Reverse Inheritance in Statically Typed Object-Oriented Programming Languages

Ciprian-Bogdan Chirilă
University Politehnica of Timişoara, Romania
chirila@cs.upt.ro

Markku Sakkinen
University of Jyväskylä, Finland
sakkinen@cs.jyu.fi

Philippe Lahire
University of Nice, France
Philippe.Lahire@unice.fr

Ioan Jurca
University Politehnica of Timişoara, Romania
ionel@cs.upt.ro

ABSTRACT

Reverse inheritance is a new class reuse mechanism, an experimental implementation of which we have built for Eiffel. It enables a more natural design approach, factorization of common features (members), insertion of classes into an existing hierarchy etc. Due to its reuse potential in Eiffel we consider exploring its capabilities in other industrial-strength programming languages like C++, Java and C#.

Categories and Subject Descriptors

D.3.1.a [**Programming Languages**]: Semantics—*reverse inheritance*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*member factoring, type exheritance*; D.2.13 [**Software Engineering**]: Reusable Libraries—*class reuse*

General Terms

Design, Languages

Keywords

reverse inheritance, generalization, class design, statically typed object-oriented programming languages

1. INTRODUCTION

In this paper we present **reverse inheritance** (RI), a language mechanism which is little known, although its basic idea is not so new. It is especially promising for statically typed object-oriented programming languages (OOPLs), and we have already designed and implemented it as an extension of **Eiffel**. Now we discuss also the potential of RI for **C++**, **Java** and **C#**, which are among the most popular and frequently used languages in the industry today.

The concept of **inheritance** is intensively studied in literature and highly present in all object-oriented programming languages. It is defined as an incremental class creation mechanism which allows transforming the superclass into subclass by augmentation [3]. We will refer to this concept as ordinary inheritance (OI) in distinction to reverse inheritance. OI can be single (SI) when subclasses have only one parent, or multiple (MI) when subclasses (can) have two or more parents.

Reverse inheritance (RI) is a class relationship that implements the generalization concept: subclasses exist first and a superclass is created afterwards. The RI concept has its origins in the database world [14]. Originally, it was used for the generalization of database schemata to achieve the goal of *object reuse*, while in object-oriented programming RI facilitates *class reuse*.

RI is known in the literature also as *exheritance* [11], *adoption* [6], *generalization* [8], and *upward inheritance* [14]. To show that a class is defined using RI, it is called *foster class* [6], *generalizing class* [11], or *adoptive class*. Here we will use ther term 'foster class'. RI can be single when there is only one direct subclass involved, and multiple when a foster class is built on top of multiple subclasses.

RI was implemented experimentally for the first time in Eiffel [7] and the extended language is named RIEiffel (Reverse Inheritance Eiffel) [12]. To integrate RI into other industrial-strength programming languages, its interaction with the main language mechanisms must be taken into account.

We will use the Java terms 'member', 'field' and 'method' instead of the corresponding Eiffel terms — 'feature', 'attribute' and 'routine'. We will use the Eiffel terms 'parent' and 'heir' for *direct* superclass and subclass, respectively; likewise 'ancestor' and 'descendant' for superclass and subclass in general (and including the class itself). In the class diagrams, we extend standard UML notation by a downward pointing triangle arrowhead to denote reverse inheritance.

The rest of this paper is structured as follows. In section 2 we present briefly some of the RI class reuse capabilities. In section 3 we discuss what are the possible choices for a superclass in the context of RI. Section 4 presents the main issues related to member factoring. In section 5 we discuss several aspects related to superclass implementation. Section 6 presents a special adaptation mechanism used to adapt subclass members having different signatures to a common superclass signature. In section 7 we present some related work, while in section 8 we draw the conclusions and set the future work.

2. CLASS REUSE CAPABILITIES

One benefit of having the RI class relationship in a language is the facility of a more natural class design. Heirs can be independently modeled first, and commonalities factored into a common parent later. Of course, this is a common refactoring operation, but in current languages it requires the definitions of all heirs to be modified. With RI, the equivalent refactoring is *nondestructive*.

It is important to note that a foster class is a completely normal class, and the semantics of all classes in an inheritance hierarchy with RI are exactly the same as if the same hierarchy had been defined using OI.

2.1 Capturing Common Functionalities

RI facilitates the factoring of common functionalities from classes that have been designed in different contexts. Thus, they can be used uniformly with the help of the newly created parent. In figure 1 we present three heirs: *Rectangle*, *Ellipse* and *Triangle*, which share one common functionality, namely method *draw()*. Creating the *AbstractShape* parent on top of the three heirs and factoring method *draw()* enables polymorphic calls to that method. However, in practice it is very unlikely that methods having similar functionalities also have exactly the same name and signature. For that reason RI is equipped with a member adaptation mechanism, which will be presented in section 6.

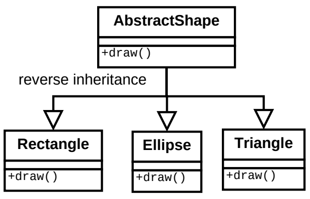

Figure 1: Capturing Common Functionalities

2.2 Inserting a Class Into an Existing Hierarchy

RI in combination with OI allows the insertion of a new class "between" two classes of an existing hierarchy. Such a class we call an *amphibious* foster class. All members that exist in its parent(s) must be also exherited from its heirs; we call them amphibious as well. In RIEiffel we had reasons to keep the original direct inheritance link instead of putting the new class literally in between — as in figure 2. However, this is impossible in Java, C# and other single-inheritance languages, and not workable in C++ because it could change the semantics. In figure 2, the classes *Shape* and *Rectangle* must have existed first. Afterwards the decision is taken to add a new abstraction layer, namely to insert class *Parallelogram* as an heir of *Shape* and as a parent of *Rectangle*.

2.3 Extending a Class Hierarchy

RI and OI can be combined freely in extending an existing class hierarchy. Figure 3 gives an example of that. The original class hierarchy there was composed of three classes: *Parallelogram* and its heirs *Rectangle* and *Diamond*. Later on, a new class *AbstractShape* was created by RI, and then *Ellipse*, *Circle* and *Triangle* by OI. If the designers will later wish to add, e.g. a class *Polygon*, it can be easily defined by OI from *AbstractShape* and by RI from *Parallelogram* and *Triangle*.

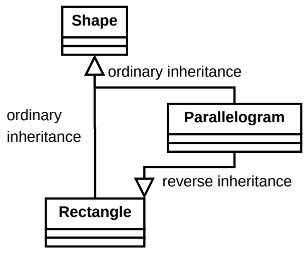

Figure 2: Inserting a Class Into an Existing Hierarchy

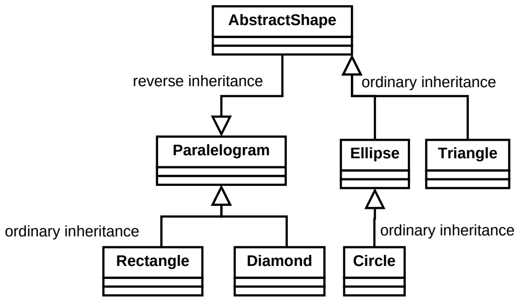

Figure 3: Extending a Class Hierarchy

RI makes possible also other kinds of class reuse, like:
i) reusing partial behavior of a class — by factoring selectively a subset of the heir members into the parent;
ii) creating a new supertype for classes that must be handled homogeneously in special contexts;
iii) decomposing and recomposing classes — using RI, behavior can be extracted from a heir, stored in the parent and then using multiple OI it can be composed with another parent resulting a composed heir;
iv) facilitating the integration of heirs into design patterns [4] (Adapter, Template Method, etc).

3. THE GENERALIZING CLASSIFIER

3.1 The Nature of the Classifier

In this section we will analyze what kinds of classifiers (in the UML sense) are available in the four discussed OOPLs, and how they can be combined in RI. We must ignore genericity in this paper, although it is supported in the current versions of all those languages.

In Eiffel and C++ there are only classes. They can be either concrete (effective) or abstract (deferred), but this difference does not restrict OI, so it must not restrict RI either. In Java and C# there are two kinds of classifiers: classes and interfaces. Both the *extends* and *implements* relationships in Java can be regarded as inheritance. Thus we have the restriction that an interface cannot inherit a class; the same holds in C#. For RI this implies, of course, that a class cannot exherit an interface, while the inverse is possible. Where needed, we will speak about *foster interfaces* (of classes or/and other interfaces) in analogy with foster classes.

In C++ classes cannot be prevented from having heirs, but such a restriction can be stated in Java (*final*) and C#

(*sealed*) This possibility has recently been added also to Eiffel (*frozen*), but it does not affect so-called non-conforming inheritance (another new feature in Eiffel).

It seems obvious at first that a foster class could not logically be declared as final. However, that might be allowed with the effect that the class cannot have any heirs except those from which it has been formed with RI. — It would be possible to add another keyword that would prevent foster classes to be built from a class A. This is the inverse of final, but here it seems completely natural that A can have parents by OI. We have not added such a keyword to RIEiffel.

3.2 Interaction with Multiple Inheritance

A very important language feature is the presence of MI. Generally speaking, MI adds a lot of complications to the language, like member sharing vs. replication dilemmas and dynamic binding ambiguities. In Eiffel the MI approach is attribute-based (member-based), while in C++ it is subobject-based [9]. This difference is important for ancestors that are inherited over more than one inheritance path ("diamond inheritance"): sharing or replication can be chosen in Eiffel for each member separately (based on renaming); in C++ the choice is made for the whole ancestor class subobject (replication unless the inheritance is declared "virtual"). Also in RI we need to declare whether sharing or replication is intended in a diamond.

Complex MI class hierarchies that combine both sharing and replicating inheritance are difficult to understand and handle in both languages. In C++ the combinations can become truly anomalous [10]; therefore it seems reasonable to allow an RI extension of C++ to create only purely sharing and purely replicating hierarchies.

In Java and C# MI is enabled only for interfaces, so a class can have only one parent class but several parent interfaces. Therefore, in RI only one foster class F can be defined for any class A, and F *must* be inserted between A and its original parent — a structure like in figure 2 is not possible. As a consequence of this, multiple RI is possible only from subclasses of a common parent class.

MI from interfaces is much simpler and less problematic than MI from classes. However, Java (but not C#) has the problem that methods inherited from different parent interfaces are always unified if they happen to have the same signature. It seems that RI to foster interfaces can be quite useful; the restriction that interfaces cannot have fields is a real drawback, but exheriting method implementations (bodies) is seldom useful except in single RI (see section 5).

4. THE FACTORING MECHANISM

We consider the factoring mechanism to be the core of RI. It consists in determining the **exheritable** members from the heirs and then selecting which of them are actually exherited to the foster class. For example, in figure 1 the method *draw()* is common to all heirs and is factored to the parent. In principle, if there is a set consisting of one member in each heir that are all regarded to have the same semantics, this set corresponds to an exheritable member for the foster class. The major practical issues are discussed in the following subsections.

4.1 Signature Compatibility

In order to have an exheritable member, the corresponding members of each child class should preferably have compatible signatures. The signature of a field consists of name and type, and that of a member consists of name, parameter types and result type. In Eiffel, the members' assertions (pre- and postconditions) must also be considered, but we cannot discuss them in this paper.

By default, members having same names are taken as candidates to be exherited. In practice, however, very often there are *name conflicts* of two opposite types [6, 11]: i) "lost friends" — members having the same semantics but different names; ii) "false friends" — members having different semantics but the same name. In Eiffel the built-in renaming mechanism is an immediate solution for solving such conflicts also in of RI. In the other considered OOPLs special adaptation techniques must be used, which will be presented in section 6.

The rules for compatibility of the types (of a field, parameters and result) depend on the language, and are discussed next.

4.2 Type Compatibility

The type compatibility rules for OI in a language determine the rules that must be set for RI. For the result type of a method, Eiffel has always had the covariant rule: the type in an heir's method must be the same or a subtype of the parent's method. This principle has later been adopted also in C++, Java and C#. Eiffel has the same covariant rule also for the types of fields method results, while the other languages require nonvariance, i.e., the same type in parent and heir.

For RI in Eiffel the above implies that in order for the corresponding members in the heirs to be exheritable together, each corresponding set of types in their signatures must have at least one common supertype. The corresponding type in the exherited member of the foster class must then be such a common supertype. For RI in the other three languages the same rules holds for method results; field and parameter types must be exactly the same in all heirs, and will then be the same also in the foster class:

It is beyond the scope of this paper to discuss the actual type systems of the four languages, except giving a couple of short notes. Only in Eiffel are all types based on classes, while the other languages have also primitive data types. Genericity is a language feature that complicates the type system considerably. It has always been there in Eiffel, is an old feature in C++, and has been more recently added to Java and even to C#. — These issues cause a lot of work in the exact definition and implementation of RI, but they do not affect the principles.

4.3 Member Selection

An important aspect is the programmer's selection of members to be actually exherited, because not all exheritable members may be needed in the parent. In RIEiffel we defined four keywords for the selection: i) **all** — to select all exheritable members; ii) **nothing** — to select no member; iii) **only** — to select an explicit list of members; iv) **except** — to select all exheritable members except an explicit list. Choices iii) and iv) may be difficult to express for members that have different names in different heirs. In Eiffel the renaming mechanism solves this problem, while in the rest of the analyzed OOPLs the adaptation mechanism presented in section 6 is needed.

Note that if the foster class is amphibious (see subsection 2.2), a member that is also inherited from its parent(s) cannot be excluded.

Here we did not yet take into account the impact of the protection mechanisms of the analyzed OOPLs. This was not an issue in RIEiffel, because in Eiffel the accessibility of a member in an heir class can be arbitrarily different than in its parent(s).

5. MEMBER IMPLEMENTATION

In OI superclasses are typically more abstract than subclasses: inherited abstract methods are implemented (effected) sooner or later. Consequently, it seemed natural to us that in RI all exherited methods will be abstract by default. The exception are amphibious methods (section 2.2), whose implementation in the parent class is inherited by default. Of course, it is possible to write a new implementation in the foster class if desired.

The interesting case is when the implementation of a method is exherited (imported) from an heir class. In RIEiffel the **moveup** keyword is used in this sense. For the other OOPLs some similar syntax would be needed.

Regardless of programming language, importing a method implementation from one heir to the parent is often impossible. The reason is that all other members that the method uses (directly or indirectly) must also be exherited for the method to work. In multiple RI, some of those members might not be even exheritable. These facts restrict severely the possibilities of implementation exheritance. Additional, although solvable, difficulties can be caused by the use of keywords like **this** (**current** in Eiffel) and **super** (**precursor** in Eiffel) in the exherited code.

In Eiffel, the choice between abstract and concrete concerns even fields, because an inherited method (concrete or abstract) can be redefined in OI also as an attribute with the same signature. Of course, this is possible only for a result-returning method without parameters. This issue does not exist in the other discussed languages.

6. MEMBER ADAPTATIONS

As mentioned in subsection 4.1, semantically corresponding members in different heirs can often have different signatures, so that they are not exheritable by default. A member adaptation mechanism is needed to cater for such situations; the adaptation always happens when a member of an heir class object is accessed through a reference of the foster type.

The easiest difference to handle is a name conflict; only renaming is needed, and in Eiffel it already exists in the standard language (see subsection 4.1). In any case, this adaptation can be done at compile time.

Not much more difficult is the situation where a method's parameter lists are otherwise the same in all heirs, but the order of the parameters varies. One of those orders must be chosen for the method in the foster class, and for all heirs with a different order the method call must be reordered at run time.

Our adaptation mechanisms in RIEiffel support also some more advanced adaptations, which are not purely syntactic but concern also semantics. One is type conversion for fields, parameters and results. Another is scale conversion, or more generally value conversion. Both are illustrated in the following example (figure 4). The RI features there are given just as annotations similar to the existing RIEiffel syntax, embedded as comments in Java code. All access modifiers have been omitted for simplicity. First there are the two classes *Rectangle* and *Ellipse*, and the abstract foster class *Shape* is then built by RI. We know that the methods *size*

```
01 class Rectangle {
02   void size(int s) {...}
03   double area() {...} // m^2
04   ... // other members
05 }
06 class Ellipse {
07   void scale(int f) {...}
08   float surface() {...} // dm^2
09   ... // other members
10 }
11 //@foster
12 abstract class Shape { //@exherits Rectangle, Ellipse
13   void scale(Integer factor)
14   {//@adaptations
15    //{Rectangle}{size(factor.intValue())}
16    //{Ellipse}{scale(factor.intValue())}
17   }
18   double area() // cm^2
19   {//@adaptations
20    //{Rectangle}{return area()*10000;}
21    //{Ellipse}{return surface()*100;}
22   }
23 }
```

Figure 4: Adaptation Example

(line *02*) and *scale* (line *07*) correspond to each other, although they have different names. We exherit them to the foster class under the name *scale* (line *13*); we also change the type of the parameter, and therefore add the conversions on lines *15* and *16*. These show how a method call made through a *Shape* reference is modified and forwarded to an instance of *Rectangle* or *Ellipse*.

The methods *area* (line *03*) and *surface* (line *08*) also have the same semantics, but different names and result types. In the foster class we take the signature of *area* from class *Rectangle*. The result type of *surface* from class *Ellipse* would be automatically convertible to that of *area*, but we add a twist that makes the situation more interesting. The method *Rectangle.area* returns a value in square meters, *Ellipse.surface* in square decimeters, and *Shape.area* in square centimeters. Therefore we need the adaptations on lines *20* and *21* to get the necessary scale conversions.

7. RELATED WORK

There are surprisingly few previous papers that are directly relevant to our research on RI; virtually all we have found were referenced in section 1. On the other hand, there are many published and even implemented approaches intended to add reusability and flexibility to traditional inheritance. Some rather recent ones are traits [13], classboxes [1], and expanders [15]. Even paradigms like aspect-oriented programming (AOP) [5] aim at increasing reusability and flexibility, but AOP is not focused on inheritance.

Traits [13] are reusable and composable parts that can be used in building classes. Their original major purpose was to add a special kind of multiple inheritance to languages that only support single inheritance (especially Smalltalk). Since traits must be written before classes, the approach is rather opposite to RI.

Classboxes [1] allow adding and replacing methods in a class. The changes made by a classbox are only visible in that classbox or other classboxes importing it.

The expander is a construct that supports object adaptation [15]. Classes are adapted in a syntactically non-intrusive manner by adding new fields, methods and super-interfaces. Each client can adapt the same class in different contexts independently with different expanders.

Both classboxes and expanders modify existing classes for particular use contexts. The purpose of AOP is also to modify the semantics of classes (without touching their source code), but it is especially targeted for crosscutting concerns in software systems. This is in strong contrast to RI as presented in this paper: the semantics of existing classes are carefully preserved, but classes from different contexts can be taken and adapted to be used together in a homogeneous manner.

We have actually suggested in an earlier paper [2] even the possibility of defining new members in a foster class, with the effect that these members would be "retrofitted" also to all existing descendants. However, we have not continued research in this direction yet, since a consistent definition and implementation of RI has proved to be challenging enough without such additions.

8. CONCLUSIONS AND FUTURE WORK

We emphasize that RI is intended to be the symmetrical counterpart of OI, so RI should not be able to create class hierarchies which are not possible to achieve with OI, and vice versa. The main exception are adaptations, which are necessary to compensate for minor syntactic and semantic differences between corresponding members in exherited classes that have different origins. Such adaptations are not needed in OI, except for Eiffel's renaming facility.

This paper is our first step toward adding RI to other OOPLs besides Eiffel. We took into account only the main language mechanisms, but a deeper analysis is necessary — already in the case of Eiffel we learned that the devil really is in the details. On the current level of discussion, the factoring mechanism seems quite similar for all four studied OOPLs.

Whether and how a base language supports MI is an important factor for a possible RI enhancement. On one hand, taking MI into account makes RI more complicated, but on the other hand the lack of MI restricts the possibilities for RI. C++ is already such a complicated language (not only concerning MI) that integrating RI into it will be a tough challenge, if not worse. Java and C# look like much easier targets. Due to the plethora of tools and models available for Java we intend to define and implement the RI semantics by translating the extended source code into pure Java source code.

Another, quite ambitious, future goal is to describe a generic RI semantics, which can be configured with appropriate parameters to extend a particular language with RI. This means that each described mechanism must be abstracted to be general enough to fit to a large set of OOPLs.

9. ACKNOWLEDGMENT

We would like to thank Günter Kniesel from the University of Bonn for the valuable advice given in the RI implementation for Eiffel.

10. REFERENCES

[1] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module supporting local rebinding. In *JMLC'03: Proceedings of Joint Modular Language Conference*. Springer, 2003.

[2] C.-B. Chirilă, P. Crescenzo, and P. Lahire. A reverse inheritance relationship dedicated to reengineering: The point of view of feature factorization. In *Proceedings of MASPEGHI Workshop at ECOOP 2004*, Oslo, Norway, June 2004.

[3] P. H. Fröhlich. Inheritance decomposed. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of European Conference on Object-Oriented Programming*, pages 220–242. Jyväskylä, Finland, 1997.

[6] T. Lawson, C. Hollinshead, and M. Qutaishat. The potential for reverse type inheritance in Eiffel. In *Technology of Object-Oriented Languages and Systems (TOOLS'94)*, 1994.

[7] B. Meyer. Eiffel: The language. http://www.inf.ethz.ch/ meyer/, September 2002.

[8] C. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 407–417. 1989.

[9] M. Sakkinen. Disciplined inheritance. In *Proceedings of ECOOP '89 (European Conference on Object-Oriented Programming)*, pages 39 – 56, 1989.

[10] M. Sakkinen. A critique of the inheritance principles of C++. *Computing Systems*, 5(1):69 – 110, Winter 1992.

[11] M. Sakkinen. Exheritance - Class generalization revived. In *Proceedings of the Inheritance Workshop at ECOOP*, Malaga, Spain, June 2002.

[12] M. Sakkinen, P. Lahire, and C.-B. Chirilă. Towards fully-fledged reverse inheritance in Eiffel. In *Proceedings SPLST 09 and NW-MODE 09*, pages 132–146, Tampere, Finland, 2009.

[13] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP*, pages 248 – 274, Darmstadt, Germany, July 2003.

[14] M. Schrefl and E. J. Neuhold. Object class definition by generalization using upward inheritance. In *ICDE*, pages 4–13. IEEE Computer Society, 1988.

[15] A. Warth, M. Stanojevic, and T. Millstein. Statically scoped object adaption with expanders. In *Proceedings of Conference on Object-Oriented Programing, Systems, Languages and Applications (OOPSLA'06)*, Portland, Oregon, USA, 2006.