Transformational Support for Model-Based Testing – from UML to QML

Fredrik Abbors[1], Tuomas Pääjärvi[1], Risto Teittinen[2], Dragoş Truşcan[1], and Johan Lilius[1]

[1] Dept. of Information Technologies, Åbo Akademi University, Joukahaisenkatu 3-5, 20520, Turku, Finland
Email: fabbors|tpaajarv|dtruscan|jolilius@abo.fi

[2] Nokia Siemens Networks, Linnoitustie 6, 02600, Espoo, Finland
Email: risto.teittinen@nsn.com

Abstract. Model-Based Testing (MBT) has lately gained increased popularity due to the benefits that it provides in terms of automation of the test generation process. There are several tools capable of applying MBT using behavioral models of the system under test (SUT). However, as complex systems are specified using different perspectives, like architecture, data, behavior, which benefit from proper tool support especially in the Unified Modeling Language (UML) community, there is a gap between the graphical capabilities and expressiveness of the UML-based and MBT tools. In this paper, we present an approach in which the information describing different perspectives of the system under test is collected from the UML models of the SUT and transformed into input for a MBT tool, to be used for automatic test generation, execution, and evaluation.

1 Introduction

Testing has become an important activity of the software development, which according to some studies can take more than 40 percent of the development resources [1]. Recently, Model-Based Testing (MBT) has gained momentum by advocating the use of models for automatic test generation, execution, and evaluation. The basic idea in MBT is to check the conformance of a system implementation, aka *system under test* (SUT), against a model specifying its behavior. Currently there are established approaches for performing MBT from behavioral models, and the corresponding tools are available (an exhaustive list can be found in [2]). However, complex systems are often described using several perspectives like behavioral, architectural, data, which should also be considered during the testing process. Thus, one should include such information in the specification of the SUT and use it for test generation. Most of the current MBT tools provide support for modeling the behavior of the SUT, but few are able to consider different types of information or the information is expressed using textual format.

In recent years, the Unified Modeling Language (UML) [3] has become one of the most popular languages for system specification. UML provides a collection of diagrams that can be used to model different perspectives of the system and there is a plethora of tools that can be used for UML-based specification. Therefore, we would like to take advantage of the existing UML tool support for specifying the SUT, and later use the

resulting models for test generation. In order to have a smooth transition between the UML and the MBT tools, transformations should be used for collecting necessary information and creating the input for the testing tools. Such transformations will also enable a smooth integration of the UML tools in the MBT tool chain.

In this paper we employ a transformational approach to reduce the gap between UML and MBT tools. As an instance of our approach, we exemplify with a concrete transformation that translates UML models into input for a tool used for automated test design, namely Conformiq's Qtronic [4]. The transformation is defined in a general manner with respect to UML, such that it can be applied to different UML modeling tools. The approach is exemplified with a telecommunications case study[3].

1.1 System Modeling with UML

In our system specification, we apply a systematic approach for creating a set of models from the requirements in order to capture different views of the SUT. UML is used as specification language. The main purpose of the modeling process is to describe several perspectives of the SUT, like requirements, architecture, data, behavior, such that the information contained in these models can be used not only for development, but also for automated test generation using specialized test generation tools. As the models are derived from requirements, it is important to track how different requirements reflect in the models, on different perspectives and on different abstraction levels. It is also important to propagate requirements through the test generation and test execution processes, such that one can verify what parts of the models and consequently, which requirements have been tested and validated.

Our approach is divided into five (horizontal) phases as depicted in Fig. 1. The first phase deals with *requirements analysis*, and has as main purpose the identification of the system requirements from the related standards and protocol specifications. The second phase structures the features and functional requirements of the system, using the UML and Systems Modeling Language (SysML) [6] diagrams, respectively. In the third phase, the main usage scenarios of the system are specified in a use case model both textually and via *message sequence charts* (MSCs). The fourth phase looks into deriving the domain, behavioral and data models of the system starting from the previously specified models. The process is iterative so phases three and four can be visited several times and the models are constructed

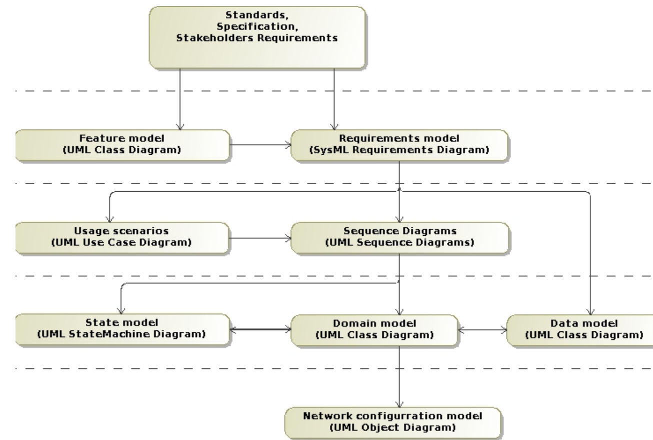


Fig. 1. System modeling process

[3] detailed examples can be found in [5]

incrementally. The last phase deals with creating specific system configurations by instantiating the concepts abstracted in the precedent models. Several models result after applying this process and they are described in more detail in the following sections.

A set of rules and guidelines are defined for ensuring the quality of the resulting models, by checking that the models are syntactically correct [7]. These rules also ensure that the models are consistent to each other and moreover, that they contain the information needed in the later phases of the testing process. Tool support is provided for automatically verifying these rules using the *Object Constraint Language* (OCL) [8].

1.2 The Qtronic tool

Conformiq Qtronic [4] is a tool for model-driven test case design, which generates tests from the specification of the SUT using various coverage criteria. The *Qtronic Modeling Language* (QML) is used for describing different aspects of the SUT like input/output ports and messages, message structure, behavior, etc. The behavior of the SUT can be described either textually in Java or graphically using UML statemachines. State machines are drawn with the Qtronic Modeler, whereas the rest of the system specification is done in QML, following object-oriented principles. As such, the SUT is specified as a `class` that can have *attributes* and *methods*. The methods are later used as the action language in the state model. In Qtronic, the messages sent or received by the SUT are defined as `records`, that is user-defined types that can contain variables, methods, operators, and nested types. Records can inherit other records. `Inbound` and `Outbound` ports describe which records may be sent or received. Several input/output ports can be specified. The ports and the messages that are exchanged constitute the interface of the SUT to its environment. This interface is declared under the `system`-block. In QML, a state machine is a separate thread. Other threads can be declared by extending the `Thread`-class or by declaring a class that implements the `Runnable` interface. One important aspect is that multiple instances of the same state machine can be executed in parallel in order to allow testing of concurrent behavior. The communication between threads can be implemented either by message passing or or by shared variables.

2 Deriving the QML Model from UML

The main purpose of the transformation is to automatically extract the necessary information from the UML models and use it for creating the model used by Qtronic for test generation. The transformation can be seen as composed of several parts (see Fig. 2), each described in the following subsections. Throughout this section, we will use excerpts from a telecommunications case study modeling a *Mobile Switching Server* (MSS), which will be used as the SUT. The MSS is a central element of 2nd and 3rd generation mobile telecommunication networks. The MSS connects calls between mobile

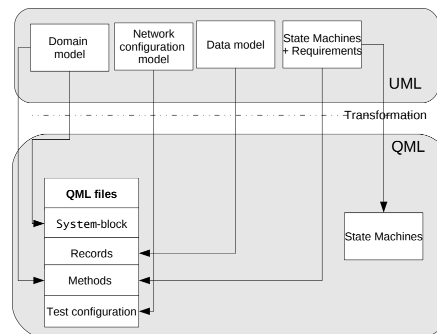


Fig. 2. Transformation from UML to QML

phones and fixed network. It takes into account movement of mobile phones at the time of call set up and during the calls.

2.1 Generating the Interfaces and Ports of the System

In our UML modeling approach, a domain model (Fig. 3) is used to depict the components of the domain, and how they are connected via interfaces. Domain components can communicate with each other via messages belonging to various protocols. For cutting down the complexity, the messages sent and received on each protocol level, for instance Mobility Management (MM), are modeled separately by `interface` classes.

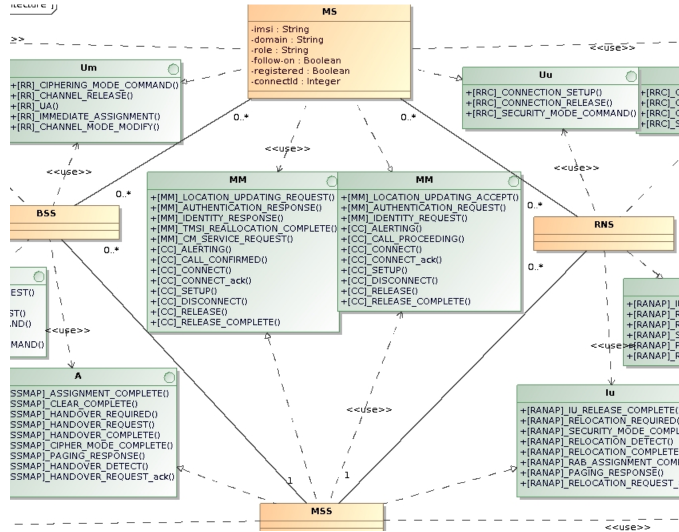


Fig. 3. Class diagram representing a Domain Model

In QML, interfaces are described with the `system`-block which describes ports that can be used to communicate with the environment, and what messages on each port can be sent and received. The `Inbound` ports declare messages to be received by the SUT from the environment, whereas the `Outbound` ports declare messages to be sent from the SUT to the outside world. As such, the ports of the SUT are obtained directly from interface classes in the domain model (see Fig. 3). `Inbound` messages are taken from the interface realization offered by the SUT, and `Outbound` messages are taken from the interface realization used by the SUT. The name of the ports will be composed of two components, the direction and the interface name, respectively. UML operations are listed as messages transferred through the ports, and they are declared elsewhere as `records`. Section 2.2 describes in more detail how QML records are created and how record variables are defined. The partial result of applying the transformation on the `MM` interfaces in Fig. 3 is shown in Listing 1.1.

```
//System block example
system {
  Inbound MM_in: location_updating_request, authentication_response, identity_response, TMSI_reallocation_complete,
        CM_service_request, alerting, call_confirmed, connect, connect_ack, setup, disconnect, release,
        release_complete;
  Outbound MM_out: location_updating_accept, authentication_request, identity_request, alerting, call_proceeding,
        connect, connect_ack, setup, disconnect, release, release_complete;
}
```

Listing 1.1. Example of QML `system`-block for Fig. 3

2.2 From UML Data Models to QML Message Types

In order to generate proper test cases from system models, a description of the data used by the system is needed. We model these data explicitly via class diagrams. We refer to this model as the *data model*. The data model of the SUT depicts each message type as a class, whereas the parameters of the message are represented as class attributes. We structure the message definition based on their corresponding protocols and use inheritance to model common parameters for a given message. Figure 4 presents an example of a UML data model from our case study.

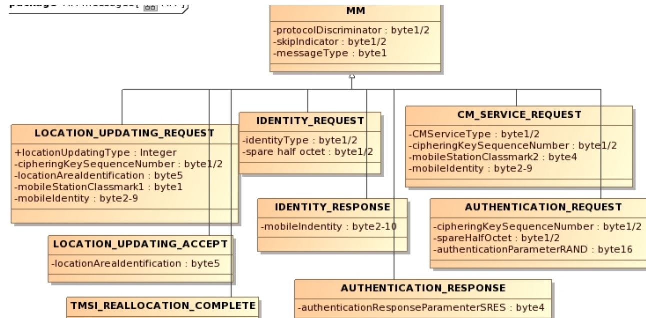


Fig. 4. Message declaration in UML

In QML, messages are described as records that are used for communicating with the environment. QML records are user-defined types similar to classes. The fields of a record can be of type: `byte`, `int`, `boolean`, `long`, `float`, `double`, `char`, `array`, `String`, or of another record type. In our transformation, records are obtained from classes in the UML data model. Attributes of the UML classes are transformed into the fields of the record. Inheritance in UML is reflected in QML using the `extends` relationship. For instance, the `location_update_request` record in Listing 1.2 is obtained from the `LOCATION_UPDATE_REQUEST` class in Fig. 4 following the described approach. We point out that the model does not indicate value ranges of the fields. Instead, the value ranges are checked later on by the protocol codecs provided by the test system.

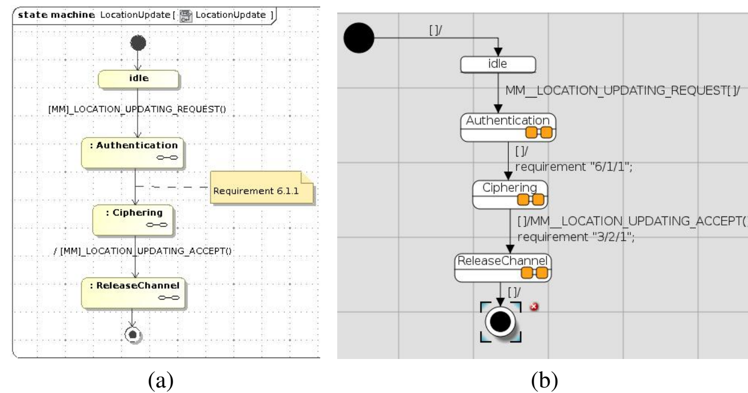
```
//example record of LU
record MM_messages{
  public String protocol_discriminator;
  public String skip_indicator;
  public String message_type;
}
//record inheritance
record location_updating_request extends
        MM_messages{
  public int location_updating_type;
  public String ciphering_key_sequence_number;
  public String location_area_identification;
  public String mobile_station_classmark_1;
  public String mobile_identity;
}
```

Listing 1.2. QML record declaration

2.3 Mapping the UML State Machine to the QML State Machine

As mentioned previously, the behavior of the SUT can be specified in Qtronic either textually in QML or graphically using a restricted version of the UML state machines. For simplicity, we have chosen to follow the latter option as the target of our transformation. Thus, the transformation is basically a matter of transforming the UML state machine into the corresponding state machine used by the Qtronic tool, which in practice is equivalent with a transformation at the XMI-level. Figure 5(a) shows a UML state machine[4], whereas Fig. 5(b) shows the same state machine transformed to QML. As one can notice,

[4] The state machine has been created in the MagicDraw UML modeling tool



(a) (b)

Fig. 5. Example of UML state machine (a) and its equivalent in Qtronic Modeler (b)

there is a strong similarity between the two models, however with small differences. For instance, both state and sub-state machines are supported and propagated at Qtronic level. In UML, triggers and actions are declared as methods (selected only from the operations of the interface classes in the domain model).

In QML, triggers are implemented by messages (record instantiations) received on a certain port, whereas actions can be seen as methods of the SUT class definition. This approach allows one to perform further processing of the system data before sending a given message to the output port[5]. The method generated for the `MM__LOCATION_UPDATING_ACCEPT()` in Fig. 5(b) is shown in Listing 1.3.

```
void MM__LOCATION_UPDATING_ACCEPT(){
 MM__LOCATION_UPDATING_ACCEPT location_updating_accept;
 MM_out.send(location_updating_accept);
 return;
}
```

Listing 1.3. Example of a generated QML method

2.4 Propagating System Requirements from UML to QML

In our modeling approach, requirement models are used for structuring and interrelating the requirements of the SUT using SysML Requirement Diagrams. The requirements are defined on several levels of abstraction following a functional decomposition and they can be related to other requirements or linked (traced) to different diagrams and model elements in UML, for example to state machines and class diagrams. Figure 6 shows an example of a requirement diagram. The main purpose of tracing requirements is in analyzing which parts of the specification "implement" different requirements and it will allow later on to propagate these requirements from models to tests.

Qtronic provides support for requirement coverage during test generation. Requirements are associated to state models, more precisely to the actions on transitions via the `requirement` statement. This is achieved by either attaching a requirement as an action on a transition, or in a method in the QML textual notation file. Basically, the re-

[5] If needed, additional QML instructions can be manually inserted in the generated methods, before the `.send()` statement

quirements in Qtronic are tags that are used to trace if a specific action in the state model has been covered by the generated test cases.

In the current approach, we are interested in collecting from the UML state models only those requirements that are attached to state transitions. However, nothing prevents us from collecting the requirements from other UML diagrams, as well, if needed. The requirements are captured from UML transitions and placed on the corresponding transition in QML, using the `requirement`-statement (see Fig. 5(b)). Hierarchy can also be propagated from UML to Qtronic by analyzing how the requirements are structured in the requirement models. For instance, requirement "6/1/1" in Fig. 5(b) is obtained from requirement "6.1.1" in Fig. 6, which at UML-level was attached to the transition `Authentication`→`Ciphering` in Fig. 5(b). Upon importing a QML model in Qtronic, requirements are displayed hierarchically in Qtronic's UI (Fig. 7) from where requirements-based test derivation can be pursued.

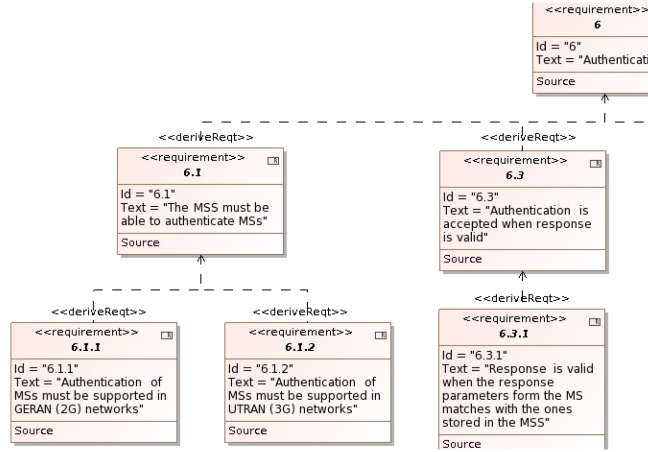


Fig. 6. Example of SysML requirements model

2.5 Generating the QML Test Configuration

In UML, we use object diagrams (see Fig. 8) for specifying network configurations instantiated from the domain model (see Fig. 3). Multiple instances of the same class can be instantiated and properties specific to each instance are initialized. Such diagrams can be used for defining the testing setup in the QML model. In this case the `:MSS` depicts the SUT, while the other instances represent the test environment.

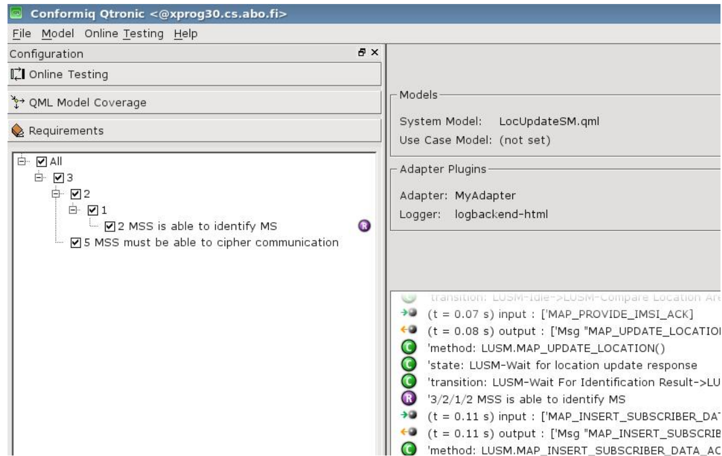


Fig. 7. Example of Qtronic's UI with requirements

As illustrated in Figs. 3 and 8, a :MS (mobile subscriber) is a user of the SUT and communicates with the SUT via the MM interface. The test configuration in our example consists of three mobile subscribers. In QML, a subscriber is modeled as a record (Fig. 9(a)) with the same attributes as in the domain model (Fig. 3). The test configuration is translated into QML in two steps: first the properties of the test components are extracted from the UML domain model and are declared in the constructor method of the SUT specification class (MSS in our case) as shown in Fig. 9(b). In the second step, each test component is instantiated and its properties are initialized with the values taken from the configuration diagram (Fig. 9(c)). The test components are stored into an array, which is later used for starting as threads separate versions of the SUT.

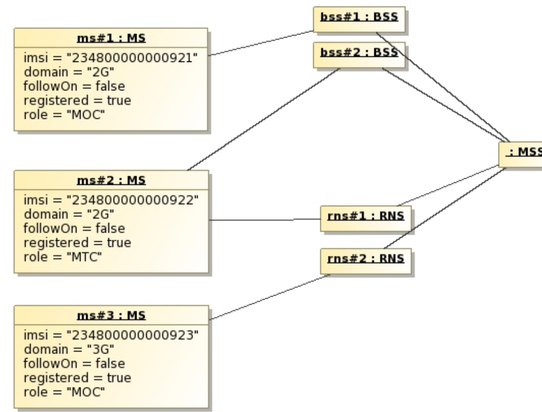


Fig. 8. Example of configuration diagram

2.6 Assigning the State Model to the SUT Specification

At this point, the only thing left to be done is to connect the SUT class specification to the graphical state model. This is done by calling the constructor method for the state machine. Once the state machine has been constructed, the concurrency of the SUT can be tested by starting (via the Thread.start() method) separate execution threads for each test component (i.e., subscriber) (Listing 1.4). The approach allows for different mobile subscribers to concurrently communicate to the SUT using different configuration parameters. This is needed as in telecom systems such as MSS one needs to test the presence of multiple mobile subscribers interacting with the MSS (in practice, one MSS can serve up to few millions of users). In addition, we need to test calls between pairs of subscribers, where one call requires two subscribers, i.e. the one who calls and the one who receives the call (known as A and B subscribers).

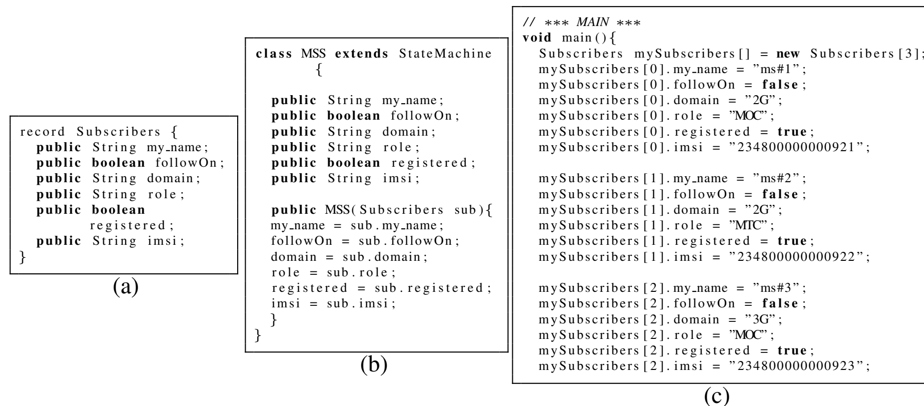
```
for(int i = 0;i<=2; i++){
 MSS mss = new MSS(mySubscribers[i]);
 Thread t = new Thread(mss);
 t.start();
}
```

Listing 1.4. State machine instantiation in Qtronic

3 Tool Support

The mappings from UML to QML discussed in the previous section have been defined in such a way that any set of models based on UML and containing the required information can be used as a source of the transformation. However, as various UML tools implement the UML metamodel differently, the implementation of the transformation, from UML to QML, had to be customized for the specific tool used.

In our case, the MagicDraw [9] modeling tool has been used for creating (via the process described in Fig. 1) and for assuring the quality of the UML models via validation



```
record Subscribers {
  public String my_name;
  public boolean followOn;
  public String domain;
  public String role;
  public boolean
        registered;
  public String imsi;
}
```

(a)

```
class MSS extends StateMachine
        {

  public String my_name;
  public boolean followOn;
  public String domain;
  public String role;
  public boolean registered;
  public String imsi;

  public MSS(Subscribers sub){
  my_name = sub.my_name;
  followOn = sub.followOn;
  domain = sub.domain;
  role = sub.role;
  registered = sub.registered;
  imsi = sub.imsi;
  }
}
```

(b)

```
// *** MAIN ***
void main(){
  Subscribers mySubscribers[] = new Subscribers[3];
  mySubscribers[0].my_name = "ms#1";
  mySubscribers[0].followOn = false;
  mySubscribers[0].domain = "2G";
  mySubscribers[0].role = "MOC";
  mySubscribers[0].registered = true;
  mySubscribers[0].imsi = "234800000000921";

  mySubscribers[1].my_name = "ms#2";
  mySubscribers[1].followOn = false;
  mySubscribers[1].domain = "2G";
  mySubscribers[1].role = "MTC";
  mySubscribers[1].registered = true;
  mySubscribers[1].imsi = "234800000000922";

  mySubscribers[2].my_name = "ms#3";
  mySubscribers[2].followOn = false;
  mySubscribers[2].domain = "3G";
  mySubscribers[2].role = "MOC";
  mySubscribers[2].registered = true;
  mySubscribers[2].imsi = "234800000000923";
```

(c)

Fig. 9. Example of subscriber record (a); `StateMachine`-class (b); `main` method of the SUT specification class (c)

rules written in OCL. Once completed, the models are exported to XMI [10] format. A Python script is used for automating the transformation. The script consists of seven separate modules (Python-source files). A *parser* module parses the XMI file and structures the information in a tree-like representation (using the *lxml*[11] library). This representation is used by the other modules for creating QML-related information. A *state machine creator* module uses the collected information in the parser to generate the QML state machine. A *record creator* module uses the collected information in the parser to generate the QML records. A *method creator* module uses the information collected by the parser to generate the textual annotations for ports, methods, and for the `main` function. A *definition module* declares patterns of the QML elements that are used for creating the QML models. A *manifest creator* module generates the QML manifest file, which lists the generated files for a given project, whereas a *main* module is used to control and to invoke the functionality of the other modules during the transformation.

The script is structured in such a way that it easily can be modified and extended to suite input from other modeling tools or generate models for other MBT tools. For example, if another modeling tool is used, only the parser module has to be modified.

4 Related Work

Several research works discuss transformational approaches for model-based testing. In the following we will only discuss several of them which we consider related to our approach. A transformation from test models, expressed using the UML 2 Testing Profile (U2TP), to TTCN-3 is proposed in [12]. This transformation is closer to code generation and has as the main purpose the creation of test specifications from test models, while our transformation translates UML models into input for a test derivation tool. A transformational framework in the context of MDA is suggested in [13] with the main purpose of deriving tests from system models expressed in a restricted version of UML, called *essential UML*. A set of transformational rules are specified between the source and the

target language, the *essential Test Modeling Language* (eTML), a variant of the U2TP. A similar approach is described in [14], where test models are integrated along with a MDA development process. Again, the main focus of the transformation is on deriving the test specifications expressed in TTCN-3 at different levels of the MDA process. The source models for the proposed transformations are expressed using a UML profile for Enterprise Distributed Object Computing (EDOC) or in Sun's J2EE. This work is closer to ours since it generates the final test specifications by combining model transformation and code generation.

5 Conclusions

We have presented an approach in which the UML models used for system specification are automatically transformed into input for a test generation tool. Our work had two main goals: on the one hand, we wanted to show that one can benefit from the graphical capabilities and expressivity of the UML specifications during the test generation process. Such an approach circumvents the currently limited modeling capabilities of certain test generation tools. We also wanted to take advantage of the validation capabilities of existing UML tools, which currently exceed those of the MBT tools. On the other hand, we intended to provide a solution for improving the transition between different steps of the MBT process, by suggesting a set of mappings between UML and QML that can be used to automate the transition and thus improve the existing tool chain.

References

1. Scott, E., et al.: The Alignment of Software Testing Skills of IS Students with Industry Practices. Journal of Information Technology Education (2004)
2. Utting, M., Legeard, B.: Practical Model-Based Testing - A Tools Approach. Denise E. M. Penrose (2007)
3. Object Management Group: Unified Modeling Language (OMG UML) ver. 2.1.2 Document formal/2007-11-02, available at `http://www.omg.org/spec/UML/2.1.2/`.
4. Conformiq: Qtronic `http://www.conformiq.com/`.
5. Abbors, F., et al.: A semantic transformation from UML models to input for the qtronic test design tool. Technical Report 942, TUCS (Apr 2009)
6. Object Management Group: Systems Modelling Language (OMG SysML) ver. 1.0 Document formal/2007-09-01, available at `http://www.omg.org/spec/SysML`.
7. Abbors, J.: Increasing the Quality of UML Models Used for Automatic Test Generation. Master's thesis, Åbo Akademi University (2009)
8. Object Management Group: Object Constraint Language v2.0. OMG. (May 2006) `http://www.omg.org/spec/OCL/2.0/PDF`.
9. NoMagic: MagicDraw `http://www.magicdraw.com/`.
10. Object Management Group: XML Metadata Interchange (2007) `http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm`.
11. lxml homepage: `http://codespeak.net/lxml/`.
12. Zander, J., et al.: From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. In: TestCom. (2005) 289–303
13. Busch, M., et al.: A model transformer for test generation from test models. In: 9th International Conference on Quality Engineering in Software Technology (CONQUEST). (2006)
14. Born, M., et al.: Combining System Development and System Test in a Model-Centric Approach. In: Rapid Integration of Software Engineering Techniques (RISE). Volume 3475 of LNCS. Springer Berlin/Heidelberg (2005) 132–143