Experiences with Formal Engineering: Model-based Specification, Implementation and Testing of a Software Bus at Neopost. *

M. Sijtema[1], M.I.A. Stoelinga[2], A. Belinfante[2], and L. Marinelli[3]

[1] Sytematic Software, the Hague, the Netherlands marten@sytematic.nl
[2] Faculty of Computer Science, University of Twente, The Netherlands
{marielle,axel.belinfante}@cs.utwente.nl
[3] Neopost, Austin, Texas, USA l.marinelli@neopost.com

Abstract. We report on the actual industrial use of formal methods during the development of a software bus. At Neopost Inc., 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, 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.

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 [27, 15].

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 [24, 17, 27, 29, 30], we investigate this claim for a more modest and non-safety-critical project, namely the development of a software bus.

Developing the XBus In this paper, we report on our experiences with formal methods during the development of the XBus at Neopost Inc. Neopost is one

* This research has been partially funded by NWO and DFG by grant Dn 63-257 (ROCKS), and by the European Union under FP7-ICT-2007-1 grant 214755 (QUASIMODO).

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 (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 [31], see Fig. 2, using formal methods during the design and testing phase. The total running time of this project was 14 weeks.

An important step in the design phase was the creation of a behavioral model of the XBus, written in the process algebra mCRL2 [23, 4]. This model pins down the interaction between the XBus and its environment in a mathematically precise way. Performing this modeling activity greatly increased the understanding of the XBus protocol, which made the implementation phase a lot easier.

Testing of the XBus After implementing the protocol, we tested the implementation, where we distinguished between data- and protocol behaviour. Data behaviour concerns the input/output behaviour of a function. This behaviour is static, the input/output behaviour is independent of the order in which the methods are called. Protocol behaviour 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. First, data behaviour was tested using unit testing, and all errors found were repaired. Then, protocol behaviour was tested using JTorX (details below), since the purpose of the mCRL2 model was exactly to pin down the protocol behaviour.

With JTorX we tested the implementation against the mCRL2 model. JTorX [7, 3] is a model-based testing tool (partly) developed during the Quasimodo project [6]. It is capable of automatic test generation, execution and evaluation. During the design phase, we already catered for model-based testing, and designed for testability: we took 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, to use model-driven test 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.

Our findings We ran JTorX against the implementation and the mCRL2 model (once configured, JTorX runs completely automatically) and found five subtle bugs that were not discovered using unit testing, since these involved the order in which protocol messages should occur. After repairing these, we ran JTorX several times for more than 24 hours, without finding any more errors.

Since writing the model, simulating it, and testing the implementation with JTorX only took 17% of the total development time (counting only human working time), 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. Therefore, as in [19], we clearly show that formal engineering is not only beneficial for large, complex and/or safety-critical systems, but also for more modest projects.

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. Section 4 reflects on the lessons learned in this project and finally, we present conclusions and suggestions for future work in Section 5.

2 Background

2.1 The XBus and its context

Neopost Incorporated [5] 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[4] 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 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

Model-based testing Model-based testing (MBT, a.k.a. model-driven testing) is an innovative testing methodology that provides methods for automatic test

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

generation, execution and evaluation. Model-based testing requires a formal model m, usually a transition system, of the system-under-test (SUT, a.k.a. implementation-under-test or IUT). This model m pins down the desired system behavior in an unambiguous way: traces of m are correct system behaviors, and traces not in m are incorrect.

The concept of model-based testing is visualized in Fig. 1. Tests derived from a model m are applied to the SUT, and 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, which is a component of the tester in Fig. 1. 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, it was an important design rationale us to keep the adapter simple, which we achieved via a close correspondence between m and the system architecture.

However, given a model m and the 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 [34], Conformiq Qtronic [1], and AGEDIS [26]. Each of these tools varies in the capabilities, modeling languages and underlying theories, see [8, 25] for an overview.

JTorX These techniques have been implemented in the model-based test tool JTorX. JTorX [7, 3] was (partly) developed during the Quasimodo project [6]. It improves over its predecessor TorX [9, 33]—which was one of the first model-based testing tools in the field. JTorX is based on newer theory, 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 particularly helpful in this case study.

JTorX has built-in support for models in graphml [21], the Aldebaran (.aut) file format, and the Jararaca [2] file format. Moreover, it is able to access models via the mCRL2 [23], LTSmin [11] and CADP [20] tool environments.

In JTorX 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 we do not explicitly show test cases in Fig. 1.

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, i.e., they yield the correct verdict when executed against any implementation, see e.g. [32]. More technically, the test case derivation methods underlying JTorX are provably correct, i.e. have been shown *sound* and *complete*. That is, any correct implementation of a model m will pass all tests derived from m (soundness).

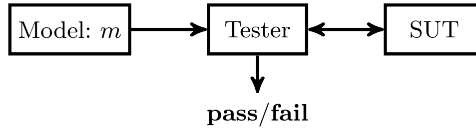

Fig. 1. Model-based testing.

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. Hence, the test case exhibiting the error may or may not be among those test cases that are executed. As stated by the famous quote by Dijkstra: "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 [16, 28, 13], data-intensive systems [18], object-oriented systems [22], and systems with measure imprecisions [12]. Several of these extensions have been developed during the Quasimodo project as well.

2.3 The specification language mCRL2

The language mCRL2 [23, 4] is a formal modeling language for describing concurrent systems, developed at the Eindhoven University of Technology. It is based on the process algebra ACP [10], 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 $X(x_1 : D_1, x_2 : D_2, x_n : D_n) = t$, 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 potentially parameterized actions and the operators alternative composition, sum, sequential composition, conditional choice (if-then-else), parallel composition, and 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` format, they can be proven correct using e.g. the theorem prover PVS, and they can be tested against with JTorX.

3 Development of the XBus

We developed the XBus implementation using the classical V-model [31], see Fig. 2. In our approach we have three testing phases: unit testing, integration testing and acceptance testing.

The sequel describes the activities carried out in each phase of the V-model. Each section below corresponds to an activity in the V-model. As stated, the total running time of the XBus development was 14 weeks.

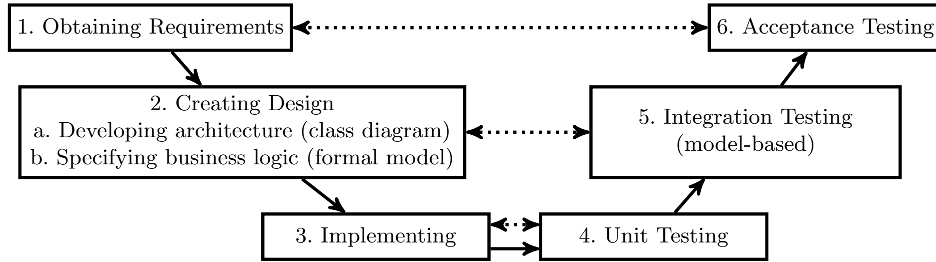

Fig. 2. The V-model that was used for development of XBus.

3.1 XBus Requirements

We have 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. Below, we summarize the functional requirements; as said before, important non-functional requirements are testability, maintainability and backwards compatibility with the XBus 1.0.

Functional requirements are as follows.

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 $\mathsf{Conn}_{\mathsf{req}}$—$\mathsf{Conn}_{\mathsf{ack}}$—$\mathsf{Conn}_{\mathsf{auth}}$ 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 are able to announce services they provide, using the $\mathsf{Serv}_{\mathsf{ann}}$ message.
8. Clients can inquire about services, by specifying a list of service names in a $\mathsf{Serv}_{\mathsf{inq}}$ message. Service providers that provide a subset of the inquired services will respond to this client with the $\mathsf{Serv}_{\mathsf{rsp}}$ 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.

XBus protocol messages are the following.

$\mathsf{Conn_{req}}$ (implicit) implied by a client establishing a TCP connection with XBus
$\mathsf{Conn_{ack}}$ sent from XBus to a client just after the client establishes a TCP connection with the XBus, as part of the handshake.
$\mathsf{Conn_{auth}}$ sent from a client to the XBus to complete the handshake.
(Un)Sub 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.
$\mathsf{Notif_{conn}}$ sent from XBus to clients that subscribed connect notifications.
$\mathsf{Notif_{disc}}$ sent from XBus to clients that subscribed disconnect notifications.
$\mathsf{Serv_{ann}}$ 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.
$\mathsf{Serv_{inq}}$ sent (just after connecting) from client to XBus, which broadcasts it to all other connected clients, to ask what services they provide.
$\mathsf{Serv_{rsp}}$ sent from a client via XBus to another client, as response to $\mathsf{Serv_{inq}}$, to tell the inquirer what services the responding client provides.

3.2 XBus Design

The design phase encompassed two activities: we created an architectural design, given by the UML class diagram in Fig. 3, and we made an mCRL2 model, describing the protocol behavior. An important feature of the UML design is that is already catered for model-based testing.

The architectural design and the mCRL2 model 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 the mCRL2 model. The UML diagram specifies which methods are provided, and the mCRL2 model 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.

Architectural Design The architecture of the XBus is given in Fig. 3, following 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.

We have catered for model-based testing already in the design: the XBusManager has a subclass JTorXTestableXBusManager. As we elaborate in Section 3.5, the JTorXTestableXBusManager class overrides the send message from the XBusManager, allowing JTorX to have more control over the state of the XBus server.

Modeling strategy When creating the model, the first step is to define *what* and *what not* to model, to determine the abstraction level and boundaries of the model.

Included in the model The messages that come into the server, their handling and their response messages are modeled. The handling of the messages is modeled as follows. After a message is received, the server will handle it. This means that the server will send a response, relay the message, broadcast a message,

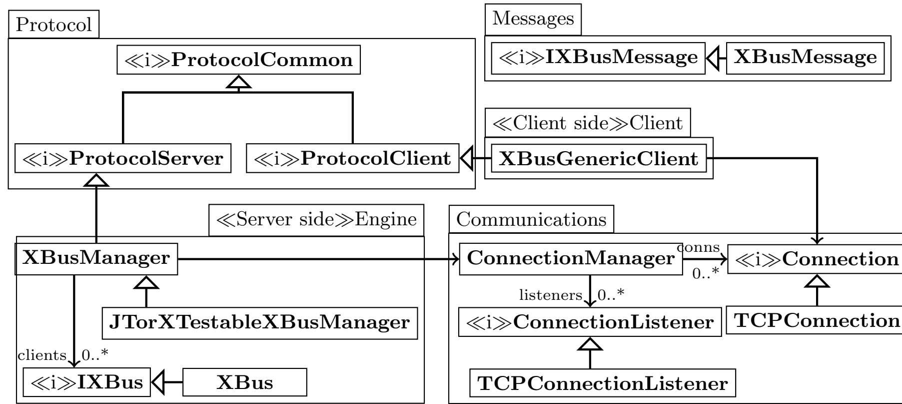

Fig. 3. High level architecture of the XBus system. It contains a server side package, and a client side package. Furthermore, it has functionality for TCP connections and XBus messages. Both server and client implement the Protocol abstract class. All interfaces are indicated with ≪i≫.

and/or modify its internal state, depending on the type of message that arrived. Furthermore, the server keeps track of the client's state by keeping an internal list of client objects, just as in the Engine package in the architecture.

Excluded from the model The Communications package is not included in the model. The model just describes that messages are received by and sent from the server (i.e. the XBusManager). This corresponds to the hand-over of incoming messages (from the perspective of XBus) from the Communications package to the Engine package, and the hand-over of outgoing messages in the opposite direction. So, the boundary between packages in the architecture corresponds with the boundary between the model and its environment.

Thus, we do *not* model internal components like TCP-sockets, queues, or programmatic events. The correctness of these internal components will be verified by unit tests. We will use the model discussed here to simulate and test the XBus protocol (i.e. the business logic).

The XBus model We modeled the desired behavior of the XBus as an mCRL2 process. We chose mCRL2 because of its powerful data types, which make the modeling of the messages and its parameters convenient. In particular, we benefitted from its concise notation for enumerated types, records, and lists, and the ability to define functions. Functions are defined in a functional programming style. They can be called from the server process, and can modify data structures defined in the data part.

Data All the data that the server keeps track of is kept in one data object: a list of clients. This is modeled as a list of data structures, that for each client contains the following items:

– an integer that represents the identity of the client;
– the connection status of the client, which is an enumeration of: disconnected, awaitingAuthentication, 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 $\mathsf{Conn}_{\mathsf{req}}$ 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.

We defined functions to model manipulations on these data types. Most of our functions operate on the lists of clients and the client's lists of subscriptions and services. Typical operations are insertion, lookup, update, and removal of items in these lists.

Behavior The behavior of the XBus server is modeled as a single process that—for all kinds of incoming messages that it may receive—accepts a message, processes it (which may involve an update of its state), and sends a response (where appropriate), after which it is ready to accept the next message.

Listing 1 shows part of the definition of this process (slightly simplified). The process is named listening. It has a single parameter: c, the list of clients. The fragment shows that for each client—where j is used as index in c (line 2)—that currently is in DISCONNECTED state (line 3), the server is willing to accept a ConnectRequest message, after which it will send out a ConnectAcknowledge message (line 4), after which it updates the status of the $\mathtt{j}^{th}$ client in the list to AWAIT_AUTH and continues processing—modeled by the recursive call to listening with the updated client list (line 5).

The language mCRL2 allows modeling of systems with multiple parallel processes, but this is not needed here. Having multiple concurrent processes would make the system as well as the model more complicated, which would make them harder to maintain and test. One might choose to use multiple processes when performance of the system is expected to be a problem, but that is not an issue here. In a large mailing room there may be 20 clients at the same time, a number with which the single-process server can easily cope.

Model size The entire model consists of 6 pages of mCRL2, including comments. 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, where we used 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 phase 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. Due to time constraints, model-checking was not performed. It would have allowed validation of (basic) properties like the absence of deadlocks, as well as checking more advanced properties, e.g. that every message sent eventually reaches the specified destination(s).

3.3 Implementation

Once we had sufficient confidence in the quality of the design—to a large extent due to modeling and simulation—it was implemented. The programming language used was C#—use of .NET is company policy.

3.4 Unit Testing

As mentioned in the introduction, the overall test strategy was to test data behaviour using unit testing, and protocol behaviour using model-based testing. The classes in the Communications and Messages packages were therefore 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.

3.5 Model-based Integration Testing

After unit testing of data behaviour, we used model-based testing for the business logic, i.e. to test the interaction between XBus and its clients. This is because here the dynamic behavior, i.e., the order of protocol messages, crucially determines the correctness of the protocol.

Test architecture To test whether the XBus interacts correctly with its environment, we chose a test set up with 3 XBus clients, see Fig. 4. Thus, JTorX plays the role of 3 XBus clients, which are able to perform all protocol actions described in Section 3.1. We first discuss how we connected JTorX to the XBus server, and then we briefly discuss an alternative.

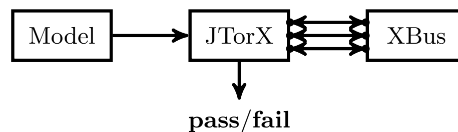

Fig. 4. Testing XBus with JTorX playing the role of 3 clients.

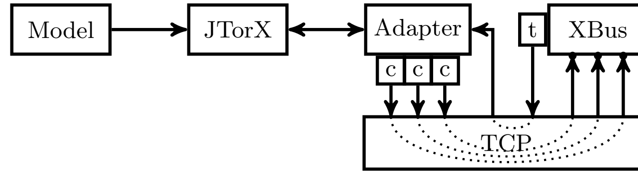

Fig. 5. The Test Architecture that we used: JTorX provides stimuli to XBus via generic clients (c) over TCP, and observes responses via test interface (t), also connected via TCP.

Our solution is depicted in Fig. 5. We provide stimuli to the XBus using three XBusGenericClient instances, each of which is connected to the XBus via TCP. We observe the responses from the XBus not via the XBusGenericClient, but via a direct (testing) interface that has been added to XBus. This interface is provided by the JTorXTestableXBusManager in the Engine package, see Fig. 3. JTorXTestableXBusManager overrides the function that XBus uses to send a message to a specified client, and instead 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 XBus test interface on the other hand. From JTorX the adapter receives requests to apply stimuli, and from the XBus test interface 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 that had to be implemented was 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.

JTorX and the adapter communicate via TCP: the adapter works as a simple TCP server to which JTorX connects as a TCP client. This is one of two possibilities offered by JTorX; which of the two is chosen does not really matter.

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 stimuli to the XBus.

An alternative approach would have been to not add the direct (testing) interface to XBus, to observe its responses, but to use the XBusGenericClient instances for

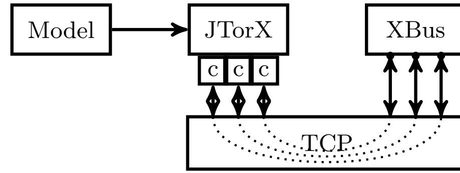

Fig. 6. Alternative solution: JTorX connected to XBus via generic clients (c) over TCP.

this, as depicted in Fig. 6. This alternative approach has the clear advantage that no additional (testing) interface has to be added to XBus, and thus the interaction via the XBusGenericClient instances is (in the perception of XBus) identical to the interaction during deployment.

However, this alternative approach also has one slight disadvantage. From the perspective of an observer of XBus responses, each of the TCP connections between an XBusGenericClient instance and the XBus resembles a first-in first-out (FIFO) queue, where a message that is sent later, over one connection, may overtake a message that was sent earlier over another connection. This means that the order in which XBus responses, via XBusGenericClient instances, arrive at the adapter, and thus ultimately at JTorX, may differ from the order in which XBus sends them. This, in turn, may result in incorrect fail verdicts—because the model does not reflect the FIFO-queue behavior of the TCP communication medium between XBus and the adapter. We have seen this reordering effect before, for example in our experiments with model-based testing of a simple chatbox protocol entity [9] and know that we can deal with it by extending the model with FIFO buffers that model the FIFO queue behavior of the TCP connections.

With both test architectures we have to deal with the reordering effect of the TCP connections, either by extending the XBus with a specific testing interface, or by extending the model. In this case we chose to extend the XBus.

Running JTorX Once we had the model, the XBus implementation to test, and the means to connect JTorX to it, testing was started. We ran JTorX in random mode. Figure 7 shows the settings in JTorX. 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 using JTorX 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 is actually helping us. We found 5 bugs. Typically these 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.

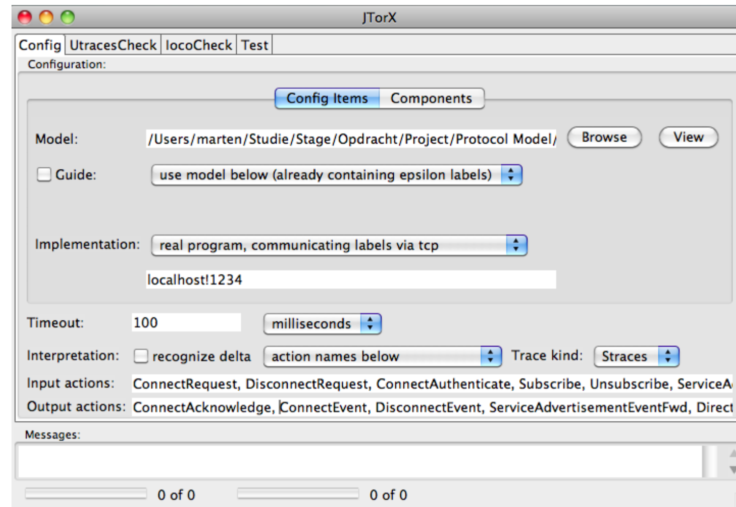

Fig. 7. Screen shot of the configuration pane of JTorX, set up to test XBus. JTorX will connect to (the adapter that provides access to) the system under test via TCP on the local machine, at port 1234. The bottom two input fields list the input and output messages.

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.

3.6 Acceptance Testing

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 worked. In particular, we demonstrated that it implements the features required in Section 3.1.

4 Findings and Lessons Learned

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.

Automated testing with JTorX Writing an adapter can be a large project, but in this case it was relatively straightforward. This can be attributed to having an architectural design that closely resembles the formal model, and a one-to-one mapping between the actual XBus messages and their model representation.

5 Conclusions and Future Research

We conclude that model-based testing using JTorX was a success: with a relatively limited effort, we found five subtle bugs. We needed 17% of the time to develop the artefacts 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.

We also conclude that making the formal model together with the architectural design had a positive effect on the quality of the design. Moreover, the resulting close resemblance between model and design simplified the construction of the adapter.

Although construction of the adapter was relatively straightforward, it would have been even easier if (parts of) the adapter could have been generated automatically, which is an important topic for future research. Another very important topic is to implement test coverage metrics (e.g. from [14]) in JTorX, so that we can quantify how thoroughly we have been testing.

References

1. Conformiq webpage (April 2011), http://www.conformiq.com/
2. Jararaca manual. http://fmt.cs.utwente.nl/tools/torx/jararaca.1.html
3. JTorX webpage (August 2009), http://fmt.ewi.utwente.nl/tools/jtorx/
4. mCRL2 toolkit webpage (September 2009), http://www.mcrl2.org/
5. Neopost Inc. webpage (August 2009), http://www.neopost.com/
6. Quasimodo webpage (August 2011), http://www.quasimodo.aau.dk/
7. Belinfante, A.: JTorX: A tool for on-line model-driven test derivation and execution. In: TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer (March 2010)
8. Belinfante, A., Frantzen, L., Schallhart, C.: Tools for test case generation. In: Model-Based Testing of Reactive Systems: Advanced Lectures, LNCS, vol. 3472, pp. 391–438. Springer (2005)
9. Belinfante, A., et al.: Formal test automation: A simple experiment. In: Csopaki, G., Dibuz, S., Tarnay, K. (eds.) IWTCS 1999. pp. 179–196. Kluwer (1999)
10. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes. In: de Bakker, J.W., Hazewinkel, M., Lenstra, J.K. (eds.) Proceedings of the CWI Symposium on Mathematics and Computer Science. CWI, Amsterdam, The Netherlands (1985)
11. Blom, S.C.C., van de Pol, J.C., Weber, M.: Bridging the gap between enumerative and symbolic model checkers. Technical Report TR-CTIT-09-30, Centre for Telematics and Information Technology, University of Twente, Enschede (2009)

12. Bohnenkamp, H.C., Stoelinga, M.I.A.: Quantitative testing. In: Proceedings of the 7th ACM International conference on Embedded software. pp. 227–236. ACM, New York (2008)
13. Brandán Briones, L.: Theories for Model-based Testing: Real-time and Coverage. Ph.D. thesis, University of Twente (March 2007)
14. Brinksma, E., Briones, L.B., Stoelinga, M.: A semantic framework for test coverage. In: ATVA'06. LNCS, vol. 4218, pp. 399–414 (2006)
15. Cofer, D.D., Fantechi, A. (eds.): FMICS 2008, Revised Selected Papers, LNCS, vol. 5596. Springer (2009)
16. David, A., Larsen, K.G., Li, S., Nielsen, B.: Timed testing under partial observability. In: ICST. pp. 61–70. IEEE Computer Society (2009)
17. Ferrari, A., Grasso, D., Magnani, G., Fantechi, A., Tempestini, M.: The metrô rio atp case study. In: Kowalewski and Roveri [27], pp. 1–16
18. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A symbolic framework for model-based testing. In: Formal Approaches to Software Testing and Runtime Verification. LNCS, vol. 4262, pp. 40–54. Springer (2006)
19. Garavel, H., Viho, C., Zendri, M.: System design of a cc-numa multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. STTT 3(3), 314–331 (2001)
20. Garavel, H., et al.: Cadp 2006: A toolbox for the construction and analysis of distributed processes. In: CAV 2007. pp. 158–163 (2007)
21. GraphML work group: GraphML file format. http://graphml.graphdrawing.org/
22. Grieskamp, W., Qu, X., Wei, X., Kicillof, N., Cohen, M.B.: Interaction coverage meets path coverage by SMT constraint solving. In: TESTCOM 2009 and FATES 2009. LNCS, vol. 5826, pp. 97–112. Springer (2009)
23. Groote, J.F., et al.: The mCRL2 toolset. In: Proc. International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008) (2008)
24. Hansen, H.H., Ketema, J., Luttik, S.P., Mousavi, M.R., van de Pol, J.C.: Towards model checking executable uml specifications in mcrl2. Innovations in Systems and Software Engineering 6(1-2), 83–90 (March 2010)
25. Hartman, A.: Model based test generation tools survey. Tech. rep., Centre for Telematics and Information Technology, University of Twente (2009)
26. Hartman, A., Nagin, K.: The agedis tools for model based testing. SIGSOFT Softw. Eng. Notes 29, 129–132 (July 2004)
27. Kowalewski, S., Roveri, M. (eds.): FMICS 2010, LNCS, vol. 6371. Springer (2010)
28. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using uppaal: Status and future work. In: Perspectives of Model-Based Testing. Dagstuhl Seminar Proceedings, vol. 04371 (2004)
29. Leveson, N.G.: Experiences in designing and using formal specification languages for embedded control software. In: Lynch and Krogh [30], p. 3
30. Lynch, N.A., Krogh, B.H. (eds.): HSCC 2000, LNCS, vol. 1790. Springer (2000)
31. Rook, P.E.: Controlling software projects. IEE Software Engineering Journal 1(1), 7–16 (January 1986)
32. Timmer, M., Brinksma, E., Stoelinga, M.: Model-based testing. In: Software and Systems Safety: Specification and Verification. NATO Science for Peace and Security Series - D: Information and Communication Security, IOS Press (2011)
33. Tretmans, J., Brinksma, H.: TorX: Automated model-based testing. In: Hartman, A., Dussa-Ziegler, K. (eds.) First European Conference on Model-Driven Software Engineering, Nuremberg, Germany. pp. 31–43 (December 2003)
34. Veanes, M., et al.: Model-based testing of object-oriented reactive systems with spec explorer. In: Formal Methods and Testing, LNCS, vol. 4949, pp. 39–76. Springer (2008)