Specification Based Testing of the UMTS Protocol Stack

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 present a specification based testing setup for the RLC layer of the UMTS protocol stack. Requirements are specified in the formal language CSP. From this we automatically generate real-time test scripts for the RLC, which is developed from SDL sources. Our testing approach is highly adaptable to changes in the UMTS standard and implementation: we developed an interface code generator with automated consistency checks, and modularized the requirements according to functional properties. We report on testing results and experiences with this setup.

1 Introduction

Current methods for testing embedded real-time control software can be classified as structural or specification based. *Structural testing* methods try to execute as many different parts of the *program code* as possible, where coverage is measured in terms of statements, conditionals, branches, function calls, and so on. *Specification based* methods treat the system under test as a black box and focus on testing the required *properties* of the system.

We have applied the latter approach in the development of the Radio Link Control (RLC) protocol layer of the Universal Mobile Telecommunication System (UMTS), a new generation of high-speed, multi-media mobile phone systems. The work is part of an ongoing cooperation between Siemens AG, Salzgitter, and Technologie-Zentrum Informatik (TZi), Bremen, where Siemens develops the code for the RLC layer, and TZi provides testing support.

In the case of UMTS, a standard is being defined by the 3GPP consortium (the 3rd Generation Partnership Project [1]). User equipment and base stations are to be developed by different companies and at different sites. Moreover, even the development of the software for different layers of the protocol often is distributed between several teams. 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. For such systems, specification based testing is more appropriate than structural testing.

Specification based methods treat the system under test (SUT) as a *black box* and focus on testing the required *properties* of the system. This has a number of significant advantages:

- the testing process concentrates on the user requirements and functionality aspects rather than on implementation details,
- ambiguities of the informal requirements (here, the UMTS standard) are exhibited,
- misinterpretations can be found, including errors arising from omissions and missing cases,
- a formal requirements specification implicitly contains test scripts of arbitrary length, the test coverage is limited only by the time available for a test run,
- a change in the SUT does not affect the test suites, and a change to a requirement affects one requirements module only.

In this project, without even running the tests, already in our first formalizations we found a number of deviations between our interpretation of the standard and the actual implementation. For example, frequently problems stem from cases which are under-specified in the standard (i.e., 3GPP did not state precisely what the required reaction to certain sequences of inputs should be) and which are interpreted differently by different readers. An annotated list of such deviations is a valuable documentation of design decisions arising from different views onto the standard. A formal specification can even be used as a reference which helps to achieve consistency and correct interoperability between components developed at different sites.

In conventional testing approaches, test cases are often formulated using a set of *test scripts*. These are explicit sequences of test inputs and of expected system outputs, describing in a step by step manner how the test should proceed. Specification based approaches do not need these long test scripts. Instead, the test scripts are implicitly contained in much shorter *formal specifications*. Tools can expand their powerful choice and concurrency operators automatically on the fly to conventional test scripts. 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. Test results are evaluated on the fly in real time during the run of the SUT.

With structural testing, all scripts have to be revised after each change in the SUT. Therefore, the testing process is costly and time-consuming. In specification based testing, due to its black box nature, all test cases can be re-used even if the implementation is changed. Thus errors can be corrected and regression tests can be done at virtually no additional costs. Since in the distributed development of the UMTS protocol stack inconsistencies are very likely, this feature is extremely important.

If some requirement is changed, with structural testing this usually means that several steps of several test scripts have to be changed, such that many test scripts need to be re-worked. In specification based testing, all these points of change are folded into the same few lines of the formal specification of this requirement. The formal specification is modular, each module describing a different aspect of the system's behaviour. Only the module which describes the new requirement has to be updated. For the UMTS protocol stack this is especially important since a number of items in the interpretation of the 3GPP documents are expected to be subject to change at any time.

Figure 1 describes our overall approach, where requirement specifications are formulated in CSP (Communicating Sequential Processes) [2], and system specifications

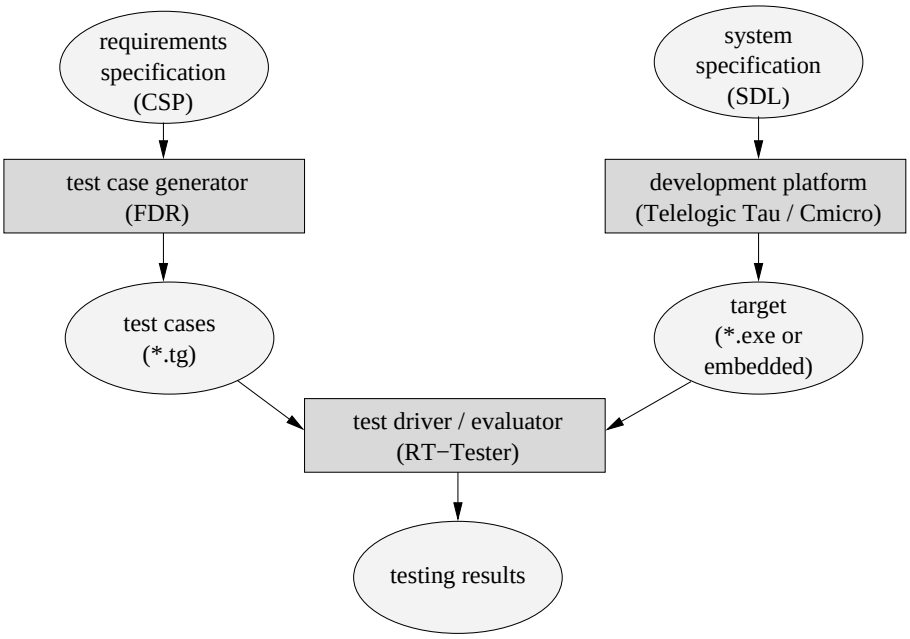

Fig. 1. Overall specification based testing approach.

are formulated in SDL (Specification and Description Language) [3]. From this, automatic tools are used for the generation of code and for the generation of test cases, and the system is tested automatically. This high degree of automation reduces the overall development time, which is particularly important in the current UMTS race.

Our paper is organized as follows: in Section 2, we give an overview of the functionality and properties of the RLC layer, and its implementation in SDL. Section 3 is the main part and deals with our automated testing environment: formal CSP specifications and the testing tool RT-TESTER, interfacing between the SUT and the testing tool, the formal CSP specification of the RLC layer, and testing of multiple instances in parallel and real-time. In Section 4, we describe and interpret the testing results, and in Section 5 we summarize our work.

2 The RLC Layer of UMTS

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 geograph-

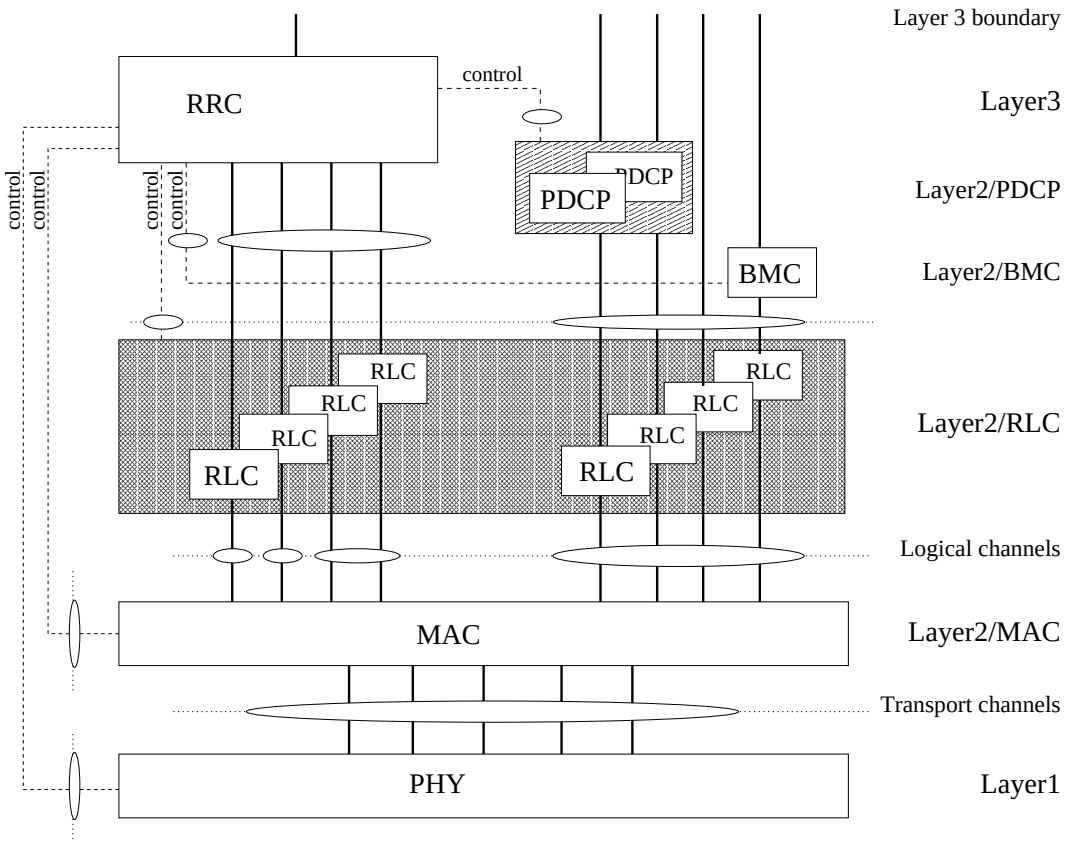

Fig. 2. Overall architecture of the UMTS radio interface protocol stack.

ical 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 is split into 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. An overview of this architecture is given in Figure 2, which is from the 3GPP standard.

2.1 Overview of Functionality and Properties

The RLC layer of the UMTS protocol stack [4] 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.

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.

2.2 Implementation in SDL

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 RLC instances. Earlier versions of the standard were accompanied by a detailed implementation suggestion described in the specification and description language SDL [3].

For example, in Figure 3 on the following page, we show the acknowledged mode states, and in Figure 4 on page 7, we show part of the corresponding SDL diagrams for the initialization of a connection in acknowledged mode (bold arrow in Figure 3).

The implementation is developed 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.

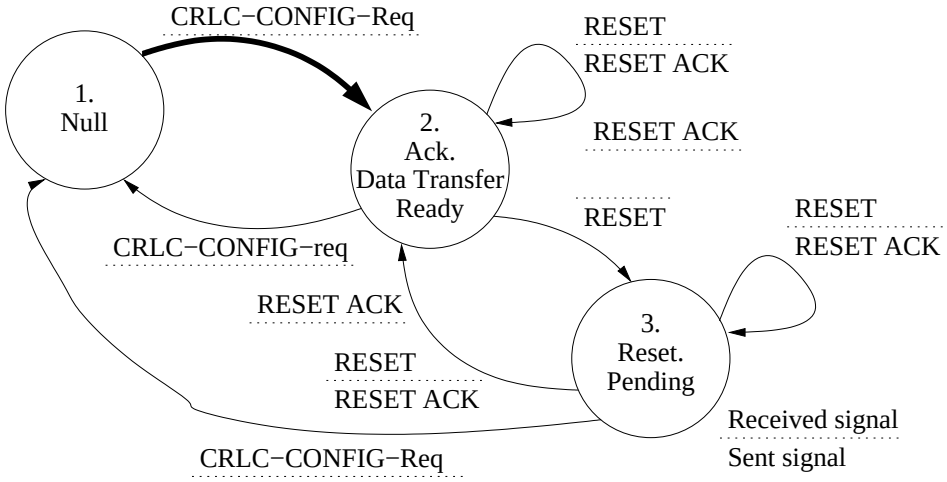

Fig. 3. RLC layer acknowledged mode states.

3 The Automated Testing Environment

In order to be able to find errors arising from misunderstandings or omission of cases, it is important that the system tests are developed independently from the system implementation. In the present project, we have formalized the requirements for the RLC using the process algebraic language CSP. These *formal specifications* describe the expected behaviour of the system under test (SUT) at its interfaces. From a CSP test specification of just a few pages, test scripts of arbitrary length are generated automatically by the supporting tool RT-TESTER [5]. In this section, we describe the general testing approach based on formal specifications, and give a detailed description of our interface between the CSP testing system and the SDL target. Then, we report on some specifics regarding the formal specification of the RLC layer, and, in particular, on testing multiple parallel instances of the RLC.

3.1 Formal CSP Specifications and RT-TESTER

CSP (Communicating Sequential Processes) [2] 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.

Example 1 on page 8 introduces to a few basic operators of CSP, in particular to 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 reads the CSP input and generates an internal repre-

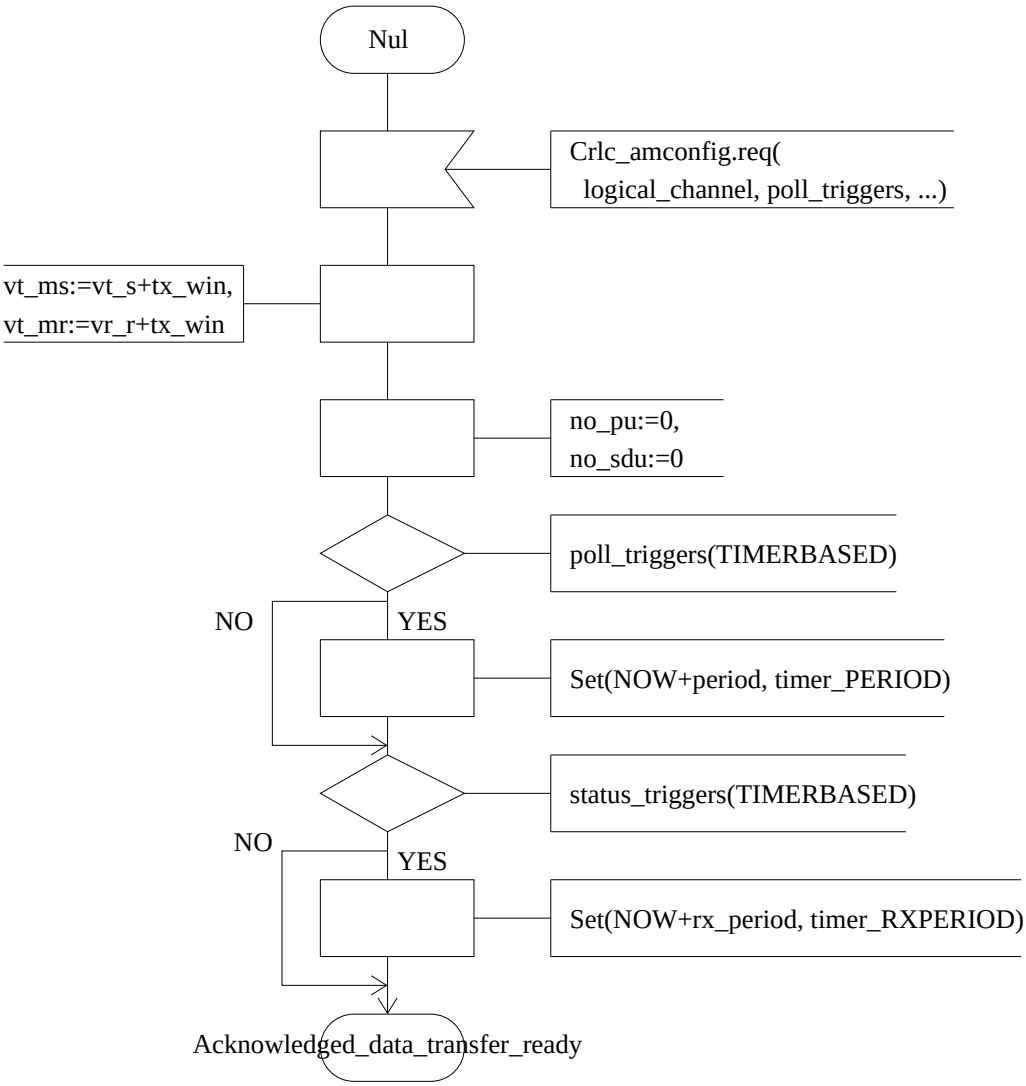

Fig. 4. SDL diagram for the initialization of a connection in acknowledged mode (bold arrow in Figure 3).

sentation from it, which is a huge graph of all allowed state transitions of the target. 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 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. It guarantees that the testing coverage increases steadily, approaching a full verification of the specified requirements. Functional properties such as correct transmission and packet routing can be tested together with real-time properties such as correct time slots and sufficient performance within the same test run.

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 sub-process `AWAIT` waits for any of the outputs specified (sequential process composition "`;`"). The definition of this process is listed in Example 3 on page 15 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`.

An example of a possible testing configuration is shown in Figure 5 on the facing page. In this setup, the testing machine is connected to three different development stages of the system under test: with an SDL runtime system, with a real-time operating system simulator and with the embedded target. During our actual testing, similar setups were used.

3.2 Interfacing SDL and CSP

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, dozens 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 structures, such as 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

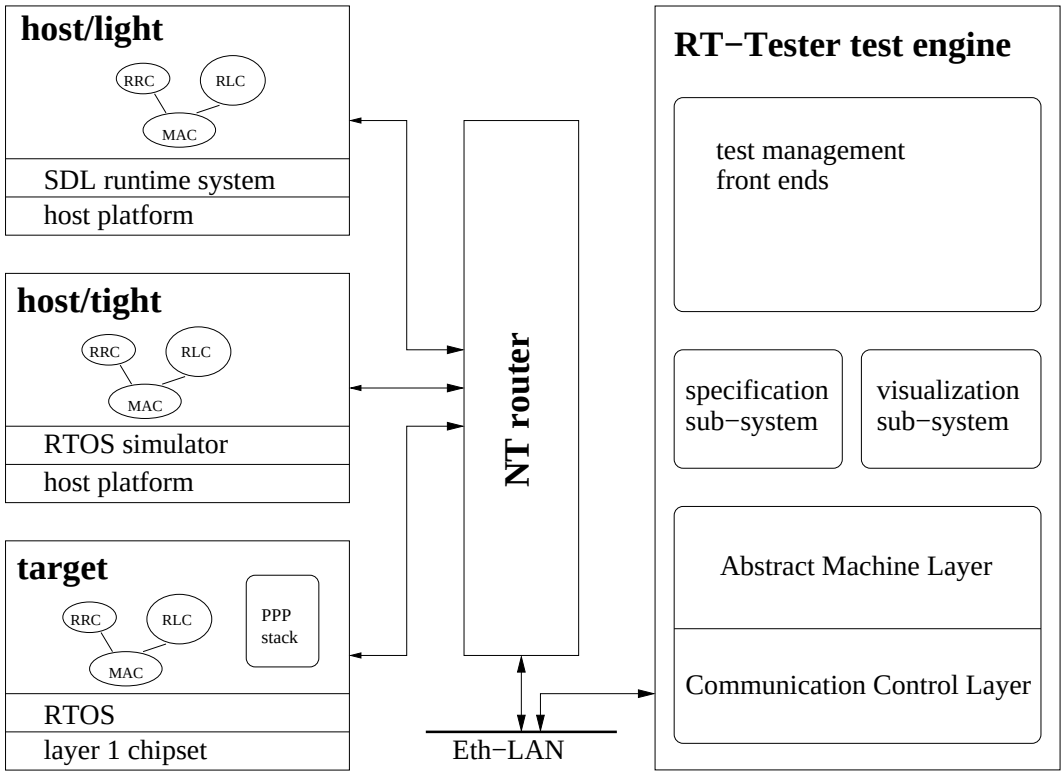

Fig. 5. Configuration for testing during development.

- 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
- modularizing the formal behaviour specifications into largely independent functional requirements.

The interface of the RLC layer relevant for testing is specified in SDL. Our automated approach makes it the only relevant interface. The SDL compiler generates a C language representation of the interface, and the C compiler generates a machine language representation of the interface, but both other representations are of no concern to our tests. 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.

These internal interfaces have to be consistent only. 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.

The interface in terms of SDL signals needs to be mapped to an interface in terms of RT-TESTER's native CSP channels. We need a translation between the two forms of

syntax. 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 flags no error, the interface descriptions of the SDL specification and of the CSP specification are guaranteed to be consistent, and the next test run can check whether the behaviour is consistent, too. If the SDL interface has been changed in a part that is relevant to the CSP tests, the mismatched items are logged and we can directly investigate the problem. With the RLC protocol layer this feature is especially important: this module uses large and complex data structures as 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 6 on the next page presents the concept of the automated interfacing. At the top there is the informal UMTS standard. Both the implementation team and the testing team interpret it and produce a formal specification in SDL and CSP, respectively. The names of the SDL signals and of the CSP channels must be the same in both specifications and are mapped automatically.

Channels and signals also can have parameters, which have to be mapped, too. In the simplest case, the first channel parameter is mapped onto the first signal parameter, and so on. In practice, this most simple scheme is rarely used. The mapping of the parameters usually requires user interaction, since CSP has a different concept of data structures than SDL, and since the UMTS standard provides data descriptions with huge data spaces, for example for the protocol data units (PDUs). Therefore, the CSP specification makes sensible abstractions to the state space, and the testing environment instantiates them with concrete values. Examples for sensible abstractions are blocks of, e.g., 512 bytes of data that will be transmitted transparently. Testing only a few representatives is sufficient.

We thus annotate each parameter of each CSP channel with the corresponding parameter of the corresponding SDL signal, as demonstrated in Example 2 on page 12. Particularly interesting is the "SKIP" annotation when we have data records as parameters that contain further sub-records. This is often the case for the protocol data units of the RLC layer. There are SDL signals with hundreds of sub-record components, of which most are entirely irrelevant to the protocol behaviour and thus to the testing purposes. An entire tree of such components is skipped implicitly by naming the top-most component only.

After the CSP interface has been annotated, "make" files and the generator tool can be invoked for the following steps (compare Figure 6 on the next page, again):

- The CSP specification is compiled as usual for the RT-TESTER runtime system.
- The interface adapter in the RT-TESTER runtime system also needs a (compiled) description of how the events on the CSP channels are mapped to byte strings. The generator tool produces this description automatically from the CSP specification.
- The SDL specification is compiled into a C language program, including C language header files that define C language names for the signals, their parameters, and

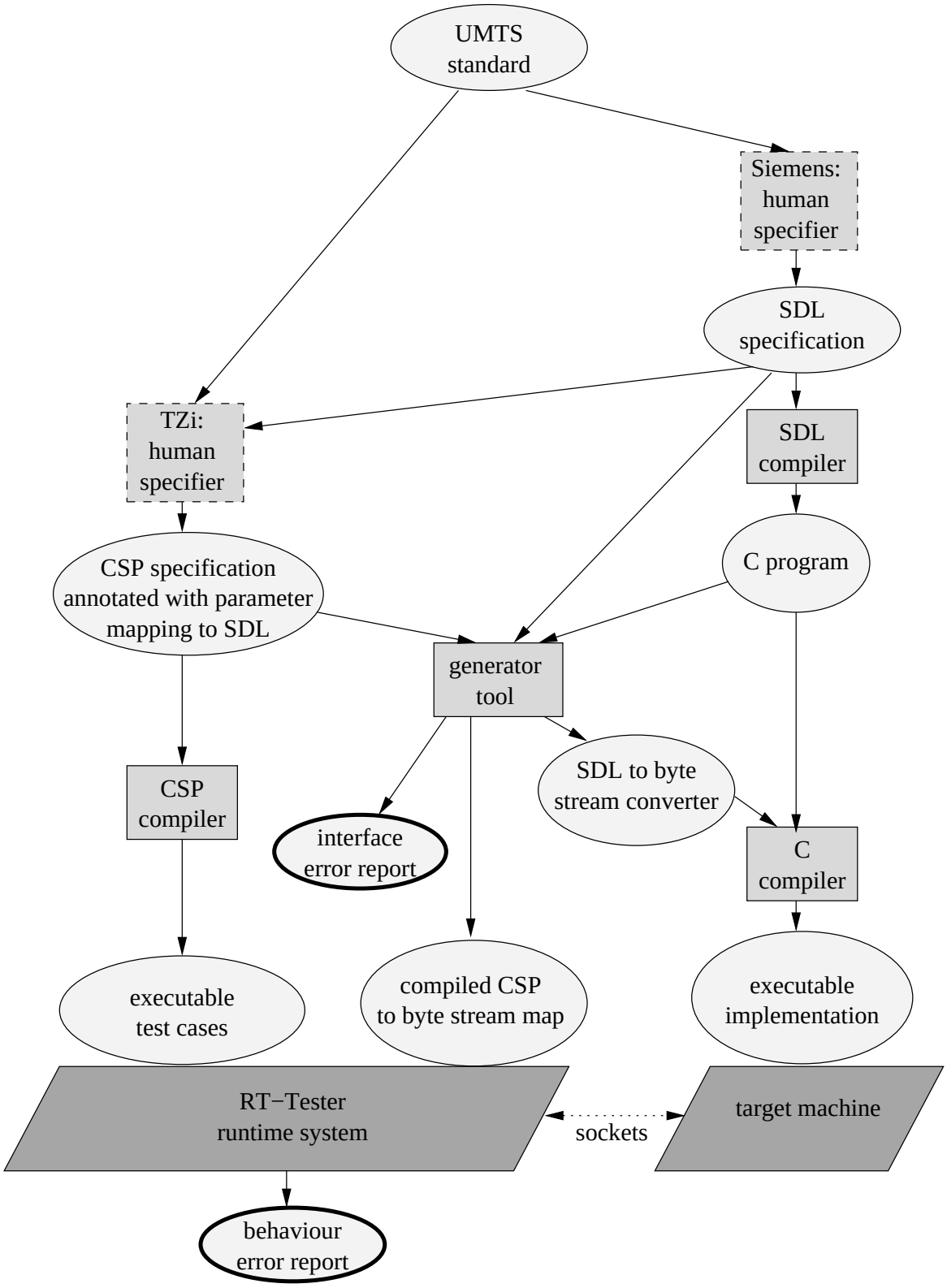

Fig. 6. Automated interfacing of SDL signals with CSP channels.

Example 2 An annotated CSP channel matching an SDL signal.

CSP

```
nametype Rb_identity = {0 .. maxRb_count}
datatype Rlc_data = dummy_value
channel rlc_tr_data_req :
pragma SDL_MATCH PARAM 1!RB_Id
      Rb_identity .
pragma SDL_MATCH TRANSLATE dummy_value 0x99 * 16
pragma SDL_MATCH PARAM 1!RLC_SDU_Data SUBSET_USED
      Rlc_data
pragma SDL_MATCH SKIP 1!length DEFAULT_VALUE 16
```

SDL

```
syntype RB_Identity = integer
      constants 0:MaxRb_Count
endsyntype;
synonym RLC_MAX_SDU_SIZE integer = 512;
newtype RLC_SDU_A
      carray(RLC_MAX_SDU_SIZE, octet)
endnewtype;
newtype RLC_SDU struct
      RB_Id RB_Identity;
      RLC_SDU_Data RLC_SDU_A;
      length integer;
endnewtype;
signal RLC_TR_DATA_Req(RLC_SDU);
```

The example defines a CSP channel named rlc_tr_data_req which has two parameters. The corresponding SDL signal RLC_TR_DATA_Req has one parameter which is a record of three components. With respect to mapping, each record component is treated like a separate parameter, in the order in which it is defined.

The first "pragma SDL_MATCH" line maps the first CSP channel parameter to the first SDL record component of the first (and only) SDL parameter, and it requires that the component is named "RB_Id". The CSP channel parameter is of type "Rb_identity" (a set of numbers serving as radio bearer ids). The SDL record component must have a type with the same set of values as "Rb_identity", otherwise the generator tool will flag an error.

The second CSP parameter is matched with the second SDL record component named "RLC_SDU_Data". This is actually an array of up to 512 octets (which are transmitted transparently). We do not want to test all possible values for this parameter, therefore we define the corresponding CSP type to comprise only one single value, called "dummy_value". In order to suppress the type mismatch warning message, we append "SUBSET_USED" to the pragma line. Furthermore, we want to use a well-recognizable pattern of data in the array. The "TRANSLATE" pragma line therefore specifies that the CSP dummy value should be sent as an array of sixteen hexadecimal bytes 99 to the SDL system under test. Similarly, only this pattern is recognized as a valid parameter value when received from the SDL system, all other patterns will result in a runtime error message in the test log.

Finally, the last "SKIP" line indicates that the third SDL record component "length" has no counterpart in CSP, but that it always should be filled with the value 16 (indicating the array length used).

the data types of the parameters, and a C language template file is generated that inputs and outputs SDL signals from and to the environment of the SDL system.

- This template is filled by the generator tool. The inserted code is part of the test interface adapter. The tool
  - takes the SDL specification and analyzes the signal definitions and the data type definitions in it,
  - matches it with the annotated CSP channel parameters,
  - takes the generated C language header files and determines the corresponding C language data types and parameter names, and
  - takes the C language template file and fills it. The encoding of the parameters into byte strings is the same as generated for the CSP side.
- The filled-out template and the other C files are then compiled into an executable implementation for the target machine.

The representation of the signals as byte strings is determined by the generator tool for both the CSP and the SDL side. The representation is independent of any machine architecture (as long as it provides strings of bytes). Each parameter is mapped to one byte, in the order in which they appear in the parameter list. If a parameter can take more values than a single byte can hold, it is mapped to more bytes. The byte order is defined by the tool to be least-significant-byte first.

The machine representations of data structures may be different on the target machine and on the machine running RT-TESTER. These machine representations may furthermore change during the project, e.g., due to changed choices of the hardware platform or of the embedded operating system.

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 target machine representation of the data entirely to the target 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.

3.3 Formal CSP Specification of the RLC Protocol

We not only have to cope with changing requirements, but also with different testing scenarios. We accommodate for both by a modular structure of the formal behaviour specification.

In one scenario, we want to test an individual layer of the UMTS protocol stack. In another scenario we want to test the integration of several layers. Furthermore, some of the tests will be driven by external stimuli from the real world which are not under the control of the testing system. We therefore have to specify the test stimulus generator and the test observer as separate modules, to separate the specifications of the different layers, and to compose these modules in different ways.

Furthermore, some crucial aspects of the SUT have to be tested more thoroughly than others. We therefore encapsulate these aspects into sub-modules, compose suitably tailored instances to complex test suites as needed, and run them using the RT-TESTER tool. In all these cases, the actual test scripts are generated automatically on the fly.

The vending machine toy example in Example 1 on page 8 above composes the components already in the same way as the CSP specifications of the RLC layer (Figure 7): 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. Example 3 on page 15 shows the definition of the CSP process `AWAIT` from Example 1 which assures that one of the specified set of legal outputs occurs in due time.

We can see how the definition of the process `AWAIT` is separated into a different file, and how it is integrated into the toy example by an `include` statement. We use a similar process in our test suite specifications in the same way.

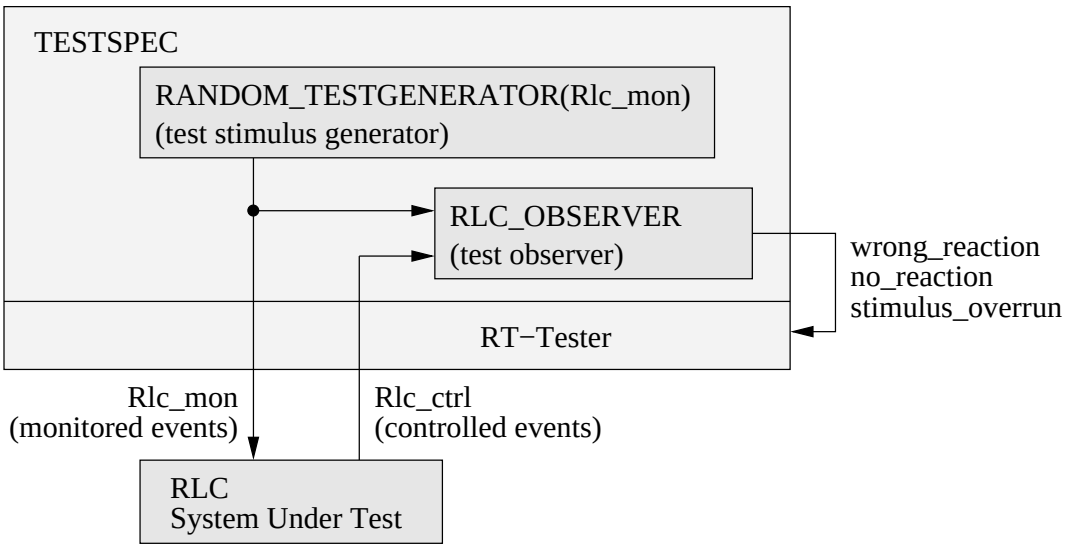

Fig. 7. The split between the stimulus generator and the test observer.

We separated the definitions of the different layers of the UMTS protocol stack into separate files. Beyond this, we also separated the data type definitions, the channel definitions, and the actual behaviour definitions. For several of these definitions, we designed different variants that can be composed in various ways.

For example, a test stimulus generator similar to the one in Example 1 generates test stimuli completely randomly. Another variant performs random choices with different probabilities, such that "interesting" states are reached more often, and yet another variant performs a fixed, explicitly specified trace of events. The latter has no internal choice operators at all, it just consists of a long sequence of event prefix operators ("`->`").

Example 4 on page 16 presents a CSP text fragment from our RLC layer specification. It describes part of the initialization of a connection in acknowledged mode (bold arrow in Figure 3 on page 6, SDL code in Figure 4 on page 7).

3.4 Testing Multiple RLC Instances in Parallel

The RLC layer comprises a number of protocol machine instances that run independently. This is true at least at the black box behaviour level, even though the implementation in SDL has a different, less decoupled structure. Therefore we specify and perform the testing of the entire RLC layer by replicating the CSP process for a single protocol instance, and by parameterizing it with the instance number. These processes run concurrently in parallel, each generating test stimuli for one of the RLC instances, and checking the reactions.

This testing setup is an example of a testing setup with several components tested at once. It is also possible to combine different kinds of components, for example RLC and MAC instances. Such a testing setup can be built by combining the CSP processes for the test observers into one test specification, and by selecting a suitable test generator.

Example 3 The timer-related CSP processes for Example 1 on page 8.

```
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, ExpectedEvSet) @
            x -> wrong_reaction -> SKIP)
     [] -- Flag overrun by next monitored event:
        ([] x: MonEv @ x -> stimulus_overrun -> 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.

4 Testing Results

Our tests yielded two kinds of results: the one kind, discussed in Section 4.2 below, are deviations in the behaviour of the SUT. The other kind was due to the specification based testing approach. In writing the formal requirements specification and comparing its interface to the implementation's interface, we found several ambiguities in the UMTS standards document.

4.1 Ambiguities in the Standards Document

The ambiguities which we found can be subject to different, equally legal, incompatible interpretations. These places will need special care to avoid inter-operability problems with software developed at other sites or by different manufacturers. (Compare Figure 8 on page 17.)

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"

Example 4 CSP specification of the initialization of an RLC connection in acknowledged mode.

```
-- The null state of the RLC (AM) entity:
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.

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 possibility of data loss or alteration,
- the handling of signals that cannot be received.

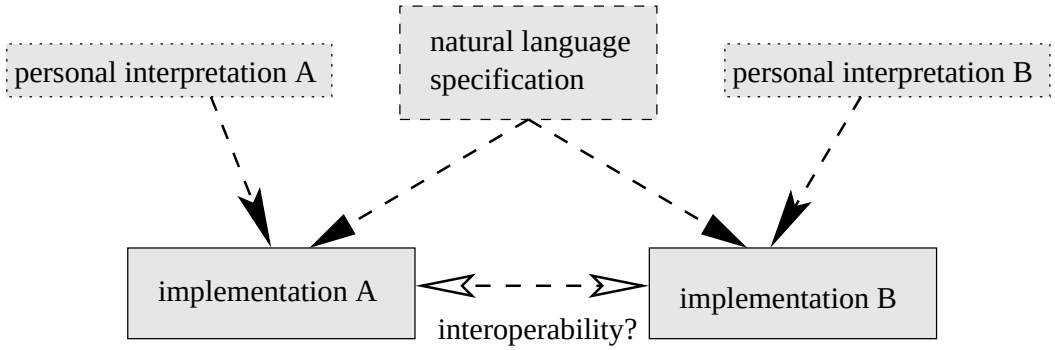

Fig. 8. Potential interoperability problems through an incomplete specification.

We can probably assume safely that neither SAPs nor logical channels lose or alter signals. But the other five issues can have an impact on the behaviour of the system. In particular, delays in the delivery of signals can result in *race situations*, or prevent them.

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 (number of independent queues per layer, ...).

Besides these ambiguities in the standard, we also found a few deviations in the implementation's interface definition from the interface defined in the standard.

4.2 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 strangely 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. With structural testing, 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 different instances of the RLC protocol machines. The requirements allow different 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 the different instances of the RLC protocol machines are not implemented entirely as 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.

Another interesting observation showed up only intermittently, after a certain history of input: even though the SUT always *should* have gone into the same state, it did not. Besides the random test stimulus generation, we also used a generator which performs a fixed, explicitly specified trace of events, and which starts all over from the beginning when the listed trace is through. A correct SUT should return to the initial state at the end of the listed trace, and then it should go through the same states again, delivering the same reactions. In fact, it turned out that in the second, fourth, etc. rounds some data transmission requests `rlc_tr_data_req` from the upper layers towards the MAC were lost, which were transmitted correctly by the RLC layer in the first, third, etc. rounds. The morale is that a protocol machine instance implementation does not necessarily return to its initial state just because the instance is "freed". Implementation optimizations can retain parts of the old state, and this can lead to subtle misbehaviour. The on-the-fly generation of the test scripts allowed us to run the test suite for a long period of time, which was necessary to detect the above problem.

5 Summary

In this paper, we described a testing setup for the UMTS protocol stack. Specific features of this setup are

- test scripts are generated automatically from formal requirements specifications which are developed independently of the implementation,
- the interface between the system under test and the requirements specification is generated by an automated tool. The tool also performs consistency checks. This allows last-minute changes in the data formats.

The paper describes ongoing work. In particular, we did not yet run the test cases on the embedded target (prototype of UMTS user equipment). We expect to be able to re-use all test specifications directly since they refer to external interfaces only.

References

1. 3rd Generation Partnership Project. http://www.3gpp.org.
2. A. W. Roscoe. "The Theory and Practice of Concurrency". Prentice-Hall (1997).
3. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. "SDL – Formal Object-oriented Language for Communicating Systems". Prentice-Hall (1997).
4. 3rd Generation Partnership Project, 3GPP TS 25.322 V4.0.0. "RLC protocol specification" (March 2001).
5. Verified Systems International GmbH. http://www.verified.de.