Object-oriented software reusability through formal specifications

L.M. Favre and G.M. Diez
Isistan. Facultad de Ciencias Exactas
Universidad Nacional del Centro de la Pcia de Bs. As
San Martín 57. (7000)Tandil. Argentina
Tel/Fax: 54-293-40362, E-mail:lfavre@necsus.com.ar

Abstract

Software reusability has two main purposes: to increase the reliability of software and to reduce the cost of software development. Most current approaches to object-oriented reusability are based on empirical methods focussing on reuse of implementation code. However the most effective forms of reuse are generally found at more abstract levels of software design. A model for the definition of the structure of a reusable component allowing descriptions of object class hierarchies at different levels of abstraction is presented. This model integrates algebraic specification and concrete classes in an object-oriented language. An object-oriented programming method with reuse, based on the model, is described. Our approach reconciles formal specifications with their concrete implementations.

Keywords
Reusability, object-oriented programming, algebraic specification, formal methods.

1 INTRODUCTION

Reusability is the ability to use the same software elements for constructing many different applications. An ideal software reusability technology should facilitate a consistent system implementation, starting from the adaptation and integration of "implementation pieces" that exist in reusable component libraries.

Systems Implementation 2000 R. Nigel Horspool (Ed.)
© 1998 IFIP. Published by Chapman & Hall

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. Two crucial problems arise:

- How can we retrieve a component from a software library?
- How might we adapt and integrate existing "implementation pieces" into a consistent system implementation?

In object-oriented programming, reusability is achieved by extending existing libraries. Abstract classes and inheritance are crucial for extensibility and reusability. To obtain flexibility, class hierarchies are structured through mechanisms of dynamic binding and polymorphism.

Object-oriented programming languages, such as Eiffel, Smalltalk and C++, provide large reusable class libraries. By consulting these libraries, users can develop their own domain-specific reusable components. However, most object-oriented applications have inheritance hierarchies that are imperfectly designed and methods that are imperfectly factored. Identification of components in a hierarchy and their adaptation to a particular context is therefore difficult. Some considerations are:

- Class hierarchies are structured by subclass relations that preserve some characteristics while adding new ones, as well as using parts of a class without constraint. The various uses of inheritance and dynamic binding can make dependencies harder to find and analyze.
- Object-oriented design distributes program functions among several classes. Then, the designer's strategies are often dispersed through several non-contiguous program segments.

Software reusability is difficult because it requires taking many different requirements into account, some of which are abstract and conceptual, while others, such as efficiency, are concrete. A good approach for reusability must reconcile the abstract concepts with their concrete implementations.

This work proposes the SRI model for the definition of the structure of a reusable component. It allows us to describe object class hierarchies at different levels of abstraction. It takes advantage of the power given by algebraic formalism to describe behavior in an abstract way while respecting domain classification principles adopted for the design of class libraries in object-oriented languages.

Object classes can be formally specified through structured algebraic specifications. The specification language selected for this work is GSBL (Clerici, 1989). The outstanding characteristic of GSBL is that it allows incomplete specifications, which facilitate the description of hierarchies of classes in object-oriented languages. We have extended the language to increase its expressiveness. We call our extended language GSBL+.

To facilitate component identification and code reuse, a SRI component has three different abstraction levels:

- Identification: hierarchies of incomplete algebraic specification related by formal subtyping relations.
- Realization: hierarchies of complete algebraic specifications related by formal realization relations.
- Implementation: hierarchies of object-oriented class schemes related by implementation relations.

The letters S, R and I, which name the model, refer to the relations used to integrate specifications on three levels: subtyping, realization and implementation.

In this paper we describe a rigorous method for the construction of object-oriented software based on SRI-model. The proposed approach reconciles formal specifications with their concrete implementation. Eiffel was chosen as the language to demonstrate the power of the model. In such a framework, an Eiffel application is produced semi-automatically from previously existing classes by applying specification building operators for extension, renaming, restriction and composition. However, the ideas are presented independently of a particular language.

2 MOTIVATION

Object-oriented software construction is the building of software systems as structured collections of possibly partial abstract data type implementations (Meyer, 1992). In the purest form of object technology, only two kinds of relations exist: client and inheritance. They correspond to different forms of possible dependency between two object types A and B:

- B is a client of A if every object of type B may contain information about one or more objects of type A.
- B is an heir of A if B denotes a specialized version of A.

Inheritance can be viewed as a relation between classes, which suggests the way in which classes can be arranged in hierarchies. The client relation covers many different forms of dependency. For example, aggregation, generic dependency and reference dependency.

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 behavior, 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. Object–oriented techniques support the process of working from abstract to concrete, from general to specific. Abstract classes and inheritance address such situations.

Meyer (1997) 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" (Meyer, 1997).

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 analyzing 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. We take the following considerations into account:

- Mechanisms of object oriented languages have different semantics in class hierarchies.
- Abstract classes defer behavior and implementation decisions in classes whose behavior is completely defined.
- To understand the behavior of a class, the programmer must access the code of several subclasses that interact with one another.
- Dynamic binding does not allow us to precisely infer either the types of objects in the arguments or the result of a method.
- A method can be redefined in the subclasses to adapt it to a new context without preserving behavior.
- Object-oriented design distributes the program functions among several classes. Then the designer's strategies are often dispersed through several non-contiguous program segments.

In order to overcome these problems, we adopt the SRI model for the definition of the structure of a reusable component and a method, based on the model, for reusability of object-oriented software.

3 SPECIFYING REUSABLE COMPONENTS

Object classes can be specified in an implementation independent way, starting from 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.

Traditionally, the process of building specifications has been based on horizontal decomposition: a problem specification is decomposed into appropriate subproblems until they can be specified. Then, these subspecifications can be combined to obtain the final one. This approach is rigid and inadequate for reuse of specifications; decisions that are taken in each design phase cannot be changed later. Having to be precise about issues that could produce inconsistencies with later requirements is inconvenient.

To avoid these problems we have selected GSBL as the specification language (Clerice, 1990). This language allows incremental construction of incomplete specifications that describe only partial aspects of the problem to be solved.

The only kind of specification unit in GSBL is the class, which denotes an incomplete specification. Here is the syntax of a GSBL class:

```
CLASS class-name OVER <overlist> SUBCLASS-OF <subclasslist>
    WITH
        SORTS <sortlist>
        OPS <opslist>
        EQS <varlist> <equationlist>
    DEFINE
        SORTS <sortlist>
        OPS <opslist>
        EQS <varlist> <equationlist>
END_CLASS
```

Classes in GSBL are incomplete specifications with a structure of components. The basic idea is to distinguish between incomplete and complete parts of a specification The OVER clause corresponds to an enrichment construction. A specification is extended by the components declared in <overlist>. The SUBCLASS clause refers to specialization or refinement of the specifications of classes that appear in <subclasslist>. The WITH clause adds new sorts, operations or equations not completely defined which means that there are not enough equations to specify the new operations or there are not enough operations to generate all the values of a sort. The DEFINE clause either adds new sorts, operations or equations that are completely defined, or completes the definition of some sort or operation.

Specifications are built integrating horizontal and vertical refinements. A horizontal refinement expresses extension relations; an incomplete part can be completed by vertical refinements. Horizontal and vertical refinements correspond, in this approach, to loose extension and refinement morphism. Combinating these two types of refinements produces a similar effect, but is more powerful than the explicit genericity presented in other algebraic specification languages (Wirsing, 1995). A GSBL specification is implicitly parameterized by its incomplete parts. A detailed description of the theoretical principles of the formalism and the semantics of the language appears in Clerice (1989).

We consider it beneficial to extend GSBL. The new language is GSBL+. From a practical point of view, it has been necessary to define mechanisms for error treatment, explicit parameterization and restriction of specifications.

4 THE SRI MODEL

Software reusability takes many different requirements into account, some of which are abstract and conceptual, while others are concrete and bound to implementation properties. They must be specified in an appropriate way. For example, at more abstract levels, we need descriptions satisfying three conditions:

- 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.

At more concrete levels, we need to include the specification, together with the implementation, in the software itself.

On one hand, the theory of abstract types reconciles the need for precision and completeness in abstract specifications with the desire to avoid overspecification. On the other hand, assertions in an object-oriented language are tightly connected to the notion of state and state transformations caused by execution of methods.

There is empirical evidence that the most effective forms of identification are generally found at more abstract levels of software design. Component identification must satisfy the following criteria:

- The identification process should be efficient.
- The set of candidates to be reused should not be too large.
- The set should include the nearest matches.

Adaptation of reusable components, which consumes a large portion of software cost, is penalized by the difficulty of implementing changes in software products, and by overdependency of components on the physical structure of data.

Considering the issues described above, we introduce the SRI model for the definition of the structure of a reusable component. It decribes object classes at three conceptual levels: identification, realization and implementation.

The identification level allows us to describe type hierarchies that share common properties as a family. The object universe is classified into subdomains, according to methodological principles that yield the best libraries in object-oriented languages, but reflecting only aspects related to behavior, in an implementation independent manner.

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 (Liskov, 1994).

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

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

If E and E1 are specifications, then E can be realized by E1 (written E~~~>E1) if E and E1 have the same signature and every model of E1 is a model of E (Hennicker and Wirsing,1992). Every specification at the realization 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 " ≈≈≈>".

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 behavior.
- Internal nodes of the realization level components, including the root, correspond to an abstract class that defers implementation in the object-oriented level.
- Leaves in the realization level correspond to complete Eiffel classes.

The implementation level can contain classes that are not related to the specifications in the identification and realization levels. They reflect implementation aspects.

A reusable component CONTAINER appears in Figure 1. All specifications that describe "containers" are descendants of an incomplete specification (Container-I). Every leaf in the identification level is a complete specification, corresponding to a subcomponent at the realization level. For example, Set-C is a leaf that links different realizations: Set-Tree and Set-List. They correspond to subcomponents at the implementation level that represent concrete classes.

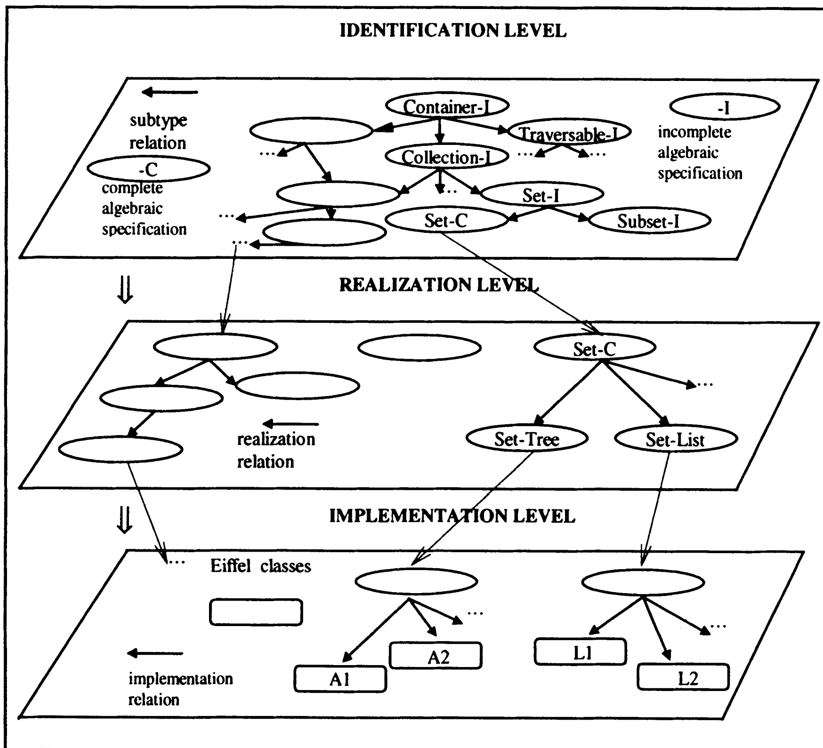

Figure 1. The SRI model: a reusable component CONTAINER

5 A RIGOROUS METHOD FOR REUSE

The method has the following steps:

Decomposition

Formalize the decomposition of a specification E into subspecifications E_1, E_2, ..., E_n . The decomposition is expressed as a transformation pattern based on GSBL+ subspecifications. It relates classes through the OVER and SUBCLASS clauses.

Identification

For each subspecification E_i identify a component C_i (in the identification level) and a sequence $s_1,s_2,....,s_n$ 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 realization 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.

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 renaming. 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 $OP_1(........(OP_K(LEAF_j)....)$, verifying that $OP_1(.....(OP_K(Root(CR_j)......) \sim\sim>$ $OP_1(....OP_K(LEAF_j))...)$ is a realization relation.

Select a class scheme ESQ_m in a subcomponent of the implementation level with root $LEAF_j$. Apply the operator sequence OP_1, OP_2 , ...OP_K to ESQ_m, i.e. construct the specification $OP_1(........(OP_k(ESQ_m)....)$, verifying $OP_1(.....$ $(OP_K(LEAF_j)\ ......) \approx\approx> OP_1(....OP_K(ESQ_m))...)$ is an implementation relation.

Recomposition
Compose the subspecifications E_i and their implementations according to the transformation pattern.

Optimization
Restructure the object-oriented versions to generate efficient concrete classes. Given a collection of classes, their inheritance hierarchy, and a program that uses the hierarchy, a set of rules allows us to restructure the inheritance hierarchy into an efficient one, while preserving the semantics of the program.

6 TRANSFORMING SRI SPECIFICATIONS

6.1 Operators on GSBL+ specifications

Renaming
This transformation is based on the notion of signature morphism. Let $\Sigma = (S,F)$ and $\Sigma' = (S',F')$ be two signatures with sets of sorts S, S' respectively, and sets of operation symbols F, F' respectively. A signature morphism can be defined as a function $\rho: \Sigma \to \Sigma'$, where $\rho=(\rho_{sorts}, \rho_{oper})$ are the mappings ρ_{sorts}: S-> S' and ρ_{oper}: F -> F'. ρ_{sorts} and ρ_{oper} are compatible with the functionality of the operations, i.e. $\forall f \in F$: $f: s_1, \dots, s_n \to s$, ρ_{oper} has functionality $f: \rho_{sorts}(s_1), \dots, \rho_{sorts}(s_n) \to \rho_{sorts}(s)$.

Then, we can say that given a specification SP, a signature Σ and a signature morphism ρ: sig(SP) -> Σ that represents a rename, *renaming(SP, ρ)* is a specification with signature Σ. This means that all of its sorts and operations have been renamed (Hennicker and Wirsing, 1992).

Restriction
Restriction is a transformation operator that creates a new specification formed by a subset of the sorts and operations of other ones.

Let SP be a specification with signature Σ=sig(SP), so *restriction(SP, Σ')* is a specification with signature Σ', where $\Sigma' \subseteq \Sigma$. After application of this operator, the resulting specification does not include the sorts and operations of Σ which do not belong to Σ' (Hennicker and Wirsing, 1992).

Extension
Let SP be a specification with a set of sorts S and a set of operations F, the extension creates a new specification that enriches SP. Then, *extension(SP,S',F',Eqs),* where S', F' and Eqs are sets of sorts, operations and equations, results in a specification SP' where the set of sorts is S'$\cup$S and the set of operations is F$\cup$F' (Hennicker and Wirsing, 1992).

Composition
The composition operator combines two specifications, SP1 and SP2. *Composition(S1,S2)* denotes a specification whose signature is the union of the signatures SP1 and SP2; its axioms are the union of the axioms of SP1 and SP2.

6.2 Transforming realization subcomponents

Building operators on specifications can be extended to manipulate subcomponents in the realization level. Informally, this implies simultaneous application of an operator to every node of the subcomponent. The subcomponents are inductively defined by the operator: *realize(S,{RS_1,RS_2,....})* where *S* is a specification and RS_1, RS_2,...are reusable components (such roots are realizations of *S*). In particular, every leaf L is defined by $_{rl}$L.

Operators are inductively defined by
rename_c(realize(ESP,{ESP1,ESP2,..ESPn}),r)=
realize(renaming(ESP,r),{rename_c(ESP1,r), rename_c(ESP2, r), ...,
rename_c(ESPn, r)})

restrict_c(realize(ESP,{ESP1,ESP2,..ESPn}),S)=
realize(restriction(ESP,S),{restrict_c(ESP1,S),
restrict_c(ESP2,S),...,restrict_c(ESPn,S)})

extend_c(realize(ESP,{ESP1,ESP2,..ESPn}),S,F,E)=
realize(extension(ESP,S,F,E),{extend_c(ESP1,S,F,E) ,...,
extend_c(ESPn, S,F,E)})

For example, *rename_c* is defined as a renaming of its root and, recursively, all its children. The restriction and extension operators are applied in the same way. Applying a renaming to Figure 2 yields the reusable component shown in Figure 3.

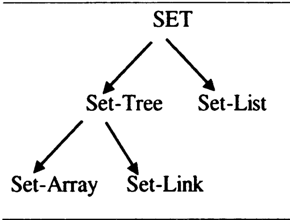

Figure 2. Set subcomponent

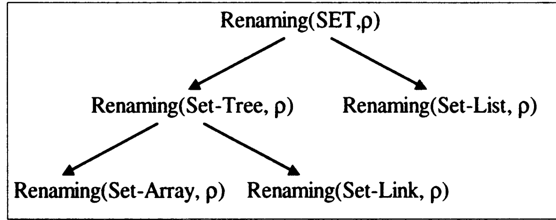

Figure 3. Renaming Set

Composition requires a more detailed analysis (Hennicker and Wirsing, 1992). Composition of two reusable components is defined as composition of its roots and, recursively, all its children. Formally, composition is defined as

$$compose_c(realize(E,\{CRR_1,CRR_2,...,CRR_p\}),\ realize(E',\{CRR'_1,CRR'_2,...,CRR'q\})) = realize(compose(E,E'), \Delta)$$

where Δ =

- if p>0 and q>0 $\{compose_c(CRR_i, CRR'_j) / i \varepsilon \{1,....,p\}, j \varepsilon \{1,....,q\}\}$
- if p>0 and q=0 $\{compose_c(CRR_i, {}_{rl}E') / i \varepsilon \{1,....,p\}\}$
- if p=0 and q>0 $\{compose_c({}_{rl}E, CRR'_j) / j \varepsilon \{1,....,q\}\}$
- else ϕ.

Applying composition to Figure 4 yields the component shown in Figure 5.

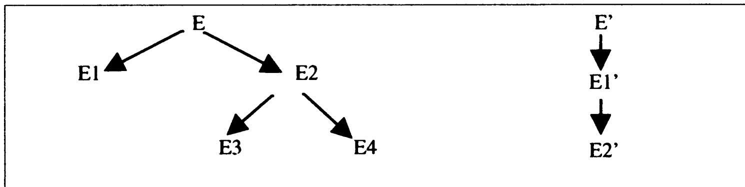

Figure 4 - Realization subcomponents

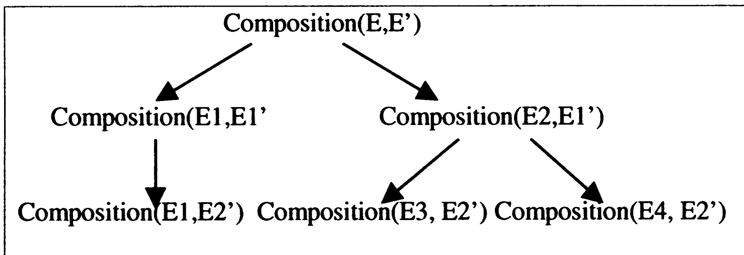

Figure 5 - Composition operator.

6.3 Transforming implementation subcomponents

Building operators for specifications are extended to manipulate subcomponents at the implementation level. Informally, this implies application of an operator to

every scheme of a subcomponent. Subcomponents are inductively defined by the operator: *implement(E,{*ESQ_1*,* ESQ_2 *,....* ESQ_n *})* where *E* is a specification and ESQ_1, ESQ_2 ,... ESQ_n are schemes of concrete classes in an object-oriented language. Operators are defined inductively by

rename_s(implement(E,{ESQ_1*,....* ESQ_n *}),r)=*
implement(renaming(E,r),{rename_s(ESQ_1*,r), ,....rename_s(*ESQ_n*,r)})*

restrict_s(implement(E,{ESQ_1*,....* ESQ_n *}),r)=*
implement(restriction(E,r),{restriction_s(ESQ_1*,r), ..restriction_s(*ESQ_n*,r)})*

compose_s(implement(E,{$ESQ_{1..}$*,*ESQ_n *}), implement(E',{*ESQ'_1*,.* ESQ'_m *})=*
implement(compose(E,E'),{compose_s(ESQ_1*,*ESQ'_1*),...*
compose_s(ESQ_1*,*ESQ'_m*)....compose_s(*ESQ_n*,*ESQ'_m*)}).*

The renaming, restriction and composition operators preserve implementation relations and guarantee *partial correctness*, i.e. if correct code is grouped, the generated code is correct. Since the programmer plays a part in the implementation of the methods, the extension operator does not guarantee correctness.

Our method also transforms the object-oriented versions into efficient concrete classes. We propose a transformational method for restructuring inheritance hierarchies based on the "rules+strategies" approach. In such a framework, an efficient collection of hierarchies is produced from previously existing ones by applying semantics-preserving transformation rules.

Transitions between versions are made according to formal rules. The basic idea is to obtain a final version with the same semantic value as the initial version by constructing a sequence $<V_0,V_1,......,V_n>$ of versions such that $(\forall i:1\leq i\leq n:\ SEM[V_i] = SEM[V_{i-1}])$ for some given semantic function SEM. A given cost function COST, which measures the space or time complexity of the execution of a program, should satisfy $COST[V_0] \geq COST[V_j]$ for some $j>0$.

During the transformation process, we need strategies that guide the application of transformation rules and which allow us to derive hierarchies with improved performance. A detailed analysis may be found in Favre (1997).

7 EXPERIMENTAL RESULTS

To demonstrate the feasibility of our approach, a prototype (TAROOL) was implemented (Favre, 1997) as an undergraduate thesis. It could be refined to be a practical tool for class reuse. A detailed description appears in Carbajo (1997). The prototype offers facilities for handling reusable components and verifying the user's expectations. The prototype assists in:

- Specification editing.
- Analysis of specifications written in GSBL.
- Specification validation.

- Component reuse: The prototype assists in the creation of new applications by applying the transformation operators to existing classes.
- Transformation of GSBL+ specifications to Eiffel code.
- Design maintenance.

Results of experiments with the prototype 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 analyze. Real design maintenance requires automation, which depends on formalization.

The proposed method forces systematic reuse of behavior from structured algebraic specifications. The application of building operators to SRI subcomponents and recording of the "design history" permits good maintenance.

In the object-oriented paradigm, changes are made incrementally to classes, and specialization is achieved by means of the subclass mechanism. Access to objects that are instances of specialized classes may be inefficient due to dynamic binding. Inheritance can also waste memory space. The proposed method uses dynamic binding and polymorphism only where needed. Code is generated in its purest form, omitting such mechanisms as method redefinition, direct repeated inheritance, etc.

An advantage of our formal approach is that identification and retrieval of components can be partially automated. For reuse to be effective, it must be less expensive to identify a component than to construct it. Directions for future work include finding more practical ways to match both signatures and specifications, and applying our ideas to matching large components.

For construction of this prototype we have specified a subset of the libraries provided by the language ISE Eiffel version 3. Improving the existing classes of the implementation language would require a reengineering process.

8 CONCLUDING REMARKS

This work presents a rigorous method for the systematic reuse of object-oriented software. A prototype system, TAROOL, assists the reuser in the administration of reusable components and in the semiautomatic conversion of specifications to code. Our main contributions are: definition of the SRI model, a method based on this model, and an application of our principles in integrating algebraic specifications with Eiffel. The SRI model allows identification of different types of relations between complete and incomplete specifications and classes, and permits the specification of incomplete behaviour and its association with deferred classes at the implementation level.

The method separates the design strategy from the generated code, facilitates the automatic identification and retrieval of components, and extends building operators to implementation level classes. The application is a particular case of

our model and our method, which is based on the characteristics of the Eiffel language for class specification.

9 REFERENCES

Biggerstaff, T.and Perlis, A., Eds (1989). *Software Reusability Volume 1: Concepts and Models.* ACM Press.
Carbajo, M., Diez, G. and Palomeque, C. (1997) Undergraduate Thesis, Departamento de Computación, Universidad Nacional del Centro de la Pcia. de Buenos Aires, Argentina.
Clerici, S. and Orejas, F. (1990). The Specification Language GSBL. *Recent Trends in Data Type Specification.*
Clerici, S. (1989). Ph.D. Thesis, LSI Department. Universidad Politécnica de Catalunya, España.
Favre, L., Diez, G., Carbajo, M. and Palomeque, C. (1997). Object-oriented Software Reusability: A Rigorous Method, *in Proceedings SEKE'97,* Knowledge System Institute, USA.
Favre, L. (1997). Formal Methods and Object-oriented Reusability, *Research Report 35,* Isistan, UNCPBA, Argentina.
Hennicker, R. and Wirsing, M. (1992). A Formal Method for the Systematic Reuse of Specification Components in *Lecture Notes in Computer Science* **544**, Springer-Verlag.
Meyer, B. (1997). *Object-oriented Construction.* Prentice Hall.
Meyer, B. (1992). *Eiffel The Language.* Prentice Hall.
Wirsing, M. (1995). Algebraic Specification Languages: An Overview *in Lecture Notes in Computer Science 906* (eds. E. Astesiano, G. Reggio and A. Tarlecki) Springer-Verlag, Germany.

10 BIOGRAPHIES

Liliana Favre is an Associate Professor of Computer Science at the "Universidad Nacional del Centro de la Pcia. de Buenos Aires" in Argentina. She is leader of the "Software Technology Group" of the "Tandil System Research Institute". Her research interests are in the area of specification and formal methods, especially algebraic methods for software specification and design.

Gabriela Diez is a Research Assistant of Computer Science at the "Universidad Nacional del Centro de la Pcia. de Buenos Aires" in Argentina. She has worked in the "Software Technology Group" of the "Tandil System Research Institute" during 1997.