From UML Diagrams to Object Oriented Code

Liliana Favre
Isistan. Facultad de Ciencias Exactas
Universidad Nacional del Centro de la
Pcia. de Buenos Aires.
Tandil. Argentina
lfavre@necsus.com.ar

Abstract

Software reuse, the use of existing software artefacts or knowledge to create new software, has two main purposes: to increase the reliability of software and to reduce the cost of software development. The SRI model for the definition of the structure of a reusable component and an object oriented method with reuse based on the model have been introduced in previous works. Our current goal is to map design artefacts to object oriented code. A rigorous method that bridges the gap between UML diagrams and Eiffel is described. The idea is to link UML diagrams with SRI components. This will enable the simulation and execution of correction tests in an independent implementation way and the subsequent transformation to efficient code.

Keywords: reusability; algebraic specifications; formal methods; object oriented programming; object oriented design.

1. Introduction

Reusability is the ability to use the same software elements for constructing many different applications. From more reusable software we may expect improvements on timeliness, efficiency, decreased maintenance effort, reliability and consistency. Making software reusable is a way to preserve the inventions of the best developers.

Most current approaches to object oriented reusability are based on empirical methods. Our work hypothesis is that *the development of rigorous methods for their systematic reuse permit building "correct" and efficient object oriented formal specification of reusable components and the software*. If, instead of being developed for just one project, a software element has the potential of serving again and again for many projects, it becomes economically attractive to submit it to the best possible quality techniques, such as formal specification of components.

There are many benefits to applying formal specifications to software reuse:

- A formal specification describes the function of a software piece free from most implementation details.
- Formal specifications and their associated formal system provide a basis for automated reuse.

Reusable software should be retrievable. There may be no components in the software library that do exactly what is wanted; therefore, it is necessary to find one or more components that can be easily modified to do the job.

Taking into account the above the SRI model for the definition of the structure of a reusable component and a rigorous method for the reusability of object oriented software were proposed [4].

The SRI model takes advantage of the power given by algebraic formalism to describe behaviour in an abstract way while respecting the domain classification principles adopted for the design of the class libraries in object oriented languages. It allows us to describe object hierarchies at three different abstraction levels: identification, realisation and implementation.

The identification level integrates hierarchies of incomplete algebraic specifications by means of formal subtyping relations. The realisation level links hierarchies of complete algebraic specifications by means of formal realisation relations and the implementation level integrates hierarchies of object oriented class schemes, that respond to a same realisation by means of implementation relations. The letters that give the name to the model refer to these three types of relations: *subtype, realisation and implementation*. A reusability method, based in the transformation of a library of SRI components by means of renaming, restriction, composition and extension operators, was presented.

In order to show the feasibility of our approach a prototype (TAROOL) was implemented [4]. It could be refined to be a practical tool for class reuse. The prototype assists in: specification editing, analysis of algebraic specifications, specifica-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CSIT copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Institute for Contemporary Education JMSUICE. To copy otherwise, or to republish, requires a fee and/or special permission from the JMSUICE.

Proceedings of the Workshop on Computer Science and Information Technologies CSIT'99
Moscow, Russia, 1999

tion validation, component reuse and transformation of specifications to Eiffel code.

The following objective in our investigation was to integrate the SRI model with object oriented analysis and design models. UML (Unified Modeling Language) has emerged as a de-facto standard for expressing these models. It was proposed by Booch, Rumbaugh and Jacobson [17].

This language is formally described in terms of itself and is based on seven kinds of graphical diagrams. UML is a visual language to model; it does not have visual and semantic support to replace the programming languages. UML must be integrated with object oriented programming languages. Then the integration of the diagrams provided by it with the SRI model was analysed.

Which is the relation between the UML diagrams and object oriented languages? Concepts such as Class and Package exist in both the UML and in many object oriented languages. The UML diagrams structure classes starting from generalisation, association and aggregation relations. In the object oriented languages the basic relations to structure objects are inheritance and client. The first one enables expressing generalisation (the refinement of classes to subclasses), the second one permits certain types of aggregation. The object oriented languages do not contain syntax or semantics to express all kinds of relations directly. The latter is an essential difference between both.

This work proposes a method, based on the SRI model, that bridges the gap between UML diagrams and object oriented languages. The SRI model serves as nexus between these two levels.

The emphasis in this presentation is given to the integration of UML static models with the SRI model. Within the framework of our approach, two UML diagrams are of special interest: Class Diagrams and Collaboration Diagrams. The design transformation to code is done starting from a SRI component library.

2.Motivation

In the purest form of object technology, only two kinds of relations exist: client and inheritance. The client relation covers many different forms of dependency. For example, aggregation, generic dependency and reference dependency. Inheritance can be viewed as a relation between classes, which suggests the way in which classes can be arranged in hierarchies.

Meyer [12] presents an inheritance taxonomy that includes twelve different categories, conveniently grouped into three families: model inheritance, variation inheritance and software inheritance. The classification is based on the observation that any software system reflects an external model itself connected with some outside reality in the software application domain. It is distinguished as follows:

- "Model inheritance, reflecting "is-a" relations between abstractions.
- Software inheritance, expressing relations within the software, with no obvious counterpart in the model.
- Variation inheritance, which describes a class through its differences with another class" [12].

To obtain flexibility, object oriented languages provide abstract classes. They also provide mechanisms for dynamic binding and polymorphism, method redeclaration, renaming and class interface restriction. They are applied in subclass relations that extend the hierarchy. Abstract classes classify groups of related types, capture incomplete common behaviour, and play an important role in connection with dynamic binding and polymorphism. They also have purely implementation-related uses. Eiffel provides deferred classes as the mechanism to support abstract classes.

Mechanisms of object oriented languages have different semantics in class hierarchies. For example, abstract classes defer behaviour and implementation decisions in classes whose behaviour is completely defined. For example, the class ARRAYED-STACK, present in Eiffel hierarchy CONTAINER, describes stacks implemented as arrays. This class is a descendent of both STACK (a deferred class that describes general stacks) and ARRAY. STACK gives ARRAYED-STACK its abstract interface, whereas ARRAY gives its implementation. The role of two parents is definitely not symmetric.

In the component model of object oriented languages, the different types of relations are not made explicit. A programmer must distinguish the different types of relations present in a hierarchy by analysing its code. Many users of object oriented classes perceive this as one of the major obstacles to reuse. They want to understand the relations between classes and operations without having to study implementation details.

Other aspects to be considered are:

- Dynamic binding does not allow us to precisely infer either the types of objects in the arguments or the result of a method. To understand the behaviour of a class, the programmer must access the code of several subclasses that interact with one another.
- A method can be redefined in the subclasses to adapt it to a new context without preserving behaviour.
- Object oriented design distributes the program functions among several classes. Then the designer's strategies are often dispersed through several non-contiguous program segments.

UML class diagrams describe structured objects. This structure is based on different relations: generalisation, aggregation and association.

The object oriented languages express generalisation through the inheritance mechanism and certain types of aggregation through client relations. However, they do not possess an explicit syntax to express all kinds of associations. These are buried in the instance variables and in the class methods: then the structure of system is not transparent. This disturbs their maintenance and mainly their reusability.

Many existing programming languages limit implementation options. For example, EIFFEL has only one way to represent associations, namely object references in one, the other or both of the associated classes.

It is worth considering the associations as semantic constructions of equal weight to the classes and the generalisation and not only as implementation constructions. The associations allow abstracting the interaction between classes in the design of large systems. The main decisions regarding efficiency aspects are related to implementations of relations. Note that associations are important for the design of large systems because they affect the partitioning of a system in modules.

3. Background

3.1. Specifying Reusable Components

Object classes can be specified in an implementation independent way by using structured algebraic specifications of data types. The basic idea of the algebraic approach consists of describing data structures by just giving the names of the different sets of data, the names of the basic functions and their properties, which are described by formulas (mostly equations). A (many sorted) signature Σ is a pair (S, F) where S is a set of sorts and F is a set of function symbols.

We have selected GSBL+ as the specification language. This language extends GSBL [1, 2] with mechanisms for error treatment, explicit parameterisation and restriction of specifications [4].

In GSBL+, we use two algebraic specification techniques for defining the set of values belonging to the sort associated with a type. The first technique is based on a set of first order formulae and the second one on a set of equations.

The first specification technique leaves much freedom to the specifier in the elaboration of a type specification However, GSBL+ also supports another style of specification based on the "term generation principle" used in a more classical approach of algebraic specification based on equations. In this approach, it should be possible to associate a term with each value belonging to the sort. This style is sometimes difficult to adopt at the requirements level because of the identification of generators operations associated with the generation of all the values of a sort and the restriction on formulas to be equations between terms (equational logic).

The mechanism of the language that creates the new specification is the class definition (Figure 1).

In GSBL$^+$ strictly generic components can be distinguished by means of explicit parameterisation. The EXPORT clause describes which names (of sorts and operations defined inside the class) are visible from outside.

GSBL+ specifications are considered structured objects. This structure is based on two relations associated to two specification building mechanisms. The OVER relation defines which specifications are considered components of a given specification. The specification is extended by the components declared in <overlist>. Similarly, the SUBCLASS-OF relation defines which specifications must be considered subclass or superclasses of a given specification. Note that the SUBCLASS-OF relation is conceptually linked to the inheritance relation in the object oriented level and the OVER relation to the client one.

The WITH clause declares new sorts, operations or equations that are incompletely defined, i.e. there are not enough equations to specify the new operations or there are not enough operations to "generate" all values of a given sort. A GSBL+ specification is implicitly parameterised in its incomplete parts.

The DEFINE clause either declares new sorts, operations or equations, that are completely defined, or completes the definition of some sort or operation, belonging to some superclass, that was not completely defined.

A class may introduce any number of new sorts; if one of them has the same name as the class, this sort is considered the *sort of interest* of the class.

The syntax of a class specified with the second technique includes the BASIC CONSTRUCTOR clause that refers to generator operations and does not contain WITH clause.

Incomplete Specification Syntax	**Complete Specification Syntax**
CLASS class-name[<parameterlist>] **EXPORT** <exportlist> **OVER** <overlist> **SUBCLASS-OF** <subclasslist> **WITH** **SORTS** <sortlist> **OPS** <opslist> **EQS** <varlist> <equationlist> **DEFINE** **SORTS** <sortlist> **OPS** <opslist> **EQS** <varlist> <equationlist> **END-CLASS**	**CLASS** class-name[<parameterlist>] **EXPORT** <exportlist> **BASIC CONSTRUCTORS** <constructorlist> **OVER** <overlist> **SUBCLASS-OF** <subclasslist> **DEFINE** **SORTS** <sortlist> **OPS** <opslist> **EQS** <varlist> <equationlist> **END-CLASS**

Figure 1. GSBL+ Class Syntax

Local instances of a class may also be defined in the OVER clause and SUBCLASS-OF clause with the following syntax:

CLASS T
OVER A: B[s'_1:s_1......,o'_1:o_1.....]
SUBCLASS-OF C: D[s'_2:s_2; ....o'_2:o_2....] ....

in which every sort s_i and operation o_i are renamed as s'_i and o'_i respectively. For example, the OVER clause has the following effects: a class A is locally created within the scope of T; A is defined by renaming, according to the declaration, the sorts and operations. A becomes a local component of T.

As examples, we propose in Figure 2. an incomplete specification of a class *Traversable* and a complete specification of a class *Sequence*.

3.2. The SRI Model

Considering the issues described in section 2., we introduce the SRI model for the definition of the structure of a reusable component. It describes object classes at three conceptual levels: identification, realisation and implementation.

Why the SRI model? Software reusability takes many different requirements into account, some of which are abstract and conceptual, whereas others are concrete and bound to implementation properties. Reusable components must be specified in an appropriate way. For example, at more abstract levels, we need descriptions satisfying three conditions [12]:

- "They should be precise and unambiguous.
- They should be complete or at least as complete as we want, in each case.
- They should not overspecify."

Identification level reconciles the need for precision and completeness in abstract specifications with the desire to avoid overspecification.

Adaptation of reusable components, which consumes a large portion of software cost, is penalised by overdependency of components on the physical structure of data. The realisation level in the SRI model allows us to distinguish design decisions related to the choice of physical structure data.

The identification level describes a hierarchy of incomplete specifications in GSBL+ as an acyclic graph G=(V,E), where V is a non-empty set of incomplete algebraic specifications in GSBL+ and $E \subseteq V \times V$ defines a subtype relation between specifications. In this context, it must be verified that if P(x) is a property provable about objects x of type T, then P(y) must be verified for every object y of type S, where S is subtype of T [11].

Every leaf in the identification level is associated with a subcomponent at the realisation level. A realisation subcomponent is a tree of complete specifications in GSBL+:

- The root is the most abstract definition.
- The internal nodes correspond to different realisations of the root.
- Leaves correspond to subcomponents at the implementation level.

If E and E1 are specifications, then E can be realised by E1 (written E~~~>E1) if E and E1 have the same signature and every model of E1 is a model of E [8]. Every specification at the realisation level corresponds to a subcomponent at the implementation level, which groups a set of implementation schemes associated with a class in an object oriented language. This level defines implementation relations denoted by the symbol " ≈≈≈>".

CLASS Traversable [G:ANY] **EXPORT** first, rest, end **OVER** Boolean **SUBCLASS-OF** Container **WITH** **SORTS** Traversable **OPS** first: Traversable -> G rest: Traversable-> Traversable end: Traversable -> Boolean **END-CLASS**	**CLASS** Sequence[element:ANY] **EXPORT** emptyseq, put, empty?, first, rest,... **BASIC CONSTRUCTORS** emptyseq, put **SUBCLASS-OF** Collection **OVER** Boolean **DEFINE** **SORTS** Sequence **OPS** emptyseq: -> Sequence put: Sequence x element -> Sequence empty? : Sequence-> Boolean first: Sequence(s) -> element **pre**: not empty?(s) rest:Sequence(s)->Sequence **pre**: not empty?(s) .... **EQS{c:Sequence; e:element}** empty?(emptyseq) = TRUE empty?(put(c,e)) = FALSE first(put(c,e))= e rest(put(c,e))=c .... **END-CLASS**

Figure 2. GSBL+ Specifications

Eiffel was chosen as the language to prove the power of the model. It is used as a tool for the design and implementation of object oriented code. It is reflected in the powerful "Design by Contract" principle, which is based on the protection of both sides of the contract. It protects the client by specifying how much should be done, and the contractor by specifying how little is acceptable. Contracts imply obligations and benefits for both parties, and are made explicit by the use of assertions. They allow us to integrate axioms of specification levels with the implementation level.

There is a relation between the other two levels and the implementation level:

- Every incomplete GSBL+ class in the identification level is associated with a deferred Eiffel class that matches the specified incomplete behaviour.
- Internal nodes of the realisation level components, including the root, correspond to an abstract class that defers implementation in the object oriented level.
- Leaves in the realisation level correspond to complete Eiffel classes.
- The implementation level can contain classes that are not related to the specifications in the identification and realisation levels. They reflect implementation aspects.

The transformation operators for specifications in GSBL+ and their extension to SRI components were defined. The transformation operators are informally described as follows:

Syntactic renaming: changes the name of sorts or operations.

Restriction: forget those parts of a specification, which are not necessary for the actual application.

Extension: adds sorts, operations or axioms to a specification.

Composition: combines two or more specifications in only one.

[4] includes a formal description of the transformation-building operators.

In order to integrate the SRI model with UML diagrams, this work proposes to extend the identification level with an OCL view. OCL (Object Constraint Language Specification) is a formal language developed as business-modelling language within the IBM Insurance Division and it has its roots in the Syntropy method [16].

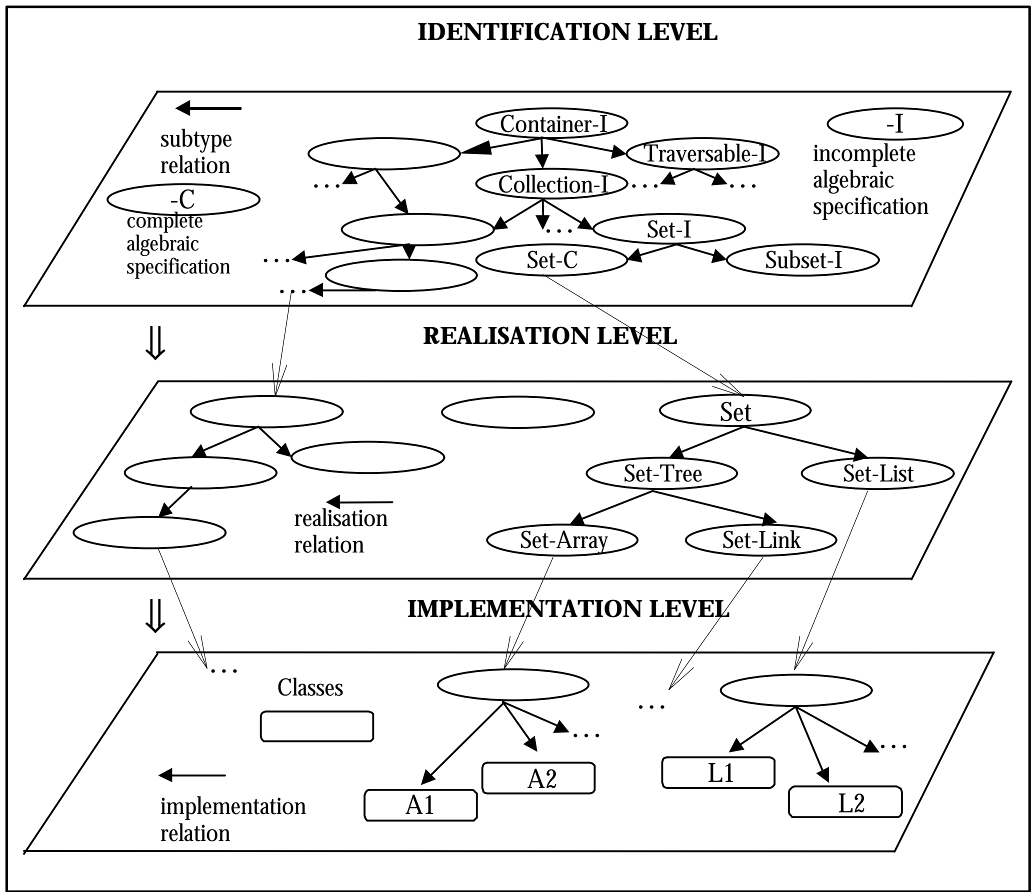

Figure 3. Component Container

The specification in OCL provides the user with the possibility of accomplishing the component identification from the UML diagrams without enforcing it to a change of specification formalism. The algebraic view will allow early validations and to automate the components' transformation.

The UML users can make use of OCL to specify constraints and other expressions associated to their models. It can also be applied to specify invariants on classes, to describe preconditons and postconditions on operations and methods, to specify constraints on operations and as a navigation language.

The integration of UML models with object oriented languages requires orienting the reuse from components library whose taxonomy is conformed to the OCL one.

Figure 3. shows a component CONTAINER. All specifications that describe "containers" are descendants of an incomplete specification (CONTAINER-I). Every leaf in the identification level is associated with a subcomponent in the realisation level. For example SET-C is a leaf that links different realisations: SET-TREE, SET-LIST, SET-ARRAY and SET-LINK. These are associated with subcomponents in the implementation level that represent concrete classes.

4. The Component Relation

A special type of SRI component is RELATION. The SRI model allows us to represent the meaning of the relation rather than its implementation. Relation taxonomy starting from the SRI model is presented here. The identification level describes the different relations through incomplete specifications. These are classified according to:

- Its degree (unary, binary, ternary and in general as n-ary)
- Its kind (Aggregation, composition, association). The degree restricts the relation type; for example one binary association could be an aggregation or an association. An aggregation can be "shared" or "composite". If it is an association it can be a qualified association or a class association.
- Its navigability, for example a binary association can be unidirectional or bidirectional.
- For its connectivity (one-to-one, one-to-many, many-to-many, etc)

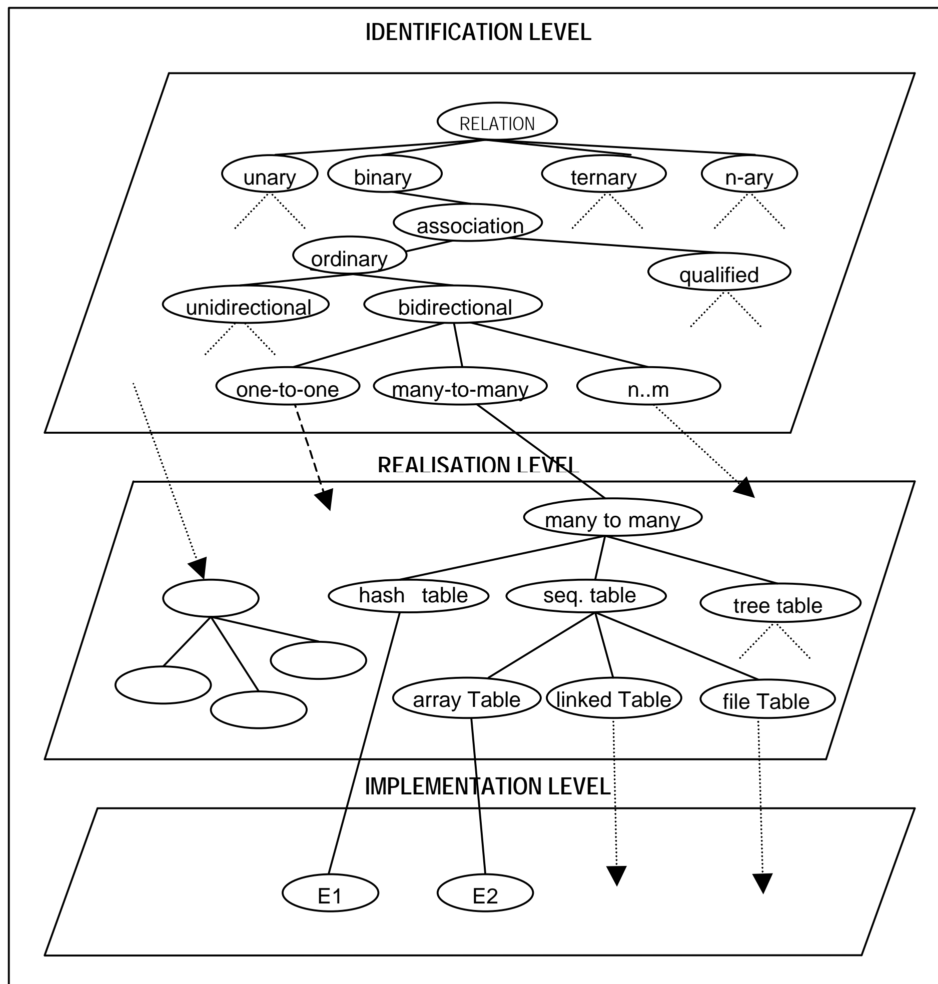

Figure 4. Component RELATION

The realisation level describes a hierarchy of complete specifications associated to different realisations. For example for an association (binary, bidirectional and many-to-many) different realisations through hashing, sequences or trees could be associated.

The implementation level associates each leaf of the realisation level to a concrete class scheme that must be instantiated with the schemes of concrete classes that intervene in the relation.

Figure 4. partially depicts a RELATION hierarchy.

5. Mapping Design to Code

The following is a description of a rigorous method for the reusability of object oriented software. The UML artefacts created during the design phase- the collaboration diagrams and design class diagrams- will be used as input to the code generation process. In UML, class diagrams statically represent the classes of the system and their relations. A collaboration diagram shows the messages that are sent in response to a method invocation.

The transformation process from design oriented class diagrams to class definitions, and from collaboration diagrams to methods is relatively straightforward.

Implementation in an object oriented language requires writing code for:

- Class definition.
- Relations (associations, aggregation, composition).
- Method definition.

5.1. Defining a Class

For each class in the class diagram it is necessary to build one algebraic specification, reusing other existing ones in the components library. The component identification is done starting from the OCL view of the identification level.

It might happen that:

- It is likely to identify one specification in the identification level of a reusable component that can adapt itself to the expected behaviour
- The class specification could be constructed from other existing ones.

The transformation process has the following steps:

Identification

Formalise the decomposition of a specification into subspecifications $E_1, E_2, \ldots E_n$. The decomposition is expressed through a transformation pattern (E), which is based on GSBL+ specifications. Specifications relate classes through the OVER and SUBCLASS-OF clauses.

For each subspecification E_i identify a component C_i (in the identification level) and a sequence $s_1, s_2, \ldots, s_n$ of GSBL+ specifications that verify subtype relations. If s_n is complete, it is associated with the root of a subcomponent CR_j in the realisation level. If s_n is not complete, select a leaf in C_i, (i.e. those specifications for which there is a path in the graph from s_n) as a candidate to be transformed.

The identification of a component is correct if renaming, restriction, extension and composition operators can modify it to match the query E_i. The sorts and operations must be connected with E_i's by an appropriate rename. The renamed version must be extended with sorts, operations and axioms. The visible signature must be restricted to the visible signature of E_i. Let OP_1, OP_2, ...OP_k be the sequence of operators applied to these transformations

Adaptation.

Select a leaf ($LEAF_j$) in the subcomponent CR_j and apply the same sequence of operators used in the previous matching to it, i.e. construct the specification $OP_1(\ldots\ldots(OP_K(LEAF_j)\ldots)$.

It is verified that $OP_1(\ldots OP_K(LEAF_j))\ldots)$ is a realisation of $OP_1(\ldots(OP_K(Root(CR_j)\ldots\ldots)$.

Select a class scheme ESQ_m in the subcomponent of the implementation level whose root is $LEAF_j$. Apply the sequence of operators OP_1, OP_2 , ...OP_K to ESQ_m, i.e. construct the specification $OP_1(\ldots\ldots(OP_k(ESQ_m)\ldots)$. It is verified that $OP_1(\ldots OP_K(ESQ_m))\ldots)$ is an implementation of $OP_1(\ldots(OP_K(LEAF_j)\ldots\ldots)$.

Possible transformations are renaming, restriction, extension and composition. Renaming and restrictions will be applied to the identified class following the definition made for these operators in the identification and realisation levels. It is worth pointing out that this is done above the class "text", not through the Eiffel language mechanisms. For example, a class A has an OVER relation with another class Reusable:

CLASS A

OVER Reusable(s1:x1; s2:x2;undefine:x3;s4:x4)

Let us consider the user selected a concrete scheme C for Reusable. The classes x1, x2 and x4 are renamed in C by s1,s2 and s4 respectively. The class x3 is deleted from C. When making the "textual substitution" it must be taken into account the fact that a transformation (renaming or restriction) can not only affect the class A but also its hierarchy.

Composition

In this phase, the subspecifications E_i and their implementations are composed, according to the transformation pattern E. The relation introduced in the pattern E using the clause OVER will be translated into a client relation in Eiffel. The relation expressed through the keyword SUBCLASS-OF in E will become a subtype relation in Eiffel.

The construction of new classes by transformation of existing ones implies access redefinition for client classes and inherit features and, by this, the creation of an interface for the new client or superclasses.

5.2. Defining Associations

In UML associations are describes as "structural relations between objects of different types" [17]. For each relation and starting from the information registered in class diagrams and the component RELATION, identify a specification (in the identification level), select a realisation (in the realisation level) and select an implementation (in the implementation level).

Implementation subcomponents express how to implement associations and aggregations. They must be instantiated with the schemes of concrete classes that intervene in the relation.

The schemes in the implementation level suggest:

- To include reference attributes in the class.
- To introduce an intermediate container or collection.

For example, a bidirectional binary association is usually implemented as an attribute in each associated class containing a reference to the related object or to a set of related objects. If the association is "many to many" the best approach is usually to implement the association as a different class, in which each instance represents one link and its attributes.

Aggregations are transformed into reference attributes. Each end of an association is called a role. Roles may optionally have: name, multiplicity expression and navigability. If a role name is present in a class diagram, it can be used as the basis for the name of the reference attribute during code generation.

Multiplicity defines how many instances of a type A can be associated with one instance of a type B, at a particular moment in time. It is usually evident from the multiplicity value in a class diagram that a class must maintain visibility to a group of others classes. In object oriented programming languages these relations are often implemented with the introduction of an intermediate collection.

5.3. Defining New Methods

The application of the extension operator expresses that the class contains new functions and axioms. Every function given on the algebraic specification will be translated into a new feature on the Eiffel class. This translation will have to take into account that Eiffel has four categories of features: variables, constants, procedures and functions

From the design class diagram, a mapping to the basic attribute definitions and method signatures is straightforward. A collaboration diagram shows the messages that are sent in response to a method invocation. The sequence of these messages is translated to a series of statements in the method implementation.

The "create" method is often excluded from the class diagram because of multiple implementations, depending on the target language. In this phase it is possible to construct an implementation for it.

A key characteristic in Eiffel is the possibility to express formal properties of a class writing assertions (preconditions, postconditions and invariants) Axioms in a formal specification language represent the constraints that the class introduces on the operations. Analysing these axioms we can derive the assertions that will be included in the Eiffel classes. Preconditions and axioms of a function written in GSBL+ are used to generate preconditions and postconditions for routines and invariant for Eiffel classes.

The programmer intervenes in the implementation of methods when it is not possible to construct them automatically.

6. Advantages of the Method

The objective of our approach is to complement the UML diagrams with a compatible, software-platform- independent specification and a rigorous method that enables mapping into efficient code.

There are many benefits to formally specify UML diagrams:

- It allows identifying ambiguous and inconsistent structure in the UML diagrams.
- Powerful analysis and validation techniques can be applied.
- Identification and retrieval of components can be partially automated.

There is, however, a wide gap between formal specifications and object oriented code. To help bridge this gap, we propose a rigorous method based on the SRI model. To be useful, an approach should allow for smooth incorporation of executable code, so that tools can map actions and operations into code efficiently.

We can summarise several advantages of our method:

- Our formal approach allows to derive knowledge from specifications in order to support assessment of solutions and to formally verify implementations against specifications.
- Most of the transformations can be undone which provides great flexibility in program development.
- The transitions between the UML diagrams and all intermediate specifications of the program are done exclusively by applying transformation operators, the correctness of which is proved with respect to the semantics of the specification language.
- Formal specification, resulting code and all transitions relating the one to the other provide a good documentation.
- It allows real design maintenance. Software developers perform maintenance and evolution on the specification of the system, not on implementations. Modifications at algebraic specification levels must be applied again to produce a new efficient implementation.

There are a couple of tools in the marketplace, which perform the mapping to code. Our contribution is more towards an embedding of the code generation within a rigorous process that facilitates reuse.

Our approach provides a basis for automatic reuse, automatic optimisations and reverse engineering.

7. Related Work

Hennicker and Wirsing [8] introduce a model for the definition of reusable components. They define a reusable component as a tree of algebraic specifications with behavioural semantics. The realisation level sub-components of the SRI model may be associated to the model described above.

A classical reference in this subject is the Larch family of specification languages [7]. A Larch specification has components written in two languages: one that is designed for a specific programming language and another that is independent of any programming language. The former kind is Larch interface languages and the latter is the Larch Shared Language (LSL).

Larch/Smalltalk was the first Larch interface specification language with subtyping and specification inheritance [3]. Other Larch interface languages with similar characteristics are Larch/Modula3 and Larch/C++. Larch/Smalltalk is notable for a clear separation of types from classes. The most interesting feature of Larch/C++ is that a class specification can have multiple interfaces. Object oriented extensions have been proposed for several specification languages (Z, VDM, etc.) [19]. There is a wide range of research that proves that software reusability can be addressed from structured algebraic descriptions [6, 10, 11, 18]. France formalises the FUSION object oriented analysis modelling techniques [5]. Most works focus on the integration between object oriented modelling techniques and formal specifications [13,14].

The language TROLL [9] was designed for use in the conceptual modelling or requirement specifications phase in the development of information systems. OASIS is a class definition language to model information systems [15].

8. Conclusions

In previous work, the SRI model for the definition of the structure of reusable components and a rigorous method, based on the model, for the semi-automatic generation of efficient object oriented code have been introduced.

To demonstrate the feasibility of our approach a prototype was implemented. Results of experiments with the prototype TAROOL reveal advantages of the reusability method as well as limitations. The object oriented paradigm offers great potential for productivity improvements but it creates unfamiliar problems for maintainers. The various uses of inheritance, binding dynamics and polymorphism can make the dependencies between classes harder to find and analyse. Real design maintenance requires automation, which depends on formalisation.

The application of building operators to SRI subcomponents and recording of the “design history” permits good maintenance. Code is generated in its purest form, omitting such mechanisms as method redefinition, direct repeated inheritance, etc. In this paper, a method for object oriented reusability that bridges the gap between UML diagrams and object oriented code was presented. From the proposed method it can be shown that it is possible to integrate UML static diagrams with SRI reusable components.

The definition of the reusable component RELATION enables to express relations (associations, aggregations, and compositions) as semantic constructions and at the same time to guide in the election of efficient implementations for themselves.

The extension of the SRI model with an OCL view for the identification level (incomplete specifications) and the definition of formal relations from specifications in OCL allow the identification of SRI components from UML diagrams. Considering that the SRI components of general use extend the repertory of OCL types, and that these are associated to the identification level, the SRI model allows a smooth transition to efficient programs.

The SRI model allows the generation of a system with a different structure from the UML models. The system specification can be mapped into different implementations at significantly reduced costs.

References

1. Clérici,S. *Un lenguaje para el diseño y validación de especificaciones algwebriacas.* PhD. thesis LSI Department. Universidad Politécnica de Catalunya. España.1989.

2. Clérici,S.; Orejas, F. "The Specification Language GSBL" *Recent Trends in Data Type Specification* 1990; April :17-20.

3. Cheon,Y, Leavens,G. "The Larch/Smalltalk Interface Specification Language." *ACM Trans. on Soft. Eng. and Meth.* 1994;Vol 3, 3: 221-253.

4. Favre,L, Diez, G. "Object oriented software reusability through formal specifications". In: Nigel Horspool(ed) *System Implementation 2000*. IFIP. Chapman Hall, 1998. pp 235-248.

5. France, R, Bruel, J, Larrondo-Petrie, M. "An integrated Object oriented and Formal Modeling Environment" . *Journal of Object Oriented Programming (JOOP)* 1997; November-December : 25-34..

6. Guerreri, E. (ed).; *Second International Workshop on Software Reusability*; 1993 Italy.

7. Guttag, J, Horning, J. "Larch: Languages and Tools for Formal Specification". Springer-Verlag.1993.

8. Hennicker, R., Wirsing, M.. "A Formal Method for the Systematic Reuse of Specification Components" Springer-Verlag, 1986. *(Lecture Notes in Computer Science N° 544).*

9. Junclauss, R, Saake, G, Hartmann, T, Sernadas, C. "TROLL-A Language for Object oriented Specification of Information Systems" *ACM Transactions on Information Systems* 1996; Vol.14, 2:175-211

10. Krueger, C. "Software Reuse" *ACM Computing Surveys* 1992; Vol. 24, 2:131-183.

11. Liskov, B, Wing, J. "A Behavioral Notion of Subtyping;" *ACM Trans. on Programming Languages and Systems* 1994; Vol 16, 6

12. Meyer, B. "Object Oriented Software Construction" Prentice Hall Object oriented Series, 1997.

13. Moreira, A, Clark, R.. "Combining Object Oriented Analysis and Formal Description Techniques In: 8th. European Conference on Object Oriented Programming, Heidelberg, Springer-Verlag, 1994. (*Lecture Notes in Computer Science N°821).*

14. Overgaard, G. "A Formal Approach to Relationships in the UML" In: Workshop on Precise Semantic of Modeling Notations, International Conference on Software Engineering. ICSE'98, Japan, April 1998.

15. Pastor, O., Ramos, I. "OASIS 2.2: A Class Definition Language to Model Information System Using an Object Oriented Approach" SPUPV-95.788. 1995 Universitad Politécnica de Valencia.

16 Rational Software Corporation. "Object Constraint Language Specification Version 1.1" 1997. (www.rational.com/uml)

17 Rational software Corporation. "UML Semantics. version 1.1". 1997 (www.rational.com/uml)

18. Schafer, W., Prieto-Díaz, R, Matsumoto, M.(ed) *Software Reusability* Ellis Horwood. 1994.

19. Wirsing, M. "Algebraic Specification Languages: An Overview. In Astesiano, E, Reggio,G, Tarlecki,A. (ed) *Recent Trends in Data Type Specifications*, Springer-Verlag. 1995, pp 351-367.