Early Fault Detection with Model-Based Testing

Jonas Boberg

Erlang Training and Consulting Ltd.
29 London Fruit & Wool Exchange
Brushfield Street
London, E1 6EU, UK
jonas@erlang-consulting.com

Abstract

Current and future trends for software include increasingly complex requirements on interaction between systems. As a result, the difficulty of system testing increases. Model-based testing is a test technique where test cases are generated from a model of the system. In this study we explore model-based testing on the system-level, starting from early development. We apply model-based testing to a subsystem of a message gateway product in order to improve early fault detection. The results are compared to another subsystem that is tested with hand-crafted test cases.

Based on our experiences, we present a set of challenges and recommendations for system-level, model-based testing. Our results indicate that model-based testing, starting from early development, significantly increases the number of faults detected during system testing.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Verification

Keywords Model-based testing, system testing

1. Introduction

The cost of finding and fixing faults in software typically rises as the development project progresses into a new phase. Faults that are found after the system has been delivered to the customer are many times more expensive to track down and correct than if found during an earlier phase [1]. Current and future trends for software include increasingly complex requirements on interaction between systems [2]. The increased complexity means that a system may have potentially infinite combinations of inputs and resulting outputs. It is difficult to get satisfactory coverage of such a system with hand-crafted manual or automatic test cases [3].

Model-based testing is a test technique where test cases are generated from a model of the system. There are model-based testing tools that can automate the generation of test cases from a behavioral model, including *test oracles* that can determine whether the system under test behaved correctly at the execution of the test case [4]. Test cases generated from a model have been shown to give a high coverage of system interaction points, given that the generation is carefully guided [5].

Intensive research on model-based testing has been conducted, and the feasibility of the approach has been demonstrated. Still, few conducted studies focus on early-fault detection and the application of the technique on specific test levels. Industrial adoption of model-based testing remains low [6]. Although this is partially due to technical limitations, process-related issues remain a large concern. The model-based testing practice must be integrated into current software processes [6]. Limited understanding of the benefits model-based testing delivers at different levels of testing, and the associated challenges of its application in real world projects, is therefore an obstacle to adoption of the technique.

Finding problems faced in industrial software development, and finding solutions that developers will embrace, is an often listed basis for successful technology transfer [7]. We have conducted a pre-study of an ongoing system development project at Erlang Training and Consulting (ETC). The project develops a message gateway product with two subsystems, an E-mail gateway and an Instant Messaging (IM) gateway. The gateway is implemented using the Erlang/OTP platform. Both subsystems essentially interconnect networks that use different communication protocols, by performing the required mapping of protocol messages. The E-mail gateway allows a client to access multiple types of e-mail servers using a single communication protocol. The IM gateway provides mobile devices an interface to multiple instant messaging protocols.

The pre-study indicates that faults which should have been found during system testing of the IM gateway subsystem, have repeatedly been left undetected until customer acceptance testing. Several negative consequences as a result of this fault-slip through have been observed:

- Reproducing and locating the source of the fault requires more effort as the customer anomaly reports often are on a high level.
- The developing organization and customers' confidence in the system passing the acceptance test exit criteria is reduced.
- Additional effort has to be spent on building and deploying new release candidates as faults are found and fixed.

In this study, we show that applying model-based testing to the E-mail gateway system, starting from early development, significantly increased the number of faults found during system testing.

1.1 Purpose

The purpose of this mixed methods study is to better understand how model-based testing can be used as a system-level test technique, starting from early development. This will be done by converging both quantitative and qualitative data. *Faults-slip-through* will be used to measure the relationship between using model-

based testing as a system-level test technique from early development, and the number of faults that should have been detected during system testing but are left undetected until customer acceptance testing.

As described above, the message gateway, developed at ETC (the research site), has two subsystems. In this study, model-based testing will be used for system-level testing of the E-mail gateway subsystem. The fault-slip-through measurements for this subsystem will be compared to the measurements of the baseline subsystem – the IM gateway. The IM gateway is tested with a combination of manual and automated test cases that are hand-crafted without the support of a model. There are many different kinds of testing. This study will focus on black box functionality testing (both positive and negative). At the same time, the managers' and developers' perception of the impact of the model-based testing will be explored using observations and qualitative interviews.

1.2 Approach

Studies of software process improvement suggest that regardless of whether a quality initiative is technical or organizational, the human factor should be considered, because of the potential barriers to change [8]. ETC has not used model-based testing as a system-level test technique before. Model-based testing is not just a matter of generating tests, and executing them to detect defects. It involves several other activities, such as creation of the system model, analyzing the output, reporting defects and generating reports. Another important activity is regression testing, which is often cited as the most important benefit of test automation [9]. The model-based testing practice must therefore be integrated into the project's test process [6]. This integration can meet resistance as existing local practices may directly conflict with the model-based testing technique [10]. In view of this, this study will be conducted in the form of a *software process improvement* initiative.

1.3 Overview

The rest of this paper is organized as follows. Section 2 gives an overview of relevant concepts, summarizes related studies and presents our research questions and hypothesis. Section 3 describes the method used in the study, including how we collected and analysed our data. Section 4 describes the two subsystems under study. Section 5 gives examples of how the system under test was modeled, describes encountered challenges and presents a set of recommendations for model-based testing on the system-level. Section 6 describes the actual results, in terms of changes in fault-slip-through. Section 7 discusses the results and presents issues in the conducted comparisons. Finally, we conclude and suggest questions for further research in section 8.

2. Related Research

This section has four parts. The first part gives an overview of model-based testing. The second part presents an overview of studies that relate directly to this one – model-based testing as a fault-detection practice in industry, and its impact on the development process. The third part presents selected measurements of early fault detection. Finally, the fourth part specifies the research questions and hypothesis of this study.

2.1 Model-based testing

The term model-based testing is commonly used for a wide variety of test generation techniques. In this article, *model-based testing* is a test technique, by which test cases are generated from a *behavior model* of the system under test [10]. Furthermore, we constrain ourselves to the generation of test cases that include a *test oracle* [10], which can assign a pass/fail verdict to each executed test.

Model-based testing typically involves the following steps: [4, 10, 11]

1. Building an abstract model of the behavior of the system under test. The model captures a subset of the system requirements.
2. Definition of test selection criteria. The criteria defines what test cases to generate from the model.
3. Validating the model. This is typically done by sampling abstract test cases from the model and analyzing them. This step is performed to detect major errors in the model that may even hinder generation of test cases.
4. Generating abstract tests from the model, using the defined test selection criteria. At this stage, the generated test cases are expressed in terms of the abstractions used by the model.
5. Transforming (concretize) the abstract test cases into executable test cases.
6. Executing the test cases. At execution time, an adaptor component transforms the output of the system to the abstraction of the model.
7. Assigning of a pass/fail verdict to executed test case.
8. Analyzing the the execution result.

The remainder of this section gives an overview of the variations within the model-based testing practice.

2.1.1 Model types

The behavior of a system can be described using a variety of different model types. Common model-types, such as finite state machines, extended finite state machines, state charts, markov chains and temporal logic are widely described in literature.

2.1.2 Model abstraction

The model must be validated against the system requirements, which may be specified at any level of formality. This implies that the model must be more abstract than the system under test. If it were not, validating the model would require as much effort as validating the system. At the same time, details of the system that are not modeled, cannot be verified using the model [12]. An overview of abstractions that can be applied in the creation of the model is provided in [13] and [12].

2.1.3 Model notation

In principle, all notations with formal semantics can be used as a basis for model-based testing. Examples of commonly used notations are formal specification languages (such as Z), tool vendor specific languages, general-purpose programming languages, the Unified Modeling Language (UML) and domain specific languages. [4]

2.1.4 Concretizing abstract test cases

The approach to concretization of the test cases depends on the nature of the model abstraction. When only the system input data is abstracted by the model, an *adaptor* component (sometimes called *driver* [12]) is typically used. The adaptor adapts the input part of the test case to the format accepted by the system under test [11]. This adaption may also be delayed to test execution time. If the test case is on a higher level of abstraction a template is used to derive a concrete test case. The template adds additional semantics making the test case executable [4].

2.1.5 Online and offline testing

The online testing technique generates the steps of a test case from the model in lock-step with executing them. This generation

technique handles the non-determinism that arises in the testing of reactive and concurrent systems [14]. With off-line test generation, a complete test case is generated before execution. This has other practical advantages, for example the test case can be executed repeatedly (regression testing). It also allows for analyzing and simplification of the test case before it is executed.

2.1.6 Available tools

Generating test cases from high-level specifications is not a recent idea. In 1986 Hayes [15] showed how to systematically derive tests of abstract data structures from a formal specification. At that time, however, the generation and execution of test cases was performed manually. Today, there is a growing number of tools available that automate many of the steps involved in model-based testing. Utting and Legeard [4] provides a comprehensive overview of model-based testing tools.

2.1.7 Quviq QuickCheck

QuickCheck, developed by Quviq AB, is a testing tool for guided random and model-based testing. QuickCheck can simplify a failed test case to a minimal failing test case [16], thereby reducing the problem of deducing the cause of failure for complex test cases. A minimal test case is a test case where every part of the system input is significant in reproducing the failure.

QuickCheck provides a framework for modelling the system under test using an Abstract State Machine. A model is built using Erlang, a general-purpose programming language [17], with support of the library provided with QuickCheck.

Applicability A case study, where Ericsson's Media Proxy was tested using QuickCheck, indicates that the tool can be applied to testing communication protocols. The study also found that QuickCheck potentially reduces the required investment compared to hand-crafted test cases [16]. The system that was tested with QuickCheck in the study had already been pre-release tested by the development team. There are no studies on using QuickCheck earlier in the development.

2.2 Model-based testing in industry

This section outlines and evaluates prior studies on model-based testing as a fault detection practice in industrial projects. Only studies where the test case generation and execution steps were automated are included.

AGEDIS (Automated Generation and Execution of Test Suites in Distributed Component-based Software) was a three-year research project on the automation of software testing funded by the European Union. Five case studies were conducted. These studies focused on applying model-based testing methods and tools to test problems in industrial settings. The studies were conducted at France Telecom, Intrasoft and IBM. The findings were that modeling increased the understanding of the system under test and was found to be an effective way to analyze complex requirements. It was also found that when a requirement changed, adapting the model and regenerating the test cases required less effort compared to updating manually constructed test cases. [18]

Artho et al. [19] presents a case study that applied model-based testing to the controller component of NASAs K9 planetary rover. The modeling language and test framework used was based on a discrete temporal logic. The technique was found promising and located a fault in the controller. The model was developed, and the testing conducted after implementation of the full system was finished.

Dalal et al. [10] reports on obstacles of introducing model-based testing into test organizations. Four case-studies of four large-scale projects are presented. One finding was that model-based tests were sometimes seen as mysterious. This was because the *objectives* of each test case are not as clearly defined as in a typical manually crafted test case. The study suggests that the projects' test process, including test strategies and planning must be adapted, so that the model-based testing is well integrated. Establishing the infrastructure for running and logging the massive amount of generated test cases requires effort. Also, modeling was found to improve the specification and the understanding of the system, which is in line with the findings of the AGEDIS studies [18].

Dalal et al. suggests that defects in the model can be minimized by ensuring traceability from the requirements to parts of the model. This allows fault analysis to faster determine whether the requirements or the implementation is incorrect. The suggestion originates in that more than half of the failed tests related to defects in the model, rather than the system under test. Other studies indicate similar model defect ratios (see Pretschner et al. [12], Blackburn et al. [3]). The following limitations in the case studies can be identified: Only individual components were tested. Test oracles were not generated from the model, but added manually.

Dalal et al. also identifies the following questions for future work:

- What are the challenges of applying model-based testing during different phases of testing?
- What benefits will model-based testing deliver at different phases of testing? [10]

Pretschner et al. [12] investigates whether the model-based testing approach pays off in terms of quality and cost. Quality of model-based test cases are compared to traditional, hand-crafted test cases. The system under test is a network controller for a automotive infotainment system. The findings of the study were that model-based testing did not detect more faults in the implementation than hand-crafted test cases. Also, no correlation between severity of errors and types of tests were found. On the other hand, the model-based tests resulted in the detection of significantly more requirement defects. The study indicates that tests were executed after the system was completely implemented. Therefore, the benefit of test case re-generation were not seen in the study. Also, the length of the test cases was restricted, as a failing test case was inspected manually, and the execution time was long due to hardware limitations. The study acknowledges these deficiencies and points out that the economics of using model-based testing based on behavioral models is not yet understood, in particular whether the life-cycle spanning updating of the model is efficient.

Blackburn et al. [3] discusses the specific skills and practices that are needed to incorporate model-based testing into an organization's test process. The presented material is based on learnings from working with model-based testing in companies and projects during multiple years. The article suggests that an incrementally developed model detects inconsistencies and missing details in requirements early in the development process. Also, objective measurements are pointed out as important to make the effects of the model-based testing visible.

Evaluation Summary Based on evaluation above, it can be concluded that existing research on model-based testing for early-fault detection is lacking. In conducted studies, model-based testing has typically not been an integrated part of the development process. Also, most studies apply model-based testing on the component level, or to a limited part of the system. Few studies focus on the the application of the technique on the system-test level.

2.3 Measuring early fault detection

The central concept underlying this study is that the cost of finding and fixing faults in software rises as the development of the

software progress into a new phase [9]. Reducing the number of defects that are left undetected until customer acceptance testing is a type of improvement work – improvement of the test process. In improvement work, measurements are important in order to know whether you are actually improving. A common test process metric is the number of defects found on different test levels[9]. An important criteria for selecting a performance measure is that it relates closely to the issue under study. As shown by Damm, most measurements cannot acknowledge that not all faults are effectively found on the test level that they were introduced on. Instead, Damm proposes the use of a method called Faults-Slip-Through (FST) [9].

2.3.1 Faults-Slip-Through

When applying the FST method, faults are classified according to the phase in which they *should* have been found. The method has the following steps [9]:

1. Determine which defects should be found on which test level. This is part of the test strategy. It is not already documented by the organization, that has to be done first.
2. For each reported fault, find the test level the defect was found on, and which test level it should have been found on, according to the documented test strategy (see 1)
3. For each test level, summarize the number of faults that should have been found earlier, per test level.

Phase Input Quality Phase Input Quality – the fault-slip-through ratio, can be calculated from the absolute FST data as follows [9]:

$$\text{Phase Input Quality}(PIQ) = \frac{SF}{PF} \cdot 100$$

where *SF* is the number of faults found on test level X that slipped from earlier test levels and *PF* the total number of faults found on test level X.

This formula calculates the percent of faults found at a test level that should have been found earlier. Damm et al. observed a relationship between the number of faults found and the PIQ; if the PIQ for the test level is high, and many faults are found on that test level, this indicates that the test strategy compliance of the earlier test level is low [9]. It is important to analyze the PIQ in the context of the absolute number of found faults. For example, the PIQ of a test level could be high, although only a few faults were found on that test level. Few faults found typically indicates that none or little improvement is needed.

2.3.2 ODC trigger analysis

One way to classify faults is by software triggers. A trigger is a type of stimulus that activates a fault into a failure. A method of such classification is ODC trigger analysis[20]. In ODC trigger analysis, each fault is assigned to an activator category. For example, faults that go into the *Test Sequencing* category are such that require execution of a sequence of functions for the fault to surface. On the other hand, faults that go into the *Interaction* category require the execution of a sequence of functions with varying parameters [20]. ODC trigger analysis can be used to evaluate the test process by identifying what types of faults are found on a specific test level. In combination with the fault-slip-through, it can be used to evaluate what activities in the test process are in need of improvement, and the success of such improvements [9].

2.4 Research Questions and Hypothesis

The central question asked in this study is:

> How can model-based testing be applied at the system-level to enable early fault-detection and increased confidence in the system?

Based on the evaluation of related work in section 2.2, and the challenges identified, the following sub-questions will be addressed by this study:

Q1. How can model-based testing be used to reduce the number of faults that are left undetected until customer acceptance testing?

Q2. What are the challenges of applying model-based testing at the system-level?

The introduction of automated system tests with a high level of interaction coverage should decrease the number of faults that are left undetected until customer acceptance testing. This expectation constitutes the hypothesis for this study:

H1. Applying model-based testing on the system level will decrease the *fault-slip-through* from system testing to customer acceptance testing.

3. Methods

The study was conducted using the action research method [21]. This was motivated by the practice oriented nature of the study, and the author's involvement in both practice and research. Action research is cyclic. Each cycle typically includes planning, acting, observing, and reflecting.[21]

The studied development project used a development process based upon the DSDM framework (The DSDM framework is further discussed in DSDM, Business Focused Development [22]). The length of each time-box was approximately four weeks. The customer conducted an acceptance test on each system release.

The study covers two releases of the messaging gateway. Figure 1 shows the timeline of the study. Due to confidentiality reasons, we cannot state the actual release names. We denote these releases as Release X and Release X+1, respectively. The second release of the E-mail gateway subsystem was developed during two shorter time-boxes, with an interim release, which we call Release X+0.5. No acceptance test was conducted on the interim release. We present the results for this release separately.

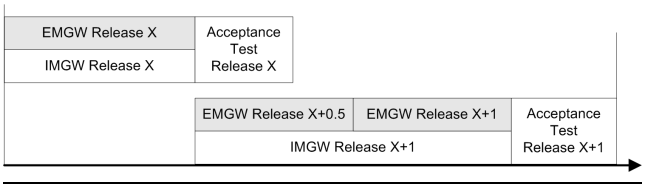

Figure 1. Study timeline

3.1 Research cycles

Each action-research cycle corresponded to a time-box in the E-mail gateway project. The following actions were conducted during each of the three time-boxes:

Planning A set of functional requirements to add to the system model was selected. The selection was based on the given priority of requirements for the current time-box. In addition the developers were asked to prioritize the modeling of the requirements. The latter was done to allow for early testing of features that the developers delivered for system testing (the features due for system testing in the time-box were not all delivered at the same time).

Acting The selected functionality was modeled. A set of abstractions was applied in the process of modeling the system. The abstractions selected were based upon the project's test strategy (what aspects to test on the system level) and trade-offs including difficulty of modeling and ease of validation of the model.

As soon as features were released by the developers for system testing, the model was used for test generation and execution. Detected anomalies were reported in the projects issue tracking system. After analyzing an anomaly, a fault was typically found to be present in either the model or in the system. As faults were corrected, the tests were re-executed.

Observing After completion of the time-box, the results of the testing were observed and analyzed.

Reflecting The developers and test manager were involved in reflecting on the results of the model-based testing and suggesting improvements for the next time-box.

3.2 Site of study

The site of study was Erlang Training and Consulting (ETC). Part of ETC's core business is to develop distributed fault tolerant systems utilizing the Erlang/OTP platform, most of which are network-intensive. Erlang/OTP includes the general purpose programming language Erlang, which has built-in support for concurrency, fault-tolerance, and a set of libraries for application development [17].

3.2.1 Experience of model-based testing

Prior to this study, model-based testing was used by some developers for testing on the unit level. The tool used was QuickCheck (see section 2.1.7). The majority of the developers of ETC had undergone training in use of the tool.

3.3 Researcher's role

The author's role in this study, except from data collection and analysis, was to introduce model-based testing as a technique for system testing in the project. The author also constructed the model of the system, and executed the tests generated from the model. The risk of bias inherent due to the author's involvement and interventions is acknowledged.

3.4 Data collection and analysis

This section presents the applied data collection and analysis procedures. Both quantitative and qualitative data was collected in this study. Found faults were analyzed and the fault-slip-through to acceptance testing measured. The impact of the model-based testing was verified through qualitative interviews with the test manager and developers. In addition, the experiences of system-level modeling, test execution were logged.

3.4.1 Fault Analysis

Faults found during system testing and acceptance testing were measured at the end of each time-box for each of the two subsystems. Data on found faults was collected from the organization issue tracking system. Both the internal organization and the customer reports detected anomalies into this system. The reports include the details of the anomaly, the reporter and the date of the report. This data was sufficient for the FST measurement. Each anomaly report was analyzed according to the following criteria:

1. The anomaly has been confirmed to have been caused by a fault in one of the two subsystems
2. The fault related to a functional aspect of the system

Note that (1) also implies that faults in a shared component of the two subsystems were filtered out. Reports that fulfilled these two criteria were used as input for the fault-slip-through measurement (see section 3.4.2) and classified through ODC fault analysis (see section 3.4.3).

The detected faults for the E-mail gateway system were further analyzed according to whether they were found due to execution of a manually crafted test case, or a test case generated from the model. This distinction was useful to evaluate the model-based testing at the end of each time-box. For example, what faults did the manually crafted test-cases detect, that the ones generated from the model did not? The evaluation was used as input to the improvement of the model for the next time-box.

In case of duplicate anomaly reports for a fault in the system, both reports were used to classify the fault, but only one fault was counted. No faults had an anomaly report from both a manually crafted and a generated test case.

3.4.2 Fault-slip-through

The fault-slip-through to system testing and acceptance testing was measured at the end of each time-box, for each of the two subsystems. The Phase Input Quality for the two test levels was then derived from the fault-slip-through data.

Definition of test levels The FST measurement requires a definition of what defects should be found on which test level (see section 2.3.1). This definition was created by means of open-ended interviews with the test manager, and four developers. In the interview, the subject was first asked to identify the test levels of the test process. For each identified test level, the subject was then asked to describe the type of defects that should be found on that level. At the end of the interview, the subject was presented ten anomaly reports, selected from the issue tracking system. The anomaly reports were randomly selected from both development projects, with the constraint that they were all reported during the last two time-boxes, and that the reported anomaly had been found to be caused by a fault in the system. For each report, the subject was asked to classify on which test level the fault should have been found. This second step was performed to validate the answers in the interview.

Defined test strategy We formalized the test strategy based on analysis of the interview results. The test strategy defined the following test levels: unit testing, integration testing, external integration testing, system testing and acceptance testing. The definition of the test strategy was verified with the interview subjects.

3.4.3 ODC trigger analysis

Faults were classified using ODC trigger analysis. For about 80% of the reported anomalies, the information present in the anomaly report was sufficient for classification. In the rest of the case, the involved project members were consulted. As recommended by Damm et al [9], we iteratively developed the classification scheme during fault analysis. The scheme used is shown in table 1.

Category	Description
Coverage	Execution of a single function
Sequencing	Execution of a sequence of functions
Interaction	Execution of a sequence of functions and multiple parameters interacting with each other
Variation	As Interaction, but including invalid parameters (negative testing)
Fault tolerance	Recovering from faults and fail-over scenarios
Concurrency	Faults that only occur due to concurrent interaction with the system
Configuration	Faults related to specific configurations

Table 1. ODC Trigger Classification Scheme

The E-mail gateway and IM-gateway requires the client to connect and login before any other function can be used. Faults that were triggered by the execution of a single function, after connection and login, were therefore classified to the *Coverage* category. An exception was made for faults where the parameters to the login function affected whether the fault was triggered or not.

3.4.4 Qualitative interviews

We conducted qualitative interviews with the test manager and four developers. The purpose of the interviews was to explore the perceived impact of the model-based system testing. The test manager was involved in the testing of both subsystems, while two of the developers worked primarily on the IM gateway and two on the E-mail gateway.

Time and location The subjects were planned to be interviewed at two instances. Once after the end of the second research cycle, with the intent to get detailed in-process feedback on the improvement initiative. In addition, once before the acceptance test of the last studied system release (Release X+1). Due to time constraints, we conducted only the first set of interviews. The interviews were conducted at the research site.

Interview outline The interviews were conducted using the interview guide approach and had two parts. In the first part, the subjects were asked about their confidence in the testing of the respective subsystem and their confidence that the acceptance test exit criteria would be fulfilled without extending the acceptance test phase due to detected faults. Second, they were asked to elaborate on the factors involved in their level of confidence. In the second part, the subjects were directly asked about their perceived impact of the model-based system testing.

3.4.5 Observations

Observations of team and customer meetings, and e-mail correspondence between the customer and the developing organization provided additional data on the confidence in the system and feedback on the improvement initiative.

To assist in the data collection, a field log was used to record observations. The field log was also used to document experiences on the system modeling, test execution, and reporting of the test results.

3.5 Model-based testing tool

The tool used in this study was Quviq QuickCheck (see section 2.1.7). QuickCheck uses Erlang as a specification language. This means that there was in-house competence in the specification language used to model the system under test. Also, the tool had been used and proven at the site of study (see section 3.2). It was not within the scope of this study to compare different tools for model-based testing. Therefore, the selection of a tool that has seen successful use in the development environment allowed the study to stay within its focus area.

3.6 Verification

The concurrent triangulation strategy[23] was used to verify the findings of the study. The fault-slip-through measurements were compared to the impact of the model-based testing, as perceived by the interview subjects.

4. System under study

The E-mail gateway (EMGW) provides e-mail clients a uniform interface to message store servers that use a variety of access protocols. Supported servers include those that use the Post Office Protocol version 3 (POP3), the Internet Message Access Protocol version 4rev1 (IMAP4rev1) and the Mobile Services Protocol (MSP). The client access a message store, through the gateway, using a subset of the IMAP4rev1 protocol. The gateway also supports the IMAP IDLE extension. The extension enables the client to be notified as messages arrive to a mailbox, without having to poll the server.

The Instant Messaging Gateway (IMGW) uses the Wireless Village Client-Server protocol to provide mobile clients access to multiple instant messaging protocols.

Both of these subsystems were developed using the Erlang/OTP platform. They share large parts of the architecture, and a set of core components, originally developed for the IMGW. Most of the implementation was conducted in Erlang, while some parts were done in C. All development was conducted in a GNU Linux/OpenSuse environment. The project team consisted of 9 persons on full time. Some of the developers worked solely on one of the subsystem, while some were involved in both systems.

4.1 The IMAP4 protocol

The IMAP4rev1 is specified by a Request For Comments (RFC) and is ratified as an *internet standard* by the Internet Engineering Task Force (IETF) [24]. The 108 page long specification defines a set of commands that the client can send to the server, how each command is to be interpreted by the server and the responses that may be returned. The IMAP4rev1 protocol builds on the Multipurpose Internet Mail Extensions (MIME), defined by a range of RFCs [25] which specifies the format of e-mails.

An important part of the protocol is that the connection can be in different states (see figure 2).

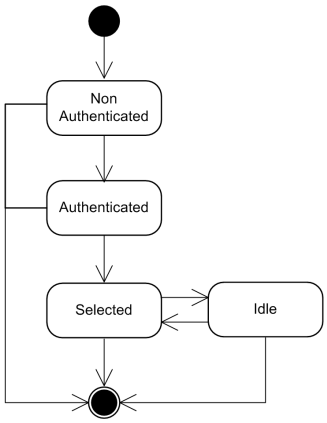

Figure 2. IMAP connection states

Only a subset of the commands are valid in each state. The connection starts in the `Non Authenticated` State. A successful login command results in a transition to the `Authenticated` state. The client can now select a mailbox to work with. In this state, commands that affect the messages in the mailbox can be executed. In addition, the client can issue an *idle* command, causing a transition to the `Idle` state, in which the client is notified about changes to the mailbox. The connection can be terminated from any state, either by the client sending a logout command, or by the connection being closed (due to a client or server error).

Although only a subset of the IMAP4 protocol is implemented by the EMGW, there is still a wide range of variations that the system must be able to handle. The subset of the protocol that the system must handle is defined by the internal ETC system requirement specification document. This document and the relevant RFCs were the main sources of information when developing the model of the system.

4.2 The Wireless Village protocol

The Wireless Village Client-Server protocol is part of a set of specifications for mobile instant messaging. The protocol uses the HTTP1.1 protocol as a bearer and operates over TCP/IP. The Extensible Markup Language (XML) is used to exchange data between the client and server. [26] The core of the protocol is specified by a set of five documents, in total 220 pages, not including the HTTP1.1 and XML specifications.

5. Modeling and Testing Challenges

We constructed a QuickCheck Abstract State Machine (ASM) model of the EMGW system, which was subsequently used to generate and execute test cases. This section gives examples of how the system was modeled, and describes encountered challenges in the modeling and test execution.

5.1 QuickCheck Abstract State Machines

A QuickCheck ASM is specified by a state, a callback module and a set of command generators which are used by the callback module. The ASM is normally used in a QuickCheck property which is then tested. QuickCheck properties are further discussed in Testing Telecoms Software with Quviq QuickCheck [16].

A generated test case is a list of symbolic commands, on the form:

```
{set,{var,1},{call,Module,Function,Arguments}}
```

each of which represents the execution of an *external function*. ASM external functions allow the modeling of non-determinism in the system environment, see Sequential Abstract-State Machines Capture Sequential Algorithms [27] for more information on this topic. During test case generation, QuickCheck represents the results of a symbolic command by a symbolic variable (`{var 1}` in the example above), that can be used as part of subsequent commands. During test case execution the symbolic variables are replaced by the actual value provided by the environment.

QuickCheck implements a subset of the ASM execution theory (described by Gurevich [28]). A limitation is that there is no built in support for separating parts of the model. That is, separating different concerns of the system, in order to avoid a monolithic model that is difficult to validate and maintain. At the same time, it is relatively easy for the modeler to create nested state machines, as the model is specified using a general purpose programming language. This is a technique that was used extensively for the EMGW model.

5.2 The EMGW state

Only a subset of the IMAP commands are valid in each protocol state. The state of each client connection must be modeled to be able to constrain the test case generation to mostly positive test cases. In addition, the state of each mailbox must be modeled, to be able to generate valid commands that affect the messages.

The EMGW state was modeled by an Erlang record:

```
-record(state, {clients=[], accounts}.
```

where `clients` is a list of client connections, and `accounts` a list of e-mail accounts. The following Erlang record models each client connection:

```
-record(client, {connection, user_id, imap_state,
                 selected=undefined, idle=false}).
```

`connection` is a reference to the client connection in the adaptor component, `user_id` an abstraction of the login credentials, that is also used to identify the account. `imap_state` is the current state of the IMAP connection, `selected` is the name of the selected mailbox and `idle` specifies whether the connection is in the `idle` state.

5.3 Adaptor

One of the first tasks was to develop the *adaptor* (see section 2.1.4). The adaptor is an abstract concept. In practice, the adaptor may be split into multiple components. For the EMGW, we used two adaptor components:

- Adaptor for the IMAP protocol
- Adaptor for sending e-mail messages to an account, to be able to test the message related IMAP commands.

We recognized that the IMAP protocol abstractions would have to change as additional system features were incrementally added to the model. With the design principle "encapsulate what varies", the IMAP protocol adaptor was constructed in two parts:

- An IMAP client that enables communication with the EMGW using a functional Erlang interface. The client formats the IMAP commands and parses the server response into an Erlang term representation.
- An interface to the aforementioned client that maps from and to the abstraction of the model

In this way, changes to the model abstractions led to isolated changes in the second part of the adaptor. Also, the IMAP client was not specific to the model abstractions, and could later be reused by the project team for system load testing.

5.4 Generating IMAP commands

IMAP commands were generated in the form of calls to the adaptor. As a running example, this and the following section will use the IMAP *select* command.

```
select_cmd(Client, State) ->
   {call, imap_adaptor, select,
          [Client#client.connection,
           mailbox_name(Client,State)]}.
```

Here, a `select` command is generated, given a client and the current state. A random mailbox name is generated by the `mailbox_name` generator, which is specified as follows:

```
mailbox_name(Client, State) ->
   ?LET(Mailbox,
        oneof(account(Client#client.user_id, State))
          #account.mailboxes),
        Mailbox#mailbox.name).
```

The clients account is looked up from the state, and a random mailbox is picked, whose name is returned by the generator.

5.5 ASM callback functions

A QuickCheck ASM callback module specifies a `precondition`, `next_state` and `postcondition` function. The callback functions for the `select` command were specified as follows.

The `precondition` determines whether to include a symbolic command in a test case, given the current state.

```
precondition(State, {call, _, select, [CPid,_]}) ->
   min_state(client(CPid,State), ?AUTH)
```

Here, `min_state` ensures that the client is at least in the `Authenticated` state for the command to be included in the test sequence.

The `next_state` function updates the state, given the executed command, its result, and the state the command was executed in.

```
next_state(State, _Result,
           {call, _, select,
                    [CPid, MailboxName]}) ->
   Client = client(CPid, State),
   update(Client#client{imap_state=?SELECTED,
                        selected=MailboxName},
          State);
```

The `next_state` function specifies that the client connection transitions to the `Selected` state, and the selected mailbox name is updated when the `select` command is executed.

The `postcondition` evaluates the result of a command, given the state the command was executed in. A test case is assigned a *failed* verdict if a postcondition fails.

```
postcondition(State, {call, _, select,
                      [CPid,_]}, Result) ->
   is_status(Result, ?OK_RESP);
```

Here, `is_status` checks that the server responds to the select command with the `OK` response.

5.6 Challenges and lessons learned

This section presents a set of encountered challenges and recommendations for system-level modeling and test execution.

Develop the model iteratively Iteratively developing the model is seen as crucial. There is a high investment in creating the model. Using it from early development is seen to give higher returns as the partial model can be used to test the system in early development. Early modeling also allowed the modeler to gradually build up the high level of domain knowledge required. In addition, we found that the modeling practice contributed to the understanding of the system requirements, as validation of the model and the analysis of detected faults led to the discovery of unspecified behavior.

Finding abstractions Finding abstractions that allow the model to be more abstract than the system under test can be difficult. We found that this can be remedied by using multiple layers of abstraction, and a combination of multiple model types. We used a Backus-Naur form (BNF) grammar to generate a parser for the IMAP protocol, which was used in the IMAP client part of the adaptor. The parser threw an exception for any malformed system output. In practice this means that part of the verification was performed by the adaptor, but allowed for a simpler ASM model.

Test techniques are complementary Manually crafted test cases can test samples of complex behavior, without having to create a complete model of the behavior. We found that the techniques of model-based testing and manually crafting test cases are complementary. For example, the system fail-over scenarios, that make sure that service is maintained in the event of a failure were seen as overly complex to model and were instead tested using a set of hand-crafted test cases. A useful technique, that can be applied when a system feature is overly complex to model completely, is to only model the feature partially, and limit the command generators to test cases that the model is valid under.

Put effort into the adaptor Developing an adaptor for a system-level model requires considerable effort. On unit and component level, the interface under test is typically less complex. Testing the IMAP protocol required the development of an IMAP client, and applying of multiple layers of abstractions. The total effort involved in developing the adaptor exceeded 40 working hours. On the other hand, we found that a well designed adaptor can simplify the modeling, as described above. Parts of a layered adaptor can also potentially be re-used in other parts of the project.

Model validation by testing The complexity of a system-level model means that the modeling has to start as soon as the requirements for the current time-box have been established. On the other hand, the validation of the model, aside from code review and sampling, cannot start until the system is ready for system testing. We experienced that validation of the model had to continue throughout the full system-test phase. We also found that faults in the system sometimes not only hid other faults in the system, but also faults in the model. As faults in the system were repaired, the next test runs often found faults in the model that were previously undetected. We recommend that a project uses short time-boxes, with small increments in both the implementation and model. This allows for shorter feedback-cycles which eases the model validation.

Reliance on external components System level tests rely on a large number of external components, many of which cannot be be replaced by a dummy (stubbed). For the EMGW system, the test environment included a POP3 and IMAP server which was used through the gateway. During system testing, we found two faults in this server that hindered further testing, and could not be worked around. Tracking down the problem and patching the server took two working days. There is a high risk of failures in external components as the model generated test-cases tests intricate scenarios. We recommend that the suppliers of external components are carefully selected, and that good contacts are maintained with the suppliers, in case of failures.

Execution of partial tests When a failure in the system is detected by the model-based testing, testing cannot be continued since the same failure is likely to occur again. It should be possible to continue testing, in the presence of minor faults in the system. QuickCheck does not provide any support for constraining test case generation or disable parts of the model, in order to allow for this. Instead we had to perform temporary changes to the model. In the absence of tool support, we recommend that such changes are tracked. We used a commenting convention to mark temporarily changed model parts.

6. Results

6.1 Detected faults

The following sections presents the faults found in the studied releases. We present faults detected during system testing and acceptance testing of two releases of the messaging gateway, Release X and Release X+1. We also present the faults detected in the interim release, Release X+0.5 that only included changes in the E-mail gateway.

6.1.1 Release X

Table 2 and 3 shows the fault-slip-through to system testing and acceptance testing for the two subsystems.

Found/Belonging	**System Test**	**Acceptance test**
Unit Test	1 (0)	1
Integration Test	1 (1)	0
External Integration Test	0 (0)	0
System Test	9 (6)	3
Acceptance Test	0 (0)	2
Total found/test level	**11** (7)	**6**

Table 2. FST Release X – Email Gateway

As can be seen in table 2, 11 faults were found in the E-mail gateway during system testing. The numbers in parenthesis show

how many of the faults were detected by the model-based tests. 7 of the 11 faults were detected by the model-based tests. Of the 4 faults found with the hand-crafted test cases, 3 could have been found with the model-based tests (but were found by a hand-crafted test case before that), one had a trigger classified to the category *Fault tolerance*. 6 faults were found during acceptance testing. Of the 6 faults found during acceptance testing, only two should have been found on that test level.

Found/Belonging	**System Test**	**Acceptance test**
Unit Test	6	4
Integration Test	0	0
External Integration Test	0	0
System Test	8	10
Acceptance Test	0	4
Total found/test level	**14**	**18**

Table 3. FST Release X – IM Gateway

Although more faults were found in the IM Gateway during system testing (see table 3), significantly more faults were also found during acceptance testing. Compared to the E-mail gateway, a larger percentage of the faults should have been detected already during unit testing.

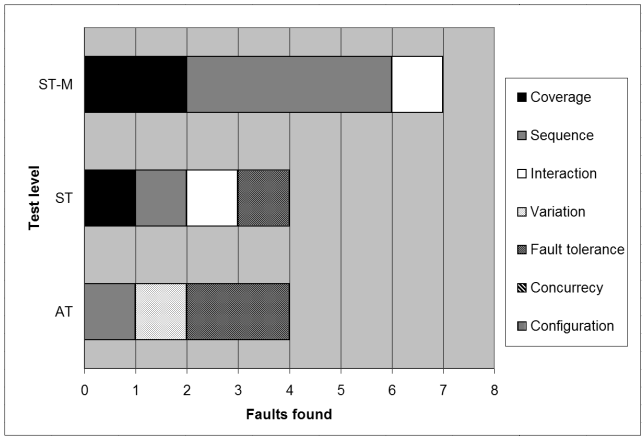

Figure 3. EMGW Release X, ODC trigger distribution by test level

ODC Trigger Analysis Figure 3 shows the ODC trigger distribution for the faults found during system and acceptance testing of Release X of the E-mail gateway. The bar labeled *ST-M* shows the distribution of the faults found by the model-based tests, while the bar labeled *ST* shows the distribution of the faults found by the manually-crafted tests. *AT* shows the distribution of the faults that slipped through to acceptance testing. A majority of the system test triggers are in the sequence category.

As can be seen in figure 4, a large proportion of the IM Gateway fault triggers are in the interaction category, for both test levels. The number of faults with a sequence trigger are about the same as for the E-mail gateway. It is also notable that four of the faults found during acceptance testing have triggers in the coverage category, as each of these faults could have been found earlier by a test case invoking only a single function (with a specific set of parameters).

6.1.2 Release X+0.5

The E-mail gateway had an interim delivery halfway to release X+1. No acceptance testing was conducted on this delivery. Table

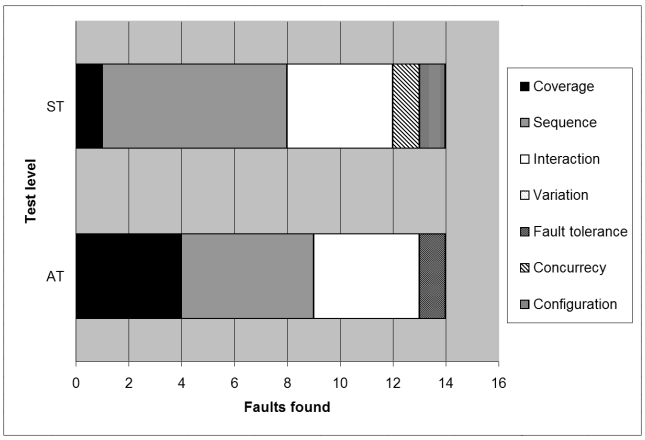

Figure 4. IMGW Release X, ODC trigger distribution by test level

4 shows the number of faults found during system testing. The numbers in parenthesis show how many of the faults were detected by the model-based tests. In total, 17 faults were found, of which 11 were found with the model-based tests. The 6 faults found with the hand-crafted test cases and not by the model-based tests were further analyzed, with the following findings:

- Four of the faults related to the server sending the wrong error code in response to an invalid request. Although the error conditions were part of the system model, the specific error code for invalid requests were not. They were subsequently added to the model.
- Two of the faults were classified to the trigger category *Fault Tolerance*. Such faults were not to be found with the model-based tests, according to the test strategy.

Found/Belonging	**System Test**
Unit Test	4 (4)
Integration Test	0 (0)
External Integration Test	0 (0)
System Test	13 (7)
Acceptance Test	0 (0)
Total found/test level	**17** (11)

Table 4. FST Release X+0.5 – E-mail Gateway

6.1.3 Release X+1

Table 5 and 6 shows the fault-slip-through for the two subsystems.

Found/Belonging	**System Test**	**Acceptance test**
Unit Test	24 (21)	3
Integration Test	1 (1)	0
External Integration Test	0 (0)	0
System Test	39 (26)	4
Acceptance Test	0 (0)	10
Total found/test level	**64** (48)	**17**

Table 5. FST Release X+1 – E-mail Gateway

As can be seen in table 5, 64 faults were detected during system testing of the E-mail Gateway. 48 of these faults were detected by

the model-based testing. The table indicates that 16 of the faults were found by manually crafted test cases. In fact, 9 of these faults were found while manually analysing the system input and output traces, resulting from the model-based tests. It is notable that 24 of the faults should have been detected already during unit testing. Analysis of the anomaly reports show that most of these are related to parsing of e-mail messages. 17 faults were found during acceptance testing of this release. Of these 7 should have been found earlier.

Only 14 faults were detected in the IM Gateway during system testing (see table 6). The same number of faults were detected during acceptance testing. A majority of the faults should have been found earlier.

Found/Belonging	**System Test**	**Acceptance test**
Unit Test	1	4
Integration Test	1	0
External Integration Test	0	1
System Test	12	7
Acceptance Test	0	2
Total found/test level	**14**	**14**

Table 6. FST Release X+1 – IM Gateway

ODC Trigger Analysis Figure 5 shows the ODC trigger distribution for the faults found during system and acceptance testing of Release X+1 of the E-mail gateway. Most of the triggers are in the sequence category, but a significant number of the faults found by the model-based testing are in the coverage category. Most of the faults with a coverage trigger are those that slipped through from unit testing, and are related to parsing. The model-based tests detected six faults with interaction triggers.

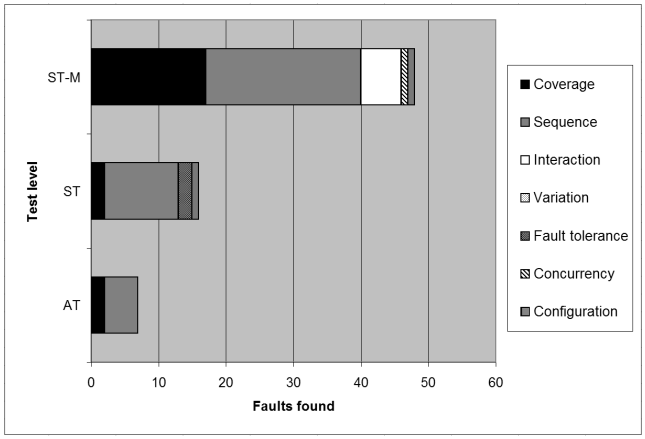

Figure 5. EMGW Release X+1, ODC trigger distribution by test level

Figure 6 shows the ODC trigger distribution for the IM gateway. The trigger distribution is similar to that of Release X of the subsystem. Again, a large proportion of the triggers are in the interaction category.

6.1.4 Summary – Phase Input Quality

By calculating the Phase Input Quality (PIQ) from the FST measurements (as described in section 2.3.1), we can compare the fault-slip-through data of the two subsystems. As relatively few faults were detected for some test levels, the statistical power of the PIQ

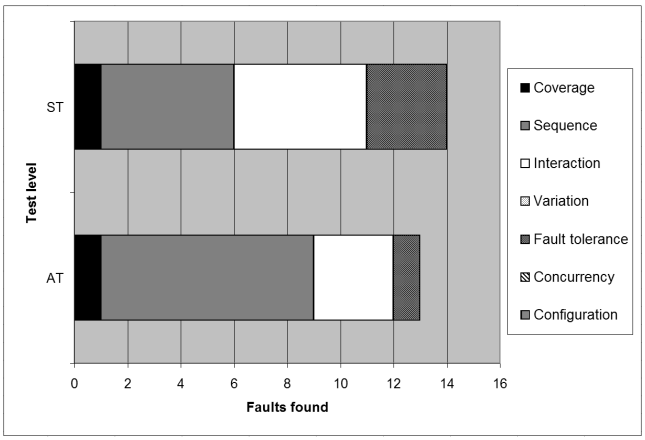

Figure 6. IMGW Release X+1, ODC trigger distribution by test level

is low. The comparisons should therefore only be reviewed in context of the absolute number of faults.

As can be seen in table 6.1.4, the E-mail gateway system test PIQ was 18% for Release X, the first studied release, and increased for subsequent releases. For the IM gateway the system test PIQ has instead decreased from 79% to 14%, although this latter value has a very low statistical power due to the few number of faults found during system testing. Although 17 faults were found during the acceptance testing of release X+1 of the E-mail gateway, a majority of these faults were not slip-throughs. This gives a acceptance test PIQ of 41%. This can be compared to the IM gateway, were a majority of the faults (86%) found during acceptance testing should have been found earlier.

Release/FST	**X**	**X+0.5**	**X+1**
EMGW FST to ST	18%	24%	39 %
EMGW FST to AT	67%	–	41%
IMGW FST to ST	43%	–	14%
IMGW FST to AT	78%	–	86%

Table 7. PIQ - Comparison between releases of subsystems

6.2 Perceived impact and feedback

This section summarizes the results of the first set of interviews, conducted at the end of the second research cycle.

IM gateway The three subjects involved in the development and testing of the IM gateway generally expressed a low confidence in release X of this subsystem. The fact that system testing was not completely finished as the acceptance testing started was stated as a factor in this low confidence level. Two subjects also stated that they perceived the system testing as not sufficiently covering all feature interaction points, with the risk that new features might cause yet undetected side effects in other parts of the system.

E-mail gateway The three subjects involved in development and testing of the E-mail gateway expressed a high level of confidence in release X and release X+0.5 of this subsystem. All of the subjects stated that improved testing on all levels contributed to this level of confidence. The three subjects (who had also experience with the development and testing of the IM gateway) pointed out improved unit testing and the model-based system testing as contributing activities. One of the subjects thought that the results of the model-based testing was clear to internal organization. The two other

subjects thought that only the capability of detecting defects was clear, but that visibility in terms of executed tests and system coverage needed improvement. All three subjects stated that they perceived the results as unclear to the external organization.

7. Discussion

A high number of faults were found in both subsystems during the acceptance test of release X. Over half of these defects should have been found during system testing. We can thus not measure any significant difference in fault-slip-through between the two subsystems for this release. A high number of faults were detected by the model-based tests during system testing of Release X+0.5 and Release X+1. Of the faults detected in the E-mail gateway during the acceptance testing of release X+1, most could not have been detected on previous test levels. There is a significant difference compared to the IM gateway, where most of the faults detected during acceptance testing should have been found earlier. The finding that our improvement initiative did not see effect until Release X+1 is in line with prior studies that suggest that successful implementation of improvements requires multiple iterations [29].

7.1 Complexity of protocol

The difference in complexity of the two subsystems external protocols are likely to have influenced the ODC trigger analysis results. In release X, the number of detected faults with a interaction trigger were significantly higher for the IM gateway, while few faults with the interaction trigger have been found in the E-mail gateway overall. The IM Gateway uses the Wireless village protocol (see section 4.2), which we perceive as being more complex than the IMAP protocol (see section 4.1), in terms of the number of interacting request and response parameters.

7.2 Stability of IM gateway

Significantly less faults have been found during the system test of the IM gateway in release X+1. Differences in the types of features added to the two subsystems is likely to have influenced these figures. Multiple new components were developed for the E-mail gateway for this release. The new IM gateway features on the other hand, were mostly implemented by the extension of existing components that have been thoroughly tested in previous releases. Notwithstanding the higher stability of the IM gateway, more faults in this subsystem slipped through to acceptance testing, compared to the E-mail gateway.

7.3 Fault-slip-through to system testing

We can see that the fault-slip-thorough to system testing was significantly lower in the E-mail gateway, compared to the IM-gateway, in release X. We attribute this to improvements in unit testing, attributed to another improvement initiative (see section 7.4, Internal validity). In subsequent releases of the E-mail gateway, more faults have slipped through to system testing. One plausible reason for this is that as the system testing improves, the developers perform less unit testing. This has been indicated by two of the developers, who during development of release X+1 stated that they would like to start system testing early, due to the model-based tests' potential of finding defects.

7.4 Validity Threats

When conducting an empirical study in industry, the subject cannot be controlled as in a laboratory research experiment. The following validity threats may therefore be of concern in this study.

Internal validity A possible threat to the internal validity of this study is another process improvement initiative that was executed at the research site during the duration of this study. The objective of the other initiative is to increase the quality of the releases delivered to the customer. It might therefore be of concern as to whether any observed changes in fault-slip-through can be attributed to this model-based testing initiative. This threat is mitigated by two facts. First, this other initiative is project wide (it affects both subsystems), secondly the fact that the model-based test cases have found a majority of the faults in the E-mail gateway.

Another validity threat is whether other factors than the model-based testing have had an impact on an eventual reduction of fault-slip-through to acceptance testing. The interviews conducted with the test manager and developers increase the validity, as they show that the internal organization is in agreement about the impact of the model-based testing.

External validity In this study, the results are overall not fully generalizable since they are dependent on the studied project using certain processes and tools. Nevertheless, the results should be generalizable within similar contexts.

A threat to the generalizability of the results of this study is the fact that the tested system is developed in Erlang, which is also the specification language used with the model-based testing tool. Concerns that the results are not generalizable to projects where the implementation and specification languages differ may therefore be raised. However, the interfaces of the system, that the system tests interacted with, are internet standard protocols, layered over TCP/IP. While Erlang mechanisms were taken advantage of to issue test specific commands (for example to restart components to ensure a consistent state at the start of each test case), the effort to implement these test specific interfaces in another environment is seen as negligible, compared to the total effort required for the model-based testing.

8. Conclusions and future work

This study set out to contribute to the understanding of system-level model-based testing as a test technique for early fault detection. Our experiences of modeling and test execution are generally in line with those reported by prior studies. We contribute further to the understanding of system-level model-based testing by presenting a set of challenges and recommendations specific to this test level.

A substantial initial investment is required to integrate the model-based testing into the test process. As the management and customers are highly dependent on measurable results and progress reports from system testing, introducing model-based tests on this level requires considerable planning and effort.

The test results for the two subsystems shows that, for the subsystem tested with the model-based tests, significantly less faults slipped through from system testing to customer acceptance testing. This supports our hypothesis H1, that model-based testing would decrease our fault-slip-through from system testing to customer acceptance testing.

During the study, we observed that the customers' confidence in the system is dependent on the availability of test reports. The customer (external organization) requires these reports to see that the system has been sufficiently tested. As identified by prior studies, it is also important to make the results of the model-based testing visible within the internal organization, to get commitment to the technique. We identify the following question for further research:

- How can reports on the coverage and results of model-based tests be presented to the internal and external organizations?

Acknowledgments

Part of this work has been carried out during my time at the IT University of Göteborg. The work has been sponsored by Erlang Training and Consulting Ltd and Quviq AB. I would like to thank my supervisor Thomas Arts for his help and support during the project.

References

[1] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–147, January 2001.

[2] Barry Boehm. Some future trends and implications for systems and software engineering processes. *Systems Engineering*, 9(1):1–19, 2006.

[3] Mark Blackburn, Robert Busser, and Aaron Nauman. Why model-based test automation is different and what you should know to get started. In *International Conference on Practical Software Quality*. Software Productivity Consortium, NFP, 2004.

[4] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[5] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *The Journal of Software Testing, Verification and Reliability*, 13(1):25–53, March 1997.

[6] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Adrian M. Colyer. From research to reward: Challenges in technology transfer. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 569–576, New York, NY, USA, 2000. ACM.

[8] Mahmood Niazi, David Wilson, and Didar Zowghi. A maturity model for the implementation of software process improvement: an empirical study. *Journal of Systems and Software*, 74(2):155–172, 2005.

[9] Lars-Ola Damm. *Early and Cost-Effective Software Fault Detection - Measurement and Implementation in an Industrial Setting*. PhD thesis, Blekinge Institute of Technology, Department of Systems and Software Engineering, 2007.

[10] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[11] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Working Papers 2006. Department of Computer Science, The University of Waikato (New Zealand), April 2006.

[12] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401, New York, NY, USA, 2005. ACM.

[13] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. In *Electronic Notes in Theoretical Computer Science. Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, volume 116, pages 59–71, Los Alamitos, CA, USA, January 2005. Elsevier Science Publishers Ltd.

[14] Colin Campbell, Margus Veanes, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer, technical report MSR-TR-2005-59. Microsoft Research, May 2005.

[15] I J Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124–133, 1986.

[16] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with Quviq QuickCheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM.

[17] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[18] Ian Craggs, Manolis Sardis, and Thierry Heuillard. Agedis case studies: Model-based testing in industry. In *1st European Conference on Model Driven Software Engineering*. AGEDIS, December 2003.

[19] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, 2005.

[20] Ram Chillarege and Kathryn A. Bassin. Software triggers as a function of time - odc on field faults. *DCCA-5: Fifth IFIP Working Conference on Dependable Computing for Critical Applications*, September 1995.

[21] Wesley Vernon. An introductory guide to putting action research into practice. *PodiatryNow*, February 2007.

[22] Jeniffer Stapleton. *DSDM, Business Focused Development, Second Edition*. Pearson Education, 2003.

[23] John W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications Inc., 2003.

[24] Network Working Group. Request for comments 3501 - internet message access protocol - version 4rev1. The Internet Engineering Task Force, http://www.ietf.org/rfc/rfc3501.txt, March 2003.

[25] Network Working Group. Request for comments 2045 - multipurpose internet mail extensions (mime) part one: Format of internet message bodies. The Internet Engineering Task Force, http://www.ietf.org/rfc/rfc2045.txt, November 1996.

[26] Wireless Village. Wv client-server protocol v1.1. Open Mobile Alliance Ltd, 2002.

[27] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.

[28] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and validation methods*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[29] Anna Borjesson and Lars Mathiassen. Successful process implementation. *IEEE Software*, 21(4):36–44, 2004.