

Formal Specification and Verification of the Multiparty Call in ATM UNI protocol

Gordan Jezic
gordan.jezic@fer.hr
Department of Telecommunications
Faculty of Electrical Engineering and Computing
University of Zagreb
Zagreb, CROATIA, HR-10000

Abstract

This paper presents formal specification and verification of the multiparty call in ATM UNI signalling protocol. The protocol specification is written in the CCS process algebra (Calculus of Communicating Systems). Verification is done by using the NCSU-Concurrency Workbench. Due to the complexity of the protocol, the model is decomposed into three components. The protocol is verified by the model-checking feature of the Workbench. Each of its components is checked for safety and liveness properties using temporal CTL (Computation Tree Logic) and modal mu-calculus logic.

1. Introduction

A typical protocol design process includes a specification, which is either an informal description or a partially formal description in a protocol description language, and an implementation that is tested extensively for bugs. The bugs in the final implementation could appear either because of implementation errors or design errors. Implementation errors can be found easily. Design errors, however, are not only subtle and hard to trace, but also quite expensive to correct as they require going back to the design phase. Therefore, it is necessary to apply the techniques that detect the errors early at the early design stage. One of them are formal methods.

This paper presents formal specification and verification of the multiparty call in ATM UNI signalling protocol. The protocol specification is written in the CCS process algebra (Calculus of Communicating Systems). Verification is done by the NCSU-Concurrency Workbench [4, 5].

The model of the protocol includes the specification of the procedures for point-to-multipoint connections and model includes three users: one calling and two called users. Two users enjoy the same rights: both

can initiate the basic call (point-to-point) and have of a third party join the call.

Due to the complexity of the protocol, the model is decomposed [8-10] into three components. The first one covers all procedures for point-to-point call. The second includes the procedures for a point-to-multipoint call without clearing point-to-point connection. In that case, after setting up a point-to-point connection, only one additional user can be added to the connection. These two components represent two possible “flows” of the protocol, because after establishing a point-to-point connection, it is possible either to release a point-to-point call or to connect a new party to it. The third component consists of the procedures for a point-to-point and a point-to-multipoint connection with a restriction to one passive user. A passive user can only be a called user. The third component includes one calling, one called and one added user. Thus, the relations between the first two components are checked: the procedures of releasing a point-to-point call with the procedures for connecting and disconnecting an added user.

The protocol is verified by using the model-checking feature of the Workbench. Its each component is checked for safety and liveness properties using the temporal CTL (Computation Tree Logic) and the modal mu-calculus logic.

The paper is organised in the following way. Informal description of the model, model architecture and message flows for successful establishing and clearing of a multiparty call are presented in Section 2. Formalization of the protocol, components of the model as well as model assumptions are described and presented in Section 3. Decomposition of the protocol and verification results are reported in Section 4. Conclusions are given in Section 5.

2. Model Description

The procedures for establishing and clearing of a multiparty call are defined by the User-Network Interface (UNI 3.1) signalling protocol [1]. It is

assumed that there are three users or ATM end-stations, each connected to the ATM network through a user network interface (UNI). Two users participate in a point-to-point call and a third one can join the connection at the request of the calling (root) user (Figure 1). The UNI has two components: one defines the control procedures for the user side (USI) and the other defines the control procedures for the network side (NSI). The USI and the NSI are assumed to be connected by means of a reliable network and that the underlying ATM network is also reliable and error-free.

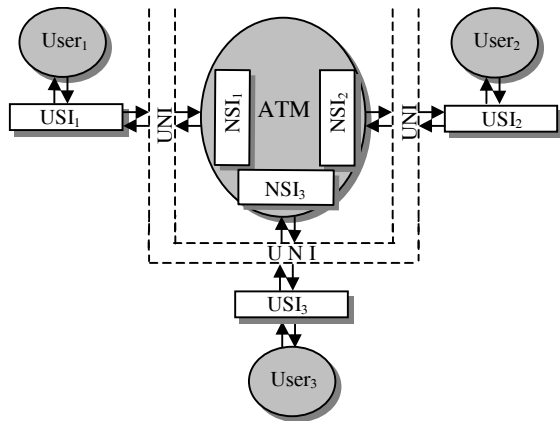
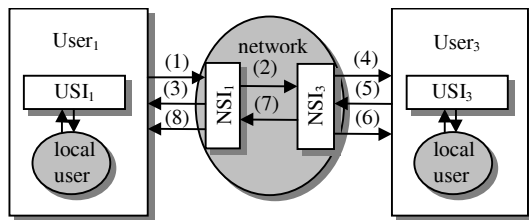


Figure 1. Architecture of the model

A multiparty call is set up by first establishing a point-to-point connection between a calling and a called user. After this set up is complete, an additional user (party) can be added to the connection by an ADD_PARTY request from the calling user. A party may be connected to or disconnected from a multiparty call at any time after the connection is established. A party can be added to an existing point-to-point connection only via the calling user who issues an ADD_PARTY message, or be dropped from a connection at the request of either the calling user or himself (but not by the other user).

A typical execution sequence for a successful addition of a third party to an already existing (point-to-point) call is as follows (Figure 2).

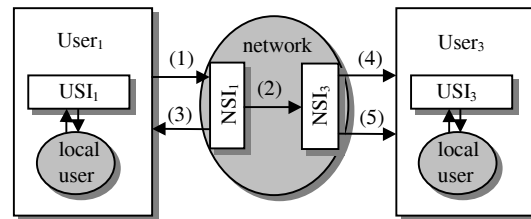


- | | |
|-----------------------|---------|
| ADD PARTY | (1) (2) |
| CALL PROCEEDING | (3) |
| SETUP | (4) |
| CONNECT | (5) (7) |
| CONNECT ACKNOWLEDGE | (6) |
| ADD PARTY ACKNOWLEDGE | (8) |

Figure 2. Message flow for successful addition of a party

The term *User* refers to the combination of a local user and the USI. A calling user initiates the addition of a new user (party) and sends an ADD_PARTY request to the NSI₁. Index 1 refers to a calling side, index 2 to a called side and index 3 to a added (party) side. The NSI₁ forwards the request across the network and sends a CALL_PROCEEDING message back to the calling user. When the NSI₃ (party side) receives the request, it forwards the request to the party as a SETUP message. If the party responds with a CONNECT message, the NSI₃ responds with a CONNECT_ACKNOWLEDGE message and sends a CONNECT message to the NSI₁. The NSI₁ then sends an ADD_PARTY_ACKNOWLEDGE message to the calling user. When the calling user receives this message, the multiparty call is established.

An execution sequence for dropping of a party from an existing multiparty call by the calling user is as follows (Figure 3).



- | | |
|------------------------|---------|
| DROP PARTY | (1) (2) |
| DROP PARTY ACKNOWLEDGE | (3) |
| RELEASE | (4) |
| RELEASE COMPLETE | (5) |

Figure 3. Message flow for dropping of a party out

3. Formal Specification

Formal model of establishing and clearing of a multiparty call is written in the process algebra CCS (Calculus of Communicating Systems) [2, 3]. The terms in it are described by the following grammar (1).

$$P ::= \text{nil} \mid \alpha.P \mid (P + P) \mid (P \mid P) \mid P[f] \mid P \setminus L \mid \text{proc } C = P \quad (1)$$

“.” represents the prefixing operator; “+” is the summation of choice operator; “|” is the parallel composition operator; “f” is a relabelling function; “\” is the restriction operator, and “proc =” is used for defining processes.

The model consists of 13 processes or components which are executed simultaneously; therefore they are connected by a parallel composition operator. The model is added to the model for establishing and clearing a point-to-point call [7]. Given that only a calling user can add a party to the call, the model of a multiparty call must distinguish a calling and a called user. In this case, the model consists of three users. Two users have the same rights and both of them can

initiate the point-to-point call and then can realise the multiparty call. A third user is a party user and can be only an added user. Thus, after establishing a point-to-point call, a calling user can either clear the call or initiate the procedures for establishing the multiparty call. If a calling user clears a point-to-point call, the model returns to the starting state. In the other hand, after having added a party to the call, either a calling or an added user can disconnect a party.

To establish a multiparty call, the model must have the following components:

1. the models of the local party, local calling and local called user,
2. the models of the USI on the party side, calling side and called side,
3. the models of the NSI on the party side, calling side and called side,
4. the models of UNI channels (between USI and NSI) on the party side, calling side and called side, and the network channels.

The components (models) of the calling and called side are equal these users having the same rights.

The model includes the following simplifying assumptions:

1. The channels are formalized as one-place buffers and all of them are reliable and error-free,
2. The contents (information elements) of a message are ignored,
3. The model includes point-to-point and point-to-multipoint calls, multipoint-to-multipoint calls are not considered and “calls” and “connections” are equivalent,
4. The model does not include timeouts and retransmissions.

3.1. Model of the USI on the calling side

After establishing of a point-to-point call, the USI from *Outgoing Call Proceeding* state enters to *Null* state where is possible to add a new user.

```
procNullAddUSI1=
add_req1.'add_req_USI1_send.AddPartyInitUSI1+
rel_req1.'rel_req_USI1_send.RelReqUSI1
```

In the *Null Add* state, the USI can receive either a request for clearing of a basic call or a request for adding a party. If it receives the request for adding of a party, an *ADD_PARTY_REQUEST* message, it forwards the message to the network side and then enters *Add Party Initiated* state.

```
procAddPartyInitUSI1=
call_proc_NSI1_rec.OutgoingCallUSI1+
add_reject_NSI1_rec.'add_err_USI1.NulAddUSI1
```

In the *Add Party Initiated* state, the USI waits for a response from the NSI. If the USI receives a *CALL_POCEEDING* message, then goes into the *Outgoing Call Proceeding* state. If the USI receives a message for rejecting of a party, an *ADD_PARTY_REJECT*, then sends an error message, an *ADD_PARTY_ERROR*, and goes back to the *Null Add* state. In the *Outgoing Call* state, the USI waits for a response from the NSI, again.

```
procOutgoingCallUSI1=
add_ack_NSI1_rec.'add_conf_USI1.ActAddUSI1+
add_reject_NSI1_rec.'add_err_USI1.NullAddUSI1
```

From the network side, the USI can receive an *ADD_PARTY_ACK* message and then sends a confirmation of successful adding of a party to the local user, an *ADD_PARTY_CONFIRMATION* message, and goes into the *Active* state. In the other hand, if it receives an *ADD_PARTY_REJECT* message, then sends an error message, an *ADD_PARTY_ERROR*, to the local user and enters the *Null* state.

```
procActAddUSI1=
drop_req1.'(drop_req_USI1_send.DropPartyInitUSI1+
drop_req_NSI1_rec.'drop_req_USI1_send.'drop_conf_
_USI1.NullAddUSI1)
+drop_req_NSI1_rec.'drop_ind_USI1.drop_resp1.'rel_
complete_USI1_send.NullAddUSI1
```

In the *Active* state, the USI can receive a request for dropping of a party either from the NSI or the local user. On receipt of a *DROP_PARTY_REQUEST* from the network side, the USI sends a *DROP_PARTY_INDICATION* to the local user, and after receiving of a *DROP_PARTY_RESPONSE* from the local user, it sends a *RELEASE_COMPLETE* message to the NSI. In the case that it receives the request from the local user, forwards it to the network side and goes to the *Drop Party Initiated* state.

```
procDropPartyInitUSI1=
drop_ack_NSI1_rec.'drop_conf_USI1.NullAddUSI1+
drop_req_NSI1_rec.'drop_conf_USI1.NullAddUSI1
```

In the *Drop Party Initiated* state, the USI waits for a response from the NSI. It can be either a confirmation for dropping, a *DROP_PARTY_ACK* message, or a request, a *DROP_PARTY_REQUEST* message, and then the USI sends a *DROP_PARTY_CONFIRMATION* to the local user and goes back to the *Null Add* state.

3.2. Model of the USI on the party side

In the *Null* state, the local party can receive a *SETUP_REQUEST* message from the NSI.

```
procNullPartyUSI3=setup_req_NSI3_rec.'setup_ind_
USI3.AddPartyRecUSI3
```

On receipt of the message, it sends a SETUP_INDICATION message to the local party and enters the *Add Party Received* state.

```
procAddPartyRecUSI3=
setup_err3.'add_reject_USI3_send.NullPartyUSI3+
'call_proc_USI3_send.IncomingCallUSI3+
setup_resp3.'connect_USI3_send.ConnReqPartyUSI3
```

In the *Add Party Received* state, if the USI receives a SETUP_RESPONSE message from the local party then a connection can be established immediately. In this case it sends a CONNECT message to the NSI and enters the *Connection Request* state. If a SETUP_ERROR is received from the local party, then the connection cannot be set up. In this case the USI responds to the NSI with a ADD_PARTY_REJECT message and goes back to the *Null* state. If the connection cannot be established immediately, the USI sends a CALL_PROCEEDING message to the NSI and enters the *Incoming Call Proceeding* state.

```
procIncomingCallUSI3=
setup_resp3.'connect_USI3_send.ConnReqPartyUSI3+
setup_err3.'add_reject_USI3_send.NullPartyUSI3
```

In the *Incoming Call Proceeding* state, if the USI receives a SETUP_RESPONSE message from the local party then a connection can be established immediately. In this case it sends a CONNECT message to the NSI and enters *Connection Request* state. If a SETUP_ERROR message is received from the local party, then the connection cannot be set up. In this case the USI responds to the NSI with a ADD_PARTY_REJECT message and goes back to the *Null* state.

```
procConnReqPartyUSI3=
connect_ack_NSI3_rec.ActDodUSI3
```

In the *Connection Request* state, the USI waits for a CONNECT_ACK message. On receipt of the message it enters the *Active* state.

```
procActPartyUSI3=
rel_req3.(rel_req_USI3_send.DropPartyInitUSI3+
rel_req_NSI3_rec.'rel_req_USI3_send.'rel_conf_
USI3.NullPartyUSI3)+
rel_req_NSI3_rec.'rel_ind_USI3.rel_resp3.'rel_
complete_USI3_send.NullPartyUSI3
```

In the *Active* state, the USI can receive a RELEASE_REQUEST from the NSI or from the local party. On receipt of the message from the local party, it forwards the request to the network side and enters the *Drop Party Initiated* state. If it receives a RELEASE_REQUEST message from the network side, then it forwards the request to the local party. On receiving a RELEASE_RESPONSE from the local party, it sends a RELEASE_COMPLETE to the network side and goes back to the *Null* state.

```
procDropPartyInitUSI3=
rel_req_NSI3_rec.'rel_conf_USI3.NullPartyUSI3+
```

```
rel_complete_NSI3_rec.'rel_conf_USI3.NullParty
USI3
```

In the *Drop Party Initiated* state, the USI waits for a message from the network side. After receiving either a RELEASE_REQUEST or a RELEASE_COMPLETE message, the USI sends a confirmation to the local party (RELEASE_CONFIRMATION message) and enters *Null* state.

3.3. Model of the NSI on the calling side

After establishing of a point-to-point call, the NSI enters the *Null* state where can either release the basic call or add a party to the connection.

```
procNullAddNSI1=add_req_USI1_rec.Validate1+
rel_req_USI1_rec.'release_pdu_NSI1_send.'rel_
complete_NSI1_send.NullNSI1
```

If the NSI receives a request for adding of a party, an ADD_PARTY_REQUEST message, then enters the *Validate* state. Otherwise, if it receives a request for clearing of the point-to-point call, it forwards the request across the network and enters the *Null* state.

```
procValDod1
='call_proc_NSI1_send.OutgoingCallProcNSI1+
'add_reject_NSI1_send.NullAddNSI1
```

The NSI enters the *Validate* state from the *Null* state after receiving of an ADD_PARTY_REQUEST from the USI. If the request is valid, it sends a CALL_PROCEEDING message back to the USI and enters the *Outgoing Call Proceeding* state. Otherwise, it sends an ADD_PARTY_REJECT message back to the user.

```
procOutgoingCallProcNSI1=
'add_pdu_NSI1_send.OutgoingCalINSI1+
'add_reject_NSI1_send.NullAddNSI1
```

In the *Outgoing Call Proceeding* state, the NSI can either send an ADD_PARTY_PDU message across the network and then enters the *Outgoing Call* state or send an ADD_PARTY_REJECT to the user side.

```
procOutgoingCallINSI1=
add_pdu_NSI3_rec.'add_ack_NSI1_send.ActNSI1+
reject_pdu_NSI3_rec.'add_reject_NSI1_send.
NullAddNSI1
```

In the *Outgoing* state, the NSI waits for a response from the other side of the network. If it receives an ADD_PARTY_PDU, it sends ADD_PARTY_ACK message to the USI and enters the *Active* state. If it receives an ADD_PARTY_REJECT message from the other side, it forwards the message to the USI goes back to the *Null* state.

```
procActAddNSI1=
drop_pdu_NSI3_rec.(drop_req_NSI1_send.
DropPartyRecNSI1+
drop_req_USI1_rec.'drop_ack_NSI1_send.
NullAddNSI1)+
```

```
drop_req_USI1_rec.'drop_pdu_NSI1_send.
'drop_ack_NSI1_send.NullAddNSI1+
drop_pdu_NSI3_rec.'drop_req_NSI1_send.
NullAddNSI1)
```

In the *Active* state, the NSI can receive a request for dropping the party from either the USI or from the other side of the network. In the first case, the NSI forwards the request to the other side of the network, a DROP_PARTY_REQUEST message, and sends a DROP_PARTY_ACK message to the USI. If the NSI receives the request from the other side of the network, it forwards the request to the USI and enters *Drop Party Received* state.

```
procDropPartyRecNSI1=
rel_complete_USI1_rec.NullAddNSI1+
drop_req_USI1_rec.NullAddNSI1
```

In the *Drop Party Received* state, the NSI waits for a message from the USI, a RELEASE_COMPLETE or a DROP_PARTY_REQUEST message and goes back to the *Null* state.

3.4. Model of the NSI on the party side

In the *Null* state, the NSI waits for a call from one of the user.

```
procNullPrtyNSI3=
add_pdu_NSI1_rec.'setup_req_NSI3_send.
CallPresentNSI3+
'reject_pdu_NSI3_send.NullPartyNSI3)+
add_pdu_NSI2_rec.'setup_req_NSI3_send.
CallPresentNSI3+
'reject_pdu_NSI3_send.NullPartyNSI3)
```

After receiving an ADD_PARTY_PDU message, the NSI can either reject the request with an ADD_PARTY_REJECT message or accept the request with a SETUP message and then it enters the *Call Present* state.

```
procCallPresentNSI3=
connect_USI3_rec.'connect_ack_NSI3_send.'connect_
pdu_NSI3_send.ActPartyNSI3+
call_proc_USI3_rec.InCallProcNSI3+
add_reject_USI3_rec.'reject_pdu_NSI3_send.
NullPartyNSI3
```

In the *Call Present* state, the NSI waits for a response from the USI. If the USI responds with a CONNECT message, it responds with a CONNECT_ACK, sends a CONNECT_PDU across the network and enters the *Active* state. On receipt of a CALL_PROCEEDING message from the USI, it enters the *Incoming Call Proceeding* state. If it receives a ADD_PARTY_REJECT message, then it forwards the message across the network.

```
procInCallProcNSI3 =
connect_USI3_rec.'connect_ack_NSI3_send.'connect_
pdu_NSI3_send.ActPartyNSI3+
```

```
add_reject_USI3_rec.'reject_pdu_NSI3_send.
NullPartyNSI3
```

In the *Incoming Call Proceeding* state, if the NSI receives a CONNECT message from the USI, it responds with a CONNECT_ACK and sends a CONNECT across the network. If it receives an ADD_PARTY_REJECT message, it forwards the message across the network.

```
procActPartyNSI3=
rel_pdu_NSI1_rec.'rel_req_NSI3_send.
DropPartyRecNSI3+
rel_req_USI3_rec.'rel_complete_NSI3_send.
NullPartyNSI3)+
rel_req_USI3_rec.'rel_pdu_NSI3_send.'rel_complete
_NSI3_send.NullPartyNSI3+
rel_pdu_NSI1_rec.'rel_req_NSI3_send.
NullPartyNSI3+
rel_pdu_NSI2_rec.'rel_req_NSI3_send.
NullPartyNSI3)+
rel_pdu_NSI2_rec.'rel_req_NSI3_send.
DropPartyRecNSI3+
rel_req_USI3_rec.'rel_complete_NSI3_send.
NullPartyNSI3)
```

In the *Active* state, the NSI can receive a request for dropping of the party, a RELEASE_REQUEST message, either from the USI or from the other side of the network. If it receives the request from the USI, it forwards the message across the network. Otherwise, it forwards the request to the USI and enters *Drop Party Received* state.

```
procDropPartyRecNSI3=
rel_complete_USI3_rec.NullPartyNSI3+
rel_req_USI3_rec.NullPartyNSI3
```

In the *Drop Party Received* state, the NSI can receive either a RELEASE_REQUEST message or a RELEASE_COMPLETE message and then enters the *Null* state.

3.5. Model of the local calling user

In the *Null* state the calling user can either start clearing of the established point-to-point call with RELEASE_REQUEST message or start adding of a party with ADD_PARTY_REQUEST message.

```
procNullAdd1='add_req1.(add_conf_USI1.ActAdd1+
add_err_USI1.NullAdd1)+
'rel_req1.rel_conf_USI1.NullUser1
```

After sending of the request for clearing of the basic call, the calling user waits for a RELEASE_CONFIRMATION message and enters the *Null* state. After sending of the request for adding of a party, the calling user waits for response from the USI. If the calling user receives an ADD_PARTY_CONFIRMATION message, enters the *Active* state. Otherwise, if the calling user receives an

ADD_PARTY_ERROR, then it goes back to the *Null* state.

```
procActAdd1= 'drop_req1.drop_conf_USI1.NullAdd1
+ drop_ind_USI1.drop_resp1.NullAdd1
```

In the *Active* state, the calling user can either send a DROP_PARTY_REQUEST message to the USI or receive a DROP_PARTY_INDICATION message from the USI. In the first case it waits for a DROP_PARTY_CONFIRMED from the USI and on receipt of the message it goes to the *Null* state. In the second case, it sends a DROP_PARTY_RESPONSE and goes to the *Null* state again.

3.6. Model of the local party

In the *Null* state, the party waits for a call.

```
procNullParty3=
setup_ind_USI3.(setup_resp3.ActParty3+
'setup_err3.NullParty3)
```

After receiving of a SETUP_INDICATION message from the USI, the connection can be established if the party sends a SETUP_RESPONSE message back to the USI and then goes into the *Active* state. Otherwise, the party sends a SETUP_ERROR and goes to the *Null* state.

```
procActParty3 = rel_ind_USI3.'rel_resp3.NullParty3 +
'rel_req3.rel_conf_USI3.NullParty3
```

In the *Active* state, the party can either do a RELEASE_REQUEST or receive a RELEASE_INDICATION from the USI. In the first case it waits for a RELEASE_CONFIRMED from the USI and on receipt of the message it goes back to the *Null* state. In the second case, it sends a RELEASE_RESPONSE and goes to the *Null* state, too.

3.7. Model of the channels

The channels are modelled as one-place buffers. It is assumed that the channels are reliable and bidirectional. The formal model of the ATM network (*Net*) and the subnets connecting the two sides of the interfaces of the local users (*Subnet₁*) and the party (*Subnet₃*) are as follows. The model includes only the messages which are included in the procedures for establishing and clearing a multiparty call. All messages in the model exchanged on the subnet are of form *messagename_sender_send* or *messagename_receiver_rec*.

```
procNet=
add_pdu_NSI1_send.'add_pdu_NSI1_rec.Net+
add_pdu_NSI2_send.'add_pdu_NSI2_rec.Net+
reject_pdu_NSI3_send.'reject_pdu_NSI3_rec.Net+
connect_pdu_NSI3_send.'add_pdu_NSI3_rec.Net+
drop_pdu_NSI1_send.'rel_pdu_NSI1_rec.Net+
drop_pdu_NSI2_send.'rel_pdu_NSI2_rec.Net+
rel_pdu_NSI3_send.'drop_pdu_NSI3_rec.Net
```

```
procSubnet1=
add_req_USI1_send.'add_req_USI1_rec.Subnet1+
add_reject_NSI1_send.'add_reject_NSI1_rec.Subnet1
+add_ack_NSI1_send.'add_ack_NSI1_rec.Subnet1+
drop_req_USI1_send.'drop_req_USI1_rec.Subnet1+
drop_ack_NSI1_send.'drop_ack_NSI1_rec.Subnet1+
drop_req_NSI1_send.'drop_req_NSI1_rec.Sucelje1
```

```
procSubnet3 =
setup_req_NSI3_send.'setup_req_NSI3_rec.Subnet3+
add_reject_USI3_send.'add_reject_USI3_rec.Subnet3
+call_proc_USI3_send.'call_proc_USI3_rec.Subnet3+
connect_USI3_send.'connect_USI3_rec.Subnet3+
rel_req_USI3_send.'rel_req_USI3_rec.Subnet3+
rel_req_NSI3_send.'rel_req_NSI3_rec.Subnet3+
rel_complete_USI3_send.'rel_complete_USI3_rec.
Subnet3+
rel_complete_NSI3_send.'rel_complete_NSI3_rec.
Subnet3+
connect_ack_NSI3_send.'connect_ack_NSI3_rec.
Subnet3
```

4. Protocol Verification

This section presents some verification results. Verification is based on two important properties of the protocol: freedom from the deadlock and liveness. The first property is checked for the safety of the protocol. Checking of the second one proves that if nothing bad happens, a connection will be eventually established. The properties are verified using the model-checking feature in the Concurrency Workbench.

4.1. Model Decomposition

Due to complexity of the protocol, the model is decomposed into the following three components:

1. procedures for establishing and clearing of a point-to-point call,
2. procedures for establishing of a point-to-point call with the procedures for adding and dropping of a party,
3. procedures for establishing and clearing of a multiparty call with a restriction to one passive user in a point-to-point call.

The reason for protocol decomposition is a lack of memory and time wastage, given that over 40.000 states and transitions are checked.

Decomposition of the model will be presented by messages which are important in particular components (key messages). The model for establishing and clearing of a multiparty call with the corresponding messages is shown in Figure 4.

Two users establish a point-to-point call (SETUP₁ and SETUP₂) after which a calling user may clear the call (RELEASE₁ or RELEASE₂) or initiate the

procedures for adding a party (ADD_PARTY₁ or ADD_PARTY₂). Having added a party to the connection, a calling user or a party can initiate the procedures for dropping the party out (DROP_PARTY₁ or DROP_PARTY₂ and RELEASE₃).

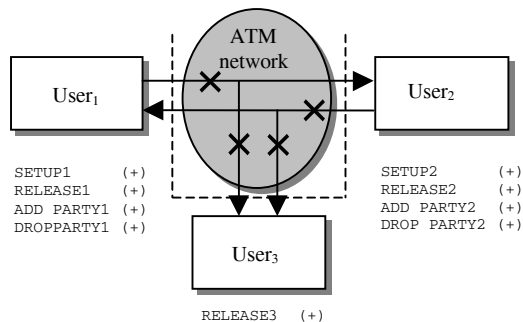


Figure 4. Key messages for a multiparty call

The first component covers all procedures for establishing and clearing of a point-to-point call (Figure 5). This component is chosen naturally, because all actions for a multiparty call are connected on point-to-point call.

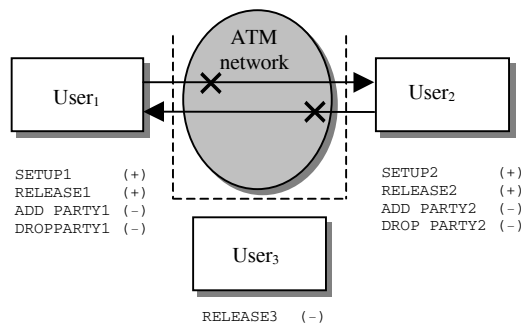


Figure 5. Key messages of the first component

There are two users that can establish (SETUP) and clear (RELEASE) point-to-point call. A party is not included.

The second component includes the procedures for establishing and clearing of a multiparty call without clearing a point-to-point connection (Figure 6). In that case, after setting up a point-to-point connection, only a party user can be added to the connection. These two components represent two possible “flows” of the protocol, because after establishing a point-to-point connection, it is possible either to release a point-to-point call or to add a party to the connection.

There are two users that can establish (SETUP) a point-to-point call and after that a calling user can add (ADD_PARTY) a third user (party), but can not clear a point-to-point call (RELEASE). After adding the party, the procedures of dropping the party can initiate the party out (RELEASE) or the calling user (DROP_PARTY).

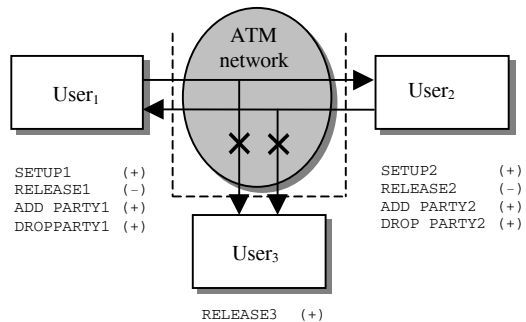


Figure 6. Key messages of the second component

Third component consists of the procedures for a point-to-point and a point-to-multipoint connection with the restriction to one passive user. In Figure 7 *User₂* is passive user and can be a called user only. This component defines the interrelations between the first two components and thus the procedures for clearing of a point-to-point call (RELEASE) with the procedures for establishing (ADD_PARTY) and dropping a party out (DROP_PARTY) are checked.

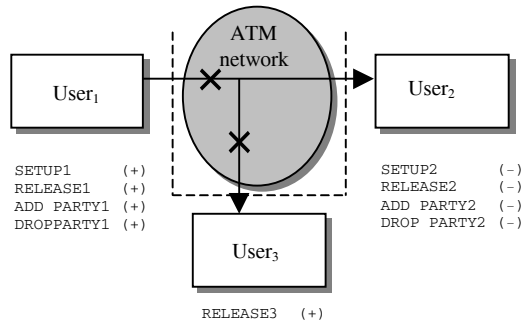


Figure 7. Key messages of the third component

There are three users: *User₁* is a calling user, *User₂* is a called (passive) user and *User₃* is a party. After establishing of a point-to-point call (SETUP₁), a calling user can clear this call (RELEASE₁) or add a party to the connection (ADD_PARTY₁).

4.2. Model Checking

The properties (deadlock and liveness) are specified in a temporal logic, known as the modal mu-calculus [6]. The syntax of mu-calculus is given by the following grammar (2):

$$\Phi ::= tt \mid ff \mid X \mid \neg\Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \langle\alpha\rangle\Phi \mid [\alpha]\Phi \mid \nu X.\Phi \mid \mu X.\Phi \quad (2)$$

The formula *tt* holds of every state, whereas the formula *ff* holds of no state. The formula $\Phi_1 \vee \Phi_2$ holds of a state if either Φ_1 or Φ_2 hold of the state; likewise $\Phi_1 \wedge \Phi_2$ holds of a state if both Φ_1 and Φ_2 hold of the state. α refers to actions. $\langle\alpha\rangle$ and $[\alpha]$ are referred to as path modalities. The modal formula $\langle\alpha\rangle\Phi$ holds of a state if the state has some α -derivative at which Φ holds, and $[\alpha]\Phi$ holds at a state if all α -derivatives of the state satisfy Φ . ν represents

the greatest fixpoint operator (*max*) and μ represents the least fixpoint operator (*min*).

The mu-calculus is extended to include the CTL (Computation Tree Logic) operators [6]. The CTL formulas and corresponding mu-calculus translations that were used in the verification (3, 4):

$$AG \text{ prop } \max X = \text{prop} \wedge [-\{\}]X \quad (3)$$

$$A(\text{prop}_1 \cup \text{prop}_2), \min X \text{ prop}_2 \vee (\text{prop}_1 \wedge [-\{\}]X \wedge <-\{\}>t) \quad (4)$$

Intuitively, *A* means that the property should hold for all computations, *G* means “always” and *U* means “until”.

Each component of the protocol is checked for safety (deadlock freedom) and liveness properties. The first property says that the protocol is free of deadlocks. The following mu-calculus formula expresses this property (5):

$$\text{can_deadlock} = \min X = [-\{\}]ff \vee <-\{\}>X \quad (5)$$

The second property requires introducing some new visible actions in the model. These actions are used for checking of the model behaviour and the model progress as expected. The new models for the local *User₁* and *User₂* (these models are the same) and added *User₃* are shown in Figure 8 and Figure 9. The visible actions are bolded.

```

procNullUser1=
'setup_req1.setup_request1.(setup_conf_USI1.
NullAddP1+ setup_err_USI1.setup_error1.NullUser1)
+setup_ind_USI1.('setup_resp1.setup_response1.
ActiveUser1+ 'setup_err1.setup_error1.NullUser1)

proc ActiveUser1 = rel_ind_USI1.rel_resp1.NullUser1

procNullAdd1=
'add_req1.add_request1.(add_conf_USI1.
add_confirmation1.ActAdd1+
add_err_USI1.add_error1.NullAdd1)+
rel_req1.release_request1.rel_conf_USI1. NullUser1

procActAdd1=
'drop_req1.drop_request1.drop_conf_USI1.NullAdd1
+ drop_ind_USI1.drop_resp1.NullAdd1

```

Figure 8. Model of a local user (*User₁* and *User₂*)

Each component includes only its visible actions, characteristic for this component, which are interesting for checking. In this case, for each component some corresponding property is being checked.

```

procNullParty3=
setup_ind_USI3.('setup_resp3.setup_response3.
ActParty3+ 'setup_err3.setup_error3.NullParty3)

procActParty3 = rel_ind_USI3.rel_resp3.NullParty3 +
'rel_req3.release_request3.rel_conf_USI3. NullParty3

```

Figure 9. Model of a party (*User₃*)

In the first component, a liveness of the procedures for establishing (6) and clearing (7) of a point-to-point call are checked.

$$\text{prop liveness_setup} = AG ([\text{setup_request}_1] A([\text{setup_error}_1]ff \wedge [\text{setup_error}_2]ff) U <\text{setup_response}_2>tt) \quad (6)$$

$$\text{prop liveness_release} = AG ([\text{setup_response}_2] A([\text{setup_request}_1]ff \wedge [\text{setup_request}_2]ff) U <\text{release_request}_1>tt) \quad (7)$$

In the second component, liveness of the procedures for adding (8) and dropping (9) of a party and a liveness of the procedure for establishing of a point-to-point call (6) too are checked.

$$\text{prop liveness_add} = AG ([\text{add_request}_1] A([\text{add_error}_1]ff \wedge [\text{setup_error}_3]ff) U <\text{setup_response}_3>tt) \quad (8)$$

$$\text{prop liveness_drop} = AG ([\text{add_confirmation}_1] A([\text{add_request}_1]ff U (<\text{drop_request}_1>tt \vee <\text{release_request}_3>tt))) \quad (9)$$

In addition checking (6) and (7), the third component includes checking of liveness when a calling user can concomitantly add a party or release a point-to-point call (10) (interrelation between first two components).

$$\text{prop liveness_add_release} = AG ([\text{setup_response}_2] A([\text{add_error}_1]ff \wedge [\text{release_request}_1]ff) U <\text{setup_response}_3>tt) \quad (10)$$

For example, the formula (8) says that if after *add_request₁* nothing bad happens (i.e. events *add_error₁* and *setup_error₃* do not occur), a *setup_response₃* takes place.

For the last property the model had 8.833 states and 30.186 transitions. Verification of the properties took about 3 minutes on a SUN Ultra 1 workstation with 256 Mbytes of memory.

5. Conclusions

Formal specification and verification of the multiparty call in ATM UNI signalling protocol are reported. The specification is written in the CCS process algebra and verification is done by using model-checking feature of the Concurrency Workbench. Due to complexity of the protocol, the model is decomposed into three components and each is checked for safety and liveness properties using temporal logic CTL and modal mu-calculus logic.

In the protocol design, there was a constant interaction between the modelling and verification phases. Therefore, in reality, the border between these two phases is being lost.

Further studies would be a more detailed modelling including timeouts and retransmissions, modelling of more complex protocols with more users, possibility of negotiations and mobile agents.

References

- [1] The ATM Forum, *ATM User-Network Interface (UNI) Specification, Version 3.1*, Prentice-Hall International, 1995.
- [2] Milner, R., *A Calculus of Communicating Systems*, Springer-Verlag, 1980.
- [3] Milner, R., *Communication and Concurrency*, Prentice-Hall International, 1989.
- [4] Cleaveland, R., Parrow, J., Steffen, B., The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems, *ACM Transactions on Programming Languages and Systems*, January 1993., Vol. 15, No. 1, pp. 36-72.
- [5] Cleaveland, R., Sims, S. T., Generic Tools for Verifying Concurrent Systems, *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design*, Zagreb, June 1997., pp. 3-8.
- [6] Cleaveland, R., Sims, S., *The Concurrency Workbench of North Carolina, User's Manual*, September 1996., North Carolina.
- [7] Bhat, G., Cleaveland, R., "Verifying the ATM UNI 3.1 Signalling Protocol, *Personal communication*, 1997.
- [8] Holzmann, G. J., Protocol Design: Redefining the State of the Art, *IEEE Software*, January 1992, pp. 17-22.
- [9] Lin, F. J., Liu, M. T., Protocol Validation for Large-Scale Applications, *IEEE Software*, January 1992, pp. 23-26.
- [10] Hailpern, B. T., Owicki, S. S., Modular Verification of Computer Communication Protocols, *IEEE Transactions on Communications*, January 1983., Vol. 31, No. 1, pp. 56-68.