

Executable Platform Independent Models for Data Intensive Applications*

Grzegorz Falda¹, Piotr Habela¹, Krzysztof Kaczmarek², Krzysztof Stencel³,
and Kazimierz Subieta¹

¹ Polish-Japanese Institute of Information Technology, Warsaw, Poland

² Warsaw University of Technology, Warsaw, Poland

³ Institute of Informatics Warsaw University, Warsaw, Poland

{gfalda, habela, stencel, subieta}@pjwstk.edu.pl,
k.kaczmarek@mini.pw.edu.pl

Abstract. In this paper we investigate the capabilities and shortcomings of UML 2 in the field of executable modelling, with special focus on database applications. As envisioned in the MDA initiative, this sets the challenges like achieving platform-independence of the base model, complete specification of behaviour in an adequate way and making the model an unambiguous and suitable input for model compilers. Additionally, we need to provide sufficiently powerful and abstract means of creating expressions over data, so that our platform-independent language is not inferior in that matter compared to database query languages known from DBMSs.

Keywords: Executable Models, Platform Independent Modelling, Database Modelling, Database Query Languages, Model Driven Architecture, UML, OCL.

1 Introduction

The main expectation behind model-driven development based on platform-independent models (PIMs) is earning more value in the process of model development. As far as the idea of Model Driven Architecture (MDA) [3] is considered, this leads to the following requirements:

- Ability to accompany the modelling solution with highly automated means of producing executable code.
- Platform-independence, which isolates the investment made on producing a model, against the future changes of the target deployment platform (for example from particular solution of data persistence – relational or object database).
- Representing behaviour inside the model in a form that offers a level of abstraction higher than the one anticipated at the target platform code (otherwise the expressiveness of the model would be questionable and producing efficient code would be problematic).

* Supported by the EC 6-th FP, Project VIDE, IST 033606 STP.

- Modelling notions offered to the developer based on existing technologies and standards. This is important from the point of view of the cost of adoption, as well as because the reuse of existing modelling tools is desirable.
- All details (e.g. type information) essential for model compilers need to be easily available in the model. For example, if one platform-specific programming language represents updates of local variables and persistent data uniformly, but on the other platform the processing may differ, model should make the distinction easily available. In other words, while in the concrete syntax offered to a modeller transparency and uniformity are desirable, the amount of information actually recorded in the model needs to take into account processing requirements of various target platforms and respective model compilers.

In this paper we describe a solution for creating and executing platform independent models based on UML [4] in the field of data intensive applications, being able to process large data sets using a higher-level query language. In the rest of this section, we discuss model execution. In the second section, we describe a language created for purposes of PIM specification, while in the third section we briefly present the IDE for this kind of models.

1.1 Model Execution

The technology of executable models is originated from the Shlaer-Mellor method [6]. According to [2], Executable UML is a profile of UML allowing the developer to define the behaviour of a single subject matter in a sufficient detail that it can be executed and tested. An executable platform-independent UML model describes only the data and the behaviour, without making any platform specific decisions. It does not specify organization of data, even an implementation involving classes and objects is not necessarily required. An Executable UML model does not define neither system distribution nor the number and allocation of separate threads. Each platform calls for a different model compiler according to specific organization of hardware and software. This covers also the operating system(s) (if any), the programming language(s) of the platform and solutions concerning data persistence. Further platform specific issues are the approaches to optimization, transaction framing, fault-tolerance and more.

We consider the ability to execute the model to be valuable for the following three main purposes:

- **Software construction.** Complete platform-independent models can be used to produce software for the target platform. That's where the increased effort on creating models is expected to pay with the savings on coding activities at target platform. The degree to which this step can be automated depends, among other factors, on the complexity of platform-specific technical details.
- **Testing and debugging.** Assuming that the model compilers under consideration constitute a mature technology (i.e. construction of platform specific artifacts is a more or less automated step and is not error prone itself), the burden of assuring that software would work according to the developers' intent is shifted to the PIM level. Hence the need of easily available model execution engine (not necessarily the same as the target deployment platform assumed), which could be invoked

from the development environment. In this case, the additional challenge (apart from just running the model) is allowing its stepwise execution and tracking.

- **Prototyping and validating with final users.** In cases when moving from PIM to the target code involves significant effort (which – given the complexity and multi-tiered nature of the technologies used for building typical business applications – may be a quite usual situation), it becomes essential to validate models somehow before proceeding to platform-specific work. Involvement of the prospective users is desirable; however, the model itself can be not comprehensible enough. Hence, we assume the need for at least rudimentary means of prototyping the user interface that could cooperate with executable PIMs.

The above considerations on model execution lead us towards a language that, at least from the designer/programmer point of view, resembles typical database programming languages. This means, that although its area of application lies somewhere between traditional modelling tools and programming languages, the level of precision and skills necessary brings the requirement for the developers of programming background.

1.2 A Textual Language for the PIM

Here an important question arises: why do we need a textual language at all? Textual language is necessary to describe precise behaviour of models since visual syntax operates on higher level of abstraction and usually leaves many details unspecified. Visual programming/modelling is therefore used only for a limited set of programming constructs leaving possibility for programmers/modellers to switch to textual syntax any time and complete programming of details.

There is also no CASE tool currently available that is able to create complete UML actions and activities models. Most of them model only selected features of UML standard. This selection is arbitrary and different in each tool making them very often incompatible.

Efficient processing of data structures in typical business applications requires means of specifying expressions (queries) with similar power to SQL, OQL or XMLQuery. Again, a natural choice for this is to adapt an existing OMG standard – namely, the OCL (Object Constraint Language). The language is extensible, prepared to be used with UML, and the role of “query language” is among the applications anticipated by its authors.

Concluding, our textual VIDE language defines a textual syntax that is human readable and similar to other programming languages, supporting necessary UML behavioural constructs accompanied with OCL expressions and queries (in this part using the standard OCL’s syntax) but used exclusively on PIM level.

2 A Programming Language Built on UML and OCL

As stated in OCL 2.0 [5] specification, the main reason for introducing this language in UML models is to be able to describe expressions:

“Object Constraint Language (OCL) is a formal language used to describe expressions on UML models. (...) OCL expressions can be used to specify operations/actions

that, when executed, do alter the state of the system. (...) UML modellers can also use OCL to specify queries on the UML model, which are completely programming language independent.”

We then might expect, that OCL makes UML a tool not only for modelling but also capable of creating platform independent executable models, with behaviour specified by OCL and UML. In this section we shall analyze the integration of these two standards and try to conclude whether they can be used for real-life business database applications development.

2.1 Responsibilities of the Standards

It is important to note that not all features of UML and OCL are useful from VIDE PIM language’s point of view. If we analyze OCL 2.0 and UML 2.1 standards, we can observe that their functionality overlaps in many places, for example: we have *ReadStructuralFeatureAction* in UML and similar *PropertyCallExp* in OCL, *CallOperationAction* in UML and corresponding *OperationCallExp* in OCL, etc. To avoid functional and semantic inconsistencies, we divided responsibilities between these two languages following a simple rule: OCL is used as a query and expression language to access data, while UML is used to update data and to cover other imperative constructs. Here we describe the responsibilities of both standards in our solution:

- UML – defines data structures, defines program behaviour in the domain of imperative constructs (loops, conditionals, program blocks and exceptions), updates values and objects in the system;
- OCL – defines queries over data objects described by UML models and stored in the system and represents operation calls.

2.2 Problems of Integration

The described integration exposed various problems, however. We must admit that even though OCL 2.0 is advertised as a "language that may be used in UML models in any place where an expression may be used", it is not fulfilling this expectation. Below we enumerate just a few of the most important problems:

- OCL expression cannot access local variables defined for example in method bodies;
- Although there is a conversion specified from UML types to OCL types, there is no explicit definition of the opposite conversion. It is then formally impossible to consume OCL expression results in UML actions and other UML constructs;
- There is an important problem of correct and common interpretation of collection types. In UML, a collection is represented by multiple values. OCL defined dedicated collection types, which are containers for stored values. When OCL expression is accessing UML multiple value, it is converted to appropriate OCL collection instance. On the other side also OCL expression (or query) may return multiple values, which are packed in a collection type. However, from UML’s point of view, OCL collection is just a single value. There is no reverse mapping from OCL collections to UML multiple value variables. Because of that, standard UML cannot treat OCL collections properly and cannot handle them for example in Expansion Regions;

- OCL allows a programmer to define custom types (for example for collections) but does not define how they should be stored in UML models;
- OCL expressions cannot refer to any of UML actions. This may be unimportant if OCL is only used simply as expression and query language, but may become critical if one would like to use do meta-programming and analyze UML behavioural models during query execution.

2.3 Limitations of Textual Programs Represented in UML Models

It was surprising to notice that UML cannot cover, in a reasonably straightforward way, all variety of constructs that are handy for textual programmers. In fact, some constructs become so popular through their presence in mainstream programming languages, that their straightforward support in UML seems highly desirable. There are three types of these limitations.

First one is connected to textual syntax shortcuts. For example programmers often use shortcuts for assignments or incrementing operations (like +=, *=, ++, etc). There are no corresponding notions in UML. When such constructs are expanded to complete statements and then translated to appropriate UML elements, we loose information about the original syntax, which is important if we want to be able to switch back to a textual view of the model.

Second one appeared in constructs like loops and switch statements. UML contains only one element for all kinds of loops (although with appropriate flags to distinguish for example between do...while and while...do). Similarly, the switch statement is modelled with elements which are also used for if...then...else statement. Also different types of comments fall into this category. Again, we loose information on the original syntax in the textual code.

Last but not least, there is a potentially useful information about the current state of textual view, which also should be stored in the model to be able to present exactly the same view to the programmer when moving back from model view.

Above limitations are not really any kinds of flaws of the standard, but rather points, which must be taken into consideration when designing a tool for high-level programming based on UML. Thanks to extensibility mechanisms and model annotations, these programming requirements may be fulfilled quite simply.

2.4 UML Class Diagram as an Object Database Schema

Adopting class diagrams to represent object database schema requires several additional explanations. From object oriented database point of view we need at least names for all objects (names for classes are not enough) and starting point for queries (so called set of root objects). Root objects constitute main part of the schema, which may be compared to a main module or an entry point for procedural programs. Fig 1. presents an example schema adopted to needs of object database.

There are five classes: *Person*, *Author*, *Editor*, *Book* and *SoldItem*. They are defined using regular UML notions. From database point of view we need to give also names to objects, that is instances of these classes. To achieve this goal we introduce a container singleton object, which will be a main module of our system: *BookstoreSystem*. It contains properties named authors, editors, books and soldItems which in fact model a place, where our root object will appear. *BookstoreSystem* is in the same time context and starting point for all queries.

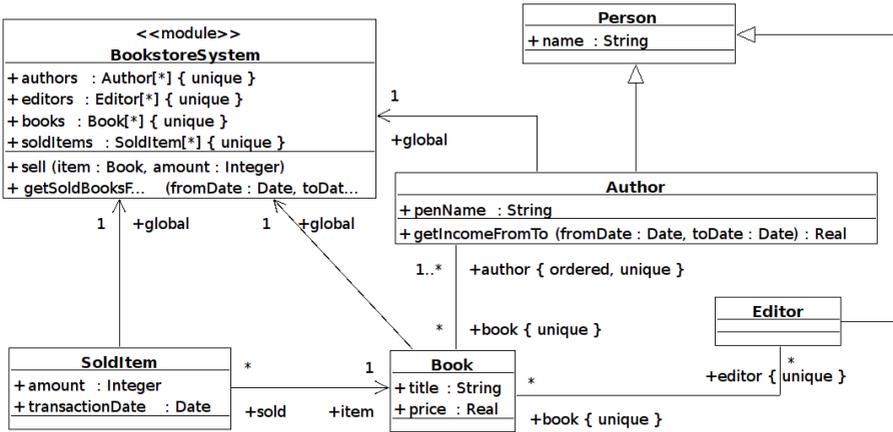


Fig. 1. Example UML schema adapted to database needs

Another observation which we made during implementation of UML execution engine is lack of global variables which may be important for database programs. We simply should be able to write a query starting from root objects in any possible context in the schema. Unfortunately even when *Author* class instances are embedded as properties of *BookstoreSystem* (as authors collection) procedure *getIncomeFromTo*(from:Date, to:Date):Real cannot access for example books collection. To make it possible we added explicitly associations from all objects to *BookstoreSystem* object, which is the global context for the schema.

2.5 Means of Achieving Semantically Complete Model Compiler Input

In order to achieve functionally complete and semantically consistent models, we have chosen not to extend UML's meta model, willing to be able to analyze VIDE models in any UML 2.0 capable CASE tool. Thus, all the constructs which appear in the VIDE code repository are standard meta model compliant. There are, however, three types of modifications that had to be done to achieve semantic completeness and to overcome the fact some constructs are inexpressible in the standard way:

1. Adding more information to UML constructs. This was implemented using EMF annotations facility. All information required for modelling tools and for easing the translation from textual code to models and vice versa is provided this way (like annotation letting us to distinguish := and += when transformed to Add*ValueAction element) .
2. Limiting model details to the level acceptable by VIDE programs and expressible by VIDE programmers. This was done simply by omitting UML meta model constructs that are not relevant to VIDE.
3. Extending the semantics of OCL-UML integration. This was done by clear explanation of new or updated behaviour of already existing UML constructs, which, similarly to UML and OCL standards, needed to be communicated to all involved

developers of the toolset element; particularly – to model compiler developers, and may be subject to future standardization process.

3 Model Editor and Compiler (IDE)

3.1 Model Creation and Compilation Process

A user of the VIDE editor first creates the data model of the application. The data model has the form of a UML class diagram. Then, the user generates stubs of the methods declared by this diagram. These stubs can be filled by means of one of the code editors: either textual, visual or other domain specific wizards. The method bodies are then type checked, compiled and eventually put into the UML model repository besides the elements implied by the class diagrams.

If the user wants to execute and test the model, he/she will have to provide a connection to an execution platform (in our case ODRA database server). Model compilation generates code for specific platform (in our case ODRA) directly from the UML repository. The generated code is then run on the ODRA server thus installing all the objects of the compiled model. The user can now write and run ad-hoc code (e.g. method calls) in order to test methods or the sole schema. The whole process is presented on the semi-formal Fig. 2. Three actors are depicted on this picture: the user, the VIDE editor and the ODRA database. Dashed arrows denote derivation dependencies between documents (diagrams, code and repository content). Solid labelled arrows represent authoring or usage relationships between actors and documents. For example, the solid arrow, which connects the user with the class diagram, means that the user *creates* this diagram.

3.2 ODRA Platform and Platform Specific Parameters

The experimental DBMS ODRA (Object Database for Rapid Application Development) [1] is the model execution platform in our project. ODRA is built according to the Stack-Based Architecture (SBA) which has been presented in a number of papers (e.g. [7]), but currently the best description of SBA can be found in [8]. ODRA DBMS perfectly suits the needs of a model execution platform because of language SBQL, programming abstractions and adaptive semi-strong type checker.

This platform implies that persistent objects are similar to volatile ones and are generally indistinguishable. ODRA assumes that all local variables are only temporal, while properties of classes marked as *module* are persistent. ODRA model compiler translates typical object manipulating statements to ODRA language, which operates on dedicated object store. Other compilers, for example for Java platform, could translate them to code working on JDO or other JEE solution.

ODRA also assumes bidirectional references, exceptions throwing and catching and procedure driven program execution. All these platform dependent features are introduced by ODRA model compiler.

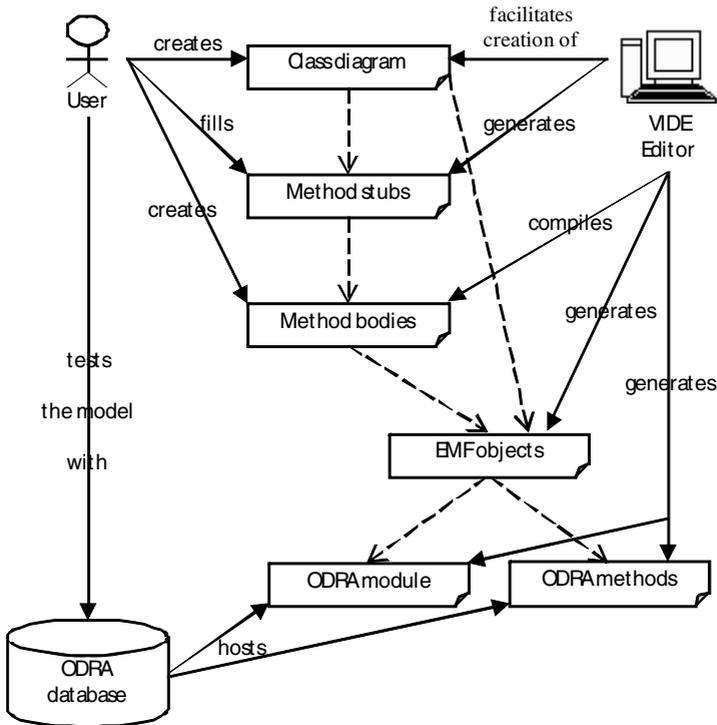


Fig. 2. ODRA IDE with integrated model compiler

4 Example Model and Its Execution

In this chapter we present a continuation of an example application shown in Fig 1. The structure of objects is already defined there. Now, we add some very high level behaviour inside procedure `getIncomeFromTo`. The language is a minimal textual representation for UML Actions and Activities supplied with OCL expressions. Its code may look as follows:

```

context Bookstore::Author.getIncomeFromTo body {

    authorsBooks : Bag[0..*]( Book );
    soldItemsFromTo : Bag[0..*]( SoldItem );

    authorsBooks insert global.books->
        select( b | b.author.name->includes(self.name) );

    soldItemsFromTo insert global.soldItems->select(
        (transactionDate >= fromDate) and
        (transactionDate <= toDate) );

    return soldItemsFromTo->select( i | global.books->
        includes(i.item) )->collect( i |
        i.amount * i.item.price * 0.05 ) -> sum();

}
    
```

This simple procedure was divided into three queries but obviously can be much more compressed. This rather declarative code is transformed automatically to UML PIM model, which is partially shown in Fig. 3. You may see an activity with all behavioural UML constructs inside. They are then transformed to code acceptable by ODRA database, sent there and executed. A user may analyse results, call procedures or execute queries on a resulting object database, which is implementing all structures and behaviours from UML model.

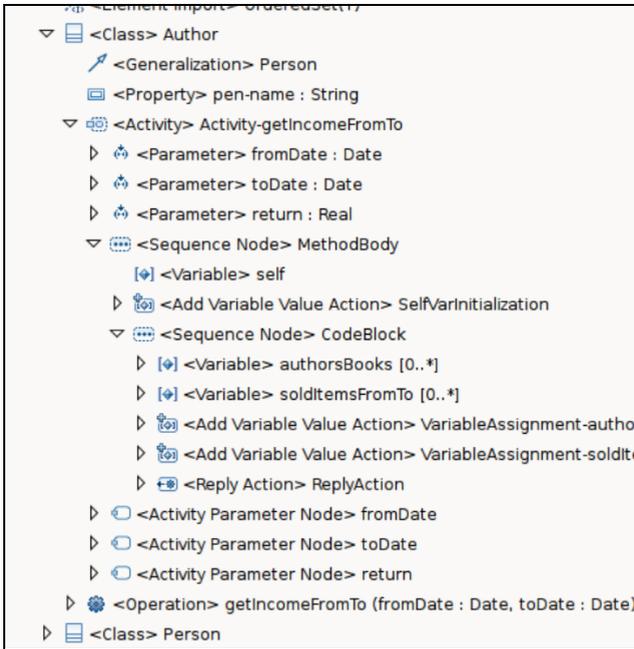


Fig. 3. A part of PIM repository for BookstoreSystem example

5 Conclusions

In this paper we investigated possibilities of using UML for executable modelling in the field of database applications. Due to its popularity, rich data model and the presence of modelling and model transformation tools, UML seems a promising mean of development of platform independent executable models. Applying that approach to applications based on a database brings a significant challenge for the development of model compiler solutions. Investigating the model compilers based on a fully object oriented system eases the development of model execution engine and will allow to find out, how much of the model compiler functionality is related to data model differences.

Apart from that, this area of application requires powerful constructs for building expressions. This brings the need of adapting and applying OCL as a query language for Executable UML PIMs. However, due to the shortcomings of existing versions of

those specifications, achieving the above outlined functionality needs to involve research and updates to future versions of the standard.

References

1. Lentner, M., Subieta, K.: ODRA: A Next Generation Object-Oriented Environment for Rapid Database Application Development. In: Ioannidis, Y., Novikov, B., Rachev, B. (eds.) ADBIS 2007. LNCS, vol. 4690, pp. 130–140. Springer, Heidelberg (2007)
2. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison Wesley, Reading (2002)
3. OMG: MDA Guide Version 1.0.1 (June 2003), <http://www.omg.org/cgi-bin/doc?omg/03-06-01>
4. OMG: Unified Modeling Language Specification (Superstructure and Infrastructure) Version 2.1.2 (November 2007), <http://www.omg.org/spec/UML/2.1.2>
5. OMG: Object Constraint Language OMG Available Specification Version 2.0 (May 2006), <http://www.omg.org/cgi-bin/doc?omg/06-05-01>
6. Shlaer, S., Mellor, S.J.: Object-Oriented Systems Analysis: Modelling the World in Data. Yourdon Press (1988)
7. Subieta, K., Kambayashi, Y., Leszczyłowski, J.: Procedures in Object-Oriented Query Languages. In: VLDB 1995, pp. 182–193 (1995)
8. Subieta, K.: Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL), Description of SBA and SBQL (2007), <http://www.sbql.pl>