From OMT to Beta: An Integrated Environment

Arianna del Soldato* Serenella Rampichini†
Doris Schmedding
Dept. of Computer Science, University of Dortmund
Informatik 10 (Software Technology), D-44221 Dortmund, Germany
email: schmeddi@ls10.informatik.uni-dortmund.de

June 13, 1996

Abstract

This paper shows how to integrate the phases design and implementation of the software life cycle. For this purpose we have selected the object oriented analysis and design methodology OMT and the object oriented programming language Beta. We face the problem to minimize the differences between the design and implementation languages: this integration is realized by providing a mapping between the fundamental OMT concepts and their appropriate counterparts in Beta.

We discuss these issues not only theoretically but we also present an integrated environment for developing object oriented software, composed of a graphical editor for the OMT methodology and a textual editor for a subset of Beta respectively. Both editors are syntax driven and integrated by means of a Communication Protocol Subsystem that ensures the consistency between both representations of the model.

1 Introduction

Object orientation is a way of thinking that exploits models organized around real world concepts instead of computer concepts. In the *analysis* phase the relevant concepts and phenomena of the real world are identified and described. During the *design* phase these concepts are transformed in a formal description that is then refined into an executable software system in the *implementation* phase.

Although object orientation aims at integrating these three phases, some problems related to the differences between the languages used in the phases, still exist. For this reason many Object Oriented Analysis and Design (OOAD) methodologies have been developed in order to fill the semantic gap between analysis and design, while the bridge between design and implementation is still an open problem. This is due to the fact that usually some mechanisms in design languages are not directly supported by programming languages.

To face this problem we present an example of integration between the OOAD methodology OMT [RBP+91] and the object oriented (later called OO) programming language Beta [LMN93].

In general, when building a model, the expressiveness is limited by the language used for describing the model. To reduce this restriction we have chosen OMT as the design language. OMT is a well established and very expressive OOAD methodology. A comprehensive study of

*Work done during a stay at University of Dortmund
†Work done during a stay at University of Dortmund

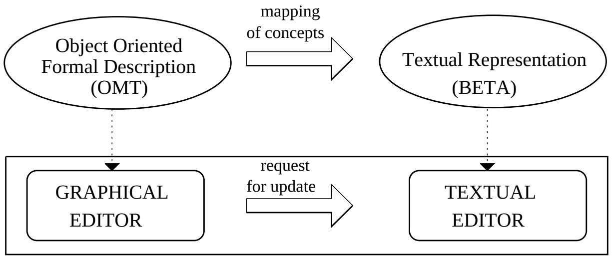


Figure 1: Depiction of our Goal

several OOAD methodologies also shows that OMT is the most suitable to be integrated with Beta [RS95].

The programming language Beta is actually used at the University of Dortmund to teach the computer science freshmen. We study the integration of OMT and Beta to construct an object oriented software development environment for students with an advanced curriculum.

In our work we translate the fundamental OMT concepts into Beta concepts that are semantically equivalent (see fig. 1). We also show the limitations of this approach.

Moreover, the transformation is supported by an integrated software development environment, that contains a syntax driven graphical editor for the OMT methodology and a syntax driven textual editor for Beta. The object oriented design is automatically translated into Beta code skeletons that can be filled by means of the textual editor. The integration of the used design and implementation methods is done by the definition of consistency constraints between the documents which realize the mapping between OMT and Beta. The user is always offered a consistent view on his model in the OMT notation or in the Beta programming language.

2 OOAD with OMT

In the integration of the design and the implementation phase the choice of an appropriate OOAD methodology plays a decisive role. A methodology provides an approach and a notation for supporting analysis and design. In particular, it provides the formal definition of systems using a graphical notation, which allows the designer to define the system in terms of real word concepts abstracting from implementation choices and details. We have analyzed and compared several methodologies. Among them Coad and Yourdon [CY91], Shlaer and Mellor [SM89], Booch [Boo91] and Rumbaugh [RBP+91] turned out to be the most expressive. Basically, these methodologies adopt similar base OO concepts in their earlier analysis phase and most of them support a graphical notation for the development in the first two phases. However, they vary to large extent in their support of advanced object modeling concepts. Moreover, each methodology emphasizes on different target users and practitioners.

We have selected Rumbaugh's OMT methodology, since OMT is the most suitable OOAD methodology for constructing a formal description of complex systems given our purpose, viz., transforming the design into executable Beta programs; this method is also one of the most complete and unambiguous.

Our experience in designing our tool by means of OMT has reinforced the need of an additional abstraction level. Rumbaugh proposes to break a complex system into subsystems [RBP+91], but he defines no graphical notation for them and describes this concept only informally.

The OMT notation uses three models to describe a system [RBP+91]:

- The **Object Model** captures the static structure of the objects and their relationships in a graph called *Object Diagram*.

- The **Dynamic Model** describes the interactions among objects, i.e. a state diagram models the aspects of the system that change over time.

- The **Functional Model** describes the data transformations of the system by data flow diagrams.

The Object Model is the most important for our purpose, since it provides all the information for our transformation process. In fact, the Object Diagram is a graph whose nodes are objects and classes and whose arcs are relationships among them. A class describes a group of objects with similar properties (attributes) and behavior (operations). Classes are arranged into hierarchies sharing common structure and behavior, organized in aggregations, associations and inheritance.

3 Object Oriented Programming with Beta

Beta is a modern language in the Simula tradition. Although Beta is primarily intended for the OO programming style, it contains comprehensive facilities for procedural and functional programming: it is a multi perspective language. The main characteristic of Beta is a single abstraction mechanism, called *pattern*, to realize classes, objects, operation, attributes and types. A pattern has the following form:

Pattern_name:	(# Decl1; Decl2; ... Decln	$\{declaration - part\}$
	enter In	$\{enter - part\}$
	do Imp	$\{do - part\}$
	exit Out	$\{exit - part\}$
	#)	

where:

- *Decl1; ...; Decln* are attribute and local pattern declarations.

- The *enter-part* contains a list of input parameters.

- The *do-part* is built by a list of imperative statements that describe the actions to be performed when the object is executed.

- The *exit-part* contains a list of output expressions.

Each of these elements is optional. The structure of a pattern allows the declaration of a class with *do-*, *enter-* and *exit-part*. This is in contrast to the common view of a class as a model for objects encapsulating only attributes and methods and modifying their states only by means of the methods defined on them. The programming language Beta provides uniform access to both attributes and operations (as Eiffel) but unlike other languages it does not provide encapsulation. In Beta an object can directly change the attributes of another object without invoking any method.

Also the redefinition of attributes is impossible. Moreover, Beta does not explicitly support associations which should be available in a class library, like container classes [RBP+91, p. 319]. Also important, unlike many other OO languages, Beta does not support multiple inheritance, because it provides inheritance by the use of super/sub-patterns. Every subpattern inherits exactly from one superpattern. The lack of support of multiple inheritance gives rise to tree structured and easy comprehensible classification hierarchies. Beta's inner-concept to realize redefinition of operations has a clear semantics and offers different possibilities to specialize operations. In Dortmund Beta was chosen to teach the students because it offers object oriented concepts as well as parallel programming concepts.

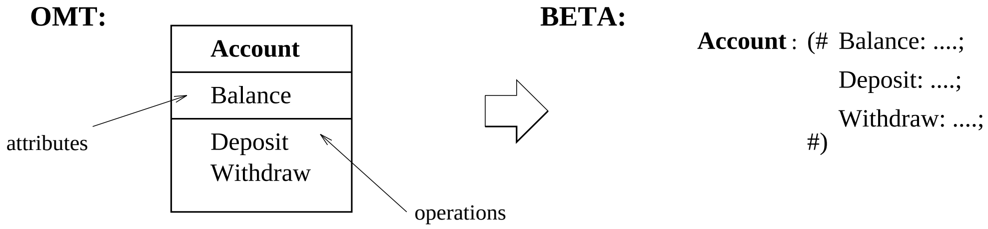


Figure 2: Translation of the Class Account

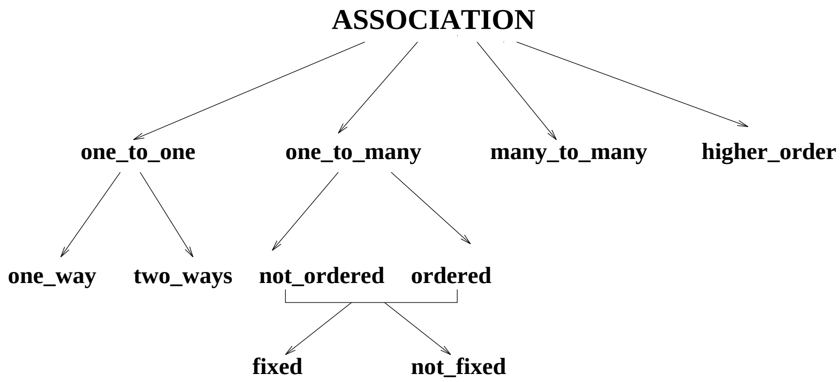


Figure 3: Classification of Associations

4 Mapping from OMT to Beta

The goal of the integration between OMT and Beta is:
- representing as much as possible the OMT concepts,
- generating Beta program skeletons structured in an OO programming style.

The integration has been realized defining for each OMT concept corresponding transformation rules. In particular, the definition of classes, instantiations, calling operations and the use of single inheritance have their counterparts in the corresponding Beta concepts.

To illustrate our approach we present the translation of the real world phenomenon of a bank account, which contains the balance and the operations of deposit and withdraw (see fig. 2). The OMT notation for a class is a box divided into three parts: the name of the class (Account), a list of attributes (the single attribute Balance) and a list of operations (Deposit and Withdraw). In Beta the class Account is represented by a pattern, also called Account, with the attributes Balance, Deposit and Withdraw. (Analogous to the use of the single abstraction mechanism *pattern*, Beta uses the same name for attributes in the OO meaning and for operations.)

In agreement with the common view of a class as a container for attributes and methods, we translate classes with patterns without *enter-, do-* and *exit-part*. This avoids a class to be executed and constrains state changes only by means of operations.

An **association** describes a group of links between classes with common structure and common semantics. To define the transformation rules we have classified them at the first level on the bases of the number of objects involved in each instance of the relationship (see fig. 3).

Since Beta lacks the notation of associations, we have implemented them with pointers from an object to another in the case of One_to_One and One_to_Many associations. We use distinct association objects to realize Many_to_Many and Higher_order associations. To translate **One_to_One associations** correctly we have to know the direction of the pointers. Therefore we distinguish

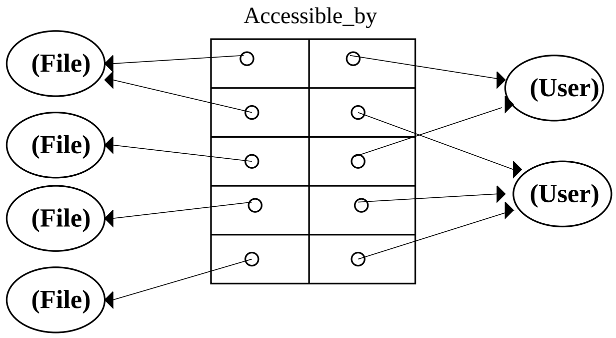


Figure 4: View of Many_to_Many Association

One_way and **Two_way associations** in contrast to the original OMT.

The use of pointers caused us pay attention to the problem of consistency. The pointers are in fact defined independently in the classes involved in the association: deleting or updating only one of them may lead to an inconsistent link. For example, if a One_to_One Two_way association should be deleted, both pointers which build together the link have to deleted. We solved this problem at the level of the tool, allowing the updating of the pointers only by manipulating the entire association.

In a **One_to_Many association** each instance of a class can be in relation with many instances of the other class. This relationship is further characterized by the multiplicity and the order of the related objects. Depending on the characteristics of the One_to_Many association we use one of the following data structures for its translation: a set of pointers, a list of pointers or a repetition.

In a **Many_to_Many association**, in each instance of the relationship, many instances of a class can be in relation with many instances of the other. For example, it may be useful to represent a relation between files and users: one file is accessible by many users and a user can access a certain number of files. The realization of this association by means of a distinct association object can be seen in figure 4. Here the association is a set of pairs of associated objects stored in a single variable size object. We have implemented the structure above with a binary tree in which each node represents a link between two objects. This structure allows simple and efficient realization of those operations that the programmer can use to handle the association hiding implementation choices.

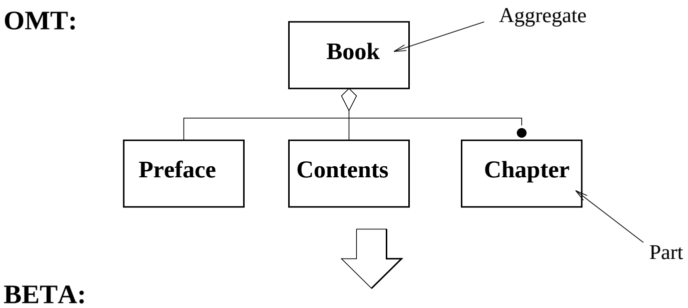
An example of **Ternary association**, the simplest form of higher_order association, is given by programmers who use computer languages on projects. This association is an atomic unit and cannot be divided into two binary associations without loosing information. We implement this association using a binary tree, where each node has three pointers, instead of two.

Aggregation is a strong form of association in which an aggregate object is made of components objects. An aggregation is an association combined with the part_of concept.

In figure 5 we give an example of aggregation. A book is composed of a preface, a table of contents and a certain number of chapters.

In translating aggregation we follow the realization of the association declaring instances of objects instead of pointers to objects. A local object in a class is called *part-object* in Beta [LMN93].

If the number of the components is fixed, as for the preface and the table of contents, the corresponding instances are declared in the aggregate. If the number is variable, the aggregate should contain of set of instances. Unfortunately, some limitations of Beta do not allow to handle instances using statements similar to those defined for references. Because of this technical reasons is was impossible to implement sets of instances to realize a variable number of parts in an aggregate. An alternative may be using a set of references to objects but in this way aggregation matches with association loosing the additional meaning of the part-of concept and some properties like



BETA:

```
Book: (# BookPreface:@Preface;
         BookContents:@Contents;
         Chapters:@Set(# Element:< Chapter #)
      #);
```

Figure 5: Aggregation

transitivity which is one the most significant properties of aggregation [RBP+91].

Multiple inheritance has come up as a generalization of single inheritance, where a subclass is allowed to have more than one superclass. Modeling the real world we often find more than one classification hierarchy for the same set of objects, or we need to apply the so-called *multiple override criterion* [Car91] [Gui91]. Moreover, multiple inheritance is often mis-used by combining unrelated classes for the purpose of reuse of code. This leads to complicate inheritance relationships.

We have analyzed the different uses of multiple inheritance and we found that some of them may be simulated by means of part objects and virtual patterns in Beta. By specifying singular part objects that are specializations of some general classes, it is possible to add attributes and to redefine virtual procedures. So, for each part object a specialization may be obtained realizing the multiple override criterion. We simulated the reuse of code for attributes but we were not able to realize renaming of inherited operations [RS95], because Beta does not allow the renaming of attributes. Moreover, the provision of multiple subtyping is not covered by part objects. This is a relevant drawback for a programming language like Beta, that mainly uses inheritance for hierarchical classification of concepts. For this last reason and because a unique implementation that covers all the aspects of multiple inheritance does not exist, we conclude that in Beta it is not possible to simulate this concepts in an intuitive and straightforward way.

5 Design of the Environment

We saw how design and implementation can be integrated. Transformation rules enable us to translate the fundamental OMT concepts into the semantically equivalent Beta concepts.

Now we show how this integration is supported by an integrated software development environment. This environment is realized by means of a CASE tool which allows to draw an OMT Object Model, to automatically generate Beta skeletons and to complete them to achieve an executable software system.

We design the tool complying with the OMT methodology, which has been extended to include the concept of *subsystem* to increase the comprehension of the diagram by adding one abstraction level. Each subsystem may be refined by another OMT diagram. The graphical symbol chosen for a subsystem is a shadowed box (You find some examples in fig. 6). A special notation for subsystems is missing in the original OMT notation, although the concept is introduced [RBP+91, p. 199]. In a diagram refining a subsystem, those classes and subsystems which lay outside and are directly connected to classes inside have to be shown again to demonstrate the connections between the components.

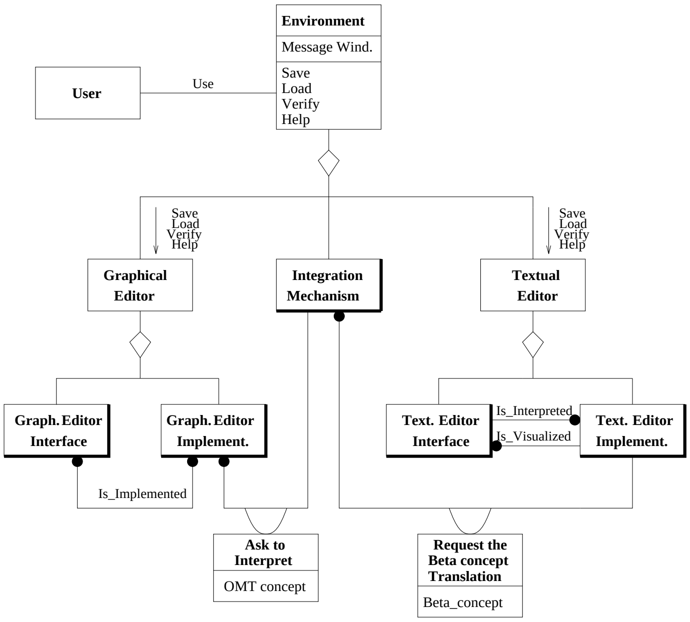


Figure 6: First Level of the Tool Design

The first level of the architecture of our tool is shown in figure 6. Its components are:
- a **graphical editor** to draw an OMT Object Model,
- a **textual editor** to visualize the Beta skeletons and to complete them,
- an **integration mechanism** between the two editors which implements the mapping previously defined.

In order to provide the user with the best support in producing syntactically and semantically correct documents, we realized our tool syntax driven in both the graphical and the textual editor.

We did not require the reverse engineering, which is in [LMN93,p.315] defined as the problem of going back from the code to the design diagram. We ensure the consistency between the graphical and the textual document in another way. Using our tool the architecture of the system can only be changed by means of the graphical editor. The part of the code that is automatically generated cannot be modified by means of the textual editor, relationships to other classes cannot be added textually. The integration mechanism forwards messages from the graphical editor to the textual editor to propagate a change of the structure. So, the user is allowed to develop the graphical model and the textual one concurrently and to implement more than one Object Model at the same time.

A documentation can be added during the design of the OMT Models and is automatically propagated to the textual document. The interface of the tool obeys to the OSF/Motif Style Guide [OSF92].

6 Realization of the Environment

Graphic visualization is accepted as indispensable for many domains, but the use of this technique is not wide spread in today's computer application. This is due to the great effort needed to implement a satisfying graph layout and to integrate the tool in a software development environment. We have compared some approaches to build textual and graphical editors and have decided to follow the approach that prefers editor generators instead of common toolkits [RS95].

In this way syntax driven editors can be easily generated in an automatic way from their spe-

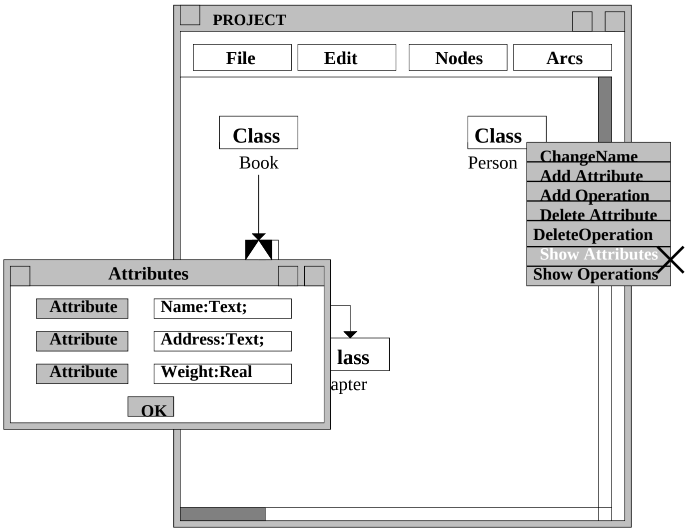


Figure 7: Two Abstraction Levels

cification and they can also be easily modified. The graphical and the textual editor generators chosen are respectively Graph Project (GP) [CI94], which was developed by Engineering in Pomezia (Italy) since 1994, and GENESIS [GOO94], which has been developed within the ESPRIT-III Project (GOODSTEP) at the University of Dortmund. Since both generator specification languages follow the OO paradigm, we were able to implement the tool complying the OMT design shown in figure 6.

6.1 Graphical Editor

The graphical editor is the fundamental part of our tool. In fact, in developing a system, the user mainly interacts with this editor which has the task to handle the syntactic and semantic checks and the user interactions. The specification of the editor is given through the description of terms composing the graphical language and their composition rules. From this specification and from the implementation of the methods associated with each term, GP generates a graphical editor. For each term of the graphical language a corresponding C^{++} object is generated.

By specifying the graphical language we aimed to represent the original OMT notation as far as we were allowed by the capabilities of GP. But we decided to add of a second abstraction level to the OMT notation which has greatly contributed to improve the structure of complex OMT diagrams (see fig. 7). For extended systems it is not useful to show attributes and operations at the same level of classes, especially if the number of these features is high. In this way, we offer a better view of the architecture and we allow to design on a more abstract level like e.g. the OOAD methodology BON does [WN95].

6.2 Textual Editor

As we said before, we have realized our textual editor by means of GENESIS. According to the OO paradigm the specification of the editor is structured into component specifications, where each component is a class defining particular properties of an increment of a given grammar.

Our textual editor development process has followed different phases which are automatically guided and supported by the GENESIS environment. The most interesting phase was the derivation of a new version for the Beta grammar, which has to be modified and transformed in a normalized EBNF to generate the textual editor specification [RS95].

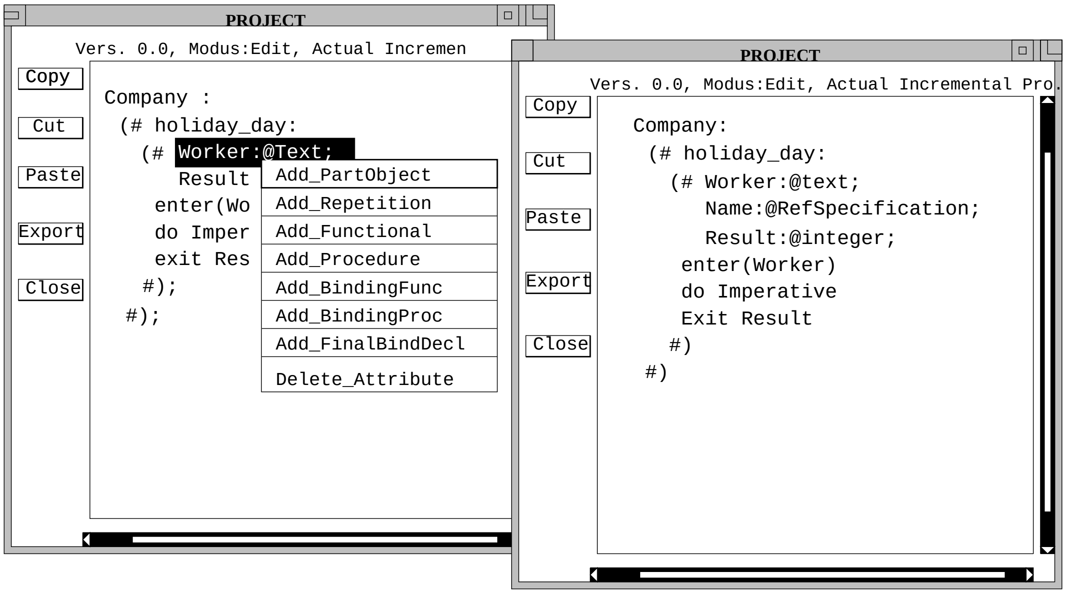


Figure 8: Replacement of a Template in a Syntax Driven Mode

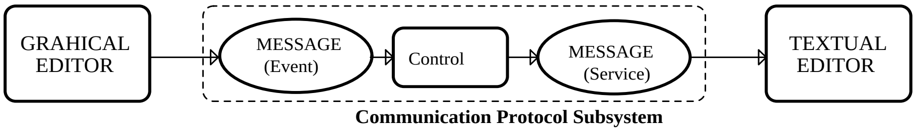


Figure 9: Communication Protocol

We require our textual editor to be syntax driven. Figure 8 shows how a new local variable is inserted into the pattern *holiday_day*, which is a local pattern in the pattern *Company*. The implementation of the corresponding method (Add_PartObject) causes the tool to insert a new part object template behind the currently selected increment. In the next step the generated place holders *Name* for the name of the local variable and *RefSpecification* for a type pattern must be replaced by suitable identifiers.

As this example suggests, syntactical correctness is always preserved because the tool only inserts syntactically correct templates. In this way only those commands are offered whose execution cannot violate the syntax definition.

The textual editor lets the user to decide whether or not to use a structure oriented mode of editing. For instance, experienced users who master the language sometimes prefer to freely edit increments with a conventional textual editor. Then, to ensure the static correctness, the tool parses the new inserted text.

Apart from ensuring syntactical correctness, our textual editor also supports the user in achieving correctness of the static semantics. Since Beta allows to use features which are later defined, the editor temporarily accepts this inconsistent state, drawing the user's attention on it.

6.3 Integration Mechanism

The integration between the graphical and the textual editor is realized by means of the **Communication Protocol Subsystem** which provides our tool with a communication protocol for sending messages from the graphical editor and receiving service requests by the textual editor as shown in fig. 9 to propagate changes of the graphical representation to the textual one.

We use this integration mechanism, instead of operating system primitives, to hide the basic communication mechanism from the rest of the tool architecture. This allows to arrange for portability and provides a dedicated, safe and application-specific protocol for the communication between the two editors.

By means of this message mechanism we have implemented the real mapping from OMT to Beta. For each message defined, representing a service request, we have also defined the correspondent service which executes the request by invoking some methods of the textual editor. These messages handle the variations of the textual representation on the basis of the received requests.

For example for the OMT concept class, messages and services to create, to delete, to add an attribute and a method and to delete an attribute and a method are defined. If an attribute is added to a class by means of the graphical editor, the textual editor consistently adds the attribute to the Beta representation.

7 Conclusion

The integration of the tools for the design and implementation phases in the software development process has been discussed. An approach to map the OMT object model to corresponding Beta concepts has been presented. The development of the mapping has been the most interesting part of our work. It led us to identify those characteristics that make Beta a multi paradigm language. The generated Beta code skeletons are structured in an OO style.

The integration of the design (OMT) and implementation (Beta) languages has been realized by a tool composed of a graphical and a textual editor. This tool allows to draw an OMT Object Model with a graphical editor, and to complete Beta skeletons with the textual editor. Different classes of the system under development can be in different states at the same time. A change of the structure of the system done by the graphical editor is propagated directly to the textual representation. Both the graphical and the textual editor are syntax driven and, in addition, the textual editor allows the user to program in a free textual input mode.

The tool is structured in an object oriented way. This allows to exchange the design or the implementation language, while most of the tool may be reused. Providing the graphical editor with a proper interface, our tool can be extended in the future to support the development of the Functional and the Dynamic Model of OMT. To cover the entire OMT methodology, graphical editors for the other two notations should be specified and generated. The presented integration mechanism can be used to get consistency between the Object Model and the other two models. The definition of appropriate consistency constraints according to the proposed mapping from the Object Model to Beta is an open problem, which still has to be solved.

We studied the approach where the tools are generated and integrated by means of a message passing mechanism to built a software development environment with the programming language Beta to teach our students. We found that this approach was quite easy to realize, i.e. the generation of the editors as well as the their integration. The obtained degree of integration of the tools makes the development of software quite comfortable.

Unfortunately the graphical editor was not robust enough to be used by hundreds of undergraduate students. Instead, we realized a very simple and not well integrated environment for them with a graphical editor for the design and a conventional text editor. At the end of the design phase, Beta code skeletons are generated by a simple compiler which is based on the presented mapping.

Acknowledgments: We would like to thank Prof. E. E. Doberkat, Prof. V. Ambriola and Wolfgang Emmerich for all the provided useful suggestions.

References

[Boo91] Grady Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Co., Inc., Redwood City, CA 94065, 1991.

[Car91] M. Carrol. *Using Multiple Inheritance to Implement Abstract Data Types.* The C^{++} report, 3(4), Apr. 1991.

[CI94] P. Corte and A. Inferrera. *Graph Project, Preliminaries Notes.* Engineering - Ingegneria Informatica S.p.a. GP - Version 1.0 Feb. 1994.

[CY91] P. Coad and E. Yourdon. *Object Oriented Design.* Prentice Hall, Englewood Cliffs, N.J., 1991.

[GOO94] GOODSTEP Team. *The GOODSTEP Project: General Object Oriented Database for Software Engineering Processes.* In K. Ohmaki, editor, *Proc. of the Pacific Software Engineering Conference,* Tokyo, Japan, IEEE Computer Society Press, pp. 410-419, 1994.

[Gui91] N. Guimaraes. *Building generic User Interface Tools: an Experience With Multiple Inheritance.* In OOPSLA'91, Object Oriented Programming Systems, Languages and Applications, Vol. 26, No. 11, pp. 89-97, Nov. 1991.

[LMN93] Ole Lehrmann Madsen, Birger Moller-Pedersen, Kristen Nygaard. *Object-Oriented Programming in Beta Programming Language,* Addison-Wesley, 1993.

[OSF92] OSF/Motif Style Guide. Revision 1.2, 1992. (For OSF/Motif Release 1.2). Open Software Foundation 11 Cambridge Center Cambridge, MA 02142.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design,* Prentice Hall, 1991.

[RS95] Serenella Rampichini and Arianna Del Soldato. *From OMT to Beta: An Integrated Environment.* SWT Memo No.83, Chair of Software Technology, University of Dortmund, 1995.

[SM89] S. Shlaer and S.J. Mellor. *An Object-Oriented Approach to Domain Analysis.* ACM SIGSOFT Software Engineering Notes, Vol. 14 No. 5, pp. 66-77, July 1989.

[WN95] Kim Walden and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture, Analysis and Design of Reliable Systems,* Prentice Hall, 1995.