Model-Based Testing with the Escalator Tool

MASS SOLDAL LUND

Mass Soldal Lund is Research Scientist at SINTEF ICT

Software testing is the chief method for ensuring correctness and quality of software systems. Model-based testing is testing where test cases are derived automatically from models of the system under test, and the test cases themselves are treated as a model. Model-based testing can therefore make software testing more effective, and facilitates both reuse of test cases and testing of system models. In this article we present Escalator, a tool for model-based testing based on UML 2.x sequence diagram specifications. Further, we show how this tool can be applied in the context of the Rational Unified Process (RUP) for software development. The presentation is case driven, and is based on examples from the Parlay X Web Service specification.

1 Introduction

Most system development methodologies recommend testing as the method for improving the correctness and quality of the software developed. In its most general sense, software testing means that an executable program is subjected to experiments where the program is observed in order to find out how it behaves under certain circumstances. Typically, the circumstances are input to the program and what is observed is the output of the program, in which case the process is called functional testing.[1] The observations are used to ensure that the program operates in accordance with the specification of the program.

In what is usually referred to as model-based testing (or specification-based testing), tests are derived from formal or semi-formal models (specifications) of the system under development – often state machines, statecharts, or as in our case sequence diagrams – and the tests are used for investigating whether the program conforms to the model (specification) [4].

Another aspect of model-based testing is that the tests themselves are treated as a model – a test model [5]. The test model distinguishes itself from, e.g., test scripts by being on the same abstraction level as the system model (specification) instead of the abstraction level of the implementation. This means that the test model uses the same primitives and has the same semantics as the system model. Such test models are also called abstract tests [4].

Important system development methodologies such as the Rational Unified Process (RUP) [6], and other variants of the Unified Process, emphasize use case-based and scenario-based development. In such development, requirements to the system are specified as usage scenarios which then become the basis for system design, implementation and testing. Also agile methods like Agile Model Driven Development (AMDD) [7] and Test Driven Development (TDD) [8] recommend scenario- or use case-based specifications.

Sequence diagrams are good for specifying usage scenarios and therefore easily find their place in this kind of development. There are in particular two reasons they are well suited for this: 1) They focus on the interaction in the system, and 2) they are partial specification, as opposed to complete specification, in the sense that they specify example behavior instead of the full behavior of a system. This in itself motivates test generation from sequence diagrams, because it combines the approach of model-based or specification-based testing with the approach of use case- or scenario-based development. In [4] is shown how a model-based testing approach fits together with RUP and with agile methods, and in [9] an agile methodology based on modeling with UML and model-based testing is outlined. Both RUP and agile methods recommend writing tests based on scenario descriptions. In the approaches cited above, usage scenarios (use cases) are formalized with UML, and tests are generated automatically from these formalized scenarios instead of being written manually.

To bring matters a step further, we argue for testing not only implementations, but also specifications. In an iterative model-based development process, like e.g. RUP, both requirement specifications (use cases) and design specifications are likely to become more and more refined as the project loops through the iterations. In the same way as we would be interested in investigating whether an implementation conforms to the

1) *There are of course other types of testing as well (see e.g. [1][2][3] for different overviews). For example, if the circumstances are the hardware the program is executed on and what is observed is how fast it performs, we can talk about performance testing. In the following, however, we concentrate on functional testing.*

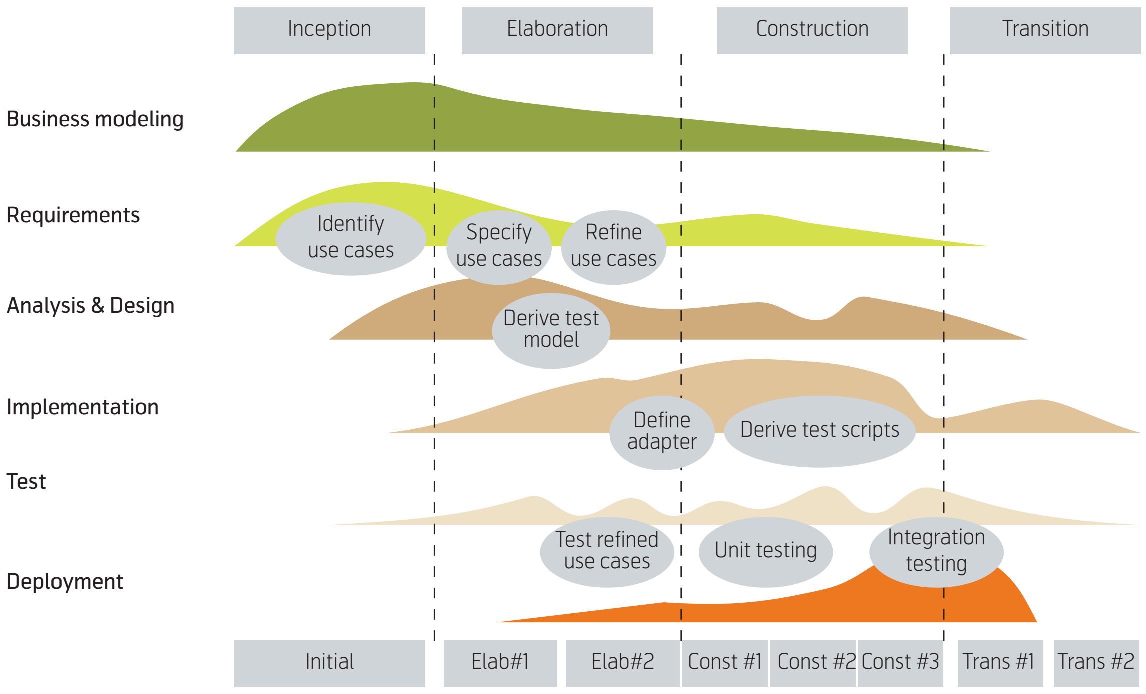


Figure 1 Model-based testing in the Rational Unified Process (RUP)

specification, we should also be interested in investigating whether a refinement of a specification conforms to the original specification. The implementation will most likely be based on the most refined specification, and we should therefore gain an advantage by keeping conformance throughout the refinement steps. We treat a sequence diagram specification as a program for the purpose of testing, and this way we can use testing as a means for investigating whether a refinement of a sequence diagram specification is correct. We refer to this as refinement testing.

Escalator [10], [11] is a computerized tool supporting model-based testing. The tool is based on UML 2.x [12] sequence diagram specifications and the STAIRS method for system development with sequence diagrams [13], [14]. The tool generates abstract tests from sequence diagram specifications, producing test models that themselves are formalized as sequence diagrams. With Escalator, such test models can also be executed against sequence diagram specifications. The tool therefore both supports model-based testing in the traditional sense, and provides a method for refinement testing.

In the next section we go through the imagined development of a web portal for mobile phone users, and demonstrate how the Escalator tool is applied for performing model-based testing in the development. At the end of the article we summarize and provide conclusions.

2 Model-Based Testing in Development of a Web Portal

We define a case inspired by examples given in the Parlay X Web Services specification [15].[2)] We imagine a telecom company developing a web portal providing a number of services to their mobile phone users. These services, and hence the web portal, are based on web services provided by Parlay X. We follow the development through its first phases, with a focus on the use of model-based testing in this development.

2.1 Development Process

The development of the web portal is based on the Rational Unified Process (RUP). RUP is structured around two dimensions. Figure 1 shows the typical presentation of RUP with its phases, workflows, and iterations [6]. One dimension progresses along the lifespan of the system development, divided into the four phases *inception*, *elaboration*, *construction* and *transition*; the other dimension is activities, or disciplines, such as *requirements*, *design*, *implementation* and *test*, that run across all the phases. This structure comes from the recognition that iterative and incremental approaches reduce the risks involved in a software development project, and the recognition that requirements and designs usually undergo changes and refinements as a development project proceeds.

The process is highly iterative, with the possibility of several iterations within each of the phases. As can be

2) *Specifically Figure 2 in Part 2, Figure 3 in Part 4, Figure 1 in Part 6, and Figure 2 in Part 7 of [15].*

seen from the figure, each iteration includes all cross-cutting activities. The process emphasizes component-based development and continuous integration as a way of reducing risk. This also leads to a natural partitioning of the work, since development of components can be associated with iterations

RUP has a high emphasis on testing. It is a use case-driven approach that prescribes using use cases for capturing requirements. Further, these use cases form the basis for the implementation and testing of components, in such a way that testing becomes a natural part of each iteration. In the RUP figure we have also given the approximate placement of the activities involved in model-based testing [4], [10]. In this article we focus on the elaboration phase, but we start by saying something about the inception phase, and we will also say something about the construction phase towards the end.

2.2 Inception – Identify Use Cases

The inception phase is where the first ideas and plans for the system are conceived and sketched. An important part of this phase is to gather and organize the requirements of the system. RUP recommends using use cases for this task, so in modeling terms the requirements should be organized in use case diagrams. Figure 2 shows a use case diagram for the web portal.

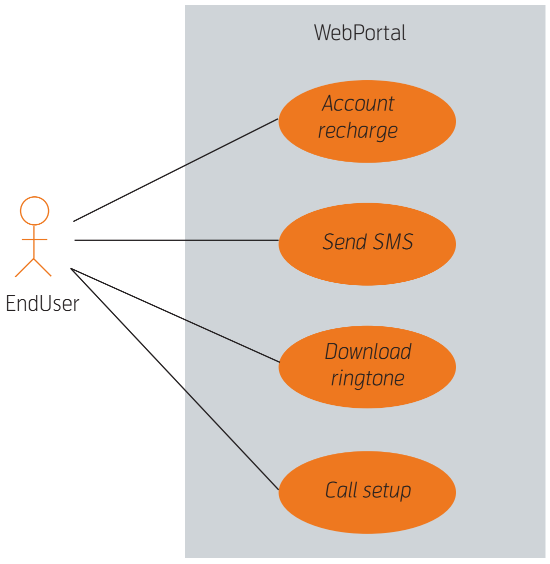


Figure 2 Requirements, specified as use cases, of a web portal providing services to mobile phone users

The requirements to the portal are four services which the end user should have access to: *Account recharge* (for pre paid mobile phone subscriptions), *Send SMS* (from the portal), *Download ringtone* (with payment), and *Call setup* (between two mobile phones from the portal).

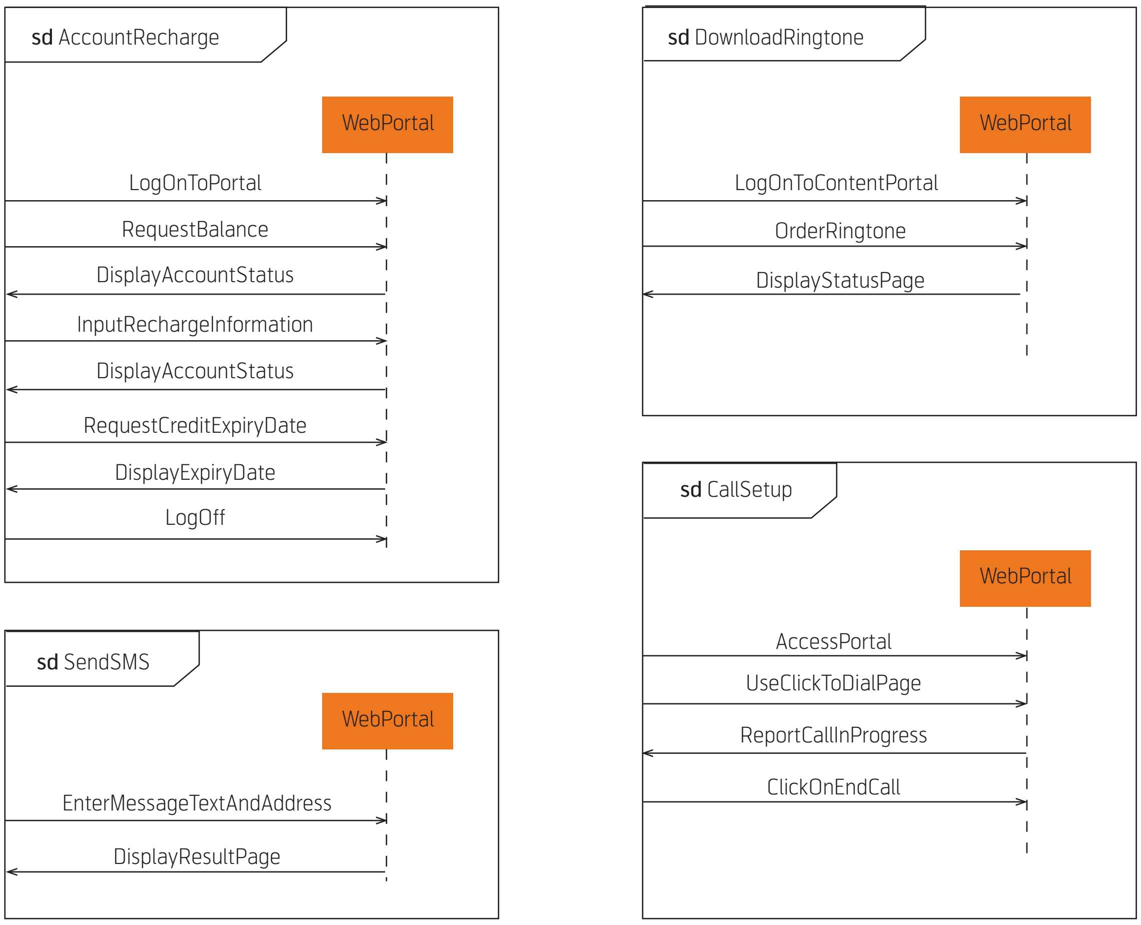
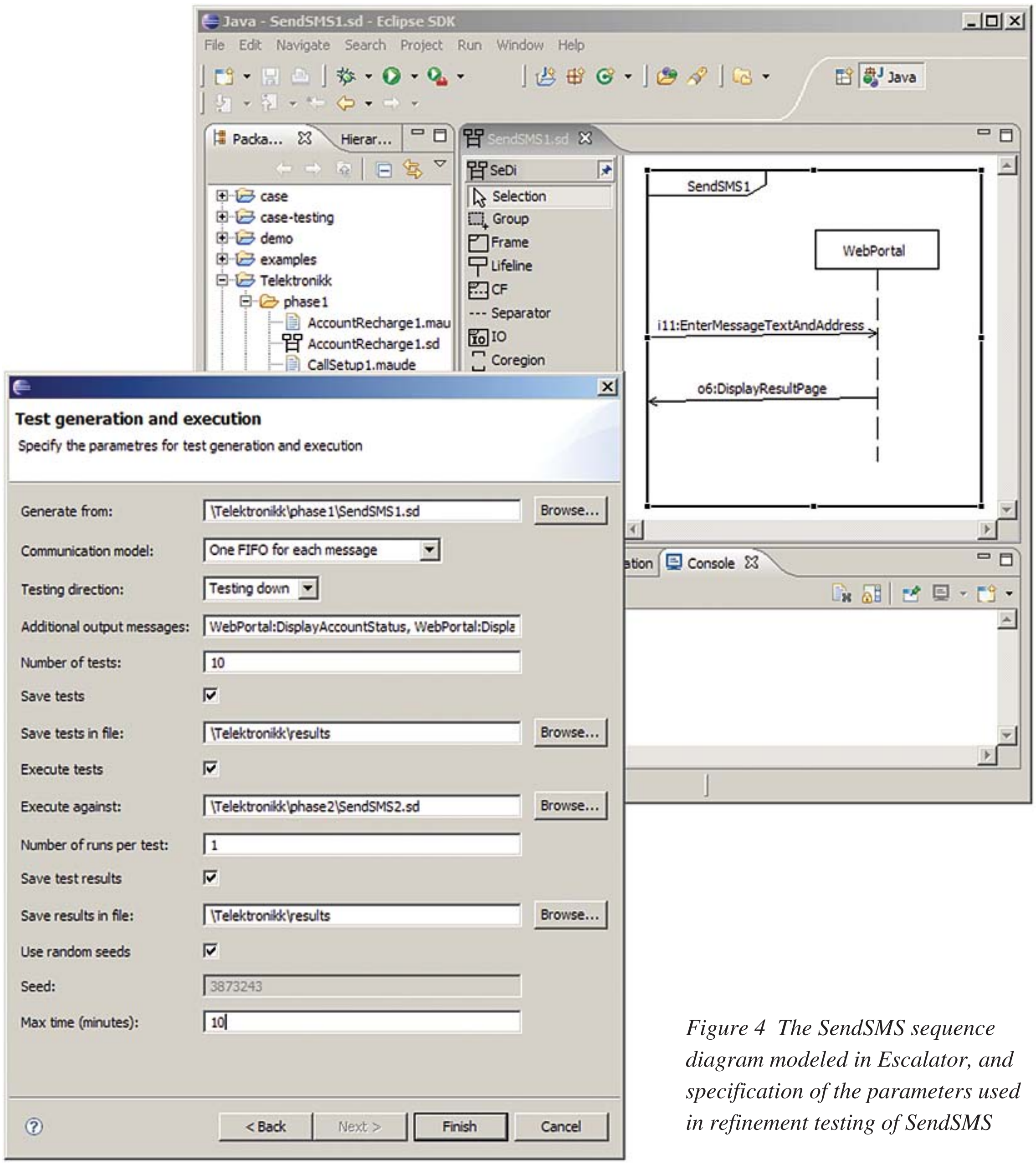


Figure 3 Sequence diagrams specifying the use cases in greater detail. The end user is not shown in these diagrams



Test generation and execution

Specify the parametres for test generation and execution

Generate from:	\Telektronikk\phase1\SendSMS1.sd
Communication model:	One FIFO for each message
Testing direction:	Testing down
Additional output messages:	WebPortal:DisplayAccountStatus, WebPortal:Displa
Number of tests:	10
Save tests	✓
Save tests in file:	\Telektronikk\results
Execute tests	✓
Execute against:	\Telektronikk\phase2\SendSMS2.sd
Number of runs per test:	1
Save test results	✓
Save results in file:	\Telektronikk\results
Use random seeds	✓
Seed:	3873243
Max time (minutes):	10

Figure 4 The SendSMS sequence diagram modeled in Escalator, and specification of the parameters used in refinement testing of SendSMS

2.3 Elaboration

2.3.1 Specify Use Cases

In the elaboration phase the use cases should be evolved, i.e. specified in more detail.[3)] Our preferred way to specify use cases is to make sequence diagrams that characterize the interaction involved in each of them. In the first iteration of the elaboration phase we therefore make sequence diagrams that specify the interaction between the end user and the web portal needed in order to provide the use cases of Figure 2. The four sequence diagrams are presented in Figure 3. For convenience, the end user is not shown explicitly in the diagrams.[4)] Instead we have the web portal interacting with the frame of the diagrams, which represents the environment of the system. In our case, of course, the environment of the web portal is the end user.

2.3.2 Derive Test Model

These sequence diagrams do not tell the full story, and will later be refined. But before we do that we want to produce a test model. A test model is a representation of the tests we later want to perform, and one of the sources of the test model should be the identified use cases. If the use cases were specified using only text, then we obviously would have to define the test model manually. However, one of the benefits of model-based testing is that when we have specified the use cases in a formal language like the

3) *It should be noted that this is of course not the only activity of the elaboration phase, but the activity relevant for the sake of this presentation.*

4) *This has to do with the input format of the Escalator tool.*

sequence diagrams presented above, the test model can be generated automatically.

Escalator [10], [11] is a tool that produces test models from sequence diagrams. Given a sequence diagram, Escalator will generate a suite of what in [4] is called abstract tests. An abstract test can be seen as a specification of a test program or a description of a test using the same language and/or vocabulary (operations and values) as the model describing the system. Hence, Escalator produces test cases in the form of sequence diagrams. An advantage of this is that we can understand the test cases using the same interpretation of the specification language as we use for understanding the system specification. Another advantage is that we have test cases that are independent of specific technologies, such as platforms and programming languages.

The Escalator tool generates the test cases using a randomized algorithm based on an enhanced version of a testing theory called input-output conformance testing (ioco) [2]. The algorithm builds the test cases by simulating executions based on the specifications (the sequence diagrams specifying the use cases). The simulated system is given stimulus, and its possible outputs observed. At the bottom of the tests cases, verdicts are assigned based on whether the paths leading there consist of outputs specified as legal outputs or not. That the algorithm is randomized means that the input to the simulated system is chosen randomly, and that choices the simulated system has to make are randomized. The reason for randomizing the algorithm is that systematically applying every possible input and exploring every branch of the simulated system can result in an explosion in the number of test cases.

Figure 4 shows a screen shot of the Escalator tool. We see the *SendSMS* sequence diagram modeled in an integrated editor [16], and the specification of the parameters applied in the test generation[5)] from that diagram. We do not go into detail on all the parameters, but two of them are worth noting; the additional output messages, and the number of tests. We want the test cases to be able to handle all possible output of the system even though they are generated from different diagrams. For each test generation we therefore specify additional output messages, which means we specify messages that the system may output but that are not present in the diagram we are generating from. For the case of *SendSMS*, the additional output

Diagram	#tests
AccountRecharge	8
SendSMS	2
DownloadRingtone	4
CallSetup	6

Table 1 Number of distinct tests generated from the diagrams of Figure 3

messages are *DisplayAccountStatus*, *DisplayExpiryDate*, *ReportCallInProgress*, and *DisplayStatusPage*, i.e. the output messages of the other diagrams in the specification.

For each of the four diagrams we specified the generation of ten test cases. However, because the test generation algorithm is randomized instead of exhaustive, and because not all the diagrams have ten paths, some of the same test cases were generated multiple times. The number of distinct tests generated from the diagrams is summarized in Table 1.

Figure 5 shows two out of a total of 20 test cases generated from the four sequence diagrams shown in Figure 3. *Test1* is generated from the sequence diagram *SendSMS*, while *Test2* is generated from *DownloadRingtone*. The lifeline ‘tester’ in these test cases can be seen as the specification of a test program / test component which interacts with its environment. The environment of the test program is of course the system under test. In this interaction, the test program will alternately provide stimulus to the system under test and observe its output.

Based on observations of the system under test, the test program will reach a verdict. It should be noted that one of the possible observations a test program can make is that there is no output from the system under test. Theoretically we can say that the test program can observe the absence of output, while in practice a timeout will occur in the test program while it is waiting for output from the system under test. In the test diagrams, these timeouts are represented as actions[6)] labeled with θ’s.

Three different verdicts may be given in a test diagram: **pass**, **fail**, and in addition **inconc** (for inconclusive). These test verdicts reflect positive behavior (behavior specified as legal), negative behavior (behavior specified as illegal) and inconclusive behavior

5) *Actually, we specify that the tool should do test generation and test execution at the same time. We will say more about the test execution in the next section.*

6) *In UML sequence diagrams, internal actions in a lifeline can be specified in a rectangle on the lifeline. In the standard, such actions are defined as a variant of* ExecutionSpecifications.

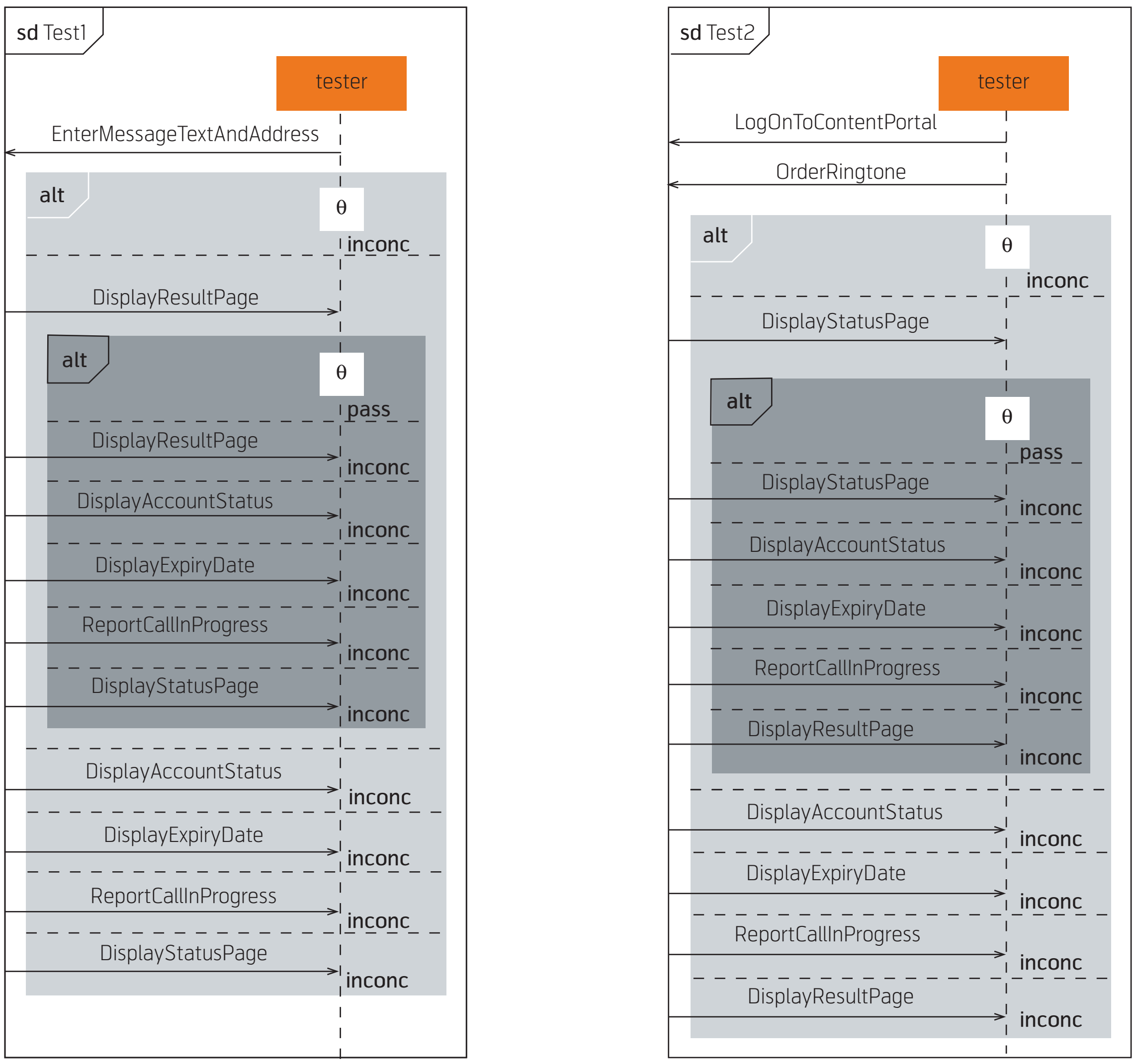


Figure 5 Test cases generated with the Escalator tool from the sequence diagrams SendSMS (Test1) *and* DownloadRingtone (Test2)

(unspecified behavior) in the specification the test case is generated from. The verdicts are used in the test execution to determine whether or not the system under test is a correct refinement of the specification. In general, correct refinement can mean two things: reducing the inconclusive behavior by making it positive (which corresponds to adding functionality to the system) and reducing the positive behavior by making it negative (which corresponds to making design choices).

2.3.3 Refine and Test Use Cases

In the second iteration of the elaboration phase, the use cases from Figure 3 are refined into more detailed sequence diagrams. These sequence diagrams (shown in Figure 6 through Figure 9[7)]) not only specify the interaction between the web portal and the end user, but also the interaction between the portal and different Parlay X web services.

It will often not be obvious that a refined specification like this is consistent with the original specification (though in a small case like this it can be straightforward). By applying the refinement testing functionality of Escalator, we can automatically check whether or not this more detailed specification is a correct refinement of the specification we started with. This is done by executing the abstract test cases generated from the original specification against the refined specification. Each such test run will return **pass** or **fail** as verdict[8)] and the sequence of events leading to the verdict. The screen shot of Escalator in Figure 4 shows how this can be done together with the test generation. There we have specified that each test generated from the original *SendSMS* diagram

7) *These sequence diagrams are taken from [15], Parts 2, 4, 6, and 7.*

8) *These verdicts are not necessarily the same verdicts as in the test case diagrams. A test run verdict is based on an analysis of the verdict of the test case diagram together with information on whether or not the test run has entered a region of negative behavior and whether or not the test run is complete.*

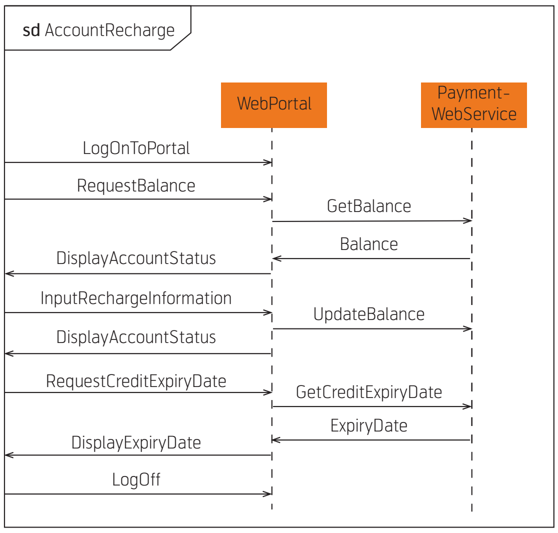


Figure 6 Refinement of the sequence diagram AccountRecharge *where also the interaction between the Web Portal and the Parley X service is shown*

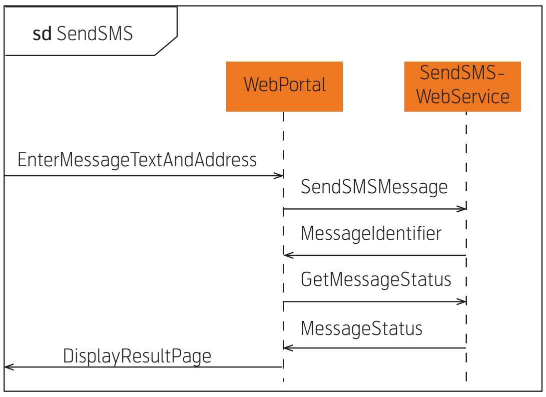


Figure 7 Refinement of the sequence diagram SendSMS *where also the interaction between the Web Portal and the Parley X service is shown*

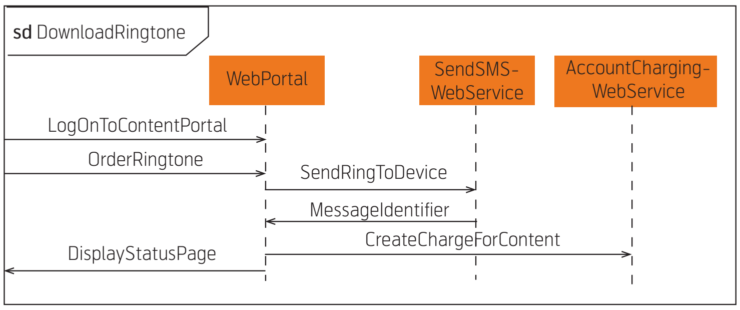


Figure 8 Refinement of the sequence diagram DownloadRingtone *where also the interaction between the Web Portal and the Parley X service is shown*

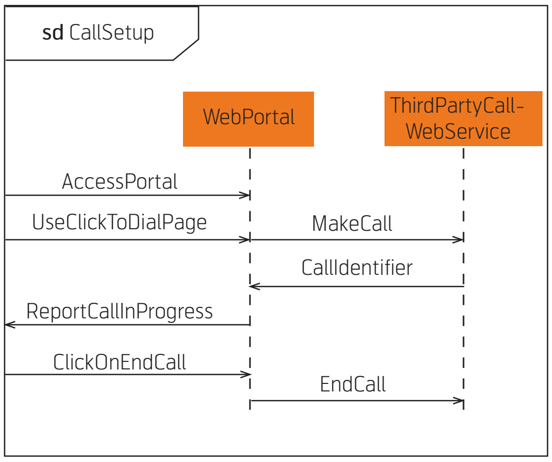


Figure 9 Refinement of the sequence diagram CallSetup *where also the interaction between the Web Portal and the Parley X service is shown*

(Figure 3) should be executed against the refined *SendSMS* diagram (Figure 7), with the number of test runs equal to one for each test.

The results of executing *Test1* and *Test2* from Figure 5 against the new specification of *SendSMS* in Figure 7 and the new specification of *DownloadRingtone* in Figure 8, respectively, are shown in Figure 10. We can see that both test runs resulted in a verdict **pass**. This means we get a higher confidence in the consistency of the new specification with the old specification. In addition, each test run gives a trace representing the interaction between the test and the specified system during the test execution, where a trace is a sequence of events. In these traces an exclamation mark '!' before a message name denotes the event of transmitting the message, while a question mark '?' before the message denotes the event of receiving the message. As before, θ represents timeout, or absence of output from the tested system.

After running all 20 test cases of the test model against the appropriate sequence diagrams – and getting the verdict **pass** in all cases – we conclude that the new, refined specification of the use cases is indeed a correct refinement, and hence conforms to the old specification.

2.4 Construction – Define Adapter and Derive Test Scripts

The construction phase is where the main part of the implementation of the system is made. Providing an implementation of the web portal is outside the scope of this article. We do, however, give some indications on how we could proceed to make use of model-based testing in the further development of the web portal.

We can imagine that (part of or the full) web portal has been implemented in accordance with the specification given above. We would like to test the implementation to see if it conforms to the specification it is supposed to implement. In this situation we would obviously like to reuse the test model we already have derived from the specification. In the elaboration phase we executed the abstract test cases of the test model directly against the sequence diagrams

Test1:
pass: ⟨!EnterMessageTextAndAddress, ?EnterMessageTextAndAddress, !DisplayResultPage, ?DisplayResultPage, θ⟩

Test2:
pass: ⟨!LogOnToContentPortal, ?LogOnToContentPortal, !OrderRingtone, ?OrderRingtone, !DisplayStatusPage, ?DisplayStatusPage, θ⟩

Figure 10 Results of executing Test1 *and* Test2 *against the refined* SendSMS *and* DownloadRingtone *sequence diagrams*

specifying our use cases. We could do this because the Escalator tool contains an interpreter that allows us to execute sequence diagrams.

When we want to execute the test cases against the implementation, however, they cannot be applied directly. The test model is on the abstraction level of the specification and hence operates on the abstract interface of the specification. The concrete interface of the implementation, on the other hand, will most likely be more detailed and use different primitives, dependent on the concrete technology used to implement the system. Because of this, we will in most cases need to define an adapter for the tests.

An adapter is a wrapper around the system under test that abstracts from its interface and defines an interface closer to the abstraction level of the tests [4]. The test cases need to be executed, so we also want to define a transformation of the abstract test cases to test scripts that can be executed by a suitable execution engine, e.g. the run time system of the technology used to implement the system.[9] An example of such a transformation from abstract test model to test scripts is found in the UML Testing Profile [17], where a transformation of tests formalized as sequence diagrams to JUnit test scripts is provided. In the context of the case in this article, we would need an adapter and test scripts with the capability of sending HTTP requests to the web portal and to interpret the output from the portal properly.

After defining the adapter and the transformation, we end up with a suite of test scripts that interact with the system under test through the interface of the adapter. These can then be applied to perform well-known testing activities such as unit testing, regression testing, and integration testing.

3 Conclusion

In this article we have demonstrated the use of model-based testing with the Escalator tool in the development of a web portal for mobile phone users. Escalator provides the capability to automatically generate test models from sequence diagram specifications, and also to execute the abstract test cases of test models against sequence diagram specifications – a process we refer to as refinement testing. To the best of our knowledge, Escalator is unique in that it is developed specifically for test generation from, and refinement testing of, sequence diagram specifications. We have shown how this functionality can be applied in the context of a RUP development. We have also explained how a test model generated for this purpose can be reused for testing of an implementation later in the development process.

Through this small case we have demonstrated 1) how model-based testing allows us to benefit from modeling activities by automatically generating test models; 2) how this is not restricted to testing of implementations, but is also applicable for testing of refined specifications; and 3) that model-based testing is not necessarily a 'heavy-weight' method, but with advantage can be utilized also within iterative development methodologies such as RUP.

Acknowledgements
The research on which this article reports has partly been funded by the Norwegian Research Council through the SARDAS project (15295/431) and partly by the European Commission through the MODELPLEX project (Contract no. 034081) under the IST 6th Framework Programme.

[9] *It is of course possible to make an adapter that abstracts all the way to the abstract test model, or to define a transformation of the abstract test cases all the way down to the level of the system under test. But the combined use of adapter and test scripts is the most general approach, and probably the most practical one in many cases.*

References

1 Jusristo, N, Moreno, A M, Vegas, S. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9 (1-2), 7-44, 2004.

2 Tretmans, J. Testing concurrent systems: A formal approach. **In**: *Proc. 10th International Conference on Concurrency Theory (CONCUR'99)*, 46-65. *Lecture Notes in Computer Science*, no. 1664, Springer, 1999.

3 Whittaker, J A. What is software testing? And why is it so hard? *IEEE Software*, 17 (1), 70-79, 2000.

4 Utting, M, Legeard, B. *Practical model-based testing : A tools approach*. Morgan Kaufman, 2007.

5 Baker, P et al. *Model-driven testing : Using the UML Testing Profile*. Springer, 2008.

6 Kruchten, P. *The Rational Unified Process : An introduction*, 2nd ed. Addison-Wesley, 2000.

7 Ambler, S W. *The object primer : Agile model-driven development with UML 2.0*, 3rd ed. Cambridge University Press, 2004.

8 Jeffries, R, Melnik, G. TDD : The art of fearless programming. *IEEE Software*, 24 (3), 24-30, 2007.

9 Rumpe, B. Agile test-based modeling. **In**: *Proc. International Conference on Software Engineering Research and Practice (SERP'06)*, 1, 10-15. CSREA Press, 2006.

10 Lund, M S. *Operational analysis of sequence diagram specifications*. Faculty of Mathematics and Natural Sciences, Univ. of Oslo, 2008. (PhD dissertation) [Online]. Available: http://heim.ifi.uio.no/~massl/publications/phd-thesis.pdf

11 Lund, M S. *The Escalator Tool*. [Online] (2008, May). Available: http://heim.ifi.uio.no/~massl/escalator

12 OMG. *Unified Modeling Language : Superstructure, version 2.1.1 (non-change bar)*. 2005. (OMG Std. formal/2007-02-05)

13 Haugen, Ø, Husa, K E, Runde, R K, Stølen, K. STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling*, 4 (4), 355-367, 2005.

14 Runde, R K. The pragmatics of STAIRS. **In**: *Proc. 5th International Symposium on Formal Methods for Components and Objects (FMCO'06)*, 88-114. *Lecture Notes in Computer Science*, no. 4111. Springer, 2006.

15 ETSI. *Open Service Access (OSA); Parlay X Web Services*. 2006. (ETSI Std. ES 202 391 V1.2.1)

16 Limyr, A. *Graphical editor for UML 2.0 sequence diagrams*. Department of Informatics, Univ. of Oslo, 2005. (Master's thesis)

17 OMG. *UML Testing Profile, version 1.0*. 2005. (OMG Std. formal/2005-07-07)

Mass Soldal Lund received an MSc in 2002 and a PhD in 2008, both from the Department of Informatics, University of Oslo. He is employed as Research Scientist at the Oslo site of SINTEF ICT. He has since 2001 been doing research on various topics within formal methods, formal and semi-formal modeling languages, and model-based testing.

mass.s.lund@sintef.no