 5, when an evolvable object is created ($b = true$), instead of binding the *self* parameter within M definitively to a particular method environment, as in the non-evolvable case, it is bound to the *se* variable within the combiner environment. This indirection, which is the key to dynamic behavior evolution, is schematically presented in Fig.11. Before going on with the semantic function for adjustment declaration, notice that the semantic function for the *"I_1 combine I_2 with I_3 rep I_4"* clause, which enables the use of the metaCombiner mechanism to statically compose behavior definition modules, is omitted in Fig. 5, since this clause is equivalent to the sequence: I_1 *:= evolve* I_2*;* I_1 *insert* I_3 *rep* I_4.

Similar to wrappers used to model mixins in [13], the evaluation of adjustment declarations in Fig.7 results in a function of the *super* and *self* parameters. However, it differs from wrapper functions, as follows. There are two additional parameters: the label *lb*, and the combination rule *cr*, both to be bound at insertion time, as indicated by the functions in Fig. 10. *lb* is further passed as a parameter to $[\![\text{M}]\!]_{SM}$, where it will be used to construct scope identifiers. The function $[\![]\!]_{SM}$ (Fig. 8) creates so-called scoped method environments encapsulated by a metaCombiner, as informally described in the previous section. Scoped method environments differ from standard ones created by $M[\![]\!]$ in that they expect an additional parameter: the *super* scoped method environment, $r_{superMeth}$. The new scoped environment for a message I is created as follows. For each association (sc_i, m_i) in $r_{superMeth}$, (a) the new scope identifier is created by appending *lb* to sc_i, and (b) C is evaluated with the *super* parameter (that may appear free in it) bound to m_i.

$[\![\text{adjustment I defines V in M }]\!]_A \mathrm{r} = \text{result}[\mathrm{I} \mapsto \mathrm{a}]$, where:

$$\mathrm{a} = \lambda \mathrm{lb}.\lambda \mathrm{cr}.\lambda \mathrm{x}_{super}.\lambda \mathrm{x}_{self}.\lambda \mathrm{s}_{insert}.((\mathrm{a}_{ref} \oplus \mathrm{a}_{rep}) \boxed{\oplus} \mathrm{r}_{super}, \mathrm{lb})$$

$$\mathrm{a}_{ref}(\mathrm{x}) = \begin{cases} r_{meth}(x) \boxdot r_{super}(x) & \mathrm{x} \in \mathrm{dom}(\mathrm{r}_{meth}) \cap \mathrm{dom}(\mathrm{r}_{super}) \cap \overline{dom}(\mathrm{cr}) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathrm{a}_{rep}(\mathrm{x}) = \begin{cases} r_{meth}(x) \boxed{\oplus} r_{super}(x) & \mathrm{x} \in \mathrm{dom}(\mathrm{r}_{meth}) \cap \mathrm{dom}(\mathrm{r}_{super}) \cap \mathrm{dom}(\mathrm{cr}) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathrm{r}_{meth} = [\![\mathrm{M}]\!]_{SM}([\text{self} \mapsto ([flav \mapsto \mathrm{lb}] \oplus \mathrm{r}_{self})] \oplus \mathrm{r}_{loc} \oplus \mathrm{r})\ \mathrm{lb}$$
$$(\mathrm{r}_{loc}, \text{-}) = ([\![\mathrm{V}]\!]\mathrm{r}\ \mathrm{s}_{insert}),\ (\mathrm{r}_{self}, \text{-}) = (\mathrm{x}_{self}\ \mathrm{s}_{insert}),\ (\mathrm{r}_{super}, \text{-}) = (\mathrm{x}_{super}, \mathrm{s}_{insert})$$

$\boxdot$, $\boxed{\oplus}$ are the application operation, respectively the distributive version of the left-preferential record combination operator as defined in [9]

Fig. 7. Adjustment definition functions

$$[\![\text{meth I}(\mathrm{I}_1, \ldots, \mathrm{I}_n)\ \mathrm{C}]\!]_{SM}\mathrm{r} =$$
$$\lambda \mathrm{lb}.\left[I \mapsto \lambda \mathrm{r}_{supMeth}.\lambda \mathrm{d}_1.\ \ldots\ \lambda \mathrm{d}_n.\left[\begin{matrix} sc'_1 \mapsto [\![\mathrm{C}]\!]\ (\mathrm{r}_{arg} \oplus [super \mapsto \mathrm{m}_1] \oplus \mathrm{r}) \\ \ldots \\ sc'_k \mapsto [\![\mathrm{C}]\!]\ (\mathrm{r}_{arg} \oplus [super \mapsto \mathrm{m}_k] \oplus \mathrm{r}) \end{matrix} \right] \right],$$
$$\mathrm{r}_{arg} = \left[\begin{matrix} I_1 \mapsto \mathrm{d}_1 \\ \ldots \\ I_n \mapsto \mathrm{d}_n \end{matrix} \right],\ \mathrm{sc'}_i = \mathrm{lb} \cup \mathrm{sc}_i,\ \mathrm{sc}_i \in \mathrm{dom}(\mathrm{r}_{supMeth}),\ \mathrm{m}_i = (\mathrm{r}_{supMeth}\ \mathrm{sc}_i)$$
$$[\![\mathrm{M}_1\ \mathrm{M}_2]\!]_{SM}\mathrm{r} = ([\![\mathrm{M}_2]\!]_{SM}\mathrm{r}) \oplus ([\![\mathrm{M}_1]\!]_{SM}\mathrm{r})$$
$$[\![\varepsilon]\!]_{SM}\mathrm{r} = [\,]$$
$$[\![\text{meth I}(\mathrm{I}_1, \ldots, \mathrm{I}_n)\ \mathrm{C}]\!]_M\mathrm{r} = \left[I \mapsto \lambda \mathrm{d}_1.\ \ldots\ \lambda \mathrm{d}_n.\ [\![\mathrm{C}]\!]\ \left(\left[\begin{matrix} I_1 \mapsto \mathrm{d}_1 \\ \ldots \\ I_n \mapsto \mathrm{d}_n \end{matrix} \right] \oplus \mathrm{r} \right) \right]$$
$$[\![\mathrm{M}_1 \mathrm{M}_2]\!]_M\mathrm{r} = ([\![\mathrm{M}_2]\!]\mathrm{r}) \oplus ([\![\mathrm{M}_1]\!]\mathrm{r})$$
$$[\![\varepsilon]\!]_M\mathrm{r} = [\,]$$

Fig. 8. Method definition functions

The *cr* parameter (Fig. 7) controls the integration of the method environment resulting from $[\![\mathrm{M}]\!]_{SM}$ with r_{super}. For non-replica methods, i.e. $x \in \overline{dom}(cr)$, the *super* method, $r_{super}(x)$, is applied to the function bound to x in the environment returned by $[\![\mathrm{M}]\!]_{SM}$. In contrast, no *super* parameter is applied to the

replica-methods ($x \in dom(cr)$). Thus, in contrast to mixin-wrappers the application of the *super* parameter has been shifted to the method level. Notice that the *self* environment of adjustment methods (see the binding for *self* in the environment where $[\![M]\!]_{SM}$ is evaluated, in Fig 7) binds the pseudo-variable *flav* to the adjustment label *lb*, marking the methods with the label of the adjustment they belong to. The rest of the *self* environment consists of r_{self}. This results from applying the fix-point operator to the generator entry (*gen*) of a metaCombiner after the adjustment get inserted into it, as indicated by the functions in Fig. 10.

After all its parameters are bound, the adjustment declaration function returns a pair consisting of a scoped method environment and the assigned label. The method environment is a combination of the new method environment derived from the adjustment definitions, $a_{ref} \oplus a_{rep}$, with the *super* method environment, r_{super}, whereby the former overwrites the latter. Before leaving the semantics of the behavior definition constructs (K,A), let us briefly consider the evaluation of variable definitions. Both definitions for class and adjustment declarations in Fig. 5 and 7 share the non-recursive allocation of the instance variable environment. With respect to the visibility of instance variables, strong encapsulation is exploited: in both cases only r_{local} is used for the evaluation of method definitions.

$[\![\text{new E}]\!]_E\text{r} = [\![\text{E}]\!]\text{r} * \lambda\text{c.(c false)} * \text{Class?} * \lambda\text{c'.}\lambda\text{s.(Fix(c'))s}$

$[\![\text{newEvol E}]\!]_E\text{r} = [\![\text{E}]\!] * \lambda\text{c.(c true)} * \text{Class?} * \lambda\text{c'.}\lambda\text{s.(c' } x_{mc}\text{ s)},$
$\quad x_{mc} = \text{Fix}([\![\text{evolve E}]\!]_{ME}\text{r})$

$[\![\text{E.I(E}_1,...,\text{E}_n)]\!]_E\text{r} = [\![\text{E}]\!]_R\text{r} * \text{Object?} * \lambda\text{o.(EObj o)} * \text{cond((match d } flavor\text{), d)}$
$\quad * \text{Method?} * \lambda\text{m.}[\![\text{E}_1]\!]_R\text{r} * \lambda\text{d}_1\text{. ....}[\![\text{E}_n]\!]_R\text{r} * \lambda\text{d}_n\text{.m(d}_1\text{, ...,d}_n\text{)},$

$\quad flavor = \text{(o } flav\text{)}, \quad \text{d} = \text{(o I)}, \quad \text{EObj} = \lambda\text{o.}\begin{cases} false\ \text{(o } flav\text{)} = \bot \\ true\ \text{otherwise} \end{cases}$

$\quad \text{match: Env}\times\text{Lab} \to \text{Method}_n,\ \text{match} = \lambda\text{r.}\lambda\text{lb.(r sc)},\ \exists\ \text{sc} \in \text{dom(r)},\ \text{lb} \in \text{sc}$

$[\![\text{E}]\!]_R\text{r} = [\![\text{E}]\!]_E\text{r} * \text{deref} * \text{Rv?},$

$\quad \text{deref: Dv} \to \text{Store} \to [\text{Dv} \times \text{Store}], \quad \text{deref} = \lambda\text{e.}\begin{cases} cont\ e\ \text{e} \in \text{Loc} \\ error\ \text{otherwise} \end{cases}$

Fig. 9. Semantics of instantiation and message invocation

As shown in Fig. 9, the creation of non-evolvable objects (*"new E"*) remains essentially the same, except that the boolean parameter of the class function is first bound to false. In the case of evolvable objects (*"newEvol E"*), a new combiner environment (x_{mc}) is created, as already discussed above, and passed as a parameter to the class function. As already discussed above and illustrated in Fig. 11, the *self* parameter within the class declaration function (Fig. 5) will be

bound to the *se* entry of the combiner parameter, i.e. *self* is not directly bound to a method environment, but instead to a variable that contains a method environment. The content of this variable is modified as adjustments are inserted or removed, as shown by the semantic functions for metaexpressions in Fig. 10.

$$[\![I_1 \text{ insert A rep } I_2]\!]_{ME}r = [\![I_1]\!]_E * \text{Comb?} * \lambda mc.[\![A]\!]_A * \lambda a.[\![I_2]\!]_E * \lambda cr.\text{newLabel} * \lambda lb.(\lambda x.\lambda s.(\begin{bmatrix} gen \mapsto gen_1 \\ se \mapsto rs_1 \end{bmatrix} \oplus r_{mc}))$$

$$[\![I_1 \text{ insert } A_1 \text{ after } A_2 \text{ rep } I_2]\!]_{ME}r = [\![I_1]\!]_E * \text{Comb?} * \lambda mc.[\![A_1]\!]_A * \lambda a_1. [\![A_2]\!]_A * \lambda a_2.[\![I_2]\!]_E * \lambda cr.\text{newLabel} * \lambda lb.(\lambda x.\lambda s.(\begin{bmatrix} gen \mapsto gen_2 \\ se \mapsto rs_2 \end{bmatrix} \oplus r_{mc}))$$

$$[\![I \text{ remove } A]\!]_{ME}r = [\![I]\!]_E * \text{Comb?} * \lambda mc.[\![A]\!]_A * \lambda a_3.(\lambda x.\lambda s.(\begin{bmatrix} gen \mapsto gen_3 \\ se \mapsto rs_3 \end{bmatrix} \oplus r_{mc}))$$

$(r_{mc}, -) = (\text{Fix}(mc))s$, $gen_0 = (r_{mc}\ gen)$,
$gen_1 = \text{Extend}(gen_0, (a\ lb\ cr))$ $gen_2 = \text{AfterExtend}(gen_0, (a_1\ lb\ cr), a_2)$,
$gen_3 = \text{Extract}(gen_0, \pi_2(a_3))$, $rs_i = \text{Fix}(\text{GenComp}\ gen_i)s$ (i = 1,2,3),
Extend: GenList → Adjust → GenList, $\text{Extend} = \lambda gen.\lambda a.(a, gen)$,
GenComp: GenList → Class,

$$\text{GenComp} = \lambda gen.\begin{cases} gen & \text{if (sz gen)} = 1 \\ comp(\pi_1(hd\ gen), GenComp(tl\ gen)), & \text{otherwise} \end{cases}$$

comp: Adjust × Class → Class, $\text{comp} = \lambda a.\lambda c.(a \boxdot \text{result}(c))$
The auxilairy functions *sz*, *hd*, and *tl* return the size, the head, and the tail of a list, respectively. π_i (i = 1,2) are the projection functions

Fig. 10. Meta-expression functions (2)

Let us consider the function for the first insert clause in Fig. 10. First, the current generator list, gen_0, is updated by *Extend*. Within *Extend*, the label generated by the auxiliary function *newLabel* (*lb*) along with the combination rule (*cr*) are passed as parameters to the result of evaluating the adjustment to be inserted, *a*. This results in a pair, containing the scoped method environment of the adjustment – a function of *super* and *self* – and the generated label, as already shown in Fig.7. The *Extend* function adds this pair into the current generator list, yielding the new generator list gen_1 to be bound to *gen*. The generator entity of the combiner has been modeled as a list, in order to keep the function for the remove clause simple. After the binding for *gen* is modified, the *GenComp* function recursively bounds the *super* parameters of the adjustments in the list. The fix-point application binds the *self* parameter of the resulting chain yielding the method environment (rs_i) which becomes the new value of *se*. The semantic functions for the other two clauses are defined in a similar way: *AfterExtend* adds the adjustment method environment at a certain position into the list, while *Extract* removes it from the list.

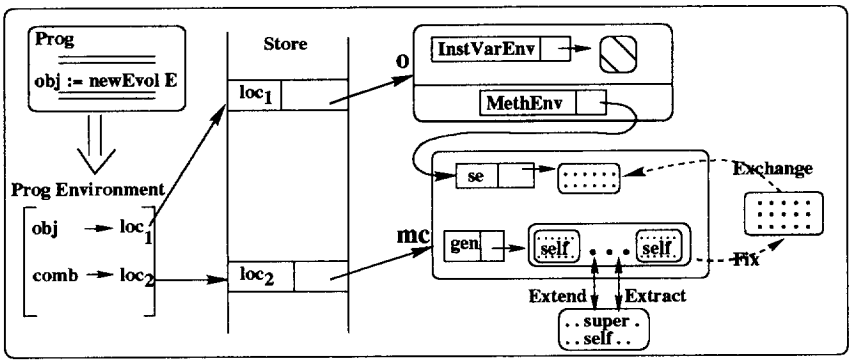

Fig. 11. Indirect binding of self

5 Implementation Issues

The proposed model has been prototypically implemented as a metaextension of the standard Smalltalk image. In this section we will briefly present how the mechanism is integrated into the standard Smalltalk system. Additionally, we will show how the metalevel facilities of Smalltalk are exploited to keep the overhead related to the metalevel indirection to a minimum.

The functionality of the *metaCombiner* is implemented as a subclass of *Class*, which models the behavior of class objects in Smalltalk-80 [10]. Thus, a *metaCombiner* is a class which additionally supplies combination and explicit dispatching functionality. A new method for creating evolvable objects is added on the instance creation functionality of classes. Instead of creating an instance of the receiver class, this new method creates an empty *metaCombiner*, makes this a subclass of the receiver class, and finally instantiates the created *metaCombiner*. Fig. 12 illustrates the result of creating a new object *Cl-object* as an instance of the class *CL*.

By making the *metaCombiner* the class of the object, it automatically becomes the default place where the dispatching of the messages to the object starts. However, its explicit dispatcher functionality remains inactivated as long as no adjustment is inserted into it. Since the automatically created *metaCombiner* is an empty subclass of the original class, all messages are further dispatched to the original class. This remains valid for the unmodified messages also after adjustments are inserted into the *metaCombiner*, since only modified messages are inserted into its method dictionary, which plays the method environment role. It is evident that there is no extra overhead for these messages. Adjustments are implemented as a special kind of of classes which provide only instance-template functionality. Instead of using the pseudo-variable *super* to

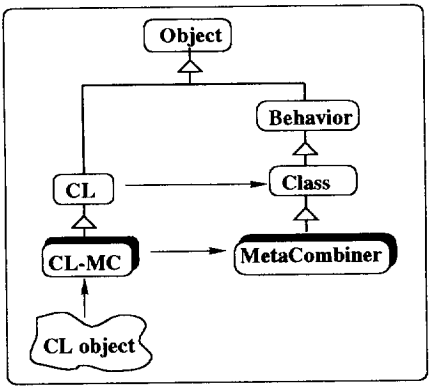

Fig. 12. Creating a new object

invoke the method they refine, adjustment methods make a special call, *self class next*. This invokes the dispatching functionality (*next* operation) of the *metaCombiner* – the class of *self* in this call – instead of the implicit dispatching functionality. The lookup part of the dispatching functionality makes use of the order structure associated with each message in the method dictionary of a *metaCombiner*. Since this is implemented as a list of pointers to those inserted modules that contain a method for the message in the order of their execution, there is no additional overhead related to method lookup. In order to keep the potential overhead related to the execution of the method returned by the lookup part small, we make use of the reflective facilities of Smalltalk-80 [10], which allow the reification of the interpretation context chain by means of the pseudo-variable *thisContext*.

From the discussion above, the execution of a modified message m results in a sequence of pairs, EP_i, of the form: *(application-code, dispatch-code)*, where application-code denotes the implementation of the message provided by the module i in the order structure and dispatch-code is the code for the explicit dispatch operation, *next*, which may get invoked within application-code. In order to illustrate the use of the context reification facility, consider the execution of a certain pair, EP_i. Let *code-context* and *dispatch-context* be the context where the execution of EP_i.*application-code*, and EP_i.*dispatch-code* occurs, respectively. The interpretation steps of m are given below.

1. The EP_i.*application-code* is executed in *code-context*.
2. This execution may invoke the dispatch operation *next*; in this case the interpreter opens a new context, the *dispatch-context*.
3. The execution of EP_i.*dispatch-code* will be performed in *dispatch-context*, as follows. First, the sub-definition of the message following the current application-code will be found in the order structure, and the sender context

(*thisContext sender*) will be reinitialized with it. It should be noticed that the sender of *thisContext* is *code-context*, where the interpretation continues after the execution of EP_i.*dispatch-code*.

4. At this point, the execution of EP_i.*dispatch-code* has finished, *dispatch-context* gets closed, and the interpretation process continues with step 1 for the next execution pair.

Except for step (3), all other steps are parts of the interpretation process of the virtual machine. Step (3) represents our explicit intervention point in this process. The interpretation process described above ends with the execution of a method which does not invoke *next* calls, or with the method provided by the class of the object. The latter does not invoke any explicit dispatching operation. Instead, it may invoke *super* calls. At this point, the dispatching of the modified part is finished and the dispatching of the *super* call follows the original class inheritance chain; consequently, it cannot be the source of additional overhead. In the discussion above we have implicitly assumed that the lookup part of the dispatching returns only one successor method at each execution step. The execution of the replicas-methods is realized in a similar way.

6 Discussion and Related Work

In this section we discuss the properties of the proposed model and show how the additional abstraction provided by the explicit combination layer enables the orthogonalization of the conflicting issues: dynamicity versus encapsulation of the client interface, and incremental modification versus internal data encapsulation. In both cases, related work is presented, too.

6.1 Encapsulated Object Modification

The object composition provided by the *metaCombiner* mechanism satisfies the requirement for dynamic composition. An object's behavior can be modified quite naturally after it has been created, by sending an insert/remove request to the object's metaCombiner. In this way, our model is superior to other approaches providing a restricted form of object modification, like the mixin-methods approach [35]. By allowing both the insertion and cancellation of adjustments, our solution satisfies the requirement for behavior alterations that may remain valid under certain conditions. The dispatcher role of the metaCombiner enables modifications to be performed transparently, but nevertheless they immediately affect an object's future behavior.

Despite object modification, the encapsulation of the client interface is preserved in our proposal. In [35], Steyaert et al. have formulated the *immaculate client interface* design principle, which states that an object should expose only the client interface to its message passing clients, and hide knowledge about how the object can be modified from them. Objects in our model satisfy the immaculate client interface principle. They remain records just like in the class-based

model and expose only the client interface to their clients. As it can be verified by the semantic functions of the previous section, modifications of the *self* environment happen in conjunction with the creation of the object and adjustment insertion/removal. In contrast to object-based inheritance, message passing involves neither wrapping nor fixing.

In [35] Steyaert et al. identify the provision of explicit constructs for behavior generators, like classes in Smalltalk, as a potential source for violating encapsulation. We only partly agree with them in this point. Their assertion certainly holds for object-based systems, where such explicit generator constructs are the only existing abstraction. In our model, *metaCombiners* are explicit entities responsible for modifying the behavior of the underlying object. The increased power provided by such explicit constructs is indeed related to a certain danger if their availability is not restricted. As already mentioned, the main motivation behind the design of our approach is to support the design of metaarchitectures for adaptable systems. Since it is the system itself that controls its own adaptability, it is also the only client that uses metaCombiners in order to dynamically adapt the behavior of the objects it is composed of. Additionally, the functionality of the *metaCombiners* can be orthogonally enhanced by supplying suitable tools for enhancing the reliability, by restricting the availability of their explicit combination functionality only to certain "privileged" users. Another extension could be concerned with providing explicit tools for controlling which kind of modifications are allowed.

The *predicate objects* approach proposed by Chambers [8] is similar to our approach in that it provides language support to allow the modification of an object's behavior to reflect changes in the object's state. In this approach the implementation of an object is factored into a group of state-specific prototypes. The choice of a particular prototype behavior is based on predicates over the state of the object. In this way, predicate objects avoid the drawbacks of representing state (a) as data, resulting in the case-like style of programming, or (b) through subclasses, resulting in the combinatorial explosion problem. While serving a similar purpose, our proposal is more general. The *metaCombiner* approach can emulate the state-specific behavior modeled by the predicate objects. However, except for internal state conditioned modifications, it is able to deal with changing the implementation of an object because of some external conditions in the environment this object operates on, as it is required by adaptive systems, or for extending the functionality of already existing persistent objects. Furthermore, in contrast to the predicate objects approach, in our approach automatic method combination is provided, and the set of supported behavior modifications need not be fixed at the object's creation time. With respect to the last aspect, predicate objects are more similar to the *mixin-methods* approach. Last but not least, predicate objects support state-related object modification in the prototype-based language *Cecil*. In contrast, our model is intended to enable object behavior modification in class-based languages.

The *metaCombiner* model is also related to the *composition filters* model of the language *Sina* [1]. Both models aim at a modular extension of the conven-

tional object model to facilitate the construction of large-scale complex systems which support multiple application requirements in an extensible way. The *meta-Combiner* approach seems to simulate the same data abstraction techniques as those supported by composition filters, however with a simpler object model. The roles played by different elements of composition filters, such as *internals*, and *filters*, are uniformly modeled by adjustments in our proposal. The inheritance data abstraction supported by the *dispatch filter* in Sina is an object-based one. An object encapsulates an internal object for each of its behavioral flavors. This results in an increased number of objects in the system, and more importantly may cause the duplication of the attributes of "parent" objects that are inherited through two different internals. Sina requires the programmer to avoid inconsistencies related to this duplication. Since the internals participating in the behavior of an object should be fixed at the class definition time, it is not possible to extend the behavior of existing persistent objects. Additionally, the *metaCombiner* approach provides a richer semantics for method combination. The composition filters object model does not support incremental combination of the implementations of a single message provided by different internals. The default strategy for solving name collisions is an explicit ordering of the involved internal objects in the dispatch filter which very much resembles the linearization approaches to multiple inheritance.

Metaobject protocols [15], also support some kind of object modification. The main idea underlying these systems is that certain aspects of an object's behavior are put under the control of metaobjects which communicate among each other by means of message passing. When a certain aspect becomes "activated", a request is sent to the corresponding (handler) metaobject to take care of it. Modification of a behavioral aspect is provided by exchanging the corresponding metaobject. For example, two different object models for concurrent and distributed objects are modeled in the CodA architecture [20]. An object can become concurrent by attaching a concurrent model as its meta. However, combining two overlapping object models (containing metaobjects for the same aspect), such as the concurrent and distributed object model, requires programmer intervention [20]. This is due to the modeling of behavioral aspects by means of objects. Modifying a behavioral aspect now requires modification of the metaobject. The problem is thus simply shifted to the metaobject level, but not eliminated.

In contrast, in our model different aspects are modeled by software modules which are descriptive entities that can be combined more easily than objects. The single metaobject associated with an object does not deal with a specific aspect, but instead with the combination of aspect descriptions. In this respect, our model can be considered as a special case of the *aspect-oriented programming* paradigm [17]. An aspect is a self-contained subprogram that describes a global property of a program. Each application domain might have a different set of aspects, such as synchronization and distribution aspects of client/server applications. Aspects are automatically combined into executable code by a new kind of compiler called *Aspect Weaver*. Thus, our model can be seen as apply-

ing the aspect-oriented decomposition principle to the design of object-oriented systems. Our model is a special case in that all aspects are described in the same language, while one of the goals of the aspect-oriented approach is to allow different aspects to be described in different languages. In contrast to aspect-oriented programming which emphasizes the separation of concerns beyond the functional modularization boundary, the emphasis of our approach is on dynamic (re)weaving.

Recently, Seiter et al. [30] proposed the context relationship between classes in order to achieve evolution of object behavior in class-based systems. The basic idea is that if class C is context-related to a base class B, then B-objects can get their functionality dynamically altered by the presence of C-objects. In general, a context class contains method updates for several base classes. A context object may be explicitly attached to a base object, or it may be attached to a method invocation, in which case it is implicitly attached to a set of base objects involved in the method invocation. The context relation supports behavior evolution because the C-object attached to the B-object may vary at run-time. More importantly, the evolution is not based on aggregation as with metaobject protocols, but on method environment update. This is similar to our approach. However, the evolution achieved by the context attachment is more restricted as compared to our approach. Although there is some evidence in the paper about the advantage of supporting incremental attachement to allow multiple implementations to be executed for the same method, only overriding is supported at the semantic level [29]. Consequently, there is no support for more general modifications, as illustrated by a shared and ATM-account.

6.2 Internal Encapsulation

While the emphasis of this paper has been on the dynamic behavior alteration, it is obvious that the *metaCombiner* mechanism can be applied to static behavior composition as well, in this case being a more expressive alternative to the conventional static inheritance. This is also reflected in the denotational semantics presented earlier in this paper where the syntactic clause $I_1 \, combine \; I_2 \; I_3 \; rep \; I_4$ is provided for static composition. In this use, the *metaCombiner* approach to behavior composition is related to other more expressive and modular alternatives to static inheritance, such as *mixin-based* inheritance [6], its derivative proposed in [39], and *feature-oriented programming* [28]. However, the static *metaCombiner* model is more general than these approaches with respect to dealing with the trade-off between internal encapsulation and incremental modification.

In the *metaCombiner* composition model, incremental modification and internal encapsulation are treated separately. While internal encapsulation is realized by means of visibility scopes, incremental modification happens independently of these scopes, individually for each method. Adjustments are separated in visibility scopes according to their replicated methods. This ensures that a module in one scope of a replicated method cannot invoke the definition of this method from another scope. Nevertheless, definitions from adjustments in different scopes can flexibly be arranged in an incremental modification relationship

by means of their individual execution ordering, or they can invoke shared attributes from each other by means of *self* calls. This different kind of treatment is made possible due to the additional abstraction level separating the inheritance structure from behavior definition. In order to emphasize the importance of this separation, we will show in the following how the flexibility problems of advanced approaches to the name collision problem, such as [6, 7, 39], can be traced down to its absence.

In the absence of the combination layer, a single kind of relationship between modules is supposed to globally regulate both the internal encapsulation and the incremental modification relation between their corresponding methods. This leads to the following possible situations. The first possibility is to use the replicated methods as the basis for the relationship between modules. Two modules defining replicas of the same message are made globally invisible for each other. The consequence is that methods which represent incremental modifications of each other are automatically considered as replicas. Dummy subclasses are needed, simply for reestablishing the incremental modification relationship between the latter. This situation is typical for the graph oriented multiple inheritance approaches [4, 7, 21, 36]. For illustration, the hierarchy in Fig. 13 a) represents the definition of a shared and ATM-account, modeled in the *point of view notion of multiple inheritance* [7] approach. The single role of the class *ASAccount* is to reimplement *debit:*, merely because otherwise it would be impossible to connect both versions of *debit:*, defined in *SharedAccount* and in *ATMAccount*, respectively, such that they are executed one after the other. With the explicit as-expressions, explicit inheritance structure information is unnecessarily hard-coded into the implementation of classes, which damages the flexibility especially in dynamic environments.

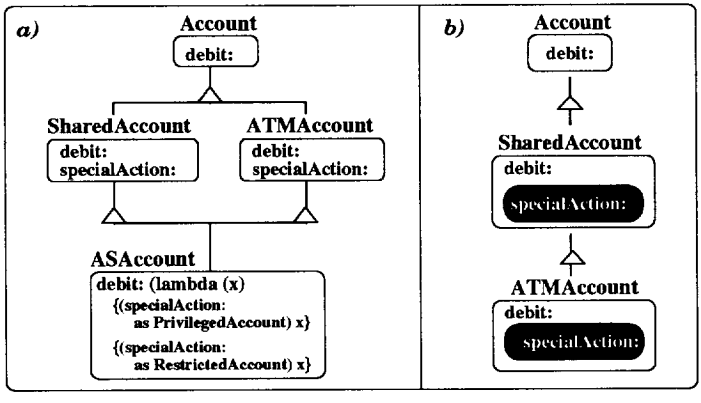

Fig. 13. Double restricted account in graph and linear models

A second alternative is to use the refinement methods as the basis of the relationship between modules. All modules are put into a total order, which results in a single visibility scope. This corresponds to linear approaches such as [6, 39]. Inflexible fixing and renaming is needed in order to provide individual scopes for the replicated methods, which are otherwise impossible when the modules are totally ordered. The definition of a shared and ATM-account in the linear approach presented in [39] is shown in Fig. 13 b). Black boxes within the definitions for *debit:* are used to show that *specialAction:* has been fixed in these definitions and then removed from the domain of both modules. This results in a kind of class qualification. In order to refine the *specialAction:* attribute of the *SharedAccount*, say by means of a class *SpecialShared* reimplementing *specialAction:* to provide a special sharing strategy, the hierarchy of modules must be rebuilt and the fix-point operator must be reapplied. Obviously, the problem related to message qualification is here less severe than in graph oriented approaches [4, 21], due to the total expressiveness on the inheritance structure provided by the mixin-based approach. Nevertheless, this is true only for static environments; any kind of class qualification is not practicable in a dynamic environment.

7 Conclusions

In this paper, a new approach to dynamic evolution of object behavior in a class-based environment has been proposed. The approach is based on introducing an additional abstraction layer between objects and classes in order to control the compositional aspects of the behavior of evolving objects. The metaobjects on this explicit combination layer provide support for dynamic behavior alterations in specific contexts without violating the encapsulation of the client interface and support for internal encapsulation. A formal description of the approach was presented and its feasibility was demonstrated by implementing it as a metalevel extension of Smalltalk-80. There are several areas for future research. First, it is interesting to study the usefulness of the proposed model in various applications areas where dynamic object modification is required. Second, an interesting area is to investigate how the *metaCombiner* mechanism can be utilized to provide object behavior evolution in a static typed language like C++.

Acknowledgements

The author is grateful to Wolfgang Merzenich for promoting this research, and to Bernd Freisleben for encouraging the work on this paper and for useful suggestions on earlier versions of it. Many thanks to the anonymous referees for their useful comments.

References

1. Aksit M., Wakita K., Bosch J., Bergmans L., and Yonezawa A. Abstracting object interactions using composition-filters. In Guerraoui R., Nierstrasz O. and Riveill

M., eds., *Object-based Distributed Processing*, LNCS 791, pp 152-184, Springer-Verlag, 1993.
2. Atkins M.S. and Coady M.Y. Adaptable concurrency control for atomic data types. In *ACM Transactions on Computer Systems*, vol. 10, no. 3, pp. 190–225, 1992.
3. Black A, Consel C., Pu C., Walpole J., Cowan C., Autrey T., Inouye J, Kethana L., Zhang K.. Dream and reality: incremental specialization in a commercial operating system. Technical Report TR-95-001, Oregon Graduate Institute of Science and Technology, 1995.
4. Borning A. H. and Ingalls D. H. H.. Multiple inheritance in Smalltalk-80. In *Proceedings at the National Conference on AI*, Pittsburgh, 1982.
5. Bracha G, and Cook W.. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, vol. 25, no. 10, pp. 303–311, 1990.
6. Bracha G. and Lindstrom G. Modularity meets inheritance. Technical Report UUCS-91-017, University of Utah, 1991.
7. Carré B. and Geib J. M. The point of view notion for multiple inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, vol. 25, no. 10, pp. 312–321, 1990.
8. Chambers C. Predicate classes. In W.Olthoff, ed., *Proceedings ECOOP '93*, LNCS 707, pp. 268–297, Springer-Verlag, 1993.
9. Cook W. and Palsberg J. A denotational semantics of inheritance and its correctness. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, vol. 24, no. 10, pp. 433–443, 1989.
10. Goldberg A. and Robson D. *Smalltalk 80: the Language and its implementation.* Addison-Wesley, 1983.
11. Gottlob G., Schrefl M., Roeck B. Extending object-oriented systems with roles. In *ACM Transactions on Information Systems*, vol.14, no.3, pp 268–296, 1996.
12. Harrison W. and Ossher H. Subject-oriented programming: (A critique of pure objects). In *Proceedings OOPSLA '93*, ACM SIGPLAN Notices, vol. 28, no. 10, pp. 411–428, 1993.
13. Hense A. V. Denotational semantics of an object oriented programming language with explicit wrappers. In *Theoretical Aspects of Computer Software 1991*, LNCS 526, pp. 548–567, Springer-Verlag, 1991.
14. Keene S. Object-oriented programming in Common Lisp: a programmer's guide to CLOS, Addison-Wesley, 1989.
15. Kiczales G., des Rivières J., Bobrow D. G. *The art of the metaobject protocol*, MIT Press, 1991
16. Kiczales G. Towards a new model of abstraction for the engineering of software. Invited Talk in OOPSLA '94, (http://www.xerox.com/PARC/spl/eca/oi.html).
17. Kiczales G., Irwin J., Lamping J., Loingtier J. M., Lopes C. V., Maeda C., Mendhekar A. A position paper on aspect-oriented programming. (http://www.parc.xerox.com/spl/projects/aop/position.html).
18. Knudsen J. L. Name collision in multiple classification hierarchies. In S. Gjessing and K. Nygaard, eds., *Proceedings ECOOP '88*, LNCS 322, pp. 93–109, Springer-Verlag, 1988.
19. LaLonde W. R., Thomas D. A., Pugh J. R. An exemplar based Smalltalk. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 322–330, 1986.
20. McAffer J. Meta-level programming with CodA. In W. Olthoff, ed., *Proceedings of the ECOOP '95*, LNCS 952, pp. 190–214. Springer-Verlag, 1995.

21. Meyer B. *Object-oriented software construction.* Prentice Hall, 1988.
22. Mezini M. Supporting evolving objects without giving up classes. In B. Meyer C. Minings and R. Duke, eds., *Proceedings of the 18th TOOLS Conference*, pp. 183–197, Prentice Hall, 1995.
23. Mezini M. Dynamic metaclass construction for an explicit specialization interface. In *Proceedings of the Reflection '96 Conference*, pp. 203–219, 1996.
24. Mezini M. Incremental redefinition of open implementations. In Ch. Zimmermann, ed., *Advances in object-oriented metalevel architectures and reflection*, pp. 265-290, CRC Press Inc., 1996.
25. Mezini M. Ph.D. Thesis (in preparation)
26. Moon D. A. Object-oriented programming with Flavors. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 1–8, 1986.
27. Nierstrasz O and Tsichritzis D. Object-oriented software composition. Prentice Hall, 1995
28. Prehofer C. Feature-oriented programming. To appear in *Proceedings of ECOOP '97.*
29. Seiter L. M. Design Patterns for Managing Evolution. Ph.D. Thesis, Northeastern University, 1996.
30. Seiter L. M., Palsberg J, Lieberherr K. Evolution of object behavior using context relations. In Garlan D., ed., *Proceedings of the 4th ACM SIFSOFT Symposium on Foundations of Software Engineering*, Software Engineering Notes, vol. 21, no. 6, pp. 46–56, ACM Press, 1996.
31. Snyder A. Inheritance and development of encapsulated software components. In B. Shriver and P. Wegner, ed., *Research Directions in Object-Oriented Programming*, pp. 165–188, MIT Press, 1987.
32. Stein L. A., Lieberman H., Ungar D. The treaty of Orlando. In W. Kim and F. Lochovsky (Eds.), *Object-Oriented Concepts, Databases and Applications*, pp. 31–48. ACM Press and Addison-Wesley.
33. Stein L. A. Delegation is inheritance. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, no. 12, pp. 138–146, 1987.
34. Steyaert P., Codenie W., D'Hondt T., D'Hondt K., Lucas C., Van Limberghen M. Nested Mixin-Methods in Agora. In O. Nierstrasz, ed., *Proceedings ECOOP '93*, LNCS 707, pp. 197–219. Springer-Verlag, 1993.
35. Steyaert P. and De Meuter W. A marriage of class-based and object-based inheritance without unwanted children. In W. Olthoff, ed., *Proceedings ECOOP '95*, LNCS 952, pp. 127–145, Springer-Verlag, 1995.
36. Stroustrup B. *The C++ programming language.* Addison-Wesley, 1986.
37. Stroud, R.J. and Wu Z. Using metaobject protocols to implement atomic data types. In W. Olthoff, ed., *Proceedings ECOOP '95*, LNCS 952, pp. 168–189, Springer-Verlag, 1995.
38. Ungar D. and Smith R. B.. Self: the power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN notices*, vol. 22, no. 12, pp. 227–242, 1987.
39. Van Limberghen M. and Mens T. Encapsulation and composition as orthogonal operations on mixins: A solution to multiple inheritance problems. In *Object-Oriented Systems*, 3(1), 1996.
40. Weihl, W.E. and Liskow, B. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2), pp. 244–269, 1985.