INTRODUCING SEPARATION OF ASPECTS AT DESIGN TIME

José Luis Herrero[1], Fernando Sánchez[1], Fabiola Lucio[1], Miguel Toro[2].

[1]Departamento de Informática
Universidad de Extremadura
{jherrero, fernando, flucio}@unex.es
[2]Departamento de Lenguajes y Sistemas Informáticos.
Universidad de Sevilla.
mtoro@lsi.us.es

Abstract .- *Different proposals have been developed in order to separate object functionality from other non-functional propertiers like concurrency, synchronization, distribution,etc. ATOM, AspectJ, Composition Filters and Disguises Model are four of them. In these models object functionality is separated form non-functional propertiers, and composed later at compile or runtime. Separation is necessary when applications need a high degree of reusability or adaptability. The models mentioned above achieve this goal at implementation level, but what happends at design level? Why we don´t take advantage of separation of aspects at design time? Our work introduces separation of aspects at design time. UML is used as the modeling language and its semantic has been extended in order to represent object functional design separated from other non-functional designs. The main goal of our work is to generate code for different separation models starting from the same design. In this way a generator application has been developed. This application can generate code for AspectJ, Composition Filters and Disguises Model starting from a UML design. The introduction of separation of aspects at design level provides two basic advantages: the generated code is not affected by inheritance anomaly, and a higher degree of reusability is achieved, due to designs being reused in different domains.*

Keys : Aspects, Separation of concerns, Synchronization, System design, UML.

1.- INTRODUCTION

Object-oriented systems represent the world as a collection of related objects [1]. In this model, the object is an indivisible unit that executes a set of actions. Object code specifies object behaviour as a whole, i. e., every action must be implemented in the same place, even if their objectives are different (functional, synchronization, concurrency, distribution, real-time, etc). This mixture produces well known and well researched problems such as inheritance anomaly [2], low adaptability and reusability degree [3].

In order to solve these problems, it has become essential to separate concerns, in such a way that the object basic behaviour is defined in object implementation, while other non-functional properties, like concurrency, synchronization, distribution, real-time, etc are defined separately in other entities and composed later at compile or run time.

Based on this concept, some models have been developed with greater or lesser degree of separation. Among them, Composition Filters [4], AspectJ [5], and Disguises Model [6]. A new metaentity is defined by all of these models in order to represent non-functional properties (filter, aspect and disguise respectively). Furthermore, some Patterns [7] have been developed in order to describe different aspects [8]. But all of them are focused on developing this separation at implementation level, that is, they separate object functional code from other non functional code.

But, what happens at design level? Do we need separation? Separation is necessary when applications need a high level of reusability or adaptability. Object functionality design separated from other non-functional properties, contributes to increase in design modularity and in degree of reusability. System design is more modular, because non-functional properties are designed separately, instead of being mixed. Moreover, these designs can be reused in different domains with an appropriate composition mechanism.

Recently, new research has appeared which tries to represent this separation in other stages of object life cycle. Junichi [9] extends the Unified Modeling Language (UML) in order to define aspects. As such, the UML metamodel is modified and a new metaentity, called *Aspect,* is added. This new meta-entity represents the aspect concept at design level. An aspect is defined by a set of attributes, operations and relationships. Aspect attributes are used by its weave definitions, operations are considered as its weave declarations, and relationships includes generalization, association, and dependency. Bäumer [10] defines the Role Object Pattern as different views of an object. In other words, as separated role objects which are attached to and removed from the core object. The resulting composite object is called *Subject*, which consists of one core element and its role objects. Object diagrams are also introduced in order to represent object behaviour as a composition of different entities. Both works represent object behaviour by separated aspects even at design time, however in a general way. They do not focus on problems introduced by each new aspect.

This work tries to go one step further and represents aspects at earlier stages of object life cycle. Concretely, synchronization is the aspect we are focused on. UML[11] and AspectJ, Composition Filters and the Disguises Model are used in order to design and implement synchronization policies separately from object functionality. UML is a language for specifying and constructing software systems. It is the result of combining three object methodologies (Booch, OMT, OOSE). Furthermore, OMT has adopted UML as standard modeling language. An application has been developed in order to, starting from UML designs, generate code for different models (Disguises Model, AspectJ, Composition Filters).

The rest of the paper is as follows: Section 2 introduces all elements that define completely object synchronization. Section 3 extends UML semantics in order to represent synchronization separately from object behaviour. Section 4 defines how to generate code starting from a UML design. And Section 5 shows the conclusions and future works.

2.- SYNCHRONIZATION ELEMENTS

Synchronization is the aspect we are focused on. Because there are different definitions of this word, first of all, synchronization must be defined. Synchronization is defined as the set of constraints, imposed on an object, which decide to delay or accept method invocation in order to preserve object integrity The identification of all properties that define synchronization policies is the first task. There is different research, mentioned before (Composition Filters, AspectJ or Disguises Model), that separate synchronization polices from functional code. Analysing them, some common properties are observed. These properties are :

- **Synchronization state** :Information about the state of synchronization.

- **Synchronization methods** :Methods for changing synchronization state.

- **Synchronization predicates** :Represents an invocation. The predicate can contain the following elements :

  - **Guard** : Condition for activating the invocation.
  - **Preaction** : Set of actions that are executed before the invocation.
  - **Postaction** : Set of actions that are executed after the method has been completed.

3.- REPRESENTING SYNCHRONIZATION IN UML

UML has been extended in order to represent, separately, synchronization from object functionality. In this way, each object is designed in two ways. Functional level, where object functionality is defined, and synchronization level, where its synchronization policy is described. Currently, in UML one class is represented as a box with its class name, atributes and methods. Synchronization policies are not represented because they are defined inside method bodies.

In this way, UML must be extended in order to represent synchronization policies separately from object functional behaviour. UML semantic can be extended with new stereotypes, constraints and tagged values. Constraints allow the specification of a new semantic for a model element. Stereotypes provide a way of classifying elements, adding new attributes, associations and operations. And Tagged Values permit arbitrary information to be attached to any model element.

In order to represent syhchronization policies, a new stereotype called <Synchronization>, is created. This stereotype must allow the specification of each synchronization property previously defined (abstract states, preactions and postactions). As such, synchronization state and operations are represented by stereotype attributes and methods respectively. Figure 1 shows this new stereotype.

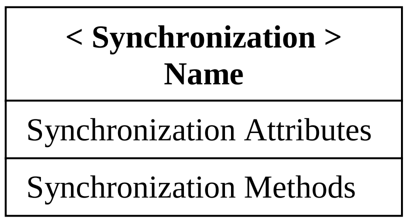


Figure 1. Synchronization stereotype

There is one property left that has not been represented. Synchronization predicates define synchronization dynamic behaviour, i. e., they determine when an invocation can be executed. As such, these predicates cannot be represented in a class diagram. So a different UML diagram must be used in order to represent synchronization predicates. States and transitions between them can be used in a statechart diagram to define dynamic behaviour. Each transition represents an event. The next figure shows a statechart diagram which represents synchronization policies.

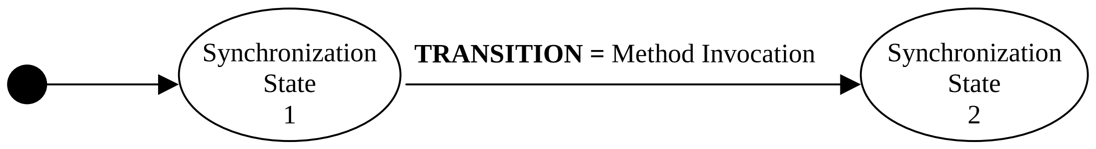


Figure 2. Representation of dynamic synchronization behaviour

Some constraints must be added to a normal statechart diagram in order to represent synchronization policies. These new constraints limit the statechart semantic in order to represent synchronization behaviour only. As such, this new type of statechart diagram is called *synchronization statechart diagram*, and its construction rules are the following:

1. A synchronization state represents the set of methods that can be invoked in one moment.

2. A Statechart transition represents a relationship between two states, and indicates that, an object in one state will only pass to another state when an event happens. A transition between two synchronization states can only happen if a method is invoked. This is a definite restriction of a normal statechart.

3. The transition guard evaluates the possibility of triggering the transition; i.e., it evaluates the possibility of triggering a method.

All models studied have defined the necessity of introducing preactions and postactions in order to represent synchronization policies completely. Preactions are a set of actions that must be executed before the method is invoked. Postactions are a set of actions that must be executed after the method has been completed. In this way, some new information must be attached to a transition in order to represent preactions and postactions. In figure 3, a transition is represented with these new elements.

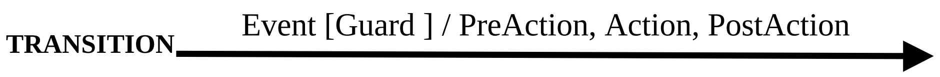
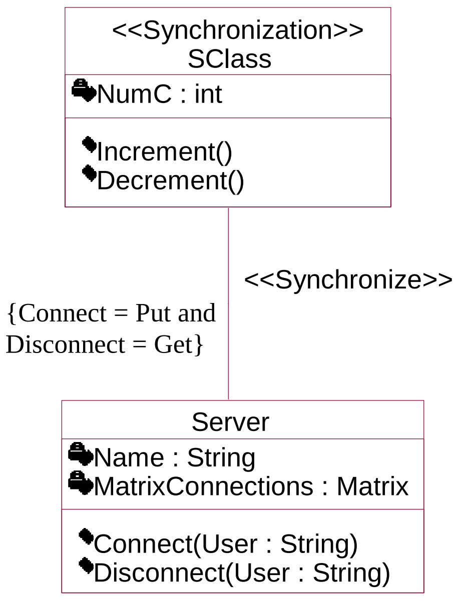


Figure 3. Complete UML transition representation.

In figure 4 the server example is represented. It is designed with a class diagram and one synchronization statechart diagram.



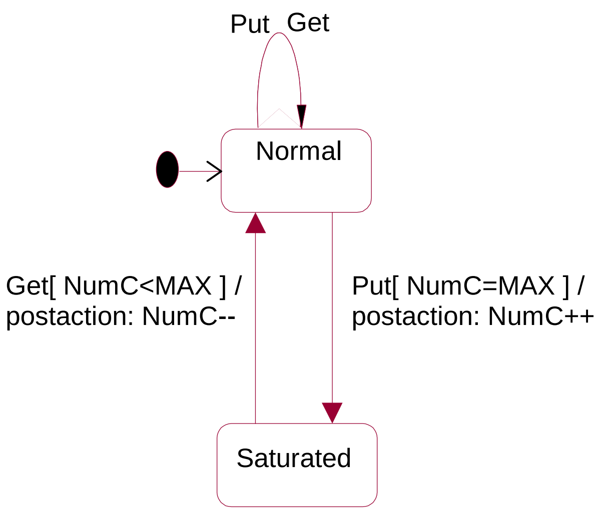

Put Get
Normal
Get[NumC<MAX] / postaction: NumC--
Put[NumC=MAX] / postaction: NumC++
Saturated

Figure 4. Class diagram of Server example

Figure 5. Synchronization statechart diagram

The stereopype <Synchronization> Sclass is a new stereotype which represents the contraints for invoking server methods. As such, it is representing server synchronization. But Sclass is still not completely defined, because method constraints and its dynamic behaviour have not yet been defined. In figure 5, a synchronization statechart diagram represents these two concepts. This statechart diagram is associated with Sclass and defines a synchronization policy. In this case, a user can only connect with the server if it is not saturated, i. e., if there are not many connections.

In order to reuse synchronization designs in different domains, a composition mechanism has been developed. The association between <synchronization> and the base class design is represented with an association between both elements. Another stereotype < Synchronize > is created, and defines the way both elements are related. In this example, there are two related methods. Although connect and disconnect methods are not defined in the synchronization statechart diagram, their behaviour is exactly the same as Put and Get methods. So the composition mechanism defines a mapping relation and expresses the elements of both diagrams which represent the same concept.

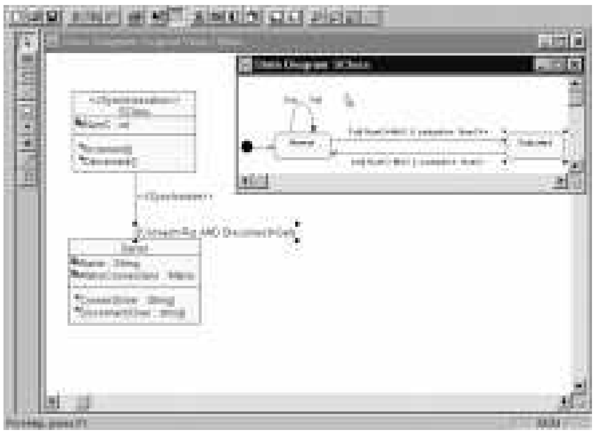
In this way synchronization can be reused in different domains simply by defining a mapping between the synchronization stereotype and the base class. The synchronization statechart diagram of this example has been extracted from a bounded buffer example. This demonstrates the high degree of reusability.

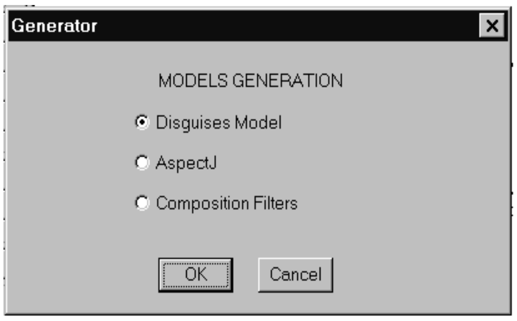
4.- CODE GENERATION STARTING FROM UML

In section 4, it has been shown how UML has been extended in order to represent synchronization aspect separately from functional behaviour. The next step is to generate code starting from this representation.

The Rose Scripting language has been used in order to create a Disguises Model, Composition Filters and COOL (first version of AspectJ) code generators. These generators read UML design, request all the information about the elements, and generate the skeleton of the designed application.

Starting from the same design, the generator application can produce code for three different models : AspectJ, Composition Filters and Disguises Model. The application asks the user about the target model, and atomatically, the skeleton of the application designed is generated. In Figure 6 is represented the server example designed in Rational Rose and figure 7 shows all the models that can be generated.





Generator

MODELS GENERATION

[✓ Disguises Model]

AspectJ

Composition Filters

Figure 6. Server example. Figure 7. Generation Form

5.- CONCLUSIONS AND FUTURE WORKS

An extension of UML has been defined for keeping object synchronization policy and its functional behaviour separated. A new stereotype <Synchronization> has been created. This stereotype defines, on one hand, static behaviour of synchronization policy with a class diagram, and on the other hand, the dynamic behaviour with a synchronization statechart diagram. Thanks to this extension, synchronization policies can be designed independently and separately. It also allows high reusability of synchronization policies. Furthermore, an application has been developed in order to generate Disguises Model code, starting from UML designs.

Future works will introduce other concepts like adaptability representation at design level, where each aspect could be added or suppressed at design, compile or runtime. Additionally, final version of AspectJ will be generated.

1. Bibliography

[1] Grady Booch. *Object oriented design with applications*. The Benjamin Cummings publishing Co. Inc., 2nd Edition, 1994.

[2] S. Matsuoka , K. Wakita and A. Yonezawa, *Synchronization constraints with inheritance: What is not possible-so what is?* Technical Report 10, Department of Information Science, the University of Tokyo, 1990

[3] W. Hursch, C.V. Lopes. *Separation of Concerns*. Northeastern University, February 1995.

[4] L. Bergmans. *Composing Concurrent Objects*. Ph.D. Thesis, University of Twente, 1994.

[5] Gregor Kiczales et all. *Aspect-Oriented Programming*. European Conference on Object-Oriented Programming (ECOOP), 1997.

[6] Fernando Sánchez Figueroa. *Modelo de disfraces: hacia hacia la adaptabilidad de restricciones de sincronización en los LCOO*. PhD Thesis. Departamento de Informática, Universidad de Extremadura, Spain 1999.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[8] Antonio Manuel Ferreira Rito da Silva. *Concurrent Object-Oriented Programming: Separation and composition of concerns using design patterns, pattern languages, and object-oriented frameworks*. PhD Thesis. Lisbon University.

[9] Junichi Suzuki and Yoshikazu Yamamoto. *Extending UML with aspects: Aspect support in the design phase*. 3rd Aspect-Oriented Programming (AOP) WorkShop al ECOOP'99.

[10] Dirk Bäumer et all. *The role object Pattern*. The 4th Pattern Languages of Programming Conference (PLoP '97).

[11] Object Management Group. *Unified Modelling Language version 1.3.* http://www.rational.com/uml/resources/documentation.