COMBINING FORMAL SPECIFICATIONS WITH DESIGN BY CONTRACT [1]

Authors: Begoña Moros Valle, Joaquín Nicolás Ros, Jesús García Molina,
José Ambrosio Toval Álvarez

Software Engineering Research Group. Department of Computing, Languages and Computer Systems.
University of Murcia.
e-mail: {bmoros | jnicolas | molina | atoval }@dif.um.es; http://www.um.es/~giisw
tlf: 34-968-36 46 42; fax: 34-968-36 41 51.

ABSTRACT

In this paper, we present an approach to the object-oriented software development which is based on: i) automatic generation of a throwaway prototype from the initial specification in a formal, declarative, object-oriented specification language, ii) validation of user requirements and refinement of the specification by using this prototype, and iii) automatic translation from the validated specification types to programming classes including the semantics of the formal specification by means of assertions. The last step is achieved by using an object-oriented implementation language supporting Eiffel-like assertions and the “Design by Contract” technique; therefore, these classes force the first evolutionary prototype (that will evolve to the final software) to be formally consistent with the validated specification. This approach is supported by a high level CARE (Computer-Aided Requirements Engineering) tool.

1. INTRODUCTION

Requirements elicitation is a crucial task in the software construction process with implications in correctness and productivity. Formal specification languages allow to express the captured requirements in an accurate and rigorous way; in contrast to non-formal notations, validation and verification techniques are possible by formal reasoning. On the other hand, the creation of prototypes from the requirement specification is considered a suitable technique for checking how much the final user development process.

In that sense, we present an approach to the object-oriented software development supported by a CARE (*Computer-Aided Requirements Engineering*) prototyping environment for a formal, declarative and object-oriented language called OASIS [Pastor95] (although the approach could be adapted to any similar language, i.e. Object-Z, TROLL, etc.). This environment allows to model the considered system by an OASIS specification and later to generate a throwaway prototype for validating, being functionally equivalent to the specification. Once the validation process has concluded, we intend to start the implementation process with the guarantee that it will be consistent with the conceptual model. For this purpose, another prototype is generated – an evolutionary one – also functionally equivalent to the specification, but written in a programming language that includes the assertion mechanism. The consistency between the program and the specifications is ensured by applying the "Design by Contract" technique [Meyer92].

Our approach is in the line with the project recently presented by B. Meyer [Meyer98], trying to obtain a set of trusted components by using several techniques, including "Design by Contract" and formal validation, among others.

The features of the chosen language, OASIS, are very similar to **TROLL** [Jungclaus91] and **LCM** (*Language for Conceptual Modeling*) [Wieringa94]. These both the throwaway one in section 4 and the evolutionary one in section 5. Finally, section 6 comments related works and section 7 presents conclusions and further work.

2. OASIS OBJECT MODEL

Before presenting the prototyping approach we are going to briefly describe the main concepts of the object model of the specification language OASIS. Our purpose is not to explain every feature rigorously, rather we will outline its most relevant aspects by means of an example: a simple bank application for managing accounts, clients and employees. Figure 1 shows the object model expressed by means of a UML class diagram (see [Rational97]):

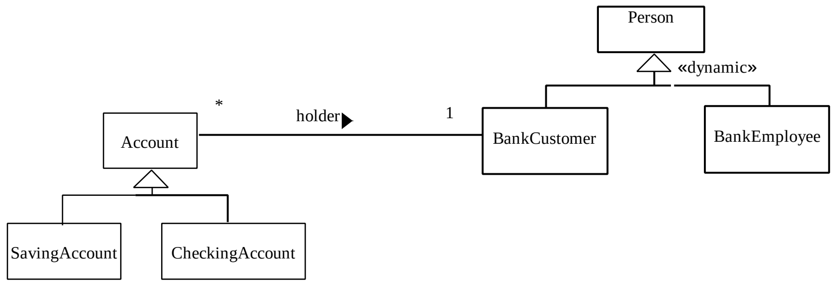


Figure 1. UML class diagram of the running example

An OASIS **class** plays two roles: **intensional**, describing the structure and behavior of the objects belonging to the class (type); and **extensional**, denoting a collection of objects with the same features (population). The properties of an object are represented

- **Derived attributes**: their values are calculated by a derivation formula expressing the computation in terms of other attributes.

An **event** is an atomic instantaneous action that modifies the state of at least one object, that is, one or more attributes are changed in the way that is stated in the **evaluation** part of the specification (see Figure 3). An evaluation is a dynamic logic formula [Harel84] expressed in the form Φ[*e*]Φ’, which means: “ if Φ is true in a given state, every possible execution of the event *e* leads to a situation in which Φ’ is true”. Evaluations are structured as it is shown in Figure 2:

$$\underbrace{\text{red= false}}_{\text{conditions}}\text{ , }\underbrace{\text{credit=C}}_{\text{instantiations}}\text{ [deposit(M)] }\underbrace{\text{credit=C+M}}_{\text{actions}}$$

Figure 2. Structure of the evaluations

An object could be affected by the occurrence of an event if and only if this object is in a certain state. This fact is stated by means of **preconditions**. Preconditions are formulas that must hold to allow a valid event occurrence. Moreover, there are some conditions which any object in a stable state must hold, called **integrity constraints**. An object is in a stable state, before and after the execution of an action on it.

The set of possible relationships between classes are described in contrast to UML [Rational97]:

```
conceptual_schema BankSystem

class Account relation of BankCustomer
(relational, static, non disjoint, flexible, univalued, not null)

var_declarations
          M:nat;
          C:nat;
          D:nat;
constant_attributes
      key code: string;
          type_account:string;
variable_attributes
          credit: nat;
          debit:nat;            #when you are in red
          credit_limit:nat;
derived_attributes
          red: bool;
          maxWithdrawal: nat;
proper_events
          new     open_account.
          destroy close_account.
                  deposit (M).
                  withdraw (M).
                  new_credit_limit (M).
constraint
          type_account='saving' or type_account='checking'.
          debit≤credit_limit.
          BankCustomer.age>18.
valuation
          [new_credit_limit (M)] credit_limit=M.
          red=false , credit=C [deposit (M)] credit= C+M.
          red=true , debit=D [deposit(M)] if (M≤D) then debit=D-M
                                    else debit=0 , credit=M-D endif.
          red=false , credit=C [withdraw(M)] if (C≥M) then credit=C-M
                                    else credit=0 , debit=M-C endif.
          red=true , debit=D [withdraw(M)]debit=D+M.
derivation
          red = (debit>0).
          maxWithdrawal = (credit - debit + credit_limit).
preconditions
          close_account if (credit=0 and debit=0).
          withdraw(M) if (M ≤ maxWithdrawal).
end_class
```

```
class SavingAccount specialization of Account where
                                      type_account='saving'
constant_attributes
          interest_rate : nat;
          ...........
constraint
          red=true.
          credit_limit=0.
          ...........
end_class

class CheckingAccount specialization of Account where
                                      type_account='checking'
constant_attributes
          num_cash_card : nat;
          ...........
end_class

class Person
constant_attributes
          ID: nat;
          name:string;
          birthday:date;
variable_attributes
          address:string;
          ...........
derived_attributes
          age: integer;
          ...........

                    {update attributes events}
end_class


class BankCustomer specialization of Person
                    relation of Account

     ............................

end_class

end_conceptual_schema
```

Figure 3. OASIS running example specification

In the example illustrated in Figure 3, the properties of the *Account* class mean: at the creation of an account object we must associate it with a unique bank object (*relational, not null, univalued*) who will not change over the object life (*static*). A client can be the holder of more than one account (*non disjoint*). He/she could be registered in the bank even if he/she does not have any account (*flexible*).

is able to change its class within the subclass hierarchy and, in addition, the same object can play several roles at the same time. According to the running example (Figure 3) a person could be a bank customer at a given moment, and moreover, he/she could finish playing that role and would be still a person.

* **permanent specialization**: an object belongs to one of the specialized class since its creation instant according to a specialization condition. This fact is shown in the account hierarchy included in Figure 3. An account will be *saving* or *checking* depending on the value given to the constant attribute *type* when the account is opened.

3. DESCRIPTION OF THE PROTOTYPING APPROACH.

The inherent ambiguities of the languages and notations involved in the non-formal analysis/design methods produce specifications on which it is very difficult to rigorously reason. Further, it is not possible to ensure the consistency of the final implementation from the obtained models [France97].

In order to face up to the aforementioned problem, we propose an approach based on the automatic generation of two prototypes. Firstly, a throwaway prototype is created and used to validate the specification, through an environment for editing and animating between the throwaway prototype and the initial requirements of the users, we generate a verified evolutionary prototype. That is, the generation process ensures the consistency of the evolutionary prototype with the prior validated model.

Once the user requirements are collected and organized, the working process with the environment is basically as follows (Figure 4):

1. A system OASIS specification is graphically introduced, and then it is translated automatically into OASIS notation. This specification is named the conceptual schema.
2. Now, we are able to animate the specification by means of a functional throwaway prototype automatically generated. The animator (described in section 4) allows to create objects, to inspect the effect of the events on the objects, to check the validity of integrity constraints, to consult the state and history of the objects and even to move the system clock backwards in order to observe previous states of the objects. Thus, the analyst can interactively validate the conceptual schema with the users.
3. Once the specification has been validated, we should start the design and implementation phase. In this moment, it is essential to verify that the program produced is fully consistent with the conceptual schema previously validated. In this respect, our proposal is based on the automatic generation of a first

model is ensured by assertions, by using the principles of the “Design by Contract” technique. Once a skeleton of a class (set of method signatures and attributes) has been generated, the semantic of the specification, that was expressed by means of events preconditions, evaluations and integrity constraints, is automatically translated into class invariants, and preconditions and postconditions of the methods (as we will see in section 5). In this way, we introduce the semantic concepts already included in the conceptual schema in the first implementation model (totally deferred classes).

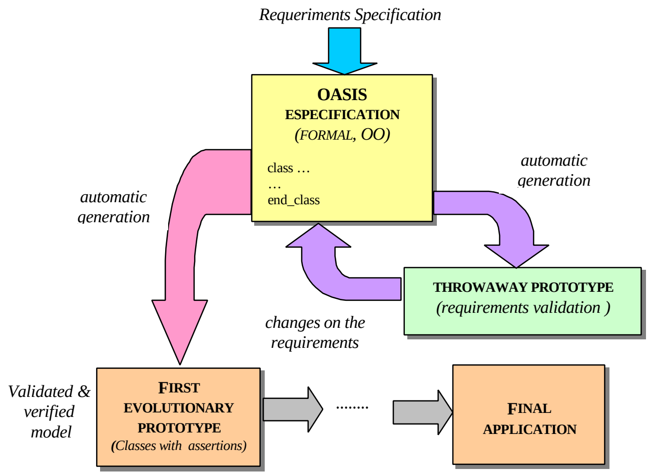

Requeriments Specification
OASIS
ESPECIFICATION
(FORMAL, OO)
class …
…
end_class
automatic generation
automatic generation
THROWAWAY PROTOTYPE
(requirements validation)
changes on the requirements
Validated & verified model
FIRST EVOLUTIONARY PROTOTYPE
(Classes with assertions)
……..
FINAL APPLICATION

Figure 4. Execution environment schema

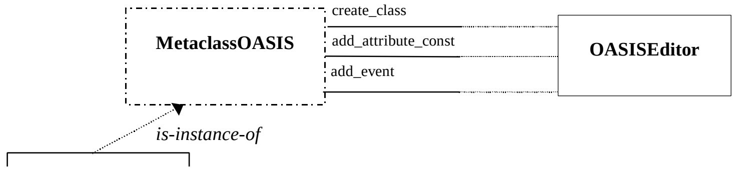
At this moment, we need to explain the reasons for the creation of two separate prototypes. If we directly generated deferred classes with assertions, then we would have to write the body of all the methods in order to be able to animate the specification,

4. EDITION, RAPID PROTOTYPING AND VALIDATION OF THE SPECIFICATIONS.

In order to develop the animation environment two things are needed. On the one hand, we have to keep the OASIS specification that we want to validate. On the other hand, we have to create the classes that will be used during the animation. The last statement requires an interpreted language, such as Smalltalk, that allows to create the specification classes at execution time.

We have chosen the OODBMS (*Object-Oriented Data Base Management System*) **GemStone** [Maier86] [Stein91] for implementing a first prototype of the prototyping tool. GemStone was designed by extending Smalltalk with the main DBMS features. So Gemstone provides features such as persistence, multi-user connections and functions for data migration that are required when there are changes on the specification.

The information associated to the OASIS specification could be considered as the system **metainformation**. From this point of view, the architecture of the environment could be described as it is shown in Figure 5, extracted from [Pelechano96], where an OASIS editor is described relying on the **metaclass** concept.



(*create_class*, *add_constant_attribute*, *add_event*, etc.) and, therefore, it could be understood as an abstract OASIS class editor. Any OASIS class will be represented as an instance of *MetaclassOASIS* (e.g. *Class Account* in figure 4). Hence, its instance variables will represent constant attributes (*attributes_const_set*), variable attributes (*attributes_var_set*), etc. (*events_set*,...). In order to validate the requirements, it will be necessary to create objects, instances of the generated classes (see the *Account* object in figure 4), during the execution of the throwaway prototype (specification animation)

We have developed an animation environment for managing all the metainformation, generating the throwaway prototype and executing this prototype. Figure 6 shows the structure of classes of the tool which is a Smalltalk implementation of the architecture showed in Figure 4. Now, we explain the mapping between both figures.

The relationship *is_instance_of* between the MetaclassOASIS and the *Account* class in Figure 5, now is implemented as follows: each OASIS class will be translated into an object and a GemStone class; the object is a instance of the class OASISClass (or some of its subclass depending on the kind of the class) which contains all the information from the specification. This object is used to create the GemStone class (pointed by the *realClass* attribute) whose instance variables are the set of attributes that determine the state of an object (constant and variables attributes). Besides, the GemStone class includes one method for each event declared in the conceptual schema. Therefore, the possible to create objects of the class *Account*, whose state will satisfy the integrity constraints (*code='c001'*, ...).

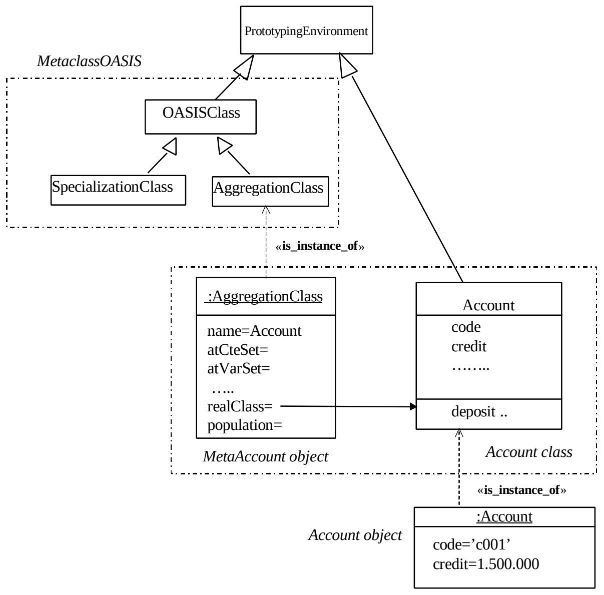


Figure 6. Classes and objects created from the *Account* class specification (throwaway prototype)

In a concise form, the equivalence between the OASIS concepts and the GemStone elements is:

- Every OASIS class is mapped into a GemStone class.
- Constant and variable attributes in OASIS are mapped into instance variables of the GemStone class. Their types in the specification are their constraints in GemStone[2] (Figure 7 shows an example). In that figure we can notice some new variables: *trace* and *Metainfo*. The former is an instance variable that

```
PrototypingEnvironment subclass: #Account
        instVarNames: #(#credit_limit #debit #credit #code #holder #trace)
        classVars: #( #Metainfo)
        constraints:#[ #[#credit_limit, Number],
                     #[#debit, Number],
                     #[#credit, Number],
                     #[#code, String]
                     #[#holder, BankCustomer] ]
```

Figure 7. GemStone definition of the *Account* class from Figure 5

- Derived attributes are calculated when they are needed, so they are modeled by a method that returns the value. For instance, there is a derived attribute in the *Account* class whose name is *red*. In consequence, there is a method in the GemStone class whose name is *red* and the body is the derivation formula (*derivation* section in the OASIS specification Figure 3), as we can see in Figure 8.
- The events are translated into methods that include the preconditions and evaluations from the specification, that is their functionality. Coming back to the running example, Figure 8 shows the code for the *withdraw* method.

```
Account >> red
        ^(debit>0)
```

```
Account>>  withdraw: anArray
        "The array contains the arguments: name,value."

                | M C D cond1 cond2 arg |

        arg:= anArray detect:[ : a | a name = 'M'.].
        M := arg value.
        (M ≤ (self maxWithdrawal)) ifTrue:[
                cond1:=(self red=false).
                cond2:=(self red=true).
                (cond1) ifTrue:[
                        C:=credit.
                        (C≥M) ifTrue:[
                                        credit:=C-M.]
                                ifFalse:[
```

conditions that are true are marked. Then, the actions corresponding to the marked conditions are executed. The returned value is *false* if the event did not occur (either precondition was false or any of the conditions was true), *true* otherwise.

5. CREATING THE FIRST EVOLUTIONARY PROTOTYPE

Starting from the "*Design by Contract*" technique [Meyer92], the BON method [Walden95] propose a "seamless" software development process in which the semantic gap between each stage of the life cycle is reduced. This is possible by using the same notation for analysis, design and implementation: an OO language with assertions (the method is based on concepts of the Eiffel language, although it can be considered an independent-language method).

Since the expressiveness of the assertion language that can be supported in a realistic programming language is rather limited, it would be very interesting that the developer could use a formal specification language at the starting and then, totally deferred classes with assertions would be generated automatically from the formal specification of the requirements. Our proposal is addressed in this sense.

According to this idea, we have devised a translation mechanism from an OASIS specification to deferred classes with assertions, particularly to the Eiffel assertion language (although this method could adapt to any programming language supporting methods.

- OASIS evaluations of an event are mapped to post-conditions of the associated method, as follows:

  Let Φ_1 [ev(N)] Φ_1', ..., Φ_n [ev(N)] Φ_n' be the associated evaluations to the event *ev*. The equivalent postcondition to the method associated to *ev* would be: (Φ_1''$\rightarrow$ Φ_1''') or ... or (Φ_n''$\rightarrow$ Φ_n'''), where each Φ_i'' contains all the Φ_i conditions (but the instantiations formulas are not included), and each Φ_i''' is like Φ_i', but replacing the variables instantiated in Φ_i by the attribute name preceded with the *old* prefix.

- For an aggregated or associated OASIS class, the counterpart Eiffel class models the relationship by attributes whose type is the aggregated or associated class. In OASIS, component classes (or associated) are accessible by means of dot notation (for instance, *BankCustomer.age>18* in *Account* class, in Figure 3). In Eiffel, we only have to substitute the class name for the attribute name, following the same example, *holder.edad>18,* see Figure 9).

- A universal specialization is translated according to the Eiffel inheritance mechanism. The temporal specialization or role can be implemented by means of patterns described for dynamic classification [Fowler97].

```
class Account feature
      -- constant and variable attributes
      code : STRING;
      type_account : STRING;
      credit : INTEGER;
      ...
      holder: BankCustomer;

red:BOOLEAN is do
      -- derived attribute
      Result:=debit > 0
end

-- events

withdraw(M: INTEGER) is
require           -- precondition
      M <= maxWithdrawal

ensure            -- valuations
(red = false)  and  (old credit >= M)
         implies credit = old  credit – M
(red=false)  and  (old credit < M)
         implies  (credit = 0  and  debit = M - old debit )
(red=true)
       implies  debit = old debit + M

end --withdraw

invariant   -- integrity constraints
      debit <= credit_limit;
      type = 'saving' or type = 'checking';
      holder.edad>18


end class
```

Figure 9 Translating OASIS into Eiffel (evolutionary prototype)

6. RELATED WORK

Another prototyping environment has been developed for the formal specification language **TROLL** [Jungclaus91]. It is called **Tbench** [Kusch95] and uses an editing and execution system for the specifications. In contrast to our environment, TBench does not automatically generate a first software model, as a first step in the development process of the final system. There is another CASE tool for **LCM** [Wieringa94] called but also by generating the first version of the application code.

Our environment has also similarities with **SmallVDM** [Lemos94]. SmallVDM generates a first Smalltalk prototype starting from a specification in the formal language VDM. We believe our work improves that one since we use a formal OO language as the specification language (while VDM is not OO) and a language with assertions (supporting "Design by Contract") as implementation language, instead of using an *ad hoc* definition of pre and postconditions.

7. CONCLUSIONS AND FURTHER WORK

We have presented an approach that addresses object-oriented software development in a formal and seamless way. Our proposal basically consists of automatically obtaining deferred classes with assertions from abstract data types expressed by means of a formal specification language. Moreover, we use a throwaway prototype to validate the initial specification, in order to generate programming classes from a validated specification. We believe on the usefulness of combining throwaway and evolutionary prototypes, where the design by contract ensures consistence between the program and the specification.

Within this approach, software development is assisted by an environment that allows i) to introduce the initial specification graphically, ii) to generate a throwaway

Smalltalk [Carrillo96], C++ [Porat95], Java [Payne98], and even there are some proposals which make that extension *ad hoc* [Lemos94].

Among other applications, our work can extend the BON method, by allowing to obtain the deferred classes with assertions, such as it is proposed by the BON analysis phase, from abstract data types expressed by means of a formal object-oriented language.

Further work is focused on integrating a UML graphic editor in the development environment, so that we could generate automatically OASIS specifications from UML diagrams (for the present, this is done by using an OASIS-specific graphic notation).

Another way to improve our work is to consider the changing nature of requirements. So, we can include the ability to generate the evolutionary prototypes incrementally from the formal model, while it is refined, instead of generating all the programming classes every time. In this way, we try to avoid wasting the work already performed on the current evolutionary prototype, when any requirement included in the formal specification is changed.

8. REFERENCES

[Carrillo96]M. Carrillo, J. García Molina, E. Pimentel. *"Design by Contract in Smalltalk"*. Journal of Object-Oriented Programming, vol. 9 (7), pp. 23-28, 1996.

[Davis93] A. M. Davis. *"Software Requirements: Objects, functions and states"*. Prentice Hall

Second Year Report, September 1991.

[IS-C93] Udo W. Lipeck, G. Koschorreck (eds.). *Proc. of the Int. Workshop on Information Systems - Correctness and Reusability IS-CORE'93*. Institut für Informatik, Hannover, September 1993.

[Jungclaus91] R. Jungclaus; T. Hartmann; G. Saake; C. Sernadas. "*Introduction to TROLL- A Language for Object-Oriented Specification of Information Systems*". Proc. IS-CORE Workshop WS'91, pp. 97-128. 1991.

[Kusch95] J. Kusch; P. Hartel; T. Hartmann; G. Saake. "*Gaining a Uniform View of Different Integration Aspects in a Prototyping Environment*", LNCS number 978, Database and Expert System Applications (DEXA'95) Springer-Verlag, 1995.

[Lemos94] S. Lemos and C. Souza. "*SmallVDM: An Environment for Formal Specification and Prototyping in Smalltak*". In "Object-Oriented Specification Case Studies". Edited by K. Lano and H. Haughton. Prentice-Hall. 1994.

[Maier86] D. Maier and J. Stein. "*Development and Implementation of an Object-Oriented DBMS*". OOPSLA'86 Proceedings (ACM). October 1986.

[Meyer92] B. Meyer, "Design by Contract" in "Advances in Object-Oriented Software Engineering", ed. by B. Meyer and D. Mandrioli, pp. 1-50, Prentice-Hall, 1992.

[Meyer97] B. Meyer. *"Object-Oriented Software Construction"*. Second Edition, Prentice-Hall, 1997.

[Meyer98] B. Meyer, C. Mingins and H. Schmidt. "*Providing Trusted Components to the Industry*". Computer. pp. 104-105. May 1998.

[Pastor95] O. Pastor; I. Ramos. "*OASIS 2.1.1: A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*". Octubre 95 (3rd ed.).

[Payne98] J. Payne; M. Schatz; M. Schmid. "*Implementing assertions in Java*". Dr. Bobb's. pp 40-

[Stein91] J. Stein; A. Otis; P. Butterworth. *“The GemStone Object Database Management System”*. Communications of the ACM. pp 64-77. October 1991.

[Walden95] K. Walden. *“Seamless object-oriented software architecture : analysis and design of reliable systems”*. Prentice-Hall, 1995.

[Wieringa94] R. J. Wieringa. *“LCM 3.0. Specification of a control system using dynamic logic and process algebra”*. Ed.C. Lewerentz and T. Lindner. Lecture Notes in Computer Science number 891. pp 333-355. Springer-Verlag, 1994.