Towards Fully-fledged Reverse Inheritance in Eiffel

Markku Sakkinen1, Philippe Lahire2, and Ciprian-Bogdan Chirilă3

[1] Department of Computer Science and Information Systems, University of Jyväskylä `sakkinen@cs.jyu.fi`
[2] I3S Laboratory, University of Nice – Sophia Antipolis and CNRS `Philippe.Lahire@unice.fr`
[3] Politehnica University of Timisoara `chirila@cs.upt.ro`

Abstract. Generalisation is common in object-oriented modelling. It would be useful in many situations also as a language mechanism, reverse inheritance, but there have been only few detailed proposals for that. This paper defines reverse inheritance as a true inverse of ordinary inheritance, without changing anything else in the language. Eiffel is perhaps the most suitable language for that purpose because of its flexible inheritance principles. Moreover, there exists good previous work on Eiffel, on which we have built. We describe the most important aspects of our extension, whose details proved to be more difficult than we had assumed. It would be easier if some modifications were made to Eiffel's ordinary inheritance, or if one designed a new language. Finally, we present an implementation in which reverse inheritance is changed to ordinary inheritance by automatic transformations, thus no compiler modifications are needed.

1 Introduction

Generalization is widely used in object-oriented (OO) modelling and design, but it is not available on the implementation level in any widely used language or system. We propose to extend object-oriented languages with a new relationship, *reverse inheritance* (RI) or *exheritance*, which is an inverse of ordinary inheritance (OI). Reverse inheritance allows non-destructive generalization, just like ordinary inheritance allows non-destructive specialization. This is not a completely new idea, but it has been very little treated in the literature. Of course, we are building on applicable previous research (see Section 2).

Of course, if the source code can be modified, it is always possible to add a direct superclass (*parent* in Eiffel terminology) to an existing class. This is a common *refactoring* operation, but it may introduce many side effects and affect the robustness of the existing classes. Further, it is very undesirable or even impossible in many cases to modify existing classes, e.g. from standard libraries. The situation is particularly difficult when one needs to combine two or more large class hierarchies from different sources.

None of the previous proposals that we know has been implemented. This time we wanted to allow RI to be tried out in practice, and therefore designed an extension to an existing industrial-strength language, instead of a nice, formally defined toy language. Such an exercise gives better possibilities to weigh the potential benefits of reverse inheritance against its cost (added language complexity).

Eiffel is a particularly interesting and suitable language to extend with RI, because of its well thought-out design principles. Most importantly, its flexible and clean implementation of multiple inheritance with explicit clauses for adaptation allows us to propose a solution that is both integrated and expressive enough. Because no implementation of the new, significantly changed version of Eiffel [1] [2] existed yet, we based our extensions on the stable old version often known as Eiffel 3 [3]. We use mostly the terminology of Eiffel literature, except the term 'method' instead of 'routine'. We'll try to explain those Eiffel terms and concepts that could be too alien to many readers.

To give a taste of RI, Figure 1 is a small example. Suppose that we have two classes *RECTANGLE* and *CIRCLE* designed independently from each other. It is noted later that some of their features can be factored into a common parent class, which is named *FIGURE*. We do not explain all details here, but the example should be understandable; the new keyword **foster** denotes a class defined using RI. For reference purposes, we have added line numbers, which are not part of the code. By default, all *features* (the common superconcept

```
01 class CIRCLE
02    feature
03       radius: REAL
04       location: POINT
05       draw is do ... end
06 end – class CIRCLE

07 class RECTANGLE
08    feature
09       height: REAL
10       width: REAL
11       location: POINT
12       draw is do ... end
13 end – class RECTANGLE

14 deferred foster class FIGURE
15    exherit
16       CIRCLE
17       RECTANGLE
18       all
19 end – class FIGURE
```

Fig. 1. Simple example of reverse inheritance

of attribute and method in Eiffel) with the same name and signature in both CIRCLE and RECTANGLE will be exherited to FIGURE; this means *location* and *draw*. However, they will become abstract (*deferred* in Eiffel) by default. For programmers using these three classes, the example is fully equivalent to standard Eiffel code in which FIGURE would be defined first and the others as its direct subclasses (*heirs* in Eiffel).

We will present the main features of our approach in the rest of this paper. Section 2 gives a brief overview of previous literature, whereas in Section 3 we address the main principles to be followed. We continue in Section 4 giving the fundamentals of our approach. Sections 5 and 6 illustrate the use of RI in the two main situations: adding a superclass at the top level of the hierarchy, and inserting a class between two or more classes in the hierarchy. Finally we conclude and set a perspective for future research in Section 7.

We already have a quite comprehensive implementation of our RI extension for Eiffel, by a transformation to standard Eiffel. Unfortunately, space does not permit us to describe it in this paper, but we refer interested readers to the website
https://nyx.unice.fr/projects/transformer .

2 Previous research

The earliest article we have found that discusses a concrete generalization mechanism is [4], which uses the term 'upward inheritance'. Its purpose is enabling the integration of different OO databases into a multidatabase system, or building a homogeneous global view of heterogeneous systems. The paper [5] has a similar purpose. Generalization is much more important in database integration than in "ordinary" programming, because the homogenization of the underlying databases is usually out of the question. It is also easier, because the generalization classes live on a different layer of the system than the actual database classes. On the other hand, there is the additional problem that one real-world *object* (instance) may well be represented in several databases.

Our goal is essentially different from the above, namely allowing classes to be defined either by specialization (OI), generalization (RI), or a combination of both, within the same context. To our knowledge, the first paper proposing such an approach and mechanism is [6]. We consider it a seminal paper, although it is somewhat simplistic or even erroneous on some points.

A significant step forward was made in the paper [7], which presents a detailed proposal for adding reverse inheritance into Eiffel. It also discusses many problems both on the conceptual level and in the implementation. We have adopted the most important terms from there, in particular '*foster class*'. However, we could not see a reason for speaking about reverse *type* inheritance, because inheritance is always a relationship between classes in Eiffel and most other OOPLs.

One surprisingly missing aspect in both [6] and [7] is the possibility of a class being defined by a combination of simultaneous ordinary and reverse inheritance, i.e., inserted into the inheritance hierarchy between a superclass (parent) and its subclasses (heirs).

The workshop paper [8] was written unaware of [7], so the new term 'exheritance' was coined there. It is quite optimistic about RI and suggests several new ideas.

The workshop paper [9] discusses the application of RI to Java, including implementation aspects. Adding RI to a single-inheritance language had not been treated in earlier papers. It is both much simpler and much less powerful than with multiple inheritance, but not trivial. Our first example (Figure 1) did not need multiple ordinary inheritance.

Since 2005, we have cooperated and tried to combine our different viewpoints on reverse inheritance. The current paper builds on the earlier work, especially [7] and [8], with essential improvements on several points. Because we are also implementing our approach, we needed to be more thorough than the earlier papers.

3 Main principles

Before going any further in the description of reverse inheritance it is important to state the main principles of our approach. The rules are presented in an approximate order of importance. We do not claim them to be self-evident; there can be approaches based on different principles.

Firstly, since we are designing an extension to an existing language, it is important that classes and programs which do not use the extension will not be affected.

Rule 1: Genuine Extension
Eiffel classes and programs that do not exploit reverse inheritance must not need any modifications, and their semantics must not change.

Secondly, it is very important that after a class has been defined using RI, it can be used just as any ordinary class. Otherwise, foster classes would be far less useful, and the additional language complexity caused by RI would certainly not pay off.

Rule 2: Full Class Status
After a foster class has been defined, it must be usable in all respects as if it were an ordinary class.

In particular, a foster class can be used as a parent in ordinary inheritance and as an heir in further reverse inheritance,

Thirdly, in OI the semantics of a given class is not affected if a new class is defined as its direct or indirect subclass (*descendant* in Eiffel), or if some existing descendant is modified. In contrast, any modifications to a superclass (*ancestor* in Eiffel) affect all its subclasses, and can even make some existing descendants illegal unless their definitions are changed also. We want RI to be a mirror image of OI in this respect, i.e., the dependencies between classes to be the opposite of what they are in RI (see [7]).

Rule 3: Invariant Class Structure and Behaviour
Introducing a foster class as a parent C of one or several classes $C_1, \ldots, C_n$ using reverse inheritance must not modify the structure and behaviour of $C_1, \ldots, C_n$.

Fourthly, the reverse inheritance relationship is intended to be symmetric with ordinary inheritance. This means that it should be as completely interchangeable with ordinary inheritance as possible. In the new version of Eiffel [2] this would imply also that conforming and non-conforming reverse inheritance relationships must be distinguished.

Rule 4: Equivalence with Ordinary Inheritance
Declaring a reverse inheritance relationship from class A to class B should be equivalent to declaring an ordinary inheritance relationship from class B to class A.

Of course, this does not mean that the syntactic definitions of the two classes would be the same in both cases.

As a consequence of this rule, it would be good if all adaptation capabilities provided for RI had their counterparts in pure Eiffel language. However, we actually wish to have some adaptations that cannot be exactly translated to OI (see Section 4). On the other hand, we did not consider it worthwhile to implement all possible complications of Eiffel OI also in RI; Rule 7 is an example of that.

Fifthly, we want reverse inheritance to leave the existing inheritance hierarchy as intact as possible.

Rule 5: Minimal Change of Inheritance Hierarchy
Introducing a foster class must neither delete direct inheritance relationships (parent-heir relationships) nor create *any* inheritance relationships (ancestor-descendant relationships) between previously existing classes.

Note that, taking Rule 4 into account, RI may well create new inheritance *paths* between existing classes, but only for existing ancestor-descendant pairs (Section 6).

The paper [8] suggested that it could be possible to define also new parent-heir relationships, and even equivalence relationships, between existing classes (if they are feasible). However, we decided not to include that possibility in our Eiffel extension, to keep things simpler.

Sixthly, we need to define which features are candidates to be *exherited* in reverse inheritance. The following rule is essentially a consequence of the previous rules and the adaptation possibilities of OI in Eiffel extended for RI (as just mentioned).

Rule 6: Exheritable Features

The features $f_1, \ldots, f_n$ of the respective, different classes $C_1, \ldots, C_n$ are exheritable together to a feature in a common foster class if there exists a common signature to which the signatures of all of them conform, possibly after some adaptations. Each of the features $f_1, \ldots, f_n$ can be either immediate or inherited.

In pure Eiffel these features could be similarly factored out to a common parent, but any extended adaptations (see above) would require new or modified methods in the heir classes.

Some common special cases are simpler than the general case: In single RI, all features are trivially exheritable. In multiple RI, all f_i may already have the same signature, or one of them may have a signature to which all others can be made to conform. We will explain the possible adaptations in Section 4.

Lastly, we want to avoid the complexity of allowing one feature in a foster class to correspond to several features in the *same* exherited class, although this would be a direct equivalent of repeated inheritance with renaming.

Rule 7: No Repeated Exheritance

Two different features of the same class must not be exherited to the same feature in a foster class.

The definition of the semantics of reverse inheritance in the following sections, on both the conceptual level and the concrete language level, relies on the above six rules.

4 Basics of our approach

Where needed to avoid ambiguities, we will call the proposed extended language 'RI-Eiffel' in distinction to pure Eiffel. Details in the concrete syntax used in our code examples are not important, and the syntax may be slightly modified in the future.

Following the paper [7], a class defined using a RI relationship is called a foster class and is preceded by the keyword **foster** in order to point out the special semantics of this class with respect to normal Eiffel classes. In fact, a foster class also requires special implementation (see [7]). In a new language with both OI and RI, the 'foster' keyword would be needed no more than a 'heir' or 'subclass' keyword.

A foster class may be effective (concrete) or marked as **deferred** (abstract) like any other class. It is a fully-fledged class in all respects; in particular, further classes can be derived from it by both OI and RI. Otherwise reverse inheritance would hardly be useful and interesting.

In order to reverse-inherit or exherit from one or several classes we use a clause **exherit**[4] in a foster class, in the same way as we use a clause **inherit** in

[4] We do not use the keyword **adopt** from [7], because we have introduced **adapt** and **adapted** (see later).

order to reuse and to extend the behaviour of one or several classes. Figure 1 was a simple example.

The set of *exheritable* features is defined by Rule 6 in Section 3. Because it is not always desired to exherit all of them, the set of really *exherited* features can be further restricted by using some rather intuitive keywords. The keyword **all** in Figure 1 is actually redundant, because we take it as the default.

In ordinary inheritance, also the implementation of every feature is copied to the heir class by default, but in Eiffel it is also possible to copy only its signature, i.e., make it deferred, using the clause **undefine**. A reasonable approach for exheritance is exactly the reverse: the default is that a feature is deferred in the foster class. Therefore, the keyword **undefine** is not needed in RI. When the implementation of a feature *should* be moved (or copied) to the foster class, that is specified explicitly by the clause **moveup**[5].

The strongest reason for the above default is that usually it is not even possible to copy the body of a *method* from a heir class. That would require all other features accessed by the method to be exherited also, but in multiple RI they may not be even exheritable [8]. It seemed best to us to have the same default also for attributes.

If an exherited feature is a method, a body can be written in the foster class just as in an ordinary class. In that case, it seems consistent with OI to require a **redefine** clause for each exherited class in which the feature is effective (either as a method or as an attribute).

In Figure 1 the features of the exherited classes that should be unified in the foster class have the same name. In general, it is very likely that some corresponding features have different names, and in converse that some features with the same name should *not* be unified. That has been recognized in all previous papers, and was also taken into account in Rule 6 (Section 3). It is therefore necessary that we allow renaming in an **exherit** clause by **rename** subclauses; this facility exists for OI in standard Eiffel. Examples of that will appear in later sections.

We already mentioned above the use of **redefine** in Eiffel to announce the reimplementation (overriding) of methods. The same keyword — a bit unfortunately — is used also to announce the redefinition (redeclaration) of method signatures and attribute types. We allow such redefinitions also in RI, as might be deduced from Rule 6. Since type/signature redefinitions in OI in Eiffel are covariant, they must be the inverse in RI. This means that the type of an attribute, or of a parameter or the result of a method, in the foster class must be a common ancestor of the types of the corresponding entities in the exherited classes.

In multiple RI, the type/signature of an exherited feature *must* be redefined in most cases in the foster class. The exception are cases where the signature is exactly the same in all exherited classes (ECs) and it is not changed in the foster class. If the signatures in all other ECs conform to the common signature in a

[5] We invented this keyword, because 'move' is probably a rather common identifier in programs.

subset of the ECs, we could take the latter as the default for the foster class, but for the sake of clarity we require a **redefine** clause for the other ECs. — Note that even a feature that is to be deferred in the foster class needs a **redefine** clause if its signature is changed.

It is a speciality of Eiffel that a method which has no parameters and returns a result can be redefined as an attribute in a descendant class. The opposite is not allowed, because assignment to an attribute has no counterpart with a method. This implies for RI that a feature from the exherited classes can always be redefined as a method in the foster class, but it can be redefined as an attribute only if it is an attribute in all exherited classes.

Because the exherited classes often have not been developed in the same context, it is possible that even the number of parameters, their scales or the scale of the result or an attribute is not the same for features that represent the same thing (see [4] for more). It should be possible to do some adaptations to take into account these aspects and then unify the adapted features in RI. Such adaptations do not exist in Eiffel, because they are not needed in OI.

Adapting a feature must not change the exherited class or its objects, according to our Rule 3. Therefore, the conversion is made on the fly, when the feature is accessed through a variable whose type is the foster class. This is one special characteristic of foster classes: in standard Eiffel the type of the referencing variable does not affect the behaviour of a feature, except that it may affect the dynamic binding if repeated inheritance is involved. Note that adaptation makes sense independently of whether the feature is deferred, moved or (re)defined in the foster class.

We introduce two new keywords for expressing adaptation. In the **exherit** clause, for every exherited class the features to be adapted must be listed after the keyword **adapt**. After all these clauses, for every feature that needs adaptation from at least one heir class, the adaptations must be specified after the keyword **adapted**. Each adaptation subclause must specify the name(s) of the heir class(es) to which it applies, and then the adaptation itself.

For methods, the adaptation must specify by expressions, first the actual parameters to be submitted to the method of the heir class, and second the result to be returned to the caller. Formal parameters of the foster class method can be used in both expressions, and the result from the heir class method in the latter one. Features of the foster class can also be used, at their state before or after the invocation of the heir class method, respectively. For attributes, the adaptation must specify two conversion expressions, from the heir class representation to the foster class representation and vice versa.

We omit describing the complete syntax for adaptation expressions here. It is important to note that they must not cause side effects, as a corollary of Rule 3.

5 Adding a root class as a parent

The simplest cases of RI are those where the foster class is on the top of the hierarchy, i.e., it has no explicit parent. It will then implicitly have the universal root class *ANY*[6] as parent, but we can ignore it, except in the rare case that some exherited class has renamed, redefined or undefined some feature inherited from *ANY*. Therefore, there is no interference caused by the combined use of OI and RI in the same class. In order to illustrate such an RI relationship, but a non-trivial one, we enhance slightly the example of Figure 1 (Section 1). Figure 2 contains only the code of the foster class.

```
01 deferred foster class FIGURE
02    exherit
03       CIRCLE
04          redefine location
05          adapt location
06          end
07       RECTANGLE
08          redefine location
09          rename display as draw
10          end
11       all – all exheritable features
12       feature
13          location: GEN_POINT
14             adapted CIRCLE
15             to x := result.x/10, y := result.y/10
16             from x := result.x*10, y := result.y*10
17       end
18    end – class FIGURE
```

Fig. 2. Insertion of class FIGURE on top of two classes developed separately

We assume only one change in the exherited classes from Figure 1: class *RECTANGLE* has a method named *display* instead of *draw*. However, it has the same meaning as *draw* in class *CIRCLE*, and thus these two features should be exherited together. To achieve this, *display* is renamed as *draw* in the exheritance. By default, the feature becomes deferred in class *FIGURE*, and so the class itself has to be declared as deferred (line *01*).

The attribute *location* is exherited automatically because it satisfies Rule 6 from Section 3. However, to keep it as an attribute and not a deferred feature in the foster class, it must be either explicitly moved from one exherited class or redefined. Here we choose the latter alternative (line *10*): for some reason, we want it to be of type *GEN_POINT*, which must be an ancestor of *POINT* (line *12*).

[6] It corresponds to *Object* in many other languages.

Let us assume next that the class *POINT* has the attributes x and y of type *REAL*. and that the scale of these attributes is in millimetres within an object of type *RECTANGLE*, while in class *CIRCLE* it is in centimetres. We decide to handle it in millimetres also in class *FIGURE*, and therefore we need the **adapt** clause for *CIRCLE*. In the later **adapted** clause (lines *14* to *16*), we present a tentative syntax for the adaptation of an attribute. The **to** subclause specifies the conversion needed for writing (assigning to) the attribute through a variable of type *FIGURE*, and the **from** subclause the conversion needed for reading it.

Figure 3 is a class diagram of this situation. In this and later diagrams we use "RI-UML", where reverse inheritance is denoted by dashed lines and downward pointing triangle arrowheads (upward might actually be a better choice).

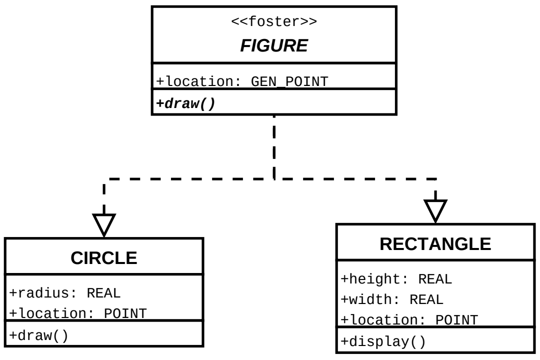

Fig. 3. Class diagram for Figure 2

While renaming and redefinition are the inverses of the corresponding modifications in OI, adaptation has no counterpart in OI (see Section 4). In this small example we have no adaptation of methods; it would not even be relevant for *draw*, because it has neither a result nor parameters.

6 Adding a class with both reverse and ordinary inheritance

Here we study situations in which a foster class is defined "in the middle" of an inheritance hierarchy, i.e., using both ordinary and reverse inheritance. Such a foster class we will call '*amphibious*'[7] and other foster classes 'non-amphibious' when needed. Because RI must not create new inheritance relationships between existing classes (Rule 5, Section 3), every class that the new foster class inherits from must already be a common ancestor of all classes being exherited.

[7] A metaphor from biology: the features of these classes come partly from above ("the land") and partly from below ("the water") in the hierarchy.

In the simplest case, the exherited classes have a common parent and the foster class is inserted between the parent and its original heirs. We present a slightly more complex case, which is a continuation of our previous example (Fig. 3).

The classes *CIRCLE* and *RECTANGLE* have no method for moving the objects. Suppose that they are kept as such, but the heir classes *MOVABLE_CIRCLE* and *MOVABLE_RECTANGLE* that have a *move* method are added. Later one wants to give them a common parent *MOVABLE_FIGURE*, which exherits at least the *move* method. It is quite natural that this new class is also an heir of *FIGURE*, and therefore inherits all its features.

Figure 4 gives the code of the new foster class (the new heir classes are trivial), and Figure 5 shows the augmented class diagram. To prevent some possible confusions, we have changed the RI relationships of Figure 3 into equivalent OI relationships; this is possible according to Rule 4 (Section 3).

```
deferred foster class MOVABLE_FIGURE
   inherit FIGURE
      redefine location
      end
   exherit
      MOVABLE_CIRCLE
         moveup location
         end
      MOVABLE_RECTANGLE
         rename display as draw
         end
   feature
end – class MOVABLE_FIGURE
```

Fig. 4. Inserting a class between a class and its descendants

The adaptation of the attribute *location* in class CIRCLE (Figure 2) makes this example trickier. The implementation of that attribute is moved to the amphibious class *MOVABLE_FIGURE* from *CIRCLE*. Therefore no scale conversion must be performed when *location* in a *CIRCLE* object is accessed through a reference of type *MOVABLE_FIGURE*. However, the inverse conversion must be performed when a *MOVABLE_RECTANGLE* object is accessed through a reference of that type. The case would be different if the implementation of *location* were moved from *MOVABLE_RECTANGLE* or simply inherited from *FIGURE*.

In Eiffel terminology, those features of a class that are not inherited from its parent(s) are called *immediate* features. In contrast, a foster class cannot have immediate features, because all its features are exherited from its heir(s) (even those that are also inherited). Thus it makes sense to classify them into amphibious (those that are both inherited and exherited) and non-amphibious

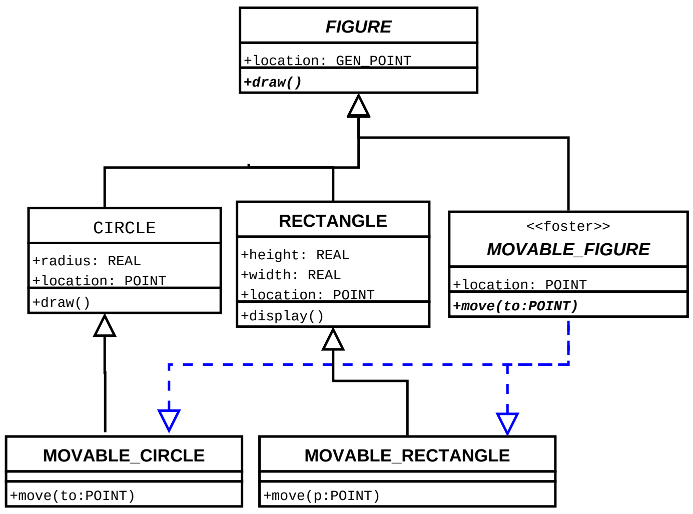

Fig. 5. Class diagram for Figure 4

features. In the rest of this section, we discuss only the *amphibious* ones, because the existence of a parent class is irrelevant to the others.

In the sequel, we will use the abbreviations 'PC' for the parent class(es), 'FC' for the new foster class, and 'EC' for the exherited class(es). We assume at first that there is only one PC. Thus, for each amphibious feature in the FC, there exists a PC version, an FC version, and a version in each EC. We must study what relationships are *possible* between these versions, and what are sensible *default* relationships.

The *type* (or signature in the case of a method) of the FC version can always be the same as that of the PC version, because all EC versions already conform to it. Therefore, we choose this as the most natural default type for the FC version. If all EC versions have the same type, that type is likewise trivially possible also for the FC version. In general, the FC version can have any type that conforms to the type in PC and to which the types in all ECs conform.[8]

The possibilities for the *implementation* of the FC version are slightly more complicated. While the implementation of a method is a body, here we consider the implementation of an attribute to be simply the fact of being an attribute (in contrast to deferred or a parameterless method), and the implementation of a deferred (abstract) feature to be empty.

The implementation in the FC can be the same as in the PC, except if it is a method body and the signature is redefined; then the body must also be redefined, as required in standard Eiffel. If the feature is an attribute in the PC, it *must* be an attribute also in all ECs and in the FC, again by the rules of

[8] In particular, if the feature has retained its original type in any EC, it cannot be changed in the FC either.

standard Eiffel. Otherwise, it *can* be an attribute in the FC only if it is so also in all ECs, but it can always be an effective (implemented) method or deferred. However, exheriting the body of a method from an EC is usually impossible, as explained in Section 4. — All in all, it is most natural that also the default implementation for the FC version is inherited from the PC.

If an amphibious feature is effective in the PC and redefined (i.e., reimplemented) in the FC, a **redefine** clause is required in the inheritance by standard Eiffel rules. For consistency, we require the clause likewise if the feature is moved from an EC.

In Figure 4, the attribute *location* of the PC (*FIGURE*) has retained its name in the ECs, although its type has been redefined. Therefore, it will by default retain that name also in the FC. The name of the inherited method *draw* has been changed to *display* in *MOVABLE_RECTANGLE*, and therefore we require it to be explicitly renamed in the exheritance. This is consistent with standard Eiffel and makes things clear, although the correspondence between EC and FC features would, in this case, be unambiguous even without explicit renaming.

In other situations, renaming in Eiffel can cause an inherited feature to be replicated; this happens with repeated inheritance. For instance, if a common heir of *CIRCLE* and *RECTANGLE* is defined without renaming, it will have the two distinct methods *draw* and *display*.

Eiffel allows also the inverse of the previous situation, namely that two features from the same parent class are unified into one feature in an heir class. Likewise, two features from *different* parents can be unified in multiple inheritance. We will not discuss these complications in this paper.

7 Conclusion and Perspectives

This paper proceeded from previous proposals to introduce a generalization relationship, reverse inheritance (RI), to object-oriented programming, in particular to the Eiffel language. Its main goal is to improve the non-destructive reuse of classes by adding new abstraction levels in the middle or on top of a class hierarchy, whereas ordinary inheritance (OI) is devoted to extending the hierarchy at the bottom.

Reverse inheritance is symmetric and complementary with ordinary inheritance. In the design of this new relationship we gave particular attention to its orthogonal integration with all other language constructs, and we also strove to keep the traditional language flavour and code readability of Eiffel. We gave tried preference to robustness and simplicity over expressiveness of the adaptation mechanism. In our work, we have covered virtually the whole Eiffel language. Unfortunately, the paper length limitation forced us to omit here even some very interesting aspects, such as pre- and postconditions and genericity.

We think that RI can have several useful application possibilities besides those already mentioned in Section 1. One example is *interface inheritance*, which is often recommended in theoretical papers, but not offered by any well-known language. In our approach, it can be achieved simply by exheriting all public features of a class as abstract (deferred). Another example is bridging the gap between *subobject-oriented* (as in C++) and *attribute-oriented* (as in Eiffel) multiple inheritance: any set of attributes of a class can be made into a subobject by exheriting them into the same foster class.

Introducing and using RI in an object-oriented language can also have negative effects. One is that it may decrease the readability of code: *with OI you don't know the descendants of a class, and with RI you don't know even all its ancestors*, as Peter Grogono remarked at the ECOOP 2002 Inheritance Workshop [10]. Also, the set of features that a parent class inherits in RI is not as straightforward as the set of features that a child class inherits in OI (see Section 3).

Another negative effect is that RI makes a language larger and more complex. That disadvantage can be minimised if a language is originally designed with RI, or at least RI is designed to be as completely as possible a mirror image of OI. Because this paper proposes an extension to an existing language, we have striven to achieve the latter goal (see Section 3).

In the design of RI it did not appear convenient to keep the syntax so similar to that for OI as we had originally done. We could also not maintain complete symmetry between OI and RI. That was because RI clearly requires stronger adaptations between parents (superclasses) and heirs (subclasses) than are offered for OI in Eiffel or other well-known languages.

Eiffel was a good target for introducing RI, but we intend to look also on other languages and propose adapted solutions for reverse inheritance. That should be rather simple but nevertheless interesting for single-inheritance languages such as C# or Java. It would be very interesting for C++, but probably too difficult because the language is already very complicated, especially its mechanisms of multiple inheritance. A large part of the advantages of RI concern typing, and therefore it would be far less useful for dynamically typed languages such as Smalltalk and CLOS.

Acknowledgments

We gratefully acknowledge Pierre Crescenzo and K. Chandra Sekharaiah for their comments and feedback on previous versions of the paper, and the anonymous reviewers of earlier versions for their useful observations. A significant part of the work by the authors Sakkinen and Chirila was performed during their visits at the I3S Laboratory.

The implementation of RI-Eiffel by transformation to standard Eiffel is based on an approach of Günter Kniesel and his valuable cooperation. It was to a large part performed by Mathieu Acher and Jean Ledesma.

References

1. Meyer, B.: Eiffel: The language. http://www.inf.ethz.ch/~meyer/ (June 2006)

2. ECMA International: Standard ECMA-367 Eiffel: Analysis, design and programming language. http://www.ecma-international.org (June 2006)
3. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
4. Schrefl, M., Neuhold, E.J.: Object class definition by generalization using upward inheritance. In: Proceedings of the Fourth International Conference on Data Engineering, February 1-5, 1988, Los Angeles, California, USA, IEEE Computer Society (1988) 4–13
5. Qutaishat, M., Fiddian, N., Gray, W.: Extending OMT to support bottom-up design modelling in a heterogeneous distrubuted database environment. Data & Knowledge Engineering **22** (1997) 191–205
6. Pedersen, C.H.: Extending ordinary inheritance schemes to include generalization. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, ACM Press (1989) 407–417
7. Lawson, T., Hollinshead, C., Qutaishat, M.: The potential for reverse type inheritance in Eiffel. In: Technology of Object-Oriented Languages and Systems (TOOLS Europe'94). (1994) 349–357
8. Sakkinen, M.: Exheritance — class generalization revived. [10] 76–81
9. Chirilă, C.B., Crescenzo, P., Lahire, P.: A reverse inheritance relationship for improving reusability and evolution: The point of view of feature factorization. [11] 9–14
10. Black, A.P., Ernst, E., Grogono, P., Sakkinen, M., eds.: Proceedings of the Inheritance Workshop at ECOOP 2002. Number 12 in Publications of Information Technology Research Institute, University of Jyväskylä (June 2002)
11. Lahire, P., et al., eds.: Proceedings of The 3rd International Workshop on MechAnims for SPEcialization, Generalization and Inheritance – MASPEGHI'04, Sophia-Antipolis, France, Laboratoire I3S (2004)
12. Cook, S., ed.: ECOOP '89 - European Conference on Object Oriented Programming (Nottingham, July 1989) Proceedings. BCS Workshop Series, Cambridge University Press (1989)