Experiences with Formal Engineering: Model-Based Specification, Implementation and Testing of a Software Bus at Neopost.[☆]

M. Sijtema[b], A. Belinfante[a,**], M.I.A. Stoelinga[a,*], L. Marinelli[c]

[a]*Faculty of Computer Science, University of Twente, The Netherlands*
[b]*Sytematic Software, The Hague, The Netherlands*
[c]*Neopost, Austin, Texas, USA*

Abstract

We report on the actual industrial use of formal methods during the development of a software bus. During an internship at Neopost Inc., of 14 weeks, we developed the server component of a software bus, called the *XBus*, using formal methods during the design, validation and testing phase: we modeled our design of the XBus in the process algebra mCRL2, validated the design using the mCRL2-simulator, and fully automatically tested our implementation with the model-based test tool JTorX. This resulted in a well-tested software bus with a maintainable architecture. Writing the model ($\mathsf{m}_{\mathsf{dev}}$), simulating it, and testing the implementation with JTorX only took 17% of the total development time. Moreover, the errors found with model-based testing would have been hard to find with conventional test methods. Thus, we show that formal engineering can be feasible, beneficial and cost-effective.

The findings above, reported earlier in [1], were well-received, also in industrially oriented conferences [2, 3]. In this paper, we look back on the case study, and carefully analyze its merits and shortcomings. We reflect on (1) the added benefits of model checking, (2) model completeness and (3) the quality and performance of the test process.

Thus, in a second phase, after the internship, we model checked the XBus protocol—this was not done in [1] since the Neopost business process required a working implementation after 14 weeks. We used the CADP tool evaluator4 to check the behavioral requirements obtained during the development. Model checking did not uncover errors in model $\mathsf{m}_{\mathsf{dev}}$, but revealed that model $\mathsf{m}_{\mathsf{dev}}$ was not complete and optimized: in particular, requirements to so-called bad weather behavior (exceptions, unexpected inputs, etc.) were missing. Therefore, we created several improved models, checked that we could validate them, and used them to analyze quality and performance of the test process. Model

[☆]This research has been partially funded by NWO Dn 63-257 (ROCKS), by STW 12238 (ArRangeer), by EU under grants PNO 318490 (SENSATION) and 318003 (TREsPASS).

[*]Corresponding author

[**]Principal corresponding author

Email addresses: marten@sytematic.nl (M. Sijtema), axel.belinfante@gmail.com (A. Belinfante), M.I.A.Stoelinga@utwente.nl (M.I.A. Stoelinga), l.marinelli@neopost.com (L. Marinelli)

checking was expensive: it took us approx. 4 weeks in total, compared to 3 weeks for the entire model-based testing approach during the internship.

In the second phase, we analyzed the quality and performance of the test process, where we looked at both code and model coverage. We found that high code coverage (almost 100%) is in most cases obtained within 1000 test steps and 2 minutes, which matches the fact that the faults in the XBus were discovered within a few minutes.

Summarizing, we firmly believe that the formal engineering approach is cost-effective, and produces high quality software products. Model checking does yield significantly better models, but is also costly. Thus, system developers should trade off higher model quality against higher costs.

Keywords: formal methods, formal engineering, model-based testing, IOCO, JTorX, mCRL2, LTSmin, CADP, evaluator4, MCL, LPS, lps2torx

1. Introduction

Formal engineering, that is, the use of formal methods during the design, implementation and testing of software systems is gaining momentum. Various large companies use formal methods as a part of their development cycle; and several papers report on the use of formal methods during ad hoc projects [4, 5].

Formal methods include a rich palette of mathematically rigorous modeling, analysis and testing techniques, including formal specification, model checking, theorem proving, extended static checking, run-time verification, and model-based testing. The central claim made by the field of formal methods is that, while it requires an initial investment to develop rigorous models and perform rigorous analysis methods, these pay off in the long run in terms of better, and more maintainable code. While experiences with formal engineering have been a success in large and safety-critical projects [6, 7, 4, 8, 9], we investigate this claim for a more modest and non-safety-critical project, namely the development of a software bus.

The experiences that we report on were obtained during two phases: a first phase in which we developed the XBus at Neopost Inc., and a second, post-case study analysis phase, where we performed model checking of the XBus protocol, and measured the quality and performance of the model-based testing process.

1.1. First phase: Developing the XBus

The XBus. Neopost Inc. is one of the largest companies in the world producing supplies and services for the mailing and shipping industry, like franking and mail inserting machines, and the XBus is a software bus that supports communication between mailing devices and software clients. The XBus allows clients to send XML-formatted messages to each other over TCP (the X in XBus stands for XML), and also implements a service-discovery mechanism. That is, clients can advertise their provided services and query and subscribe to services provided by others.

We have developed the XBus using the classical V-model [10], see Fig. 2 on page 9, and used formal methods during the design and testing phase. The total running time of this project—an internship carried out in the summer of

2009 by the first author of this paper, at that time a Computer Science MSc. student—was 14 weeks.

An important step in the design phase was the creation of a behavioral model $\mathsf{m}_{\mathsf{dev}}$ of the XBus, written in the process algebra mCRL2 [11, 12]. We chose mCRL2 because of its powerful data types and function declarations, which turned out to be very helpful for our purpose. Model $\mathsf{m}_{\mathsf{dev}}$ pins down the interaction between the XBus and its environment in a mathematically precise way. We simulated the model to check its validity, which greatly increased our understanding of the XBus protocol, and made the implementation phase a lot easier. Due to time-constraints we did not use model-checking during XBus development.

Testing the XBus. After implementing the protocol, we tested the implementation, i_1, distinguishing between data and protocol behavior. *Data behavior* concerns the input/output behavior of a function and is static, i.e., independent of the order of the methods calls. *Protocol behavior* relates to the business logic of the system, i.e. the interaction between the XBus and its clients. Here, the order in which protocol messages occur crucially determines the correctness of the protocol. Therefore, we used unit testing to test the data behavior and model-based testing for the protocol behavior.

Model-based testing with JTorX. We used JTorX to test the implementation against mCRL2 model $\mathsf{m}_{\mathsf{dev}}$. JTorX [13, 14] is a model-based testing tool capable of automatic test generation, execution and evaluation. During the design phase, we already catered for model-based testing: we designed for testability by taking care that at the model boundaries, we could observe meaningful messages. Moreover, we made sure that the boundaries in the mCRL2 model matched the boundaries in the architecture. Also, the use of model-based testing technology required us to write an adapter. This is a piece of software that translates the protocol messages from the mCRL2 model into physical messages in the implementation. Again, our design for testability greatly facilitated the development of the adapter.

After unit testing and repairing the issues uncovered by it, we ran JTorX against implementation i_1 and mCRL2 model $\mathsf{m}_{\mathsf{dev}}$ (once configured, JTorX runs completely automatically) and found five subtle bugs. We believe that it is much harder to discover these bugs with unit testing, because they involve the order in which protocol messages should occur. After repairing them, we ran JTorX several times for more than 24 hours, without finding any more errors. After an acceptance test, the XBus was released for use in Neopost.

1.2. Second phase: Analysis

The development of the XBus, carried out in the first phase, supported the central claim made by Formal Methods—the use of rigorous methods during the development is cost effective. Nevertheless, it leaves room for questions: how thorough was the process carried out in 14 weeks? How good was the model—this is important, since the model-based test process is as good as the model. Would model checking have helped to produce better code? Was the testing thorough enough; what can we say about coverage? In this paper, we investigate these questions. In particular, we focus on (1) the added benefits of model checking, (2) the quality of the models, and (3) test coverage.

Ad 1: The added benefits of model checking. We started out by model checking the model $\mathsf{m}_{\mathsf{dev}}$ that was created during the development phase. We created a series of new models with different features. Firstly, we needed to change $\mathsf{m}_{\mathsf{dev}}$ to make it finite, yielding the model $\mathsf{m}_{\mathsf{dev,fin}}$: $\mathsf{m}_{\mathsf{dev}}$ allows an unbounded number of client connections, and uses arbitrary integers as connection identifiers—this is not a problem for (online) testing, but for model checking it is, as it leads to an infinite state space.

We used evaluator4 [15] from the CADP toolset to model check the XBus requirements obtained during the first phase; these requirements were formalized in the logic MCL [16], which is an extension of the μ-calculus with data. The main reason that we chose evaluator4 is that it allows reasoning over the individual parameters of the messages (labels). We used the mCRL2 and LTSmin toolsets to obtain, from the mCRL2 model, the binary coded graph (`.bcg`) file that evaluator4 needs. Model checking did not uncover errors in model $\mathsf{m}_{\mathsf{dev,fin}}$. With hindsight, this is not so surprising, because model $\mathsf{m}_{\mathsf{dev}}$ had been used extensively already in simulation and model-based testing, and because of the simple structure of the model, which we describe in Section 4.1 on page 15. We did find $\mathsf{m}_{\mathsf{dev,fin}}$ (and hence $\mathsf{m}_{\mathsf{dev}}$) to be not complete. In particular, those requirements related to so-called bad weather behavior are not present: invalid or unexpected messages were not modeled, and neither were empty lists of services.

Compared to testing, the model checking process was very labor intensive: i.e. reworking the models, formalizing requirements, playing round with tricks to reduce the state space and making sure that the time needed for model checking was manageable. This took us 4 person weeks. The entire model-based testing approach, i.e. writing the model $\mathsf{m}_{\mathsf{dev}}$, creating the adapter, executing and analyzing the tests, took 3 person weeks.

Ad 2: Model quality. We included the additional requirements that we uncovered during our model checking activities and created a family of models, all in mCRL2, as shown in Table 4 on page 18 and depicted in Fig. 6 on page 17. For each model, we constructed an infinite variant (for testing) and a finite one (for model-checking). Also, we investigated the use of queues: model $\mathsf{m}_{\mathsf{dev}}$ is simple in the sense that it does neither model the incoming message queue, nor the fifo-queue-like behavior of the TCP connections between the XBus server and its clients. We show that including these queues in the model does not affect test coverage, but does significantly increase testing time. Finally, we constructed several model variants that were more liberal wrt the accepted inputs. An overview of this family of models is given in Section 4.2 on page 16.

Ad 3: Test coverage. Code coverage metrics [17] are standard measures to evaluate the quality of a test suite: the higher the coverage, the more faults a test suite can potentially find. We extensively evaluated the thoroughness of our testing, by measuring code coverage and model coverage. Since the original XBus implementation is proprietary software of Neopost, we had no longer access to it after the internship. Therefore, we used a carefully reconstructed implementation. We used branch coverage as our code coverage metric, i.e. the percentage of all branches in the control flow graph that were executed during testing. To do so, we instrumented the code of the (reconstructed) implementation by hand. For model coverage, we used the percentage of *linear process specification* (LPS) summands executed. LPSs are a uniformized representation of mCRL2 models.

Complete LPS coverage basically means that each nondeterministic alternative is executed at least once.

We have extensively analyzed model and code coverage from short test runs (10,000 test steps, 5–30 minutes) and long ones (250,000 test steps, 2–40+ hours), with each of our (infinite) model versions. We found that the maximal code coverage is typically already reached after 1000 test steps, i.e. after at most two minutes of testing.

We found that, the more complete a model was (wrt requirements or accepted inputs), the higher code coverage could be obtained. Note that all the tests were derived fully automatically by doing a random walk over (the state space of) the model.

1.3. Our findings

In the first phase, during the internship, writing the model, simulating it, and testing the implementation with JTorX only took 17% of the total development time. Therefore, we conclude that the formal engineering approach has been very successful: with limited overhead, we have created a reliable software bus with a maintainable architecture. Thus, as in [18], we clearly show that formal engineering is not only beneficial for large, complex and/or safety-critical systems, but also for more modest projects.

In the second phase, during the analysis, after the internship, we found that, within the limits of the model, the model-based testing that was done during the project was rather thorough. However, we also found that the model, used to derive these tests, was not fully complete, and more thorough analysis of the requirements, during the project would have been desirable. This could have been achieved with model checking, but at a high cost. We expect that more light weight methods that trace the requirements in the model are more cost effective.

Finally, we experienced that model and code coverage metrics can provide valuable insight in the quality, effectiveness, and progress of the model-based testing process. Based on our experiences, we advocate that formal engineering pays off, and that investing in high-quality models is worth-while: the quality of model-driven development lies within the quality of the model. To do so, we believe that models should be as complete as possible, i.e. accept all inputs and include all requirements. Extensive simulation and—though expensive—model-checking help. Also, we believe that measuring coverage is helpful: if less than 100% code coverage is achieved, then the model should be augmented.

About this paper. An earlier version of this paper was published as [1]. As elaborated above, the contributions of the current paper over [1] involves the entire second phase, in particular: (1) model checking of the XBus protocol via a series of new models, (2) testing against these new models, and (3) extensive test coverage measurements.

Remainder of this paper. The remainder of this paper is organized as follows. Section 2 provides the context of the XBus implementation project. Then, Section 3 describes the activities involved in each phase of the development of the XBus, including the activities done during the analysis after the project. Section 4 gives the details of modeling, creation of additional models, and model checking, and Section 5 gives the details of model-based testing, and of code coverage and model coverage analysis. Section 6 reflects on the lessons learned in this project. Finally, Section 7 presents conclusions.

2. Background

2.1. The XBus and its context

Neopost. Neopost Incorporated [19] is one of the world's main manufacturers of equipment and supplies for the mailing industry. Neopost produces both physical machines, like franking and mail inserting machines, as well as software to control these machines. Neopost is a multinational company headquartered in Paris (France) that has departments all over the world. Its software division, called Neopost Software & Integrated Solutions (NSIS) is located in Austin, Texas, USA. This is where the XBus implementation project took place.

Shipping and franking mail. Typically, the workflow of shipping and franking is as follows. To send a batch of mail, one first puts the mail into a folding machine, which folds all letters. Then an inserting machine inserts all letters into envelopes[1] and finally, the mail goes into a franking machine, which puts appropriate postage on the envelopes and keeps track of the expenses.

Thus, to ship a batch of mail, one has to set up this process, selecting which folding, inserting and franking machine to use and configure each of these machines, setting the mail's size, weight, priority, and the carrier to use. These configurations can be set manually, using the machine's built-in displays and buttons. More convenient, however, is to configure the mailing process via one of the desktop applications that Neopost provides.

The XBus. To connect a desktop application to the various machines, a software bus, called the XBus, has been developed. The XBus communicates over TCP and allows clients to discover other clients, announce provided services, query for services provided by other clients and subscribe to services. Also, XBus clients can send self-defined messages across the bus.

When this project started, an older version of the XBus existed, called the XBus version 1.0. Goal of our project was to re-implement the XBus while maintaining backward compatibility, i.e. the XBus 2.0 must support XBus 1.0 clients. Key requirements for the new XBus were improved maintainability and testability.

2.2. Model-based testing

The concept of model-based testing. Model-based testing (MBT, a.k.a. model-driven testing) is an innovative testing methodology that provides methods for automatic test generation, execution and evaluation from a formal model m. Model m, usually a transition system, of the system-under-test (SUT, a.k.a. implementation-under-test), pins down the desired system behavior in an unambiguous way: traces of m are correct system behaviors, and traces not in m are incorrect.

[1]Alternatively, a combined folding/inserting machine can be used

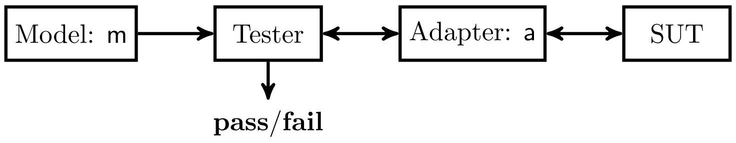


Figure 1: Model-based testing.

The concept of model-based testing is visualized in Fig. 1. Tests are derived from a model `m` and applied to the SUT. Based on observations made during test executions, a verdict (**pass** or **fail**) about the correctness of the SUT is given.

Each test case consists of a number of test steps. Each test step either applies a stimulus (i.e. an input to the SUT), or obtains an observation (i.e. a response from the SUT). In the latter case, we check whether the response was expected, that is, if it was predicted by the model `m`. In case of an unexpected observation, the test case ends with verdict **fail**. Otherwise, the test case may either continue with a next test step, or it may end with a verdict **pass**.

Test execution requires an adapter `a`. Its role is to translate actions in the model `m` to concrete commands—in our case to TCP messages—of the SUT. Writing an adapter can be tricky, for instance if one action in the model corresponds to multiple actions in the system. Therefore, keeping the adapter simple, was an important design decision for us. We achieved this by keeping a close correspondence between `m` and the system architecture.

Key advantage of MBT techniques is that, given a model `m` and an adapter `a`, model-based testing is fully automatic: MBT tools can fully automatically derive test cases from the model, execute them, and issue verdicts. There are various MBT tools around, like SpecExplorer from Microsoft [20], Conformiq Qtronic [21], AGEDIS [22], and—used here—JTorX. Each of these tools varies in the capabilities, modeling languages and underlying theories, see [23, 24] for an overview.

JTorX. JTorX is an MBT tool developed at the University of Twente. It implements automatic test case generation, execution and evaluation from models in a number of formalisms. JTorX has built-in support for models in graphml [25], the Aldebaran (`.aut`) and Jararaca [26] file formats, and for STS-es [27] in XML (`.sax`). Moreover, it is able to access models on-the-fly (on demand) via interfaces offered by mCRL2 [11], LTSmin [28] and CADP [29]. This allows JTorX to deal with infinite models, as long as they are finitely branching.

JTorX improves over its predecessor TorX [30, 31], one of the first model-based testing tools in the field. JTorX is based on a newer, and more practical, version of the *ioco*-theory[2]and much easier to install, configure and use. Moreover, it has built-in adapter functionality to connect the model to the SUT via TCP/IP. All this turned out to be very helpful in this case study.

[2]TorX was based on the *ioco* theory of [32] in which test cases are non input-enabled. JTorX is based on both the refined *ioco* theory of [33] in which test cases are input-enabled, and the *uioco* theory [34], a weaker relation than *ioco* developed for models that contain underspecified traces.

JTorX uses so-called online (aka on-the-fly) test case generation and execution. This means that the test derivation and test execution functionalities are tightly coupled: test cases and test steps are derived on demand (only when required) during test execution. This is why Fig. 1 does not show test cases. JTorX can be used in 3 modes: (1) fully automatic, i.e. random, (2) manual, i.e. via interactive user guidance, and (3) guided via test purposes. We only used the first mode here.

Correctness of tests. MBT provides a rigorous underpinning of the test process: it can be shown that, under the assumption that the model correctly reflects the desired system behavior, all test cases derived from the model are correct: they yield the correct verdict when executed against any implementation, see e.g. [35]. More technically, the test case derivation methods underlying JTorX have been shown *sound* and *complete*. That is, any correct implementation of a model `m` will pass all tests derived from `m` (soundness). Moreover, for any incorrect implementation of `m`, there is at least one test case derivable from `m` that exhibits the error (completeness). Note that completeness is merely an important theoretical property, showing that the test case derivation method has no inherent blind spots. In practice, only a finite number of test cases are executed, and therefore, the test case exhibiting the error may or may not be among the executed ones. As stated by Dijkstra's famous quote: "testing can only show the presence of errors, not their absence".

Rich and well-developed MBT theories exist for control-dominated applications, and have been extended to test real-time properties [36, 37, 38], data-intensive systems [27], object-oriented systems [39], and systems with measure imprecisions [40].

2.3. The specification language mCRL2

The language mCRL2 [11, 12] is a formal modeling language for describing concurrent systems, developed at the Eindhoven University of Technology. It is based on the process algebra ACP [41], and extends ACP with rich data types and higher-order functions. The mCRL2 toolset facilitates simulation, analysis and visualization of behavior; model-based testing against mCRL2 models is supported by the model-based test tool JTorX. Specifications in mCRL2 start with a definition of the required data types. Technically, the behavior of the system is declared via process equations of the form $X(x_1 : D_1, x_2 : D_2, \ldots, x_n : D_n) = t$, where x_i is a variable of type D_i and t is a process term, see the example in Section 3.2. Process terms are built from (1) (potentially parameterized) actions; (2) operators: alternative composition, sum, sequential composition, conditional choice (if-then-else), parallel composition; and (3) encapsulation, renaming, and abstraction. Actions represent basic events (like sending a message or printing a file) which are used for synchronization between parallel processes. Apart from analysis within the tool set, mCRL2 interoperates with other tools: specifications in mCRL2 can be model checked via the CADP model checker by generating the state space in `.aut` or `.bcg` format, they can be proven correct using e.g. the theorem prover PVS, and they can be tested against with JTorX. For model checking, we used the evaluator4 tool from the CADP tool set. The tool evaluator4 is able to check whether a model-checking formula, given in its input language MCL, holds for (the state space generated from) an mCRL2 model `m`.

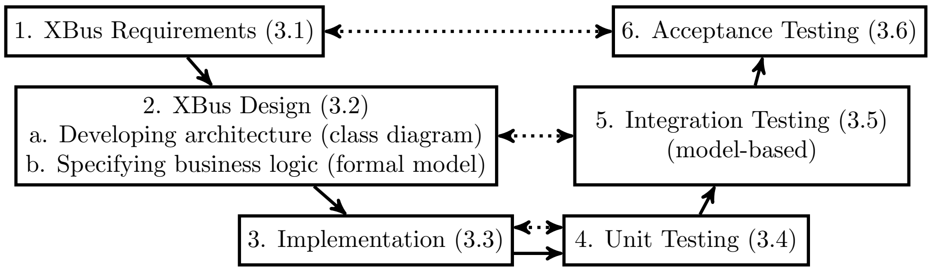

1. XBus Requirements (3.1)
6. Acceptance Testing (3.6)
2. XBus Design (3.2)
a. Developing architecture (class diagram)
b. Specifying business logic (formal model)
5. Integration Testing (3.5) (model-based)
3. Implementation (3.3)
4. Unit Testing (3.4)

Figure 2: The V-model used for development of XBus; parenthesized numbers refer to sections.

3. Development of the XBus and post case-study analysis

Below, we describe all the activities in the development (phase 1; during the internship) and analysis (phase 2; after the internship) of the XBus. We developed the XBus according to the classical V-model ([10], see Fig. 2). For each step in the V-model we report the activities carried out in both phases—each section below corresponds to one step in the V-model, see Fig. 2 again.

During the development, the overall test strategy was to test data behavior using unit testing, and to test protocol behavior, i.e. the interaction between XBus and its clients, using model-based testing. We chose to use model-based testing for protocol behavior, because here the dynamic behavior, i.e., the order of protocol messages, crucially determines the correctness of the protocol. We only started with model-based testing of protocol behavior after we had completed unit testing of data behavior.

3.1. XBus requirements

First phase. We obtained the functional and nonfunctional requirements by studying the documentation of the XBus version 1.0 (a four page English text document) and by interviewing the manager of the XBus development team.

The functional requirements express that the XBus is a centralized software application which can be regarded as a network router: clients can connect and disconnect at any point in time; connected clients can send XML-formatted messages to each other. Moreover, clients can discover other clients, announce services, and query for services that are provided by other clients. Also, they can subscribe to services, and send self-defined messages to each other. Table 3 on page 11 gives an overview of the XBus protocol messages. Table 1 on the following page summarizes the functional requirements; important non-functional requirements are testability, maintainability and backwards compatibility with the XBus 1.0.

Second phase. While formalizing the requirements in Table 1 and model checking them on model $\mathsf{m}_{\mathsf{dev}}$, we realized that so-called bad weather behavior was not present in $\mathsf{m}_{\mathsf{dev}}$ nor in the requirements. Therefore, we extended both the model, and the list of requirements. Table 2 on the next page shows the additional requirements, pinning down what to do with unexpected inputs.

3.2. XBus design

First phase. In the first phase, the design step encompassed two activities: we created

1.	XBus messages are formatted in XML, following the same Schema as the XBus 1.0.
2.	Clients connecting to XBus perform a handshake with the XBus server. The handshake consists of a $\mathsf{Conn_{req}}$—$\mathsf{Conn_{ack}}$—$\mathsf{Conn_{auth}}$ sequence.
3.	Newly connected clients are assigned unique identifiers.
4.	Clients can subscribe to be notified when a client connects or disconnects.
5.	Clients can send messages to other clients with self-defined, custom, data. Such messages can have a self-defined, custom message type. In addition there are protocol messages for connecting, service subscription, service advertisement.
6.	Clients can subscribe to receive all messages, sent by other clients, that are of one or more given types (including self-defined messages), using the Sub message.
7.	Clients can announce services that they provide, using the $\mathsf{Serv_{ann}}$ message.
8.	Clients can inquire about services, by specifying a list of service names in a $\mathsf{Serv_{inq}}$ message. Service providers that provide a subset of the inquired services will respond to this client with the $\mathsf{Serv_{rsp}}$ message.
9.	Clients can send *private* messages, which are only delivered to a specified destination.
10.	Clients can send *local* messages, which are delivered to the specified address, as well as to clients subscribed to the specified message type.

Table 1: Overview of XBus requirements obtained in the first phase.

11.	Invalid messages are discarded.
12.	Unexpected messages are discarded.
13.	$\mathsf{Serv_{ann}}$, $\mathsf{Serv_{inq}}$ and $\mathsf{Serv_{rsp}}$ messages with an empty list of services are not broadcasted to other clients.
14.	$\mathsf{Notif_{local}}$ messages are not broadcasted to source or destination of a $\mathsf{Local_{req}}$ message.

Table 2: Overview of additional XBus requirements obtained in the second phase.

(A) an architectural design, given by the UML class diagram in Fig. 3 on the following page, and

(B) an mCRL2 model, $\mathsf{m_{dev}}$, describing the protocol behavior.

We used the mCRL2 simulator to validate the design and model $\mathsf{m_{dev}}$. As said, time constraints prevented us to use model-checking in this phase.

The architectural design and mCRL2 model $\mathsf{m_{dev}}$ were developed in parallel. Central in their design are the XBus messages: each message translates into a method in the class diagram and into an action in mCRL2 model $\mathsf{m_{dev}}$. The UML diagram specifies which methods are provided, while the mCRL2 model $\mathsf{m_{dev}}$ describes the order in which actions should occur, i.e. the order in which methods should be invoked. Thus, the architectural model in UML and the behavioral model in mCRL2 are tightly coupled and complementary.

Ad A: Architectural design. The architecture of the XBus is given in Fig. 3 on the next page, and is based on a standard client-server architecture. Thus, the XBus has a client side, implemented by the XBusGenericClient, and a server side, implemented by the XBusManager. The latter handles incoming protocol messages and sends the required responses. Both the server and the client use the Communications package, which implements communication over TCP. As illustrated in Fig. 4 on page 12, the ConnectionManager class in the Communications

Connection establishment and release		
$\mathsf{Conn}_{\mathsf{req}}$	*input*	(implicit) implied by a client establishing a TCP connection with XBus.
$\mathsf{Conn}_{\mathsf{ack}}$	*output*	sent from XBus to a client just after the client establishes a TCP connection with the XBus, as part of the handshake.
$\mathsf{Conn}_{\mathsf{auth}}$	*input*	sent from a client to the XBus to complete the handshake.
$\mathsf{Disc}_{\mathsf{req}}$	*input*	(implicit) implied by a client closing its TCP connection with XBus.
Service announcement and inquiry		
$\mathsf{Serv}_{\mathsf{ann}}$	*input*	sent (just after connecting) from a client c to XBus, which broadcasts it to all other connected clients, to announce the services provided by c.
$\mathsf{Serv}_{\mathsf{inq}}$	*input*	sent (just after connecting) from client to XBus, which broadcasts it to all other connected clients, to ask what services they provide.
$\mathsf{Serv}_{\mathsf{rsp}}$	*output*	sent from a client via XBus to another client, as response to $\mathsf{Serv}_{\mathsf{inq}}$, to tell the inquirer what services the responding client provides.
Event subscription and notification		
(Un)Sub	*input*	sent from a client to XBus, with as parameter a list of (custom) message types, to (un)subscribe receipt of all messages of the given types.
$\mathsf{Notif}_{\mathsf{conn}}$	*output*	sent from XBus to clients that subscribed connect notifications.
$\mathsf{Notif}_{\mathsf{disc}}$	*output*	sent from XBus to clients that subscribed disconnect notifications.
$\mathsf{Notif}_{\mathsf{local}}$	*output*	sent from XBus to clients that subscribed to non-private messages.
Messages to other clients		
$\mathsf{Local}_{\mathsf{req}}$	*input*	sent from client to XBus, to be delivered to indicated client (as $\mathsf{Local}_{\mathsf{ind}}$), and to other clients that have subscribed the given message type (as $\mathsf{Notif}_{\mathsf{local}}$).
$\mathsf{Local}_{\mathsf{ind}}$	*output*	sent from XBus to clients, as consequence of a received $\mathsf{Local}_{\mathsf{req}}$.
$\mathsf{Priv}_{\mathsf{req}}$	*input*	sent from client to XBus, to be delivered to indicated client only.
$\mathsf{Priv}_{\mathsf{ind}}$	*output*	sent from XBus to clients, as consequence of a received $\mathsf{Priv}_{\mathsf{req}}$.

Table 3: Overview of XBus protocol messages.

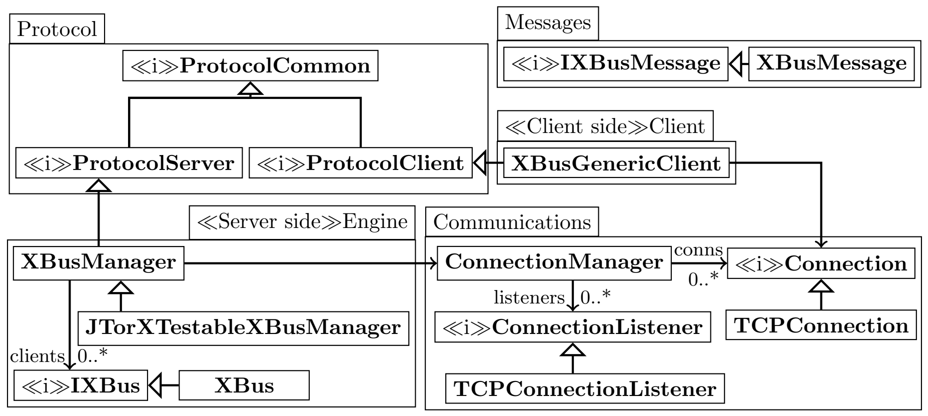


Figure 3: High level architecture of the XBus system. It contains a server side package, and a client side package. Furthermore, it has functionality for TCP connections and XBus messages. Both server and client implement the Protocol abstract class. All interfaces are indicated with ≪i≫.

package uses a queue data structure as a buffer for incoming messages. When a message is handed over from the Communications package to its user—in the server this user is the XBusManager—it is popped from the queue.

We catered for model-based testing already in the design: class XBusManager

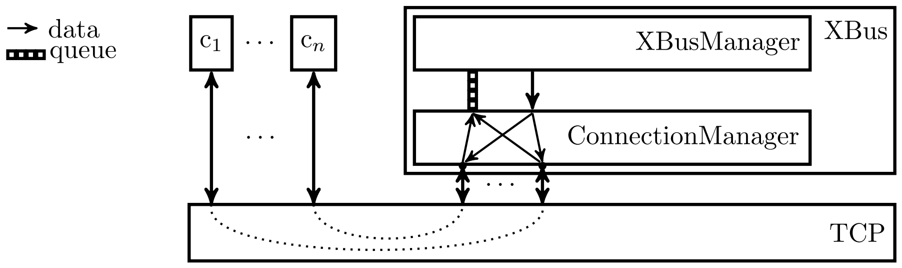


Figure 4: Communication between clients $c_1, \ldots, c_n$ and XBus. We show how The XBus is decomposed into XBusManager and ConnectionManager. Incoming messages are collected in a queue in the ConnectionManager—here drawn as the left connection between XBusManager and ConnectionManager. The XBusManager processes these messages one by one; it sends responses using methods offered by the Communications package—this is represented by the arrow from XBusManager to ConnectionManager.

has a subclass JTorXTestableXBusManager. During testing, this subclass overrides the send message of class XBusManager, allowing JTorX to have more control over the state of the XBus server; see Section 3.5 for more details.

Ad B: The mCRL2 model. We modeled the required XBus behavior as an mCRL2 process; we chose mCRL2 because of its powerful datatypes and function declarations that can be defined in a functional programming style. We profited from mCRL2's concise notation for enumerated types, records, and lists, and the ability to define functions.

A key decision in creating a model is what to model, and to determine the abstraction level and model boundaries. We chose to model the XBusManager, i.e. the handling of the messages that come into the server; this is the most critical part of the XBus functionality. Thus, the Communications package is not included in model $\mathsf{m}_{\mathsf{dev}}$, and neither are the internal components like the TCP-sockets, nor the queue that the Communications package uses as a buffer for incoming messages. Thus, for each message that arrives at the server, $\mathsf{m}_{\mathsf{dev}}$ models how to handle this message: it will either send a reply, relay, or broadcast the message. Then, $\mathsf{m}_{\mathsf{dev}}$ will update its internal state: in order to determine the correct response, the server keeps track of the client's state by keeping an internal list of client objects.

In Section 4.1 we discuss the model in more detail.

Second phase. In the second phase, we evaluated the quality of model $\mathsf{m}_{\mathsf{dev}}$, where we looked at both completeness and correctness—using model checking—of the model. When we found that model $\mathsf{m}_{\mathsf{dev}}$ was incomplete—i.e., not all requirements are represented in it—we created additional models, and used model-checking to check their correctness. We elaborate on these activities, and on the additional models, in Section 4.2.

3.3. Implementation

First phase. In the first phase, we created implementation i_1, at Neopost, for use by Neopost. Implementation i_1 was only created once we had sufficient confidence in the quality of the design—to a large extent due to modeling and simulation. The programming language used was C#—use of .NET is Neopost

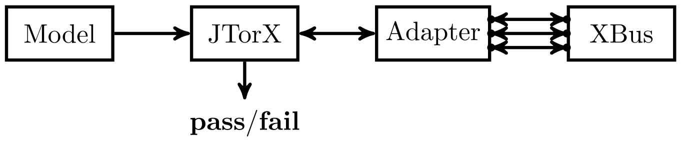


Figure 5: Testing XBus with JTorX playing the role of 3 clients.

company policy. Together with XBus server i_1, also an XBus client library was implemented, to ease construction of XBus clients. As we will see in Section 3.5, this client library was also used during model-based testing of XBus server i_1.

Second phase. Since implementation i_1 is proprietary software of Neopost, it was not available during the second phase. Therefore, we carefully created a second implementation, i_2, to allow analysis of the thoroughness of model-based testing. Implementation i_2 was written in the programming language Go [42]. Implementation i_2 has the same functionality as i_1, except that i_2 uses labels from the model, rather than XML formatted messages.

3.4. Unit testing

First phase. In the first phase, implementation i_1 was tested using unit tests as described below. Because the overall test strategy was to test data behavior using unit testing, and to test protocol behavior using model-based testing, the classes in the Communications and Messages packages were tested using unit testing. For the Communications package, unit tests were written to test the ability to start a TCP listener and to connect to a TCP listener, to test the administration of connections, and to test transfer of data. For the Messages package, unit tests were written to test construction, parsing and validation of messages. The latter was tested using both correct and incorrect messages.

Each error that was found during unit testing was immediately repaired.

Second phase. In the second phase, no unit tests were run—implementation i_2 was only tested using model-based testing.

3.5. Model-based integration testing

For both implementations, we used model-based testing for the business logic, i.e. to test the interaction between XBus and its clients. In the first phase, for implementation i_1, we did this after we had completed unit testing of data behavior.

General test set up. To test whether the XBus implementations interact correctly with their environment, we first have to decide on a test set up. In both phases, we used the same test set up with three XBus clients, see Fig. 5 (although, as we show, the chosen test architecture differed). Three XBus clients is the smallest number that allows testing interesting scenarios that involve multiple clients. Thus, JTorX plays the role of three XBus clients, which are able to perform all protocol actions described in Section 3.1.

Typically, such scenarios require one client to trigger the activity—for example by connecting or disconnecting, or by sending a $\mathsf{Serv}_{\mathsf{inq}}$ message. A second client is necessary to cooperate in the activity, i.e. to witness or to realize the effect—by receiving a message and, possibly, responding to it. The third client can be used either to show the effect on a client that does not cooperate in the activity, or to show that the XBus is correct when multiple clients do cooperate in the activity. Typically, a single test run contains (many) instances of either of these roles for the third client.

It is our experience that model-based testing can easily generate long test runs, in which, at least for models that are as small as the one in this project, each possible scenario that can take place, does take place, multiple times. We come back to this in the discussion of (code- and model) coverage, in Section 5.

Dealing with potential message reordering. As mentioned above, what we modeled is the XBusManager. However, the XBusManager implementation that we want to test is just one component of the XBus server. So, as we have seen more often when applying model-based testing (see e.g. Section 4 of [30]), we could not connect the test tool directly to the implementation that we wanted to test (the XBusManager), at interfaces that coincide with the model boundaries. An obvious way to test the XBusManager, is via the XBus server in which it is contained, and interact with the XBus server via TCP connections—one for each XBus client impersonated by JTorX that has a connection to the XBus. However, messages that are sent at approximately the same moment, in the same direction, over different TCP connections between the XBus and its clients (whether impersonated or not), may overtake each other.

For *stimuli*, JTorX is in control: it can, if necessary, reduce the rate at which stimuli are sent to the point that, when the next stimulus is sent, the previous one will already have been received by the XBus. In both phases we just assumed that, compared to the network, JTorX is slow, such that the pace at which JTorX sends stimuli is slow enough to avoid one stimulus overtaking another one.

We used different solutions to deal with the possibility that *responses* would overtake each other—if we would not have dealt with this possibility, the tester might have emitted a **fail** verdict to a sequence of responses whose order was scrambled by the TCP channel. In the first phase, we extended the XBus implementation with an additional interface that provided JTorX access to the responses in the order in which the XBusManager produced them. In the second phase, instead, we relaxed the model, to not only accept the responses to a single stimulus in the single order in which they were produced by the XBusManager, but also accept any possible reordering. We discuss details in Section 5.

First phase. After unit testing had been completed, and all errors that were found had been repaired, we tested implementation i_1 against model $\mathsf{m}_{\mathsf{dev}}$ using JTorX, to find errors. We found 5 bugs. Typically, a bug was found within 5 minutes after the start of a test. All these bugs concern the order in which protocol messages must occur. Therefore, it is our firm belief that they are much harder to discover with unit testing. After these bugs had been repaired, we ran JTorX several times for more than 24 hours, without finding any more errors. In Section 5.1 we discuss the details of the test architecture that we used, and of the bugs that we found.

Second phase. In the second phase, we tested implementation i_2 against all (infinite) models that we created in that phase, not to find errors, but to investigate the thoroughness of the testing process, by looking at code coverage and model coverage. We did not test i_2 against model $\mathsf{m}_{\mathsf{dev}}$, because the different solution (for responses that might overtake each other) led to a different test architecture than in the first phase, one that was not consistent with $\mathsf{m}_{\mathsf{dev}}$. Among the models that we did test i_2 with, though, is model $\mathsf{m}^{\mathsf{order}}_{\mathsf{dev}}$: this model was derived from $\mathsf{m}_{\mathsf{dev}}$, by extending it to cater for the slightly different test architecture. We ran tests of 10,000 steps, and of 250,000 steps, fully automatically.

Regarding code coverage, we found that with all models the maximal coverage was already reached in the test runs of 10,000 steps. With model $\mathsf{m}^{\mathsf{order}}_{\mathsf{dev}}$ we obtained 79% code coverage. This is no surprise: we know that $\mathsf{m}^{\mathsf{order}}_{\mathsf{dev}}$ does not contain all possible messages, and thus certain stimuli can not be generated from it. With the most complete model, $\mathsf{m}^{\mathsf{req,ie}}_{\mathsf{opt}}$, we obtained 100% code coverage. The coverage obtained with the other models was between these two numbers. Regarding model coverage, we saw that with each model we reached the maximal coverage possible, in the runs of 250,000 steps. However, we needed many more test steps to reach maximal model coverage, than to reach maximal code coverage.

In Section 5.2 we discuss the details of the test architecture that we used, and the test that we ran; in Sections 5.3–5.5 we discuss in more detail the coverage that we obtained; and in Section 5.6 we discuss the test execution time.

3.6. Acceptance testing

First phase. Acceptance testing was done in the usual way: we organized a session with the manager of Neopost's ISS group, and showed how the XBus 2.0 implementation worked. In particular, we demonstrated that it implements the features required in Section 3.1.

Second phase. In the second phase no acceptance testing was performed.

4. Modeling & Model Checking of the XBus

This section zooms in on the modeling and model checking activities described in Section 3.2.

4.1. The model $\mathsf{m}_{\mathsf{dev}}$

As mentioned, the mCRL2 model $\mathsf{m}_{\mathsf{dev}}$ describes the desired functioning of the XBusManager package, which is responsible for the handling of XBus messages and therefore the most central part of the XBus. Internally $\mathsf{m}_{\mathsf{dev}}$ keeps track of the state of all connected clients. Based on this state $\mathsf{m}_{\mathsf{dev}}$ decides, when a message arrives, how to handle it: send a reply or broadcast, relay it, or simply ignore it. After handling the message, $\mathsf{m}_{\mathsf{dev}}$ updates its internal state.

Data. Model $\mathsf{m}_{\mathsf{dev}}$ stores its internal data in a single data object: a list of clients, modeled as a list of data structures. For each client, the following information is kept.

- an integer that represents the identity of the client;
- the connection status of the client, being either: disconnected, awaiting-Authentication, or connected;

```
1 proc listening(c:Clients) =
2    (sum j:Int.(j >= 0 && j < numClients(c) &&
3                getClientStatus(j, c) == DISCONNECTED )
4       -> (ConnectRequest.ConnectAcknowledge.
5           listening(changeClientStatus(j, c, AWAIT_AUTH)))
6       <> delta
7    ) + ...
```

Listing 1: Definition of XBus handling of $\mathsf{Conn_{req}}$ message in mCRL2.

- the subscriptions of the client, which is a list of message types.
- the services that the client provides, which is a list of integers.

Behavior. Model $\mathsf{m_{dev}}$ consists of a single process that operates in the following loop: (1) accept a message, (2) send zero or more responses, (3) update the internal state, i.e., the client list. After these steps, $\mathsf{m_{dev}}$ is ready to process the next message. For example, when $\mathsf{m_{dev}}$ receives a $\mathsf{Conn_{req}}$, it replies with a $\mathsf{Conn_{ack}}$, and adds the new client to the client list.

Listing 1 shows a (slightly simplified) part of $\mathsf{m_{dev}}$. The process is named listening, and has as single parameter the list of clients c. The listing shows that from each client j that currently is in disconnected state (line 3), the server is willing to accept a $\mathsf{Conn_{req}}$ message, after which it will send out a $\mathsf{Conn_{ack}}$ message (line 4). Then it will update the status of the j^{th} client in the list and continue processing via a recursive call (line 5).

Model size. The entire model consists of 6 pages (180 lines, 12kB) of mCRL2, including comments (without comments and blank lines: 142 lines, 9kB). Approximately half of it concerns the specification of data types and functions over them; the other half is the behavioral specification.

Model validation. During the construction of the model, we exhaustively used the simulator from the mCRL2 toolkit. We incrementally simulated smaller and larger models, using both manual and random simulation. This was done for two reasons. First, to get a better understanding of the working of the whole system, and to validate the design already before the implementation activity was started. This was particularly useful to improve our understanding of the XBus protocol, of which only a (non-formal) English text description was available, which contained several ambiguities. Second, to validate the model, to be sure that it faithfully represents the design, i.e. to fulfill the assumptions stated in Section 2.2, such that when we use JTorX to test our implementation against the model, all tests that JTorX derives from the model will yield the correct verdict.

4.2. Model checking & model transformation

Model completeness. During the analysis, we carefully studied the requirements, and tried to formalize and model check them on model $\mathsf{m_{dev}}$. We found that model $\mathsf{m_{dev}}$ is incomplete, in the sense that not all requirements are represented in it. Model $\mathsf{m_{dev}}$ does not contain self-defined messages, (i.e. no private or local messages) and thus requirements 5 and 6 can only be checked partially, and requirements 9 and 10 can not be checked. Also, $\mathsf{m_{dev}}$ does not model

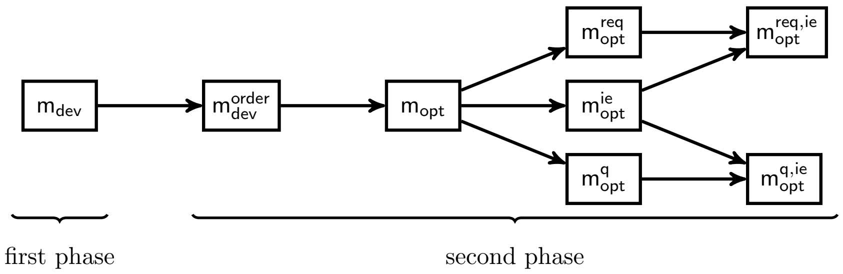

$\mathsf{m}_{\mathsf{dev}}$
$\mathsf{m}^{\mathsf{order}}_{\mathsf{dev}}$
$\mathsf{m}_{\mathsf{opt}}$
$\mathsf{m}^{\mathsf{req}}_{\mathsf{opt}}$
$\mathsf{m}^{\mathsf{ie}}_{\mathsf{opt}}$
$\mathsf{m}^{\mathsf{q}}_{\mathsf{opt}}$
$\mathsf{m}^{\mathsf{req,ie}}_{\mathsf{opt}}$
$\mathsf{m}^{\mathsf{q,ie}}_{\mathsf{opt}}$
first phase
second phase

Figure 6: Relation between the XBus models discussed in this paper.

lists of services, but only uses a singleton list with exactly one service, and thus requirements 7 and 8 can only be checked partially. Finally, $\mathsf{m}_{\mathsf{dev}}$ does not consider the formatting of the messages, and thus requirement 1 can not be represented in it.

During this analysis we also found that the list of requirements was incomplete: it only dealt with good-weather behavior; bad-weather behavior was left unspecified. Thus, requirements 11–14 were added during this analysis.

Family of models. In order to incorporate the missing behavior, we created a family of mCRL2 models, see Figure 6 and Table 4 on the next page, that incorporate all requirements except for requirement 1: we do still not consider the XML formatting of the messages. We added the following two sets of features: (1) self-defined, private and local messages, and non-empty lists of services (instead of a singleton list with one element), (2) empty lists of services, invalid messages, and other bad weather behavior. We did this in several steps, to control the amount of change introduced by each step, and to be able to observe (and show) the impact of the change on state space size (see Table 5 on page 19) and testing speed and coverage (see Sections 5.3–5.6).

We started with $\mathsf{m}^{\mathsf{order}}_{\mathsf{dev}}$, which is the same as $\mathsf{m}_{\mathsf{dev}}$ except that it caters for the test architecture from the second phase. We optimized this model to $\mathsf{m}_{\mathsf{opt}}$, in which each state has a unique representation. From $\mathsf{m}_{\mathsf{opt}}$, we investigated three different variants: $\mathsf{m}^{\mathsf{req}}_{\mathsf{opt}}$ extends the requirements with the first set of features mentioned above; $\mathsf{m}^{\mathsf{ie}}_{\mathsf{opt}}$ is an (semi) input-enabled variant of $\mathsf{m}_{\mathsf{opt}}$, i.e. when it is ready to accept input, it accepts any input; finally, $\mathsf{m}^{\mathsf{q}}_{\mathsf{opt}}$ is obtained from $\mathsf{m}_{\mathsf{opt}}$ by adding a message queue. We combined $\mathsf{m}^{\mathsf{req}}_{\mathsf{opt}}$ and $\mathsf{m}^{\mathsf{ie}}_{\mathsf{opt}}$ into $\mathsf{m}^{\mathsf{req,ie}}_{\mathsf{opt}}$. Similarly, we combined $\mathsf{m}^{\mathsf{ie}}_{\mathsf{opt}}$ and $\mathsf{m}^{\mathsf{q}}_{\mathsf{opt}}$ into $\mathsf{m}^{\mathsf{q,ie}}_{\mathsf{opt}}$. As explained below, each model comes in two variants: an infinite one for testing, and a finite one—indicated via a subscript fin—for model checking.

Finite state space variants. The original model $\mathsf{m}_{\mathsf{dev}}$ was infinite: it could accept an unbounded number of clients, where it used an unbounded integer as connection identifier in the protocol messages. For on-the-fly testing this is not a problem, because we only generate the portion of the state space that the system is currently in. For model checking, however, we need the complete state space, and therefore models have to be finite. We achieved this by restricting the number of times that the server accepts a new client connection to a finite number, namely three. With three connections we can trigger the majority of the interesting scenarios and verify the requirements. Still, the number is low

$\mathsf{m}_{\mathsf{dev}}$	the original model, created in the first phase, during development of the XBus. This model alternates between accepting an input and producing the corresponding outputs, and it is not input-enabled: for example, after a client has sent a Sub message for a certain event e, a subsequent Sub message for e is only accepted after an Unsub message for e.
$\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$	obtained from $\mathsf{m}_{\mathsf{dev}}$, with one very small change, to accommodate the slightly different test architecture that we used in the second phase, as discussed in Section 3.5: it allows all possible interleavings of the outputs produced for a single input.
$\mathsf{m}_{\mathsf{opt}}$	an optimized version of $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, in which each state has a unique representation (finite state space variant of $\mathsf{m}_{\mathsf{opt}}$ is branching bisimilar to finite state space variant of $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$).
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$	obtained from $\mathsf{m}_{\mathsf{opt}}$, (semi) input-enabled: when the system accepts input, all (known) inputs are allowed.
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$	obtained from $\mathsf{m}_{\mathsf{opt}}$, by extending it such that all requirements (except requirement 1) are represented (but without input enabling, and: no invalid messages, and no messages with an empty list).
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$	derived from $\mathsf{m}_{\mathsf{opt}}$, by extending it such that all requirements (except requirement 1) are represented, and making it (semi) input-enabled.
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$	obtained from $\mathsf{m}_{\mathsf{opt}}$, by adding a queue context (but without input enabling, and without extending it to represent additional requirements).
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$	obtained from $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$, by making it (semi) input-enabled.

Table 4: Overview of our XBus models.

enough to allow state space generation. Table 5 shows the sizes of the state spaces of these model variants.

Model optimization. To make model checking feasible, we needed to optimize the model. Thus, we produced the optimized model, $\mathsf{m}_{\mathsf{opt}}$, because in the original model, $\mathsf{m}_{\mathsf{dev}}$, a single event—a client connecting to the server—could result (non-deterministically) in multiple different configurations of the client administration data structures. This badly affected state space generation: it took several hours, whereas with model $\mathsf{m}_{\mathsf{opt,fin}}$ it took in the order of minutes. As discussed in Section 5.6, the speed of testing with JTorX is influenced in a similar way. The optimized model $\mathsf{m}_{\mathsf{opt}}$ is almost fully deterministic, which greatly reduces the work of JTorX' on-the-fly determinization algorithm.

Model correctness. We checked Requirements 2, 3, 4, 7 and 8 on model $\mathsf{m}_{\mathsf{opt}}$ and on model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, using the evaluator4 tool of the CADP tool set. We found that these requirements are all satisfied by the model.

To investigate feasibility of checking the other requirements on model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, we also checked requirements 9 and 10 on it, and checked requirement 7 with messages that contain a list of two services. Listing 2 shows (some) of the properties that we used to check requirement 2.

For those requirements that we checked, we typically formulated and checked multiple formulas, to verify a single requirement. For example, for requirement 9 we not only tried to verify that the intended destination receives the private message that is sent to it, but also that a client c only receives a private message, when there was client that sent that message with c as destination.

model	*#states*	*#transitions*	*#labels*
$\mathsf{m}_{\mathsf{dev,fin}}$	4,198,090	21,476,661	71
reduced (strong bisimulation)	686,151	1,236,486	71
reduced (branching)	362,958	867,666	71
$\mathsf{m}^{\mathsf{order}}_{\mathsf{dev,fin}}$	8,129,310	30,217,652	71
reduced (strong bisimulation)	198,095	425,112	71
reduced (branching)	83,414	194,271	71
$\mathsf{m}_{\mathsf{opt,fin}}$	133,857	1,019,196	71
reduced (strong/branching bisim.)	83,414	194,271	71
$\mathsf{m}^{\mathsf{ie}}_{\mathsf{opt,fin}}$	133,864	1,699,188	71
reduced (strong/branching bisim.)	16,620	69,550	71
$\mathsf{m}^{\mathsf{req}}_{\mathsf{opt,fin}}$	41,264,499	743,604,503	230
reduced (strong bisimulation)	29,586,657	58,206,010	230
reduced (branching bisimulation)	21,643,798	50,263,151	230
$\mathsf{m}^{\mathsf{req,ie}}_{\mathsf{opt,fin}}$	44,643,962	1,538,273,570	245
reduced (strong bisimulation)	4,371,205	12,321,042	245
reduced (branching bisimulation)	3,135,659	11,085,496	245
$\mathsf{m}^{\mathsf{q}}_{\mathsf{opt,fin}}$	> 467,940,404	> 2,937,662,934	?
$\mathsf{m}^{\mathsf{q,ie}}_{\mathsf{opt,fin}}$	> 409,969,247	> 2,726,093,658	?

Table 5: Size of state space of finite version of our models, before and after reduction. (Only incomplete numbers for models $\mathsf{m}^{\mathsf{q}}_{\mathsf{opt,fin}}$ and $\mathsf{m}^{\mathsf{q,ie}}_{\mathsf{opt,fin}}$ available—state space generation for them aborted, probably because the state space generator ran out of memory.)

```
(* each ConnectRequest is followed by a ConnectAcknowledge *)
[ true* . ConnectRequest ]  < { ConnectAcknowledge ?m:Nat } > true

(* each ConnectAcknowledge for a connection m
   is followed by a corresponding ConnectAuthenticate *)
[ true* . { ConnectAcknowledge ?m:Nat } ]
< { ConnectAuthenticate !m } > true

(* if, after sending a ConnectAcknowledge for connection m,
   the server does not receive a corresponding ConnectAuthenticate,
   it will not send any other message on the connection  *)
[ true* .  { ConnectAcknowledge ?m:Nat } ]
[ (not  { ConnectAuthenticate !m })* .
( { ServiceAdvertisementEvent !m ?n:Nat }
| { ServiceEnquiryEvent !m ?n:Nat }
| { Subscribe !m !"mConnectEvent" }
| { Subscribe !m !"mDisconnectEvent" }
| { Unsubscribe !m !"mConnectEvent" }
| { UnsSubscribe !m !"mDisconnectEvent" }
)
] false
```
Listing 2: MCL formulas—input for model checker evaluator4—used to verify Requirement 2.

5. Model-Based Testing of the XBus

This section describes the model-based testing activities from Section 3.5 in more detail. We focus on (1) test architecture, (2) faults discovered, and (3) test coverage.

5.1. Model-based integration testing in the first phase

Test architecture. We used the test architecture from Fig. 7 on the next page to test implementation i_1. We wanted to test the XBusManager, but we could

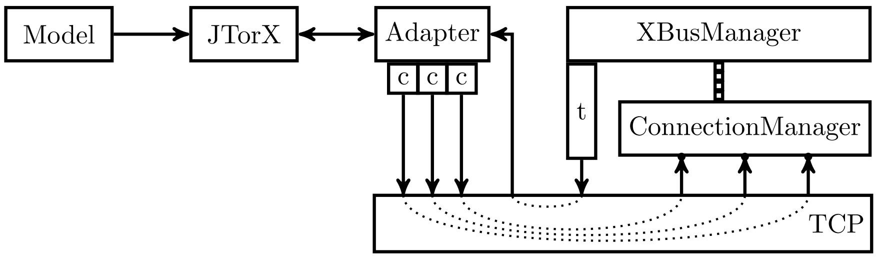


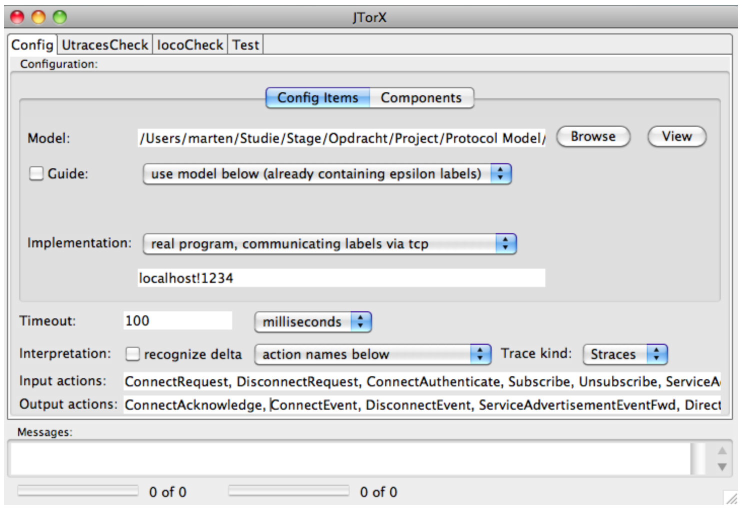
Figure 7: Test Architecture used in the first phase: JTorX provides stimuli to XBus via generic clients (c), and observes responses via test interface (t), both over TCP. Disadvantage: we have extended XBus with test interface t. Advantage: we do not have to extend the model with FIFO queues to deal with possible reordering of XBus responses by TCP.

not access it directly. We accessed it via a *test context*: everything between the adapter and the XBusManager. We provide stimuli to the XBusManager using three instances of XBusGenericClient (c in Fig. 7), each of which is connected to the XBus via its own TCP connection. We observe the responses from the XBus not via the XBusGenericClient, but via a direct (testing) interface that has been added to XBus—t in Fig. 7. This interface is provided by the JTorXTestableXBusManager in the Engine package, see Fig. 3 on page 11. JTorXTestableXBusManager overrides the function that XBus uses to send a message to a specified client: instead, it logs the message name and relevant parameters in the textual format that JTorX expects. Additional glue code—the adapter—provides the connection between JTorX and the XBusGenericClient instances on the one hand, and between JTorX and test interface t on the other hand. From JTorX, the adapter receives requests to apply stimuli, and from test interface t, it receives observed responses. The adapter forwards the received responses to JTorX without additional processing. For each received request to apply a stimulus, the adapter uses XBusGenericClient methods to construct a corresponding XBusMessage message, and send it to the XBus server (except for the $\mathsf{Conn}_{\mathsf{req}}$ message, for which XBusGenericClient only has to open a connection to XBus).

The adapter is implemented as a C# program that uses the Client package (see Fig. 3) to create the three XBusGenericClient instances, which in turn use the Communications package to interact with the XBus. The main functionality implemented in the adapter is the mapping between XBus messages and the corresponding XBusGenericClient methods, and the corresponding XBusGenericClient instances. Due to the one-to-one mapping that exists between these—by design, recall Section 3.2— implementing this mapping was rather straightforward.

Also JTorX and the adapter communicate via TCP: the adapter works as a simple TCP server to which JTorX connects as a TCP client.

It may seem that the Communications package does not play a role during model-based testing with this test architecture, also because we mentioned that we excluded it from the model. However, the Communications package is used normally in the XBus to receive the messages that clients send to it. Moreover, the only functionality of the Communications package that is not used in the XBus itself in this test architecture—the functionality to send messages over TCP—is used by the XBusGenericClient instances that are used to send the



Configuration:

Model: /Users/marten/Studie/Stage/Opdracht/Project/Protocol Model/

Guide: use model below (already containing epsilon labels)

Implementation: real program, communicating labels via tcp

localhost!1234

Timeout: 100 milliseconds

Interpretation: recognize delta action names below Trace kind: Straces

Input actions: ConnectRequest, DisconnectRequest, ConnectAuthenticate, Subscribe, Unsubscribe, ServiceA

Output actions: ConnectAcknowledge, ConnectEvent, DisconnectEvent, ServiceAdvertisementEventFwd, Direct

Figure 8: Screen shot of the configuration pane of JTorX, set up to test XBus. JTorX will connect to (the adapter that provides access to) the system under test via TCP on the local machine, at port 1234. The bottom two input fields list the input and output messages.

stimuli to the XBus.

Preserving observation order. As we wrote in Section 3.5, the TCP connections between XBus server and its client may reorder concurrently sent messages. We also wrote, that we assumed that this would not be a problem for stimuli. For observations we added an interface, t in Fig. 7, to allow JTorX to observe responses in the order in which they were created. For each incoming message the XBusManager sends at most one response to each connected client (during the analysis phase, requirement 14 was added to make sure that this property remained valid, even when we extended the model with local messages), and, the order in which the XBusManager sends the responses is exactly reflected in model $\mathsf{m}_{\mathsf{dev}}$. As we discuss later in this paper, in the analysis phase we also looked at other ways to deal with this issue, e.g. by extending the model with a queue context, hence models $\mathsf{m}^{\mathsf{q}}_{\mathsf{opt}}$ and $\mathsf{m}^{\mathsf{q,ie}}_{\mathsf{opt}}$. However, we found that code coverage obtained with these models was identical to code coverage obtained with the corresponding models without queues, but the test runs took a lot (5 to 10 times) longer.

Running JTorX. Once we had the model ($\mathsf{m}_{\mathsf{dev}}$), the XBus implementation to test (i_1), and the means to connect JTorX to it, testing was started. We ran JTorX in random mode. In the first phase, we used JTorX via its graphical user interface. Figure 8 shows the settings in the JTorX GUI. These include the location of the model file, the way in which the adapter and the XBus are accessed, and an indication of which messages are input (from the XBus server perspective) and which ones are output.

Bugs found in the first phase. One of the most interesting parts of testing is finding bugs. In this case, not only because it allows improving the software, but also because finding bugs can be seen as an indication that model based testing

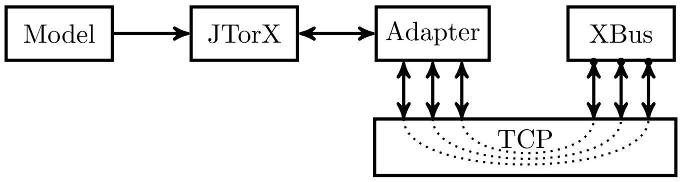


Figure 9: Test architecture used in the second phase: JTorX connects to XBus over TCP, where the TCP connections (one connection for each client of which JTorX plays the role) are used for both stimuli and responses. Advantage: XBus is unchanged. Disadvantage: we do have to extend the model to deal with the possibility that TCP reorders concurrently sent XBus responses (the responses sent for a single incoming message).

is actually helping us. We found 5 bugs when testing implementation i_1 (and a few more when testing implementation i_2—these we discuss in Section 5.2). Typically these bugs were found within 5 minutes after the start of a test. Some of them are quite subtle:

1. The $\mathsf{Notif}_{\mathsf{disc}}$ message was sent to unsubscribed clients. This was due to an if-statement that had a wrong branching expression.
2. The $\mathsf{Serv}_{\mathsf{ann}}$ message was sent (also) to unauthorized clients. Clients that were still in the handshake process with the server, and thus not fully authenticated, received the $\mathsf{Serv}_{\mathsf{ann}}$ message. To trigger this bug one client has to (connect and) announce its service while another client is still connecting.
3. The message subscription administration did not behave correctly: a client could subscribe to one item, but *not* to two or more. This was due to a bug in the operation that added the subscription to the list of a client.
4. The same bug also occurred with the list of provided services. It was implemented in the same way as the message subscription administration.
5. There was a flaw in the method that handles Unsub messages. The code that extracts subscriptions from these messages (to be able to remove them from the list of subscriptions of the corresponding client) contained a typing error: two terms in an expression were interchanged.

All these bugs concern the order in which protocol messages must occur. Therefore, it is our firm belief that they are much harder to discover with unit testing.

5.2. Model-based testing in the second phase

In the second phase, we ran JTorX on implementation i_2, with all (infinite) models except $\mathsf{m}_{\mathsf{dev}}$. We did not test i_2 against model $\mathsf{m}_{\mathsf{dev}}$, because $\mathsf{m}_{\mathsf{dev}}$ was designed for the test architecture of the first phase. Model $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ is a version of $\mathsf{m}_{\mathsf{dev}}$ that is exactly catered for the test architecture used in the second phase.

Test architecture. In the second phase, we chose a different architecture, see Fig. 9. Rather than the—quite complex—set up from the first phase, we chose to observe the SUT's responses via the same connections that are also used for the stimuli. This led to a simpler adapter and test set up, but required a more complex model: $\mathsf{m}^{\mathsf{order}}_{\mathsf{dev}}$. Observations are no longer observed via the test interface (block t in Fig. 7), but they were sent through TCP. For each XBus client—recall that JTorX plays the role of three XBus clients—there was a separate TCP connection between XBus server and adapter, such that responses might arrive at the adapter in an order, that differed from the order in which they were sent. We adapted the model to reflect this, in two ways. In model $\mathsf{m}^{\mathsf{order}}_{\mathsf{dev}}$ we directly included the different orders in the model. In models $\mathsf{m}^{\mathsf{q}}_{\mathsf{opt}}$ and $\mathsf{m}^{\mathsf{q,ie3}}_{\mathsf{opt}}$ we extended the model with a queue model that describes the behavior of the TCP channel.

Running JTorX. In the second phase, we mostly invoked JTorX via its non-graphical interface—a recent development, that did not yet exist in the first phase. We ran the tests on the same machine that we also used for the model-checking—it has two quad-core Intel Xeon X5555 processors and 144 GB of memory[4]. To ensure that the java virtual machine that ran JTorX had ample memory, we invoked it with command line options that allowed it to use 8GB.

We tested implementation i_2 with all models from Table 4, except for $\mathsf{m}_{\mathsf{dev}}$ which required the test architecture from the first phase. Table 6 on page 28 shows test execution time for runs of 10,000 and 250,000 test steps, and maximal attainable code coverage.

We discuss coverage results, and test execution time, in Sections 5.3–5.6.

Bugs found in the second phase. Testing in the second phase revealed bugs in implementation i_2. This does not help to improve the quality of i_1, but it does demonstrate the ability to find errors with our approach. We mention two bugs that we found most illustrative.

1. Implementation i_2 contained a race. Its TCP listener, that waits for new connections, would, after accepting a new connection c, do the following:

   (a) first obtain a data structure for the connection information, then

   (b) send a message m to the dispatcher, to inform it of c,

   (c) finally, update the data structure with details necessary to send messages over the new connection.

   This sequence contains a race: the implementation breaks when the dispatcher, after receiving message m, tries to send a $\mathsf{Conn}_{\mathsf{ack}}$ to the client, before the data structure update—necessary to be able to send that $\mathsf{Conn}_{\mathsf{ack}}$—has taken place. We could trigger this error with each of our models.

2. The adapter contained a resource leak. During a test run, it creates and closes many connections, but when closing a connection it did not release all associated resources. We found this when a long test run failed after approximately 125,000 test steps. Note that MBT excels at long test runs. This bug can easily be found by any test that runs long enough.

[3]Note that $\mathsf{m}^{\mathsf{q}}_{\mathsf{opt}}$ was derived from $\mathsf{m}_{\mathsf{opt}}$, rather than from $\mathsf{m}_{\mathsf{dev}}$, mainly because $\mathsf{m}_{\mathsf{opt}}$ was smaller and therefore easier to adapt.

[4]For model-based testing that machine was quite a bit oversized: we have also done test runs on a Macbook with a 2.4GHz Intel Core 2 Duo processor with 8GB of memory.

While extending model and implementation, we occasionally tested on purpose with old versions of the model or implementation, to see whether the resulting inconsistencies were found. Again, we mention two examples.

1. The list of services that appears in a $\mathsf{Serv}_{\mathsf{rsp}}$ message was not sorted correctly. We initially—on purpose—left out code to sort in the implementation, to see whether this bug would be detected; it was.

2. One version of the model incorrectly prescribed that in response to a $\mathsf{Local}_{\mathsf{req}}$, first a $\mathsf{Local}_{\mathsf{ind}}$ message is sent to the destination, and only then $\mathsf{Notif}_{\mathsf{local}}$ messages are sent to the subscribers. In the implementation, $\mathsf{Local}_{\mathsf{req}}$ messages are handled in precisely the same way. However, in a test run, a $\mathsf{Notif}_{\mathsf{local}}$ was observed first (due to reordering of responses by the TCP test context), while a $\mathsf{Local}_{\mathsf{ind}}$ was expected first. We adapted the model to allow observation of $\mathsf{Notif}_{\mathsf{local}}$ and $\mathsf{Local}_{\mathsf{ind}}$ in arbitrary order.

Once models and implementation i_2 were stable, we ran tests of up to 250,000 test steps without finding further errors.

5.3. Model coverage

LPS summand coverage. We used LPS summand coverage as our model coverage metric, i.e., the percentage of LPS summands that were hit during test execution. To test with an mCRL2 model, it has to be translated (by mcrl22lps from the mCRL2 toolset) into an intermediate format called *Linear Process Specification (LPS).* JTorX then accesses such LPS via tool lps2torx, also from the mCRL2 tool set[5]. An LPS represents a set of nondeterministic alternatives, called *summands.* A summand is a syntactic expression over model variables and parameters, containing a guard, an action to be executed, and a recursive invocation of the process. They express, respectively, when this alternative is enabled, the action to be taken, and the next state.

To measure LPS summand coverage, we extended the lps2torx tool from the mCRL2 tool set, so that each summand is assigned a unique identifier. During test execution, we record the identifiers of all executed summands and thus, LPS summand coverage can easily be computed.

Just as programs may contain unreachable code, models may contain unreachable summands, i.e. summands which are never executed, because their guard is never enabled. We do not take unreachable summands into account when we compute model coverage; we used model checking to do the analysis of summand reachability.

Coverage results. Figure 10 and Fig. 11 show the model coverage for test runs of 250,000 resp. 10,000 steps on models $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, $\mathsf{m}_{\mathsf{opt}}$, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$, and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$; recall that testing is done by taking random test steps. We see that models $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ and $\mathsf{m}_{\mathsf{opt}}$ reach 100% code coverage quickly (within 6,000 steps), model

[5] The LTSmin tool set also contains a tool lps2torx, that JTorX also can use to access an LPS. Throughout the experiments described in this paper we used $\mathsf{lps2torx}_{\mathsf{mCRL2}}$, except in the analysis of testing time, where, as discussed in Section 5.6 on page 28, we also used $\mathsf{lps2torx}_{\mathsf{LTSmin}}$. As we did above, when needed, we use subscripts to distinguish between these two lps2torx instances.

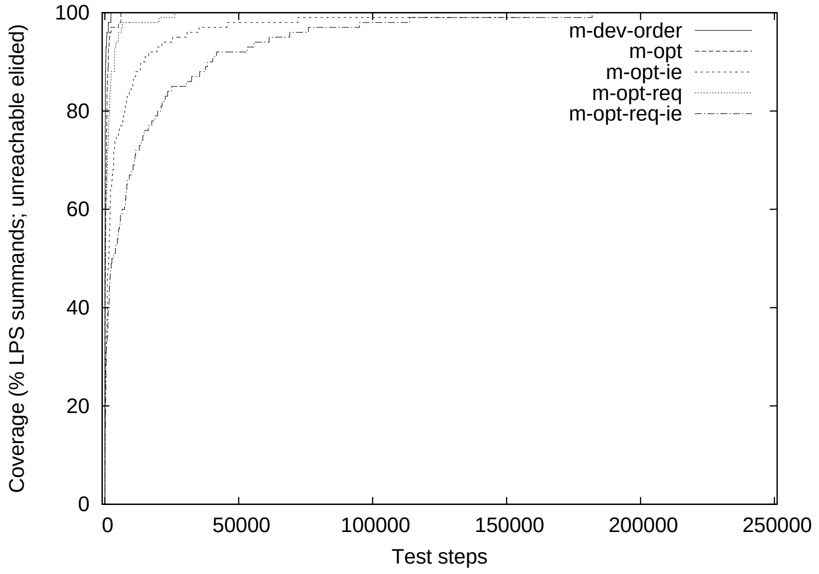


Figure 10: Model coverage obtained in test runs of 250,000 test steps.

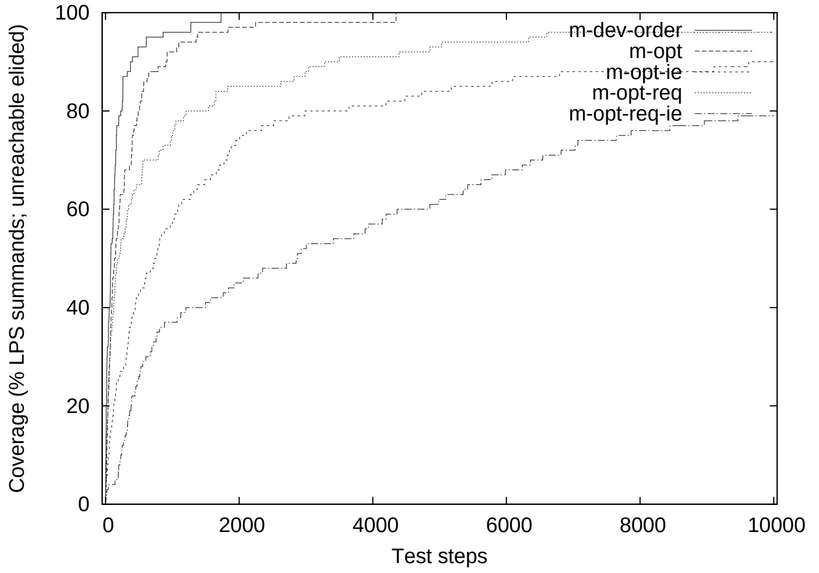


Figure 11: Model coverage obtained in test runs of 10,000 test steps.

$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$ takes somewhat more steps (slightly over 26,000), while model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$ needs about 150,000 steps, and model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ needs about 180,000 steps. We do not show model coverage results for models $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$, because the unreachable summand analysis did not terminate.

5.4. Code coverage

Branch coverage. We used branch coverage as our code coverage metric. Branch coverage [17] is a standard code coverage metric that counts the percentage of branches traversed in a program's control flow graph during test execution. It was measured by instrumenting the code.

Initial coverage analysis showed that 19 blocks were unreachable, because of the following reasons. Two blocks handle operating system errors, which never occurred in our case—testing operating system related functionality requires a different test set up, where we simulate the operating systems, and deliberately insert errors. Four blocks handle Sub and Unsub messages with an invalid message type—such messages do not appear in our most complete model, though they can easily be added (we leave that for future work). Finally, thirteen blocks handle inherently unreachable cases. For example, when an incoming message is being handled, the list of active connections will always contain at least one element, namely the sending connection. Therefore, the code that looks up the connection record for a given connection, will never encounter an empty list of connections, and thus the code that handles the case of an empty list is unreachable. One could use static analysis tools to show that these blocks can never be executed, and then safely remove these blocks—but this falls beyond the scope of this paper.

Since coverage should, in our opinion, measure the code covered by a specific test as a percentage of what can be covered, we left out the unreachable blocks from our coverage analysis.

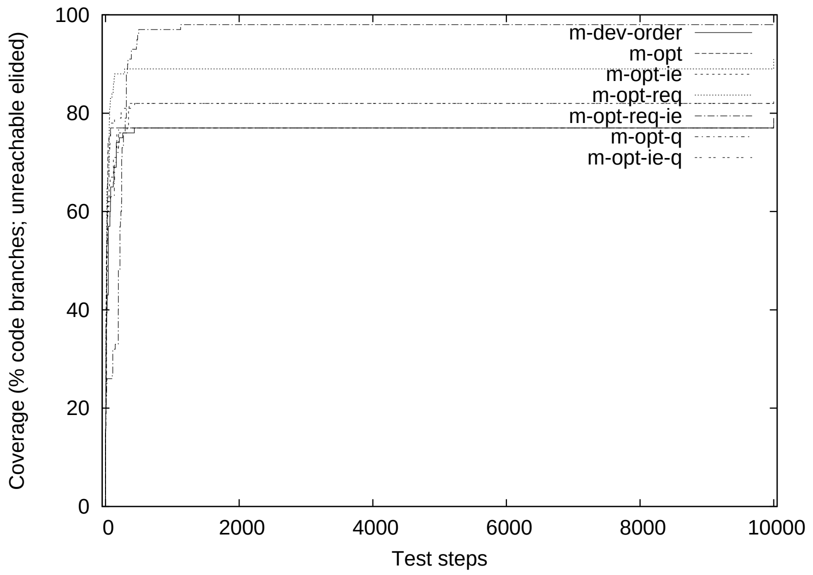


Figure 12: Code coverage (on i_2) obtained in test runs of 10,000 test steps. Note that coverage for $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, $\mathsf{m}_{\mathsf{opt}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$ converges to the same level, and so does coverage for $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$. From the test runs of 250,000 steps, we obtain an almost identical plot (not shown).

Coverage results. Figure 12 shows the code coverage results for all models, for test runs of 10,000 steps. These experiments reveal two interesting phenomena: (1) the maximum attainable code coverage varies per model, and (2) the use of queues does not affect maximum attainable coverage.

The maximum attainable code coverage figures are shown in Table 6 on the next page. As expected, we see that, the more complete a model is, the higher the maximum code coverage is: $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ is the least complete model with 79% maximal code coverage: since $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ does not contain self-defined XBus messages, it can not trigger all behavior in the implementation. Model $\mathsf{m}_{\mathsf{opt}}$ reached the same coverage. This is no surprise either, because $\mathsf{m}_{\mathsf{opt}}$ is an optimized version of $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$.

As we explain in Section 5.6, test execution from $\mathsf{m}_{\mathsf{opt}}$ is significantly faster. Also $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$ reaches 79% code coverage. This is interesting, because, apparently, the use of queues does not affect maximal code coverage. Indeed, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$ reach the same maximal coverage, namely 83%. The most complete model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ reaches 100% coverage. From these experiments, we show that measuring code coverage is important: if 100% code coverage cannot be reached, then the model is incomplete, so not all behavior can be tested. If this is the case, we advice to extend the model.

5.5. Distribution of coverage

In Figures 13 and 14 we see that all “hit” code blocks and all “hit” LPS summands were hit multiple times, although the number of hits is not evenly distributed. (Note the logarithmic scale on the vertical axis.) For the code coverage, obviously, certain blocks are hit quite often, e.g. because they are hit whenever an incoming message has to be processed, whereas other blocks are only hit once, during initialization—this explains the “gap” slightly at the right of block “0” in Figure 14.

For the plot of the model coverage (Fig. 13), it could be interesting to separate the stimuli from the responses, to see to what extent the following hypothesis is true: for stimuli, there is a direct correspondence between the number of actions that are generated from a summand, and the number of times that the summand is “hit”.

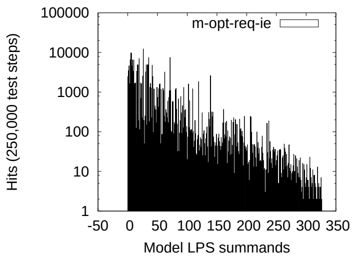


Figure 13: Model coverage obtained in a test run of 250,000 test steps with model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, showing, for each LPS summand of the model, how often it is hit. LPS summands are ordered, in order of first “hit”.

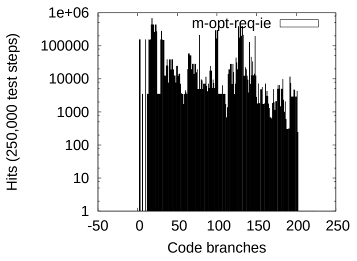


Figure 14: Code coverage obtained in a test run of 10,000 test steps with model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, showing, for each branch of the implementation, how often it is hit. Branches are ordered, in order of first “hit”.

model	*10,000 steps*	*250,000 steps*	*250,000 steps jittyc*	*max att. code coverage*
$\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$	18 minutes	24 hours	15.75 hours	79%
$\mathsf{m}_{\mathsf{opt}}$	5 minutes	2 hours	2 hours	79%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$	6 minutes	2.5 hours	2.25 hours	83%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$	6 minutes	3 hours	2.5 hours	91%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$	7 minutes	3 hours	2.5 hours	100%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$	37 minutes	77.75 hours	69.75 hours	79%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$	30 minutes	44.5 hours	38 hours	83%

Table 6: Wall-clock time for runs on implementation i_2, using JTorX in non-GUI mode, and the maximal attainable code coverage for each model.

5.6. Testing time

We also analyzed the test execution times, see Table 6 and Figures 15–20.

Table 6 shows the test execution times for the runs of 10,000 and 250,000 test steps. The 4th column shows the effect of enabling option *jittyc* of tool lps2torx; all time-related results that we show in the plots were obtained with this option enabled. When option *jittyc* is enabled, lps2torx uses a jit-compiled rewriting engine, instead of its interpreting rewriting engine—the more rewriting that has to be done, the greater the gain. Jit-compilation takes approx. 11 seconds at the start of a test run; this is not shown in Figures 16, 17 and 18, to avoid compressing the scale on the vertical axis.

Figures 15, 16, 17 and 18 present scatter plots showing, for each test step generated from respectively model $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, $\mathsf{m}_{\mathsf{opt}}$, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ (accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$), and model $\mathsf{m}_{\mathsf{opt}}$ (accessed using $\mathsf{lps2torx}_{\mathsf{LTSmin}}$), the amount of time in milliseconds it takes to execute. ($\mathsf{lps2torx}_{\mathsf{mCRL2}}$ and $\mathsf{lps2torx}_{\mathsf{LTSmin}}$ were introduced in the footnote on page 24.) Thus, in these plots a point at test step 12743 at testing time 300, means that the 12743th test step took 300 *ms*. Figures 19 and 20 present the same information differently: for each test step duration d, they shows the number of test steps that took d *ms* to execute.

Figure 16 shows two tick areas. One is below 20 *ms*, showing that most test steps took less than 20 *ms*. Another tick area is around 100 *ms*, which is exactly the value of the quiescence timer. This is to be expected: if one wants to observe quiescence (i.e. absence of outputs), one observes the system for (in our case) 100 *ms* and sees if any outputs are produced. Thus, if quiescence is observed, this step takes exactly 100 *ms*. Also the plots for models $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ contain these same two tick areas, but we have to zoom in sufficiently to distinguish them; they are also visible separately in Figures 17 and 18 but not in Fig. 15, mainly due to the different scale on the vertical axis.

Figures 15–18 show the time needed per test step as a function of the total number of test steps executed. Those plots where the model was accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$ show few (Fig. 16) resp. a significant portion (Fig. 15, 17) of test steps whose execution time grows linearly with the number of test steps executed. This may be surprising, because one would assume that executing a single test step requires a fixed amount of time. We attribute the linear behavior to the growth of state mappings in $\mathsf{lps2torx}_{\mathsf{mCRL2}}$: both lps2torx instances maintain a mapping between the state representation that they uses internally, and the state identifiers (numbers) that they exchange over their interface with JTorX. With $\mathsf{lps2torx}_{\mathsf{mCRL2}}$, more or less regularly, a map insertion takes more time, when the map adjusts itself to cope with the ever growing number of entries.

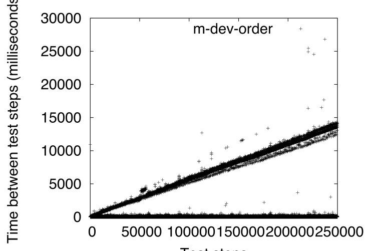


Figure 15: Time (ms) between test steps in run of 250,000 test steps with model $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ (accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$).

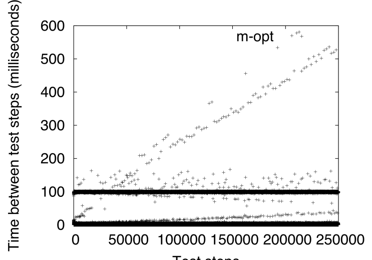


Figure 16: Time (ms) between test steps in run of 250,000 test steps with model $\mathsf{m}_{\mathsf{opt}}$ (accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$).

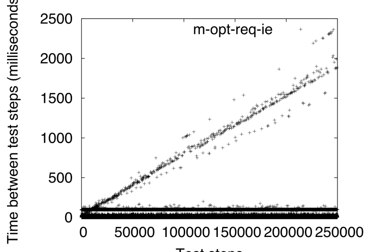


Figure 17: Time (ms) between test steps in run of 250,000 test steps with model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ (accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$).

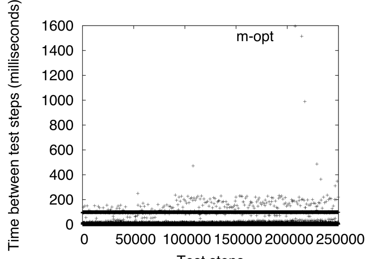


Figure 18: Time (ms) between test steps in run of 250,000 test steps with model $\mathsf{m}_{\mathsf{opt}}$ (accessed using $\mathsf{lps2torx}_{\mathsf{LTSmin}}$).

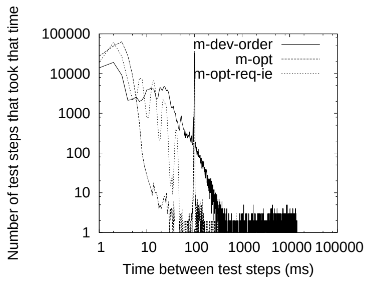


Figure 19: Distribution of the time spent per test step, for runs of 250,000 steps with models $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, $\mathsf{m}_{\mathsf{opt}}$, and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$. Note logarithmic scale on both axes.

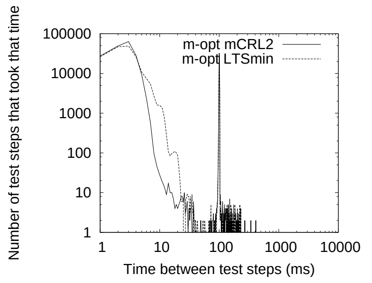


Figure 20: Distribution of the time spent per test step, for runs of 250,000 steps with model $\mathsf{m}_{\mathsf{opt}}$, accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$ resp. $\mathsf{lps2torx}_{\mathsf{LTSmin}}$. Note logarithmic scale on both axes.

With $\mathsf{lps2torx}_{\mathsf{LTSmin}}$ this appears to happen much less often, but when it does happen, it takes much longer than with $\mathsf{lps2torx}_{\mathsf{mCRL2}}$. (For this paper we did not do a full performance comparison between $\mathsf{lps2torx}_{\mathsf{mCRL2}}$ and $\mathsf{lps2torx}_{\mathsf{LTSmin}}$, we leave that for future research. The measurements that we did suggest that $\mathsf{lps2torx}_{\mathsf{LTSmin}}$ spends, in general, slightly more time on a model-access request from JTorX than $\mathsf{lps2torx}_{\mathsf{mCRL2}}$. Nevertheless, with both $\mathsf{lps2torx}$ instances, the same test run of 250,000 test steps with model $\mathsf{m}_{\mathsf{opt}}$ took the same two hours of wall clock time.) The difference in severity of the linear growth, that we see across Figures 15–17, we attribute to the different numbers of states that the mappings contain. After 250,000 test steps, with model $\mathsf{m}_{\mathsf{opt}}$ the mapping only contains approx. 900,000 states; with model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ approx. 6,300,000, and with model $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ approx. 35,000,000. To understand this huge difference, recall that in model $\mathsf{m}_{\mathsf{dev}}$ (and thus also in $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$) a single event could result (non-deterministically) in multiple different configurations of the client administration data structures, i.e. in multiple different states, whereas $\mathsf{m}_{\mathsf{opt}}$ is almost fully deterministic. Note however, that in Figures 15–18 the long test steps are a small fraction of all steps. Figures 19 and 20 show that the majority of all test steps take less than 1000 *ms*. In our experiments, test step derivation time was not a bottleneck, but it could be an issue when testing real-time systems.

6. Findings and Lessons Learned

6.1. First phase

The internship in a time perspective. So how long did it take to create the artefacts for model-based testing, namely the model, the test interface and the adapter? Programming and simulating the model took 2 weeks, or 80 hours. The test interface was created in a few hours, since it was designed to be loosely coupled to the engine. It was a matter of a few dozens lines of code. The adapter was created in two days, or 16 hours. Thus, given the total project time of 14 weeks, creating the artefacts needed for model-based testing took thus about 17% of our time.

The modeling process. Writing a model takes a significant amount of time, but also forces the developer to think about the system behavior thoroughly. Moreover, we found it extremely helpful to use simulation to step through the protocol, before implementing anything. Making and simulating a model gives a deep understanding of the system, in an early stage of development, from which the architectural design profits.

6.2. Second phase

After a thorough analysis of the model-based testing process that was carried out in the first phase, the question remains how good the approach was. We reflect on the questions raised in Section 1.2.

How good was the model? Model $\mathsf{m}_{\mathsf{dev}}$ did its job, but there is certainly room for improvement. In particular, completeness with respect to the requirements and bad weather behavior could be improved.

Was the testing thorough enough? Given model $\mathsf{m}_{\mathsf{dev}}$, we believe that testing was thorough enough. On the other hand, model-based testing is as good as the model is, so more complete models also mean better testing, as resulting in higher code coverage.

What can we say about code coverage? The model $\mathsf{m}_{\mathsf{dev}}$ does not reach 100% code coverage. For that, more complete models are required.

Would model checking have helped to produce better code? Formalizing the requirements, which is a prerequisite for model checking, helps to improve the model, and therefore the code. However, model checking requires great effort, because models need to be made finite and efficient. For model-based testing, this was not needed, since performance was not an issue here.

Despite these observations, we still believe in our approach during the first phase. If we had to redo the XBus development we would take a very similar approach, but (1) invest more effort in the modeling phase: trace back the requirements, and make models input-complete, and (2) measure coverage.

7. Conclusions and Future Research

We conclude that the approach of using formal methods in both the design step and the integration testing step of the V-model was a success: with a relatively limited effort, we found five subtle bugs. We needed 17% of the time to develop the artifacts needed for model-based testing, and given the errors found, we consider that time well spent. Moreover, for future versions of the XBus, JTorX can be used for automatic regression tests: by adapting the mCRL2 model to new functionality, one can detect automatically if new bugs are introduced.

Our post-case study analysis showed that a 14 week development process is feasible but short: the model quality would have benefited from more attention—in particular, tracing back the requirements would have been helpful.

The test execution time analysis results suggest that performance improvements can be made by optimizing the interface between JTorX and the lps2torx tools of the mCRL2 and LTSmin tool sets; for this, further measurements and analysis will be necessary.

Thus, the post-internship analysis gave us a deeper understanding of the limitations and the successes of the work done during the internship, an increased understanding of what factors are responsible for the successes, and valuable feedback that may help us to improve our tools.

References

[1] M. Sijtema, M. I. A. Stoelinga, A. F. E. Belinfante, L. Marinelli, Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at neopost, in: G. Salaün, B. Schätz (Eds.), FMICS 2011, volume 6959 of *LNCS*, Springer, 2011, pp. 117–133.

[2] M. Ferreira, V. Romanenko, Programme Booklet of the 16th Dutch Testing Day, 2010.

[3] TechWatch, Bits &Chips conference on Embedded Systems, 2011.

[4] S. Kowalewski, M. Roveri (Eds.), FMICS 2010, volume 6371 of *LNCS*, Springer, 2010.

[5] D. D. Cofer, A. Fantechi (Eds.), FMICS 2008, Revised Selected Papers, volume 5596 of *LNCS*, Springer, 2009.

[6] H. H. Hansen, J. Ketema, S. P. Luttik, M. R. Mousavi, J. C. van de Pol, Towards model checking executable UML specifications in mCRL2, Innovations in Systems and Software Engineering 6 (2010) 83–90.

[7] A. Ferrari, D. Grasso, G. Magnani, A. Fantechi, M. Tempestini, The Metrô Rio ATP case study, in: [4], pp. 1–16.

[8] N. G. Leveson, Experiences in designing and using formal specification languages for embedded control software, in: [9], p. 3.

[9] N. A. Lynch, B. H. Krogh (Eds.), HSCC 2000, volume 1790 of *LNCS*, Springer, 2000.

[10] P. E. Rook, Controlling software projects, IEEE Software Engineering Journal 1 (January 1986) 7–16.

[11] J. F. Groote, et al., The mCRL2 toolset, in: Proc. International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008), 2008, pp. 5/1–10.

[12] mCRL2 Toolkit webpage, `http://www.mcrl2.org/`, 2012.

[13] A. Belinfante, JTorX: A tool for on-line model-driven test derivation and execution, in: TACAS 2010, volume 6015 of *LNCS*, Springer, 2010, pp. 266–270.

[14] JTorX webpage, `http://fmt.ewi.utwente.nl/tools/jtorx/`, 2011.

[15] CADP evalutor4 manual webpage, `http://cadp.inria.fr/man/evaluator4.html`, 2012.

[16] R. Mateescu, D. Thivolle, A model checking language for concurrent value-passing systems, in: Proceedings of the 15th International Symposium on Formal Methods FM'08, volume 5014 of *LNCS*, 2008, pp. 148–164.

[17] G. J. Myers, The Art of Software Testing, Wiley, 1979.

[18] H. Garavel, C. Viho, M. Zendri, System design of a CC-NUMA multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation, STTT 3 (2001) 314–331.

[19] Neopost Inc. webpage, `http://www.neopost.com/`, 2009.

[20] M. Veanes, et al., Model-based testing of object-oriented reactive systems with Spec Explorer, in: Formal Methods and Testing, volume 4949 of *LNCS*, Springer, 2008, pp. 39–76.

[21] Conformiq webpage, `http://www.conformiq.com/`, 2011.

[22] A. Hartman, K. Nagin, The AGEDIS tools for model based testing, SIGSOFT Softw. Eng. Notes 29 (2004) 129–132.

[23] A. Belinfante, L. Frantzen, C. Schallhart, Tools for test case generation, in: Model-Based Testing of Reactive Systems: Advanced Lectures, volume 3472 of *LNCS*, Springer, 2005, pp. 391–438.

[24] A. Hartman, Model based test generation tools, Technical Report, AGEDIS Consortium, 2002. `http://agedis.de/documents/ModelBasedTestGenerationTools.pdf`. Accessed on June 21, 2012.

[25] GraphML file format, `http://graphml.graphdrawing.org/`, 2012.

[26] Jararaca manual, `http://fmt.cs.utwente.nl/tools/torx/jararaca.1.html`, 2012.

[27] L. Frantzen, J. Tretmans, T. A. C. Willemse, A symbolic framework for model-based testing, in: Formal Approaches to Software Testing and Runtime Verification, volume 4262 of *LNCS*, Springer, 2006, pp. 40–54.

[28] S. C. C. Blom, J. C. van de Pol, M. Weber, Bridging the Gap between Enumerative and Symbolic Model Checkers, Technical Report TR-CTIT-09-30, CTIT, University of Twente, Enschede, 2009.

[29] H. Garavel, et al., CADP 2006: A toolbox for the construction and analysis of distributed processes, in: CAV 2007, 2007, pp. 158–163.

[30] A. Belinfante, et al., Formal test automation: A simple experiment, in: IWTCS 1999, Kluwer, 1999, pp. 179–196.

[31] J. Tretmans, H. Brinksma, TorX: Automated model-based testing, in: A. Hartman, K. Dussa-Ziegler (Eds.), First European Conference on Model-Driven Software Engineering, Nuremberg, Germany, 2003, pp. 31–43.

[32] J. Tretmans, Test generation with inputs, outputs, and repetitive quiescence, Software - Concepts and Tools 17(3) (1996) 103–120.

[33] J. Tretmans, Model Based Testing with Labelled Transition Systems, in: Formal Methods and Testing, volume 4949 of *LNCS*, Springer, 2008, pp. 1–38.

[34] H. M. van der Bijl, A. Rensink, J. Tretmans, Compositional testing with ioco, in: FATES 2003, volume 2931 of *LNCS*, Springer, 2004, pp. 86–100.

[35] M. Timmer, E. Brinksma, M. Stoelinga, Model-based testing, in: Software and Systems Safety: Specification and Verification, NATO Science for Peace and Security Series - D, IOS Press, 2011.

[36] A. David, K. G. Larsen, S. Li, B. Nielsen, Timed testing under partial observability, in: ICST, IEEE Computer Society, 2009, pp. 61–70.

[37] K. G. Larsen, M. Mikucionis, B. Nielsen, Online testing of real-time systems using UPPAAL: Status and future work, in: Perspectives of Model-Based Testing, volume 04371 of *Dagstuhl Seminar Proceedings*, 2004.

[38] L. Brandán Briones, Theories for Model-based Testing: Real-time and Coverage, Ph.D. thesis, University of Twente, 2007.

[39] W. Grieskamp, X. Qu, X. Wei, N. Kicillof, M. B. Cohen, Interaction coverage meets path coverage by SMT constraint solving, in: TESTCOM 2009 and FATES 2009, volume 5826 of *LNCS*, Springer, 2009, pp. 97–112.

[40] H. C. Bohnenkamp, M. I. A. Stoelinga, Quantitative testing, in: Proceedings of the 7th ACM International conference on Embedded software, ACM, New York, 2008, pp. 227–236.

[41] J. A. Bergstra, J. W. Klop, Algebra of communicating processes, in: J. W. de Bakker, M. Hazewinkel, J. K. Lenstra (Eds.), Proceedings of the CWI Symposium on Mathematics and Computer Science, CWI, Amsterdam, The Netherlands, 1985.

[42] The Go Programming Language webpage, `http://golang.org/`, 2012.