Submission for Transformation Tool Contest 2011: Sample URDAD model

Fritz Solms, Craig Edwards and Stefan Gruner
Dept. of Comp. Science, University of Pretoria, email: fritz@solms.co.za, sgruner@cs.up.ac.za

March 8, 2011

Abstract

The Use-Case, Responsibility-Driven Analysis and Design (URDAD) method, is a method used by requirements specialists (typically business analysts) to perform technology neutral, services oriented analysis and business process design. The output of the URDAD method is an URDAD model containing the requirements and technology neutral process design which was historically encoded in UML. In order to improve the productivity and quality of technology neutral analysis and design a formal metamodel and a domain specific language with a concrete text syntax have been developed. The submission provides the URDAD metamodel as well as an example model for a typical business use case. The modl is technology neutral and does not prescribe the implementation technologies and architecture. The transformation should preferably target a typical enterprise systems architecture like the Java Enterprise Edition (JavaEE) reference architecture or a Services Oriented Architecture.

Contents

1 Contextualization

URDAD, the *Use-Case, Responsibility-Driven Analysis and Design* [3, 2, 1] method, is a method used by requirements engineers (typically business analysts) to perform technology neutral, services oriented analysis and business process design. The resultant analysis and business process design models are intended to be augmented by technical experts to formalize conditions and constraints.

The output of the URDAD method is an URDAD model containing the requirements and technology neutral process design. Historically the URDAD model has been encoded in UML. Such an encoding suffers from imprecise semantics, excessive model complexity and high model defect rates caused by the size and complexity of UML and the high level of discipline required to encode an URDAD model in UML.

To address these issues an metamodel introducing only the URDAD model features has been developed. This metamodel forms the basis for a *Domain-Specific Language* (DSL) for URDAD. The model is meant to be populated through the use of concrete text and diagrammatic syntaxes defined for the URDAD DSL. Currently a concrete EMFText-based syntax is available.

An URDAD model recursively assembles higher level services from lower level services. The leaf services are services sourced from the environment. These include services provided by the implementation language/technologies, persistence services and services sourced from external systems. For each service the the URDAD model contains

- the *service contract* with data structure specification for the request (input) and result (output) objects as well as pre- and post-conditions used for functional testing, and quality requirements used for testing non-functional requirements,
- the *service* specification specifying the functional requirements (i.e. lower level services from which this higher level service is assembled), and the actual orchestration of the process for the service across these lower level services. The outcome of the service is either the result object (return value), or the notification that the service is not provided due to a particular pre-condition not having been met by raising the exception for that pre-condition.

1.1 The URDAD meta-model

The URDAD metamodel specifies the model structure for the input model of the transformation. It is encoded in Ecore and is included in the artefacts provided to the mapping teams. Figure 1 shows a diagrammatic representation of the metamodel.

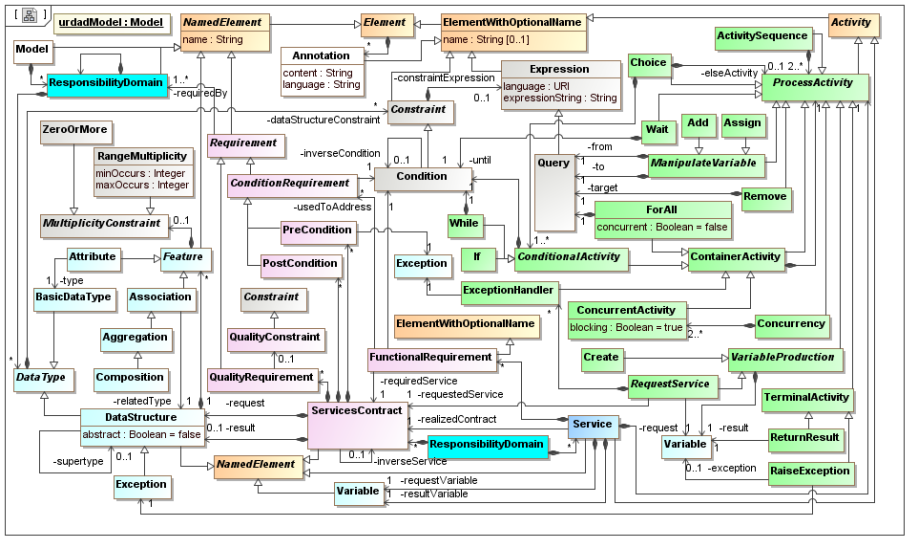

Figure 1: A diagrammatic representation of the URDAD metamodel

2 Provided artefacts

The following artefacts are provided for the case study:

1. this document,
2. the Ecore file for the URDAD metamodel,
3. a diagrammatic representation of the URDAD metamodel to speed up the understanding of the metamodel
4. a concrete EMF-Text based text syntax for the URDAD DSL
5. the use case for the case study in the form of both,
   - an XMI file, and
   - in the concrete EMF-Text based text syntax.

Upon request we can also provided

- An editor for the concrete URDAD text syntax.
- Service including process specifications for the lower level granularity services if the team wants to perform an implementation mapping for some of these.
- Diagrammatic representations of data structures, lower level functional requirements and processes.
- Systematic evaluation guidelines will be provided.

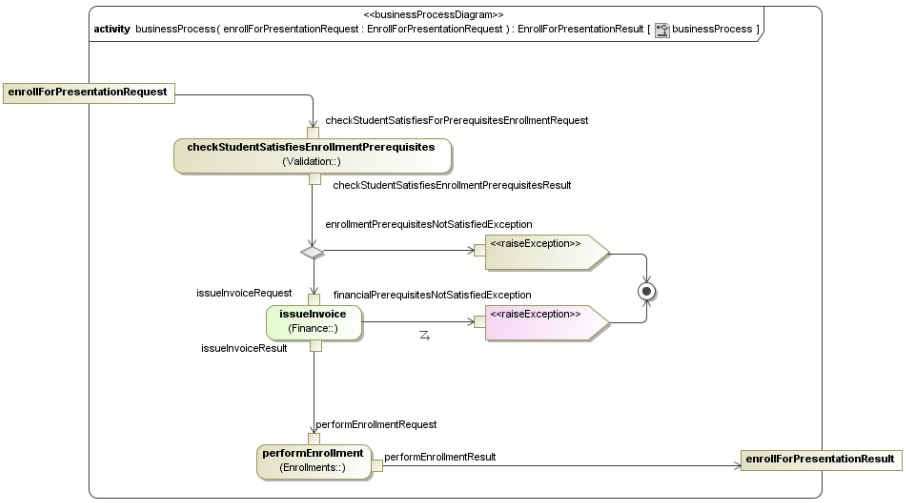

Figure 2: A diagrammatic representation of enrollForPresentation use case

3 Use case description

The provided use case is a typical business use case for enrolling a student for the presentation of a course. The target use case, `enrollForPresentation` contains the specifications for the data structures for the request and result, the functional requirements and the process specification which should enable the transformation team to generate the concrete implementation for that service assembled from the required lower level services.

3.1 Subject modeled

The input modeling language is the URDAD DSL. The metamodel specification and the example model are provided.

Since the model is meant to be a technology neutral model, the output modeling language is left up to the team and would correspond to the target technology chosen. This could be, for example, a model for either a JavaEE, Spring or SOA based implementation.

3.2 Use case variation points

Core use case variation points include

- The implementation mapping for different architectures and technologies (e.g. implementaiton mapping onto SOA or JavaEE).
- Mapping of the URDAD model onto a UML model.
- Generation of UML diagrams showing the functional requirements, services contract and process specification.

4 Guidelines for the Implementation Mapping

The following are some guidelines for the implementation mapping of specific URDAD model artefacts. The guidelines have a bias towards a typical modern object-oriented language like `Java` or `C#`, but the concepts can be easily mapped onto other technologies.

4.1 Responsibility domains

Responsibility domains are meant to be mapped onto packages or name spaces as well as onto an interface with the name of the package with the capitalization converted to the convention of the target technology.

4.2 Service contracts

Service contracts are meant to be mapped onto service/method declarations on interfaces or pure abstract classes which represent the services contract for a responsibility domain. The mapping of a contract for a specific service is meant to be done as follows:

- The request object is an instance of a request class mapping onto a single input parameter is an instance of the request class,
- Both, the request and response/result classes are service specific and should, if the target technology supports it, be mapped onto static nested classes of the interface.
- If the target technology supports notifiable exceptions (e.g. Java), the exceptions raised for the preconditions of the service should be included in the throws clause of the service.
- If the target includes unit test generation, then the unit test for a service should test that
  - the service is provided and not refused (no exception is thrown) if all pre-conditions are met,
  - all post-conditions are satisfied after the service has been provided.

4.3 Service and process specification

The concrete services in the metamodel contain the functional requirements (the lower level services from which the service is to be assembled) as well as the technology-neutral process specification which specifies how the process is orchestrated across these lower level services. There are service request activities, control structures, activities which create and manipulate local process variables and termination activities which either returning the result, or notify the user that the requested service is not provided because a particular precondition for the service is not met by raising the exception associated with that precondition.

The technology neutral design does not specify how references to the service providers of the lower level services are obtained. For modern implementation technologies this would typically be via dependency injection. The implementation mapping for other technologies might need to introduce a service provider registry and generate the appropriate service provider lookup code.

The process specification includes process variables which are accessible within the process and are maintained for the duration of the process. The inputs and outputs of the process (the request and result objects for the core service itself) are also process variables which can be queried and manipulated within the process.

4.4 Data structures

The URDAD data structure specification is essentially the same as in other object-oriented technologies supporting classes with attributes, association to other classes, inheritance and abstract classes. The different types of association relationships are meant to be implemented as follows:

- Plain *association* relationships are mapped onto simple references or pointers.
- *Aggregation* relationships are implemented the same as plain association relationships except in the case where state change notification is to be supported. In these cases one also needs to ensure that the aggregate object listens to state change events from its components and that it issues the appropriate state change event to its observers.
- *Composition* should be implemented as exclusive ownership, i.e. that the component is part of the owner and not part of any other object and should not survive the owner. Optionally the implementation mapping can implement full encapsulation for composition, i.e. that the component can only be accessed through the owner.

4.5 Unit tests

Unit tests should be developed for service contracts, not for concrete service definitions. They should only test the post-conditions for the current level of granularity, assuming that the lower level service providers fulfil their respective service contracts. In particular, they should test that

- that the service is not refused if any of the preconditions do not hold true, i.e. that no exception is thrown, and
- that, when the service is provided (no exception thrown), all post-conditions hold true after the service has been provided.

5 Evaluation guidelines

This section discusses some guidelines which we recommend to evaluate different transformation solutions.

5.1 Mapping preferences

When evaluating implementation mappings, we would recommend the following preferences, i.e. that transformations which use the preferred mapping strategies are rated higher than those which do not.

- Declarative bi-directional mappings should be preferred over operational/algorithmic uni-directional mappings.
- Standard mapping technologies like QVT relational should be preferred over custom mapping technologies.
- More simple, decoupled mapping rules should be preferred over fewer, complex rules.
- Standard target technologies commonly used for enterprise systems (e.g. JavaEE, SOA, Microsoft.Net) should be preferred over technologies seldom used for enterprise systems.

5.2 Completeness measure

The implementation should be tested against the tests developed or generated for the service contracts.

Completeness checks should include

- whether the interfaces for the service contracts have been generated and to what extend they comply with the implementation guidelines,
- the completeness of the data structure specifications and the degree to which they confirm to the implementation guidelines,
- the completeness of the service implementations including whether
  - the processes have been mapped onto method bodies,
  - whether the request and result data structures are populated in the generated process specification,
  - whether the conditionals for conditional flows and while loops are generated,
- whether unit tests are generated from the service contracts and the completeness of the unit tests in the form of degree coverage of pre- and post-conditions.

5.3 Correctness tests

The correctness can be tested by assessing whether the service is realized in such a way that

- if a particular precondition is not met, that the service is aborted raising that exception which was assigned in the model to that particular precondition.
- if all preconditions are met that the result object is obtained and all post-conditions evaluate to true after the service has been provided.

References

[1] Fritz Solms. Technology neutral business process design using URDAD. In *Proceeding of the 2007 conference on New Trends in Software Methodologies, Tools and Techniques*, page 5270, Amsterdam, The Netherlands, The Netherlands, 2007. IOS Press.

[2] Fritz Solms and Dawid Loubser. Generating MDA's platform independent model using URDAD. *Knowledge-Based Systems*, 22(3):174–185, April 2009.

[3] Fritz Solms and Dawid Loubser. URDAD as a semi-formal approach to analysis and design. *Innovations in Systems and Software Engineering*, 6:155–162, 2010. 10.1007/s11334-009-0113-4.