Architectural Concerns in Automating Code Generation

L. F. Andrade, J. C. Gouveia, P. J. Xardoné and J. A. Câmara
OBLOG Software S.A.
Alameda António Sérgio 7 – 1 A, 2795 Linda-a-Velha, Portugal
{landrade, jgouveia, pxardone, jcamara}@oblog.pt,
tel:+351-1-4146930, fax:+351-1-4144125

Key words: Code generation, object-oriented modelling, contract-based architectural style

Abstract: We report on the problems (and solutions) that we have been facing in defining an architecture that enables us to automatically synthesise production code (COBOL, CICS, SQL) from a higher level specification language that includes both primitives that handle business and architectural requirements. Our experience has been drawn from a real-life project in the banking industry where object-oriented models for large-scale projects were used. With these models, the application architecture was conceived to be robust to change, accommodating new behaviour in a systematic and encapsulated way.

1. INTRODUCTION

Critical aspects of today's banking management information systems include time to market (dealing with component development and re-use), evolution (volatility of business requirements), requirement conformance (take decisions upon correct information), scale and complexity of systems, parallelism, maintenance, robustness and security. Product distribution, management information systems and decision support systems are typical banking applications facing these problems.

A particularly acute aspect of the problems that financial companies are facing today is the need for a technology migration plan from current traditional systems to future open systems. An encapsulation mechanism to

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35563-4_35

P. Donohoe (ed.), *Software Architecture*

hide "legacy systems" is essential to guarantee a smooth transition, coping with the business support extensibility.

Our purpose in this paper is to report on the experience that we have had in the combined use of formal architecture and transformations for assisting the migration of a banking application. More specifically, we will discuss the role of architectures in enabling us to automatically synthesise production code (COBOL, CICS, SQL) from a higher level specification language that includes both primitives that handle business and architectural requirements.

Our approach is based on the use of object-oriented models for large-scale projects. By using such models, the application architecture can be conceived to be robust to change, accommodating new behaviour in a systematic and encapsulated way.

In section 2, we briefly introduce some of the requirements and describe the banking project itself. In section 3, the main problems that we had to face are identified and the possible solutions are discussed. Finally, in section 4, a technique for automating production code based on transformations applied over a chosen architecture is presented.

2. PROBLEM DESCRIPTION

The global purpose of the project at hand was to migrate and improve the information system of a European mid-size bank with the following characteristics:

- 430 branches and 5000 PCs;
- 1 million transactions per day (average);
- 2 seconds of maximum response time;
- System hardware - IBM Mainframe;
- System software - MVS, CICS, DB2;
- Language – COBOL, SQL.

Our task was to migrate and improve the Retail Network, which meant re-construction of the Branch Transaction System (more or less 90 transaction types – opening accounts, withdrawals, deposits, transfer orders, etc.).

The main business requirements for this project were to:

- Improve the system functionality to deal with the new European currency (EURO);
- "Solve" the year 2000 problem;
- Adapt the system in order to inter-operate with a new package that manages the "financial products" offered by the bank;

- Adapt the system so that the bank could be open 24 hours a day (mainly because of Internet access).

The main implementation requirements for the project were:

- The client tier could not be changed, meaning that the format of all communication messages (between the client and the server) had to be preserved;
- The target technology (MVS, CICS, DB2, and COBOL) was fixed;
- Some functionalities, like check-digit validation, time-stamps, etc. were supplied by already existing routines which we were obliged to use;
- The format of communication with other modules was fixed and not changeable;
- All the technical documentation formats were also fixed and had to be followed;
- Some customer implementation techniques had also to be followed.

The kinds of problems we had to face in this project are very common to real projects. Even with the availability of many commercial CASE Tools supporting object-oriented methods (and in particular supporting UML), our main problems were to come up with answers to the following two questions.

1. How to synthesize the final production code automatically from the high level specifications?
2. To achieve the previous goal, what language/method should we follow to specify the system, including all of its details?

3. OUTLINE OF THE MODELLING APPROACH

An obvious answer to the second question above was to choose UML because it is a standard visual-modelling notation that is already in place. However, from our experience, in order to use UML it is necessary to have confidence in all of the notations and techniques that are offered. Furthermore, integrating such techniques and notations seems to be a difficult task, the feasibility of which needs to be demonstrated particularly if the goal is to automatically obtain code from specifications.

Given these caveats, we decided instead to use a rich, yet integrated and precise subset of the UML notations – which we called OBLOG – adding to this subset a rigorous and formal specification language supporting the generally accepted OO key properties of

- support for encapsulation of services and state as objects
- the ability to create object instances from class templates

- the ability to define new object templates by monotonic modification of existing ones (base classes)

OBLOG introduces new specific features such as

- integration of the concept of module in the class concept as a way to introduce different levels of abstraction and encapsulation;
- specialised language constructors to define object behaviour at distinct levels of detail;
- visibility of objects defined by contracts as an architectural style for the construction of complex systems.

These features are supported with full integration of graphical diagrams and textual specifications.

- The graphical notation is compliant with the UML standard.
- It allows for a continuous path from high to low-level design specifications, always using the same specification language.
- The textual language is mainly used for the design details.

An effort was made to provide OBLOG with a well defined semantics as a means of supporting key aspects of object-oriented construction such as method composition and extension, direct object interaction and event multicast, behaviour inheritance, composability, and encapsulation. Some properties relevant to wider software engineering were also included, namely the ability to specify concurrent behaviour properties and to deal with non-normative behaviour (exception handling); allowing the systematic refinement of specifications to code, preferably in a compositional manner.

According to these principles, an information system is treated as a collection of interacting concurrent objects. An **object** is an abstraction of an entity with a persistent *identity*, a *public interface* defined by the provided services and recognised events and an *internal body*. The internal body includes hidden *local methods* implementing the public interface, possibly calling some hidden local auxiliary services, a *computation state* indicating the object situation in its life cycle, an *internal state* (represented by its slot values) storing the effects of method executions, *hidden enabling conditions* constraining services and reactions, and *hidden invariant conditions* constraining state changes.

In practice, specifications tend to involve a large amount of objects which makes understanding and managing them a real problem. OBLOG deals with this problem by providing a decomposition mechanism that allows complex objects to be defined that can be later detailed in terms of other simpler objects.

The ability to decompose specifications also allows the analyst to introduce new objects at any level of the specification. The way of making those new objects, introduced locally for a given complex object, visible to other objects, is through a **contract** mechanism. In this sense, contracts are

used to enrich the interface of a certain object, allowing some of its components to be seen by others.

In the following example (Figure 1), an *Account* object makes a contract with a *Customer* object (named *CtWithAccount*). This allows for public objects defined in the decomposition of Customer (e.g., *CustomerProfile*) to be used by *Account* and any object in its decomposition. On the other hand, the contract *CtWithCustomer* allows for obligations to be defined between Customer and Account.

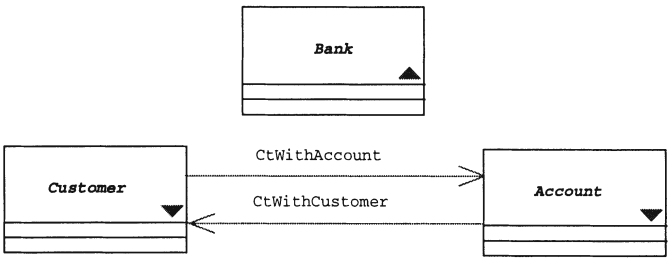

Figure 1. Contract between Account and Customer

Contracts are the privileged architectural style for OBLOG specifications. They are used to express *dependencies* that characterise *collaboration* relationships between objects. These kinds of relationships allow the analyst to perceive the way objects work together in order to perform some task.

Table 1. Withdrawal obligation requirements

Detailing obligations to define the *withdrawal* requirements		
Involved concepts	Customer obligations	Account obligations
X : Account	Customer Y owns Account	Balance W of Account X
Y : Customer	X	is decreased by Amount Z
Z : Amount	Balance of Account X is	
W : Balance	greater than Amount Z	
	owns (Y, X)	X.balance = X.balance - Z
	X.balance >= Z	
Deriving formal specifications from the above requirements (Account *withdrawal* operation declaration)		
PRE-CONDITION: ?owns(self,Y)=TRUE AND self.Balance() > Z		
OPERATION: withdrawal(Y : Customer, Z : Amount)		
POST-CONDITION: self.Balance() = old.Balance() - Z		

Contract-based architectures are also used for evaluating the *impact of changes* and to maintain *traceability* of concepts. In fact, when objects contract between them the components they need, they are explicitly, creating strong dependencies between them. These dependencies are of the

outmost importance, and constitute a very important input when analysing the impact of changes in a model. Contracts between objects enable the OBLOG tools to check for those dependencies and to make available to the analyst, at any level of the specification, detailed reports about them.

In order to achieve the production code generation, OBLOG provides some concepts to define detailed behaviour of operations and interactions.

Object interaction can be direct (calling a service operation of a known target) or indirect (event multicasting). Events are incidents (or *stimuli*) requiring some response.

An operation may be classified as

- *service* - executed by direct demand of a caller object
- *event reaction* - starts method execution for every object that recognises the event
- *self-initiative action* – internal operation initiated only by the owner object, when some condition holds

Conditions may constrain operation execution. *Enabling conditions* take into consideration the internal state of the object, avoiding invariant violations. *Preconditions* are specified only on the service and event parameters, indicating the operation client obligations when using that service, and giving no guarantee about the result of an operation if its method is executed outside them.

An operation execution is supported by a main *method* and a set of possible alternative *methods*. The main method is the one selected for execution whenever the operation happens. Only when the main method can't execute due to its enabled conditions, the object tries an alternative method for that operation, if defined.

Methods are composed of *local variables* and *quarks* that exist only within the method scope, and during a method execution. A *quark* is the minimal unit of object dynamic specification, with a guard condition and a body responsible for the effect on local state and interactions.

4. OUTLINE OF THE PROPOSED ARCHITECTURE

Having chosen a set of concepts that is rich and precise enough to build the intended models (as an answer to the second question posed at the end of section 2), the problem is then reduced to the following questions.

- What architecture should be chosen for COBOL/CICS/DB2 applications in order to support these concepts?
- How do we automatically synthesise production code from the defined architectures?

– How would we be able to easily interact with already existing applications, with that interaction clearly and rigorously expressed in our models?

Figures 2 and 3 give an overview of the approach that we implemented.

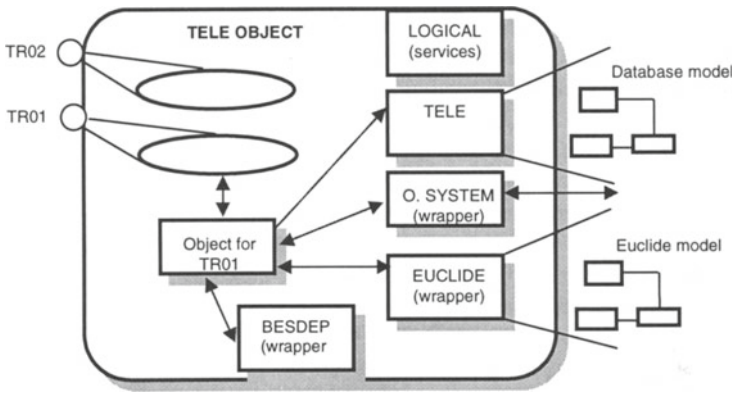

Figure 2. Logical architecture

This first figure shows the way we conceived the logical architecture for the project. The whole model is seen as an OBLOG object and therefore has an interface; in this interface we declared the set of transactions that could be called from the clients. Then, for each transaction, we created an active object that implements it, and used a delegation mechanism to direct the client calls to the right server object.

Having an object for each transaction enabled us to locate in those objects all the auxiliary operations needed for the transaction, implementing specific behaviour for that transaction. General business rules were implemented in a separate object.

We also defined two kinds of auxiliary objects.

1. objects that implemented wrappers to the data persistence mechanism (object named DB), to the external applications we had to interact with (objects EUCLIDE and BESDEP), and to the operating system (object SYSTEM)
2. an object that aggregated all of the general business rules that were used by some (or all) of our transactions (object LOGICAL)

Objects of the first kind gave us an invariant on the environment, allowing us to develop the code of the transactions without having to concern ourselves with the external changes that could have an impact on our implementation. The second object had a similar objective in the sense that it was designed to ensure that all the general business rules were

fulfilled by all transactions, and that changes in those rules would have an immediate impact on all transactions.

Between the wrappers referred to above, there was one that hid the data persistence mechanism, one for which we will provide a little more detail. The OBLOG language has a set of primitives that enables the software engineer to directly manipulate the storage and retrieval of objects from disk. Though it seemed to be the natural solution to implement data persistence, there were several reasons that led us to take a different approach, implementing it as an external object.

- Data persistence is frequently a delicate point in time-critical systems, where fine tuning is often needed for performance reasons.
- In the first stages of development it was not yet decided if all of the data persistence was managed by DB2 or if we had also to deal with VSAM files.
- Using a wrapper was already the chosen solution for other "collateral" problems.

In fact, the wrapper that was hiding the data persistence ended up being developed as an OBLOG model on its own, and all of the data access code was generated automatically.

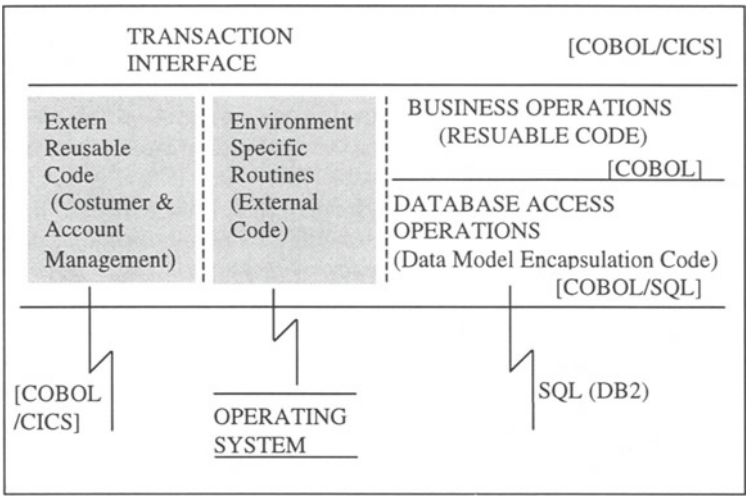

Figure 3. Physical architecture

The physical architecture is organised in terms of different layers, each of which takes care of specific tasks.

- The first layer is the interface for the clients and implements the logic of each transaction (basically, a sequential composition of logical services).

This first layer is coded in COBOL with CICS instructions for transactional purposes.
- The second layer contains all the general services that are reused by the transactions. This layer contains COBOL code only.
- The third layer implements all the data persistence operations and is coded in COBOL with SQL instructions.
- Also present are modules that implement the connections to exterior services, whether they be operating system services (e.g., get current date), standard services (e.g., check-digit validation) or accesses to other business applications (e.g., validate client information, perform changes on account balance).

In summary, the underlying ideas for the proposed architecture were
- choosing an architecture for COBOL programs/transactions in a way that they can be seen as a composition of sub-routines, each one implementing an object operation from the specification. With this strategy, we can achieve encapsulation of objects.
- Although polymorphism cannot be implemented using traditional COBOL, a certain degree of inheritance (in the perspective of code re-use) can be achieved using a code inclusion mechanism.
- Persistence of objects can be managed by a relational database (DB2). For this purpose, a model object can be created for ensuring data persistence, encapsulating all of the data accesses (either supported by DB2 or by any other mechanism). The database and the access to SQL tables are then defined through a module whose interface consists in creation, modification, retrieval, and deletion operations per object, hiding internal optimisations;
- All external components interacting with our system are isolated, in a systematic way, clearly defining the communication points and avoiding undesired collateral effects.

The next section describes how, given such an architecture, code can be automatically produced from specifications in a rigorous and continuous process.

5. SYNTHESISING PRODUCTION CODE

Automated code generation is a goal developers have been trying to include in project life cycles for a long time. It is usually viewed in two ways

1. as a feature that produces only part of the expected result, which makes it largely unused (the pessimistic view), or

2. as a feature that, when well managed, can bring significant productivity gains (the optimistic view).

The most common view is probably the first, due to the inability of current tools to address some key points in automated generation. The usual problems that real projects face are, among others:

- Insufficient code is generated (most of the times only templates).
- Code tuning may not be preserved on consecutive generations.
- Incremental generation is not available.
- There is a lack of strong customisation facilities.

We use the expression "automated generation" in the general sense of producing automatically any written information from already built models. This typically includes source code and several kinds of documentation.

With the OBLOG tools we have addressed the previous problems using the following key features.

- use of a rich specification language
- use of an open repository model
- support for customisable repository query/report technology and tools

We now discuss this technology, its principles, and how it was used in the project for enabling code generation.

The generation is a *transformation process based in rewrite rules.* A model transformation process is the application of rules to a set of objects in a certain order and according to a given strategy. The principle of a rule is to define an elementary transformation on repository objects. By transformation we mean the process of querying the repository and generating information from it, either in a textual form or into another repository (possibly the same).

In the following we briefly present the characteristics of the rule language, with small examples.

- A rule perform actions. These actions are of several types, including elementary actions to output text and values to files, creation and modification of objects, output to user, manipulate variables, etc.
- Rules can access object properties and relations, including the repository hierarchy.
- A rule can use other rules. This allows for rules to be applied in isolation, or to be used in a call sequences (procedural way).
- Iteration mechanisms are provided to iterate over lists of objects, the repository hierarchy, numerical intervals, etc. Several groups of actions may be defined on iterations to help the processing of a list, like actions to be executed before/after the iteration, as a separator/terminator of each element, etc
- A rule always has a context object of execution. This provides an object-based view of rules. Whenever a rule is applied there is always an

underlying context object. When a rule uses another rule it is implicitly applying it on the current context object.
- Some actions may change the current context object, like when iterating through a list objects. An explicit way to change the context to a given object is also provided.
- Rules are polymorphic (on the invocation context object class). A rule body may be defined to be applicable only of objects of a certain repository class. The same rule may have several bodies for different classes. According to the context's class the corresponding rule is applied at invocation time.
- Rule bodies may have pre-conditions. At invocation time, only the rule body for which the condition is true will be applied. If no condition is true, the rule fails and a warning is given to the user.
- Rules have a simple organisation structure based on hierarchical modules

A particular kind of model transformation is source code generation and project documentation. In this particular case we used transformations from OBLOG representations to RTF and HTML in order to obtain, automatically, all the project documentation. And we used transformations from OBLOG representations to COBOL, CICS, SQL to automatically obtain source code. Figure 4 illustrates some of the above characteristics using a simple text generation. The following example concerns the rules to generate DDL code, from which we present a simplified fragment.

```
<$ public ddlCreateTable> ::=
<! "Creates one table">
  <foreach Slot>
  <before>
    '-- TABLE ' $ObjName <nl>
    'CREATE TABLE '$ObjName ' ('
    <? v_targetDB = K_DB2> ' \\ ' </?>
    <nl>
  </before>
    <call ddlTableAttributeCreation>
  <sep>
    ',' <? v_targetDB = K_DB2> ' \\ '</?>
    <nl>
  </sep>
  <after>
    <? v_targetDB = K_DB2> ' \\ ' </?>
    <nl>')'
    <? v_targetDB = K_SQLSERVER>
      <nl>'go'
```

```
    <else> ';'
    </?>
    <nl 2>
  </after>
  </foreach>
</$>
```

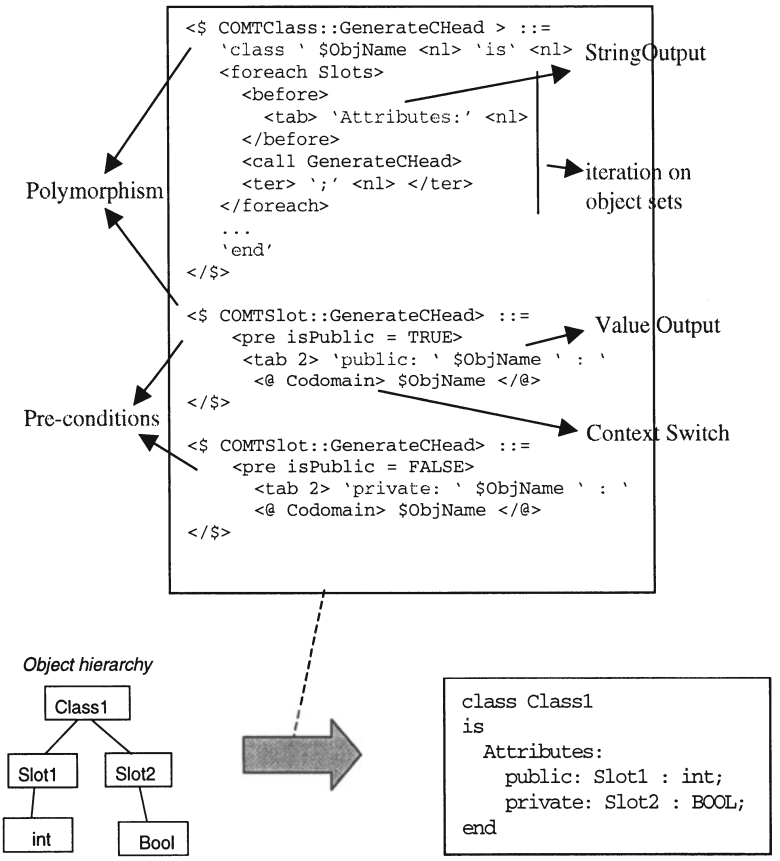

Figure 4. Some of the rule characteristics

In a real project, the ability to customise the generated output is a key issue. It is not reasonable to think that a general code generator can serve all needs. Several aspects may contribute to this.

- A project may use a very specific underlying architecture, and specific (non-standard) target languages.
- There are proven results for some design and architecture patterns that work better in certain kinds of systems.
- Each organization has its own rules and standards that must be fulfilled.

The full customisation of code generation according to well defined architectures, enabled us to develop the project with a consistent high quality level.

OBLOG provided us with a default set of rules that performed a standard code generation for a CICS/COBOL/DB2 architecture in a MVS environment. However, those rules had to be changed to meet the project needs, in terms of naming normalisation, machine-dependent details, project dependent constraints, etc. In fact, during the development phase of this project we had two major examples of almost importance concerning the ability of easily customising source code generation.

First of all, as we said previously, we had to preserve the format of all communication messages sent by the client. In those messages, together with the semantically important parameters, there was some machine-dependent header information (totally irrelevant to the specification) and some obsolete parameters. To make things even more complicated, all of the message was compacted in a stream of characters that was the only real parameter that physically arrived at the server, and that was supposed to be sent back.

In our models, it made no sense to declare both the obsolete parameters and the machine-dependent header parameters. Moreover, we didn't want to embed in the specification some machine-dependent details. However, the idea was to generate executable code from the models, and in order to do that, those architecture-specific details had to be somewhere in our specifications.

We solved this problem by acting upon the set of generation rules, defining new rules where those project specifics were expressed. This way, we were able to design "clean" models, where only the semantically relevant information was defined and no "noise" was introduced, and yet we were able to automatically obtain the production code, with all the needed particularities.

The second major problem was a self-inflicted one. Our project was, at first, designed to implement only OLTP branch transactions. As we managed to do that before the scheduled date, our prize was to implement all of the batch transactions as well. When we made the analysis of those transactions,

we realised that most of the business rules that were used for the OLTP transactions applied to the batch ones. This sounded like good news to us, but again we had an architectural problem to solve: the "call" mechanism that we used for communicating between the OLTP transactions and the object that provided all of the general business rule validations was not acceptable for a batch process, because it was too resource consuming (we had about 1500 calls in our models). As the code generation algorithm is expressed in a set of rules, it was very simple to change the call mechanism into a "code inclusion" mechanism, with no changes on the specification, coping with the environment constraints.

There were other project activities where we envisaged the use of this technology, namely model validation and impact analysis, although we did not apply it to its full extent. In terms of impact analysis, it is possible to produce reports on the interdependencies between objects, in particular the ones that are system-critical in terms of change management.

The rule engine presented above also provides a mechanism on which rules may be executed under the control of an *integration model.* This allows for transformations between a source model and a target one to be recorded in a way that they can be re-played later. In this way, consistency between source and target can be maintained much more easily. Traceability reports on transformations or inconsistency reports are easy to produce.

In the context of this project, this mechanism was applied to integrate the conceptual object-oriented OBLOG model and the relational database model. Rules were provided to transform a class model into a relational one, over which the DDL/DML generation rules were applied.

We also want to stress that the power of using specific queries on the models is increased by the facility that the tool provides in categorising objects, and relations between them, in many different ways. By classifying the objects according to some user-defined categories (e.g., architectural, persistency, interface) the application of transformation rules is much more flexible and targets correctly each object role in the system.

6. CONCLUDING REMARKS

To really have a continuous process from specification to production code it is mandatory to have

- a rigorous specification language with concepts for business as well as architectural requirements, a methodology to explain how to use these concepts to create models, and a computational tool environment
- a process to obtain production code 100% automatically generated from the specification models

– a flexible mechanism (based in interpreted re-writing rules) to query the repository meta-models to make the necessary transformations

In this approach, extendible transformation rules play an important role, because they are the only way to precisely incorporate in a generic software development process the specifics of a particular project.

Recall the three axes approach to the software development cycle (see figure 5). The solution must be clearly expressed without any concerns with the system architecture or the target software environment. Then, the solution must be matched to a given system architecture, and the target software environment must be chosen, so that the appropriate set of rules can be used.

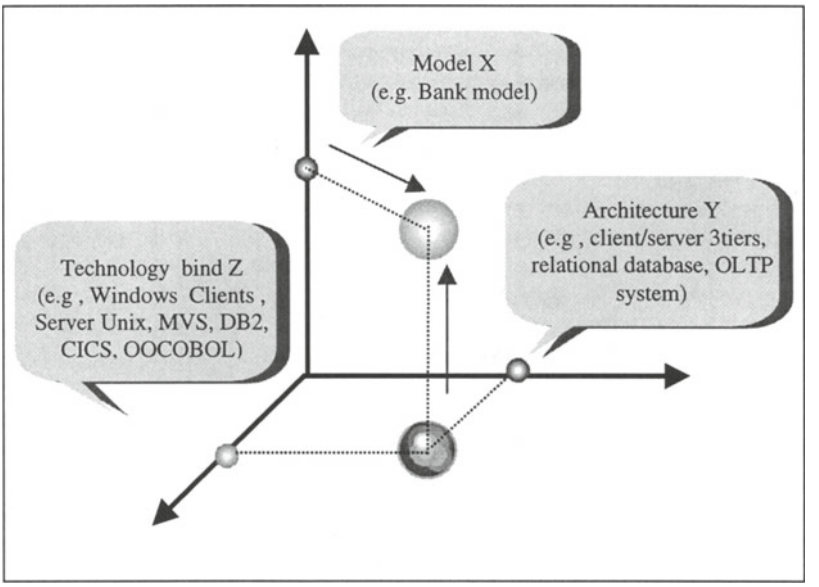

Figure 5. Software construction dimensions

At the moment, we have achieved a clear separation between the domain axis and the other ones. However, we still need to work on the separation of the system architecture and the software environment, which are currently too tied up in the rule scripts. In order to do that, we are working on the definition of architecture patterns that will be recognised by rules that will contain only the knowledge of how to translate a certain model and a given system architecture to a target software environment.

REFERENCES

L.Andrade and A. Sernadas, "Banking and management information system automation", Proceedings of the 13th world congress IFAC, Volume L, 1996.

Rational, Unified Modeling Language, http://www.rational.com, 1997.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling Technique, Prentice Hall, 1991.