Multi-Language Support for Model-Driven Requirement Analysis and Test Generation

Clémentine Nebut[1], Benoit Baudry[2], Souha Kamoun[1], and Waqas Ahmed Saeed[2]

[1] LIRMM, CNRS and Université de Montpellier 2,
161, rue Ada, 34392 Montpellier cedex 5, France
{nebut, kamoun}@lirmm.fr
[2] IRISA, INRIA and Université de Rennes 1
Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
{bbaudry, wahmedsa}@irisa.fr

Abstract. Expressing a valid requirements model is a crucial activity in a MDE context as it provides the starting point for further refinement steps that will lead to the implementation. In previous work we have proposed a requirements model from which it is possible to do simulation and high-level test generation. A requirements metamodel captures the key concepts to model the requirements and allows us to define several automatic model transformations to manipulate the requirements. In this paper, we focus on surface languages that can assist the edition of such a requirements model for simulation or testing purposes. We propose to use two different languages: a constrained natural language and UML activity diagrams. For each language, we discuss the main interesting features to express requirements and we define a model transformation to generate the corresponding requirements model.

1 Introduction

Until now, Model Driven Engineering has mainly focused on the design of software applications, but still lacks a real integration with known validation techniques. This lack is explained by the recent expansion of MDE, and certainly not by the infeasibility of transferring validation techniques to the model paradigm. When a software is validated by testing techniques, a good approach is to integrate the testing concerns as soon as possible in the development, ideally while the requirements are being made explicit. We strongly believe that having sound techniques to validate the requirements and generate high level test cases from a requirements model is a key activity to make MDE successful since this model should be the first one in the refinement process to build the final application. Several proposals have already been made concerning test generation from requirements, in a model-driven development context [1,2].
In [2] we propose an approach to validate the requirements through interactive simulation and to automatically generate test cases. This approach is based on a requirements model in which requirements are expressed

using uses cases enhanced with contracts (pre and post condition) in the form of logical expressions. This model can be simulated to analyze and validate the requirements by checking for inconsistencies or incompleteness. The model is also the input for our test generation approach. We have defined a metamodel to formally describe the requirements model and to capture the key concepts that constitute this model. This also allows MDE techniques to be applied to automate parts of the requirements process. For example, it is possible to ensure part of the traceability of the requirements in more detailed models thanks to automatic model transformations. It is also easier to integrate our requirements analysis approach in a MDE context, and to extract different views on a requirements model (documentation, tests, preliminary analysis model). At last, we are able to define the test generation process as a model transformation. Now, what is missing in this approach is a way to edit a requirements model (that conforms to the metamodel) in a user-friendly way. Moreover, since different stakeholders (with different expertises and points of view) will participate to the definition of the requirements model, we would like to have different surface languages to edit one model.
In this paper, we discuss two surface languages to edit a requirement model: a constrained natural language named RDL (for Requirement Description Language) and UML activity diagrams. For both approaches we discuss the main features of the syntaxes and define a model transformation to go from the different syntaxes to our core requirements metamodel. The transformations are implemented using the model-oriented language Kermeta [3] (note that Kermeta is also used to defined the metamodels and give them a semantics). An important benefit of automatic transformations is to ensure the consistency between the surface syntaxes and the underlying model and thus to ensure the consistency between the generated test cases and the expressed requirements.
The paper is organized as follows. Section 2 summarizes the approach proposed in [2] for test generation from functional requirements, and makes explicit the underlying requirements metamodel. Section 3 presents the RDL and the model transformation allowing to go from an RDL model to a requirements model. Then section 4 explains how activity diagrams can be used to enter requirements, and details the transformation to go from activity diagrams to a requirements model. Finally, Section 5 draws the perspectives of our work.

2 A requirement metamodel for test generation

We detail in this section the requirements metamodel and recall the main principles of test generation based on this metamodel.

2.1 The requirements metamodel

The requirements metamodel we propose was defined based on the experiments we had with THALES [4] for test generation (an overview of the metamodel is given in Figure 1).

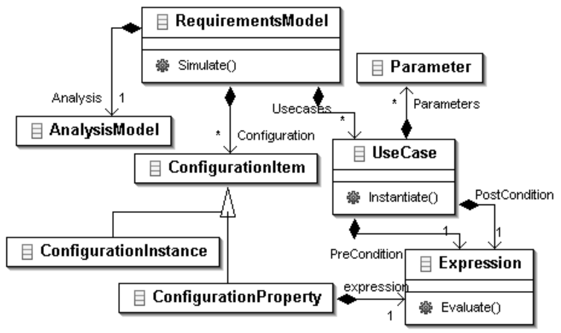


Fig. 1. Overview of the requirements metamodel

The idea that led to build this metamodel is that requirements can be expressed with parameterized uses cases associated with pre and post conditions. The precondition defines the conditions in which a use case can be executed and the postcondition expresses the effect the use case has on the state of the system. The parameters of a use case are either actors or business concepts handled by the use case. Use case contracts are represented by the `Expression` class in Figure 1. They are expressed as first order logical expressions having a set of typed parameters, combined with different logical operators. These expressions are used to describe the properties of the system (an actor state, a business concept state, etc). These expressions are Boolean expressions, thus can be either true or false. Logical operators include conjunction (and), disjunction (or), negation (not) and implication. In order to increase the expressiveness, exists and forall quantifiers are also included. Let us illustrate this metamodel with a concrete example of Library Management System, that will be used all over the paper. The requirements of this system are the following. A library is maintained by a librarian. A customer must register in the library to avail the facility of borrowing the books. Books must be registered before they are available to the customers. When a customer returns the book, the book is not available for any customer to borrow again, till the librarian performs an inventory check. In this example, we can identify several use cases conform to the previously described requirements metamodel. For example, the use case Borrow can be described as follows.

UseCase Borrow (c: customer, b:Book)
Pre: registered(c) **and** available(b) **and not** damaged(b)
Post: registered(c) **and not** available(b)

In a requirements model, we also need a data model that represents the business concepts and relationships that are manipulated in the requirements. This model is represented in Figure 1 by the class `AnalysisModel` and is very close to a class diagram. At last, for simulation purposes, we also need to define the initial configuration of the system from which we start the simulation. This is represented by the class `ConfigurationItem` and its subclasses in Figure 1.

2.2 Requirements simulation, and test generation

The simulation of the requirements is the basis for test generation as presented in [2], but it is also a way to improve the quality of the requirements (in the sense that simulating the requirements can help in detecting inconsistencies or under-specification). The principle of the simulation relies on the instantiation of the use cases, based on the data model. Instantiating a use case consists in replacing its formal parameters by actual ones, based on the data model. Simulation of the requirements is founded on a labeled transition system named UCTS (for Use Case Transition System) defined by:

- Q, a finite non-empty set of states. Each state is defined as a set of instantiated expressions, and represents the system's state at a given stage of simulation.
- an initial state, that represents the initial state of the system,
- A, the alphabet of actions, an action being an instantiated use case,
- $\rightarrow \subseteq Q \times A \times Q$ the transition function. Each of the resulting edges represents the instantiated use case it is labeled with.

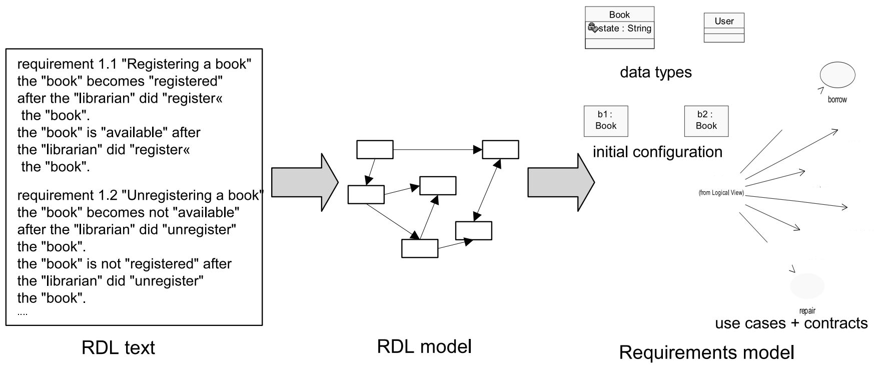
The simulation mechanism is directly implemented in the requirements metamodel, by defining the behavior (with the Kermeta language) of the `Simulate` method in the `RequirementsModel` class.

From a UCTS, it is possible to generate abstract tests, named test objectives, in the form of correct sequences of instantiated use cases. Indeed, a UCTS is a representation of all the possible orderings of the use cases. Generating test objectives just consists in selecting relevant paths of the UCTS using adequate coverage criteria (for example, the coverage of all the instantiated use cases). Several such criteria have been proposed and comparatively studied in [2]. The test objectives are quite far from concrete tests, since they are expressed at the requirement level. To bridge the gap between test objectives and test cases, design information is thus needed. In a model-driven approach, we can reasonably assume that scenarios expressing the actual messages exchanges between the system and the environment will be available. Those scenarios are sufficient to generate test cases from test objectives, by replacing a given instantiated use case of a test objective by the corresponding instantiated scenario.

Simulation and test generation is thus possible as soon as we dispose of a requirement model that conforms to the requirements metamodel given in Figure 1. However, as stated in introduction, instantiating such a metamodel is a difficult task. That is why we propose two surface languages in the following sections to instantiate the metamodel.

3 A constrained natural language to write the requirements

The RDL was developed [4] to make it easier for the requirements writers to edit their requirements model. We have developed a series of transformations to generate a requirements model from RDL sentences. Figure 2



requirement 1.1 "Registering a book"
the "book" becomes "registered"
after the "librarian" did "register«
 the "book".
the "book" is "available" after
the "librarian" did "register«
 the "book".

requirement 1.2 "Unregistering a book"
the "book" becomes not "available"
after the "librarian" did "unregister"
the "book".
the "book" is not "registered" after
the "librarian" did "unregister"
the "book".
....

Fig. 2. Transformation from an RDL text to a requirements model

gives a general overview of the transformation process: the RDL text is transformed into an RDL model. This RDL model is then transformed into the requirements model.

3.1 A language to define requirements

The RDL mainly aims at preventing from writing ambiguous, incomplete, and incorrect sentences. Its syntax has been designed to produce readable and natural language-like sentences, even if the RDL is by no means intended to deal with issues coming from natural language analysis. Figure 3 displays three lines of RDL sentences from the Library Management System example. The domain-specific words are contained in quoted strings.

```
1. there is a "book" named "beloved"
2. the "customer" must be "registered" before the "customer" can "borrow" the "book"
3. the"book" becomes not "available" after the "customer" did "borrow" the "book"
```

Fig. 3. Example of RDL sentences

3.2 From the RDL to the requirements model

As shown in Figure 2, we defined two steps to transform RDL sentences into a requirements model. The first step transforms the RDL text into an RDL model. This step is also decomposed into two sub-steps. First the text is parsed to produce a syntax tree. ANTLR [5] is used to automatically generate the parser and an abstract syntax tree from the RDL

IF S1=Action THEN O1=ObservableProperty	
O1.type	BECOMES
O1.reference	A
S1.type	DOES

Table 1. An example of pattern

Type	USE CASE
Name	S1.title
Parameters	x1 : S1.activator.type x2 : O1.reference.owner.type
Precondition	not O1.reference.observable = O1.reference.value
Postcondition	O1.reference.observable = O1.reference.value

Table 2. An example of production

grammar. Then, a model transformation generates an RDL model from the RDL concrete syntax tree. This RDL model is an instance of the RDL metamodel which is a representation of the RDL grammar in the form of a metamodel, enhanced with semantics, and with standard serialization format. The second step consists in transforming an RDL model into a requirements model: the static analysis model is built first, then use cases with contracts are generated. The static analysis model is generated simply identifying the properties and actions, with their owners. The observable properties are treated as attributes and direct (implicit) references to the property imply that the property is of boolean type, otherwise of an enumerated type. Similarly, actions (activations) are treated as operations and associated complements are added as parameters to the operation.

Let us illustrate this transformation with the third sentence of Figure 3. "available" is the implicit property of the owner "book", so a boolean type attribute named `available` is created in the class `Book`. Similarly, for action "borrow" with complement "book" and subject "customer", an operation named `borrow` with a parameter of type `book` is added to the class `Customer`.

During this transformation, semantics is attached to the RDL sentences by means of *interpretation patterns*. To identify these patterns in an RDL model, we use pattern matching techniques. Each interpretation pattern associates a semantics to a given RDL construction, it is composed of a pattern identifying the construction, and of a production specifying the corresponding semantics. For example, an interpretation pattern can express that a property of an actor or a business concept changes upon the activation of a use case.

An example of interpretation pattern is given in Tables 1 and 2. The top line of Table 1 defines the look and feel of the pattern and the

bottom part defines the constraints on the associated items in the RDL model. Table 2 shows a possible corresponding production for a result of type Use Case. The pattern defines the syntactical elements involved in a sentence for which the keyword *becomes* is used to express the fact that a boolean property of an object has changed during the service activation. According to the pattern, if the observable property is preceded by the keyword *becomes* and the service activation is preceded by the keyword *did* then the resultant use case will be defined by the production in Table 2. The name of the use case will be the name of the service, the parameters will be the actors or business concept which activated the service, and the reference whose property is changed. The precondition of the use case will be the negation of the observable value and the post condition will be the observable value.
We defined 13 patterns to interpret the RDL. They bring flexibility in the transformation process and have to be tailored for a given domain. If a requirement in the RDL model does not match any interpretation pattern, it means that either the requirement has no meaning and must be rewritten, or a new interpretation pattern must be introduced to give semantics to the requirement, and added to the set of interpretation patterns. The transformation from a RDL model to a requirements model has been implemented using the Kermeta language [3].

4 Using activity diagrams to write the requirements

To obtain a requirements model that conforms to the metamodel we have presented in Section 2, the essential information needed is the dependencies linking the requirements, to be able to obtain use cases with contracts. As previously explained, directly defining those contracts is far from being a trivial task, and a commonly-shared intuition is that a graphical notation should help. Indeed, it seems easier to define dependencies with graphical links instead of with logical expressions. This idea led us to work on a graphical notation for the dependencies, and we have chosen UML activity diagrams, since they are used in [6] for the same purpose. In this section, the features of the activity diagrams we deal with are exposed, and the corresponding transformation into the requirements metamodel is described.

4.1 Expressing requirements using activity diagrams

We work with UML 2.0 [7] activity diagrams. We recall that activity diagrams are a way to specify both control and object flows, with a semantics very near from petri nets (the semantics is given in terms of tokens moving along the control flows). Each action owns to the partition named with the main actor involved in the action. Then the control flows and control nodes (join, fork, merge, ...) defined in UML2.0 are used to express the dependencies between the actions. Figure 4 shows a small example of activity diagram for the Library example. This activity diagram simply expresses the following facts:

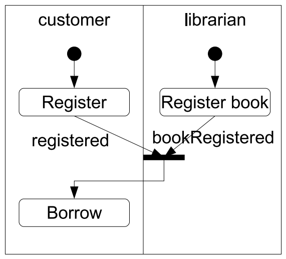


Fig. 4. An example of activity diagram

- a librarian can register a book
- a customer can register
- a customer can borrow a book if he has registered and if the book has been registered by the librarian.

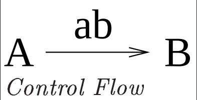
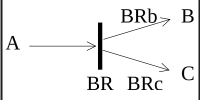
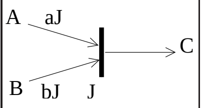
This activity diagram does not specify anything about the books: in other words, we currently only deal with actors and actions, and not with business entities. Consequently, the activity diagram we propose is not complete enough to fully specify the requirements, and we are currently working on improving them by adding parameters to the actions (using object nodes). This current lack does not harm the intent of this paper, that is to show that different user-friendly surface languages can be used to enter a requirements model allowing tests to be generated.

4.2 From activity diagrams to a requirement model

We have implemented (using the Kermeta language) a transformation that generates the use case part of the requirements model from an activity diagram. Once actions will be associated with parameters, the transformation from activity diagrams to the use case model will be completed, and we will develop a transformation to generate the static analysis model.

To generate a use case model from an activity diagram, the basic principle is to represent the position of the tokens moving along the activity diagram by predicates: a predicate corresponding to the control flow *cf* is true when there is a token on *cf*. We have defined a set of local rules, provided in Table 3, that express how each kind of activity diagram construction is transformed into use cases and contracts. Those local rules are then combined using a two-step algorithm for each activity *act*:

1. We parse down the execution flow from the outgoing flow, in order to determine where the tokens should be left after the execution of the activity. This way, part of the postcondition of the use case corresponding to *act* is obtained.
2. Then we parse up the execution flow from the incoming flow, in order to determine where the tokens are supposed to be to enable the execution of the action. That allows us to obtain the precondition

Activity Diagram	Use case
 *Control Flow*	**UC** A **pre** **post** ab **UC** B **pre** ab **post not** ab
 *Fork*	**UC** A **pre** **post** BRb **and** BRc **UC** B **pre** BRb **post not** BRc **UC** C **pre** BRc **post not** BRc
 *Join*	**UC** A **pre** **post** aJ **UC** B **pre** **post** bJ **UC** C **pre** aJ **and** bJ **post not** aJ **and not** bJ

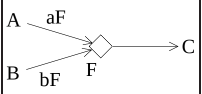
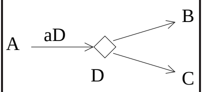
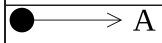
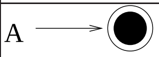
Activity Diagram	Use case
 *Merge*	**UC** A **pre** **post** aF **UC** B **pre** **post** bF **UC** C **pre** aF **or** bF **post** aF **implies not** aF **and** bF **implies not** bF
 *Decision*	**UC** A **pre** **post** aD **UC** B **pre** aD **post not** aD **UC** C **pre** aD **post** not aD
 *Initial node*	**UC** A **pre** init **post not** init
 *Final flow*	**UC** A **pre** **post**

Table 3. Local transformation rules

of the use case corresponding to *act*. During this phase, we also complete the postcondition obtained in step 1, taking into account the movement of the tokens during the execution: we specify in the postcondition where the tokens are not anymore after the execution.

We then complete the obtained use case model with predicates dealing with final activity nodes (for which no local rule can be defined), and with actors. To each use case generated from the action *act*, we add a parameter corresponding to the main actor involved in the use case, and a precondition specifying which actor can initiate the use case execution. For that, we introduce a predicate having the name of the swimlane to which *act* owns. As an example, for the action *Register* of the activity diagram of Figure 4, we obtain the following use case (presented in textual format):

UseCase Register (c: LibraryActor) **Pre**: customer(c) **and** init **Post**: registered **and not** init

5 Perspectives and conclusion

The approach presented in this paper shows that several surface languages can be used to enter (in a user-friendly way) a requirements

model, that can then be used for test generation purpose. We strongly believe that model-driven techniques are very fruitful both to obtain "good" requirements and to bridge the gap between requirements and tests. We are also convinced that it is not possible to define a universal language perfectly suitable to write any requirement.
As a consequence, none of the languages we propose is adequate to enter a whole requirements model. For example, activity diagrams have the obvious benefit of being graphical, and thus the logic behind the requirements appears in a glimpse. However, it is definitively not reasonable to ask a requirement engineer to write a single activity diagram to specify a whole requirements model. And even if mechanisms can be provided to merge small activity diagrams, and thus to build a requirements model requirement by requirement, the expressiveness of the activity diagrams is not sufficient to express any kind of requirement (as underlined in [2]). That means that for each requirement to enter, the requirements engineer should be given the choice between a family of languages. This family of languages must include mechanisms to be transformed towards the same requirements metamodel, in order to allow a set of requirements written in various languages to be merged into a single requirements model. The merging of requirements (relying on the merging of models) is a complex task that can be supported by interactive model transformations.
Another crucial point to explore is the traceability between the requirements and the generated models from the requirements, in particular the test models. Since automatic transformations are used in our approach, traceability mechanisms have to be implemented within the transformations to be able to determine which requirements led to a given test, and reciprocally which tests are generated from a given requirement.

References

1. Boddu, R., Guo, L., Mukhopadhyay, S., Cukic, B.: RETNA: From requirements to testing in a natural way. In: Proc. of the 12th IEEE International Conf. on Requirements Engineering. (2004) 262–271
2. Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.M.: Automatic test generation: A use case driven approach. IEEE Transactions on Software Engineering **32**(3) (2006) 140–155
3. Triskell project (IRISA): The metamodeling language kermeta. http://www.kermeta.org (2006)
4. Lugato, D., Maraux, F., Le Traon, Y., Normand, V., Dubois, H., Pierron, J.Y., Gallois, J.P., Nebut, C.: Automated functional test case synthesis from thales industrial requirements. In: IEEE Real-Time and Embedded Technology and Applications Symposium. (2004)
5. Parr, T.: ANTLR. http://www.antlr.org (2006)
6. Briand, L., Labiche, Y.: A UML-based approach to system testing. Journal of Software and Systems Modeling (2002) 10–42
7. OMG: UML version 2.0. http://www.omg.org/technology/documents/-formal/uml.htm (2006)