Nordic Journal of Computing

TOWARDS FULLY-FLEDGED REVERSE INHERITANCE IN EIFFEL

Markku Sakkinen
Department of Computer Science and Information Systems
University of Jyvskyl
sakkinen@cs.jyu.fi

Philippe Lahire
I3S Laboratory
University of Nice – Sophia Antipolis and CNRS
Philippe.Lahire@unice.fr

Ciprian-Bogdan Chirilă
Politehnica University of Timisoara
chirila@cs.upt.ro

Abstract. Generalization 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 that is extended. 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.

ACM CCS Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features — inheritance; D.3.2 [Programming Languages]: Language Classifications — object-oriented languages, Eiffel; D.2.3 [Software Engineering]: Coding Tools and Techniques — object-oriented programming; D.2.2 [Software Engineering]: Design Tools and Techniques — object-oriented design methods;

Key words: Eiffel, reverse inheritance, generalization, hierarchy evolution, refactoring

1. Introduction

Generalization is widely used in object-oriented (OO) modelling and design, but it is not available on the programming level in any widely used language or system. We propose to extend object-oriented languages with a new

Received October 1, 2010.

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).

Clearly, 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 for reverse inheritance 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 costs (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 (Meyer [2006], ECMA International [2006]) existed yet, we based our extensions on the stable old version often known as Eiffel 3 (Meyer [1992]). 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, when using them the first time.

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. The keyword **all** specifies that all *features* (the common superconcept 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

```
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

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 11.

We already have a quite comprehensive proof-of-concept implementation of our RI extension for Eiffel, by a transformation to standard Eiffel. 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` . Unfortunately, some of our adaptations cannot be done with such a transformation approach, but would need the Eiffel compiler itself to be modified.

2. Previous research

The earliest article we have found that discusses a concrete generalization mechanism is Schrefl and Neuhold [1988], 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 Qutaishat *et al.* [1997] 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 gen-

eralization 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 Pedersen [1989]. 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 Lawson *et al.* [1994], 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 Pedersen [1989] and Lawson *et al.* [1994] 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). We would expect that to be more common in practice than defining a foster class as a root class.

The workshop paper Sakkinen [2002] was written unaware of Lawson *et al.* [1994], so the new term 'exheritance' was coined there. It is quite optimistic about RI and suggests several new ideas. All of those are not included in our Eiffel extension, but could be relevant if we did not want to stay fully downward compatible with standard Eiffel (see Rule 1 in Section 3).

The workshop paper Chirilă *et al.* [2004] 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 Lawson *et al.* [1994] and Sakkinen [2002], 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 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 Lawson *et al.* [1994]).

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

Fourthly, the reverse inheritance relationship is intended to be the exact inverse of ordinary inheritance. This means that it should be as completely interchangeable with ordinary inheritance as possible. In the new version of Eiffel (ECMA International [2006]) 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 RI may well create new inheritance *paths* between existing classes, but only for existing ancestor-descendant pairs (see Section 6).

The paper Sakkinen [2002] 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, that would change the semantics of many programs even if they do not use RI, because Eiffel has language constructs whose effect depends on the dynamic type of a variable, e.g., the assignment attempt.

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 seven 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 Lawson *et al.* [1994], 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 Lawson *et al.* [1994]). 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** in a foster class, in the same way as we use a clause **inherit** in order to reuse and to extend the behaviour of one or several classes. We did not take the keyword **adopt** from Lawson *et al.* [1994], because we have introduced **adapt** and **adapted** (see later), and wanted to avoid confusion.

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

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 (Sakkinen [2002]). 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, as well as the the type of a parameter and 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 exceptions 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 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 Schrefl and Neuhold [1988] 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* (which corresponds to *Object* in many other languages) 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.

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

```
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

of *POINT* (line *12*).

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).

Renaming and redefinition in OI exist already in standard Eiffel, but adaptation in our sense has no counterpart in OI (see Section 4). The word 'adaptation' is used in a wider sense in Eiffel specifications: it includes renaming, redefinition and undefinition. 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.

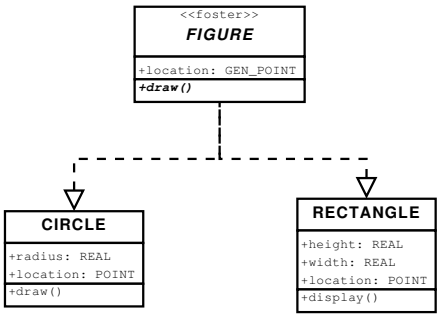

Fig. 3: Class diagram for Figure 2

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. 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. — Such a foster class we will call '*amphibious*', using a metaphor from biology: the features of these classes come partly from above ("the land") and partly from below ("the water") in the hierarchy. When needed, we call other foster classes 'non-amphibious'.

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 define *MOVABLE_FIGURE* as their common parent, 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).

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 con-

```
01 deferred foster class MOVABLE_FIGURE
02    inherit FIGURE
03       redefine location
07       end
04    exherit
05       MOVABLE_CIRCLE
06          moveup location
07          end
08       MOVABLE_RECTANGLE
09          rename display as draw
10          end
11    feature
12 end – class MOVABLE_FIGURE
```

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

version 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 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 abbreviation 'PC' for the existing parent class(es). We assume for simplicity 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. In particular, if the feature has retained its original type in any EC, it cannot be changed in the FC either.

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

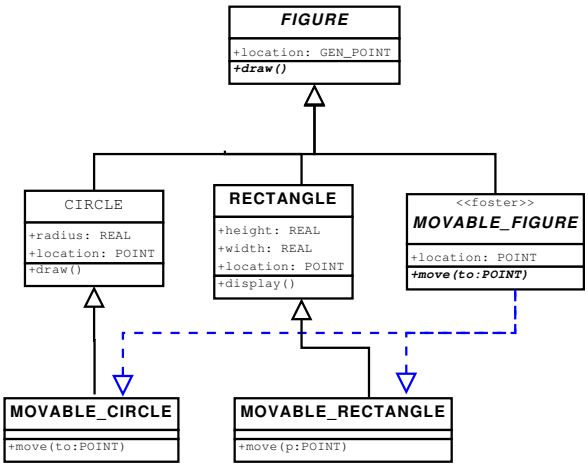

Fig. 5: Class diagram for Figure 4

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 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. Assertions in foster classes

Assertions are a key feature of Eiffel, and their treatment in RI should be compatible with the choices made initially in the language. We concentrate on the most important and interesting kinds of assertions, namely method pre- and postconditions. Class invariants are equally important, but they can be regarded as postconditions that are checked for all methods of the class.

In Eiffel, the semantics of the assertions for a redefined feature (in OI) were earlier defined as follows (Meyer [1997], p. 573):

- A routine redeclaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger.

This was enforced in a simple way: in a redefinition one could only specify an *incremental* precondition that was ORed (by the compiler) with all preconditions in the parents, and an incremental postcondition that was ANDed with all postconditions in the parents.

In Lawson *et al.* [1994] the authors directly inverse for RI the rules defined for OI, thus:

- The precondition of a feature in a foster class must be no weaker than the precondition of each corresponding feature in the classes it adopts.
- The postcondition of a feature in a foster class must be no stronger than the postcondition of each corresponding feature in the classes it adopts.

They conclude then that all preconditions should be ANDed together and all postconditions ORed together. This leads to difficulties explained in that paper: assertions of a foster class may refer to features that do not appear in that class. This is especially a problem for preconditions since the only possible precondition then is **false**, which will always fail. The exheritance of a method would therefore often be considered impossible, especially if it has elaborate preconditions in some heir class. The only possible postcondition is **true**, which will never cause so bad problems, but is unnecessarily weak in many cases.

Amphibious features were not considered at all in Lawson *et al.* [1994] (see Section 6). The above problem can be avoided very simply for an amphibious method: we inherit the post- and precondition from the PC (or PCs) by default, so they automatically fulfill the rules. Also, if the body of the method can otherwise be moved from some EC, its pre- and postconditions there already obey the standard Eiffel rule and thus will not cause problems Sakkinen [2002].

For non-amphibious methods the problem with pre- and postconditions is real. We did actually work out a solution, getting a clue from the refined treatment of preconditions in *multiple inheritance* in ECMA International [2006] (Subsection 8.10.5.). It is based on the following deduction: If the method is called on an object of the heir class through a variable of a parent class type, only the precondition of *that* parent will be ensured to hold. Any other inherited precondition might not be satisfied, and then the corresponding postcondition need not get satisfied either.

Unfortunately, our solution is rather complicated and would typically require very clumsy code in clients of the foster class. Therefore we do not present it here, but accept the fact that precondition conflicts will sometimes make the exheritance of some methods impossible in multiple RI.

8. Further aspects

In Section 6 we discussed the situations in which the features inherited by the ECs from a common ancestor are necessarily exherited by the FC because it also inherits that ancestor. The ECs can also have common features inherited from ancestors that the FC does *not* inherit. Such situations, like the former ones, had not been considered at all in the older papers Pedersen [1989] Lawson *et al.* [1994]. However, in Sakkinen [2002] it was taken as granted that inherited features can be exherited, just like exherited features can be inherited.

However, this is not self-evident when an existing language is extended. Namely, suppose that a feature in the ECs is an attribute inherited from a parent class PC, and it should be exherited into the FC also as an attribute. From the OI viewpoint, this means that attributes inherited from unrelated parents should be unified. This was not allowed in earlier versions of Eiffel for attributes, although it was for routines. The restriction was lifted in Eiffel 3, so we can allow the exheritance of inherited features in all cases.

Note that the exheritance of inherited features can be achieved nicely with an amphibious class if we want to get *all* features of some common ancestor into the FC, as in the example of section 6. In other situations, it would be possible to avoid it by first defining suitable auxiliary FCs and finally an amphibious FC, but that would often be tedious and complicated.

The hardest problems have appeared with *repeated inheritance*. Such problems were analysed already in Lawson *et al.* [1994], but no convenient general solution was found.

In one example of Lawson *et al.* [1994] the class *RECTANGLE* has a feature *boundary* and *CIRCLE* has a feature *circumference*. They also have a common subclass *CIRCULAR_RECTANGLE*, which thus has both of these as distinct features. When the foster class *FIGURE* is defined, these features are exherited to become one feature *perimeter*.[1]

[1] This example serves as an illustration, but it is quite contrived: we cannot imagine what a circular rectangle could be.

Now, when an object of class *CIRCULAR_RECTANGLE* is assigned to a variable of type *FIGURE* and the feature *perimeter* invoked through that variable, the problem is whether the dynamic binding should select *boundary* or *circumference*. The same problem would appear with OI; in Eiffel one of the alternatives must then be designated with a **select** clause when *CIRCULAR_RECTANGLE* is defined.

It is claimed in Lawson *et al.* [1994] that **select** clauses would not always work in RI: if there are more than two ECs, there can be common descendants of two of more of them that do not inherit just the EC whose feature was selected. In fact, we could add a third EC *TRIANGLE* to their example and select from it — there would then be no applicable selection in *CIRCULAR_RECTANGLE*.

The paper Lawson *et al.* [1994] suggests "a simple, though incomplete, solution": the order in which the ECs are given in the definition of the FC should be the preference order for selecting features. The authors consider it incomplete because the same preference order might in some cases not be desired for all common descendants of the ECs. We believe that different preference orders might quite often be desired also for different exherited *features*, thus this approach really does not look suitable.

We propose a better solution: if a foster class can in such cases contain extended **select** clauses specifying indirect descendants, all ambiguities can be resolved, e.g.:

select CIRCULAR_RECTANGLE**.**boundary

Admittedly, this could become tedious in larger class hierarchies.

The above problem does not concern any descendants of ECs that are defined *after* the FC. This is because the class hierarchy from their viewpoint is exactly equivalent to one built by ordinary inheritance only, and so ambiguities can be resolved by normal **select** clauses.

When we studied how more tricky MI structures should be handled in RI, we found out an interesting anomaly in traditional Eiffel, i.e., in OI. For instance, suppose that two classes A and B have a common parent P, and two child classes C and D have both A and B among their parents. If a feature f inherited from P is replicated in C and D, the existing simple **select** clause is not sufficient to disambiguate between its versions in common descendants of C and D. — It seems that the rules of the new Eiffel standard ECMA International [2006] indirectly prohibit such inheritance structures. However, no compiler conforming to the standard is available yet.

Yet another interesting aspect is how to handle the keyword **precursor** in a method body of an exherited class. Its meaning is similar to **super** in many other languages, but it can only refer to an inherited version of the *same* method. This implies that if **precursor** appears in an exherited method, the method is necessarily *amphibious* in the foster class.

In Figure 6 we present an extremely simple example of this situation, derived from the one of Figure 5.

Method *move* exists in class *FIGURE*, and is redefined in the subclass

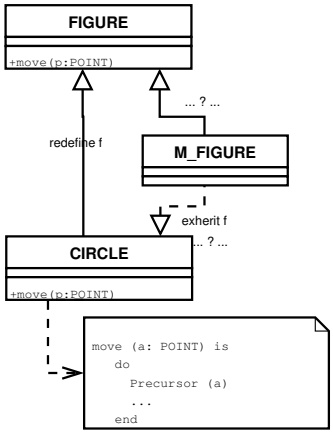

Fig. 6: Example of the *precursor* keyword

CIRCLE so that the new version of the routine body contains a statement *precursor*. This implies that the method must be effective (have a body) already in the parent class. Class *M_FIGURE* is then defined as an amphibious foster class, and we study what the semantics of *precursor* should be to equal the corresponding OI structure.

There are four different cases for the version of *move* in *M_FIGURE*, but they should not affect the behaviour of instances of *CIRCLE* (Rule 3 in Section 3):

(1) The implementation of *move* is the one of *FIGURE*, i.e., *move* is inherited from *FIGURE* and not redefined in *M_FIGURE*. Thus, there is no problem because **precursor** still refers to the same version as before.

(2) The implementation of *move* is the one of *CIRCLE*, i.e., *move* is undefined when inherited from *FIGURE*, and moved from *CIRCLE* to *M_FIGURE* (keyword **moveup** is used in the exheritance clause). Again, **precursor** refers to the same version as before.

(3) Feature *move* is deferred in *M_FIGURE*, i.e., *move* is undefined when inherited from *FIGURE* and not moved from *CIRCLE*. In this case, there is only one effective precursor version for the *move* of *CIRCLE*, namely that of *FIGURE*, so again there is no problem.

(4) The implementation of *move* is in *M_FIGURE*, i.e., the body of *move* is redefined there. In this case there are two precursor versions for the *move* of *CIRCLE*, and so a plain **precursor** is ambiguous. The ambiguity is resolved if we interpret this situation to be the same as if we had the qualified reference **precursor** {FIGURE} in the corresponding OI structure.

Let us suppose a more complex situation, such that a class C has two or more parents which have different versions of some method f, and f is

redefined in *C*. Then any use of **precursor** in the redefined method *must* already be qualified with the name of a parent class. It is thus clear that a later defined amphibious FC cannot affect the interpretation of **precursor** in any way.

Generic classes have been an important part of Eiffel from the beginning, and are well integrated with other language mechanisms. Therefore, they must be handled in a convenient and natural way also in RI. To keep things simple and the main issues in focus, we restrict the treatment here to classes with only one generic parameter. We will assume that parameter to be constrained; that is no restriction, because an unconstrained formal generic is equivalent to one with the class *ANY* as constraint.

For technical reasons, the Eiffel literature does not speak about *generated* classes but generated *types*. As in several other languages including Java, but unlike Ada, they cannot be named, but they must always be referred to by the name of the generic class and the actual generic parameters.[2]

If G[B] and G[A] are two types generated from the generic class G, then G[B] is regarded as a descendant of G[A] in Eiffel if and only if B is a descendant of A. Of course, this must hold also if the relationship between B and A is caused by RI. It is also clear that exheriting a generic class G[T-¿C] to a generic foster class H[T] does not cause any new issues for RI. The other interactions between RI and genericity are not quite so obvious.

It is possible to define a generic class as an heir of a non-generic class. Therefore, we must allow a non-generic foster class to exherit a generic class G. It could seem that the generic features of G[T-¿C] (i.e., those whose type or signature depends on the formal generic parameter T) could not be exheritable, but actually they can. It suffices to choose some ancestor D of the constraining type C and substitute all occurrences of T by D in the foster class.

The converse of the above, namely defining a non-generic class as an heir of a generic class, is impossible in Eiffel; it could not even make any sense because the formal generic parameter would just disappear. On the other hand, a non-generic class can well be an heir of a *generated type* in OI. Thus we should allow a generated type to become a parent of a non-generic class in RI. How it can be done is best explained by using an example, as follows.

Suppose that we want the attribute *f: R* of an EC *A* to be generic in the FC G[T-¿C], i.e., become redefined as *f: T*. A generated type G[X] can then be considered to be a parent of the EC if and only if X, which is necessarily a descendant of the constraint type C, is also an ancestor of R. As long as these conditions hold X and C can be chosen otherwise freely. We suggest the following syntax:

foster class G[T-¿C][X]
exherit A **redefine** f . . . **feature** f: T . . .

If several attributes of the FC are defined to have the same formal generic

[2] In C++, a **typedef** declaration can be used as a convenient shorthand like for any other type, but it does not create a distinct type.

as their type, the above conditions must hold with respect to them all. In multiple RI, X and C must fulfill the conditions with respect to the types of the feature(s) in all ECs, of course.

An attribute has only one type, even if may be constructed using several layers of generic derivation. In contrast, there can appear many entities of different kinds having different types in a routine: formal arguments, result and (in an effective routine) local variables. Nevertheless, this does not yet cause an essential difference between an attribute and a *deferred* routine without pre- and postconditions. The arguments and result (if any) can be treated in the same way as attributes.

For an effective routine in the EC that should be made generic and moved to the FC, we need stronger constraints. The reason is that the body of the routine may access all entities in its scope, and normally does access at least some of them. Clearly, if the type of any such entity would change to a proper ancestor of the original type in some type generated from the generic FC, that could break explicit or implicit assumptions of the routine. The same applies to those entities that are accessed by the pre- or postcondition of a routiner.

The additional restriction is therefore the following: If the type R of any entity that would be accessed by the body, precondition or postcondition of some exherited routine also in the FC, is changed to a formal generic T in the FC, the type T must be constrained by R. If R is expressed as **like** *Current*, the constraint is the corresponding EC.

Reverse inheritance is such a "crosscutting concern" for a language that its possible mutual interferences with many other constructs must be checked. Here we only briefly list some further interesting mechanisms and aspects of Eiffel, which we have handled in our work:

THIS SHOULD BE REVISED LATER!

- *Anchored types*, which are mostly just convenient shorthand, except for the important special case '**like** *Current*'.
- *Non-conforming inheritance*, which has recently been added to Eiffel, but at the time of writing was not yet implemented in ISE Eiffel. It is rather like **protected** inheritance in C++.
- **once** features and **frozen** features.

9. Eiffel extension implementation

This section gives an overview of the Eiffel extension implementation for reverse inheritance, which is still under development Chirilă *et al.* [2009][3], but functional enough to allow some first experiments, as will be shown in Section 10.

Two main approaches could have been chosen: to modify the Eiffel compiler itself or to translate RI-Eiffel source code to pure Eiffel, which can

[3] Readers may see the progress from the website:
https://nyx.unice.fr/projects/transformer

then be handled by a standard Eiffel compiler. The handling of exheritance clauses is quite similar in both cases, but depending on the approach, it happens either in the compiler or in a preprocessing phase. The amount of work needed for the first approach would have been far too much for a small research project, so the second one was chosen.

We would expect the further evolution of an RI-Eiffel program to happen on the untransformed RI-Eiffel code. However, evolution can proceed also using the transformed pure Eiffel program, which is effectively a *refactored* program, as a baseline. Note that even this does not prevent using RI in new classes; that will only require the transformation to be performed again.

Because this choice relies on model transformation, it seemed to be a good side benefit to experiment with some existing model-driven approach. Capturing the know-how attached to RI into models (i.e., decoupling the RI specification better from a language platform) will hopefully allow us to reuse it more efficiently for different versions of Eiffel, or even other object-oriented languages. Finally, the effort made for the formalization of RI using this approach could be good preliminary work for the first approach, i.e., the implementation of built-in RI in an Eiffel compiler.

We chose a transformation language based on Prolog, in which *conditional transformations (CTs)* are specified in a formal declarative approach Kniesel and Koch [2004] Kniesel [2006] Kniesel [2008]. Our complete transformation consists of three steps: *i)* transformation from RI-Eiffel to Prolog facts, *ii)* transformation from this reification of the RI-Eiffel program into the representation (still in Prolog) of the corresponding pure Eiffel program, and *iii)* transformation of the latter reification into pure Eiffel syntax.

Regarding the specification of RI semantics, the most interesting transformation is *ii* but the two others (front end and back end) are not trivial, either. They abstract away from the concrete syntax, which varies between different Eiffel versions and compilers. Steps *i* and *iii* are designed for the GOBO open-source Eiffel compiler Bezault [2007]. Some modifications may be needed for other compilers, because they understand slightly different de facto dialects of Eiffel.

The starting point in the process is a collection of source classes in which reverse inheritance is used. They are processed by an Eiffel parser enhanced with the RI extensions. It generates an abstract syntax tree (AST), which is then represented by Prolog facts similarly to a normalized relational database (step *i*). For example, for every class there will be facts stating the formal generics, the inheritance and exheritance sections, the feature section, and so on.

To illustrate, we show in Figure 7 the Prolog facts corresponding to the foster class declared in Figure 1. The first argument of most facts is a number which identifies the fact and allows to reference it in other facts. For example, line *01* defines the cluster *gui* where the classes are located. Line *02* corresponds to the declaration of class *FIGURE* in this cluster; the last argument is a list of formal arguments, which in our example is empty because the class is not generic. Line *03* states that class *FIGURE* is

```
01 exists(cluster(1,'gui')).
02 exists(classDecl(100,1,'FIGURE',[])).
03 exists(deferredClass(100)).  exists(foster(100)).
04 exists(exheritance(401,100,201)).
05 exists(classType(201,200)).
06 exists(exheritance(402,100,301)).
07 exists(classType(301,300)).
08 exists(onlyFeature(501,401,221)).
09 exists(onlyFeature(502,401,222)).
10 exists(onlyFeature(503,402,321)).
11 exists(onlyFeature(504,402,322)).
```

Fig. 7: Prolog rules representing the foster class

deferred and a foster class. Lines *04* and *05* define the exheritance relation between the *FIGURE* foster class and the *CIRCLE* subclass. The *only* clauses from lines *08-11* are used for the selection of the features to be exherited from the subclasses.

Next (step *ii*), the model in the factbase is transformed using *conditional transformations*. A CT consists of a precondition and a transformation; if the precondition is true, the transformation is executed. For example, before exheriting a feature, it must be tested that the corresponding feature signatures from the exherited classes are compatible: their final names are equal and their formal arguments and return types are compatible. The transformations create an equivalent class hierarchy through the elimination of the RI constructs. All foster classes will thus be transformed into effective or deferred ordinary classes; after this we will call them *new classes*. All exheritance links between classes will be transformed into ordinary inheritance links.

The current implementation focuses for now on class hierarchies in which foster classes have no superclasses (except the implicit superclass) *ANY*. In particular, the following subtransformations were implemented: feature exheritance, type exheritance, genericity issues, feature moveup, feature adaptations, modifier adjustments, class relationship transformations, RI elements removal.

Feature exheritance computes the sets of features having the same name after eventual renaming and being exherited. For each such set a new feature is created in the foster class.

Type exheritance analyses the candidate feature signatures and computes the new features signatures: *i)* a signature is defined as a list, encapsulating the formal arguments types and the return type, *ii)* these types are analysed and *iii)* the new signature is computed as follows. Class types are defined in Eiffel either by a concrete class or by an instantiated formal generic. When all candidate class types are equal then the result is that common type. Class types having actual generics imply a recursive verification of

actuals compatibility and the synthesis of the representant type, since any type of Eiffel can be an actual generic. Actuals may refer to class types having actual generics but they obey the same rules as any ordinary class types. *Like* types are special types since they refer to an anchor which can be the keyword *Current*, a feature or a formal argument. When we compute the representant type we take into account the final types of the candidate anchors. If a type anchor is exherited we exherit the anchored type reestablishing its anchor in the foster class. We consider also expanded and bit types[4], which do not imply special handling.

Feature adaptations like renaming, redefining, adapt are treated by transformations. Renaming in the context of RI is translated into the context of OI by renaming back the feature to the names within the ECs. An *undefine* clause is set on every OI branch for features having an implementation in the foster class and a deferred one in the corresponding subclass. A feature redefined on a RI branch will become redefined in the context of OI. Since the RI/OI branch identifiers are not changed, the redefinition clauses implemented as *redefine* facts, will not be affected.

In order to get the flavour of one transformation, let us take one implementing the translation of the exheritance clauses into inheritance clauses. In Figure 8 between lines *03-07* the precondition is stated. It iterates on

```
01 ct(transformExhIntoInh(FosterClassId),
02 condition((
03 exists(foster(FosterClassId)),
04 exists(classType(FosterClassTypeId, FosterClassId)),
05 exists(exheritance(ExheritanceId, FosterClassId,
ExheritedClassTypeId)),
06 exists(classType(ExheritedClassTypeId, ExheritedClassId)))),
07 transformation((
08 delete(exheritance(ExheritanceId, FosterClassId,
ExheritedClassTypeId)),
09 add(inheritance(ExheritanceId, ExheritedClassId,
FosterClassTypeId))))).
```

Fig. 8: Transforming Exheritance Into Inheritance Clauses

all foster classes (line *03*) and their corresponding class types (line *04*) involved in exheritance relationships (line *05*) pointing to exherited classes types (line *06*). In the transformation sequence all iterated exheritance clauses are deleted (line *08*) and replaced with inheritance clauses (line *09*).

The final step (*iii*) is to generate pure Eiffel source code from the transformed Prolog facts. This operation is done by an unparsing utility written in Prolog, which has an unparsing rule for each fact from the logic model. For example, in figure 9 we present how the *undefine* clauses are unparsed.

[4] Bit types have been removed from the new standard.

Line *02* detects whetherthere exists any clause *undefine* for a given inher-

```
01 unparseUndefine(InheritanceId):-
02 exists(undefine(_,InheritanceId,_))->(tab(8),
writef('undefine'),nl,
03 findall(FeatName,(exists(undefine(_,InheritanceId,FeatId)),
04 exists(featureDecl(FeatId,_,FeatName))),FeatNameList),
05 printList(FeatNameList,',',),nl);true.
```

Fig. 9: Unparsing the undefine clauses

itance branch, and prints the **undefine** keyword on the output. Next, between lines *03-04* all the undefined feature names are collected in a name list *FeatNameList*. Finally, in line *05* the feature name list is printed out using the comma as separator.

The adaptations mentioned in Sections 4 and 5 (Fig. 3) cannot be done by this approach without violating Rule 4(Section 3) slightly, because no corresponding mechanisms exist in pure Eiffel and additional methods are required.

The Eiffel program that produces the Prolog facts (step *i*). currently contains about 90 000 lines of source code. The model transformations (step *ii*) and the Eiffel code (re)generation (step *iii*) together are accomplished with about 8000 lines of Prolog.

10. Experimenting with Reverse Inheritance

In previous sections we intended to show the main facets of our approach to reverse inheritance and its implementation. The aim of this section is to experiment with reverse inheritance for improving reuse in the Eiffel standard library[5]. The whole library contains around 300 classes, and its kernel around 120 classes. We take only a small excerpt of kernel classes for a case study (Figure 10). Their names should be self-explanatory enough.

Every class inherits 31 features originating from the root class *ANY*. If we do not count these, but all other inherited features together with immediate features,[6] class *DISPOSABLE* has 2 features, *DIRECTORY* 42, *IO_MEDIUM* 89, *FILE* 276, and class *PLAIN_TEXT_FILE* 287 features. We thus show only a very small part of them.

As Figure 10 shows, the classes *DIRECTORY* and class *PLAIN_TEXT_FILE* have common functionalities, but they have not been factored into common ancestor classes. The subset of features shows that both classes can open a file or directory in the reading mode (*open_read*), close it (*close*), and move the cursor to the first position (*start*). Class *DIRECTORY* allows

[5] This library comes with EiffelStudio and may be found at *http://www.eiffel.com/products/studio/*

[6] As if the hierarchy were flattened.

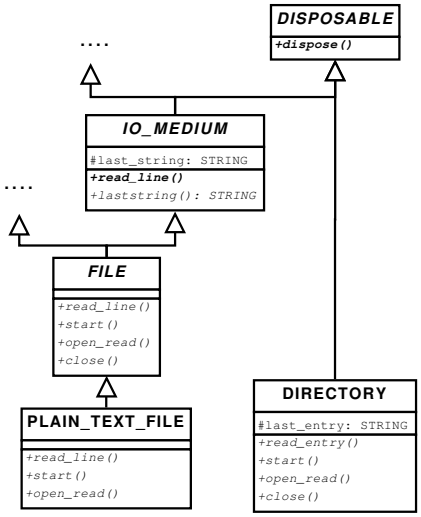

Fig. 10: Excerpt of Eiffel library

access to the names of the files or directories that a directory contains. The routine *read_entry* reads the next entry and makes it accessible through the attribute *last_entry*. Class *PLAIN_TEXT_FILE* provides *read_line* and *laststring*. They work in the same way as *read_entry* and *last_entry*: *read_line* retrieves the next line of the file and *laststring* is a function allowing access to it.

Let us suppose that we want to write a method that will print the contents of a file, whether it is a directory (like the UNIX command "ls") or an ordinary file (like the UNIX command "cat"). The appropriate file object would the most naturally be given to the method as a parameter. Unfortunately, with the existing class hierarchy the type of that formal parameter cannot be more specific than *DISPOSABLE*. We must thus inspect the type of the actual parameter and write code branches for the two cases; in addition, the case that the type is neither *FILE* nor *DIRECTORY* (or a descendant) must be handled in some way.

The erroneous third case can be eliminated if the method has three formal parameters: one of each file type and a third to tell which of them is intended. This is quite ugly and error-prone. The cleanest solution, obviously, is to write two different methods. In all three alternatives the same code must be written twice, only with one method name changed; this is something that inheritance is commonly supposed to prevent.

With reverse inheritance, we can handle this situation nicely by defining a common parent class *GEN_FILE* that exherits all common methods from the two classes, with appropriate renaming. This would be sufficient for the current purpose, but we note that two of those common methods are originally declared in the abstract class *IO_MEDIUM*. Therefore, we decide

to define first another foster class *GEN_IO_MEDIUM* and then *GEN_FILE* as its heir (see Figure 11). If the exheritance of inherited features (except amphibious ones) was not allowed, such a structure would be *necessary*.

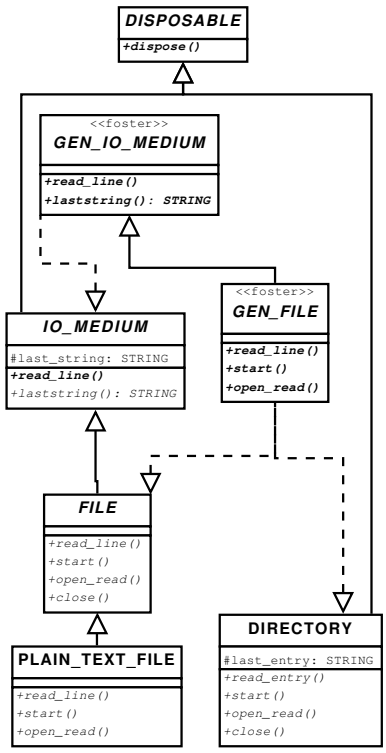

Fig. 11: Adaptation of the Eiffel Library

It seemed sensible to make *GEN_IO_MEDIUM* also an heir of *DISPOSABLE*. If there should later appear a need of descendants that do not have the feature *dispose*, we can define a new foster class that exherits all but that feature from *GEN_IO_MEDIUM*.

The source code of these foster classes is shown in Figure 12. The **only** clause of line *04*) in Figure 12 seems to exclude only the attribute *last_string*, and that on line *19* looks superfluous. However, the library classes have much more features not shown in our diagrams, as mentioned above, and many of them would also be exherited otherwise.

We applied the transformation approach described in Section 9 to this example. The result is presented in diagram form in Figure 13. Our compiler *etransformer* generated around 150 000 Prolog facts, whose total size is around 6 MBytes, in less than 7 seconds. The transformation that changed all RI into OI, and the generation of standard Eiffel classes, took about 90 seconds. The reason for the very large number of facts is that also the ancestors of the shown classes had to be analyzed. (????) We have all reason to believe that an optimized implementation fully integrated in the

```
01 deferred foster class GEN_IO_MEDIUM
02     inherit
03         DISPOSABLE
04     exherit
05         IO_MEDIUM
06         only read_line, laststring
07 end
08 end – class GEN_IO_MEDIUM

09 deferred foster class GEN_FILE
10     inherit
11         GEN_IO_MEDIUM
12     exherit
13         DIRECTORY
14             rename
15                 read_entry as read_line,
16                 lastentry as laststring
17             end
18         FILE
19         only open_read, start, read_line,
20                 laststring, close
21 end – class GEN_FILE
```

Fig. 12: Foster classes for adapting the library

Eiffel compiler would have a much smaller overhead.

This should belong better to Section 9:

The generated classes (see Figure 13) are recorded in a temporary directory. Then this directory is given to the Eiffel compiler in order to get the application ready to be executed. Presently the process is not fully automatized and the different steps (fact generation, transformation and source (re)generation, Eiffel compilation) are initiated manually by the user but this can be done easily and be fully integrated.

Not revised yet:

Summary. This experiment of our approach suggests a number of comments. It stresses that our first intent is not to refactor a class-hierarchy but to adapt its content in order to reuse it for the purpose of a given application. Nevertheless our approach by model transformation allows also to generate a new pure eiffel hierarchy without any reverse inheritance relationship. This new hierarchy could be considered as an evolution of the original one and replace it.

The rules that had been set (*e.g.* no additional inheritance path, no introduction of new features in foster classes, attribute may not be merged

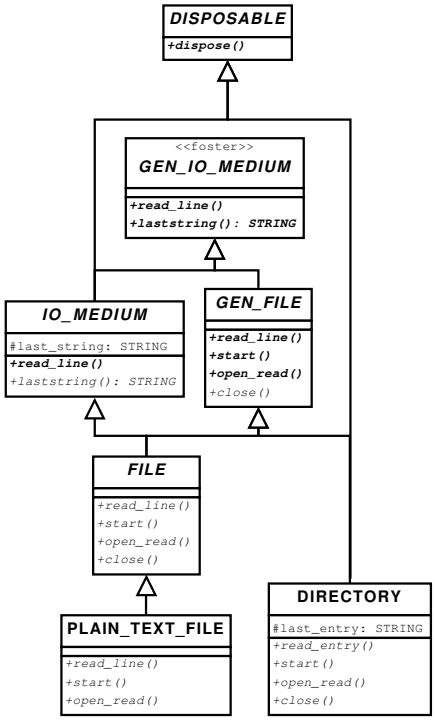

Fig. 13: Eiffel Library after transformation

when they do not have a common seed...), may appear as a limitation - for example we had been forced to create two foster classes *GEN_IO_MEDIUM* and *GEN_FILE* instead of one. But these rules are necessary in order to preserve the original semantics of existing Eiffel constructs and the strong-typing. The main outcome is that the behaviour of existing classes is not modified so that it doest not impact the robustness of existing code.

Reverse-inheritance is more than another approach for improving the reuse; it is the counterpart of ordinary inheritance. Accordingly, reverse inheritance is not designed to solve all the problems related to reusability. But, it is fully integrated in the language making it more understandable by Eiffel progammers. The various clause provided for this new relationship and their possible combinations are in line with the complexity but also the expressiveness of ordinary inheritance.

11. 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 an almost exact inverse of 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 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 intend to cover those issues in forthcoming papers.

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 (Black *et al.* [2002]). 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). Fortunately, such problems can be handled quite well by modern programming environments.

Some people who have commented on our work have claimed that reverse inheritance makes separate compilation impossible. That could indeed be a drawback in adding RI to some other languages, but in Eiffel the separate compilation of classes is not generally possible anyway.

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 Chirilă 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. Mathieu Acher and Jean Ledesma significantly contributed to it.

References

Bezault, Eric. 2007. GOBO Eiffel Project. http://www.gobosoft.com/.

Black, Andrew P., Ernst, Erik, Grogono, Peter, and Sakkinen, Markku, Editors. 2002. *Proceedings of the Inheritance Workshop at ECOOP 2002*, Number 12 in Publications of Information Technology Research Institute. University of Jyväskylä.

Chirilă, Ciprian-Bogdan, Crescenzo, Pierre, and Lahire, Philippe. 2004. A Reverse Inheritance Relationship for Improving Reusability and Evolution: The Point of View of Feature Factorization. Laboratoire I3S, Sophia-Antipolis, France, 9–14.

Chirilă, Ciprian-Bogdan, Kniesel, Gnter, Lahire, Philippe, and Sakkinen, Markku. 2009. Eiffel/RI project website. https://nyx.unice.fr/projects/transformer.

Cook, Stephen, Editor. 1989. *ECOOP '89 - European Conference on Object Oriented Programming (Nottingham, July 1989) Proceedings*, BCS Workshop Series. Cambridge University Press.

ECMA International. 2006. Standard ECMA-367 Eiffel: Analysis, Design and Programming Language. http://www.ecma-international.org.

Kniesel, Günter. 2006. A Logic Foundation for Conditional Program Transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn.

Kniesel, Günter. 2008. Detection and Resolution of Weaving Interactions. *Transactions on Aspect-Oriented Software Development* **?**, (Mar.), ?–?

Kniesel, Günter and Koch, Helge. 2004. Static Composition of Refactorings. *Science of Computer Programming 52*, 9–51.

Lahire, Philippe et al., Editors. 2004. *Proceedings of The 3rd International Workshop on MechAnims for SPEcialization, Generalization and Inheritance – MASPEGHI'04*. Laboratoire I3S, Sophia-Antipolis, France.

Lawson, Ted, Hollinshead, Christine, and Qutaishat, Munib. 1994. The Potential for Reverse Type Inheritance in Eiffel. In *Technology of Object-Oriented Languages and Systems (TOOLS Europe'94)*, 349–357.

Meyer, Bertrand. 1992. *Eiffel: The Language*. Prentice Hall.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*, 2^{nd} edition. Prentice Hall.

Meyer, Bertrand. 2006. Eiffel: The Language. `http://se.inf.ethz.ch/ meyer/ongoing/etl/LANGUAGE.pdf`.

Pedersen, C. H. 1989. Extending ordinary inheritance schemes to include generalization. In *Conference Proceedings on Object-Oriented Programming Systems, Languages*

and Applications. ACM Press, 407–417.

Qutaishat, M.A., Fiddian, N.J., and Gray, W.A. 1997. Extending OMT to support bottom-up design modelling in a heterogeneous distrubuted database environment. *Data & Knowledge Engineering 22*, 191–205.

Sakkinen, Markku. 2002. Exheritance — Class Generalization Revived. Number 12 in Publications of Information Technology Research Institute. University of Jyväskylä, 76–81.

Schrefl, Michael and Neuhold, Erich J. 1988. 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, 4–13.