

Documenting legacy applications in the context of model-driven development

Radosław Adamus, Marcin Daczkowski, Piotr Habela, Krzysztof Kaczmarek

Abstract. Having a precise model that specifies an application in platform-independent manner, as postulated by the Model Driven Architecture initiative, allows to protect the investment in design and to automate further tasks on producing executable software. This however usually assumes that such models are created from scratch at the time of the original development of a given software. Incorporating pre-existing applications into a model-driven development requires to represent them with appropriate models. Reverse engineering of complete applications is problematic: not only because the number of different legacy platforms, but especially because the inherent complexity of bidirectional mappings such reverse engineering would require. In this paper we summarize a more lightweight approach to documenting and integrating such resources using wrappers and limiting the scope of modelling to interfaces. Wrappers for web service interoperability, XML data exchange and relational database integration are described.

Introduction

The Model Driven Architecture initiative of Object Management Group [MDA] assumes the use of standardized modelling languages to assure familiar notation and uniform metamodel that would ease the traceability and model transformations. This vision is currently only partially realized – especially if the area of business applications that involve persistent data, being the focus of the VIDE project [VIDE], is considered. For such applications behaviour is relatively seldom and selectively modelled and – if so – serves a role of a communication device on the conceptual level rather than a development artifact. VIDE intends to fill this gap by developing and implementing a UML 2.1 [OMG07] compliant programming language based on UML Actions and Structured Activities that allows to specify complete application behaviour. This level of detail of a model allows to generate from it executable applications onto target platform.

The use cases for VIDE language [VIDE08a] in most cases assume the presence of a model (both in case of building a new application or in case of developing it from scratch). However, for software implemented before the time of introducing respective model-driven tooling and development process lacks such models. Hence some simple solutions have been developed in the course of project to wrap such resources and make them interoperable with models and new applications created from them. This approach is obviously limited compared to reverse engineering, as it does not allow to rebuild the internals of legacy applications. Nevertheless, it allows to uni-

formly describe their behaviour in the form of service interfaces, as well as to fully represent their relational data sources.

From the point of view followed by VIDE project and presented in this paper, we may assume legacy software to be all target platform artifacts that lacks their representation in the platform-independent model (PIM). The paper is organised as follows. Section 2 describes the VIDE language that would be used to specify code dealing with legacy resources. In section 4 the approach to integrate behaviour based on web service interfaces is described, while section 5 describes the relational database and XML data wrappers. Section 6 concludes.

VIDE language

VIDE language for PIM behaviour representation has been designed as visual and textual programming language based on UML metamodel units of Structured Activities and Actions. The details of its concrete syntaxes are not discussed in this paper as they are not relevant for the legacy integration. The abovementioned selection of behavioural constructs resulted in rather traditional set of VIDE language constructs, resembling a general purpose programming language.

- The choice of UML as the base of the language provides an object-oriented data model and brings the following specific design assumptions:
- Modelling of the structural aspect of applications using class and package diagrams that constitute the most common and mature part of UML specification.
- Availability of UML extension mechanism involving profiles and stereotypes that can represent specific notions related with integration and may provide additional details essential for executable semantics.
- Connection behaviour and structural models through operations inside class model that is setup familiar for developers used to platform-specific programming languages.
- Use of fine granularity behavioural constructs belonging to Activities unit, which potentially allows to embed the language inside activity or state diagrams.

An important observation is that the development of data-intensive applications demands data retrieval constructs of a higher level than the one offered by UML Actions. To address it and at the same time, to keep the solution standard-compliant, OMG OCL specification was adopted for the expression part of the language. Hence, the data read actions are replaced in VIDE by OCL expressions, which results with a programming language having expression constructs of expressive power analogous to the one of common query languages.

With the above mentioned design decisions, the challenges for integrating legacy applications with VIDE can be summarised as follows:

- Representing consumed web service operations in a way uniform with regular operations inside UML model and containing necessary data to generate target platform code that would successfully invoke such services.
- Mapping of XML Schema onto UML type system so us to allow UML modelling of web service interfaces as well as XML data structures to be consumed by application under development.

- Mapping of SQL data definition language into UML class model in order to make the consumed relational data manipulable by UML / OCL behavioural constructs.
- Handling the above modelling solutions in model execution mechanism and target platform code generators.

The last of abovementioned issues – model execution and code generation – will be discussed on the example of the experimental ODBMS platform ODRA [ODRA] that was one of the sample target platforms in the course of VIDE project.

Integrating behaviour

The purpose of including web services support in VIDE is to allow its users to take advantage of service oriented architectures and to reuse existing services inside Line-of-Business applications.

Functional aspect

Web service implements directed communication between client and server. Hence, we distinguish two functional aspects of their usage. web services can be consumed in order to include a remotely accessible functionalities into VIDE systems (i.e. currency exchange). Certain model elements can be published as web service to allow external connectivity with them. In that case (remote) clients can trigger execution of VIDE code from the outside.

Concepts mapping

In this section we describe how WSDL and VIDE models are mapped to each other. Both for web services consuming and publishing scenarios we provide detailed information regarding extended constructs introduced to VIDE to handle web service integration properly.

The de-facto standard for web services description is WSDL 1.1 [W3C01]. WSDL contracts consist of two parts: abstract and concrete. The first one contains interface and types involved in communication details. The second part describes implementation details such as supported messaging and transport protocols.

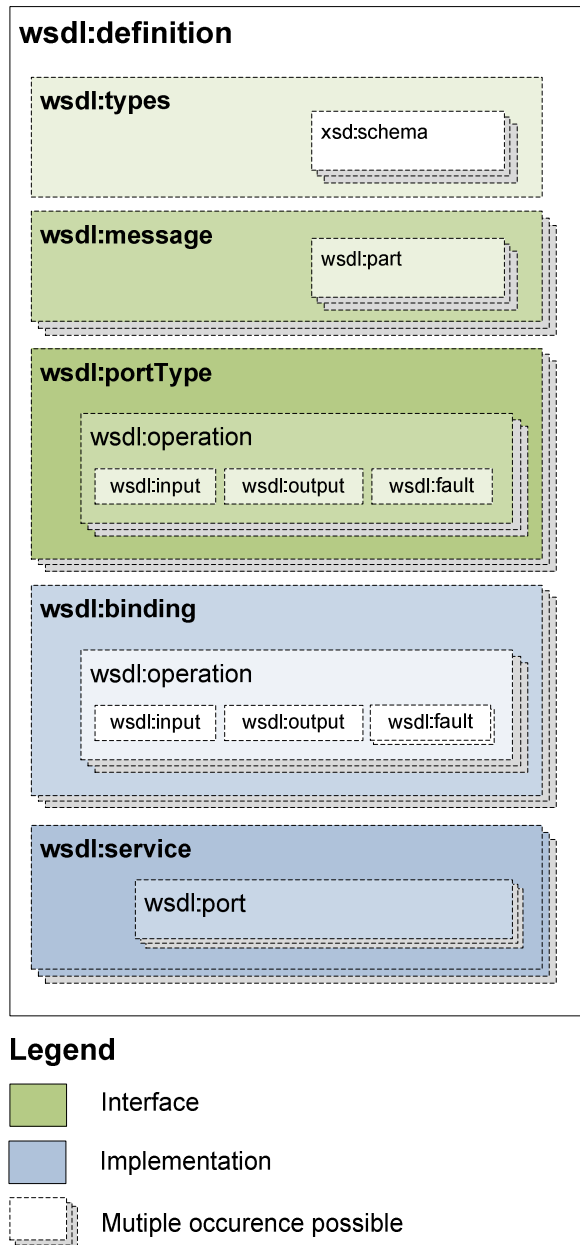


Fig. 1. WSDL 1.1 contract diagram

VIDE web services integration was based on same (as in WSDL) separation between abstract and concrete part. Mapping between WSDL abstract part and VIDE model is described in the next subsection.

WSDL definition to VIDE mapping

Publishing and consuming web services can be considered as symmetrical tasks. In mapping from WSDL to VIDE model and from VIDE to WSDL we try to follow that symmetry. On PIM level only abstract WSDL part is taken into account. This includes `types` and (specified) `portType` section. Types are described as XML Schema and are imported to the system to provide strongly typed access to a web service. Interface on the other hand is imported as specialized class to allow transparent remote methods invocations.

Since one WSDL contract maps to many VIDE entities we need to use container for them. For that purpose standard UML package is used. Container content is auto-generated and hence should not be edited by a user. Because it is impossible to forbid editing in a generic way on model level it is only an advice for VIDE users. They should not tweak such imported services manually in any way (but the restriction is not forced).

`PortType` is imported into the container package as a class marked with «`consumedService`» stereotype. Only one such element is allowed to be included in the package. The web service proxy class has no attributes but it contains an operation for each web method from target web service. Operation parameters are based on input/output web method messages and `raisedExceptions` association is based on WSDL web method faults. Types described in WSDL are imported as well. The mapping is handled by dedicated component. The types import produces UML types with optional stereotypes assigned to reflect their exact XML Schema meaning.

Stereotype «Consumedservice»

Designates that class will be a proxy to remote web service conforming to certain WSDL contract. Its operations are associated to remote web method calls of given web service. Exact shape of this stereotype will be specified based on the design decisions in model compilers development in VIDE.

Generalizations

Class

During `types` and `portType` import to model, the following naming conventions are used:

WSDL	VIDE
(encoded) target namespace	containing package names
Port type	(proxy) class name
operations names	operations names

Generic web services compilation schema notes

Web services are regular VIDE model elements marked with certain stereotypes. However because of their remote behavior, they need to be treated in a special way during compilation.

For example consumed service is visible in editor as normal class (and set of associated types generated from WSDL) and hence can be called from any other package. However compiler need to be aware of that fact and compile it using dedicated proce-

dure. All calls to such remote proxy can still be compiled in standard way. Possible problem here is to maintain precise control over the way consumed service is realized inside a target platform.

Importing service to model means generating a static proxy stub packaged with all necessary types. Same procedure is usually done on target platforms level. This is sufficient if system creation starts from PSM and there are no already collected web service information from PIM level. However in our case where such data already exists, it should not be dismissed (i.e. by deciphering again all information from WSDL contract). Doing so affirms that all depended (on service proxy) model elements will have their calls working correctly. Recreating proxy from scratch can lead to inconsistencies between what VIDE user sees and what is being executed (hence errors will be less descriptive and debugging more problematic).

We do not force compiler authors to follow tight mapping path here, but only want to make them aware of possible implications of proxy regeneration. For certain use cases, like in case of the scope of code generation solutions implemented in the course of VIDE project, this may be a sufficient solution.

Integrating data

Integration of relational data

The documenting logical structure of legacy data in VIDE supports relational model. To perform the retrieval of relational database schema into our model, we adopt the functionality of relational schema importer, which is a generic tool being developed as a part of ODRA toolset. It allows us to perform a kind of reverse engineering by representing the imported schema in terms of UML model. The way the schema import is implemented is not relevant for the final user, however, the platform specific details being set here for the sake of this reverse engineering can be also directly useful for interacting with the relational data source from the executable model. The latter step is currently realized by ODRA2RDBMS model execution runtime; however it may potentially be realized also other way, if supported by respective model compiler. For this reason and to allow a uniform conceptual modelling, the mechanism is not opaque. Hence, in the further part of the document, we describe the way RDBMS schema elements are represented in VIDE. The figure below depicts the overall architecture of the mentioned ODRA tools.

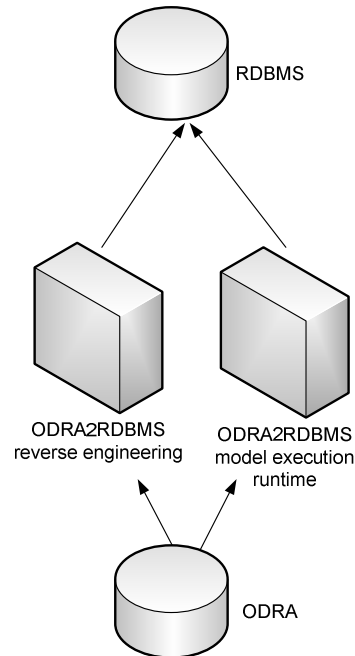


Fig. 2. The overall architecture of the ODRA wrapper

To sum up, the purpose of the ODRA tools for relational database access in the VIDE project is:

- To enable the description of existing relational data sources in the VIDE model. This can be achieved with use of the abovementioned reverse engineering tool. This way relational schemata can be introduced into the VIDE model.
- To enable transparent querying and updating of the legacy relational data with VIDE language statements. This can be achieved with use of ODRA generic model execution engine.

The sequence of activities needed to import the schema into the model and use the consumed data in the course of model execution can be summarised as follows:

1. Developer provides VIDE RDB import wizard with connection details stored in a .properties file including database name, URL, authentication data and database driver.
2. Database is queried for its schema and its representation in the form of XML file is presented to the developer.
3. After accepting this step by developer, the tool produces a UML package representing the database schema and assigns it a stereotype «*rdbWrapper*». For each table a class is generated. The following mapping rules hold:
 - a. The class attribute that corresponds to a primary key column is additionally marked with «*primaryKey*» stereotype.
 - b. The nullability of an attribute is represented by the [0..1] cardinality of the corresponding class attribute.

- c. Further integrity constraints (e.g. kind of the primary key, foreign keys, etc.) are currently not represented.
- d. The type mappings are presented in the

SQL	VIDE
varchar varchar2 char text memo Clob	String
integer int int2 int4 int8 serial smallint bigint byte Serial	Integer
number float real numeric Decimal	Real
bool boolean Bit	Boolean
date timestamp	Date

The figure below shows a sample package with generated model based on sample relational data. The package import relationship leading to that package makes it possible to specify in the other package a code that reads and updates relational data in a way fully uniform with the data structures specified from scratch using UML and VIDE. To retrieve a content of a given relational table, an expression representing class extent (i.e. `ClassName.allInstances()`) is used.

4. When the model including the representation of the consumed data source and the behavior that uses it is transferred to the model execution engine, the behaviour, including the data source consumption supported by relational wrapper, can be tested by actual running it.

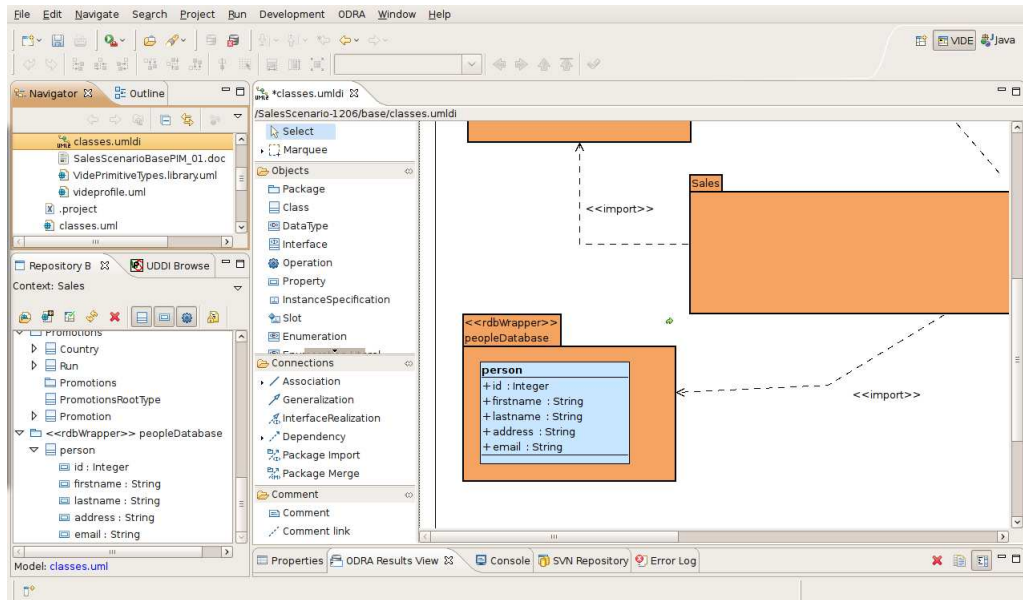


Fig. 3. Structure of a relational database seen in Repository Browser and diagram view

Importing XML data

XSD Import procedure lets users to interact with schemes of XML data. In this case the interaction mechanism is different than for relational data source. Here, instead of wrapping existing XML document and mapping queries and updates to it, the importer just copies the data into ODRA engines storage. On the other hand, handling of XML Schema description is analogous as in case of relational database schema: it is represented using UML to constitute (usually) a separate package in UML / VIDE model. The integration of XML data is performed in two layers – using the OMG terminology the M1 layer (schema to model) and M0 layer (document to user objects). The first step required is importing an XML Schema specification of the data structure to be used.

The developer is only required to provide the name of UML package that is going to contain model definition from the XML structure. The destination package will contain a «module» class which will represent Document Root of XSD Schema file. The «module» stereotype is used in VIDE as an entry point to the application and a container for applications persistent data and externally available behaviour. More details can be found e.g. in the VIDE Cookbook document [VIDE08b]. A package may only contain one module class. The import is performed via XML Schema to Ecore simplified conversion done by Eclipse plug-ins. More information on this is available in XML Schema to Ecore Mapping Documentation [Eclipse2004]. The mapping is very rich in terms of underlying data model – the model of XML Schema is additionally extended e.g. with the declarations that allow representing bidirectional associations.

After the schema is imported, user can feed data from XML document to the model execution engine so that the model can run on sample data. The XML import mechanism can be also used for prototyping of models – that is, developing the structure using XML Schema, importing it into UML and further elaborate e.g. by adding operations to the classes generated.

Similarly like in case of relational data, the imported XML can be handled by VIDE in an uniform way – that is, as other UML class instances.

Conclusions

In this paper the solutions for incorporating legacy applications and data into model driven development used in VIDE project have been outlined. Although they do not match the flexibility the full reverse engineering could provide, they allow to make pre existing resources accessible to the application logic under development with VIDE tooling. In the course of development of the wrappers described, the following directions of future extensions of those tools have been identified:

- Raising a level of abstraction in the relational database adapter by representing foreign key constraints with UML associations and supporting them appropriately in the model execution runtime.
- Adding ability not only to import but also to define and modify RDB schemas using UML class diagrams and respective profile.
- Extending ODRA platform with dynamic integration with XML resource (based on XML file mounting as a data store) could allow for all operations on XML data and also could add dynamic recognition of XML data modifications.

References

- [Eclipse2004] XML Schema to Ecore Mapping Documentation. <http://www.eclipse.org/modeling/emf/docs/overviews/XMLSchemaToEcoreMapping.pdf>
- [MDA] OMG Model Driven Architecture <http://www.omg.org/mda/>
- [ODRA] Object Database for Rapid Application development (ODRA) project website <http://www.sbql.pl>
- [OMG07] Object Management Group: Unified Modeling Language: Superstructure version 2.1.1, February 2007. www.omg.org/cgi-bin/doc?formal/2007-02-05
- [VIDE] VIDE Project website <http://www.vide-ist.eu>
- [VIDE08a] VIDE – Industrial Use Cases <http://www.vide-ist.eu/reflib/usecases.html>
- [VIDE08b] VIDE Cookbook <http://www.vide-ist.eu/reflib/cookbook.html>
- [W3C01] World Wide Web Consortium: Web Services Description Language (WSDL) 1.1 (<http://www.w3.org/TR/wsdl>)