Platform-independent programming of data-intensive applications using UML[1]

Grzegorz Falda[*], Piotr Habela[*], Krzysztof Kaczmarski[#], Krzysztof Stencel[+], Kazimierz Subieta[*]

[*] Polish-Japanese Institute of Information Technology, Warsaw, Poland
[#] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland
[+] Institute of Informatics Warsaw University, Warsaw, Poland
{gfalda, habela, stencel, subieta}@pjwstk.edu.pl, K.Kaczmarski@mini.pw.edu.pl

Abstract. The shift of development effort onto the model level, as postulated by MDA, provides an opportunity for establishing a set of modelling constructs that is more intuitive and homogeneous than its platform-specific counterparts. In this paper we confront UML with the needs specific for data-intensive applications and propose a seamlessly integrated platform-independent language with powerful querying capability, which would allow specifying a complete application behaviour. The proposal is aimed at high level of compliance with existing modelling standards – as such it is based on UML behavioural elements and on OCL for expressions. We present the motivation behind this approach, discuss the challenges implied by it, and indicate the role of the model runtime implementation.

Keywords: UML, executable modelling, query language, action language, MDA, database applications

1 Introduction

The approach of model-driven software development and the Model Driven Architecture (MDA) initiative in particular sketch the vision of the next big step in raising the level of abstraction and flexibility of programming tools. While any method that treats modelling activities as central can be considered "model-driven", the key expectation behind MDA is achieving a productivity gain through the automating of software construction based on models. This results in a significant shift of expectations regarding modelling constructs – from being merely a semi-formal mean for outlining and communicating project ideas, to machine-readable specification demanding precise semantics. Thus MDA creates a spectrum of model applications, which is often described using the following three categories:

[1] This work is supported by the European Commission 6-th Framework Programme, Project VIDE - VIsualize all moDel drivEn programming, IST-033606-STP

- *Sketches*, that represent the traditional use of UML and similar language as a help in understanding the problem and communicating ideas and solutions to other developers. Those kinds of models do not need to be complete nor fully formalized.
- *Blueprints*, that follow the traditional distinction between design and its realization – as in case of other engineering domains. The distinction of design and coding is maintained in terms of artifacts and is also reflected in assignment of those tasks to different groups of developers.
- *Executable models*, that require the presence of precise semantics and – by the automation of executable code production – blur the distinction between modelling and programming.

The last case is especially connected with the MDA initiative of the Object Management Group and has motivated significant restructuring and extension of the UML standard as experienced in its version 2 [1]. The most far-reaching variant of this vision is to replace existing programming languages with platform-independent modelling languages in majority of applications [2] (the same way the former once replaced assembly languages). This requires the presence of sophisticated transformation tools encapsulating the knowledge on particular target platform technologies, and depending on mature and widely adopted modelling standards – at least at the Platform Independent Model (PIM) level. In that case the application code produced would not be the subject of direct editing at all. The amount of work at the Platform Specific Model (PSM) level could also be reduced to minimum.

This vision is inherently challenging due to the transformations between heterogeneous high level languages involved (especially – if multi-tiered software and data processing are considered). This is probably why the idea of so highly automated MDA has not been extensively applied to the business applications so far [3]. At the same time however, applying strict MDA to that area seems especially compelling given the uniformity and reuse it could potentially provide there. This idea underlies the development of our platform-independent language aimed at the data-intense business application area, which is being created as one of the central elements of our project of visual modelling toolset VIDE (*Visualize all model-driven programming*).

In this paper we describe our approach to that problem, which is based on the following postulates:

- UML Structures unit seems to be rich and versatile enough to be considered as a foundation for a data model used in platform-independent development. A number of semantic details needs to be clarified to achieve that aim though.
- To make the model complete, the means of imperative programming need to be available at the PIM level. To raise the intuitiveness and productivity compared to the mainstream platform-specific technologies, the statements and queries should be integrated into a single language in a truly seamless way.
- Providing an execution engine for PIMs is needed as a reference implementation. It is also essential as a modelling tool component serving for platform-neutral model validation.
- Representing an application code in the form of standard metamodel instances and flexibly combining textual and visual notations for the behavioural modelling constructs introduced can provide a significant advantages over plain, purely textual languages.

The rest of the paper is organized as follows. Section 2 describes the expectations and concerns regarding the executable modelling approach and explains the motivation behind our approach. In Section 3 the UML 2 standard is presented from the point of view of precise specification of data-intensive applications. Section 4 describes the idea of a UML-based programming and query language and summarizes the current results in its development. In section 5 we outline the role of the language within a broader toolset and development process and indicate further challenges. Section 6 concludes.

2 Motivation

This should not be a surprise that pragmatic approaches to the problem outlined may depend on the programming notions known from existing programming and database languages. Specifying just a structural aspect of the model (using e.g. UML Class diagrams) is not sufficient if a high degree of code generation is the aim. Delegating the details of the behaviour specification to constraint definitions – as explained e.g. in [4] has the quality of the higher level of abstraction. However, realizing behavioural modelling this way in general is problematic from the point of view of complexity of model transformations. Moreover, in case of more complex behaviour this could be highly complicated and hard to accept by developers who are familiar with the traditional, imperative style of specification. Even at the side of imperative programming there is a dilemma regarding the selection of particular modelling notions to be supported in the executable models. What needs to be balanced is the ease of translation into other languages and making the language familiar for the developers knowing mainstream platform-specific languages, against the aim of achieving a higher level of abstraction and hiding the heterogeneity of type systems and programming paradigms.

When speaking about the current modelling standards (especially UML2 [1], MOF2 [5] and OCL2 [6]) and their development towards the vision of executable modelling, it is necessary to mention the critique this vision of model driven development faces – see e.g. [7]. While its motivation of raising the level of abstraction and controlling the level of details is recognised, the overall approach to dealing with complexity is considered problematic. A question of maintainability given the number of model representations is raised. There is even a doubt expressed if the MDA does not just push the complexity into later phases of the development process instead of reducing it (as the round-trip engineering requiring translation of lower-level notions into a higher level of abstraction is problematic). The size of the current UML specification is a concern, Especially, given that some areas essential for business applications are missing or weakly addressed there. This includes for example user interface specification, workflow/business process definition and data modelling. Moreover, the techniques that could support the stakeholders' involvement into the development process are also found missing from the language and not much visible in MDA in general [8]. The demand for a reference implementation and a human-readable operational semantics is emphasized – also for assuring the proper implementation of UML transformation tools [9].

There are also varying opinions on the role and usefulness of visual programming at the level of detail suggested by UML Actions and Activities units. In [8] an observation is made that most developers prefer text-based solutions for modelling and the focus of many tool vendors on diagram-based solutions is questioned. Fowler [10] and several other practitioners express concerns about visual programming as they indicate the diagrammatic way of code construction is incomparably slower. Indeed, majority of action languages in existence today [11, 12] are purely textual. However, if the difficulties related with visual coding at this level of granularity could be overcome, the visual notation may be advantageous under the following criteria:

- More control of the editing process, giving the possibility to assist the developer and to avoid some coding errors,
- More expressive distinction of different language constructs,
- Ability to more clearly visualize scopes and name visibilities – especially for complex expressions,
- Ability to incorporate domain specific user-defined symbols to make the code easier to follow e.g. during the validation by the domain experts.
- More potential for annotation and substitution texts / symbols use.

Given the above considerations and the implementation and transformation issues explained later, we chose to build the core of our VIDE language (called VIDE-L in the sequel) on the notions known from programming languages (expressed in terms of UML Actions and Structured Activities) rather than starting from flow-oriented activity models, state machines [11] or interactions. While this approach can be considered conservative from the point of view of the modellers' community, we note the following advantages compared to traditional programming languages:

- Depending on executable semantics for UML and the data model it assumes to support its adoption as a canonical model for various modelling and integration efforts.
- Capability of avoiding the "impedance mismatch" existing between database and programming languages in mainstream platform-specific technologies.
- Flexibility of code composition, validation, transformation and annotation gained through its representation in the model repository.
- Ease of switching syntactic options to offer an optimum combination of visual and textual notation for making the coding intuitive and productive for developers who know UML.

3 Standard base

What makes the UML a natural choice of a standard's base for the intended language is the popularity of the standard and its recent restructuring aimed precisely towards the executable modelling paradigm. Another fact that strengthens the position of this standard is recent development of the modelling tool implementation framework based on the UML 2.x metamodel [13], which may support uniform handling and exchange of UML models among various tools.

However, that selection itself is only a first step on the road for defining a platform-independent language for the area of application assumed. It is necessary to note the following factors:

- At the origin of the UML when it served rather only as an analysis and design language, some degree of ambiguity regarding the semantic details could be even considered desirable, as it leaves more freedom of applying its modelling constructs to varying technologies. The details of e.g. inheritance mechanism, parameter passing or object lifecycle could remain irrelevant on the level of abstraction assumed by those models or could be interpreted locally in terms of the technology of choice. This is not the case for precise PIM development, hence the efforts to provide UML with precise executable semantics specification [14].
- Moreover, the multi-purpose nature of UML implies that not all of its elements are capable of having a precise executable semantics defined. Moreover, from among the concepts having such capability, a subset should be selected to make the resulting language acceptably simple and suitable for its area of application (e.g. taking into account the needs of target platforms).

While it is impossible to provide a complete specification of the VIDE language here, in the rest of this section we try to present the most important decisions on selecting and detailing such a UML subset and describe motivations behind them.

The foundational problem (especially given the purpose of our language) is specifying the data definition language. We start from the complete UML Classes unit and perform the selection to achieve a data model that is expressive and universal but at the same time realistic in terms of its implementation and handling by the language statements. To this end, our motivation is to let the developer get rid of the object-relational mapping complexity. Hence we assume an object model with classes, static generalisation/specialisation supporting for substitutability and disallowing inheritance conflicts in terms of the multi-inheritance. Further work on achieving a greater flexibility of the inheritance hierarchy is aimed at exploiting the notion of dynamic inheritance in UML which we plan to realize in the form of dynamic object roles [15]. However, since it would require extending the behavioural part of the language either, we postpone this to the next version of the language.

The role of the UML Classes unit in VIDE can be summarized as follows. The current selection of UML notions used by the language seems to be the shortest way for achieving the expressive power of a programming language. The selection includes the core notions of UML Classes, Structured Activities and Actions units. Class model provides the structures that establish a context (in terms of features available to the behaviour: attributes, links, operations) under which a given behaviour is specified. It also provides a place for behaviour definition in the form of methods implementing operations of UML classes. Using VIDE for specifying behaviour in other contexts than class operations is being considered for the future (it seems to be feasible to adapt because of the strict UML compliance of VIDE constructs).

A feature that makes the language more distinct from popular OO programming languages is the realization of the Association notion. Compared to its complete definition a number of limitations have been introduced. Particularly, we skip the support for non-binary associations and association-classes. Although useful in conceptual modelling, they are problematic due to the complexity involved in their

implementation. The weak adoption of CORBA Relationship Service [16] that supported similar notions seems to support this observation. On the other hand, the language will automate the creation and referential integrity of updates of links that instantiate bi-directional associations. The current specification of UML provides big number of options for relationships among objects described by the Property notion: this includes unidirectional and bidirectional associations, plain attributes (Property not belonging to an Association) and the possibility to describe each property with the *aggregation* attribute distinguishing 3 aggregation kinds[2]. This may be considered redundant. Moreover, what blurs that distinction is allowing the UML notation to use the attribute and association notation virtually interchangeably. Those issues are considered important since our language demands the possibility of expressing nested data structures (like e.g. XML documents) – hence we need to distinguish several options for connecting two complex objects: plain bidirectional association, plain unidirectional association, bidirectional composite association, unidirectional composite association (the latter substitutable with non-primitive attribute).

There are also several considerations related to the differences in data modelling and accessing between programming languages and database environments. In programming languages the class definition does not usually determine the name of variables that will store its instances. On the other hand, this is quite natural for database schemas.

The aims and patterns of encapsulation are also rather different in case of database schema. While the current version supports just the visibility specification for classes' features, we consider future extending of the encapsulation mechanism using the notion of updateable views, which may require more precise declarations at the side of UML [1].

In contrast to programming languages like Java we do not assume the garbage collection of the objects expressed in our language; instead explicit *DestroyObjectAction* of UML is supported.

4 Language development

While "query language" is listed in the standard specification as one of the OCL possible purposes, the use of the language in VIDE-L is significantly different compared to the purpose OCL was originally designed for. So far, the expressions of OCL have been used mainly for constraint specification (where eventually were evaluated into Boolean values) or e.g. for calculating the initial values of attributes etc. In our case the area of application is much broader, since anywhere a programming construct needs to be extracted (e.g. selecting objects to be updated, removed, linked or passed as a parameter in an operation call), the expressions in OCL are used. This means the result of such expression does not necessarily need to be just an r-value.

[2] Note that the meaning of this attribute (i.e. if the owner of the property plays the "whole" or the "part" role) is unfortunately dependent on whether the property is a member of association or not).

VIDE-L language as a whole makes the similar simplifications in dealing with complex / primitive data and reference / value distinctions as e.g. Java. However, it achieves a bit higher expressiveness thanks to introducing dedicated statements for link updating and by supporting two parameter passing modes from among the ones assumed by UML: *in* and *inout*.

The use of queries as described above leads to a seamlessly integrated language, which is in contrast with embedding queries of a separate language as strings and dealing with resulting heterogeneity of type systems, syntaxes, binding phases etc. which is the issue e.g. in the ODMG standard [17] and Java-based specifications that evolved from it.

Those problems are to a big extent absent in our case, however, to achieve the goal we needed to deal with some overlap and heterogeneity resulted from this rather novel use of OCL and from the fact that UML and OCL specifications have been recently developed separately. Among those it is worth to note:

- Varying style of variable declarations: UML uses multiplicities and the ordering and uniqueness flags. OCL in turn does not support them and depends on the collection type constructors instead.
- Introducing the seamless support for OCL expressions for UML behaviour makes the following actions redundant: ReadStructuralFeatureAction, ReadSelfAction, ReadExtentAction, ReadLinkAction etc.

Apart from the language semantics, also its syntax plays an important role for the productivity and ease of its adoption. It is necessary to note that the decisions on the concrete syntax that UML2 specification leaves open for developers is not necessarily just a plain selection of the list of visual or textual symbols. The elements not having a concrete syntax specified (which refers roughly to Actions and Structured Activities units) are fairly universal and fine-grained. This encourages the designers of particular action languages to consider creation of various higher level language constructs that are useful for the intended area of application and whose mapping onto UML element instances is not necessarily “one-to-one”.

Indeed, although we tried to provide the statements that rather directly represent respective UML Actions and Activities primitives, a number of useful programming language constructs required a more complex mapping. Those cases include:

- Reusing generic Structured Activities elements to provide useful statements for loops and conditional instructions. For example, ConditionalNode does not provide dedicated construct for “else” or “otherwise” clauses. On the other hand, we do not take advantage of the ConditionalNode’s capability of providing results (i.e. serving as expressions).
- Providing useful shortcuts like the +=, -=, *= and /= assignments is especially useful when considering iterative processing of results provided by expressions over data sources. Those shortcuts also miss dedicated support from UML Actions and while it is of course easy to construct a metamodel instance of the desired semantics, the reverse mapping into a code demands for annotation or stereotype to ease it.
- Macroscopic updates. While (which is also in the spirit of UML behaviour) we avoid macroscopic updates (e.g. updating many objects with a single statement without resorting to an iterating instruction), we found the following exceptions useful. First, we allow collections to be the input of object removal statement.

Second, we allow to assign a collection to a multi-valued attribute or variable instead of the need of inserting its elements one by one. Due to the constraints imposed by UML compliance, this required an implicit use of iterative construct (that is, the ExpansionRegion).

While not providing a formal specification of the language here, we present below several illustrative code examples referring to the schema defined by the class diagram in Fig. 1, which is assumed to be defined inside a package named *Students*.

The first example illustrates a simple method of a pure query nature (i.e. having no side effects). Those kinds of methods could be called inside pure OCL statements. This distinction is possible to maintain in VIDE-L, as the calls of methods marked as having side-effects can be delegated to the CallOperationAction rather than handled at the OCL side. However, we currently do not enforce it in our language. Note also the OCL-style **context** declaration which specifies to which operation in the class model the given method body refers to. In the final version of our prototype the modelling environment will provide more assistance for this, so this header will not need to be directly used by the programmer.

```
context Students::Person.getFullName() : String
body {
  return firstName+' '+lastName;
}
```

The second example shows a more complex updating method, which uses link navigation and performs iterative updates of the objects selected by an OCL expression.

```
context Students::Department.assignScholarship(
 in amount : Integer, in noOfStudents : Integer) body {
  students->sortedBy(s |
        -s.calcAvgGrade()).subSequence(1,noOfStudents)
    foreach { s |
           if s.scholarship->size()=0
             then s.scholarship insert amount;
             else s.scholarship += amount;
           endif

         }

}
```

The third example illustrates the link manipulation that moves employees to another department (the reverse links will be maintained automatically).

```
Department->allInstances()->select(name='SE').employs
foreach { f |
  f unlink worksAt;
  f link worksAt
   to Department->allInstances()->select(name ='IS');
}
```

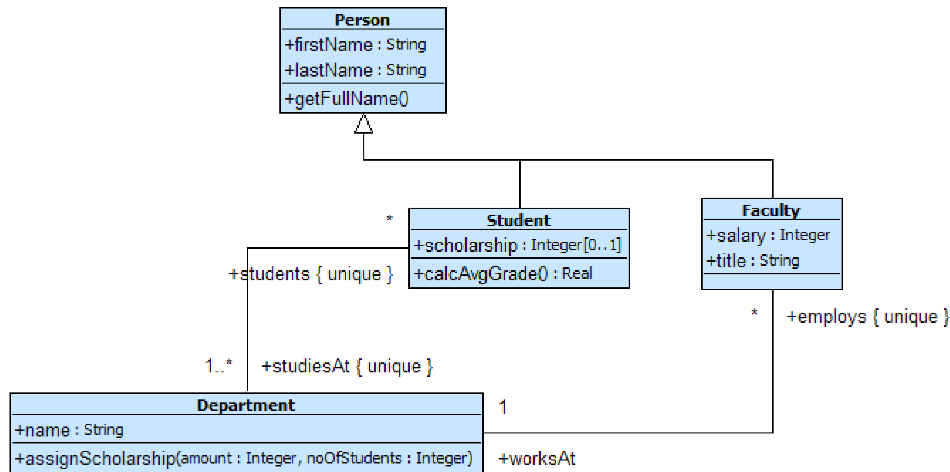

Fig. 1. Exemplary schema for code samples (the package name is "Students")

The fourth example shows an ad-hoc query which is in this case a pure OCL.

```
Department->allInstances()->select(name='IS')
  ->collect(d |
      employs.title->asSet()->collect(t |
        Tuple{ title = t, avgSal =
          d.employs->select(title=t).salary->avg()}))
```

Note that two of those examples depend on the class extent retrieval. While this can be natural for some flavours of object schemas (like e.g. in ODMG), for typical cases we assume a presence of an object that will be pre-existing with respect to the application execution (rather than explicitly instantiated later) to provide an entry point to the application. For this purpose we have introduced the class stereotype «Module».

As can be seen from the above diagram and code samples, the textual syntax can be considered a bit eclectic, as it is influenced by three trends: UML type and multiplicity declarations, OCL with its specific syntax which influenced also the VIDE-L statements to take a more postfix-style syntactic patterns, plus some solutions coming from Java as the most popular general-purpose programming language. The positive aspect is that the syntax seems well suited for extensive contextual support when coding, which is to be provided by type checking mechanisms. This can be especially visible where query expressions are involved. Compare the marked steps of the code of example 2:

```
/*1*/students /*2*/->sortedBy(s |
      -s.calcAvgGrade()).subSequence(1,noOfStudents)
  foreach { s |
         if s.scholarship /*3*/->size()=0
           then s.scholarship insert amount;
           else s.scholarship += amount;
         endif }
```

with analogous code expressed with a syntax drawn from ODMG OQL [17] and popular programming languages:

```
foreach ( select s from students s
          order by s.calcAvgGrade()
          desc)[1..noOfStudents] as std {
            if ( not exists( s.scholarship )  )
              s.scholarship insert amount;
            else s.scholarship += amount;
}
```

It can be observed that the way the OCL syntax is arranged makes it easier and more natural to provide hints[3] than in case of the select-from-where pattern. At point 1 a list of names visible in the scope (starting from the most local ones) and statements could be presented to support that step of code creation. Similarly, at point 2 the list of available collection information can be presented (since the expression *students* returns a collection of objects) as well as the properties of the Student object (because OCL allows for building path expressions in 1:n direction). A slighter advantage in terms of the contextual hits can be achieved at point 3, where the selection of proper operator (OCL operation *size()*) may be performed from among of few choices determined by the context of expression *s.scholarship*.

The visual notation considerations are rather outside the scope of this paper. We just note the dilemma between choosing the traditional diagrammatic style of syntax (and aiming at "keyboard-less programming") or staying closer to textual style of coding though supporting it with visualization. The textual coding of this level of granularity is rather predominant. We are aware of only one action language depending on visual notation – namely Scrall [19] – which deals with a similar problem in terms of combining the visual and textual notation. The similarities with our language include:

- The idea of controlling the level of detail by collapsing and expanding code elements and resorting to textual code where it is more suitable.
- Considering data processing as the purpose of the language.

The following differences can be indicated:

- Scrall assumes relational data. It provides some high level operators, but does not provide a complete query language functionality comparable with OCL.
- Scrall supports the flow-style of behaviour specification which is made possible by the visual notation. VIDE currently does not directly cover this powerful feature, but the compliance with UML provides the capability of achieving it in future extensions by embedding VIDE-L code inside the diagrams supporting UML Complete Activities.

[3] This applies to some extent also to XQuery language [18].

5 Challenges of the development process

Apart from creating a standard-compliant language of adequate expressive power, MDA solutions need to address the problem of model transformations to automate code creation. Translating between high-level languages usually involves big complexity. Examples of the potential problems that need to be faced in that area include:

- Translating PIM-defined application logic onto the multi-tier solutions of target platforms. The number of possible options in such translations complicates the process and / or undermines the idea of full platform-independence of the main model.
- Introducing a platform-independent specification of the presentation layer which is of high importance in various business applications.
- Dealing with data management – including schema definition on the target platform and hiding the "impedance mismatch" between today's database and programming languages behind a uniform platform-independent modelling language. When dealing with the code to be handled by a DBMS it is not only necessary to preserve its original semantics, but also to guarantee that the opportunities for optimization will not be lost in the course of translation, which is essential for achieving acceptable performance and resource consumption in data-intensive applications.

In some applications, an alternative to those complex translations could be the idea of "model driven runtime" [20]. This means that a platform is available that is capable of directly executing models (e.g. represented in the form of the UML metamodel instance), so running an application does not require transformation to some other programming and / or query language. The cited paper arguments that the difference between having different platform-specific models derived and having many different model runtimes deployed is less substantial than it may appear. The described solution deals with simpler scenarios of application development (some Web applications are given as an example), where no complex application logic occurs and the presentation layer is closely driven by the schema of underlying database. Moreover, the runtime described in that paper deals with different kind of behavioural models as it "interprets OCL-annotated class diagrams and state machines".

Of course this solution is not always acceptable since the use of existing platform specific tools and environments is required by customers for the applications being created. That's why VIDE assumes developing respective model compilers.

However, we have provided a runtime for direct execution of models, as we have found it important for the following reasons:

- Current standardization efforts of UML should be backed with a reference implementation to verify the consistency of the language and to disambiguate its semantics through an operational definition.
- Availability of the engine that would allow direct execution of models seems to be a feature of primary importance for model-driven development tools, as a mean of model simulation (also in terms of tracking and debugging particular elements of the application at the PIM level and in terms of PIM artifacts). Since our current runtime engine provides rather straightforward implementation for particular

model constructs compared to typical target platforms, it is an interesting option for the future development of complete model simulation and debugging environment.

Among other challenges to be faced by VIDE toolset is the integration with business modelling. This is important to meet the demand for business-process driven software development approaches and the Service Oriented Architecture viewpoint on the applications. A similar, but separate problem is an attempt to improve the business stakeholders' involvement into modelling and application prototyping.

The above considerations set the following assumptions for the current work on the VIDE project:

- UML compliant PIM, provided with the means and level of precision of a programming language becomes the central artefact of the software construction.
- Model execution capability allows to validate the system functionality directly from the tool (i.e. without the steps of explicit code generation and its deployment).
- Appropriate elements of model behaviour may be distinguished as externally available service interfaces and equipped with a complete Web service descriptions for the purpose of model's direct execution.
- For the scenarios that allow it, the model may be rather directly deployed in the flavour of a MDR using its execution engine (purely object-oriented database system prototype).
- If creating application functionality on the Java platform is the aim, a model compiler (currently under design) will be used to generate Java code defining the application logic and using data persistency through the JDO interface [21].
- At the side of initial phases of a software development process, a significant amount of work has been allocated to describe gradual, incremental transition from informal requirements set and computation independent model towards precise PIM.

6 Conclusions and future work

The aim of the research outlined in this paper can be considered challenging for several reasons. The first challenge is to provide adequate and advantageous means of software specification, taking into account new kinds of user profiles assumed by the MDA development process. It has to recognize and properly balance the needs of such user groups as modellers familiar with CASE tools, programmers familiar with traditional programming and query languages, and non-IT stakeholders seeking for model accessibility. Another challenge is the need of alignment with modelling standards on one side and target platform technologies at the other side. Those considerations draw two important goals for the next step of our research:

1. Completing the existing textual prototype with the implementation of selected concepts of visual notation and gaining feedback from users.
2. Investigating the possibilities of developing model compilers from the kind of modelling constructs VIDE employs, onto the implementation technologies used in the industry.

References

1. Object Management Group: Unified Modeling Language: Superstructure version 2.1.1, February 2007. formal/2007-02-05.
2. S.J.Mellor, K.Scott, A.Uhl, D.Weise: MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley 2004.
3. A.McNeile: MDA: The Vision with the Hole?, www.metamaxim.com 2003.
4. J.Warmer, A.Kleppe: Object Constraint Language, The: Getting Your Models Ready for MDA. Addison Wesley 2003.
5. Object Management Group: Meta Object Facility (MOF) Core Specification version 2.0, January 2006. formal/06-01-01.
6. Object Management Group: Object Constraint Language version 2.0, May 2006. formal/06-05-01.
7. B.Hailpern, P.Tarr: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal: Model-Driven Software Development. Volume 45, Number 3, 2006.
8. S.W.Ambler: A Roadmap for Agile MDA. Ambysoft, upd. 2007, http://www.agilemodeling.com/essays/agileMDA.htm
9. D.A.Thomas: MDA: Revenge of the Modelers or UML Utopia?, IEEE Software 21, No. 3, 15–17 (May/June 2004).
10. M.Fowler: UML as Programming Language, 2003. http://www.martinfowler.com/bliki/UmlAsProgrammingLanguage.html
11. S.J.Mellor, M.J.Balcer: Executable UML: A Foundation for Model-Driven Architecture Addison Wesley 2002.
12. I.Wilkie, A.King, M.Clarke, C.Weaver, C.Rastrick, P.Francis: UML ASL Reference Guide ASL Language Level 2.5 Manual Revision D, Kennedy Carter Limited 2003 http://www.omg.org/docs/ad/03-03-12.pdf
13. Eclipse Modeling Project, Model Development Tools. Eclipse Foundation http://www.eclipse.org/modeling/mdt/
14. Object Management Group: Semantics of a Foundational Subset for Executable UML Models. Request For Proposal. ad/2005-04-02.
15. A.Jodłowski, P.Habela, J.Płodzień, K.Subieta: Dynamic Object Roles - Adjusting the Notion for Flexible Modeling. Proc. of the International Database Engineering and Application Symposium (IDEAS), IEEE Computer Society, Coimbra, Portugal, 2004, pp. 449-456.
16. Object Management Group: Relationship Service Specification version 1.0. April 2000. formal/00-06-24.
17. Object Data Management Group, The Object Database Standard ODMG, Release 3.0, R.G.G. Cattel, D.K. Barry, Eds., Morgan Kaufmann, 2000.
18. World Wide Web Consortium: XQuery 1.0: An XML Query Language. W3C Recommendation 23 January 2007. http://www.w3.org/TR/xquery/
19. L.Starr: Starr's Concise Relational Action Language version 1.0. August 2003. http://www.modelint.com/downloads/mint.scrall.tn.1.pdf
20. J.Pleumann, S.Haustein: A Model-Driven Runtime Environment for Web Applications. The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings. LNCS 2863. Springer 2003, pp. 190-204.
21. Java Data Objects Expert Group: Java™ Data Objects 2.0. JSR 243 Final 23 February 2006. http://java.sun.com/javaee/technologies/jdo/