Exploration Testing

Juhana Helovuo
Tampere University of Technology, Software Systems Laboratory
P.O. Box 553, 33101 Tampere, Finland
tel: +358-3-365 3823 , fax: +358-3-365 2913
<juhe@cs.tut.fi>

Sari Leppänen
Nokia Research Center
P.O. Box 407, 00045 Nokia Group, Finland
<sari.leppanen@nokia.com>

Abstract

This paper describes a new way of testing reactive systems as investigated by the RATE-project at the Tampere University of Technology. We abandon the idea of systematically using a large library of predetermined test cases and instead use a "live" specification to generate test runs on-the-fly, as testing progresses. In order to do this, we assume that the behavior of the implementation under test is specified as a labelled transition system. This testing method is most applicable to testing concurrent, nondeterministic, and reactive behaviors rather than data-intensive computation.

1 Introduction

Traditionally, attempts to automate software testing are based on the idea of predetermined test cases: First a testing engineer creates a collection of test cases, ideally by deriving them from a specification. These test cases are essentially linear sequences of inputs and their expected outputs. Then the test engineer runs the software under test against these test cases: sends the listed inputs to the software and compares the outputs to the expected outputs as described in the test sequences.

This traditional method of testing works quite nicely with respect to traditional *transformational* programs, which can be seen as implementations of a (partial) function from the inputs to the outputs. A Turing machine models this type of a computation. There is a lot of literature on this type of testing, for example, [9] [8] [5] and more modern approaches, based on formal methods, are presented in [12] [13] [1].

However, there are problems with this approach. Many modern software systems are *reactive* in their nature, that is, they cannot be modelled as a transformation from inputs to outputs, but rather as a state-transition system, which consumes inputs and responds with outputs. Such systems exhibit *nondeterministic* behavior, i.e. for a sequence of inputs there can be several correct outcomes. A test method based on predetermined test cases cannot very easily accommodate for all of these, since the number of different legal output sequences is often very large.

In this paper we present a method, which we call *exploration testing*. Exploration testing does not use test cases at all. Instead it uses a behavioral specification of the implementation under test (IUT). The behavioral specification specifies a complete set of executions rather than one linear sequence of actions, i.e. one execution. Similar techniques have been proposed previously, for example Guided Random Walk [6] and On-the-fly testing [2].

Although the specification represents a more or less complete set of legal executions, it need not describe the behavior in too much detail. The specification may be abstracted to a rather high level so that it does not need to contain every detail about the low-level behavior of the IUT. It suffices that the test specification contains enough details, so that it can be used to generate reasonable test inputs for the IUT.

The rest of this paper is structured as follows. Section 2 describes our notion of a test specification. Section 3 develops the subject more formally and gives a test execution algorithm. We give some small examples in section 4 and a larger case study in section 5 and finally give a summary in

section 6.

2 The Test Specifications

We represent externally observable behavior as a labelled transition system (LTS), which closely resembles a finite automaton. However, an LTS has no concept of accepting states and theoretically an LTS need not be finite. It should be noted that an LTS can be *nondeterministic* and there can be *invisible actions*.

Since we are using the LTS formalism to represent behavior, we have to assume that the behavior of the IUT can be represented as an LTS. This is known as the *test hypothesis*. Note that we do not assume that we know much about the LTS representation of the IUT nor that we can actually construct such an LTS. We only assume that an LTS, which represents the behavior of the IUT, exists in a mathematical sense.

The behavioral specification (or in this case, the test specification) should be simple enough for humans to understand. The specification can be simplified by hiding irrelevant details:

- The specification need not pay any attention to the internal workings of the system, only to a relevant portion of the externally observable behavior.
- The granularity of observations can be coarsened by grouping several actions together. Coarsening the granularity of the system may change some important atomicity properties of actions and thus alter the behavior of the system in an important way. Therefore this technique must be used with care.
- The specification may be concerned with only one aspect of the system. One such sub-specification should be easier to understand than the whole system specification. The behavior of the whole system can then be composed from such simpler pieces.

2.1 Layers of abstraction

Since we formally view both the specification and the observed behavior of the actual system as LTSs, we can change our point of view and treat a specification as an implementation. The advantage here is that we can use the same testing technique to compare two specifications of the same system to each other. Namely, we can view a lower-level specification as an implementation of a higher-level specification and test the execution of the lower-lever specification against a high-level specification (Fig. 1).

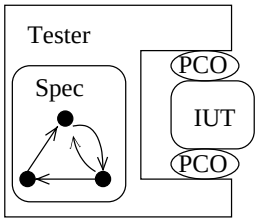

Figure 2. Testing environment

3 A Model for Exploration Testing

3.1 The Testing Environment and Specifications

The testing environment (Fig. 2) consists of the test subject, or *implementation under test* (IUT), the tester and two communication channels between them, one for input and the other for output. The communication channels are sometimes called *points of control and observation* (PCO) [4].

The input channel delivers *actions* (or events) from the tester to the IUT. We assume that the communication is synchronous, so there is no queue of input events. Some input events may sometimes be *refused* by the IUT. Refusals are used only when the physical interface being modelled is such that the receiver (or the interface) can block the sender from sending some inputs. For example, in a keyboard it is usually impossible for a key to be pressed down twice in a row without releasing it in between. Thus, we can say that a key refuses pressing actions if it has been pressed down but not released.

The output channel delivers externally observable actions of the IUT to the tester. The tester cannot refuse any outputs from the IUT.

3.2 The Specification

The test specification is a deterministic[1] LTS with the following properties:

- The specification has some set S of states, including an initial state $\hat{s} \in S$.
- For each input and output event there is a symbol (*action label*), which is transmitted through a PCO. Each specification has a set of input symbols and a set of output symbols, denoted with Σ_I and Σ_O, respectively. $\Sigma_I \cap \Sigma_O = \emptyset$.
- The specification also contains a set of *input refusal actions*, $\Sigma_{\overline{I}} = \left\{ \overline{x} \;\middle|\; x \in \Sigma_I \right\}$. An action $\overline{x}$ means that

[1] Only the specification is required to be deterministic. The IUT need not be deterministic.

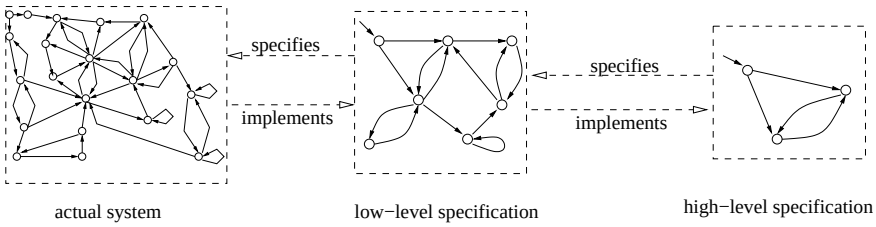

Figure 1. A hierarchy of specifications

the IUT refuses to accept input x. These are necessary for modelling interfaces which can observably refuse an input action.

- The full alphabet known by the specification consists of the input, output and input refusal alphabets and three special symbols: $\Sigma_{\text{SPEC}} = \Sigma_I \cup \Sigma_O \cup \Sigma_{\overline{I}} \cup \{\delta, \rho, \tau\}$. The implementation does not actually execute input refusal symbols. In fact, an input refusal does not change the internal state of the IUT at all. Only the specification may change its state because of an input refusal. If an input refusal changes the state of the IUT in any way, then it should not be modelled as an input refusal.

- The special symbol τ denotes an invisible, internal action, which may be executed at any time it is enabled. τ is never sent over a communication channel or a PCO.

- The symbol ρ is a special input action, which causes the IUT to be reset and returned to its initial state. The tester sends a ρ-action to start a new test run.

- The symbol δ is an artificial output action, which means *quiescence* [13], or that no output was observed. The interface between the tester algorithm and the IUT uses this action as an indication that the IUT has stopped and will send no further output. In simulated situations a quiescence is easy enough to detect, but in real applications this situation may be detected only by using timers or other special instrumentation to observe when the IUT has really stopped and does not produce any further outputs.

- The transitions of the specification are a set Δ of triples $\Delta \subseteq (S \times \Sigma_{\text{SPEC}} \times S)$.

3.3 The Implementation Under Test

We assume that the implementation under test (IUT) is a black box with an LTS-like-behavior, which we can observe in test runs. Since this is testing and not model-checking, we do not assume that we can see the internal workings or structure of the IUT. The IUT can be a simulator executing a model or an actual working implementation of some system, provided that there is a suitable interface to the tester. In our experiments we have used a simulator executing LTS models compiled from SDL specifications.

Specifically, we make the following assumptions:

- We can communicate with the IUT via input and output actions. Every action is either an input or an output. We cannot see what the IUT is about to do next or what input actions it is prepared for. Although sometimes the IUT may refuse an input, we cannot know that in advance without actually trying to send the input.

  This method of communication should not be confused with synchronous parallel composition used in verification.

- There is a special input symbol ρ, which can be used to reset the IUT to its initial state at any time.

- We can detect when the IUT has stopped and is not going to output anything without further stimulus (i.e. inputs). This situation is modelled by the artificial output action δ, which is produced by the tester when needed.

3.4 A Test Execution Algorithm

In this section we outline an algorithm, which the tester uses to test the IUT against a specification LTS.

Figure 3 presents the steps of the algorithm as a state transition diagram. The tester keeps track of the current state of the specification during the execution of the tests. A test step is executed as follows: First the tester inspects the specification to see what kinds of transitions are possible at the current state.

If only output transitions are possible in the specification (the second leftmost branch of Figure 3), we continue to receive output from the output channel. (In this context δ counts as output.) The interface to the IUT is such that the tester is guaranteed to receive something, either a real event or a δ-event. After receiving the output there are three possibilities. If the output is something the specification is not prepared for, we declare an illegal trace, or in case of unspecified δ, an illegal output refusal. Otherwise the output

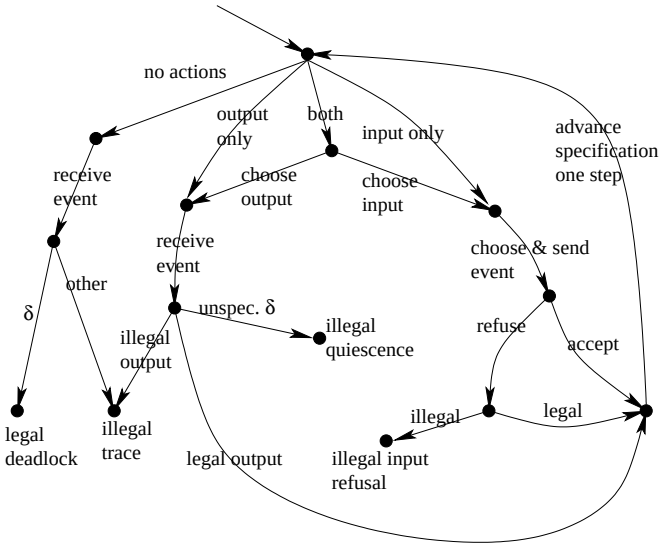

Figure 3. The exploration testing algorithm

is legal (possibly δ) and we take that transition and start the next testing step.

If only inputs are possible in the specification, the tester chooses one of these input actions and tries to send it to the IUT. If the IUT accepts the event, then we proceed to the next testing step as before. If the IUT refuses the event and there is no possibility of a refusal action in the specification, then we declare an illegal input refusal.

If there are no possible actions in the current state of the specification, the only acceptable output is δ.

Each test run is initialized by sending the ρ-action (reset) to the IUT and setting the current state of the specification to be the initial state $\hat{s}$.

3.5 Semantics of Tests

The test execution algorithm outlined in the previous subsection tests for three kinds of requirements. In other words, the test algorithm can detect three kinds of faults in the behavior of the IUT:

- **Illegal traces**. Any execution trace, which does not appear in the specification is considered illegal.
- **Illegal output failures** (quiescence). An absence of response from the IUT when the specification states that there should be some response.
- **Illegal input failure** (input refusal). A condition where an input could not be fed into the IUT even though the specification does not allow a refusal to occur. This can happen only with interfaces which can actually refuse an incoming input.

These concepts can be formalized, but first we need some notation:

Definition 1 (Arrow notation) *The following are more convenient notations for executions of an LTS.*

- $s -a\rightarrow s'$ *iff* $(s, a, s') \in \Delta$
- $s -a_1 a_2 \ldots a_n\rightarrow s'$ *iff* $\exists s_0, s_1, \ldots, s_n : s = s_0 \wedge s_0 -a_1\rightarrow s_1 -a_2\rightarrow \cdots - a_n\rightarrow s_n \wedge s' = s_n$.
- *If* $\sigma = a_1 a_2 \ldots a_n$ *then* $s -a_1 a_2 \ldots a_n\rightarrow s'$ *can be written* $s -\sigma\rightarrow s'$.
- $s -\sigma\rightarrow$ *iff* $\exists s' : s -\sigma\rightarrow s'$
- $s =a_1 a_2 \ldots a_n\Rightarrow s'$ *iff* $s -\tau^* a_1 \tau^* a_2 \tau^* \ldots \tau^* a_n \tau^*\rightarrow s'$ *, where each* τ^* *denotes any sequence of zero or more* τ*-actions, and none of* a_i *is* τ.
- $s =\sigma\Rightarrow$ *iff* $\exists s' : s =\sigma\Rightarrow s'$

Now we can give a more precise definition of traces and input and output failures.

Definition 2 (Traces) *The set of* traces *of an LTS* P *is* $tr(P) = \left\{ \sigma \in (\Sigma_I \cup \Sigma_{\overline{I}} \cup \Sigma_O \cup \{\delta\})^* \;\middle|\; \hat{s} =\sigma\Rightarrow \right\}$

Definition 3 (Output Failures) *The set of* output failures *of an LTS P is the set of traces, which end in* δ*:* $ofail(P) = \left\{ \sigma \in tr(P) \;\middle|\; \exists \sigma_1 \in tr(P) : \sigma = \sigma_1 \delta \right\}$.

Definition 4 (Input Failures) *The set of* input failures *of an LTS P is the set of pairs* $ifail(P) = \{ (\sigma, R) \in tr(P) \times 2^{\Sigma_I} \mid \exists s : \hat{s} = \sigma \Rightarrow s \wedge \forall a \in R : \neg(s = a \Rightarrow) \}$. *The set* R *is called an* input refusal set.

These three sets form a *semantic model* for this testing algorithm. They capture all the necessary information to determine whether errors (according to this algorithm) will be found or not.

We have not specified the structure of the implementation I, but since we assume it behaves in experiments like some LTS, we can observe single execution traces, output failures and input failures.

The testing algorithm presented in the previous subsection will not find an error in implementation I with respect to the specification S, if the following conditions hold and the sets of known input and output events are the same.

$$\begin{cases} tr(I) \subseteq tr(S) \\ ofail(I) \subseteq ofail(S) \\ ifail(I) \subseteq ifail(S) \end{cases}$$

This claim should be easy to check informally by comparing the definition to the testing algorithm. The algorithm can discover three kinds of errors: illegal traces, illegal input refusals and illegal output refusals (quiescences), which correspond to the three subset relations given above. Since the testing algorithm and the implementation are nondeterministic, we cannot guarantee that any specific error will be found.

In a practical implementation the IUT may contain much more details than the specification, which could have just a few high-level actions to keep the test running. In this case the tester should just ignore any actions, which are not included in the specification at all:

First, we define an abstracted version I_{A} of the implementation I as follows. The abstracted version hides those actions which are unknown to the specification, i.e. do not belong to its alphabets.

$$(\Sigma_I \cup \Sigma_O) \subseteq \Sigma_{\text{IMPL}}$$

$$I_{\text{A}} = \textbf{hide } (\Sigma_{\text{IMPL}} - (\Sigma_I \cup \Sigma_O)) \textbf{ in } I$$

where the expression "**hide** A **in** P" means converting action labels in LTS P into τ if they appear in the set A.

Then we can compare I_{A} to S as before.

$$\begin{cases} tr(I_{\text{A}}) \subseteq tr(S) \\ ofail(I_{\text{A}}) \subseteq ofail(S) \\ ifail(I_{\text{A}}) \subseteq ifail(S). \end{cases}$$

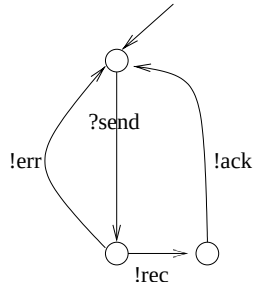

Figure 4. A small specification example

4 Small Examples

This section presents some small usage examples of the exploration testing method.

The first example is a model of a very simple communication protocol. It consists of three states and has four possible actions: ?send, !err, !ack and !rec as shown in Figure 4.

We see the behavior of the protocol from the viewpoint of the protocol user. The transmitting end issues the command "?send" and depending on the success of the transfer, the receiving end will see an output "!rec" at the other end of the channel or nothing at all. Accordingly, the sender will get either an acknowledgment message or an error indication. Every visible action has been designated either as an input(?) or output(!) to/from the protocol model. This prevents the environment from influencing the choice between "!err" and "!rec".

When this protocol is tested against itself, the testing algorithm can be executed indefinitely without finding any errors. Figure 5 describes three different implementations of the specification.

The leftmost model adds the possibility of deadlocking after acknowledging a transmission. The testing algorithm can detect this as an input refusal, since the next action after the deadlock should be "?send". In the current version of the test execution and simulation software this error will be detected typically after executing about ten actions.

The second model has a similar deadlocking state, but in this case it is harder to reach randomly, since entering the deadlock requires two consecutive "wrong" nondeterministic choices by the implementation model. The deadlocking state is detected as an output refusal after about 20 actions, on the average.

The third model represents the behavior of an ideal situation. The transmission will always succeed and there is no possibility of outputting "!err". The testing algorithm cannot find any errors, since there are no unspecified traces, output or input failures. However, it is impossible to get very high test coverage with respect to the specification,

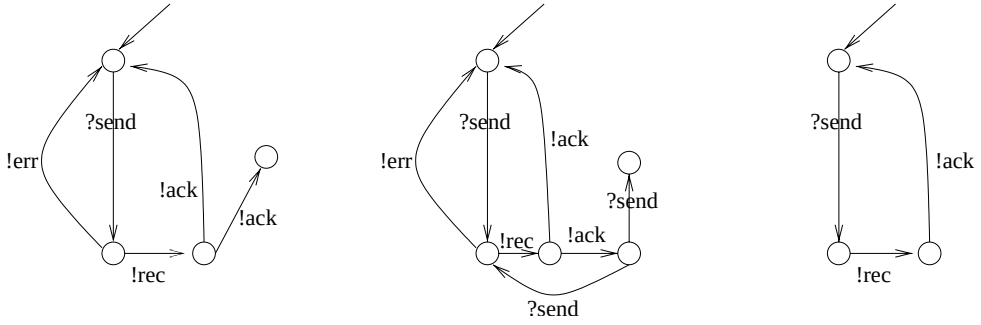

Figure 5. Example implementations

since one out of four transitions is never exercised.

5 The Radio Link Control Protocol case

UMTS (Universal Mobile Telecommunication System) is a third generation mobile telecommunication system using *WCDMA (Wideband Code Division Multiple Access)* radio access technique. The new radio access technique requires major changes in the radio access network, consisting of the network elements and protocols participating to data transmission in the radio interface. *RLC (Radio Link Control)* protocol is one of the new UMTS protocols. It is a data link layer protocol, according to the OSI reference model, providing reliable data transmission service for the upper layers over the unreliable radio interface. It uses the unreliable data transmission service provided by the subjacent *MAC (Medium Access Control)* protocol (see Figure 6).

RLC protocol was standardized in March 2000 by *3GPP*, an international standardization forum consisting of manufacturers, operators, authorities etc. interested in regulation and development of the third generation systems. The specification [11] defines several services, functions and procedures for the protocol. The RLC protocol model used here is an augmented version of the simple model described in [7].

The main task of RLC protocol is to provide data transfer service to upper layers. The protocol specification defines three different types for data transfer service: transparent, unacknowledged and acknowledged data transfer. From the analysis and testing point of view the most complex and thus the most interesting one is data transfer in acknowledged mode. The specification defines several functions that are needed to support acknowledged data transfer. In order to keep the size of the model manageable and to restrict the analysis to the elementary data communication functionality, we had to consider each of these functions carefully and do as much abstraction on them as possible. For more detailed description of the functionality of the RLC protocol see for example [3].

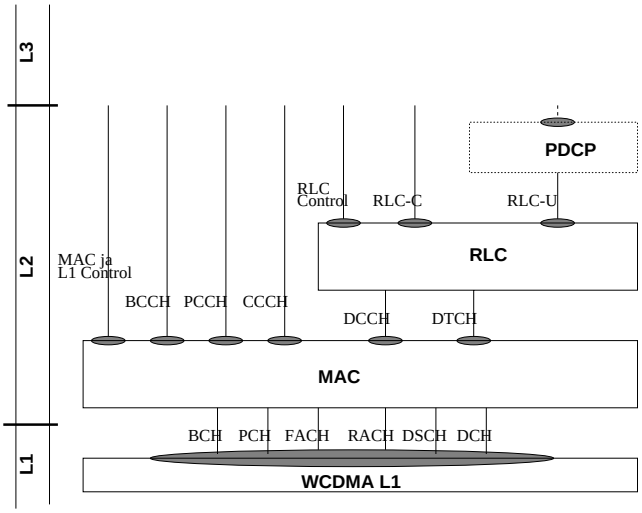

Figure 6. UMTS data link layer over the radio interface

5.1 Abstractions in the Model

We include into our model the transfer of user data with flow control, the in-sequence delivery of higher layer *PDUs (Protocol Data Unit)* and duplicate detection. These functions are defined in the RLC protocol specification [11] and they are essential for establishing the reliable data transmission. The size of the user data, i.e. the size of a RLC *SDU (Service Data Unit)*, is assumed to be exactly the same as the size of the data field in a RLC PDU. Thus the segmentation, reassembly, concatenation and padding functionalities can be left out.

For the radio interface we will use a *lossy channel*, which makes nondeterministic choices for either delivering the RLC PDU to the receiving side or losing it. However, the channel does not duplicate or corrupt the packets, so we have not included the error correction functionality into our model. Ciphering is not yet defined precisely in the specification, so we leave it out from our model. For detecting and recovering from protocol errors we include the *reset procedure* described in the specification.

Besides the abstractions on the functionality, we also make abstractions on data types. We abstract the user data, the type of SDUs, using Wolper's *Data Independence Principle* presented in [14]. We want to test the reliable transfer of user data, especially the in-sequence-delivery of higher layer PDUs and duplicate detection. According to Wolper, two separate types for user data are enough for this. For flow control, we will use the *sliding window mechanism*. Because we limit the size of the transmitting and receiving window to one, only two separate sequence numbers are needed for PDUs. Now we can define an enumerated data type with two items, 0 and 1, and use this both for the user data and sequence numbering.

The specification defines some parameters for the configuration message used by the upper layer to establish and release a RLC connection. Parameters are used to configure the RLC protocol entity to the appropriate mode and to define parameter values used in ciphering and segmentation. Because we have only one functional mode in our model and no ciphering or segmentation at all, the parameters are left out from configuration messages.

5.1.1 Functional description

In our model the connection establishment phase consists of receiving a single connection establishment message from the upper layer. The UMTS data link layer over the radio interface has two sublayers: RLC and MAC. RLC layer provides a radio solution-dependent reliable link for the user and MAC layer controls, among other things, the access signaling procedures (request and grant) for the radio channel [10]. Together with an upper layer protocol, *RRC (Radio Resource Control)*, MAC establishes the physical radio link. After this is done, RRC sends a configuration message to RLC layer to establish the RLC connection [11]. We assume that the message is received at the same time at both ends of the protocol.

The data transfer procedure is initialized when an SDU is received from the upper layer. For each SDU, the RLC protocol entity creates a corresponding PDU. The SDU is placed into the data field of the PDU as such, and the appropriate sequence number is placed into the PDU header. A timer for the PDU transmission is set right after sending the PDU to the channel, which is modeling here the lower layers and the radio interface. No further requests are accepted from the user before the data transfer procedure for the previous one is *terminated*. Data transfer procedure terminates either when the transmitting side receives an acknowledgment for the PDU or, in an abnormal case, after sending a notification of a protocol error to the user. In the normal case the transmitter receives the acknowledgment for the PDU before the maximum amount of resendings is exceeded. Resending is triggered, as usually, by the PDU transmission timer expiration. The PDU transmission timer is reset when the acknowledgment with the appropriate sequence number is received. Acknowledgments with other sequence numbers are ignored.

After receiving a PDU, the RLC protocol entity in the receiving side removes the header and delivers the SDU to the upper layer. After sending the corresponding acknowledgment to the channel, it updates the sequence number counter. The transmitting side updates the sequence number counter after receiving the acknowledgment. In our model, constructed according to a very early specification, the delivery of the SDU was not confirmed to the user in the transmitting side. In an abnormal case where the PDU transmission timer expires, and the predefined maximum amount of retransmissions for the PDU is full, the reset procedure is executed. The purpose is to resynchronize the data transmission and bring the protocol back into the consistent state. The transmitting side sends the reset message and sets a timer. The receiving side acknowledges the message and updates the sequence number to a predefined initial value. In the transmitting side the sequence number is updated to the same initial value right after receiving the acknowledgment for the reset message. Data transfer continues with the next SDU received from the user. A maximum number of resendings is defined for reset messages also. According to the specification, the RLC protocol does not notify the user of recoverable protocol errors, i.e. when the reset procedure is executed successfully. So, initially also in our model the notification was given only when the timer expired after resending the maximal amount of reset messages.

Correspondingly to the connection establishment scenario, also the disconnection phase consists of receiving a

single disconnecting message from the upper layer. We assume it to be received simultaneously on both transmitting and receiving side. Both protocol entities return directly to the initial state to wait for a new connection establishment.

5.2 Errors Found in Testing

We used as an IUT an *SDL (Specification and Description Language)* implementation with the functionality described in the previous section. We defined the desired external behavior of the protocol with a small, high-level LTS specification. In the beginning there were only 4 states and 8 transitions in the specification. As the testing proceeded, also the specification evolved to be more extensive and precise. We found various types of errors: errors and deficiencies in the high-level specification, errors in the implementation model, faults in the testing strategy and incorrect assumptions made on the behavior of the protocol environment. In the following, we will give some examples of different error types.

The processes of the original SDL implementation were first converted semi-automatically into separate LTSs, using a self-written conversion program and manual checking of the result. The elementary conversion process and the basic parallel composition arrangement for the processes are described in [7]. In this phase we found couple of conversion errors caused by the incomplete handling of some SDL language features in the conversion program. Errors were corrected into the resulting LTS manually.

5.2.1 Incorrect modeling of the environment

The first error we found was caused by unrealistic behaviour of the model. For the media, modeling the radio interface, we used two unidirectional channel components. We described them as LTS processes and composed the data channel in parallel with the transmitting process and the acknowledgment channel with the receiving process. (See [7]).

According to the RLC protocol specification, the RLC connection is released when the protocol receives a disconnecting message from the user, i.e. from the upper layer. We can assume, that when a virtual radio link is released and then soon afterwards a new one is established, the channel is empty for the new connection. Thus no old PDUs addressed for the old released process are routed to the newly created one. This is because in the implementations process deletion and creation are dynamic actions and collisions between the process ID's are quite rare. Also the routing of messages in software systems is usually based on process ID's. In our model the channel components were not synchronizing with the disconnecting messages sent by the user and separate "channel release"messages were not generated either. So, when the RLC connection was released and the transmitting and receiving processes were killed, the underlying media tried to go on without knowing about the interruption. This caused several error cases with the routing of messages and the synchronizing actions. We extended the disconnecting message to concern also the channel components. We inserted the disconnecting messages to the set of the synchronizing actions for both of the channel components and also the necessary transitions to the LTSs to create the desired functionality. As a result the size of the state space of the protocol LTS was decreased remarkably.

5.2.2 Faulty testing strategy

Another error situation occurred due to a faulty testing strategy. After giving an input the tester was waiting for all possible outputs that the IUT was able to generate. As a consequence, every time also all of the timer expirations were enabled and thus no data was going through the protocol. We changed the testing strategy so that the choice over giving a new input to the IUT or waiting more outputs from it is done nondeterministically and using carefully chosen probabilities.

5.2.3 Errors in the implementation model

We also found some functional errors in our model, which is one possible implementation from the standardized protocol specification. According to the specification, *data confirmation primitives* for the user are optional. The user defines, when giving the SDU to be delivered, whether this SDU should be confirmed or not. We did not include the confirmation mechanism into our first model. However, when the transmitting window was full, our protocol refused to accept any new data from the user. No buffering of SDUs or specific "SDU refusal"messages for the user were included into our model. These functional properties together led to a situation where the tester should have been able to guess the exactly right moment to send more data to the protocol. As we want to assume the upper layer to be as simple as possible, we inserted the data confirmation primitives to the model.

Now we met another problem: Data confirmation primitives are good for acknowledging the successfully delivered SDUs, but the user should also be informed when the protocol fails to deliver an SDU over the radio interface. Since the only parameter defined for the data confirmation primitive is the ID for the SDU, the negative acknowledgments shall be passed to the user with some other primitive. We used the status primitive for this. The specification defines one parameter for status primitive, the event code, which we can then enlarge by rich typing. We used the status primitive also for informing the user about unrecoverable protocol errors, such as data link loss, and for recoverable

protocol errors, such as successfully executed reset procedure. Status primitive can also be used to optimize the usage of radio resources by informing the upper layers about an unused radio link. We implemented this in our model with a timer for receiving any data in the receiving side.

5.2.4 Errors in the high-level specification

We started the testing with "first-thought"high-level specification consisting of 4 states and 8 transitions. The specification described mainly only the basic data transfer scenarios and some trivial error cases that can be easily envisaged. As the testing proceeded, we continually run into situations where the errors found reflected deficiencies in the specification. We found several cases, especially error cases, for which the specification did not define the desired behaviour at all. We corrected and augmented the specification and as a result the specification evolved to be more extensive and precise. At the moment the state space of the high-level specification is more than two-fold compared to the "first-thought"specification.

The testing is not yet complete, since even this kind of a simple model of the protocol is complex. Also, the specification intentionally leaves several possible ways of implementing the protocol open. Although this is necessary to enable the competition on the markets, it makes it very difficult to find errors in the protocol specification. In many cases it is a matter of interpretation whether the errors found in protocol testing are model specific or in the actual specification. Especially the deficiencies and inadequacies in the protocol specification are difficult to discern and to justify in contributions. Usually they are considered just to be implementation specific or protocol stack internal matters.

6 Summary and Conclusions

We have developed a new testing method, which lies between conventional testing and verification by model-checking. We call this method *exploration testing*. This method is relatively easy to use and understand, more comparable to testing than formal verification.

Exploration testing does not have the thoroughness of model-checking, but it tends to test for a large number of properties instead of some formally specified safety and/or liveness formulas, as in conventional verification. The method does give some indication of test coverage, but it is not very easy to guarantee that some particular special case has been covered.

Exploration testing is also applicable to partial system models at an early stage of development as well as a final implementation. According to our experience, this method also supports incremental development and debugging quite nicely. The main application area we have been experimenting with is telecommunication protocols. We presented an extensive case study where we recover several different types of errors in a mobile telecommunication protocol.

Acknowledgments

This research is part of the RATE-project, which studies automated testing of reactive systems. The work was funded by the National Technology Agency of Finland (Tekes) and also by Nokia Research Center. The authors wish to thank professor Antti Valmari for encouragement and ideas.

References

[1] Jari Arkko. *Conformance Test Generation from Nondeterminis tic Specifications*. Licenciate thesis, Helsinki University of Technology, 1995.

[2] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):382–393, 2000.

[3] H. Holma and A. Toskala. *WCDMA for UMTS, Radio Access For Third Generation Mobile Communications*. Wiley & Sons Ltd, England, 2000.

[4] Conformance testing methodology and framework – part 3: The tree and tabular combined notation (TTCN). International Standard ISO/IEC 9646-3, International Standards Organization, 1998.

[5] Paul Jorgensen. *Software Testing: A Craftsman's approach*. CRC Press, 1995.

[6] David Lee, Krishan K. Sabnani, David M. Kristol, and Sanjoy Paul. Conformance testing of protocols specified as communicating finite state machines — a guided random walk based approach. *IEEE Transactions on Communications*, 44(5), 1996.

[7] Sari Leppänen and Matti Luukkainen. Compositional verification of a third generation mobile communication protocol. In *Proc. International Workshop on Distributed System Validation and Verification*. IEEE, 2000.

[8] Brian Marick. *The Craft of Software Testing : Subsystem Testing including Object-based and Object-oriented testing*. Prentice-Hall, 1995.

[9] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[10] T. Ojanperä and R Prasad. *Wideband CDMA for Third Generation Mobile Communications*. Artech House Publishers, 1998.

[11] 3rd Generation Partnership Project. RLC protocol specification. Technical Specification 3G TS 25.322 V3.2.0, Technical Specification Group Radio Access Network, 2000.

[12] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, December 1992.

[13] J. Tretmans. Test generation with inputs, output and repetitive quiescence. *Software – Concepts and Tools*, 17:103–120, 1996. Also published as CTIT Technical report 96-23, University of Twente.

[14] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–193. ACM, ACM, January 1986.