AN AUTOMATED, FLEXIBLE TESTING ENVIRONMENT FOR UMTS

Jan Bredereke, Bernd-Holger Schlingloff
Universität Bremen, TZi · P.O. box 330 440 · D-28334 Bremen · Germany
{brederek,hs}@tzi.de · www.tzi.de/{~brederek,~hs}

Abstract We describe an automated, flexible testing environment for UMTS systems which are specified and developed with SDL. For testing these systems, we use a testing tool which generates black box tests from a formal specification of the desired properties. Since the requirements are subject to considerable change at any time, it is important to guarantee consistency between the interfaces of the test specification and the system under test. Therefore, we defined rules for the modularization of the requirements according to functional properties. Furthermore, we devised and implemented a generator tool which automatically produces all necessary interface code. This is a first step towards a general configuration utility for the automatic creation of runtime interfaces and adapters. We report on testing results and experiences with our setup.

Keywords: test tools; test interface generation; industrial testing experience; test selection and management; UMTS; SDL; conformance testing.

1. INTRODUCTION

UMTS (Universal Mobile Telecommunication System) is an evolving standard for a new generation of high-speed, multi-media mobile phone systems. It is being defined by the 3GPP consortium (the 3rd Generation Partnership Project [1]), and currently the equipment is being developed at different sites. Even though the standard has been released already in 1999, there have been a number of significant changes and additions, and further have to be expected. In a black box testing approach, all interfaces and parameters have to be provided to the testing environment in detail. Therefore, it is important that a testing environment for UMTS software is flexible and easily adaptable. If the interfaces and requirements are modified, the necessary changes in the testing specification and environment should be as small as possible.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35497-2_31
I. Schieferdecker et al. (eds.), *Testing of Communicating Systems XIV*

In this paper, we describe an automated, flexible testing environment for UMTS systems which are specified and developed with SDL (Specification and Description Language, see [2]). For testing the SDL system, we use a tool in which test suites can be described in the formal specification language CSP (Communicating Sequential Processes, see [3]). We developed a generator tool which automatically produces all necessary interface code to connect the SDL development suite and the CSP testing tool. Since the generation is completely automated, consistency between the interfaces is guaranteed.

This work evolved from testing needs in the development of the Radio Link Control (RLC) protocol layer of UMTS. It was done in cooperation with Siemens AG, Salzgitter. For this system, it was decided to apply specification based testing (black-box-testing) rather than structural testing, for the following reasons. In the case of UMTS, user equipment and base stations are developed by different companies. Moreover, even the development of the software for different layers of the protocol is distributed between different sites within Siemens AG. For the correct functioning of the whole system, it is extremely important that the standard is implemented by all participating developers in a consistent way. Therefore, in order to ensure inter-operability between devices from different providers, it is mandatory to base test suites solely on the 3GPP standards (plus additional site-specific requirements) rather than on individual program code from specific developers.

In specification-based testing, the requirements are described separately from the implementation in a formal language. Thus, a need to interface the testing specification and the implementation arises. As mentioned above, in the case of UMTS the interfaces of specification and implementation are subject to change; all changes must be carried out consistently. For example, in release 4 of the RLC layer standard, for the primitive `CRLC_CONFIG_Req` additional parameters `Stop` and `Continue` have been added. They indicate that the RLC entity shall not transmit or receive RLC PDUs or shall continue transmission and reception, respectively. If these additional parameters are taken into account either in the implementation or in the test case specification, it must be ensured that these still fit together. Additionally, the interface code between the implementation and the testing environment must be re-generated. The `CRLC_CONFIG_Req` primitive has hundreds of parameters, with more than one kilobyte of heavily structured data. Thus, without automated support, these tasks are tedious and error-prone. Especially, if the test specification and the implementation are developed by separate teams, automated consistency checks and code generation are extremely useful. We therefore devised a tool which allows to perform these tasks with

minimal manual efforts. Subsequently, we describe its design and underlying principles. Details of the interface generation part of our tool can be found in a companion paper ([4]).

This paper is organized as follows: in Section 2, we briefly introduce our specification language and testing tool. Section 3 is the main part, it presents the principles of the automated, flexible testing environment. This environment generates the interface to the implementation automatically while also automatically checking the consistency of the implementation's and the specification's interface. We present rules for the modularization of the requirements which prepare for a flexible maintenance of the initial test specification and which reduce the work necessary for modifications. In Section 4, we report on the application of the environment to a part of the UMTS protocol stack; and we describe and interpret the testing results. Section 5 summarizes our work.

2. FORMAL CSP SPECIFICATIONS AND RT-TESTER

CSP (Communicating Sequential Processes, see [3]) is a specification language which allows to give a description of a system on a high level of abstraction. The structure of the requirements is reflected by particular operators such as sequential or parallel composition, choice, iteration and hiding. Communication between the processes and with the outside is by the exchange of events. In contrast to SDL, however, this communication is synchronous (handshake); buffered communication has to be modelled explicitly. We use a timed version of CSP, where it is possible to set timers which generate events upon elapse. This way, it is possible to test real-time behaviour of applications, which is especially important for embedded systems.

Since CSP arose from theoretical considerations, it has a well-defined formal semantics and a rich theory. Several elaborate verification tools have been developed for CSP, most notably the FDR system (**F**ailure **D**ivergence **R**efinement) by Formal Systems, Ltd. At present, FDR forms a basis for our testing system RT-TESTER.

Compared to other specification languages such as TTCN-3 [5], the parallel composition is treated differently in the test case generation from CSP specifications. The testing system first constructs a *transition graph* which is a representation of all possible interleavings of the parallel tasks. The test driver then enumerates all paths through this graph and uses them as test sequences for the SUT. This way it is possible to specify separate requirements on the same interfaces as independent parallel testing tasks, which allows a modular description of testing objectives.

Example 1 A vending machine specification featuring a few basic CSP operators.

```
include "timers.csp"
pragma AM_INPUT
channel coin, buttonCoffee, buttonTea
nametype MonEv = { coin, buttonCoffee, buttonTea }
pragma AM_OUTPUT
channel coffee, tea
nametype CtrlEv = { coffee, tea }

OBSERVER = (  (coin -> HAVE_COIN)
           [] (buttonCoffee -> OBSERVER)
           [] (buttonTea -> OBSERVER))
HAVE_COIN = (  (coin -> HAVE_COIN))
            [] (buttonTea -> AWAIT({tea}) ; OBSERVER)
            [] (buttonCoffee -> AWAIT({coffee}); OBSERVER)

RANDOM_STIMULI = (|~| x: MonEv @ x -> PAUSE; RANDOM_STIMULI)

TEST_SPEC = RANDOM_STIMULI [| MonEv |] OBSERVER
```

A system described by the process `OBSERVER` accepts either of the three inputs listed (external choice "`[]`"). If the input is a `coin`, then the system behaves like the process `HAVE_COIN` (event prefix "`->`"). A system described by the process `HAVE_COIN` outputs the desired drink after the corresponding button press. In this, the subprocess `AWAIT` waits for any of the outputs specified (sequential process composition "`;`"). The definition of this process is listed in Example 2 on page 88 below. The process `RANDOM_STIMULI` non-deterministically selects one event from the set `MonEv` (replicated internal choice "`|~| x : S @ P(x)`"), waits a short time, and starts all over (recursion). The process `TEST_SPEC` describes the complete test suite: the process `RANDOM_STIMULI` provides all the test inputs, which are also tracked by the process `OBSERVER`. The latter additionally tracks the test outputs. They are combined by sharing ("`P [| S |] Q`") the input events in the set `MonEv`.

Example 1 introduces a few basic operators of CSP, in particular the event prefix, the external choice, and one of the parallel composition operators.

From formal CSP test specifications, test cases can be generated and executed on-the-fly. Our tool RT-TESTER [6] reads the CSP input and uses FDR to generate the transition graph from it. In a separate stage, this transition graph is used by RT-TESTER to generate test scripts, which are executed on a separate testing machine automatically and in real time. They may run over long periods of time: hours, days, weeks and more – without the necessity of manually writing test scripts of an according length. The testing machine and the SUT (system under test) communicate via TCP/IP sockets, and test results are evaluated on the

fly in real time during the run of the SUT, by using the compiled CSP description as a test oracle. To ensure that the tests cover the whole bandwidth of all possible system situations, a mathematically proven testing strategy is used [7].

3. AN AUTOMATED, FLEXIBLE TESTING ENVIRONMENT FOR SDL WITH CSP

The 3GPP standard is written in a mixture of formalisms. The main part is plain English and annotated figures. These are accompanied by tables describing bit-level data formats, small state transition diagrams for the different modes and control commands, plus message sequence charts for the procedural communication between sending and receiving protocol instances. Earlier versions of the standard were accompanied by a detailed implementation suggestion described in the specification and description language SDL [2].

The implementation is developed at various sites from these and similar sources with the help of suitable tools. In particular, there are commercial tools which can execute an SDL system in an emulated runtime environment, and which can compile a set of SDL diagrams plus a set of data type descriptions written in ASN.1 or as C headers into executable machine code for embedded targets.

A particular problem for the development of test suites for UMTS is that the standard describing the requirements still is subject to considerable change. Even after the "December 1999" release, which was supposed to be stable, a large number of changes were made, and more have to be expected. This concerns, for example, the parameters of the service primitives and the details of the data at the interfaces, such as the structure of the protocol data units. Even the behaviour of the protocol machines is still expected to change, in particular for error handling. Similarly, not all implementation decisions have been finalized, some details of the machine representation of data at the interfaces are not yet fixed. We therefore designed the testing environment to be highly flexible by

- defining the interface in terms of SDL signals and data structures instead of low level descriptions,
- performing an automated consistency check between the SDL description of the interface and the formal CSP specification of the interface, and
- defining rules for modularizing the formal behaviour specifications into largely independent functional requirements.

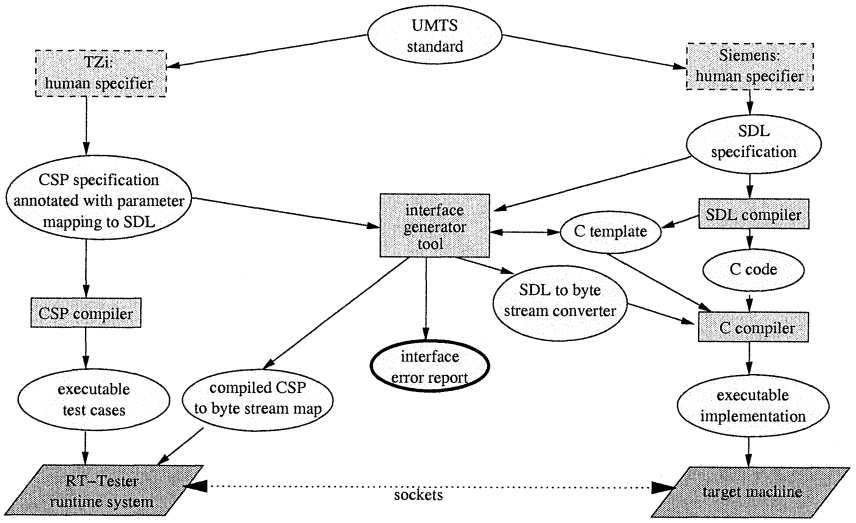

Figure 1. Automatic interface generation

3.1 Interfacing SDL and CSP

The interfaces of the UMTS protocol layers relevant for testing are specified in SDL. Our automated approach makes them the only relevant interfaces. The goal of the tests is to ensure that we can combine the tested SDL processes into a larger SDL system and achieve the desired behaviour. The actual representation of the internal interfaces between the components is not part of the visible behaviour of the combined system.

Only these internal interfaces have to be consistent. On one hand, the SDL and C compilers do this type check. On the other hand, RT-TESTER can check whether the intended inter-process cooperation indeed occurs by performing (black box) integration tests of several components.

An interface in terms of SDL signals must be mapped to an interface in terms of RT-TESTER's native CSP channels. Therefore, a translation between the two forms of syntax is needed. Due to its changing nature, we decided to automate the mapping by a generator tool. This generator tool also flags any inconsistencies between the interfaces of the two specifications. If the SDL specification is changed, the generator tool simply needs to be re-run. If it does not flag any error, the interface descriptions of the SDL specification and the CSP specification are guaranteed to be consistent. Consistency of the behaviour can then be checked in a subsequent test run. If the SDL interface has been changed

in a part that is relevant to the CSP tests, all items which do not match are logged and the problem can directly be investigated. In the case of the the RLC layer of the UMTS protocol stack, this feature is especially important. This module uses large and complex signal parameters, which are difficult to keep in sync manually. There are signals with more than one kilobyte of heavily structured parameter data; comparing their definitions manually would be extremely tedious and error-prone.

Figure 1 on the facing page presents the concept of the automated interfacing. Detailled information on the automatic generation of the interfacing software can be found in a companion paper; see [4].

Our automated approach for mapping signals obsoletes the need for a manual description of the machine representations, and it does not demand any user interaction after a change. Since we confine the machine representation of the data in the target entirely to that machine, it is sufficient to rebuild the generated files and recompile them. The socket communication with RT-TESTER uses our own byte string format which is independent of any machine representation.

3.2 Flexible Maintenance of CSP Specification

Usually, all test cases have to be revised and reorganized a number of times in the software development process. It is advisable to consider the maintenance tasks associated with a test suite already during its design. With a suitable structuring, the efforts for rewriting and recombination of test cases can be significantly reduced.

In our application, we identified two such maintenance tasks: we had to expect considerable *changes to the requirements* of UMTS, as discussed in the introduction of this paper. And we had to support *variants of our test suites* using as little duplication of description as possible. Test suite variants result from three orthogonal sources of variation: adjustments of the test coverage, stepping from component tests to integration tests, and stepping from active testing to passive testing.

The need to *adjust the test coverage* in different test suites derived from the same requirements arises for the following reasons: we have to test the SUT with *selected*, interesting sets of *signal parameters*, while using the same behaviour description. Furthermore, we want to test the SUT with different choices of its possible *behaviour*, which are either completely *random*, or which have a selectively *increased probability* for certain choices, or which are manually selected, fixed and *deterministic* test traces. All these test cases use the same description of admissible behaviour.

While starting to design *component tests* for an individual layer of the UMTS protocol stack, we already had to *prepare for integration tests* of several layers tested together, using the same description of admissible behaviour of the individual layers. Furthermore, already the RLC layer allows for multiple parallel instances of the same protocol machine. These run independently, at least at the black box behaviour level, but the implementation in SDL has a different, less decoupled structure. Therefore, integration tests for the protocol machine instances of this protocol layer were necessary.

All these tests had to be *active tests* where the test specification generates the test stimuli. It was nevertheless envisioned already at this stage to use the same behaviour descriptions in *passive tests*, where real UMTS air interfaces and upper protocol layers provide the stimuli to the SUT, and where the testing system can not monitor internal signals, but only external stimuli and reactions.

Integration of test suites and replacement of simulated by real environment are common maintenance tasks in test development. We designed the testing requirements as a *family of test suites*, with a modular structure. During this task, we identified the following *rules for modularizing requirements.*

There are two basic rules which lead to further, more specific rules:

- separate the description of the signature of a module from the description of the properties of its behaviour, and
- identify which requirements will likely change together, and to put them into one requirements module.

For the first of these rules, we discuss the case of CSP. A CSP signature consists of a set of channels with their parameters, and the behaviour is described by a composition of CSP processes over these channels. This separation allows to change the description of the behaviour of a module without changing the description of other modules that communicate with this module. The properties of the behaviour are hidden within the behaviour module. For example, with this strategy we can change the level of test coverage without changing the interfaces. Similarly, this separation enables to step from active to passive testing and from integration tests to component tests, as discussed below.

For the second rule, we note that it implies that all changes are within a particular module and do not affect the rest of the specification. Thus, the effects of a change remain local and manageable. The rule can be instantiated into several more specific rules, using the analysis of the expected changes:

- separate test stimulus generation from test observation;
- separate application specific and tester specific issues;
- for real-time testing, separate timer handling from application description; and
- for protocol testing, separate protocol layer specifications.

The modularization of requirements can be applied recursively. One example is the sub-structuring of a module into its signature and its behaviour.

In the remainder of this subsection, we discuss these specific rules. The *separation of test stimulus generation from test observation* was demonstrated in the vending machine example in Example 1 on page 82, where `RANDOM_STIMULI` and `OBSERVER` are separate CSP processes. We used the same structure for the UMTS protocol layers (see Figure 2):

In this modelling, a test stimulus generator, written in CSP, generates an input to the implementation under test and waits some defined amount of time, then it loops and generates the next input. Concurrently, a test observer process, also written in CSP, observes both the input stimuli and the output reactions of the system under test. If the behaviour of the system under test is incorrect, an error is flagged.

The choice of test coverage is private to the test stimulus generator, and the definition of the correct behaviour is private to the test observer. Therefore, this separation allows for several test suites which concentrate their coverage on a specific issue each but which use the same behaviour description, and it also allows for additional passive test suites which use only the behaviour description, but not the test stimulus generator at all.

The *separation of application specific and of tester specific issues* is a consequence of the fact that the testing tool usually is much more stable than the application requirements. Therefore, these should be put into a separate module. For our testing tool, tester specific events include `wrong_reaction` and `no_reaction`, which go into the test log, and the timer events.

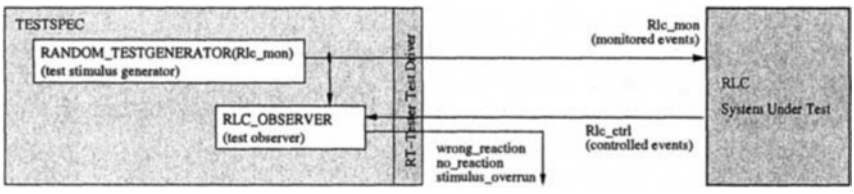

Figure 2. Separation of test stimulus generator and test observer.

Example 2 The timer-related CSP processes for Example 1 on page 82.

```
pragma AM_SET_TIMER
channel setTimer : { 0, 1 }
pragma AM_ELAPSED_TIMER
channel elapsedTimer : { 0, 1 }
pragma AM_ERROR
channel wrong_reaction, stimulus_overrun
pragma AM_WARNING
channel no_reaction

PAUSE = setTimer!0 -> elapsedTimer.0 -> SKIP

AWAIT(ExpectedEvSet) =
(   -- start timer and wait for things to come:
    setTimer!1 ->
    (   -- Accept correct reaction:
        ([] x: ExpectedEvSet @ x -> SKIP)
     [] -- Flag wrong reaction:
        ([] x: diff(CtrlEv + MonEv, ExpectedEvSet) @
            x -> wrong_reaction -> SKIP)
     [] -- Flag no reaction (timeout):
        elapsedTimer.1 -> no_reaction -> SKIP))
```

The process `PAUSE` sets a timer and waits until it elapses. Then it terminates and thereby returns to its calling process.

The process `AWAIT` assures that one of the specified set of legal outputs occurs in due time. If not, it performs one of the events `wrong_reaction` `no_reaction`, ... which go into the test log. This process also terminates and thereby returns to its calling process.

The *separation of timer handling from application description*, which should be done for real-time testing, is a particular instance of the previous rule. The realization of high-level timing processes by primitive set/elapse events of timers is completely internal to the timer handling module. Example 2 shows the definition of the CSP process `AWAIT` from Example 1 on page 82 which assures that one of the specified set of legal outputs occurs in due time.

Finally, the *separation of protocol layer specifications*, when testing protocols, encapsulates the behaviour of each layer into a separate module. Together with the separation of signatures and behaviours, this facilitates the generation of integration test suites from test suites for the components. If the rule is being followed, then specifications of admissible behaviour for the components can be re-used to get a description of the admissible system behaviour.

All these modularization rules must be implemented by mapping them to existing expressive means of the specification language. The dialect of CSP which we used allows to partition specifications into several files,

to organize these in a directory structure, and to compose the parts by an `include` statement. Example 1 on page 82 uses such an `include` statement in its first line to compose Example 2 into the specification.

In a large application, sub-directories can be used to express the hierarchical module structure, and individual files to express leaf-node modules. For example, in our UMTS application there is a sub-directory for each protocol layer. The sub-directory for the tester specific issues contains a further sub-directory for timer specific issues. All of these directories typically contain several variants of a specification part.

We have a separate directory tree which contains the actual test suites. Each sub-directory contains a file which composes the particular test specification from the requirements modules, and which contains further test-related data such as test result reports.

During integration testing, several protocol layer instances must be executed at the same time. This can be implemented by the parallel composition operator of CSP. In this composition, CSP process instances are parameterized with an instance number. These processes then run concurrently in parallel. Each process generates test stimuli for one of the protocol instances, and checks the corresponding reactions separately. The RT-TESTER tool allows to run several CSP specifications in parallel, with different interleaving strategies for the processes.

4. APPLICATION OF THE TESTING ENVIRONMENT

Subsequently, we describe a first experimental application of our testing environment in an industrial context. We provided testing support for a specific protocol stack layer in the user equipment of a mobile phone system.

4.1 The RLC Layer of the UMTS Protocol Stack

UMTS is a new international wireless telecommunication standard developed by the 3GPP consortium [1]. The standard comprises a layered architecture, where each layer relies on primitive services from the layer below and provides complex services to the layer above. Conceptually, each layer in the user equipment communicates with the same layer in the UMTS terrestrial radio access network.

Layer 1 is the physical layer of hardware services provided by the chip-set. Layer 2 is the data link layer. It provides the concept of a point-to-point connection to the network layer above. Layer 3, the network layer, provides network services such as establishment / release of a connection, hand-over, broadcast of messages to all users in a certain

geographical area, and notification of information to specific users. It includes the Radio Resource Control (RRC), which assigns, configures and releases wireless bandwidth (codes, frequencies etc.). Above layer 3 there are application layers containing functionality such as Call Control (CC) and Mobility Management (MM).

Layer 2 consists of several sub-layers: Medium Access Control (MAC), Radio Link Control (RLC), Packet Data Convergence Protocol (PDCP), and Broadcast and Multicast Control (BMC). The MAC provides unacknowledged transfer of service data units, reallocation of parameters such as user identity number and transport format. It furthermore reports local measurements such as traffic volume and quality of service indication to the RRC. The main task of the RLC is segmentation and reassembly of long data packets from higher layers into fixed width protocol data units, respectively. This includes flow control, error detection, retransmission, duplicate removal, and similar tasks.

The RLC layer of the UMTS protocol stack [8] provides three modes of data transfer: acknowledged (error-free), unacknowledged (immediate), and transparent (unchanged) mode. In acknowledged mode, the correct transmission of data is guaranteed to the upper layer; if unrecoverable errors occur, a notification is sent. In unacknowledged mode, erroneous and duplicate packets are deleted, but there is no retransmission or error correction: messages are delivered as soon as a complete set of packets is received. In transparent mode, higher layer data is forwarded without adding any protocol information; thus no error correction or duplicate removal can be done.

In all of these modes, the variable-length data packets received from the upper layer must be segmented into fixed-length RLC protocol data units (PDUs). Vice versa, for delivery to the higher layer, received PDUs have to be reassembled according to the attached sequence numbers. As additional services, the RLC offers a cipher mechanism preventing unauthorized access to message data. Thus, to transmit data, the RLC reads messages from the upper layer service access points (SAPs), performs segmentation and concatenation with other packets as needed, optionally encrypts the data, adds header information such as sequence numbers, and puts the packets into the transmission buffer. From there, the MAC assigns a channel for the packet and transmits it via layer 1 and radio waves. On the opposite side, packets arriving from the MAC in the receiver buffer are investigated for retransmissions, stripped from the RLC header information, decrypted if necessary and then reassembled according to the sequence numbering, before they are made accessible to the upper layers via the corresponding SAP.

Example 3 CSP specification of the initialization of an RLC connection in acknowledged mode.

```
-- The null state of the RLC abstract machine:
RLC_AM_NULL(instance_id) = instate_rlc_am_null.instance_id ->
(   -- Wait for crlc_config_req setup request and honour it:
    crlc_config_req.rbSetup.1.instance_id?dummy ->
        RLC_CONFIG_AM(instance_id,rlc_to_rrc)
 []
    -- No data transfer is yet possible, since the instance does not
    -- yet exist, thus no reaction is expected otherwise:
    -- (A release request is discarded in this state, too.)
    ([] x : diff(Rlc_mon,
            {| crlc_config_req.rbSetup.1.instance_id |}) @
        x -> RLC_AM_NULL(instance_id))
 []
    -- any spontaneous reaction produces a warning:
    ([] x : Rlc_ctrl @ x -> warn_spontaneous_event -> RLC_AM_NULL(instance_id))
)

-- The other states are omitted here:
RLC_CONFIG_AM(instance_id,rlc_dest) = ...

-- The main process:
RLC_AM_OBSERVER(instance_id) = RLC_AM_NULL(instance_id)
```

The observer for the RLC instance with the number `instance_id` starts in the state `RLC_AM_NULL` and waits for a configuration request event. If the event occurs, the instance goes to the next state. All other events to the SUT are ignored. Any spontaneous output from the SUT would be an error and is flagged.

A particular feature of the RLC is that there may be several instances coexisting at the same time. This is necessary since the services to the upper layers provide a variable number of connections, whereas the service of the lower layer provides a fixed number of logical channels. For efficiency reasons, however, the maximum number of parallel instances is statically fixed in the system.

Example 3 presents a CSP text fragment from our RLC layer specification. It describes part of the initialization of a connection in acknowledged mode.

4.2 Ambiguities in the Standards Document

Our tests yielded two kinds of results: the first kind are ambiguities and misinterpretations in the UMTS standards document. The second kind, discussed in Section 4.3 below, are deviations of the actual from the expected behaviour of the SUT.

One of the advantages of specification based testing is that it requires a formalization of the specification whose interfaces must be compared

to the implementation's interface. Writing this formalization, we found several ambiguities in the standard which can be subject to different, equally legal, incompatible interpretations.

For example, the RLC layer must accept several kinds of data as PDUs from the underlying MAC layer and forward it through the appropriate service access points (SAPs) of itself to its upper layers. Similarly, the RLC layer must forward data arriving through its SAPs as service primitives down to the appropriate "logical channels" of the underlying MAC layer. In both cases, the appropriate destination cannot be determined from a service primitive or PDU and their parameters alone, as given in the standard. Therefore the RLC SAP was split into two SAPs, distinguishing whether an upward bound signal should go to the RRC layer or to a different upper layer, and another parameter was added to most service primitives and PDUs which identifies the destination inside one layer. These necessary extensions have the consequence that the RLC layer can be used only with MAC and upper layers which add the same parameter and which use the same SAP split.

Another much less obvious, but potentially even more serious example is that we did not find any precise definition of the properties of a service access point (SAP), or of the MAC layer's logical channel. In particular, there is no definition of

- whether signals are forwarded instantaneously or whether they are buffered,
- a queueing discipline (FIFO, ...) in the case of buffering,
- queue delivery dependences between different SAPs (single queue/multiple queues),
- possible minimum/maximum delays between delivery and availability,
- the handling of signals that cannot be received.

The implementors had to make decisions on the above issues. Several of them were made more or less implicitly by choosing SDL as the implementation language, since the endpoint of an SDL signal route or channel has a precise, specific semantics. Further decisions were made by choosing a particular structure of SDL processes inside a layer (e.g., the number of independent queues per layer).

4.3 Testing of the SUT's Behaviour

There were several situations in which the SUT did not behave as expected. For example, in a certain state the SUT reacts to a certain signal

where no reaction was intended: an RLC protocol machine is created by the event `crlc_config_req.rbSetup` from the upper layers, and it becomes operational after receiving the event `mac_status_ind` from the underlying MAC layer. Immediately after that, the random test stimulus generator sometimes generated another (nonsensical) `mac_status_ind` event, to which the RLC layer sometimes, but not always, reacted by generating a data packet, i.e., with a `mac_data_req` event. Since no data transmission requests had been issued yet, and thus no buffered data could possibly be pending, this is unexpected. Probably no explicit test script would have been written that checks for a non-reaction in this state. The systematic random exploration of the state space in our approach found this problem automatically.

Furthermore, the tests revealed interactions between multiple instances of the RLC protocol machines. The requirements allow several instances of these machines which behave completely independent. Each instance could be tested separately. But we also performed a test where several protocol machines were tested at the same time, each one against its own copy of the requirements specification. It turned out that in such a setup there was the possibility that the entire SUT could deadlock. The reason for this effect was that the various instances of the RLC protocol machines are not implemented as entirely separate copies. Rather, there is a centralized routing SDL process which forwards data transmission requests to a set of SDL processes which implement one RLC instance each. This routing process did not handle the following case properly: if a signal arrived addressed to an RLC instance which does not currently exist, the process could loop infinitely during the instance look-up. It then ceased to perform its routing job. Even though the implementation was built with sophisticated error recovery mechanisms, this situation could not possibly have been foreseen in the development.

5. SUMMARY

In this paper, we described a testing setup for the UMTS protocol stack. A specific feature of this setup is that the interface between the system under test and the requirements specification is generated by an automated tool, which also performs consistency checks. This allows last-minute changes in the data formats.

The tool is complemented by a set of rules for designing the testing requirements as a family of test suites, with a modular structure. This prepares for a flexible maintenance of the initial test specification and reduces the work necessary for modifications.

Whereas our particular tool is tailored towards the development of UMTS software with SDL, the underlying ideas are applicable for other developments as well. Automating the interface generation between the testing tool and the system under test, and modularizing the requirements specification, is profitable under the following circumstances:

- a requirements document with complex interface definitions, which is authoritative for the final product, is being developed by an independent party;
- for the most part of the development time, the interface definition is in a draft state, with substantial changes to be expected and templates still to be filled in;
- after completion of the requirements document, an extremely short time to market is crucial for the success of the project.

Whenever these conditions are met, it is of advantage to shift verification and validation efforts into earlier design phases. By implementing automated interface generation tools already in the project preparation phase, and by additional up-front efforts for analyzing potential changes and for modularizing the requirements, the duration of the system integration and testing phase can be significantly reduced.

References

[1] 3rd Generation Partnership Project. http://www.3gpp.org.

[2] J. Ellsberger, D. Hogrefe, and A. Sarma: *SDL – Formal Object-oriented Language for Communicating Systems.* Prentice-Hall 1997.

[3] A. W. Roscoe: *The Theory and Practice of Concurrency.* Prentice-Hall 1997.

[4] J. Bredereke and B.-H. Schlingloff: Specification Based Testing of the UMTS Protocol Stack; in: *Proc. 14th Int. Software & Internet Quality Week (QW2001)*, San Francisco (May 2001).

[5] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe: On the design of the new testing language TTCN-3. In: H. Ural, R. L. Probert, and G. v. Bochmann, *Testing of Communicating Systems - Tools and Techniques*, vol. 176 of IFIP Conf. Kluwer Academic Publishers (Aug. 2000).

[6] Verified Systems International GmbH. `http://www.verified.de`

[7] Jan Peleska: *Formal Methods and the Development of Dependable Systems.* Habilitation thesis, Report No. 9612, Christian-Albrechts-Universität Kiel, and UniForM project, Technical Report 9601, Universität Bremen, Germany Dec. 1996.

[8] 3rd Generation Partnership Project, 3GPP TS 25.322 V4.0.0. RLC protocol specification (Mar. 2001).