Visualization of Use Cases Through Automatically Generated Activity Diagrams

Javier Gutierrez Rodriguez, Clémentine Nebut, Maria Escalona Cuaresma, Manuel Mejias Risoto, Isabel Ramos Roman

To cite this version:

Javier Gutierrez Rodriguez, Clémentine Nebut, Maria Escalona Cuaresma, Manuel Mejias Risoto, Isabel Ramos Roman. Visualization of Use Cases Through Automatically Generated Activity Diagrams. MoDELS: Model Driven Engineering Languages and Systems, Sep 2008, Toulouse, France. pp.83-96, 10.1007/978-3-540-87875-9_6 . lirmm-00322934

HAL Id: lirmm-00322934

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00322934

Submitted on 16 Sep 2019

Visualization of Use Cases through Automatically Generated Activity Diagrams

Javier J. Gutiérrez[1], Clémentine Nebut[2], María J. Escalona[1], Manuel Mejías[1], and Isabel M. Ramos[1]

[1] Department of Languages and Computer Systems,
University of Sevilla, Spain
{javierj,mjescalona,risoto,ramos}@us.es
[2] LIRMM, CNRS and university of Montpellier 2, France
nebut@lirmm.fr

Abstract. Functional requirements are often written using use cases formatted by textual templates. This textual approach has the advantage to be easy to adopt, but the requirements can then hardly be processed for further purposes like test generation. In this paper, we propose to generate automatically through a model transformation an activity diagram modeling the use case scenario. Such an activity diagram allows us to guess in a glimpse the global behavior of a use case, and can easily be processed. The transformation is defined using the QVT-Relational language, and is illustrated on a case study using a supporting tool.

1 Introduction

Requirement engineering is a crucial activity for a successful project. Among the different tasks to achieve in this activity is the writing of the elicited requirements in a given language (formal, semi-formal, or natural).

Structured use cases is a classical standard used in industry and taken as entry point of several research work [3][4][5][14]. By structured use cases, we mean use cases as defined in UML [22], enhanced with additional information like pre and post conditions, and scenarii. Such use cases usually follow templates like Cockburn one [3], defining in natural language usage scenarii, quality of service constraints, or graphical interface elements, like [4].

A UML use case model does not contain much information. It allows to define the frontier of the system, actors, name of use cases, and relations between the use cases, the actors, and linking actors and use cases. UML methodologies like RUP [23] or Catalysis [24] recommend to enhance this model with other UML artifacts like constraints to define contracts, and sequence diagrams to define system scenarii in a graphical way.

Both use case templates and UML methodologies thus incite the requirement writer to define scenarii, in a graphical or textual notation. Graphical notations for scenarii (system sequence diagrams [19] or activity diagrams [25]) have the advantage to give in a glimpse a good idea of the dynamic aspects of a use case (loops, alternatives, choices). Moreover, they can be easily used by a tool to be analyzed (for example for test generation purposes, also in [19]). On the other hand, they can be difficult to design, and domain experts cannot reasonably asked to draw them. On the opposite, textual scenarii are easy to write: the scenarii are composed of various steps, each one is described by natural language sentences, and linked with natural language connectors. Such textual scenarii do not give an immediate intuition of the behavior of the use case, and cannot easily be processed by a tool.

The original contribution of this paper is a process to generate in an automated way an activity diagram modeling use case dynamic from textual scenarii. The generation is performed by a model transformation, taking as input use case textual scenarii (conform to a metamodel defined in this paper), and producing an activity diagram (conform to a restricted part of the UML activity diagram metamodel). The transformation is defined using QVT-relational language and implemented in Java in a prototype tool.

This transformation allows to benefit both from graphical and textual scenarii advantages. Textual scenarii can be written. Then they can be transformed in a graphical view. This allows an easier integration to UML tools, and to use the activity diagram for further treatments like test generation for example.

The rest of this paper is organized as follows. Section 2 introduces the source and target metamodels, this means, the textual requirement metamodel and an excerpt of the UML activity metamodel. Section 3 describes the transformation to generate an activity diagram from a tabular requirement. Section 4 introduces a case study. Section 5 describes other related approaches and, finally, section 5 summarizes conclusions and ongoing works.

2 Functional Requirements and Activity Diagrams Metamodels

This section introduces the artifacts we deal with: functional requirements and activity diagrams. Both ones are described by metamodels: our own metamodel for functional requirements and a subset of the UML metamodel for the activity diagrams.

2.1 Functional Requirement Metamodel

There are many approaches to define textual requirements [3][4][5][14]. However the majority of the existing approaches are based on a common core of elements. These elements include the participants, preconditions, post-conditions, a set of main steps (to achieve the goal of the use case) and a set of exceptional steps (to manage alternative and erroneous scenarios). These common elements have been extracted from the cited approaches and modeled in the functional requirement metamodel introduced in figure 1 and briefly described in the following.

A *FRActor* element models an external actor who participates in the functional requirement performing the steps. The *FunctionalRequirement* element defines an interaction among the system and a set of external actors. The behavior of *FunctionalRequirement* is defined with three different sequences of steps. The main sequence defines the steps performed by *FRActor* elements and the system to achieve the goal of the functional requirement. The alternative steps indicate variants or additional behavior for the steps of the main sequence. The erroneous steps indicate erroneous scenarios after performing a step from the main sequence. Alternative and erroneous steps are modeled with *ExceptionalStep* element.

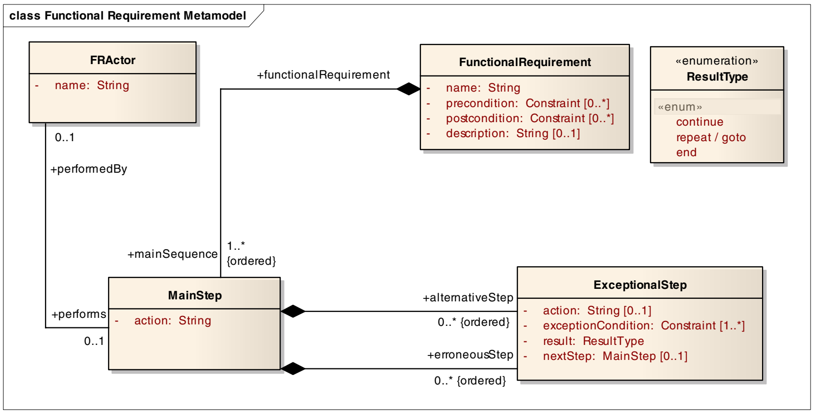


Fig. 1. Functional Requirement Metamodel

```
<useCase id="RF-01. Logging">
  <description> An user wants to access into the system.</description>
  <precondition>No.</precondition>
  <postcondition>System grants access to the user.</postcondition>
  <mainSequence>
    <step id="1"> User asks access into the system. </step>
    <step id="2"> System asks for a name and a key. </step>
    <step id="3"> User introduces a name and a key. </step>
    <step id="4"> System validates name and key. </step>
    <step id="4"> System allows access. </step>
 </mainSequence>
 <alternativeSteps>
  <astep id="2.1"> If the number of tries is greater than three,
  then the system shows an error message and this use case ends.
  </astep>
  </alternativeSteps>
  <errorSteps>
    <estep id="4.1"> If the name of the key size is less than
    four characters, then the system shows an error message and
    step 2 is repeated.
    </estep>
    <estep id="4.2">
        If the name or the key are not registered, the system
        shows and error message and step 2 is repeated.
    </estep>
   </errorSteps>
</useCase>
```

Fig. 2. Example use case (in XML format)

In this paper, the only relevant information for a *MainStep* element is the action performed (attribute *action*) and who performs the action. In an *ExceptionalStep* element, the action is not mandatory and the performer is the same one than the main step. The *exceptionCondition* attribute indicates a boolean expression that must be true to perform the step and the *result* attribute indicates the action to perform at the end of the exceptional step. Three possible results have been defined as the enumerated *ResultType* element: end the execution, continue the execution and repeat/go-to to another step. A continue result indicates that the execution of the functional requirement continues with the next step after the end of the alternative or erroneous step. The repeat or go-to result indicates that the execution of the scenario follows the execution of the step indicated in the *nextStep* attribute, and the end result indicates than the use case scenario ends. The *nextStep* attribute is only mandatory when the result type is repeat / go-to. An end result indicates that the execution of the scenario is finished. We have defined a concrete syntax to define functional requirements conform to this metamodel (figure 2). The concrete syntax is XML-based to make an approach available. One tool, available in [1], has been development to generate this XML for our modeling tool (described in section 4).

2.2 Activity Diagram Metamodel

The UML activities metamodel contains a vast amount of elements which improves the flexibility and semantic of activity diagrams. A subset of the activity diagrams metamodel (as defined in UML 2.0) has been selected (figure 3). This subset defines the elements automatically generated by the transformations introduced in next section. Indeed, after the automatic generation of the activity diagrams, additional elements, like data flow, may be added by hand.

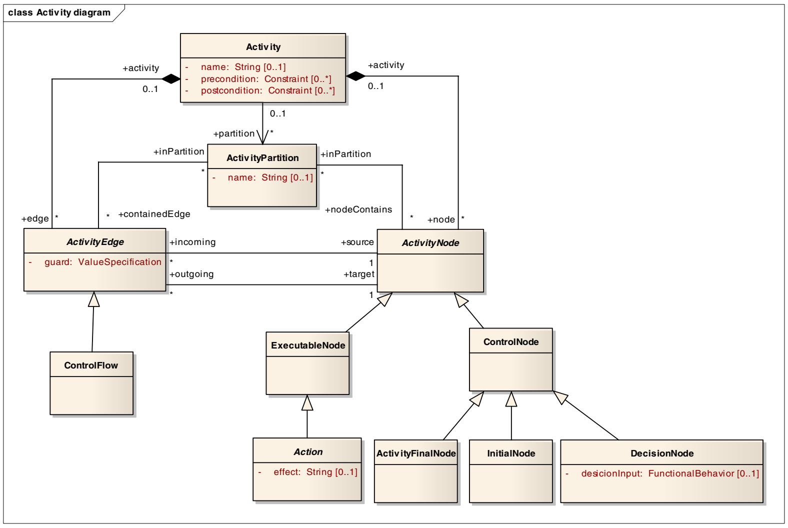
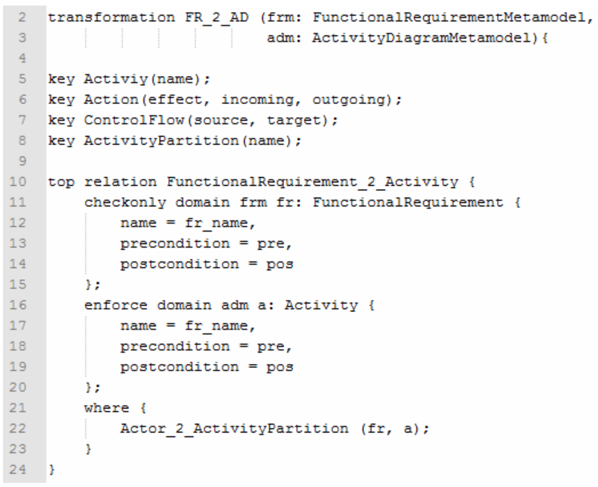


Fig. 3. Activity diagrams metamodel

Since UML 2.0, the activity diagram element has been replaced with the *Activity* metaclass. Indeed, this metaclass groups all the elements of an activity diagram: control flows, nodes and activity partitions. Activity partitions are groups of the elements of an activity diagram. Control flows indicate the incomings and outgoings of a node. Action nodes define a behavior. Decision node allows to select a flow as result of evaluating boolean predicates. Finally, the *InitialNode* and *ActivityFinalNode* elements indicate where the control flow begins and ends.

Next section, describes how these elements represents the concepts defined in the functional requirement metamodel.



```
2   transformation FR_2_AD (frm: FunctionalRequirementMetamodel,
3                             adm: ActivityDiagramMetamodel){
4
5   key Activiy(name);
6   key Action(effect, incoming, outgoing);
7   key ControlFlow(source, target);
8   key ActivityPartition(name);
9
10  top relation FunctionalRequirement_2_Activity {
11      checkonly domain frm fr: FunctionalRequirement {
12          name = fr_name,
13          precondition = pre,
14          postcondition = pos
15      };
16      enforce domain adm a: Activity {
17          name = fr_name,
18          precondition = pre,
19          postcondition = pos
20      };
21      where {
22          Actor_2_ActivityPartition (fr, a);
23      }
24  }
```

Fig. 4. From functional requirement to activity

3 Transformation from Textual Scenarios to Activity Diagrams

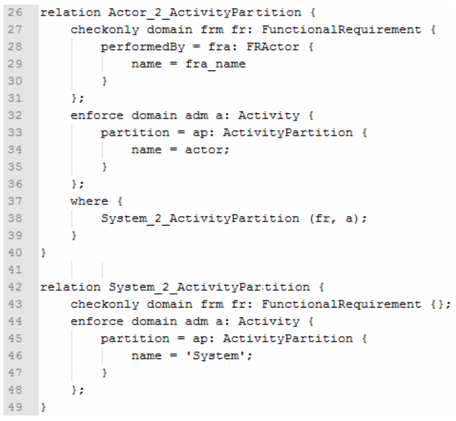
This section defines the transformation to generate an activity diagram (as an instance of the metamodel of figure 3) from a functional requirement (as an instance of the metamodel of figure 1). The objective of this transformation is to generate from a functional requirement an activity that models the whole functional requirement (with the same name, pre-conditions and post-conditions). In this activity, the activity partitions model the different actors implied in the functional requirement and the system under development, control flows model the execution paths or scenarios and actions and decision nodes model the steps and the exception conditions. Transformation performs next tasks:

1. generating an activity for each functional requirement,
2. generating an action for each main step and generate a decision node and additional actions for each exceptional step,
3. all elements in the activity diagram are connected using control flows.

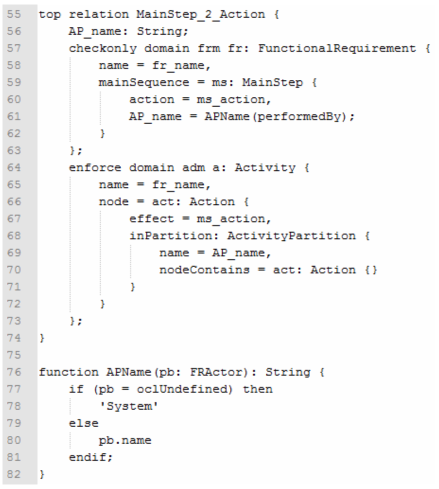
All the tasks are defined using QVT-Relational [26]. We here present only the main ones in QVT-Relational; the whole transformation can be found in [1]. The steps are described in detail in next sections.

3.1 From Functional Requirements to Activity Diagrams

Initially, one activity is generated for each functional requirement. The related relation (called *FR_2_AD*), in QVT-Relational language, is showed in figure 4. The name, preconditions and post-conditions of the activity are the same than the ones in the functional requirement (lines 17, 18 and 19).



```
26 relation Actor_2_ActivityPartition {
27     checkonly domain frm fr: FunctionalRequirement {
28         performedBy = fra: FRActor {
29             name = fra_name
30         }
31     };
32     enforce domain adm a: Activity {
33         partition = ap: ActivityPartition {
34             name = actor;
35         }
36     };
37     where {
38         System_2_ActivityPartition (fr, a);
39     }
40 }
41
42 relation System_2_ActivityPartition {
43     checkonly domain frm fr: FunctionalRequirement {};
44     enforce domain adm a: Activity {
45         partition = ap: ActivityPartition {
46             name = 'System';
47         }
48     };
49 }
```



```
55 top relation MainStep_2_Action {
56     AP_name: String;
57     checkonly domain frm fr: FunctionalRequirement {
58         name = fr_name,
59         mainSequence = ms: MainStep {
60             action = ms_action,
61             AP_name = APName(performedBy);
62         }
63     };
64     enforce domain adm a: Activity {
65         name = fr_name,
66         node = act: Action {
67             effect = ms_action,
68             inPartition: ActivityPartition {
69                 name = AP_name,
70                 nodeContains = act: Action {}
71             }
72         }
73     };
74 }
75
76 function APName(pb: FRActor): String {
77     if (pb = oclUndefined) then
78         'System'
79     else
80         pb.name
81     endif;
82 }
```

Fig. 5. (a) From actor to activity partition. (b) From main step to action.

As seen in the previous section, one activity partition is also generated for each actor participant in the functional requirement (relation *Actor_2_ActivityPartition*, called in line 22) plus one additional partition for the system (as showed in figure 5(a), lines from 42 to 49). The next task is to transform the behavior, defined with three sequences of steps, of the functional requirement.

3.2 From Main Sequence to Actions

One action is generated for each step in the main sequence. The related QVT relation is given in figure 5(b). The attribute *effect* of the action is the attribute *action* of the step (line 67). The actions are included in the corresponding activity partition which may be identified using the *performedBy* attribute of the main step (lines 69, 70 and 71). However, if a step is performed by the system, this attribute is empty, so the action is classified in the system partition. This behavior is codified in the auxiliary function *APName* (line 76, figure 5(b)). The next step is to transform alternative and erroneous sequences in decision nodes and actions.

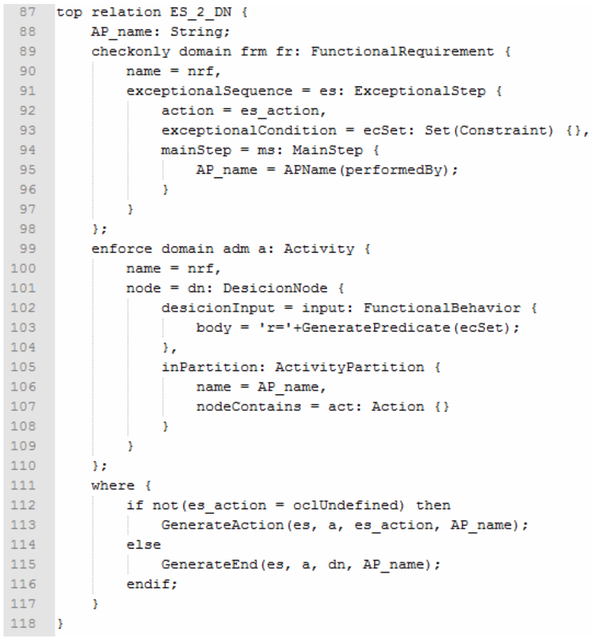
3.3 From Alternative and Erroneous Sequences to Actions

Two elements are generated from an alternative or an erroneous step. The QVT relation goes in figure 6.

The first element is a decision node (class *DesicionNode*, line 101), which evaluates the exception condition to allow or not the execution of the alternative or erroneous behavior. This exception condition is stored in the attribute *desicionInput*, which is generated with the auxiliary function *generatePredicate* (used in line 103). Due the exception condition was defined as constraints in the functional requirement metamodel (figure 1), this auxiliary function builds a string from a set of *Constraint* elements.

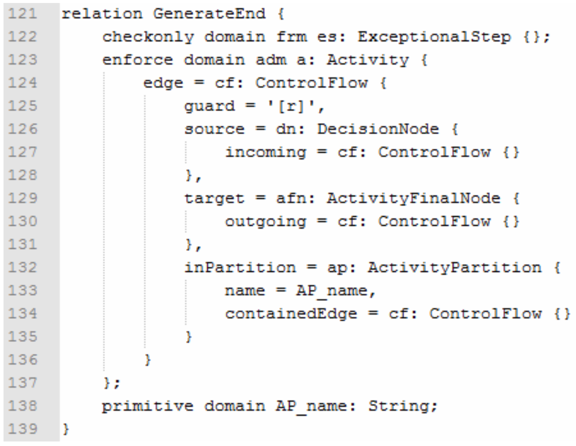
The second element is an action describing the additional behavior if any (as it has been seen in the functional requirement metamodel, an action attribute for exceptional steps is not mandatory). This action is generated by the auxiliary function *GenerateAction* (line 113). This function has been omitted from figure 7, but its code is quite similar to the QVT transformation *MainStep_2_*Action (figure 5(b)).

Then, every possible result has also a different representation in an activity diagram, so three different relations are called to generate the adequate elements for each type of results (only one, relation *GenerateEnd*, has been included in the *where* clause of figure 6, line 115).



```
 87  top relation ES_2_DN {
 88      AP_name: String;
 89      checkonly domain frm fr: FunctionalRequirement {
 90          name = nrf,
 91          exceptionalSequence = es: ExceptionalStep {
 92              action = es_action,
 93              exceptionalCondition = ecSet: Set(Constraint) {},
 94              mainStep = ms: MainStep {
 95                  AP_name = APName(performedBy);
 96              }
 97          }
 98      };
 99      enforce domain adm a: Activity {
100          name = nrf,
101          node = dn: DesicionNode {
102              desicionInput = input: FunctionalBehavior {
103                  body = 'r='+GeneratePredicate(ecSet);
104              },
105              inPartition: ActivityPartition {
106                  name = AP_name,
107                  nodeContains = act: Action {}
108              }
109          }
110      };
111      where {
112          if not(es_action = oclUndefined) then
113              GenerateAction(es, a, es_action, AP_name);
114          else
115              GenerateEnd(es, a, dn, AP_name);
116          endif;
117      }
118  }
```

Fig. 6. From exceptional step to decision node



```
121  relation GenerateEnd {
122      checkonly domain frm es: ExceptionalStep {};
123      enforce domain adm a: Activity {
124          edge = cf: ControlFlow {
125              guard = '[r]',
126              source = dn: DecisionNode {
127                  incoming = cf: ControlFlow {}
128              },
129              target = afn: ActivityFinalNode {
130                  outgoing = cf: ControlFlow {}
131              },
132              inPartition = ap: ActivityPartition {
133                  name = AP_name,
134                  containedEdge = cf: ControlFlow {}
135              }
136          }
137      };
138      primitive domain AP_name: String;
139  }
```

Fig. 7. Exceptional step of type end

With an end use case result, a new activity end node (class *ActivityFinalNode*) is added and it is linked with the decision (or action if any) generated in previous relation using a new *ControlFlow* (from line 124 to 136, figure 7).

For a continue use case, the action corresponding to the next step (in the main sequence) is obtained and linked with the decision (or action if any) using a control flow element. Finally, for a repeat/ go-to result type, the action corresponding for the next step attribute (see metamodel in figure 1) is obtained and linked with the decision (or action if any) using a control flow.

The last task is to add control flow elements among the elements that are still unlinked.

3.4 Linking Element with Control Flows

At this time of the transformation process, all elements have been created in the activity diagram, but only a few of them have been linked with other elements (the elements created in section 3.3). The final task is to link all the other elements with *ControlFlow* instances. No control flow may be added until all the elements are generated because the control flow source and target is different depending if the actions have been generated from main steps with or without alternative and erroneous steps (a step-by-step example is provided in next section).

This task is, mainly, an algorithmic task. However, it has been possible to define a set of relations to achieve every specific situation at the time to link the elements. These relations are briefly described in next points.

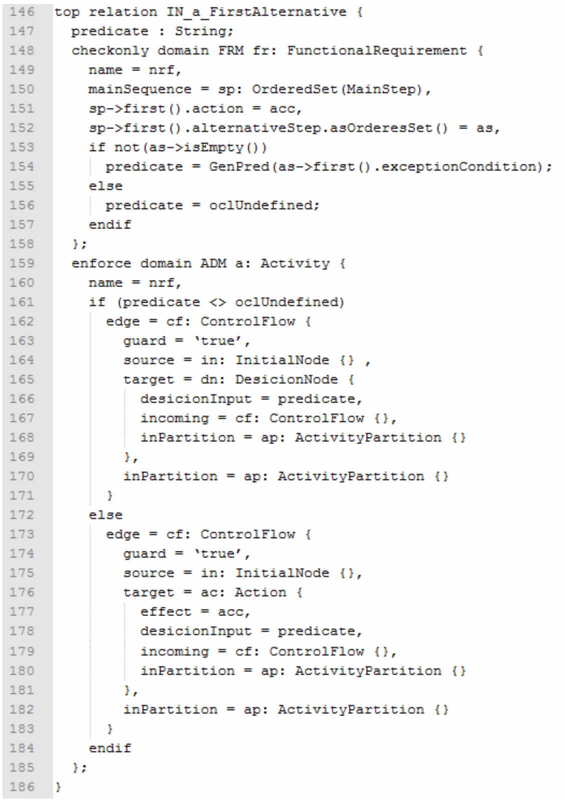
1 Initial node with first element and activity final node with last element
If the first main step of the functional requirement has not an alternative step, then, the initial node is linked with the action generated from the first step. In other case, the initial node is linked with the decision node generated from the exception condition of the first alternative. This relation is showed in figure 8. The same process is applied in the relation to link the last element in the activity diagram with an activity final node.

2 Decision nodes among them
As seen in the functional requirement metamodel (figure 1) all exceptional steps (alternative and erroneous steps) are ordered. So, the next task is to link the decision node generated from the first alternative step of a main step with the decision node generated from the second alternative step of the same main step, the decision node generated from the second one with the decision node generated from the third one, etc. This process is also done with the decisions nodes generated from the erroneous steps.

3 Decision nodes with activities
Next, the decision node generated from the last alternative step of a main step (if any) must be linked with the action element generated from this main step. In the same way, this action element must be linked with the decision node generated.

4 Nodes from one step with nodes from other step
Finally, the last decision node generated from the last erroneous step of the first main step is linked with the first decision node generated from the first alternative step of the second main step, and so on.

At this time, the activity diagram is finished.



```
top relation IN_a_FirstAlternative {
  predicate : String;
  checkonly domain FRM fr: FunctionalRequirement {
    name = nrf,
    mainSequence = sp: OrderedSet(MainStep),
    sp->first().action = acc,
    sp->first().alternativeStep.asOrderesSet() = as,
    if not(as->isEmpty())
      predicate = GenPred(as->first().exceptionCondition);
    else
      predicate = oclUndefined;
    endif
  };
  enforce domain ADM a: Activity {
    name = nrf,
    if (predicate <> oclUndefined)
      edge = cf: ControlFlow {
        guard = 'true',
        source = in: InitialNode {} ,
        target = dn: DesicionNode {
          desicionInput = predicate,
          incoming = cf: ControlFlow {},
          inPartition = ap: ActivityPartition {}
        },
        inPartition = ap: ActivityPartition {}
      }
    else
      edge = cf: ControlFlow {
        guard = 'true',
        source = in: InitialNode {},
        target = ac: Action {
          effect = acc,
          desicionInput = predicate,
          incoming = cf: ControlFlow {},
          inPartition = ap: ActivityPartition {}
        },
        inPartition = ap: ActivityPartition {}
      }
    endif
  };
}
```

Fig. 8. From initial node to first element

4 Case Study

This section describes a case study of the transformation defined in previous sections. The case study chosen is an incident report system. This system is running in a public organism in Spain. Use cases were written one year ago using NDT approach [5][6], which contains all the common elements of the metamodel in figure 1. No additional information was needed to apply our transformation. This system has 9 use cases, however, two of them have been discarded since they describe interactions among a concrete digital sign service that is not available for us. The total number of steps is 46. One activity diagram has been automatically generated from each use case. The total number of activity nodes is 41 and the total number of decision nodes is 15. The complete results of this case study can be found in [1]. One use case has been chosen (Login, from figure 2) to illustrate the transformations described in previous section.

To be able to apply our approach, we have first rewritten the use cases using our XML syntax. The transformations in previous sections have been implemented in a Java open-source tool (also available in [1]). The input is a XML file as the one showed in figure 2 (generated automatically from Sparx Enterprise Architect tool) and the output is a XMI activity diagram file. In following figures, the activity diagram has been loaded also with Sparx Enterprise Architect tool.

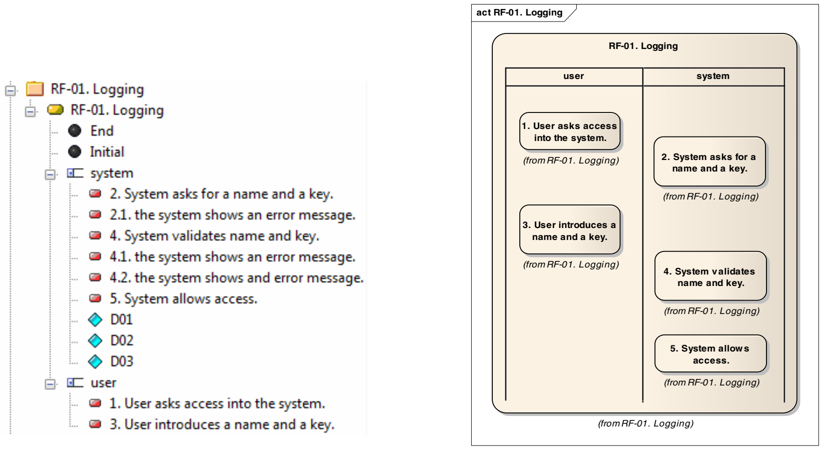

RF-01. Logging
RF-01. Logging
End
Initial
system
2. System asks for a name and a key.
2.1. the system shows an error message.
4. System validates name and key.
4.1. the system shows an error message.
4.2. the system shows and error message.
5. System allows access.
D01
D02
D03
user
1. User asks access into the system.
3. User introduces a name and a key.
act RF-01. Logging
RF-01. Logging
user
system
1. User asks access into the system.
(from RF-01. Logging)
2. System asks for a name and a key.
(from RF-01. Logging)
3. User introduces a name and a key.
(from RF-01. Logging)
4. System validates name and key.
(from RF-01. Logging)
5. System allows access.
(from RF-01. Logging)
(from RF-01. Logging)

Fig. 9. (a) Elements generated from the functional requirement. (b) Actions from main step.

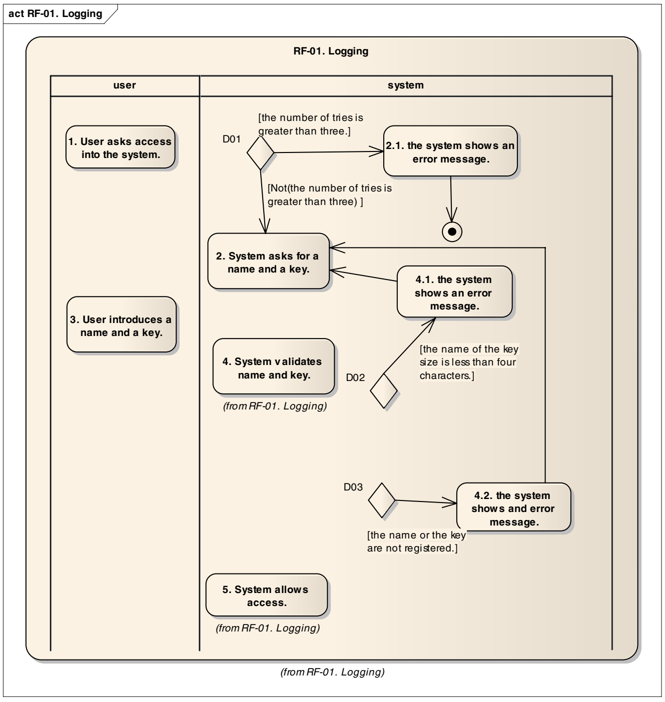
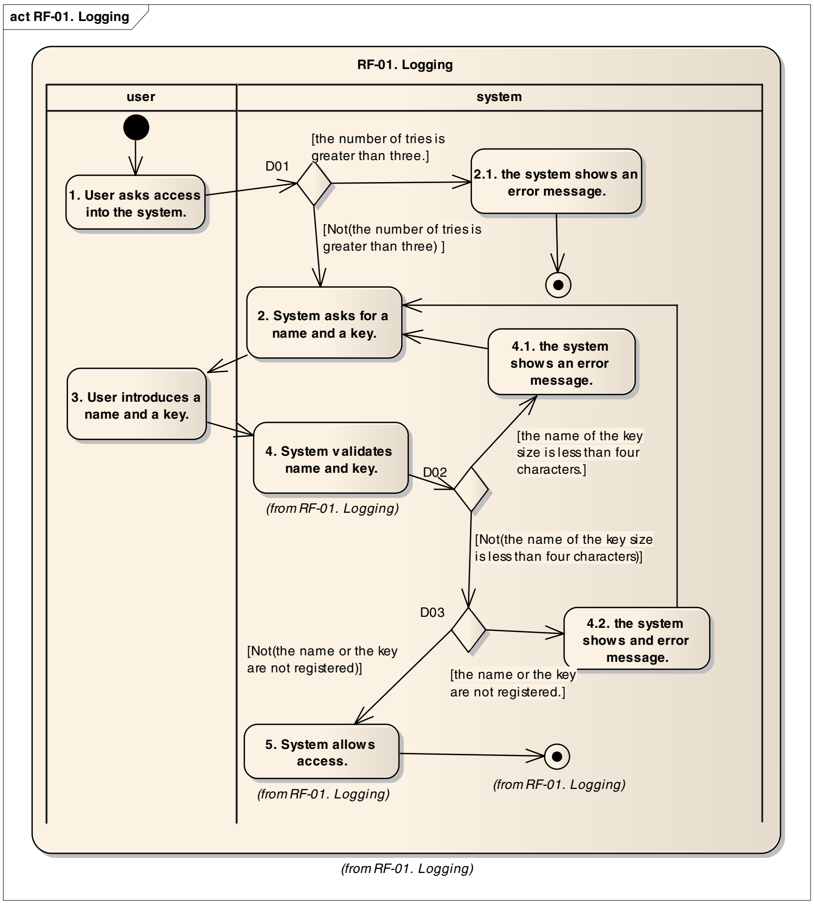

act RF-01. Logging
RF-01. Logging
user
system
1. User asks access into the system.
D01
[the number of tries is greater than three.]
2.1. the system shows an error message.
[Not(the number of tries is greater than three)]
2. System asks for a name and a key.
4.1. the system shows an error message.
3. User introduces a name and a key.
4. System validates name and key.
(from RF-01. Logging)
D02
[the name of the key size is less than four characters.]
D03
4.2. the system shows and error message.
[the name or the key are not registered.]
5. System allows access.
(from RF-01. Logging)
(from RF-01. Logging)

Fig. 10. From exceptional steps to decision nodes and actions

All the elements generated from the login functional requirement are showed in figure 9(a). The result of the relations introduced in sections 3.1 and 3.2 are showed in the incomplete activity diagram of figure 9(b). As described before, one activity, activity partitions and actions generated from main sequence have been added.



act RF-01. Logging
RF-01. Logging
user
system
1. User asks access into the system.
D01
[the number of tries is greater than three.]
2.1. the system shows an error message.
[Not(the number of tries is greater than three)]
2. System asks for a name and a key.
4.1. the system shows an error message.
3. User introduces a name and a key.
4. System validates name and key.
(from RF-01. Logging)
D02
[the name of the key size is less than four characters.]
[Not(the name of the key size is less than four characters)]
D03
4.2. the system shows and error message.
[Not(the name or the key are not registered)]
[the name or the key are not registered.]
5. System allows access.
(from RF-01. Logging)
(from RF-01. Logging)
(from RF-01. Logging)

Fig. 11. Final activity diagram

The result of the relation introduced in section 3.3 is showed in the incomplete activity diagram in figure 10. As described before, a decision node and an action (if any) have been added for each alternative and erroneous step and some elements have been linked with control flows.

Finally, the result of relations described in section 3.4 is showed in the activity diagram in figure 11 and, now, this activity diagram is complete. As described before, all the elements of the diagram, plus an initial and final node, are linked using Control Flow element.

5 Related Works

At the time to write this paper, few works exists dealing with automatic processing of textual requirements from a model-based point of view. Reference [11] introduces a requirement metamodel and a set of transformations (in Kermeta language) to build an executable model from the requirements (defined as a labeled transition system). Functional requirements are treated as black boxes and only pre-conditions and post-conditions are used to generate the executable model. On the opposite, our approach focuses in the scenarios of each use case and not in the ordering of use cases. In [12] a complete requirement model for web system, called WebRE, is introduced. This new model merges two previous approaches: NDT [5] and UWE [13]. However, this model introduces web specific concepts like the browser or the links and navigation, so it is not as generic as the approach introduced in this paper.

There are also approaches which extend activity diagrams to describe use case behavior more precisely. For example, in [16] actions are stereotyped to indicate if they are performed by an actor, by the system or reflect inclusions and extensions. In our approach this goal is achieved using activity partitions. Another related work may be found in [18]. This paper presents a method for describing use cases with activity charts. Support for inclusion and extension is provided. The main difference with our approach is that [18] works with UML Use Case, but not with textual templates. This fact imposes that activity diagrams describes a sequence of use cases, instead of the behavior of one use case. Other papers about modeling sequences of use cases are [19] and [20], which are test-generated oriented.

Another example is [2], which uses language analysis techniques to generate state-charts diagrams from unformatted textual requirements. Reference [21] describes a systematic but manual process to build state-charts from textual requirements. Reference [17] defines an algebraic framework for composing state charts and shows how to leverage the algebraic structure of UML sequence diagrams to get a direct algorithm for synthesizing a composition of state charts out of them. In this paper, scenario diagrams are used to define scenarios, as an artifact to model functional requirement. This process goes from specific (a concrete scenario) to general (a state-chart to describe the whole behavior) while our work starts from general (the behavior of a whole scenario). In our approach, scenarios may be defined as a path obtained from the generated activity diagram.

6 Conclusions

This paper introduced a metamodel for functional requirements and a set of QVT relations to combine functional requirements in the form of use cases with activity diagrams (as described in UML 2.0) in an automatic way. These transformations allow requirement engineers and system users to work with the most valuable representation at any time. However, some drawbacks appear at the time of combining textual requirements with activity diagrams. The most obvious one is the loss of flexibility, for example at the time of adding new kind of result types. The task of adding a new result type implies add a new set of transformation for implementing the right activity diagram fragment for that new result and modify existing transformations for detecting the new result.

The representation of the functional requirement is not limited to XML. This paper has introduced a fragment of XML due it is the input of our supporting tool, as mentioned above. Other implementations may use different representations and it is even possible to design adapters. For example, in the supporting web site [1], a simple Java program to obtain XML from Enterprise Architect project file is provided.

As mentioned in previous sections, the functional requirement metamodel includes a common subset of the elements that may be found in existing functional requirements approaches. The main advantage is that the metamodel may be used with any of those approaches with very little or no changes. Furthermore this metamodel may be easily extended to include additional information like priority, stability of requirements, performing issues, traceability, etc.

Although the transformation process has been defined with QVT-Relational, it has been implemented in Java language. Java allowed us to implement a prototype (also available in [1]) in a shorter time than existing implementations of QVT-Relational or other transformation languages like ATL. However, one of our main ongoing works is to develop a stable version of the tool using one of the cited transformations languages.

Actually, we use the transformations described in this paper as a first step to generate test cases from use cases automatically. Transforming the textual requirements into activity diagrams allows us to apply graph coverage criteria, like covering all nodes or covering all flows, to generate concrete usage scenarios easily. Some papers describing our preliminary work (which was not formalized with metamodels and transformation) are [7][8][9]. Two additional case studies with empirical results are introduced in [10].

Acknowledgements

This work is partially supported by the Ministry of Science and Education (Spain) under the National Program for Researching, Development and Innovation, project QSimTec (TIN2007-67843-C06-03) and REPRIS (TIN2005-24792-E). Authors also wants to remark the help provided for the student who developed the system used in the case study.

References

[1] Supporting web site, `http://www.lsi.us.es/~javierj/`

[2] Boddu, R., Guo, L., Mukhopadhyay, S.: RETNA: From Requirements to Testing in Natural Way. In: 12th IEEE International Requirements Engineering RE 2004 (2004)

[3] Cockburn, A.: Writing Effective Use Cases, 1st edn. Addison-Wesley, Reading (2000)

[4] Dustin, E., Rashka, J., McDiarmid, D.: Quality Web Systems. Performance, Security, and Usability. Addison Wesley, Reading (2002)

[5] Escalona, M.J.: Models and Techniques for the Specification and Analysis of Navigation in Software Systems. Ph. European Thesis. University of Seville. Seville, Spain (2004)

[6] Escalona, M.J., Gutiérrez, J.J., Villadiego, D., León, A., Torres, A.H.: Practical Experiences in Web Engineering. In: 15th International Conference on Information Systems Development, Budapest, Hungary, 31 August – 2 September (2006)

[7] Gutiérrez, J.J., Escalona, M.J., Mejías, M., Torres, J.: Derivation of test objectives automatically. In: Fifteenth International Conference on Information Systems Development (ISD 2006), Budapest, Hungary, 31 August – 2 September (2006)

[8] Gutiérrez, J.J., Escalona, M.J., Mejías, M., Torres, J.: Modelos Y Algoritmos Para La Generación De Objetivos De Prueba. In: Jornadas sobre Ingeniería del Software y Bases de Datos JISBD 2006, Sitges, Spain (2006)

[9] Gutiérrez, J.J., Escalona, M.J., Mejías, M., Torres, J., Torres, A.: Generación automática de objetivos de prueba a partir de casos de uso mediante partición de categorías y variables operacionales. In: XIV Jornadas sobre Ingeniería del Software y Bases de Datos JISBD, Zaragoza, Spain (2007)

[10] Gutiérrez, J.J., Escalona, M.J., Mejías, M., Torres, J., Zenteno, A.H.: A Case Study for Generating Test Cases from Use Cases. Research Challenges in Information Science, Marrakech (2008)

[11] Baudry, B., Nebut, C., Le Traon, Y.: Model-driven Engineering for Requirement Analysis. In: Enterprise Distributed Object Computing Conference, Annapolis, MD, USA (October 2007)

[12] Koch, N., Zhang, G., Escalona, M.J.: Model Transformations from Requirements to Web System Design. In: Webist 2006. LNBIP, vol. 1. Springer, Heidelberg (2007)

[13] Koch, N.: Software Engineering for Adaptative Hypermedia Applications. Ph. Thesis, FAST Reihe Softwaretechnik, Munich, Germany, vol. 12. Uni-Druck Publishing Company (2001)

[14] Ben Achour, C.: Writing and Correcting Textual Scenarios for System Design. In: Natural Language and Information Systems Workshop, Vienna, Austria (1998)

[15] Roubtsov, S., Heck, P.: Use Case-Based Acceptance Testing of a Large Industrial System: Approach and Experience Report. In: TAIC-PART 2006, Windsor, UK (2006)

[16] Kösters, G., Six, H.-W., Winter, M.: Coupling Use Cases and Class Models as a Mean for Validation and Verification of Requirements Specifications. Requirements Eng. 6(1), 3–17 (2001)

[17] Ziadi, T., Hélouët, L., Jézéquel, J.-M.: Revisiting Statechart Synthesis with an Algebraic Approach. In: 26th International Conference on Software Engineering (ICSE 2004), Scotland, United Kindom (2004)

[18] Almendros-Jiménez, J.M., Iribarne, L.: Describing Use Cases with Activity Charts. In: Wiil, U.K. (ed.) MIS 2004. LNCS, vol. 3511, pp. 141–159. Springer, Heidelberg (2005)

[19] Labiche, Y., Briand, L.C.: A UML-Based Approach to System Testing. Journal of Software and Systems Modelling (SoSyM) 1(1), 10–42 (2002)

[20] 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) (March 2006)

[21] Fröhlich, P., Link, J.: Automated Test Case Generation from Dynamic Models. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 472–491. Springer, Heidelberg (2000)

[22] Object Management Group. The UML Superstructure (2007), `http://www.omg.org`

[23] Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process, USA. Object Technology. Addison-Wesley, Reading (1999)

[24] D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach, USA. Addison-Wesley Professional, Reading (1998)

[25] Hartmann, J., Vieira, M., Foster, H., Ruder, A.: TDE/UML: A UML-based Test Generator to Support System Testing. In: 5th Annual International Software Testing Conference, India (2005)

[26] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Final Adopted Specification (2007), `http://www.omg.org`