ELSEVIER

Data & Knowledge Engineering 40 (2002) 315–353

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Automated code generation of dynamic specializations: an approach based on design patterns and formal techniques

Vicente Pelechano, Oscar Pastor [*], Emilio Insfrán

Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Camí de Vera s/n, E-46022 Valencia, Spain

Received 12 December 2000; received in revised form 9 July 2001; accepted 10 October 2001

Abstract

In this work, we present an automatic code generation process from conceptual models. This process incorporates the use of design patterns in OO-Method, an automated software production method, which is built on a formal object-oriented model called OASIS. Our approach defines a precise mapping between conceptual patterns, design patterns and their implementation. Design patterns make the code generation process easy because they provide methodological guidance to go from the problem space to the solution space. In order to understand these ideas, we introduce a complete code generation process for conceptual models that have dynamic specialization relationships. This proposal can be incorporated into CASE tools, making the automation of the software production process feasible. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Conceptual modeling; Object orientation; Formal languages; Design patterns; Code generation

1. Introduction

Interest in automatic software production is continuously increasing. Currently, most commercial CASE tools offer some contribution in this area. In these tools, conceptual models are the starting point of the code generation process. Generally speaking the elements used by these tools in the Conceptual Modeling phase do not have a well-defined semantics. This lack of rigor makes

[*] Corresponding author. Tel.: +34-6-387-7350; fax: +37-6-387-7359.
E-mail addresses: pele@dsic.upv.es (V. Pelechano), opastor@dsic.upv.es (O. Pastor), einsfran@dsic.upv.es (E. Insfrán).

it difficult to produce software which is functionally equivalent to the conceptual model in an automated way. Thus, it may be of interest to pose new production processes which are oriented towards automated software production.

In this work, we present a *code generation process* that obtains the representation in the *solution space* of conceptual patterns [1] specified in the conceptual modeling phase (*problem space*). This process is an essential part of an automatic software production method called the *OO-Method* [23,24]. The OO-Method provides an object-oriented conceptual modeling environment that includes model-based code generation capabilities and integrates formal specification techniques (the OASIS formal language) [18,25] and conventional notations like UML [12]. The process incorporates the use of patterns (*architectural* [6] and *design* [10]) to obtain the application architecture and its implementation. The use of patterns offers interesting benefits:

- It is useful for structuring the code generation process.
- It provides quality design solutions that are abstract enough to be used in any programming language.

In order to understand these ideas, we introduce a complete code generation of conceptual models that contain *dynamic specialization* relationships. The *dynamic specialization* conceptual pattern can have several interpretations when it is used in the conceptual modeling phase. It can be very difficult to implement if it does not have a precise semantics. There exist several approaches [7,14,33,40] that have accurately defined the semantics of dynamic specialization; however, they have not implemented it. OO-Method *dynamic specialization* is based on the inheritance formal model defined in OASIS [18]. The OO-Method has been created on the formal basis of OASIS. In fact, the OO-Method graphical notation is based on OASIS abstractions. Our proposal deals with the dynamic specialization from its specification in the *problem space* to its implementation in the *solution space*.

This paper is organized as follows: Section 2 presents a series of existing problems in current software production and introduces a solution based on the OO-Method automated code generation process. This section shows the basic ideas of the OO-Method approach and explains the Execution Model as an essential element in achieving complete code generation. It also details the phases of the code generation process proposed by the Execution Model. Section 3 presents the conceptual pattern *dynamic specialization* in the problem space. It explains its semantics showing how it is specified in OASIS and presents how the conceptual pattern is modeled using the OO-Method graphical notation. Section 4 shows the application of the code generation process to translate the dynamic specialization concept to the corresponding software representation in the *solution space*. In this section, we explain how design patterns are incorporated into the code generation process, and we explain how to define the mapping between the conceptual pattern and a given set of design patterns (in this case *State* and *Template Method* patterns [10]). These patterns allow us to implement the structure and behavior of *dynamic specialization* by applying the Execution Model proposed by the OO-Method. To understand this proposal, we will include an example of Java code generation that follows the code generation process shown in Section 2. Section 5 analyzes related works and Section 6 presents conclusions and further work.

[1] In our approach the term conceptual patterns is used to refer to the conceptual structures, concepts or abstractions that are used to represent elements and their relations in an application domain. We apply this term in the conceptual modeling context.

2. From problem space to solution space. The OO-Method approach

In current software development proposals [37], design patterns act as a bridge between conceptual patterns in the Domain Model and their implementation. Some problems are detected when analyzing how to achieve the transition between domain models and their implementation through design patterns:

- Industrial Object-Oriented (OO) Modeling Methods do not clearly describe domains because the languages that they use are not rich enough.
- Design Patterns are too general and not formalized enough.
- There does not exist a precise mapping between domain models, design patterns and software components.
- The transition between domain models, design patterns and software components is manually achieved. The implementation of conceptual pattern behavior is left to the programmer.

These problems make it difficult to build tools that are capable of producing software systems in an automatic way.

2.1. A possible solution

These problems can be solved by:

- Using Formal Languages (FL) or models based on FL to describe Conceptual Patterns in a precise way. We will use the OASIS formal model to specify conceptual patterns such as static and dynamic specializations, role classes, and several kinds of aggregation.
- Specializing Design Patterns (State, Role Object, Composite, Template Method, etc) to support the Conceptual Patterns used in Domain Modeling.
- Defining precise mappings between Conceptual Patterns and Design Patterns. These mappings must preserve the semantics of conceptual patterns.
- Defining an execution strategy to implement the behavior of Conceptual Patterns.

2.2. The OO-Method approach

The ideas presented above can be joined in a software production method that will give support to an automatic code generation process from conceptual models. The OO-Method is an automated software production method based on a formal object-oriented model called OASIS. The OO-Method provides a methodological approach that follows two phases (see Fig. 1):

1. The building of a *Conceptual Model* that collects the Information System's relevant properties (static and dynamic). To model system properties we provide to the modeler a well-known graphical notation (UML *compliant*). A formal and OO OASIS specification is obtained from the system description using a well-defined translation strategy. This translation process can be done because there is a well-defined *one to one mapping* between the graphic modeling elements and the concepts of the specification language. The Conceptual Model is placed in the *problem space*.
2. The application of an *Execution Model* to the Conceptual Model obtained in the previous phase. This Execution Model is a propietary strategy (introduced by our method) that accurately states the *implementation-dependent* features in order to represent the Conceptual Model

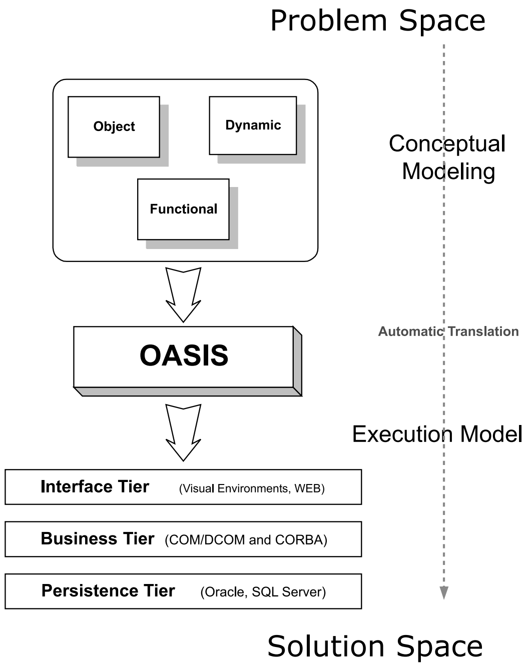
Problem Space
Object
Dynamic
Functional
Conceptual Modeling
OASIS
Automatic Translation
Execution Model
Interface Tier (Visual Environments, WEB)
Business Tier (COM/DCOM and CORBA)
Persistence Tier (Oracle, SQL Server)
Solution Space

Fig. 1. Graphical representation of the OO-Method models.

in a given development environment. This model proposes a code generation strategy, which obtains the representation of the modeling elements in a selected programming language according to a set of specific patterns. The Execution Model is placed in the *solution space*.

2.3. The execution model and the automated code generation

The Execution Model is essential to achieving a systematic and automated transition from *Problem Space* to *Solution Space*. The OO-Method proposes an *abstract execution model* that builds a *representation* of the conceptual model (including *static* and *dynamic* aspects) for any target software development environment starting from the system specification.

The Execution Model provides:

- An architecture for the system by means of *architectural patterns*.
- A *code generation strategy* to obtain the software components of the architecture. It is based on *specialized design patterns* and an *execution strategy* that objects must follow.

2.3.1. Application architecture

An important step in the application design process is the definition of the application architecture. For this purpose, we use *architectural patterns* which are adapted to the characteristics of the target application. In the OO-Method, we apply the *multitiered architecture* for developing business applications. This architectural pattern divides the application into three logical tiers: the *interface* tier, the *business* tier and the *persistence* tier. The selection of this pattern is due to the high degree of independence that exists between components at different tiers. It is a *closed architecture* [2] [29] that reduces the dependencies between tiers, and it allows for making changes easily. Architectures of this kind allow us to structure the code generation process by distributing the application in three logical tiers in a suitable way.

Once the architecture has been defined, it is necessary to obtain the software components that implement the functionality corresponding to the tiers. In this work, we are going to focus on the *code generation strategy* that allows us to obtain the software components of the business [3] tier in a systematic way. These components completely implement the structure and behavior of conceptual patterns. In this proposal, the dependencies between tiers are not showed in order to provide a more generic solution for conceptual pattern implementation. In this way this solution can be applied to another software architectures.

2.3.2. The code generation strategy

The *code generation strategy* defines precise mappings between conceptual patterns (specialization, aggregation, role classes) and its representation in a software development environment. The basic elements of this strategy are the use of *design patterns* and the application of an *execution strategy*.

The input to this process is a conceptual model (in graphical and textual mode). It is made up of a set of conceptual patterns based in OASIS concepts. The code generation strategy follows the steps below (see Fig. 2):

1. *Design patterns selection and/or creation.* Starting from the OASIS concepts, we have to look for and/or build design patterns that allow us to face the problem of properly representing the conceptual patterns in the solution space. In this step it is necessary to have a clear idea of the possible mappings between conceptual and design patterns. Design patterns can be adapted/specialized in order to support the modeled abstractions.
2. *Representation of the structural relationships preserving the semantics of OASIS concepts.* A set of mappings between design patterns and conceptual patterns must be defined. In this step, we determine the structure of classes in the *solution space* that implements the classes in the *problem space* in a way that preserves its semantics.
3. *Implementation of the behavior associated to service execution.* An *execution strategy* will be implemented to assure that the implementation of a service accurately represents the effect specified in the conceptual model. The execution strategy is a key element in the code generation process. It is a set of actions that implements the effect of a service execution. This strategy constitutes the basis for generating the behavior of software components in the application tier. Following the OO-Method approach, an object service execution is characterized by the occurrence of the following sequence of actions:
   (a) *Check state transition*: verifies that a valid transition exists for the selected service in the current object state of its State Transition Diagram (STD). [4] The STD specifies the process that represents the valid object life cycle for each class.

[2] One tier only uses services from the tier immediately below it.

[3] The OO-Method provides a complete process of code generation that includes the User Interface generation. It is generated using the information modeled in a Presentation Model placed at the Conceptual Level.

[4] Similar to the step semantics of the Object Life Cycle Diagrams introduced in [30].

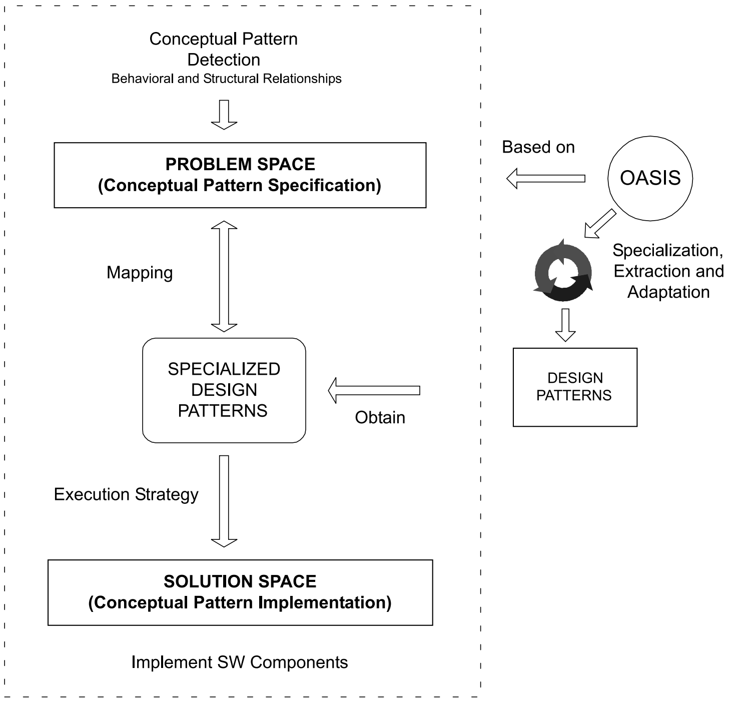
Conceptual Pattern Detection
Behavioral and Structural Relationships
PROBLEM SPACE (Conceptual Pattern Specification)
Mapping
SPECIALIZED DESIGN PATTERNS
Obtain
Execution Strategy
SOLUTION SPACE (Conceptual Pattern Implementation)
Implement SW Components
Based on
OASIS
Specialization, Extraction and Adaptation
DESIGN PATTERNS

Fig. 2. Code generation strategy.

(b) *Precondition satisfaction*: checks whether the precondition associated to the service holds. If either (a) or (b) fails, an exception will arise informing that the service cannot be executed.

(c) *Valuation fulfillment*: the induced service modifications take place in the current object state.

(d) *Integrity constraint checking in the new state*: the integrity constraints are verified in the final state to assure that the service execution leads the object to a valid state. If the constraint does not hold, an exception will arise and the previous change of state is ignored.

(e) *Trigger relationships test*: the set of *condition-action* rules that represents the internal system activity is verified after a valid change of state. If any of them hold, the specified service will be triggered.

The previous steps guide the implementation of any program to assure the functional equivalence between the object system specification collected in the conceptual model and its reification in a programming environment. This strategy will be implemented using an algorithm based on the *Template Method* design pattern.

The process of *searching* and *specialization* of design patterns (step 1), the definition of the mappings (step 2), and the definition of the execution strategy (step 3) is defined (manually) by our approach for every kind of conceptual pattern. The code generation strategy constitutes the basis for the building of code generators that automate the software production process. The code generators take a conceptual model as a source specification and generate code for this model which is consistent with its semantics. This code is obtained by automatically applying the specialized design patterns and the defined mappings to the conceptual patterns included in the conceptual model.

We will choose a basic conceptual pattern (present in OO-Method/OASIS) on which to apply the code generation strategy defined by the Execution Model. The selected conceptual pattern is the *dynamic specialization*. In the following section, we are going to show (step by step) how to cover the distance between the dynamic specialization in the *problem space* and the *solution space*.

3. Dynamic specialization in the problem space

The input to the code generation process will be the conceptual pattern specification. In this section, we are going to introduce how the *dynamic specialization* is specified in the *problem space* in an intuitive way. We will present its properties, its semantics, its specification in OASIS, and how it is modeled using the graphical notation that the OO-Method provides.

3.1. Formal semantics

The OASIS formal model defines the conceptual patterns used by the OO-Method in the conceptual modeling phase. In this section, we are going to show the properties of the OASIS dynamic specialization.

3.1.1. OASIS basic concepts

*O*pen and *A*ctive *S*pecification of *I*nformation *S*ystems (OASIS) is a formal approach for object-oriented conceptual specification of information systems. An OASIS specification is a presentation of a theory in the formal system used and is expressed as a structured set of *class* definitions. Classes can be *simple* or *complex*. A complex class is defined in terms of other classes (simple or complex) by establishing relationships among classes. These relationships provide *aggregation* or *specialization* mechanisms. A class has a name, one or more identification mechanisms for its instances (*objects*) and a type or template that is shared by every instance belonging to the class. Each object has a unique identifier (*oid*) set by the system; however, objects are referred by their identification mechanisms which belong to the problem space. A function establishes a mapping between the identification mechanisms and the *oid*. The type or template describes the structure and behavior of every object.

Thus, each object encapsulates its own state and behavior rules. As is usual in object-oriented environments, objects can be seen from two points of view: *static* and *dynamic*. From the *static* perspective, the *attributes* are properties describing the object structure. The object state at a given instant is the set of structural property values. From the *dynamic* perspective, the evolution of objects is characterized by the "change of state" notion. The occurrence of actions implies changes (by means of *valuations* and *derivations*) in the values of the attributes. Object activity is determined by a set of rules: a *process*, *preconditions*, *triggers*, and *transactions*.

The OASIS semantics can be found in [18], where the formulas and process specifications that are used to specify the sections of an OASIS class template are introduced. The complete class template is a set of formulas that belongs to a Dynamic Logic variant formalized in [20].

3.1.2. Specialization in OASIS

In OASIS , IsA relationships are basically expressed by means of two types of relations between classes: *specialization* (with static and dynamic partitions) and *player/role*. They establish an incremental specialization mechanism for class properties and a pre-order relation between class templates. In OASIS, specializing a class means to create one or more partitions of it. A *partition* is a set of subclasses that specializes the superclass taking into account some criterion. A superclass can be partitioned into one or more partitions. An object, at a given instant, is an instance of one subclass in every partition. In this paper, we are going to focus on *dynamic partitions*. Next, we give the characteristics of *dynamic partitions*.

3.1.3. Dynamic partitions and dynamic specialization

Instances in a dynamic partition can migrate between subclasses during their lifetime. This feature is called dynamic specialization. Object migration between subclasses is produced by the occurrence of actions (transition between subclasses is specified in OASIS by means of a migration process, see Example 1) or by attribute values (specified in OASIS by means of conditions over a variable attribute, see Example 2).

Example 1. A dynamic specialization of class `car` produced by the occurrence of the actions `new_car`, and `repair_car` in OASIS:

```
working, broken_down dynamic specialization of car
    migration relation is
    car = new_car. working;
    working = break_down. broken_down;
    broken_down = repair_car. working;
```

The syntax of the migration process uses process terms of a process algebra which is based on the approach introduced by Wieringa [39]. In this example `car`, `working` and `broken_down` are process variables that represent a state in a process. The name of these variables is the name of the classes involved in the partition. The migration process specifies the sequence (using the sequence operator (.)) in which an object migrates from one subclass to another subclass in the partition due to an action ocurrence. The interpretation of the migration process of the example is that the creation of a car instance (`new_car` action) implies that it starts belonging to the `working` class. The action `break_down` implies leaving the subclass `working` and migrating towards `broken_down` class. The action `repair_car` implies leaving the subclass `broken_down` and migrating towards `working` class. As an instance of the `working` class, actions from `car` and `working` templates can be recognized.

From a theoretical point of view, the process representing the life of a `car` instance is the union of the processes defined in every subclass. The connections among subclasses are given by the actions included in the migration process.

Example 2. A dynamic partition of the class `person` defined over the `age` attribute:

```
child where {age < 13}
teenager where {13 <= age and age <= 19}
```

```
  adult where {19 < age}
      dynamic specialization of person;
```

In this example, when the attribute `age` changes, the instance of `person` could migrate between subclasses `child`, `teenager` and `adult`, depending on the specified conditions.

In this model a partition is always a disjoint cover of a set of subclasses. In order to assure that these properties always hold in a dynamic partition defined over a variable attribute, we require to the modeler that: (1) the rank of values that the attribute can take in a specialization condition must be disjoint with respect to the values that it can take in the other specialization conditions (those specified for the other subclasses of the partition); (2) specialization conditions must cover every possible value that the attribute can take in the system.

3.1.4. Characteristics of a subclass in a dynamic partition

Given a dynamic partition $\{S_1, \ldots, S_n\}$ of the class P:

- The set of *properties* of any subclass S_i is given as follows:
  - $Atr_{S_i} = Atr_P$ (attributes of P) $\cup\, Atr_{S_i}$ (emerging attributes of S_i).
  - $Ev_{S_i} = Ev_P$ (events of P) $\cup\, Ev_{S_i}$ (emerging events of S_i).
- We have *behavioral compatibility* [5] between P and $\{S_1, \ldots, S_n\}$. Thus, the set of properties of any subclass S_i in the partition must fulfil the following constraints to maintain the behavioral compatibility:
  - *Valuation formulae* in S_i can only modify emergent attributes, and also inherited attributes without valuation in P. The set of valuation formulae of S_i is the union of valuations of P and S_i.
  - If a *precondition* (or *prohibition formulae*) is redefined, then the new condition must imply the P condition. The set of preconditions of S_i is the union of preconditions of P and S_i, overwriting the redefined preconditions.
  - *Triggers* and *integrity constraints* are established in the same way as preconditions.
  - The *process* that represents the life of an object of the parent class is the union of the processes defined in each subclass. Because of this, if an object behaves following the STD of the subclass, it also will behave like the STD of the parent class since the subclass STD is a subset of the STD of the parent class.

3.2. Graphical modeling

In this section we are going to introduce how the OASIS dynamic specialization is modeled by using the OO-Method graphical notation.

3.2.1. Modeling in the OO-Method

Firstly we introduce the notation that OO-Method provides for the conceptual modeling step. The concepts attached to the OASIS formal model determine the relevant information that is

[5] The term behavioral compatibility was introduced by Wegner and Zdonik [38] and means that instances of subclasses behave like instances of the parent class when operations defined in parent class are executed.

necessary to detect in the conceptual modeling phase for building the conceptual schema of an information system. For this purpose, the OO-Method provides a graphical notation that is constituted by three models. The diagrams which are associated to these three models respect and extend the UML notation; however, their semantics are conceived to properly represent only the set of information that is really necessary to describe the information system. The three models that the OO-Method provides are the following:

- *Object Model*. A graphical model where system classes including attributes, services and relationships (*aggregation* and *specialization*) are defined. Additionally, *agent* relationships are introduced to specify who can activate each class service (client/server relationship). The corresponding UML base diagram is the *class* diagram, where the additional expressiveness could be introduced by using the needed stereotypes which is a useful UML facility.
- *Dynamic Model*. A graphical model used to specify valid object life cycles and interobject interaction. We use two kinds of diagrams:
  - *State Transition Diagram* (STD) to describe correct behavior by establishing valid object life cycles for every class. By valid life, we mean a correct sequence of states that characterizes the correct behavior of the objects. Transitions represent valid changes of state that can be constrained by introducing preconditions and control conditions. *Preconditions* are those conditions defined on the object attributes that must hold for a service to occur. *Control conditions* are conditions defined on object attributes to avoid the possible non-determinism for a given service activation. The corresponding UML base diagram is the *state transition* diagram.
  - *Object Interaction Diagram* represents interobject interactions. In this diagram, we define two basic interactions: *triggers*, which are object services that are activated in an automated way when a condition is satisfied, and *global interactions*, which are transactions involving services of different objects. The corresponding UML base diagram is the *collaboration* diagram.
- *Functional Model*. In this model, the semantics associated to any change of an object state is captured as a consequence of a service occurrence. To do this, it is declaratively specified how every service changes the object state depending on the arguments of the service involved (if any), and the object's current state. A clear and simple strategy is given for dealing with the introduction of the necessary information. The relevant contribution of this functional model is the concept of the categorization of attributes [23]. Three types are defined: push-pop, state-independent and discrete-domain based. Each type will fix the pattern of information required to define its functionality. In short, Push-pop attributes are those whose relevant services increase, decrease or reset their value. State-independent attributes are those having a value that depends only on the latest service that has occurred. Discrete-domain valued attributes are those that take their values from a limited domain. The object reaches a specific state, where the attribute value can be specified, through the activation of carrier or liberator services. This categorization of the attributes allows us to generate a complete OASIS specification in an automated way, where a service functionality is completely captured.

From these three models, a corresponding formal and OO OASIS specification is obtained using a well-defined translation strategy. The resultant OASIS specification acts as a complete high-level system repository, where the relevant system information, coming from the conceptual modeling step, is captured.

3.2.2. Dynamic specialization modeling

The specialization is represented in the OO-Method Object Model adapting the notation that the UML class diagram provides. We can see in Figs. 3 and 4 two alternative graphical notations to model specialization relationships. These two types of representations do not force a particular semantics; however, in the OO-Method notation we are going to give a different semantics to each representation. Subclasses whose arrows are joined into one arrow, constitute one *partition* (see Fig. 3), and a subclass that has its own arrow (arrow is not shared with other subclasses) constitutes one *partition* on its own (see Fig. 4).

We use the type of class diagram shown in Fig. 3 to represent a dynamic partition. We will add a *discriminator* label and a *stereotype* label ("dynamic") to the proposed class diagram in order to represent the specialization *criterion* and the *type* of specialization, respectively. Both labels are attached to the specialization arrow. In addition to these labels, in the dynamic model it will be necessary to specify a state transition diagram (called migration diagram) attached to each dynamic partition. This diagram will model the *migration process*. In this diagram, we will specify any constraint that we want to impose with regard to the allowed transitions between subclasses of the same partition. The allowed transitions between subclasses can be caused (as we have seen previously) by the occurrence of events or by attribute values. The proposed notation for modeling the migration constraints of a dynamic partition will be one of the following:

- *By event occurrence*. Events will label the transitions between the migration diagram states. The states of the migration diagram will match each one of the partition subclasses (see Fig. 6 for the migration diagram of the dynamic specialization shown in Fig. 5).
- *By attribute value*. We can use one or both of the following techniques:
  - Migration constraints could be attached to each subclass in the class diagram. An object will migrate to a subclass in the partition if the constraint attached to the target subclass holds (see Fig. 7).

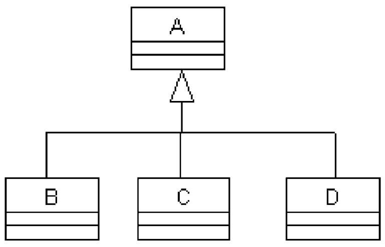

Fig. 3. Graphical representation of a partition.

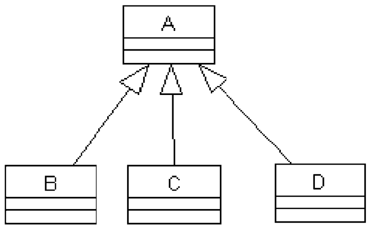

Fig. 4. Graphical representation of three partitions.

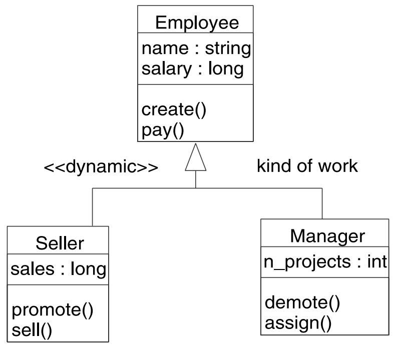

Fig. 5. Dynamic specialization of employee based on migration events.

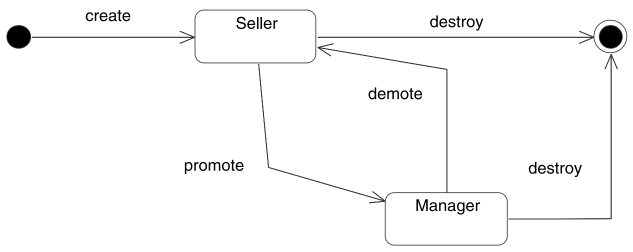

Fig. 6. STD representing the migration process of `Employee` class in Fig. 5.

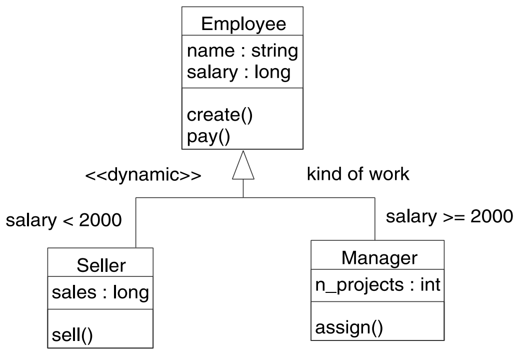

Fig. 7. Dynamic specialization of employee based on attribute values.

- ○ Migration constraints could be defined in the migration diagram. The migration constraints will act as guards (nothing happens if condition is false) on the transitions of those events that modify the attributes included in the migration constraints. These events will label the transition between the migration diagram states (see Fig. 8). This technique is more expressive than the specification of migration constraints. On the other hand, the use of the migration constraints is easier.

In Figs. 5 and 7, there are two class diagrams (in OO-Method notation) modeling a dynamic specialization of class `Employee`. Both examples are modeling that any `Employee` must be a

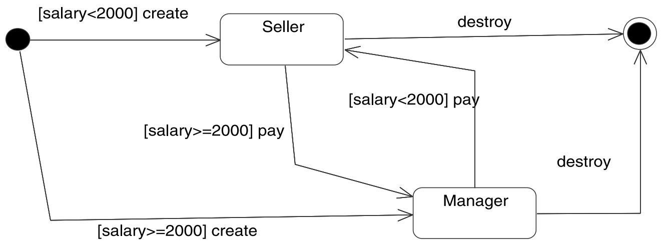

Fig. 8. STD representing the migration process of employee using attributes.

`Seller` or a `Manager`. In Fig. 5, when an `Employee` is created, it starts by belonging to a `Seller` subclass. The action `promote` implies leaving the subclass `Seller` and migrating towards `Manager` subclass. A `Manager` can be `demoted`; this action migrates the object towards `Seller` class. This semantics is specified by a STD in Fig. 6 that represents the migration diagram. The class diagram of Fig. 7 is another example of dynamic specialization. This example defines a different migration constraint based on variable attributes. Its corresponding migration diagram is shown in Fig. 8. When an `Employee` is created, it starts by belonging to a `Seller` or `Manager` subclass depending on the initial value given to the salary attribute. If a seller is paid and the salary is greater or equal to \$2000, then the Seller leaves the subclass `Seller` and migrates towards the `Manager` subclass. If a manager is paid and the salary is less than \$2000, then the manager migrates towards the `Seller` class. These examples will be used to illustrate the code generation process.

4. Dynamic specialization in the solution space

In this section, we are going to present how the code generation strategy is applied to a dynamic specialization which is specified in the problem space. Firstly, the process of choosing a design pattern that fits easily into the dynamic specialization will be introduced. In the second subsection, the mappings that define how to represent the structure of a dynamic specialization by means of the selected design pattern are introduced. In the third subsection, how to implement the object behavior following the execution strategy will be presented. Finally, the implementation of the constructors, the destructors, and the migration events will be introduced.

4.1. Choosing a design pattern. The State pattern

The implementation of the *dynamic specialization* requires great effort because mainstream OO programming languages only support *single static* classification. Currently, there are several techniques that are used to implement dynamic specialization: *flags*, *delegation to a hidden class*, *replacement* and the *State* design pattern [9]. Among them, the *State* pattern solution fits very well with the semantics of the dynamic specialization in OASIS.

The *State* pattern is a *behavioral pattern* [10] that allows an object to alter its behavior when its internal state changes. The object will appear to change its class. This solution is based on the *delegation to a hidden class* technique. This delegation is possible because all the subclass methods are moved to the superclass interface, and an instance variable is defined in the superclass to reference an object of one of the subclasses. The *State* pattern is characterized by three kinds of participant classes: ConcreteState, State and Context. The ConcreteState are subclasses of the class State. Each subclass implements the state-dependent behavior of the Context class. The State class is an abstract class [6] that defines an interface for encapsulating the behavior associated with any particular state of the Context. The Context class interface has all the methods of the ConcreteState subclasses. It maintains an object-valued attribute that will store an instance of a ConcreteState subclass. This attribute defines the current state of the Context class and it is called State Object. Context class will delegate state-specific behavior to the State Object.

4.2. Representation of a dynamic partition using the State pattern

The next step in the code generation process is the representation of the structural relationship induced by the conceptual pattern on a design structure. This process will be carried out in such a way that the final implementation must preserve the semantics of the conceptual pattern according to the OASIS object model. In order to achieve this purpose, we will define a series of *mappings* between the selected design pattern and the conceptual pattern. These mappings will allow us to obtain a *specialized design pattern* with a precise semantics. The mappings will determine:

- The structure of the classes in the *solution space* that implements the conceptual pattern in the *problem space*. This step provides:
  - The distribution of *attributes* in each class of the design structure.
  - The *methods* that implement the *events*.

Next, we are going to introduce how to define the proposed mappings. We are going to use the example of Fig. 5 to document the code generation process using Java as the programming language.

4.2.1. Defining the structure of the classes

Given a dynamic partition $\{S_1, \ldots, S_n\}$ of the class P, its representation (see Fig. 9) will be determined by the classes C_P (Context), C_A (State) and $\{C_1, \ldots, C_n\}$ (ConcreteState), such that:

- C_P will have:
  - A set of attributes Atr_{C_P} that are not modified by any class in the partition, and one attribute A_{C_A} (called State Object) that will store an object of one of the subclasses in $\{C_1, \ldots, C_n\}$.
  - A set of public methods M_{C_P} that implement the events $Ev_P \cup Ev_{S_1} \cup \cdots \cup Ev_{S_n}$. Ev_P will be completely implemented in class C_P. $Ev_{S_1} \cup \cdots \cup Ev_{S_n}$ will be implemented through delegation to the object A_{C_A}.
- C_A is an abstract class that will have:
  - A set of attributes $Atr_{C_A} = Atr_{S_1} \cap \cdots \cap Atr_{S_n}$.
  - A set of public abstract methods M_{C_A} that defines the interface necessary to implement the set of events $Ev_{C_A} = Ev_{S_1} \cup \cdots \cup Ev_{S_n}$. Each subclass in $\{C_1, \ldots, C_n\}$ will redefine these methods.

[6] In non-OO languages that support interfaces (like MS Visual Basic), we can substitute the term interface for abstract class.

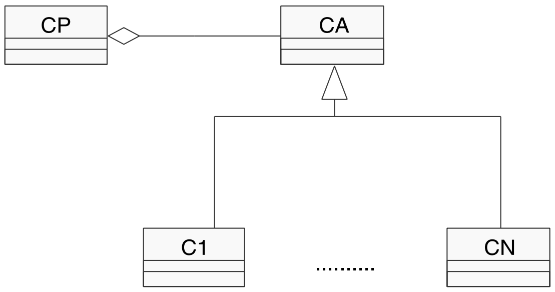

Fig. 9. Class structure of State pattern.

- $\{C_1, \ldots, C_n\}$ are subclasses of C_A and each C_i will have:
  - A set of attributes Atr_{C_i} to store Atr_{S_i} (emergent attributes of S_i).
  - A set of public methods M_{C_i} that implement Ev_{S_i} (emergent events of S_i) through redefinition.

We have adopted the *Data Members* pattern [8], a variation of *State*, to know the attributes that each class will have in the pattern. It is important to note that, in the structure of the classes C_P and C_A, we must define an additional attribute (which is not present in the signature of the OASIS class) in order to store the current object state in the STD.

To illustrate the code generation process, we are going to use the example of `Employee` shown in Fig. 5. Fig. 10 shows the design (in UML notation) after applying the mappings proposed. Its implementation in Java is the following:

```
public class Employee
{
  String name;
  StateObject StateEmployee;
  String StateSTD;
  // current state in the STD
  public Employee();
```

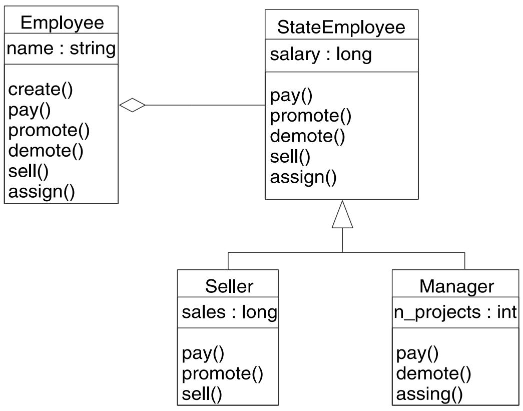

Fig. 10. Mapping the employee dynamic specialization into the State pattern.

```
    public create();
    // constructor method
    public void pay();
    public void promote();
    public void demote();
    public void assign();
    public void sell();
    public String class_name();
    // returns the class name of the current State Object
  }
  abstract public class StateEmployee
  {
    protected int salary;
    protected String StateSTD;
    ...
    public abstract void pay();
    public abstract void promote();
    public abstract void demote();
    public abstract void sell();
    public abstract void assign();
    public abstract String class_name();
    // returns the class name of the current State Object
  }
  private class Seller extends StateEmployee
  {
    protected int sales;
    ...
    public void pay();
    public void promote();
    public void sell();
  }
  private class Manager extends StateEmployee
  {
    protected int n_projects;
    ...
    public void pay();
    public void demote();
    public void assign();
  }
```

Method implementation and delegation. The implementation of the methods included in the class structure will be carried out in the following way:

- The *state independent* behavior (Ev_P events of the P class) will be implemented in C_P class by the M_{C_P} methods.

- The implementation of the M_{C_P} methods that implement the effect of the *state dependent* events $Ev_{S_1} \cup \cdots \cup Ev_{S_n}$ (events which depend on the class of the dynamic partition) will be done through *delegation* on the object stored in A_{C_A}.
- The methods of the C_A class will be abstract and will be redefined by each class in the partition $\{C_1, \ldots, C_n\}$.

In the dynamic partition of the example, each subclass will earn a different `salary`; therefore, the method that implements the event `pay` in class `Employee` will behave depending on the active subclass. Thus, the implementation of the pay method will delegate to the `State Object`.

```
public void pay()
{
    StateObject.pay(); //delegates on StateObject
};
```

4.3. The execution strategy and the Template Method pattern

The last step in the code generation process is the *implementation of the behavior* which is associated to the *service execution*. This behavior is based on the *execution strategy* introduced previously. We apply the *Template Method* pattern to implement the behavior of object services that follow the execution strategy. The *Template Method* defines the skeleton of a generic algorithm. Subclasses can redefine some steps without changing their structure.

The *execution strategy* defines the following sequence of actions in detail:
- Checking state transition correctness.
- Checking precondition satisfaction.
- Valuation fulfillment.
- Integrity constraints checking in the new state.
- Testing trigger relationships.

This set of actions constitutes a *generic algorithm* that has to be applied to any service execution in order to guarantee that objects behave according to the execution model of the OASIS formal model. This algorithm will always be the same, only the implementation of actions can be redefined in the subclasses.

We will apply the *Template Method* pattern to the *State* pattern in the following way:
- C_P, C_A and $\{C_1, \ldots, C_n\}$ classes will declare the methods that implement the actions defined in the execution strategy to be protected [7] (only visible to subclasses).
  - ○ C_P will implement the actions that do not depend on the active subclass of the partition.
  - ○ C_A will define the execution model actions as abstract methods.
  - ○ Each class in $\{C_1, \ldots, C_n\}$ will redefine the corresponding method in C_A if any *precondition*, *valuation*, *integrity constraint* or *trigger* has been added or redefined. [8] Following the guidelines defined to respect the behavioral compatibility we have that:

[7] It should be abstract in C_A.

[8] If no formula has been added or redefined the method will be implemented as a blank method. It is not necessary to include any exception handling in the method implementation because the code generation process proposed guarantees that this code will not be accessible in this situation.

  – *valuations* will be the union of the valuations defined for the subclass and its superclass;
  – *preconditions*, *integrity constraints*, *triggers* and *processes* will be those defined for the subclass if it has redefined them (redefinitions are expected to respect the behavioral compatibility so that the most restrictive will prevail). Otherwise, they will be those defined in the superclass.
- Events specified in the conceptual model will be implemented by means of public methods. These methods will implement the *execution strategy* algorithm.

As an example, we can see the declaration of the `Seller` class and the implementation of its `pay` event following the execution strategy:

```
public class Seller extends StateEmployee
{
  ...
  // methods that implement the actions of the execution strategy
  protected void check_state_transition(String Service);
  protected void check_preconditions(String service);
  protected void check_integrity_constraints();
  protected void check_triggers();
  protected void do_pay();
  // valuation of pay event
  protected void do_sell();
  // valuation of sell event
  // methods that implement events pay and sell
  public void pay();
  public void sell();
};
```

For instance the `pay` method declared in the `Seller` class will have the following body:

```
public void pay()
{
  check_state_transition(‘‘pay’’);
  check_preconditions(‘‘pay’’);
  do_pay();
  check_integrity_constraints();
  check_triggers();
};
```

Having presented the design pattern that implements the algorithm to activate a service, the next step is to describe how the primitive operations (steps in the execution strategy) proposed in the algorithm that has been presented above are implemented.

4.3.1. Checking state transition correctness

The first step in the execution strategy is to check that a valid transition exists in the STD specified in the conceptual modeling phase. There are basically three different ways to implement the verification of a valid state transition; conditional logic, the State pattern or a state machine interpreter that runs against a set of state transition rules. It has been recognized that the applicability of the state pattern may not be suitable if there are many states in the system [10]. The implementation of a state machine interpreter is beyond the scope of our proposal. Consequently, we have chosen the use of conditional logic.

An STD belonging to a given class C could be seen as a directed graph $G = (S, T)$ such that:

- S is a set of possible object states that are represented by graph nodes, and
- T is a set of state transitions represented by labeled arrows.

A state transition $t \in T$ is a tuple (s_i, s_j, e, p, cc) where

- s_i is the source state of t,
- s_j is the destination state of t,
- e is a service of class C signature,
- p is an optional *precondition* attached to e, and
- cc is an optional *control condition* of transition t.

In order to generate the `check_state_transition` method that verifies whether a valid transition exists for any service in the current object state, we define a method `check_state_transition` with `Service` as an argument. The body of this method must be generated according to the following steps.

For each state $s_{i_{/i=0..n}} \in S$, we have to generate a conditional structure that checks whether the current object state is equal to s_i.

For each service e such that $\exists t \in T/t = (s_i, s_j, e, \_, cc)$ and s_i is equal to the current object state, we have to generate:

1. A conditional sentence that checks whether `Service` is equal to e and the control condition cc of e holds.
2. An assignment sentence that assigns the s_j value to the current object state if the conditional sentence defined in step (1) holds.
3. An exception sentence to determine whether a valid transition exists in the STD, meaning by valid that the conditions defined in step (1) hold. The generated error message is "state transition violation".

As we have mentioned above in the dynamic specialization section, the process (STD) of the parent class is built through the union of the processes of the subclasses. Because of this, if an object behaves following the STD of the subclass, it also will behave like the STD of the parent class. In this way, each subclass of the dynamic partition will implement its own method `check_state_transition` following the previous steps.

As an example, we can see the STD of the `Seller` subclass (see Fig. 11) and its corresponding `check_state_transition` method implementation.

```
protected void check_state_transition(String Service)
throws ServiceNotAllowed
{
switch (stateSTD)//StateSTD is the current state of the STD
  {
  case ''S_INI'':
    switch(Service)
```

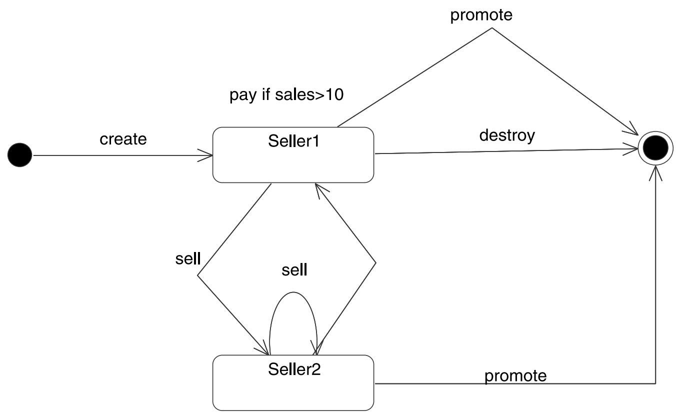

Fig. 11. STD of the `Seller` subclass.

```
      {
       case ‘‘create’’:
         stateDTD= ‘‘Seller1’’;
         break;
       default: throw new ServiceNotAllowed();
      }
      break;
   case ‘‘Seller1’’:
      switch(Service)
      {
       case ‘‘sell’’:
         stateDTD= ‘‘Seller2’’;
         break;
       case ‘‘destroy’’:
         stateDTD= ‘‘S_DESTROYED’’;
         break;
       case ‘‘promote’’:
         stateDTD= ‘‘S_DESTROYED’’;
         break;
       default: throw new ServiceNotAllowed();
      }
      break;
   case ‘‘Seller2’’:
      switch(Service)
      {
       case ‘‘sell’’:
         stateDTD= ‘‘Seller2’’;
         break;
       case ‘‘pay’’:
        stateDTD= ‘‘Seller1’’;
        break;
       case ‘‘promote’’:
        stateDTD= ‘‘S_DESTROYED’’;
        break;
       default: throw new ServiceNotAllowed();
      }
      break;
    }
  };
```

For each concrete state specified in the STD of Fig. 11 (in this example these states are non-existence, Seller1, Seller2 and post-mortem), we have declared a conditional structure. This structure ensures that a valid transition exists for the service that is being activated. If this process succeeds, the corresponding change of state is carried out. Otherwise, an exception will arise. For example, let us suppose that the current state of a Seller object is Seller1 and that the service that is being activated is the service sell(). The checkStateTransition method presented above determines that this is a valid transition and sets the new state of the reader object to Seller2.

4.3.2. Checking precondition satisfaction

Given a service *e*, a `check_preconditions` method has to verify whether its precondition holds. The generation process defines a protected method `check_preconditions` with `Service` as an argument. The body of this method must be generated according to the following steps:

For each class service *e* having a precondition $p(e)$ we have to generate:

1. A conditional structure that checks whether $p(e)$ holds.
2. An exception sentence for showing an error message when the precondition of service *e* has been violated.

Event preconditions redefined in a subclass will be more constrained than those specified in the parent class. This is due to the behavioral compatibility required by the dynamic partitions. Thus, the `check_preconditions` method that has to be executed will be obtained according to the following conditions:

- If an object of a subclass has to execute an inherited event of the superclass, then the method that must be executed is:
  - The `check_preconditions` method implemented in the subclass if the event has redefined its preconditions.
  - The `check_preconditions` method implemented in the superclass if the event has not redefined its preconditions.
- If an object of a subclass has to execute an event specified in its own signature, then the method that must be executed is the `check_preconditions` method implemented in the subclass.

Following with the example of Fig. 5, the precondition specified for the `pay` event in Fig. 11 will be implemented in the following way:

In the `Employee` class:

```
protected void check_preconditions(String Service)
throws PreconditionViolation
```

```
{
  switch(Service)
  {
    case ''pay'':
     if (StateObject.class_name().equals(''Seller''))
     // the object is specialized in the Seller subclass
     StateObject.check_preconditions(Service);
     // execute the subclass method because the pay event
     // has a precondition defined in the Seller subclass
     break;
    case ''sell'':break;
    // Other services having preconditions
    default:
  }
};
```

In the `Seller` class:

```
protected void check_preconditions(String Service)
throws PreconditionViolation
{
  switch(Service)
  {
    case ''pay'':
     if (!(sales>10)) throw new PreconditionViolation();
     // checks the precondition of the pay event
     break;
    case ''sell'':break;
    // Other services having preconditions
    default:
  }
};
```

4.3.3. Valuation fulfillment

A method must implement the change of the object attribute values according to their categorization in the Functional Model (FM). Valuations in the FM are defined by using formulas of the form $\Phi[a]\Phi'$, where Φ and Φ' are wff[9] whose semantics is the following: if service a is activated and the object is in the state Φ, the effect of a service execution will leave the object in the state Φ'. The formula Φ must be true in the object state previous to the occurrence of service a. The formula Φ' specifies the effect that the service executed has over object attribute values.

[9] Quantification is not allowed. Φ' is a wff which is built using atoms with the relational operator '$=$' and the "`and`" connective.

The generation process defines a protected method `do_ServiceName` with `Parameters` (that are the arguments of `ServiceName`) as argument. The body of this method must be generated according to the following steps:

For each class service *e*, we have to generate a protected method that implements the semantics of the valuation formulas for service *e*. The body of this method has to be built generating the following:

1. A conditional sentence that checks whether the object is in the state Φ.
2. A set of assignment sentences that changes the object attributes in the way that the formula Φ' specifies.

The valuations [10] specified in the FM for the classes `Seller` and `Manager` of the example of Fig. 5 are the following (OASIS notation):

```
valuations {Seller class}
   [pay]salary: =1000;
   {the Seller earns 1000}
   [sell]sales := sales + 1;
   {when sell occurs, sales is incremented of 1 unit}
valuations {Manager class}
   [pay]salary: =3000;
   {the Manager earns 3000}
   [assign]n_projects:=n_projects + 1;
   {when assign occurs, n_projects is incremented of 1 unit}
```

The implementation of the valuations specified above will be the following:

For the `Seller` class:

```
protected void do_pay()
{
    salary=1000;
};
protected void do_sell()
{
    sales=sales+1;
};
```

For the `Manager` class:

```
protected void do_pay()
{
    salary=3000;
};
protected void do_assign()
{
```

[10] The ":=" operator is used in formulas Φ' instead of "=" in order to provide a more intuitive interpretation of its semantics. If a Φ is not specified it is assumed that Φ is true.

```
    n_projects = n_projects + 1;
};
```

The implementation of the corresponding service (pay, sell or assign) is generated according to the effect specified in the FM. Let us suppose that the service sell() has been activated for a Manager object. The effect of this service activation in the attribute sales will be an increment of 1 unit.

4.3.4. Integrity constraints checking in the new state

A `check_integrity_constraints` method has to verify whether the valuation fulfillment leads the object to a valid state. The generation process defines a protected method `check_integrity _constraints` with no arguments. The body of this method must be generated according to the following steps:

For each integrity constraint formula Φ, we have to generate the following:

1. A conditional structure that checks whether Φ holds.
2. An exception sentence for showing an error message when the integrity constraint has been violated.

Integrity constraints redefined in a dynamic subclass will be more constrained than those specified in the parent class. So the `check_integrity_constraints` method that has to be executed will be obtained according to the following conditions:

- If the superclass has specified integrity constraints and the subclass also has specified new emergent integrity constraints, then the `check_integrity_constraints` method of the superclass must be executed. Next, the `check_integrity_constraints` method of the subclass must be executed through delegation to the State Object.
- If the subclass has redefined the integrity constraints of the superclass then the `check_integrity_constraints` method of the subclass must be executed.
- If the subclass has not redefined the integrity constraints of the superclass then the `check_integrity_constraints` method of the superclass must be executed.

In the example of Fig. 5, we have specified the integrity constraint: `salary > 500` in the `Employee` class. This constraint has been inherited by the `Seller` class, but the `Manager` class has redefined it (the second case) in the following way: `salary > 2000`. There exists behavioral compatibility because the fulfillment of the constraint in the subclass logically implies the fulfillment of the constraint in the superclass. Let us see how the `check_integrity_constraints` method is implemented in both classes (`Employee` and `Manager`).

In the `Employee` class:

```
protected void check_integrity_constraints()
throws IntegrityViolation {
{
  if (StateObject.class_name().equals(‘‘Manager’’))
  // the object is specialized in the Manager subclass
      StateObject.check_integrity_constraints();
  else if (!(StateObject.get_salary() > 500))
  // the object is specialized in the Seller subclass
          throw new IntegrityViolation();
};
```

In the Manager class:

```
protected void check_integrity_constraints()
throws IntegrityViolation {
{
  if (!(salary > 2000)) throw new IntegrityViolation();
  // check the integrity constraint in the subclass
};
```

4.3.5. Testing trigger relationships

The effect of a triggered action has traditionally been a singular research topic in the database community, specially in the context of active DBMS. A common proposal of a knowledge model for active systems has been the event–condition–action (ECA) rules mechanism. These rules are composed of an event that triggers the rule, a condition that describes a specific situation, and an action to be performed if the condition is satisfied. In this context, various proposals have been presented to include this rule mechanism in active systems.

For our purposes, we only consider a subset of ECA rules. We define an OASIS trigger as an active rule whose dynamic logic representation is specified as $\Phi[\neg a]\mathit{false}$. This formula means that if Φ holds then the activation of service a is mandatory. If Φ holds and a service other than a occurs, the system remains in a blocked state (this forces a to occur). Thus, triggers are services which are activated when the condition stated in Φ holds. In the OASIS approach, there is an active rule where the event is the current service that is being activated (the one that has produced the change of the object state which makes the formula Φ true), the condition is the wff (Φ) that must be evaluated, and the action is the OASIS service that must be activated if the condition is satisfied.

Taking into account these limitations, we propose a simple strategy of implementation. A `check_triggers` method has to check whether the condition associated to the trigger holds; in this case, the corresponding service is activated. The generation process defines a protected method `check_triggers` with no arguments. The body of this method must be generated according to the following steps.

For each trigger formula we have to generate:

1. A conditional structure that checks whether Φ (triggering condition) holds.
2. A sentence that calls service a when Φ holds. This sentence has to be prepared to throw an exception if the service a is not available (because the object server does not exist, or the precondition attached to a does not hold).

Subclasses in the partition could redefine the triggers declared in the parent class. Thus, the `check_triggers` method that has to be executed will be obtained according to the following conditions:

- If the superclass has specified triggers and the subclass also has specified new emergent triggers, then the `check_triggers` method of the superclass must be executed. Next, the `check_triggers` method of the subclass must be executed through delegation to the State Object.
- If the subclass has redefined the triggers of the superclass, then the `check_triggers` method of the subclass must be executed.

- If the subclass has not redefined the integrity constraints of the superclass, then the `check_triggers` method of the superclass must be executed.

In the example introduced in Fig. 5, we have specified the following trigger condition (a new one, not redefined) for the `Seller` class: `promote if sales > 100`. Let us see how the `check_triggers` method will be implemented in the `Seller` class:

```
  protected void check_triggers()
  {
    if (sales > 100) promote();
    // if the trigger condition holds
    // the promote event must be activated
};
```

4.4. Instance creation and destruction. The migration process

Instance creation and destruction in a dynamic partition will be determined by the migration process defined on the migration actions or by attribute values.

4.4.1. Instance creation and destruction based on migration events

The implementation of the *migration process* will be done by adapting a variation of the *State* pattern called *Owner-Driven Transitions* [8]. This variation is used to implement a finite state machine through state objects. The *Owner-Driven Transitions* pattern proposes that the Context class must be the one that *starts* and *controls* the transition between states. Thus, the class C_P will have a set of *migration methods* M_{mig} that will implement the migration events. The information specified in the migration process will be used by the migration methods to create and destroy the appropriate state objects for simulating the migration between subclasses. Below, we can see the migration process (in OASIS) obtained from the migration diagram in Fig. 6.

```
  Seller, Manager
  dynamic specialization of Employee
  migration relation is
    Employee = create.Seller;
    Seller = promote.Manager;
    Manager = demote.Seller;
```

Semantics of the migration events. There exist two types of migration events in the migration process:

- *Creation*. It is the initial event of the migration process. This event will create an object belonging to the initial subclass of the dynamic partition, and it will initialize its constant attributes. At the OASIS specification level, it will be a constructor event of the superclass. In the example of Fig. 5 `create` will be the creation event of the `Employee` class.
- *Liberators/Carriers*. Events that cause the migration between two subclasses of the dynamic partition. At the specification level these events are declared in the subclass that is the origin of the migration due to event activation. Events of this will change the state of the active object and then they will liberate the part of its state that belongs to the active subclass in the partition. These events act as carriers (or special constructors) that "carry" the object to the target

subclass in the migration process. They will create the part of the object state that belongs to the new active subclass in the partition (initializing its constant attributes). In the example of Fig. 6, `promote` and `demote` are *liberators/carriers* events that have been specified in the `Seller` and `Manager` classes, respectively.

Implementing the migration events. Migration events specified in the migration process are a subset of class events. In this way, they must also be implemented by methods (called migration methods) that follow the execution strategy presented above (like the other specified class events). Moreover, due to their migrational semantics, migration methods will be responsible for implementing the migration between subclasses. In order to implement the migrational semantics of migration methods, a special kind of private method (one for each dynamic partition) will be included in the implementation of C_P class. This method will be a *specializer* method that will create and destroy the appropriate state objects to simulate the migration between subclasses. The specializer method will have the name of the event as an argument in order to properly determine the migration between subclasses.

The type of the migration method will determine the implementation of the specializer method and its application in the execution strategy. Let us see the two kinds of migration methods detected in the previous section:

- *Creation methods.* The first time an object of class C_P is created, an object of a class C_i $i \in \{1, \ldots, n\}$ must be created, such that S_i is the *initial class* in the migration process. The object of subclass S_i (`StateObject`) is assigned to attribute A_{C_A}. This is the semantics that the specializer method must implement when it receives a creation event as an input. In the implementation of the State pattern, the Context class C_P will have a constructor that implements the execution strategy initializing C_P attributes. At the end of the execution strategy algorithm we will place a call to the specializer method with the creation event as an argument. Let us look at the implementation of the `Employee create` method and the application of the specializer method `specialize_by_action`:

```
public void create(String var_name)
{
  check_state_transition(``create'');
  check_preconditions(``create'');
  do_create(var_name);
  // valuation that initializes Employee attributes
  check_integrity_constraints();
  check_triggers();
  specialize_by_action(``create'');
  //specializer method
};
```

- *Liberators/Carriers methods.* These methods will have the responsibility of destroying [11] the active State Object. Next, they have to create a new object of the target subclass in the migration

[11] In the Java language there is no need to define a destructor method. In our approach, we need a special kind of destructor to release any used resources (for example in a three-tier implementation, the application has to delete the object in the database).

process.This object will be assigned to attribute A_{C_A} (the State Object). This is the semantics that the specializer method must implement when it receives a liberator/carrier event as an input. In the implementation of the State pattern, the Context class C_P will have a method that delegates its execution to an active subclass method because this kind of methods belong to one of the subclasses in the partition. We will place a call to the specializer method with the liberator/carrier event as an argument after the delegated method calling. Let us look at the implementation of the `Employee` `promote` and `demote` methods, and the application of the specializer method `specialize_by_action`:

```
public void promote()throws EventNotAllowed
{
  if (StateObject.class_name.equals(‘‘Seller’’)
  {
   // if the active object is a Seller
   StateObject.promote();
   // delegates the execution of promote event
   // in subclass (following the execution strategy)
   specialize_by_action(‘‘promote’’);
   // specializer method
  }
  else throw new EventNotAllowed();
  // promote event belongs to subclass Seller
};
public void demote() throws EventNotAllowed
{
  if (StateObject.class_name.equals(‘‘Manager’’))
  {
   // if the active object is a Manager
   StateObject.demote();
   // delegates the execution of demote event
   // in subclass (following the execution strategy)
   specialize_by_action(‘‘demote’’);
   // specializer method
  }
  else throw new EventNotAllowed();
  // demote event belongs to subclass Manager
};
```

Taking into account the semantics defined by creation and liberator/carrier methods, the implementation of the specializer method of the example shown in Fig. 5 will be as follows:

```
private specialize_by_action(String Action)
{
  if (StateObject.class_name.equals(NO_CLASS)
  {
```

```
  // there is no active object because
  // we are in the creation process of an Employee
    if (Action.equals(‘‘create’’))
    {
     // if create event occurs
     StateObject = new Seller();
     // creates a Seller amd assigns it to StateObject
     return;
    }
    return;
  }
  if (StateObject.class_name.equals(‘‘Seller’’)
  {
    if (Action.equals(‘‘promote’’))
    {
     // if promote event occurs
     StateObject.free();
     // destroys active Object
     StateObject= new Manager();
     // creates an object of Manager subclass
     return;
    }
    return;
  }
  if (StateObject.class_name.equals(‘‘Manager’’)
  {
    if (Action.equals(‘‘demote’’))
    {
     // if demote event occurs
     StateObject.free();
     // destroys active Object
     StateObject = new Seller();
     // creates an object of Seller subclass
     return;
    }
    return;
  }
};
```

In this implementation of the specializer method, there is no need to include the sentences that check the class of the State Object because, in the example of Fig. 6, only one possible transition and one target subclass for that transition exist. However, these sentences will be necessary when exist two or more transitions labeled with different events that have two or more target subclasses.

4.4.2. Instance creation and destruction based on attribute values

The implementation of the migration process based on *attribute values* is achieved in a similar way to the proposed in the previous section. A specializer method has to be introduced in order to implement the migrational semantics of migration conditions defined on variable attributes. This method will be implemented in C_P class as a private method. It will create and destroy the appropriate state objects depending on the class of the active State Object and its attribute values. The specializer method has to be placed at the end of the execution strategy algorithm of the following methods:

- the constructor method of C_P class and
- those methods of C_P that can modify the value of the attributes that take part in the migration conditions. These are what we call *possible migration* methods.

Below, we can see the specification in OASIS of the migration conditions obtained from the class diagram in Fig. 7.

```
Seller where {salary < 2000}
Manager where {salary >= 2000}
  dynamic specialization of Employee;
```

Taking into account this specification, the implementation of the specializer method of the example shown in Fig. 7 will be as follows:

```
private specialize_by_attribute()
{
  if (StateObject.class_name.equals(‘‘Seller’’)
  {
    // the active object is a Seller
    if (salary >= 2000)
    {
     StateObject.free();
     // destroys active Object (a Seller)
     StateObject = new Manager();
     // creates an object of Manager subclass
     return;
    }
    return;
  }
  if (StateObject.class_name.equals(‘‘Manager’’)
  {
    // the active object is a Manager
    if (salary < 2000)
    {
     StateObject.free();
     // destroys active Object (a Manager)
     StateObject = new Seller();
     //creates an object of Seller class
     return;
```

```
      }
      return;
    }
    if (StateObject.class_name.equals(NO_CLASS)
    {
      // there is no active object because
      // we are in the creation process of an Employee
      if (salary >= 2000)
      {
       StateObject= new Manager();
       // creates an object of Manager subclass
       return;
      }
      StateObject = new Seller();
      // creates an object of Seller subclass
      return;
    }
  };
```

Let us look at the implementation of the `Employee` `create` method and the application of the specializer method `specialize_by_attribute`:

```
  public void create(String var_name)
  {
    check_state_transition(‘‘create’’);
    check_preconditions(‘‘create’’);
    do_create(var_name);
    // valuation that initializes Employee attributes
    check_integrity_constraints();
    check_triggers();
    specialize_by_attribute();
    // specializer method
  };
```

The method `pay` of `Employee` class changes the value of the `salary` attribute. This a possible migration method. Let us look at its implementation and the application of the specializer method `specialize_by_attribute`:

```
  public void pay()
  {
    StateObject.pay();
    // delegates on StateObject
    specialize_by_attribute();
    // specializer method
  };
```

5. Related work

This section has been divided in two subsections according to the nature of the work presented in this paper. Our approach provides a solution in two research areas (model-based code generation and dynamic specialization modeling) presenting a *code generation process* for *dynamic specialization models*. Firstly, some of the existing proposals on model-based code generation are reviewed, including those based on design patterns. The approaches introduced in this subsection provide solutions that do not use expressive and precise specialization models. Most of them present solutions that generate code for static specializations which are not rich enough to tackle certain modeling situations. In the second subsection several modeling approaches that include the dynamic specialization abstraction are analyzed. These approaches provide partial solutions because some of them introduce solutions in the conceptual modeling phase (providing methodological support and/or precisely defining the semantics of the dynamic specialization abstraction using formal techniques) and others at the programming level (introducing new contructs into programming languages) without giving support to the conceptual modeling phase. The approach presented in this paper improves both approaches because it provides a complete automated development process that deals with dynamic specialization at the conceptual level (providing a precise conceptual construct), at design level (providing a set of quality design structures that represent the dynamic specialization) and at implementation level (completely generating code in industrial programming languages, including structure and behavior).

5.1. Model-based code generation

There exist three approaches in the model-based code generation area [4]: *structural* (generates code from structural models), *behavioral* (generates code from dynamic models) and *translational* (generates complete code from conceptual models). Our proposal can be placed in the translational category.

An interesting approach in this area is OBLOG CASE [21], a tool based on the OBLOG [21] formal specification language. Its code generation process is based on rule rewriting. The rules are written in a scripting language that the designer must be expert in. Apart from the difference in expressiveness between OBLOG and OASIS languages, the use of rules makes the OBLOG approach distinct to ours. The designers develop their own code generation process using rules. An additional problem is that those rules embed architectural and implementation aspects jointly. Another translational approach is the *Recursive Design* proposed by Shlaer and Mellor [32]. It is supported by the BridgePoint tool. It does not use a formal specification language. This method models the domain and the architecture of the application using the same notation. The domain and the architecture models are used to develop patterns of a special kind called ‘‘archetypes’’ (a kind of macro) that the designer must specify by using a scripting language (it could be the target programming language). As we have seen in OBLOG, the code generation process has to be written by the designers using scripting rules. In order to generate code, these rules will apply the archetypes to conceptual schemes stored in a repository.

Both approaches are opened to the designer, and due to this feature, the quality of the code generated depends on the designer. Both are flexible approaches. However, the tools cannot guarantee that the software solution appropriately represents the Conceptual Model. The OO-Method approach is closer to the idea of a universal code generator that has a closed code generation process. The key feature of the OO-Method is the well-defined software representation of a predefined, finite catalog of conceptual modeling constructs that assures a software representation which is functionally equivalent to the conceptual schema (as we have seen in detail in the case of dynamic specialization). These two approaches provide solutions that do not use expressive and precise specialization models (mainly static specialization models). In our approach, we only have to focus on the Conceptual Modeling phase because the other phases of the software production process are achieved automatically.

5.1.1. Code generation based on design patterns

There are some approaches to automatic code generation that are based on design patterns. The Model Based Object Oriented Software generation Environment (MOOSE) environment [2] and its component Pattern Based Simulator Generator (PSiGene) [13] generate code in narrow and well-defined domains. The MOOSE approach proposes a domain-specific software development method based on generators that gives support to reuse. This method uses different models (*base*, *component*, *glue*, *application* and *features* models) and notations in the conceptual modeling phase. It uses a library of primitives that depends on the domain problem, an architecture model, and a set of heuristics for selecting the appropriate design structures. This approach automatically generates class templates with constructors, destructors and access methods. In order to obtain a completely functional application, the code generated has to be manually modified to include part of the application behavior. MOOSE improves structural approaches but does not provide 100% code generation. In PSiGene, the designer must specify domain-specific patterns in a formal language. These patterns are stored in a pattern repository, and they are used to generate code. During the code generation process, these patterns are instantiated using the information included in the conceptual model. This approach can generate 100% code in very limited domains. MOOSE generates C++ and Visual Works code, and PSiGene only generates Visual Works. There are some differences with our approach: (1) the patterns used in PSiGene are not standard patterns (we use standard patterns) and they are completely domain-dependent (our patterns are domain independent), (2) they provide *component generators* in very specific domains, and our approach can be seen as a *universal* code generator which is oriented towards producing business applications.

Budinsky et al. [5] presents an automatic code generation process from design patterns. This approach helps the designer to instantiate design patterns with application domain specific information, but it does not provide a complete code generation because the resulting code fragments have to be manually adapted to the software system.

In the area of behavioral code generation, Statelator [3] combines the model transformation discipline with the use of design patterns. This approach represents a UML statechart through a microarchitecture based on the State design pattern. The authors introduce a generic algorithm to transform a statechart into a design structure. This algorithm specifies the transformation rules using OCL. [12] This proposal can be adapted to our approach in order to generate the STD of a

[12] Object-Constraint Language in the UML notation.

class. However, this is not a complete solution compared wtih our approach because it generates only the class behavior based on its statechart.

5.2. Dynamic specialization

In this section, we are going to review some approaches that present dynamic specialization models which are similar to the one presented in this work. These approaches are analyzed from the modeling and implementation point of view.

Syntropy [7] is a semi-formal object-oriented method. It introduces the *state-types* concept to model *dynamic classifications* as in our approach. A dynamic subclass will always contain objects of the superclass that were in a specific state of the statechart which defines the behavior of the superclass. In this way, a state of the behavior specification of the parent class (its statechart) represents the condition of belonging to one dynamic subclass. This approach is similar to ours because we introduce a state transition diagram called migration diagram (one for each partition) to model the dynamic subclasses and the possible transition between subclasses. The main difference is that Syntropy does not provide any guides to automate the code generation from its models.

The Wieringa et al. [40] approach constitutes the basis of the taxonomic relationships that are present in the OASIS formal model. The most remarkable aspect of this proposal is the distinction between roles and dynamic subclasses. This difference is based on the counting problem (it arises due to the necessity of modeling an object that can play several roles of the same role class). Roles inherit from the player object through a delegation mechanism and dynamic subclasses inherit from superclasses through a standard inheritance mechanism. This approach includes methodological and theoretical aspects, and it is considered to be one of most interesting approaches in the formal study of the taxonomic relationships. It deals with the precise specification of subclasses in a order sorted dynamic logic, but does not propose any solution in the implementation field as we present in our approach.

Snoeck and Dedene [33] distinguish between specializations and roles. This approach analyzes some ways a class can be partitioned into subclasses (attribute-defined, existence-defined and state-defined subclass). The state defined specialization is similar to our dynamic specialization. The formal object model that introduces this proposal is based on the process algebra that uses the MERODE formal language. This work treats the extension of subclass processes with a high level of detail. It is mainly based on the formal characterization of subclass behavior by process algebras and does not provide implementation techniques in imperative languages. This is once again its weak point when compared to our work.

TROLL [14] is a formal specification language that introduces roles and specializations to specify dynamic and static aspects of subclass objects. The role concept describes a dynamic specialization but not in the sense of our proposal, because this approach does not introduce the partition concept and does not provide any way to specify the migration between objects of the role subclasses. TROLL introduces TBench, a prototyping environment that allows for generating an independently executable prototype from a graphical conceptual specification called OM-TROLL. The prototype generated is a C++ program that includes the static/dynamic aspects of the system. The main difference is that TBench focuses on the validation of specifications and their model checking, while the OO-Method provides an implementation that is centered on obtaining a software product which is ready for execution.

Olivé et al. [22] define a temporal framework which allows for the categorization of the *IsA* relationships depending on the nature of the class and its subclasses. The authors define three evolution constraints on the specializations. These constraints permit us to determine the possible evolution of the population of a subclass with respect to its superclass population. The *relative static* constraint defines the object evolution constraints which are present in our dynamic specialization approach. *Dynamic* constraints are less restrictive than *static relative*. It is important to note at a methodological level that this proposal uses pure partitions (as in our approach). Moreover, it introduces the concept of *type transitions* in the relative static and dynamic partitions. This concept is used to specify the order in which objects change their current subclass. The admissible transitions are defined using state transition diagrams like we do in our migration diagram. Finally, this approach identifies the kind of transitions that can exist between subclasses. Those transitions can be: *reclassification*, *active continuation*, *suspension*, *suspension continuation* and *reactivation*. This work provides a methodological solution to the modeling of dynamic specializations and could be incorporated to our approach as a higher level guide to specify dynamic specialization. Our work improves this approach including the treatment of the structural and behavioral relationships between a class and its subclasses, and by providing a complete software generation process to implement dynamic specializations.

Su [35] defines object *migration constraints* as sets of *dynamic subclass sequences*, where each sequence represents a trace of an object through the subspaces of its state space defined by the dynamic class partitions of the model. Su's results can be applied to the study of migration diagrams for dynamic subclasses.

There exist some proposals that deal with the dynamic specialization at a lower level of abstraction. These proposals belong to the *conceptual programming* [16] area. Programming in this area is seen "*... as a modeling process where abstractions are expressed as programming language constructs*". There are two interesting proposals in this area that introduce the dynamic specialization concept: Albano et al. [1] introduce the dynamic class concept under the name of roles in their language Fibonacci (a programming language for Object Databases), and Taivalsaari [36] defines the mode concept. In this approach, each class can have several modes depending on which it will react differently to incoming messages. A mode can provide mode-specific operations which are not defined for the class whose mode it is. There are two ways to change mode: (1) one is by means of explicit mode transition events (as in our approach) and (2) the other by means of implicit mode transitions, which may be triggered as a side-effect of other operations (for instance due to the change of attribute values). Modes are clearly equivalent to our dynamic subclasses. Both approaches propose solutions at the programming level but do not provide methodological guidance at the conceptual level.

Many times in the literature the dynamic specialization term is used as the term role and vice versa. Besides the presented proposals, there exist other approaches that introduce the role concept in a broader sense (giving to this term the semantics of the role and the dynamic subclasses). These approaches consider the dynamic specialization as a particular interpretation of the role concept. In the formal modeling area, there exist several approaches such as Pernici [26], LODWICK [34], ADOME [19], MOSES [27], DOOR [41], and Sciore [31]. In the conceptual programming area Kristensen and Osterbye in [15,17] introduce the role concept as a new construction of the BETA and SMALLTALK languages, Gottlob et al. [11] present the role concept as a SMALLTALK extension, and Richardson and Schwartz [28] introduce a new language construct called *aspect*. In these approaches, the role concept is used to model behaviors (aspects) of an object in a separate way. The instances of a class can play multiple roles (and also multiple roles of the same class at the same time). This feature implies the necessity of a special treatment with regard to the identification problem and to the way a role inherits from its player. Most approaches propose that the role object shares its identifier with the player object (not in our approach, where the subclass object has the same identity as its parent class), and that the object role inherits its properties from its player through delegation (we use standard inheritance mechanisms at the specification level). This feature is not present in our dynamic specialization model; however, in the OO-Method the role concept is included as a distinct conceptual pattern.

6. Conclusions and further work

In this work, we have introduced a complete code generation process of conceptual patterns. The dynamic specialization conceptual pattern of the OASIS formal object model has been used to show this process.

We have incorporated design patterns into the OO-Method providing a framework which offers methodological guidance to go from the problem space to the solution space. The methodological framework developed in this paper is based on:

- The precise description of conceptual patterns by using formal languages.
- Design pattern specialization to properly represent the conceptual patterns used in the conceptual modeling phase.
- The definition of a set of precise mappings between specialized design patterns and conceptual patterns. These mappings are defined in a way that preserves the semantics of conceptual patterns.
- The introduction of an execution strategy that is used to implement the behavior of conceptual patterns.

The ideas presented provide the basis for a code generation process which is capable of automating the conceptual pattern translation. It can be adopted by existing software production methods in order to improve the quality and the correctness of the software produced. These ideas are being applied in a CASE tool with full code generation capabilities [23] that gives support to the OO-Method. In our approach, the developer only have to focus on the Conceptual Modeling phase because the implementation is produced automatically. If developers need to extend any application, they only have to modify the conceptual model instead of the code, and then "compile" that conceptual model with the code generator.

Research work is still underway to extend this approach to all the abstraction mechanisms that take part in the OO-Method conceptual modeling phase. Design patterns are being studied and specialized in order to implement conceptual patterns such as roles, static specializations, dynamic and static aggregations, associations and other conceptual patterns which are present in the OO-Method/OASIS approach. The work that is now being carried out on the graphical notation is being developed in a way that could be considered as an extension of the UML notation. This extension will enrich the UML notation with the conceptual patterns that OO-Method provides.

In this way, analysts that use UML can be provided with a powerful notation that is expressive enough to produce quality software in an automated way.

References

[1] A. Albano, R. Bergamini, G. Ghelli, R. Orsini, An object data model with roles, in: D. Bell, R. Agrawal, S. Baker (Eds.), 19th International Conference on Very Large Databases, Morgan Kaufmann, Los Altos, CA, 1993, pp. 39–51.

[2] J. Altmeyer, J.P. Riegel, B. Schuermann, M. Schuetze, G. Zimmermann, Application of a generator-based software development method supporting model reuse, in: Proceedings of the 9th International Conference in Advanced Information Systems Engineering, CaiSE97, Barcelona, Catalonia, Spain, Lecture Notes in Computer Science, vol. 1250, Springer, Berlin, June, 1997, pp. 159–171, ISBN 3-540-63107-0.

[3] T. Behrens, S. Richards, StateLator – behavioral code generation as an instance of a model transformation, in: L. Bergman, B. Wangler (Eds.), International Conference on Advanced Information Systems Engineering, CAiSE 2000, Lecture Notes in Computer Science, vol. 1789, Springer, Berlin, 2000, pp. 401–416.

[4] R. Bell, Code Generation from Object Models, Embedded Systems Programming, March 1998. Available from: http://www.embedded.com/98/9803fe3.html.

[5] F.J. Budinsky, M.A. Finnie, J.M. Vlissides, P.S. Yu, Automatic code generation from design patterns, IBM Syst. J. 35 (2) (1996).

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture. A System of Patterns, Wiley, New York, 1996.

[7] S. Cook, J. Daniels, Designing Objects Systems. Object-Oriented Modelling with Syntropy, Prentice-Hall, New York, 1994.

[8] P. Dyson, B. Anderson, Pattern Languages of Program Design 3, Chapter State Patterns, Software Patterns, Addison-Wesley, Reading, MA, 1998, pp. 125–142 (Robert Martin, Dirk Riehle and Frank Buschmann edition).

[9] M. Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley, Reading, MA, 1997.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, in: Design Patterns: Elements of Reusable Object-oriented Software, Professional Computing Series, Addison-Wesley, Reading, MA, 1994.

[11] G. Gottlob, M. Schrefl, B. Röck, Extending object-oriented systems with roles, ACM Trans. Inf. Syst. 14 (3) (1996) 268–296.

[12] Object Management Group, Unified Modeling Language Specification Version 1.4 draft, Technical Report, February 2001 (Version 1.3, June, 1999).

[13] F. Heister, J.P. Riegel, M. Schuetze, S. Schultz, G. Zimmermann, Pattern-based code generation for well-defined application domains, Technical Report, Computer Science Department, University of Kaiserslautern, 1998.

[14] R. Jungclaus, G. Saake, T. Hartmann, C. Sernadas, TROLL – A language for object-oriented specification of information systems, ACM Trans. Inf. Syst. 14 (2) (1996) 175–211.

[15] B.B. Kristensen, Object-oriented modeling with roles, in: B. Stone, J. Murphy (Eds.), OOIS'95: Proceedings of the International Conference on Object-Oriented Information Systems, Springer, Berlin, 1995, pp. 57–71.

[16] B.B. Kristensen, K. Osterbye, Conceptual modeling and programming languages, SIGPLAN Notices 29 (9) (1994).

[17] B.B. Kristensen, K. Osterbye, Roles: Conceptual abstraction theory and practical language issues, Theory Practice Object Syst. 2 (3) (1996) 143–160.

[18] P. Letelier, P. Sánchez, I. Ramos, O. Pastor, OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objetos, Servicio de Publicaciones, Universidad Politécnica de Valencia, Valencia, España, 1998, SPUPV-98.4011, ISBN 84-7721-663-0.

[19] Q. Li, F.H. Lochovsky, ADOME: An advanced object modeling environment, IEEE Trans. Knowledge Data Eng. 10 (2) (1998) 255–276.

[20] J.J.Ch. Meyer, A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic, Notre Dame J. Formal Logic 29 (1998) 109–136.

[21] OBLOG Software S.A. The OBLOG software development approach, Technical Report, OBLOG Software S.A., 1999.

[22] A. Olivé, M.R. Sancho, D. Costal, Entity evolution in IsA hierarchies, in: 18th International Conference on Conceptual Modeling, Lecture Notes in Computer Science, Springer, Berlin, 1999, pp. 62–80.
[23] O. Pastor, E. Insfrán, V. Pelechano, J. Romero, J. Merseguer, OO-Method: An OO software production environment combining conventional and formal methods, in: 9th Conference on Advanced Information Systems Engineering (CAiSE'97), Barcelona, Spain, Lecture Notes in Computer Science, vol. 1250, Springer, Berlin, June, 1997, pp. 149–159, ISBN 3-540-63107-0.
[24] O. Pastor, V. Pelechano, E. Insfrán, J. Gómez, From object oriented conceptual modeling to automated programming in Java, in: 17th International Conference on Conceptual Modeling (ER'98), Lecture Notes in Computer Science, vol. 1507, Springer, Singapore, 1998, pp. 183–196, ISBN 3-540-65189-6.
[25] O. Pastor, I. Ramos, OASIS version 2 (2.2): A Class-Definition Language to Model Information Systems, Servicio de Publicaciones, Universidad Politécnica de Valencia, Valencia, España, 1995, SPUPV-95.788.
[26] B. Pernici, Objects with roles, in: IEEE/ACM Conference on Office Information Systems, Cambridge, MA, 1990.
[27] D.W. Renouf, B. Henderson-Sellers, Incorporating roles into MOSES, TOOLS 15, Melbourne, Australia, 1996.
[28] J. Richardson, P. Schwartz, Aspects: Extending objects to support multiple, independent roles, SIGMOD Record 20 (2) (1991) 298–307.
[29] J. Rumbaugh, M. Blaha, W. Permerlani, F. Eddy, W. Lorensen, Object Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, 1991.
[30] G. Saake, P. Hartel, R. Jungclaus, R. Wieringa, R. Feenstra, Inheritance Conditions for Object Life Cycle Diagrams, EMISA Workshop, 1994.
[31] E. Sciore, Object specialization, ACM Trans. Inf. Syst. 7 (2) (1989) 103–122.
[32] S. Shlaer, S.J. Mellor, Recursive design of an application-independent architecture, IEEE Software (January) (1997).
[33] M. Snoeck, G. Dedene, Generalization/specialization and role in object oriented conceptual modeling, Data Knowledge Eng. 12 (2) (1996) 171–195.
[34] F. Steimann, On the representation of roles in object-oriented and conceptual modeling, Data Knowledge Eng. 35 (October) (2000) 83–106.
[35] J. Su, Dynamic constraints and object migration, in: G.M. Lohman, A. Sernadas, R. Camps (Eds.), Proceedings of the 17th International Conference on Very Large Databases, VLDB Endowment Press, 1991, pp. 233–242.
[36] A. Taivalsaari, Object-oriented programming with modes, J. Object-Oriented Programming 6 (3) (1993) 25–32.
[37] W. Tepfenhart, J. Cusick, A unified object topology, IEEE Software (May/June) (1997) 31–35.
[38] P. Wegner, B. Zdonik, Inheritance as an incremental modification mechanism or what like is and isn't like, in: S. Gjessing (Ed.), ECOOP'88: European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, vol. 276, Springer, Berlin, 1988, pp. 55–77.
[39] R. Wieringa, Algebraic foundations for dynamic conceptual models, Ph.D. thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, May 1990.
[40] R. Wieringa, W. Jonge, P. Spruit, Using dynamic classes and role classes to model object migration, Theory Practice Object Syst. 1 (1) (1995) 61–83.
[41] R.K. Wong, H.L. Chau, F.H. Lochovsky, Dynamic knowledge representation in DOOR, in: N. Pissinou, K. Makki, X. Wu, J. Tsai (Eds.), Proceedings of the 1997 IEEE Knowledge and Data Engineering Exchange Workshop, Los Alamitos, IEEE Computer Society, Silver Spring, MD, 1997, pp. 89–96.

Vicente Pelechano is an Associate Professor in the Department of Information Systems and Computation (DISC) at the Valencia University of Technology, Spain. His research interests are object orientation, conceptual modeling, requirements engineering, software patterns and model-based code generation. He received his Ph.D. degree from the Valencia University of Technology in 2001. He is currently teaching software engineering and component-based software development in the Valencia University of Technology. He is a member of the Logic Programming and Software Engineering Research Group at the DISC, member of the ACM and of the IEEE Computer Society.

Oscar Pastor is currently the Head of the Computation and Information Systems Department at the Valencia University of Technology (Spain), and the leader of the Research Group on Object-Oriented Methods for Software Production in the same department. He received his Ph.D. degree from the Valencia University of Technology in 1992, after a research stay in HP Labs, Bristol, UK. He is currently Professor of Software Engineering at the Valencia University of Technology. His research activities has been involved with object-oriented conceptual modelling, requirements engineering, information systems, web-oriented software technology and model-based code generation. Author of over 100 research papers in conference proceedings, journals and books, he has received numerous research grants from public institutions and private industry, and devoted considerable effort to issues of technology transfer from academia to industry.

Emilio Insfrán is an Assistant Professor in the Department of Information Systems and Computation (DISC) at the Valencia University of Technology, Spain. His research interests are OO methodologies, conceptual modeling, requirements engineering, specification languages and databases. He received a degree in Computer Science from the National University of Asunción, Paraguay, a MS degree in Computer Science from the Cantabria University, Spain, and spent 1999 as a visiting research scientist at University of Twente, the Netherlands. He is a member of the Logic Programming and Software Engineering Research Group at DISC and of IEEE Computer Society.