LAB 1:
Requirements Modeling
Preparations and instructions

Björn Regnell

October 12, 2014

1 Introduction

1.1 Purpose

This document provides instructions on how to prepare for and run a computer lab session on requirements modeling. The lab session illustrates how computer-supported requirements modeling can help in requirements engineering, including elicitation, specification and validation. *The preparations in this document should be completed before the actual lab is run.*

1.2 Background

Requirements engineering is a dynamic process where knowledge and perception of an imagined future system (of systems) is evolving over time. During this evolution we can capture the knowledge and creative ideas we elicit in various ways, depending on how we imagine the (later) usage of this information: we could create and use, for example, spread sheets, post-it notes, emails, wikis, video clips, mockups, sketches, diagrams, mathematical specifications, etc.

If we want to keep track many different types of inter-related requirements and if we see a future benefit of more structure beyond just a flat list, we can use *requirements models* where requirements are expressed using relevant (meta-) **entities**, **attributes** and **relations** that can capture what we want to model.

The open source tool **reqT** enables scalable requirements modeling, ranging from small models of a couple of features to large models containing elaborate structures of thousands of requirements. In this lab you will learn how to get started with requirements modeling through the use of reqT and reflect on how you could model the requirements of your own project with reqT.

2 Preparations

Before the lab session, please complete all preparations below and bring requested items to the lab. In particular you need to make sure that you can access the text files you prepare below at your lab session computer.

1. **Read reqT intro.** Read the section "Introduction to reqT" on this web page:
   `http://reqt.org/documentation.html#intro`

2. **Do the reqT "hello world".** Run the reqT hello world example on your computer, as explained here:
   `http://reqt.org/documentation.html#hello`

3. **Draw a context diagram.** Draw a context diagram that is relevant to your development project. Make the drawing on paper and bring to the lab. The context diagram should include entities outside of the system you are building that are communicating directly with the system under development.

4. **Create a context model.** Transform your context diagram into a reqT model, analogous to this example:

   ```
   Model(
     Product("hotelApp") interactsWith (
       User("receptionist"),
       User("guest"),
       System("telephony"),
       System("accounting")))
   ```

   Saved the model in a text file called `context.scala`

5. **Create a list of features.** Make a list of at least 5 features relevant to your project, divided into 2 sections, analogous to the example below. Use the exact formatting and indentation according to the example with hash tag before the name of each group and an asterisk before the name of each feature:

   ```
   # receptionFeatures
     * groupCheckIn
       Check in many guests arriving in bulk.
   # adminFeatures
     * roomForecast
       Calculate room allocation prediction.
     * exportAccounting
       Export accounting data to external system.
   ```

   Saved the list of features in a text file called `feat.txt`

3 Lab instructions

3.1 Start reqT

Download the reqT.jar file from `http://reqT.org` and start reqT in a terminal window using this command: `java -jar reqT.jar` as shown in Figure 1. Make sure you have the text files from your preparations available in the same directory as you start reqT.

Figure 1: The reqT console output after starting reqT in a terminal window.

In the reqT console these keystrokes and commands work as follows:

<Ctrl+A> to move the cursor to the beginning of a line.
<Ctrl+E> to move the cursor to the end of a line.
<Ctrl+K> to "kill" the characters from the cursor to the line end.
<Arrow Left/Right> to move the cursor backward/forward within a line.
<Arrow Up/Down> to move backward/forward in the command history.
<TAB> to invoke code completion.
`:q` to quit reqT when all threads and windows have been exited.
`:help` to get help on console commands.
`sys.exit` to exit reqT and kill all threads and any windows without saving.

Try out the commands above in the reqT console. Use the `:help` command to figure out how to enter paste mode. Paste some text when in paste mode using mouse right-click. End paste mode with <Ctrl+D>. What is the shortest, unambiguous abbreviation of the enter-paste-mode-command? _____

3.2 Create and update models using the reqT console

Type in the following lines in the reqT console after the reqT> prompt. Press enter after each line. The + operator is used to add elements to a model, and the ++ operator is used to append one model to another.

```
reqT> val m1 = Model(Req("a") has Spec("sss"))

reqT> val m2 = m1 + (Req("b") has Prio(2))

reqT> val m3 = Model(Stakeholder("x") requires Req("a"))

reqT> ((m3 + Stakeholder("y")) ++ m2).size
```

> What is the integer value result of the last evaluation?
> ```
> ((m3 + Stakeholder("y")) ++ m2).size
> ```
>
> _____

Continue to type in the following lines in the reqT console after the reqT> prompt. Press enter after each line. The for keyword is used to make a for-loop. The yield keyword is used in a for-comprehension to construct a sequence of values. The val keyword is used to declare a name that refers to an immutable value (a constant) and the var keyword is used to declare a name that refers to a mutable value (a variable). The − operator is used to remove elements from a model. With the transform method you can make transformations of specific elements in a model.

```
reqT> var m4 = (for (i <- 1 to 10) yield Req("r"+i)).toModel

reqT> (1 to 10).map(i => Req("r"+i)).toModel //alternative to above

reqT> m4 = Model(Stakeholder("x") requires m4)

reqT> m4 = m4 - Req("r7")

reqT> m4 -=  Req("r3")

reqT> m4.pp   //pretty-print m4

reqT> m4 = m4.transform{case Req(id) => Feature(id) has Status(ELICITED)}
```

> What is the size of the m4 model after the above transform?
> ```
> m4.size
> ```
>
> _____

3.3 Investigate the reqT metamodel

A reqT model can be viewed as a vector of elements. Elements can be entities, attributes and relations. An entity has an id of type String. An attribute holds a value that can be of different types. A relation connects and entity via a link of a certain RelationType to a submodel that, in turn, can contain elements. A part of the reqT metamodel is shown in Figure 2.
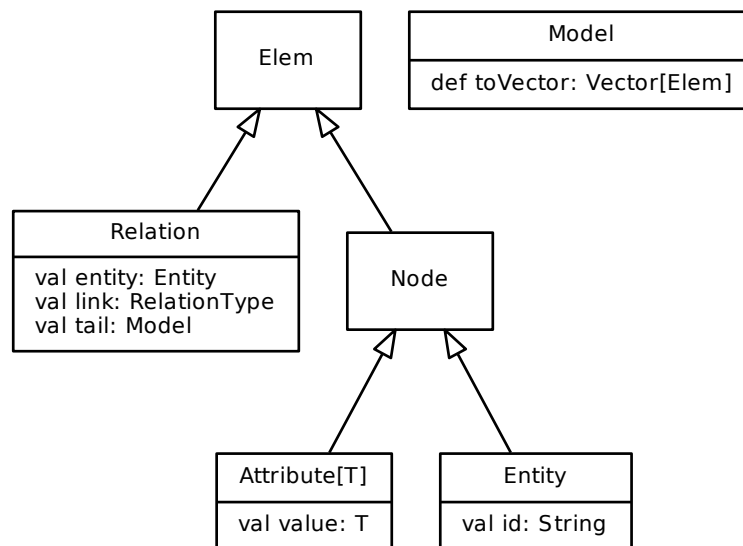
Figure 2: Some classes in the reqT metamodel.

Investigate what different entity types, attribute types and relation types that the reqT metamodel contains, using the evaluations in the reqT console below.

5

The collect method collects selected parts of a model into a vector. In the last evaluation below we collect the integer 1 for each occurrence of a Meta entity and sum all ones.

```
reqT> reqT.metamodel.  // Press <TAB> after the dot

reqT> reqT.metamodel.ent // Press <TAB> after the t

reqT> reqT.metamodel.entityTypes

reqT> reqT.metamodel.entityTypes.size

reqT> reqT.meta.model.pp

reqT> reqT.meta.model.collect{case Meta(_) => 1}.sum
```

How many different entity types, attribute types and relation types are there respectively in the reqT metamodel?

_____

How many `Meta` entity concepts are there in the reqT metamodel?

_____

The meta model elements can be used in many different ways. There are no restrictions on how to combine the elements, except for these three basic rules:

1. **Attribute identity**. A model or submodel can only contain at most *one* attribute of a specific type at its top level. However, the same type of attribute can co-exist if they reside in different submodels of the same model.

2. **Entity identity**. A model or a submodel can only contain at most *one* entity with a certain id and a certain relation at its top level. If you add an entity with the same id and the same relation at the top level of a model or submodel, it will merge the elements of each submodel, and if the above rule applies then the last same-typed attribute will be overwrite the former.

3. **Leaf entity has empty submodel**. The `has`-relation is special, as a leaf entity that has no relations to any subelements is equivalent to an entity with a has-relation to an empty submodel.

6

Try the subsequent statements in the reqT console, where `Prio` is an attribute type and `Req` is an entity type and `has` and `requires` are relation types. Make sure you can explain the evaluation results in relation to the rules above. Write the number of the rule(s) (1 – 3) that is/are in effect besides each evaluation.

```
reqT> Model(Prio(1), Prio(2))

reqT> Model(Req("x") has (Prio(1), Prio(2)))

reqT> Model(Req("x") has Prio(1), Req("y") has Prio(1))

reqT> val m6 = Model(Req("x") has (Req("sub1"), Prio(1), Prio(2)))

reqT> m6 + (Req("x") has (Req("sub2"),Prio(3)))

reqT> Model(Req("x") has ())

reqT> Model(Req("x") has Prio(1), Req("y") requires Prio(1)) - Prio(1)
```

Create a model with two stakeholders `a` and `b`, both requiring the same two features `x` and `y`. The stakeholders' features shall have different priorities: a thinks x is of higher priority than y, while b thinks the opposite. Declare a constant called `prio` that refers to the model. Write the reqT code that describe your model below and then test it in the reqT console.

When you create a reqT model, you actually create an immutable, tree-like data structure that consists of computational objects in the Java Virtual Machine (JVM) runtime environment. When you encode reqT models you are actually coding in the Scala programming language that compiles to JVM byte code. The reqT metamodel classes are actually Scala classes and the reqT language is embedded in Scala. The `reqT.jar` file includes the Scala compiler and the reqT console wraps the so called Scala Read-Evaluate-Print-Loop (REPL), which enables any

7

Scala code snippet to be interactively compiled and run on a line-by-line basis at the reqT prompt.

3.4 Access elements in models using paths

A model is represented using a recursive data structure where relation elements can include submodels, which in turn can include relation elements that include submodels. The recursive nature of the model data structure thus enables hierarchical, tree-like requirements models of arbitrary depths. The submodels and elements of submodels at different levels can be accesses using paths that are constructed using the / operator called *enter*.

A *head* is a start of a relation that combines an entity with a relation type using dot notation, such as `Feature("x").has` and `Stakeholder("b").requires`

Paths begin with a sequence of *heads* separated by / and may end with either (1) a head, (2) an entity, (3) an attribute type or (4) an attribute. The has relation is special: in paths an entity without any relation type is interpreted as an entity with a `.has` relation.

```
Try these path examples in the reqT console:

reqT> val p1 = Stakeholder("a").requires/Feature("x").has/Prio(42)

reqT> p1.depth    //write down the depth: _____

reqT> val p2 = Stakeholder("b")/Feature("x")/Prio(21)

reqT> p2.init

reqT> val p3 = p2.init/Feature("sub")/Prio(9)

reqT> p3.toModel

reqT> var pm = Vector(p1,p2,p3).toModel

reqT> pm/Stakeholder("b").has

reqT> pm/Stakeholder("b")

reqT> pm/Stakeholder("a").requires/Feature("x")/Prio

reqT> pm = pm + Stakeholder("a").requires/Feature("x")/Prio(1)

reqT> pm.leafPaths

reqT> pm.leafPaths.map(_.depth).max   //write down the max depth: _____
```

Create a random model using these commands, and note its max depth: _____

```
reqT> val rm = rndModel(10)

reqT> rm.pp

reqT> rm.leafPaths.map(_.depth).max
```

Write down the path needed to access the value of one of the deepest attributes:

```
reqt> rm/_____
```

If a path ends with an attribute type, then it refers to the corresponding value that is boxed by that attribute in a model. If the attribute is not available when the path is applied to a model with the enter operator / then a default value is produced.

If you want to check the absence or presence of a value you can use the get method on a model. The get method takes a path as parameter and returns a value boxed in an instance of the Option class; if there was, e.g., an integer value of 42 then the option class evaluates to Some(42) or if there is no value then it evaluates to None. You can get the actual value of an Option instance by calling the get method.

Try these attribute type path examples in the reqT console:

```
reqT> val m = Model(Req("x") has Prio(1), Req("y"))

reqT> m / Req("x") / Prio

reqT> m / Req("y") / Prio     //write the default Prio value here _____

reqT> m.get(Req("x")/Prio)

reqT> (m/Req("x")).get(Prio).get     //same effect as previous

reqT> m / Req("x") get Prio     //equivalent to previous

reqT> m / Req("x") get Prio get

reqT> m / Req("y") get Prio get
```

What happens if you try to call the get method on a non-existing value, as in the previous evaluation?

_____

3.5 Load and save files from the reqT console

You can load text files into strings and save strings into text files using commands similar to:

```
val s = load("myFile.txt")
"my String".save("myStringFile.txt")
```

The `ls` command prints a list of files in the working directory. The `pwd` command prints the path of the working directory. The `mkdir("tmp")` command creates a new directory called `tmp` and `cd("tmp")` changes working directory to the directory `tmp` if it exists.

You can also serialize a model to a binary file, which for large files may take less space compared to a text file and it may also be quicker to save and load a large binary model to and from disk compared to a text model. To serialize a binary model to disk, just call the save method on a model. To load a binary model from disk use the `Model.load("filename.reqt")` command. It is recommend to use the file suffix `.reqt` to show that it is a serialized binary file.

Check that you have the files from the lab preparations in your working directory and load your context model and convert it to a reqT model from a string using the following evaluations:

```
reqT> ls
context.scala
prio.txt

reqT> var m = load("context.scala").toModel

reqT> m = Model(Title("My Cool Product"), Section("Context") has m)

reqT> m.toString.save("context-v2.scala")
Saved string to file: C:/Users/bjornr/tmp/context-v2.scala

reqT> m.save("context.reqt")
Model serialized to file: context.reqt
```

Create a large random model using the command:

```
val r = rndModel(52,2)  //max 52 at top level and then max 26 etc.
```

and compare the binary model size on disk with the string model text file size, by checking the file sizes in your OS. If the files get too big or it takes too much time, reduce the number of max elements at top level.

Number of elements in the model: `r.size` _____
Size of binary file: `r.save("big.reqt")` _____
Size of text file: `r.toString.save("big.scala")` _____

3.6 Edit requirements with the reqT ModelTreeEditor GUI

The reqT ModelTreeEditor is a graphical user interface for navigating and updating reqT model. You start the editor with the `edit` command. You can start many ModelTreeEditor windows.

You can also preload a ModelTreeEditor instance with a model, e.g. called `m`, by simply passing it as a paramter to the edit command, e.g. `edit(m)`. The ModelTreeEditor is shown in Figure 3.
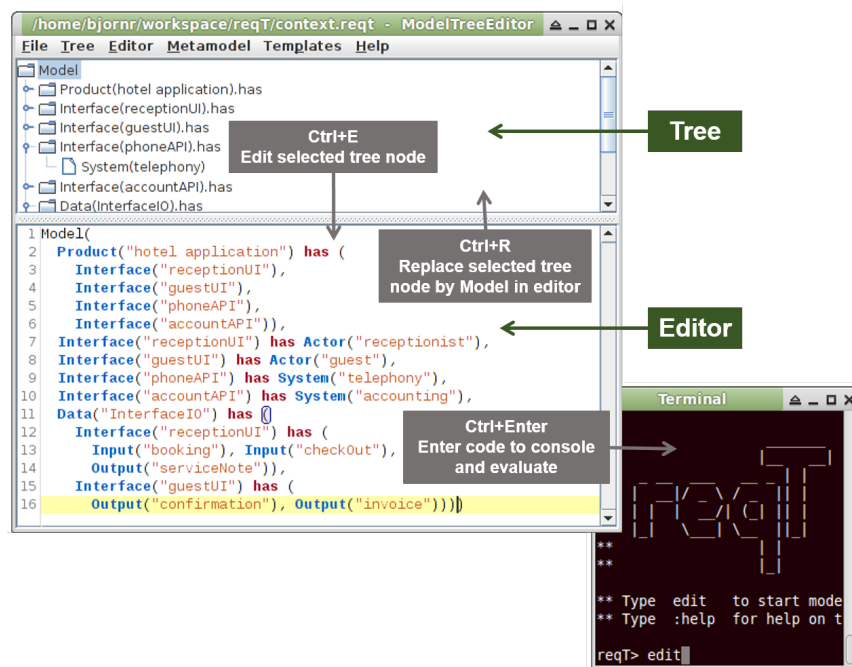

Figure 3: The reqT ModelTreeEditor graphical user interface.

The reqT ModelTreeEditor gui has two parts:

- **Tree**. The Tree displays a tree view of a model, where each head node can be collapsed or expanded using the right and left arrows or by clicking on the handle by the left of the folder symbol.
- **Editor**. The Editor

<Explain the GUI parts>
<Explain the shortcuts>
<Load your context.scala model>

11

4   Additional quests if more time is available

```
m => m.transform{case Item(i) => Feature(i); case Text(i) => Gist(i)}
_.transform{case a: Attribute[_] => NoElem}
javax.swing.JOptionPane.showMessageDialog(null,"Hello Swing!")
```

< Look at the reqT source code at GitHub >

5   Conclusion