Verifying reliable data transmission over UMTS radio interface with High Level Petri Nets

Teemu Tynjälä[1], Sari Leppänen[2], and Vesa Luukkala[2]

[1] Helsinki University of Technology, Laboratory for Theoretical Computer Science, P.O. Box 9205, 02015 Espoo, Finland,
tjtynjal@saturn.tcs.hut.fi

[2] Nokia Research Center, Mobile Networks Laboratory, P.O.Box 407, 00045 Nokia Group, Finland
{sari.leppanen,vesa.luukkala}@nokia.com

Abstract. Standard specifications of telecommunication protocols are mainly written using informal language. Therefore testing the standard, i.e. the definition of the core functionality, requires formal modelling of the protocol. In this article we describe the modelling and verification of a third generation mobile telecommunication protocol. We take the SDL description of the protocol as a starting point for our High Level Petri Net model. Since the size of the real-life protocols is enormous, we apply various abstraction techniques in the modelling phase. Basing on these and our previous experience, we introduce several heuristics for intelligent protocol modelling. Next we describe in detail the verification and modelling task, and derivation of the properties to be verified from the protocol specification. We are able to verify the most essential properties for reliable data transmission. Before executing the actual verification task, we plan a verification strategy, where for example the verification order of the properties and the appropriate configurations for the protocol and channel components for each run are considered.

1 Introduction

Today standardization of telecommunication systems is more challenging than ever. Tight competition on market shares creates great pressures on time schedules. Production cycles become shorter, although the size and the complexity of systems explode at the same time.

Specification of a new telecommunication protocol is not an easy job as such. In the framework created by the underlying technology, the services provided by the mobile network should be realized in the environment, where complicated algorithms implementing the services are distributed over the network. In standardization process all kind of problems related to concurrency, reactivity, data consistency, reliability and real-time requirements can occur. Evaluation and analysis of a concurrent and reactive system is tedious. A huge number of various computation paths should be considered already in a case of a small and simple system. For the human mind this kind of "concurrent thinking" is next to impossible and the tight time shedule makes the task even harder.

Problems met in standardization are directly reflected in the implementation phase. Implementations are started long before the standardization is finished or even stabilized. Therefore the specifications, which the implementations rest on, are often incomplete, erroneous and ambiguous. Clearly there is a compelling need for a method to test the specifications before starting the high-volume implementations.

Algorithmic verification provides methods to automatically analyze various computation paths of a protocol model. One such technique is known as model checking. The model is a simplification of reality, describing only the relevant aspects and the basic logical functionality of the protocol. When applied to standardization, the main phases of the verification would be modelling, verification and reporting. Verification process is iterative and requires strong teamwork among the verification, standardization and design people.

In the modelling phase the formal model of the protocol is created together with the standardization people and designers. Also the requirements to be verified are described with appropriate formalism and the verification strategy is defined. The verification strategy is a plan for executing the actual verification task. For example, the verification order of the requirements is fixed, and the techniques and tools to be used are chosen. In the reporting phase, in addition to verification results, also proposals for correcting the errors found could be included. This helps greatly when writing the contributions and change requests to standardization.

We will describe verification of a mobile telecommunication protocol standardized by *3GPP (3rd Generation Partnership Project)*. The protocol specification includes a formal model of the protocol, described with *SDL (Specification and Description Language)* [3]. We converted the SDL description into a High Level Petri Net model and used the *Maria (Modular Reachability Analyser)* tool [9] for verification and error tracing. We describe a collection of abstraction techniques for modelling with High Level Petri Nets. When constructing a model for large, real-life systems, abstractions and intelligent modelling play an important role. Since the purpose of model construction is to facilitate analysis, like verification, the model should be of tractable size and still present the relevant interesting properties.

The rest of the article is structured as follows. In section 2 we present the modelling formalism and in section 3 the protocol to be verified. Section 4 describes the model construction phase, including the verification model architecture and abstractions used in modelling. Section 5 describes the analysis phase consisting of the requirement definition and formulation, verification strategy and description of verification results. Conclusions are finally drawn in section 6.

2 High Level Petri Nets

The basic formalism was developed by Carl Adam Petri in 1962, and the theory has since been substantially developed. The concept of High Level Petri Nets emerged in the mid-1980s, and not until recently has a consensus been reached about the way high level features are to be modelled in the Petri Net world. This exposition is based largely on the proposed Petri Net *standard* [5], which is currently in the draft stage in the International Standards Organization (ISO).

Petri Nets provide an appealing graphical picture of a parallel and distributed system. System data is represented by *places* in the Petri Net model, and the system dynamics are taken care of by *transitions*. Only small and medium size systems are amenable to the graphical Petri Net representation. For larger systems one has to resort to a textual representation that is usually governed by the tool that will be used in the verification process.

2.1 Formal Definition

The formal definition of High Level Petri Net Graphs is as follows [5]: A *HLPNG* (High Level Petri Net Graph) is a structure
$HLPNG = (NG, Sig, V, H, Type, AN, M_0)$ where

- $NG = (P, T; F)$ is called a net graph, with
  - P a finite set of nodes, called Places;
  - T a finite set of nodes, called Transitions, disjoint from P ($P \cap T = \emptyset$); and
  - $F \subseteq (P \times T) \cup (T \times P)$ a set of directed edges called arcs, known as the flow relation
- $Sig = (S, O)$ is a Boolean signature. Here S represents the sorts and O represents the operations on the sorts
- V is an S-indexed set of variables, disjoint from O. This means that every variable is indexed by the sort that its value has
- $H = (S_H, O_H)$ is a many-sorted algebra for the signature Sig. A many-sorted algebra contains a carrier for each sort, and a carrier for each operation. For example, if a sort is defined as Int, one possible carrier for Int is the set $\{\ldots, -1, 0, 1, \ldots\}$.
- $Type : P \longrightarrow S_H$ is a function which assigns types to places. This equation means that each place is associated with a carrier for the types of data it may contain.
- $AN = (A, TC)$ is a pair of net annotations.
  - $A : F \longrightarrow TERM(O \cup V)$ such that for all $(p, t), (t', p) \in F$, for all bindings $\alpha, Val_\alpha(A(p, t)), Val_\alpha(A(t', p)) \in \mu Type(p)$. A is a function that annotates each arc with a term that when evaluated (for any binding) results in a multiset over the associated place's type.
  - $TC : T \longrightarrow TERM(O \cup V)_{Bool}$ is a function that annotates transitions with Boolean expressions (transition condition)
- $M : P \longrightarrow \bigcup_{p \in P} \mu Type(p)$ such that $\forall p \in P, M_0(p) \in \mu Type(p)$ is a marking of a system which associates a multiset of tokens of the correct type with each place.
- M_0 is the initial marking of the net. It is defined in the same way as a normal marking.

In the above, μA is used to denote multisets of a set A. In a multiset, it is possible to have a number of elements of the same kind. As an example, if our base set A is $\{a, b\}$, then an example of a valid member of the multiset μA would be $\{3'a,\ 2'b\}$. Here the number left of the ' symbol represents the multiplicity of the object in question.

2.2 A Formal Interleaving Semantics

Petri Net model of a system has dynamics which are brought about by the firing of transitions. In order to know when a transition may fire, we must define when a transition is enabled [5].

A transition $t \in T$ is enabled in a marking, M, for a particular assignment, α_t, to its variables, that satisfies the transition condition, $Val_{bool}(TC(t)) = true$, known as a $mode$ of t iff

$$\forall p \in P : Val_{\alpha_t}(\overline{p,t}) \leq M(p)$$

where for $(u,v) \in (P \times T) \cup (T \times P)$,

– $\overline{u,v} = A(u,v)$ for $(u,v) \in F$
– $\overline{u,v} = \Phi$, for $(u,v) \notin F$

where Φ is a symbol that represents the empty multiset at the level of the signature, so that $Val_\alpha(\Phi) = \emptyset$

If a transition $t \in T$ is enabled in *mode* α_t, for marking M, t may *occur in mode* α_t. When t occurs in mode α_t, the marking of the net is transformed to a new marking M', denoted $M[t, \alpha_t\rangle M'$, according to the following rule:

$$\forall p \in P : M'(p) = M(p) - Val_{\alpha_t}(\overline{p,t}) + Val_{\alpha_t}(\overline{t,p})$$

The new marking is a consequence of taking the tuples indicated on the incoming arc away from the pre-places and inserting the tuples indicated on the outgoing arc to the post-places. As an applied example from protocol design, a transition could take away a tuple from a place denoting its input queue and insert the received message (i.e. the tuple) into a local buffer, also represented as an HLPN place.

An example of the semantics of Petri Nets is given in Figure 1. Transition $t1$ may fire in the following modes (note that these are marking dependent). (t1, {x=1, y=3}), (t1, {x=1, y=4}), (t1, {x=1,y=5}), (t1, {x=1, y=7}), (t1, {x=3, y=4}), (t1, {x=3,y=5}), (t1, {x=3, y=7}).

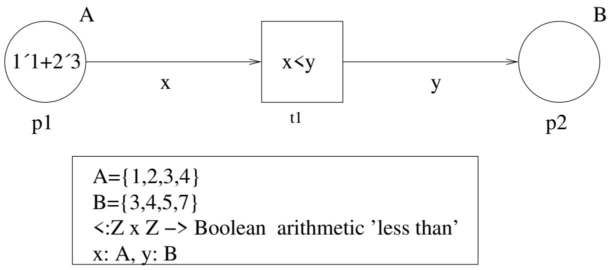

Fig. 1. Example of the definition of High Level Petri Net semantics

3 Radio Link Control Protocol

UMTS (Universal Mobile Telecommunication System) is a third generation mobile telecommunication system using *WCDMA (Wideband Code Division Multiple Access)* [12] radio access technique. Adoption of the new radio access technique requires major changes in the *Radio Access Network (RAN)*. Some of the protocols interacting over the radio interface have to be modified according to the requirements of the underlying technique and also some new protocols have to be specified.

Radio Link Control (RLC) protocol is one of the new UMTS RAN protocols. According to the OSI reference model RLC is a data link layer (layer 2) protocol providing data transmission service to upper layers. RLC was standardized by 3GPP, an international standardization forum, on March 2000.

The RLC specification [1] defines several modes for data transmission: unacknowledged mode, transparent mode and acknowledged mode. We choose the most complex one for our verification, namely the acknowledged mode. In acknowledged mode RLC provides a reliable data transmission over the unreliable radio interface. It uses the unreliable data transmission service provided by *MAC (Medium Access Control)* protocol, another data link layer protocol. The UMTS data link layer consists of MAC and RLC sublayers. RLC provides a radio solution dependent reliable link for the user and MAC controls the access signaling procedures for the radio channel. RLC operates both in control plane and user plane. In control plane the user of the data transmission service is the *RRC (Radio Resource Control)* protocol and in user plane the *PDCP (Packet Data Compression Protocol)*. The structure of UMTS data link layer and its users is described in figure 2.

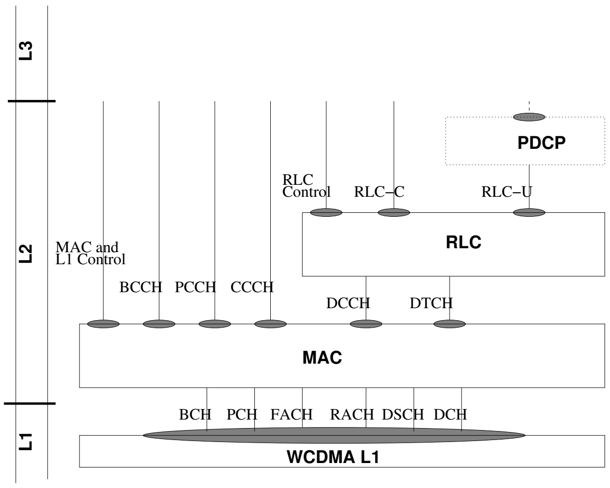

Fig. 2. UMTS data link layer.

3.1 Functional description

Here we will shortly outline the basic functionality of the RLC protocol in acknowledged mode. For more detailed description, see [1] and [6]. In acknowledged mode RLC guarantees reliable data transmission for upper layers by means of retransmission. In case RLC is not able to deliver the data correctly, the user in the sending side is notified. Acknowledged mode is used for packet switched data transfer.

RLC connection is established by the RRC protocol. In connection establishment the configuration parameters defining e.g. the functional mode and polling and status sending (defining the acknowledgement and resending methods) mechanisms are delivered. RLC connection can be reconfigured later, during the data transmission phase.

The sliding window mechanism is used in data transmission for flow control. After receiving a data packet, an *SDU (Service Data Unit)* from the user, the SDU is segmented into *PDUs (Protocol Data Units)* and each PDU is given a unique sequence number. After this the PDUs are stored in *AMD (Acknowledged Mode Data)* queue waiting to be transmitted. At each PDU transmission it is decided whether the *polling bit* is set. Polling bit asks the receiving entity to respond with a STATUS message. Poll trigger is indicated in the RLC configuration messages.

The functionality in the receiving side depends on the *status trigger* used. Status trigger regulates the sending of STATUS message. This is indicated in the RLC configuration message. If the received PDU is inside the receiving window it is stored in the assembly queue. Each time a new message is added into the assembly queue, we check whether it is possible to assemble a complete SDU. If this is the case the assembled SDU is passed to the receiving user process. On the other hand, if the receiving RLC notices a missing PDU a STATUS message is generated to the sending RLC asking for a retransmission.

In the case of unsuccesful retransmission the *SDU discard procedure* is initiated at the sending end. The procedure asks the receiving entity to move its receiving window forward and restart accepting messages. After sending the MRW (Move Receiving Window) message the sending RLC starts a timer. If no response (MRW-ACK) is received before the timer expires, the *reset* procedure is executed. MRW message can be retransmitted a fixed number of times before triggering the reset procedure.

The reset procedure is similar to the SDU discard procedure. A RESET message is transmitted up to a predetemined maximum number of times and if no RESET-ACK is received in response, the sending user is informed of an unrecoverable protocol error. In a succesful case the internal RLC data buffers are flushed and the sequence numbering is reset. The user is informed about SDU transfer acknowledgements (positive and negative), resets and unrecoverable protocol errors by means of messages that are considered to be in-stack communication.

4 High Level Petri Net model

In the modelling process of the RLC we chose to construct a Petri Net model for the separate entities shown in Figure 3. First, the SDL description of the RLC was converted into High Level Petri Nets. This process covered the sending and receiving RLC. Next, a model for the channel connecting the two RLC entities was built. Finally, the Petri Net models of the sending and receiving user processes, corresponding to the RRC/PDCP protocol (and the property to be verified) in the protocol stack, were added to the model. Although the conversion was done by hand, the rules used (which are discussed in the following section) are deterministic, and in a later release of the Maria analyzer, the conversion will be automatic as there will be an SDL front-end available.

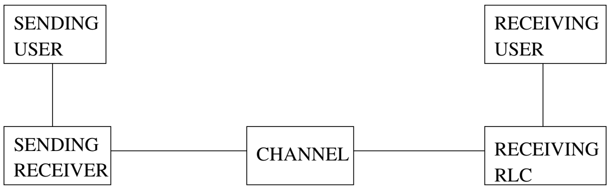

Fig. 3. Architecture for the verification process of the RLC

4.1 RLC Protocol

In the modelling process of the RLC we found a number of modelling principles useful in minimizing the resulting reachability graph's size. We divided these principles into two categories: the category of generic protocol modelling principles and the category of protocol specific modelling principles. In the following list, we discuss the generic protocol modelling principles.

1. To create a link between a protocol's description in SDL and the Petri Net model, each SDL statement should be labelled with a program counter.
2. To aid the analysis process, a protocol's transmission and reception windows should be parameterizable. The model specific errors are often caught with transmission window size one, whereas a protocol's logical errors are caught most often with transmission window size 2 or 3 [2]. If a protocol relies on retransmissions, the maximum number of allowed retransmissions should also be made a protocol parameter, which is included in the formal model.
3. To verify SDU segmentation and reassembly, the sizes of PDUs and SDUs must be modified. The correct functioning of segmentation and reassembly can be checked either if several SDUs can fit into one PDU, or vice versa. In our case, we chose to use SDUs of length 3 and PDUs of length 2.
4. One should differentiate between in-stack communication (i.e. between layers (N) and (N+1)) and communication among peer entities. An example of in-stack communication is the operation where the transport layer element ((N+1)-entity) orders the network layer element ((N)-element) to deliver a packet. A distinguising feature of in-stack communication with respect to peer-to-peer communication is that the communicating entities in in-stack communication reside in the same piece of hardware whereas peer-to-peer communication involves entities in different pieces of

hardware. Normally a specification does not elaborate on in-stack communication, but it is often considered immediate and loss-free. On the other hand, peer-to-peer communication should be modelled so as to allow transfer delay, message loss, duplication and reordering. Moreover, the former type of communication should be prioritized over the latter.

5. If the protocol involves long sequences of local actions (i.e. actions involving no communication or other parallel activity), the state space can be greatly reduced by adopting the resource place abstraction [10]. With this abstraction, local SDL sequences may be made atomic. Interleaving is allowed only in case the statement involved communication.
6. The lengths of message queues between two RLC entities and between a user entity and RLC entity should be chosen to minimize state space. As in-stack communication is loss-free and immediate, a queue of length one between a user process and a peer entity should be chosen. As for peer-to-peer communication, the input queue should be at least twice the size of the transmission window to allow message duplication and reordering.
7. The timers in the model should be abstracted to two values — on or off. This Boolean abstraction for timers increases the size of a subgraph of the reachability graph. With Boolean abstraction, all $n!$ timer firing orders are explored. However, we have found that the inclusion of local time in the Petri Net model increases the reachability graph's size even more. This timer abstraction is completely analogous to abstracting the set $\{0, 1, 2, 3, \ldots\}$ to two values $\{0, > 0\}$ as an application of abstract interpretation.
8. If the underlying channel type is reordering, it is wise to use an implicit scheduling mechanism in the model. A peer entity first reads all the messages from its input queue and reacts to each one in succession. When all messages have been processed by this peer (and some messages were actually transmitted in return), the "turn" is given to the other peer. As the channel is reordering, it may be modelled as a High Level Petri Net place. The other peer may read the messages from this place in any order. Hence, the non-deterministic behaviour of the protocol is sufficiently retained.

In addition to these protocol independent modelling principles, we employed one RLC specific abstraction. The RLC can be configured to run in several different modes. The three parameters that define the behaviour of RLC are *status trigger*, *poll trigger* and *discard trigger*. For simplicity, we used a constant set of triggers. Status trigger is *missing pdu detected*, poll trigger is *every poll pdu*, and discard trigger is *maximum number of retransmissions*.

In the RLC model, every Data PDU had its poll bit set, the receiver replied with a STATUS message each time it detected that it had missed a PDU, and the SDU discard procedure was begun after a maximum number of retransmissions of the PDU. In our case, this maximum number of retransmissions was zero, indicating that the SDU discard was begun immediately if we did not receive an acknowledgement of the data PDU.

The other RLC specific modelling choice involved the STATUS PDU. The STATUS PDU in RLC is rich in meaning since it encodes the following message types in a single PDU: ACK, WINDOW, MRW, MRW ACK. We chose to use a single message type in each STATUS message although nothing prevents one from using several in a single message. This abstraction brought some false negative acknowledgements to the protocol's execution, as a STATUS ACK was disregarded during an SDU discard procedure. In other words, even though the recipient successfully acknowledged a PDU and delivered the related SDU, the sender would disregard the ACK if it had already commenced the SDU discard procedure.

4.2 Channel Types

The verification process considered realistic and pessimistic models of communication channels. In our work, three channel types were chosen to be taken into the analysis. The first was a realistic assumption of the radio interface, a channel that may lose messages but preserves order. The second was a pessimistic view of the radio interface, namely a lossy and reordering channel. The final one was the worst possible channel type, a lossy, reordering and duplicating channel.

In the actual Petri Net model, the channel functionalities were integrated into the sending and receiving RLC entities. They were not included in a separate *channel process* as Figure 3 may suggest.

4.3 User Processes

The user processes define the way in which connection is set up and controlled. Before the RLC process is initialized, it is assumed that the user process (layer (N+1)) had already reserved a radio link for it. This assumption is valid both in the sending and receiving end. It is then a justified action to send the connection-initiating message to both RLC entities in a single transition. The normal behaviour of the user process is to send and receive data as well as both positive and negative acknowledgements for the data.

In the event of a recoverable error, the sending RLC tells the user process that it has successfully resynchronized itself with the receiving RLC. The user process will then continue sending SDUs as before.

In the event of an unrecoverable error, the sending RLC informs the user process that both RLC entities should be terminated. The user process will reset both RLC entities and subsequently restart the data transfer.

It should also be noted that our user process does not configure the underlying RLC on-the-fly, although this capability exists in the standard.

4.4 Modelling with Maria

Maria was chosen as the analyzer for three reasons: First, Maria contains built-in data types which are amenable to protocol modelling. These include queues, unions, and structures. PDUs with several data fields can be modelled exactly as each PDU can be declared as a structure in Maria language. Moreover, several PDU types can be grouped together under a single PDU type which is declared as a union of the aforementioned structures. Naturally, a communication channel is declared as a queue containing several messages of the just mentioned union type. With Maria, one can also test a queue for emptiness with a single operator.

Secondly, Maria is one of the most powerful Petri Net analyzers currently in existence, with the capability of handling state spaces up to 200 million on a fairly standard stand-alone PC. The power of Maria may be increased even more with parallelized reachability analysis which is now a feature of the tool.

Last, the Maria toolset is an open development environment, where a number of language specific front ends are envisaged to be included. Two such front ends are SDL front end and Java front end. The SDL front end will be able to perform the translation process from SDL to Maria using the principles explained in Section 4.1 automatically. At this point the development of modelling methodologies in the Maria project is advancing faster than the development of the tool, thus forcing one to make some translations manually.

5 Analysis

5.1 Requirements

The specification defines several services and functions for the RLC protocol. Our model implements *acknowledged data transfer service* and *notification of unrecoverable errors*. The specification refines the services to functions, which are needed to establish and support the service. However, the list of functions in the specification is deficient and a more exhaustive list can be found e.g. in [6]. From the following function list we derived the requirements for the verification task (for the functions present in our model):

Error detection and correction functionality is provided by using the retransmission mechanism and SDU discard procedure.
Duplicate detection guarantees that each SDU is delivered to the user at the receiving side at most once.
In-sequence delivery of SDUs preserves the order of SDUs in transmission.
Freedom from deadlocks is a generic requirement for all models

The three first mentioned ones constitute the requirement of realiable data transmission. The model contains an abstraction which could result in a false negative acknowledgement of an SDU. This property causes the sent and received data streams to be unsynchronized, but it is still compatible with the requirements in this chapter. Each of the properties is more carefully defined in the following.

– ERROR_DETECTED if the sending RLC is unable to transfer a PDU after predefined number of retries then the protocol will flush the SDU that the PDU was part of and inform the user of SDU's loss. This property ensures that when a positive acknowledgement is received, the transfer really occurred. It is allowed that a false negative acknowledgement occurs, but data loss is never ignored.

– DATA_INTEGRITY: the protocol does not duplicate data or lose data without issuing an error message to the user. Note that data corruption is not an issue as lower levels detect corruption and delete corrupt data, collapsing this case to data loss. The difference between this and the ERROR_DETECTED property is that this property will fail if data that has been acknowledged as transferred is lost. The ERROR_DETECTED property is assumed to hold before this property can be verified.
– DATA_SEQUENCE: SDUs are delivered in the same sequence as they were sent or in case of a lost SDU an error message is generated to the user. This assumes also that ERROR_DETECTED property holds.
– NO_DEADLOCK: the system is free of illegal deadlocks, where a deadlock state is a state without following states.

5.2 Expressing the properties

Data independence [11], [13] is a syntactic property of a system, stating that the values of certain data items do not affect the system's behaviour. Thus from the system point of view these data values can be arbitrarily changed as long as the change is consistent for corresponding values.

A system is data-independent with respect to a variable `x` if the only operations that `x` can be subjected to are:

– receiving its value from an external source
– sending its value to an external target
– copying its value from another data-independent variable

Especially `x` cannot be used for control flow decisions. By examining the RLC v. 3.5.0 specification, it is obvious that the RLC protocol is data-independent with respect to the user data.

Properties can be expressed as automatons representing the higher protocol layers of both peers possibly augmented with additional simple properties. The basic function of a user automaton is to provide a sequence of data items and also act as a data sink. Our adaptation of property encoding is similar to Wolper [13], but is based on Kaivola [7].

For the rest of the chapter the following naming is used: $sent$ is the sequence of SDUs that is transmitted by the sending RLC peer, where $a_1 a_2 \ldots a_{n_a}$ are individual SDUs of that sequence and n_a is the length of the sequence. Likewise $received$ is the sequence of SDUs that is sent by the receiving RLC peer to the receiving user, where $b_1 b_2 \ldots b_{n_b}$ are individual SDUs of that sequence.

Error detection The RLC protocol has some predefined upper limit for resends of a single PDU after which the upper layers are informed of SDU loss. The ERROR_DETECTED requirement can be expressed by requiring that each sent SDU is either positively or negatively acknowledged (ack or $nack$) and that a positive acknowledgement actually corresponds to a correct transfer. Note that a false negative acknowledgement is considered to be acceptable behaviour. Since RLC is data independent, it is sufficient to create a tester automaton that sends only one data value, zero, repeatedly and keeps track of the acknowledgements that are given back from the RLC. As the interface between RLC and the tester automaton is assumed to be stop-and-wait, each SDU is acknowledged before the next SDU is sent.

Note also that this assumes that the model can relate acks and $nack$s to a particular SDU. This is done by associating each SDU with an identifier that must be unique for SDUs that are being transferred by the protocol. This identifier is passed along with the protocol messages corresponding to ack and $nack$.

Data integrity If no error detection is present, the requirement DATA_INTEGRITY can be expressed as follows: at any moment the SDU sequence $received$ is a prefix of the SDU sequence $sent$. According to Kaivola this is achieved when the following property holds: if $sent$ belongs to the ω-regular language $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$ (where x^∞ is a shorthand for $x^* \cup x^\omega$) then $received$ belongs to the same language and the protocol has not issued an error message. If there is a violation of the safety requirement, then there must be some item in sequence $sent$ that is not in $received$ or $|received| > |sent|$ indicating duplication or loss.

If the protocol does not satisfy DATA_INTEGRITY, then there is a sequence of items in $received$ that is not a prefix of $sent$. At some point of the sequence at index n where $a_n \neq b_n$ or a_n does not exist ($n_a < n$).

It may be that this is because the protocol has erronously duplicated a data item of $sent$ that has been sent earlier on. This means that the smallest differing nth item on the $received$ side is equal to an earlier n'th ($1 \leq n' < n$) data item of sequence $sent$ and different from the nth item of sequence $sent$. Due to data independence the values of $sent$ and corresponding values of $received$ can be replaced and the produced sequences ($sent'$ and $received'$) represent an execution of the protocol. The data values are changed following these rules:

- globally replace the first $n' - 1$ data items of $sent$ with zeroes (this propagates over to $received$ as well)
- globally replace the n'th data value of $sent$ with 1
- globally replace the rest of data values in $sent$ with 2

Now it is clear that $sent'$ belongs to the w-regular language $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$, but $received'$ does not, because there are more than one 1 value.

The second possibility of an execution that does not satisfy DATA_INTEGRITY is similar to the previous case, but the differing element b_n, where $a_n \neq b_n$ does not occur in the $sent$ sequence (or there is no element a_n in $sent$ at all). This time change the data values to form $sent'$ and $received'$ as follows:

- gloablly replace first $n - 1$ values values with 0
- gloablly replace the nth item in $sent$ with 1 (only if a_n exists)
- globally replace the rest of data values in $sent$ with 2

Again $sent'$ belongs to the w-regular language $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$, but the nth item in $received'$ must be different than any value that occurs in the first n elements of $sent'$ and nth values of both sequences are different by assumption. Thus the only possible value must be 2 and $received'$ cannot belong in the same language as $sent'$.

We assume above that the protocol attempts to resend a PDU until the send eventually succeeds and that the channel is fair so that the resent PDU is eventually transferred. In our case the protocol gives up retransmission after some predefined number of tries, discards the related SDU and reports of SDU loss. This is considered to be legal behaviour. In order to take this into account, the accepting language must be modified so that a not-acknowledged message for a certain SDU is considered to be equal to succesful transfer of that SDU.

If positive acknowledgement for data i is called ack_i and negative acknowledgement for data i $nack_i$ then the accepting language can be rephrased as: $(0 \cdot ack_0 \cup nack_0 \cup 0 \cdot nack_0)^\infty \cdot ((1 \cdot ack_1 \cup nack_1 \cup 1 \cdot nack_1) \cdot (2 \cdot ack_2 \cup nack_2 \cup 2 \cdot nack_2)^\infty \cup \epsilon)$. The first possible outcome for sending data i is that it is transferred correctly and acknowledged, the second possibility is that no transfer is completed and a negative acknowledgement is given. The third possibility is that negative acknowledgement is given, but the data is actually transferred. This may occur if a timer on the sending side fires before the receiving peer has been able to acknowledge the transfer. This false negative is acceptable behaviour and can be considered equal to a successful transfer of data i.

This relies on the possibility of relating acks and $nack$s to a particular SDU.

The second assumption is made for resetting of the tester automaton. If the RLC protocol notices a fatal error in transferring signaling messages, then the protocol performs a hard reset. This must also be reflected on the user automaton side: protocol hard reset also resets the tester automaton to its initial state.

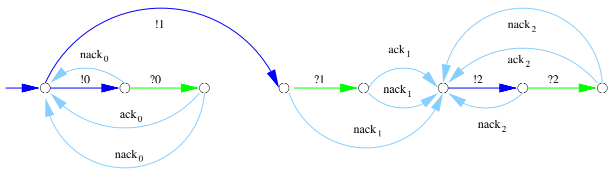

Fig. 4. Tester automaton for DATA_INTEGRITY property.

The tester automaton for this property is given in figure 4. Sending and receiving are denoted by ! and ? respectively. If some action is not possible, the automaton deadlocks signalling an error. For clarity the picture does not show the hard reset transition to the initial state, which is possible in all states. It should be noted that this tester automaton is connected to both the sending and the receiving RLC in the Maria model. This choice was made since the language containment problem (solved by the single tester automaton) would have been much more complex to solve if there had been two separate user processes on top of sending RLC and receiving RLC.

In-sequence delivery Property DATA_SEQUENCE can also be expressed using the same property as in case of DATA_INTEGRITY. If some data elements are not delivered in sequence, then there must be a situation where the $sent$ and $received$ elements differ because the sent a_n is overtaken by one or more sent elements and thus $a_n \neq b_n$. a_n appears on the $received$ at $n' > n$.

The data values can be changed using the rules as in the second case of DATA_INTEGRITY and once again $sent'$ and $received'$ differ.

It is also possible to express the DATA_SEQUENCE property using language $0^\infty \cdot 1^\infty$ with only two data values.

Table 1. State space sizes for analysis. CH_1 stands for lossy channel, CH_2 stands for lossy and reordering channel and CH_3 stands for lossy, reordering and duplicating channel.

	CH_1	CH_2	CH_3
ERROR_DETECTED	23 825 states, 32 842 arcs	24 483 states, 33 612 arcs	147 070 states, 193 628 arcs
DATA_INTEGRITY	65 636 states, 89 883 arcs	66 398 states, 90 757 arcs	393 621 states, 516 560 arcs

Freedom from deadlocks Both the DATA_INTEGRITY and DATA_SEQUENCE properties are verified by having user automatons interact with the RLC protocol. The behaviour of the automatons consists of providing the protocol with some data for transfer, receiving the transferred data, and the related acknowledgments of data transfer. This behaviour is repeated forever and is only stopped if an undesired message is received, which causes the automaton to deadlock. A hard reset of the protocol causes the user automaton to reset to its initial state, but execution is otherwise not interrupted.

If the analysis stops in a deadlock state that does not belong to the states of the user automaton, then that state must be in the RLC protocol. Thus a successful analysis with any of the presented automatons implies freedom of deadlocks in the RLC protocol. Automatic deadlock and error state detection are provided by the Maria model checker.

5.3 Verification strategy

Before starting up the actual verification task, we divided the whole task into subtasks of manageable size and defined the verification strategy. Successful verification of some properties can be used as an assumption for subsequent verifications. Also, for each requirement to be verified, we considered the appropriate channel model and configuration of the protocol.

As described in the previous section, we used one tester automaton for the verification of several properties. The tester automaton we have constructed relies heavily on the idea that the protocol acknowledges all SDUs (*ack* or *nack*) to the user. Therefore we verified the ERROR_DETECTION property first and used the result as an assumption for later verifications. Also, validity of the NO_DEADLOCK property is a prerequisite for all our verification runs. All errors appear as deadlocks in the verification. Vice versa, this means that a successful verification of some property implies always the validity of the NO_DEADLOCK property.

In the first verification runs we used reliable channel and the *minimum configuration* of the protocol model. In the minimum configuration there are no resendings of PDUs, *mrw* messages or *reset* messages. Window size at both the sending and receiving sides is limited to one.

After several iterations (verifying and correcting the model) we were able to verify the ERROR_DETECTION property with a lossless channel. In the next phase we verified the same property with the lossy channel, which is the relevant one for this property.

We started also the verification of the DATA_INTEGRITY property with reliable channel. We were able to verify the property with the lossy and reordering channel, but when augmenting the channel model with duplication generation, we ended up with state explosion. To overcome this reversal, we restricted the state space by modeling the sending and receiving as atomic actions and then introducing an implicit scheduling mechanism for them. We have described the mechanism in section 4.1. Finally, we were able to verify also the DATA_INTEGRITY property with the lossy, reordering and duplicating channel.

5.4 Results

Our verification effort did not uncover mistakes in the RLC protocol for the chosen parameters. Instead, we could prove the ERROR_DETECTED, DATA_INTEGRITY, DATA_SEQUENCE and NO_DEADLOCK properties. This was to be expected since the protocol is, after all, fairly standard and some verification effort has already been invested in it [8], [4]. The only significant difference between a standard sliding window protocol implementation and the RLC was the inclusion of SDU discard functionality. The verification runs showed that SDU discard functionality did not introduce errors into the protocol. The results of the verification runs are shown in table 1. Verification was done by first executing the system with the user automaton for ERROR_DETECTED. The last three properties could then be verified using a common user automaton. It is worth mentioning that the original versions of SDU discard did contain errors, but these were corrected during the standardization phase and before the Release 99 of RLC specification.

To gain more certainty of the correct functioning of the protocol, it is necessary to perform the analysis for transmission window sizes greater than 1. In our model the window size is a parameter, which was initially 2. Since we are dealing with a real-life protocol with a huge state space, we encountered, as already expected, a state explosion. To be able to proceed we had to restrict several parameters. In the next phase we will remove the abstractions step-by-step and thus increase our confidence to the verification results.

6 Conclusions

Modelling and verification of telecommunication protocols can be used as a method to simulate and test the standardized protocol specification before starting up the high-volume implementations. To keep the focus in the core functionality and especially to be able to manage with the verification task, the abstraction level of the model should be high enough. On the other hand, the abstractions should not violate the model with respect to the properties to be verified. This requires intelligent modelling, where known heuristics can be used.

We have modelled the 3GPP RLC protocol and verified the most fundamental properties for establishing the reliable transmission of user data. The properties are defined based on the standardized protocol specification. We have used the SDL specification included in the standard as a starting point for our High Level Petri Net model. The protocol model was extremely large, so we have used various types of abstractions to alleviate the state space explosion. Based on this work and also on our earlier experience we have introduced some heuristics to prepare ground for an intelligent modelling method. Part of the heuristics are specific for the underlying modelling formalism, but some of them can be applied more generically.

References

1. 3GPP. *RLC protocol specification 3.5.0*, 2000.
2. Yifei Dong, Xiaoqun Du, Y.S. Ramakrishna, C.R. Ramakrishnan an d I.V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, and Eugene W. Stark a nd David S. Warren. Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools. In *TACAS'99*, 1999.
3. J. Ellesberg, D. Hogrefe, and A. Sarma. *SDL: Formal Object-Oriented Language for Communicating Systems*. Prentice Hall, 1997.
4. Juhana Helovuo and Sari Leppänen. Exploration Testing. In *ICACSD 2001, 2nd International Conference on Application of Concurrency to System Design*, pages 201–210, 2001.
5. International Standards Organization. *High-level Petri Nets — Concepts, Definitions and Graphical Notation. Final Draft International Standard ISO/IEC 15909, Version 4.7.1*, 2000. The standard is available on the web, http://www.daimi.au.dk/PetriNets/st andardisation.
6. H. Kaaranan, A. Ahtiainen, L. Laitinen, S. Naghian, and V. Niemi. *UMTS Networks: Architecture, Mobility and Services*. Wiley & Sons Ltd, 2001.
7. R. Kaivola. *Equivalences, Preorders and Compositional Verification for Linear Time Temporal Logic and Concurrent Systems*. PhD thesis, Helsingin yliopisto. Tietojenkäsittelytieteen laitos, 1996.
8. Sari Leppänen and Matti Luukkainen. Compositional Verification of a Third Generation Mobile Communication Protocol. In *International Workshop on Distributed System Validation and Verification, ICDCS 2000 Workshop. IEEE Computer Society*, 2000.
9. Marko Mäkelä. A Reachability Analyzer for Algebraic System Nets. Research Report A69, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2001.
10. Markus Malmqvist. *Methodology of Dynamical Analysis of SDL Programs Usi ng Predicate/Transition Nets*. Technical report B16, Helsinki University of Technology, Digital Systems Laboratory, April 1997.
11. Krishan Sabnani. An Algorithmic Technique for Protocol Verification. *IEEE Transactions on Communications*, 36(8):924–931, August 1988.
12. A. Toskala and H. Holma. *WCDMA for UMTS, Radio Access for Third Generation Mobile Communications*. Wiley & Sons Ltd, 2000.
13. Pierre Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, 1986.