Centrum voor Wiskunde en Informatica

Verifying a smart design of TCAP; A synergetic experience

T. Arts, I.A. van Langevelde

Software Engineering (SEN)

SEN-R9910 April 30, 1999

Report SEN-R9910
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Verifying a Smart Design of TCAP

a synergetic experience

Thomas Arts
email: thomas@cslab.ericsson.se
Computer Science Laboratory, Ericsson Utvecklings AB
PO Box 1505, 125 25 Stockholm, Sweden

Izak van Langevelde
email: izak@cwi.nl
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
An optimisation of the SS No. 7 Transport Capabilities Procedures is verified by specifying both the original and the optimised TCAP in μCRL, generating transition systems for both using the μCRL tool set, and checking weak bisimulation equivalence of the two using the Cæsar/Aldébaran tool set, these steps being part of a iterative process of specification, refinement and verification. As a result, the optimisation design is debugged, a deeper understanding of the protocol is gained, and the usability of the μCRL tool set is evaluated. In conclusion, the design of an optimised TCAP indeed benefitted from the verification reported here, and μCRL and Cæsar/Aldébaran appeared to be a largely adequate combination for the verification at hand; however, since the characteristics of TCAP were not explicit from the start, and the tools used covered the functionality required not perfectly, the verification required a good deal of human ingenuity and stamina.

1991 Mathematics Subject Classification: 68Q60 Specification and verification of programs
1991 Computing Reviews Classification System: B4.4 Performance analysis and design aids; D.2.1 Requirements/Specifications; D.2.4 Program Verification
Keywords and Phrases: TCAP protocol, μCRL, process algebra, specification, verification
Note: Izak van Langevelde was supported by the Systems Validation Center, a joint project of the University of Twente, CWI and Telematics Institute (http://fmt.cs.utwente.nl/projects/SVC-html)

1. Introduction

This report describes the verification of an optimised design of TCAP, a protocol for advanced intelligent network services [11]. Initially, the goal of the verification was to establish equivalence of the original and the optimisation, but as verification proceeded the process brought to light not only the usual minor design errors, but also a number of assumptions about the intended use of TCAP that had remained implicit before. Instead of being just one stage concluding the design process, the verification appeared to be an iterative process, intensely interacting with the design process, drawing more and more information from the design, providing more and more feedback, resulting in a series of designs and specifications of increasing accuracy. It is this process of debugging, learning and polishing that is reported in this text.

The need for verification arose when the original PLEX[1] implementation of TCAP in a parallel architecture, was succeeded by an Erlang ([2, 5]) implementation for a single-processor architecture.

[1]PLEX is a proprietary language of Ericsson Telecom AB.

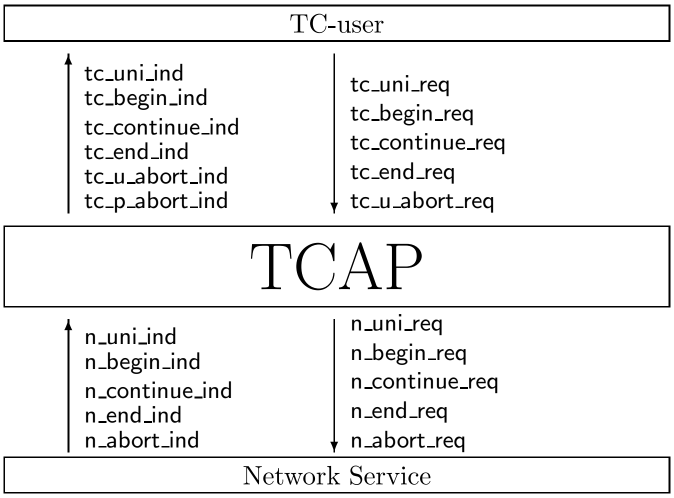


Figure 1: TCAP layer within SS No. 7 protocol

The opportunity for optimisation offered by the transition was taken, but the question whether the optimised TCAP was equivalent to the original remained open.

The answer to this question was found in process algebra. Both the original and the optimised TCAP were specified in μCRL [7] a language rooted in process algebra with data. Using the μCRL tool set, state spaces were generated for the two, where after these were compared using Aldébaran [6, 9]. Differences found were then traced back into the μCRL specification and corrected until, finally, the two specifications were exactly equivalent.

The organisation of the report is as follows. Section 2 describes the TCAP protocol to be verified, and Section 3 describes μCRL tools to be utilised for the specification and verification. In Sections 4 and 5 it is described how the original and the optimised TCAP are specified in μCRL. Section 6 describes how the environment, i.e. the context in which TCAP is used, is specified. Then Section 7 describes the actual verification process. Section 8 describes the feedback from the verification to the TCAP design and Section 9 evaluates the use of μCRL for this verification project. Finally, Section 10 describes the conclusions of the verification, as well as questions raised by this study. Three appendices present the full and final specifications used.

2. Transport capabilities

The protocol for Transaction Capabilities Procedures (TCAP, the standard is described in ITU-T Recommendation Q.774 [11][2]) enables the deployment of advanced intelligent network services in a telecommunication network. TCAP is, for example, used to determine the routing numbers associated to toll-free numbers, to check personal identification numbers of calling card users, and for authentication, equipment identification and roaming of GSM phone users.

Services that use this protocol are called *tc-users*. A tc-user interacts with TCAP by sending messages. These messages are interpreted by TCAP and passed along to the network that transports it to another TCAP. The latter TCAP receives the message, interprets it and sends it to its tc-user.

TCAP messages consist of two parts, a transaction portion with the package identifier and a component portion with the application specific data. For our verification purposes we ignore the component part and we only concentrate on the five transaction primitives that are possible, viz. *unidirectional*, *begin*, *continue*, *end*, and *abort*. TCAP can be seen as a layer between the tc-user and the network. Names of messages between tc-user and TCAP are prefixed by *tc_*, whereas messages between TCAP and the network are prefixed by *n_*. Messages from the tc-user to TCAP are called *requests* and messages

[2]We considered the 1993 version of TCAP, since this was the one used for the implementation.

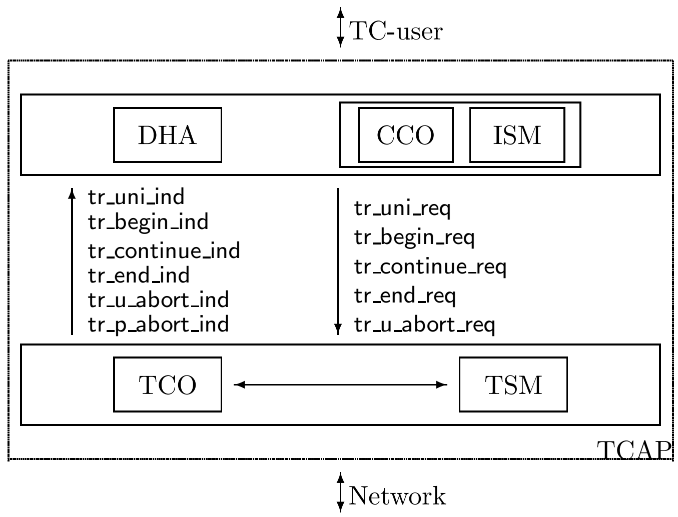


Figure 2: TCAP internal structure

from TCAP to the tc-user are called *indications*, therefore suffixed by *_req* and *_ind* respectively. Similar, messages from TCAP to the network are called requests and from the network to TCAP are called indications (see Figure 1). A request sent to the network will appear as an indication at the side it has been sent to. Note that this distinction between requests and indications is an artificial extension for specification and verification purposes. TCAP itself is specified by two layers with two and three finite state machines, respectively (see Figure 2). A TCAP implementation is connected via the network with another TCAP implementation. Both implementations need to be implemented according to the standard [11], such that they are completely transparent with respect to messages from tc-user and network side. A typical scenario of this communication is depicted in the message sequence chart of Figure 3, where tc-user A requests a communication with B, after which a continuation message follows. Continuation messages contain application specific data, but we abstract from the contents of the messages.

The optimisation concentrates on the state machines DHA, TCO and TSM that deal with the transaction portion of the messages. The basic idea of the optimisation is that of changing the design in such a way that no messages need to be send between the two layers in TCAP. The original specification has been designed with a PLEX implementation in mind in which one has a large degree of parallelism and in which sending of messages is comparable with direct jumps to the receiver. On a one-processor architecture with a language such as ERLANG in which message sending is a little less efficient, the

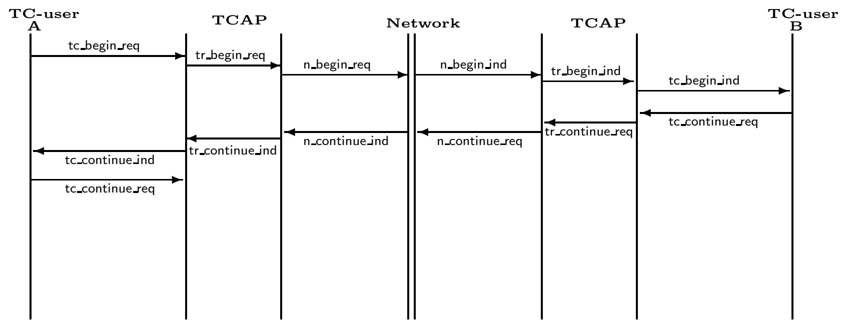


Figure 3: Two communicating tc-users

quest for getting rid of superfluous send actions comes naturally.

In Section 4 a translation into μCRL of the original TCAP is described and in Section 5 the specification of the optimisation is given. These specifications have been included in full in Appendices I and II, respectively.

3. Specifying processes with data in μCRL

The language μCRL was designed with the following objectives in mind. First, it had to be expressive enough to cover 'real-life' concurrent systems. Second, it had to have a foundation strong enough to facilitate mathematical analysis. Third and final, it had to be operational in the sense that it can be supported by software tools, assisting verification and analysis of concurrent systems.

3.1 Syntax

A μCRL specification consists of an equational specification of abstract data types and a process-algebraic specification of processes using these data types. Here, only a concise introduction is given, a complete description of the syntax can be found in [7].

The specification of a data type, or *sort* in μCRL terminology, consists of constants and function symbols, and data manipulations, or *mappings*, defined by *rewrite rules*. As an example, Figure 4 contains a definition of the natural numbers.

The processes of a μCRL specification are defined in terms of actions and the process-algebraic operators for alternative composition, sequential composition and parallel composition ('+', '·' and '‖', respectively. The actions are the smallest building blocks of the specification, composed by operators to form a process. As an example, Figure 5 shows a specification of a one-datum buffer for natural numbers.

The example buffer is able to execute two actions, put and get, for putting into, respectively getting from the buffer a natural number. Obviously, a number can only be got once the number has been put. Furthermore, in a one-datum buffer, a number can only be put if any number that has been put earlier has also been got. Thus, the process repeatedly performs one of put(0).get(0), put(1).get(1), ..., or, in the language of process algebra put(0).get(0)+put(1).get(1)+ ... where the '·' represents sequential execution of actions and the '+' represents a non-deterministic choice between two actions (with '·' higher precedence than '+'). In μCRL a concise sum notation is allowed, resulting in the specification given in Figure 5.

In order for the buffer to be used by another process, it must be able to communicate with its environment. Here, μCRL relies on the process algebra notion of communication actions, modeling synchronous communication.

Synchronous communication is communication for which sending and receiving happens at exactly the same time. If and only if the sender executes its sending action at the same time as the receiver performs its receive action, the two do communicate. So, if the sender sends just before the receiver receives, or vice versa, the two processes do not synchronise, but deadlock instead. As in process

```
sort Naturals
func 0: → Naturals
     S: Naturals → Naturals
map  add: Naturals × Naturals → Naturals
var  x, y: Naturals
rew  add(0,y) = y
     add(s(x),y) = add(x,s(y))
```

Figure 4: A μCRL abstract data type

```
act   put, get: Naturals
proc  Buffer=sum(x:Naturals,put(x).get(x)).Buffer
```

Figure 5: A μCRL process: access to one-datum buffer

algebra, the '|' operator is used to specify that two actions, say put and get, form a communicating pair put|get.

In the context of the buffer example, the put_into_buffer action, executed by its environment, forms a communicating pair with the put action, executed by the buffer, so if and only if the environment executes its put action simultaneously with the buffer executing its put, the two match and the buffer effectively accepts the value put by the environment, and the same holds for the get actions. However, since the buffer executes its get after its put, in the environment the put_into_buffer action will precede the corresponding get_from_buffer action. Therefore, for two environment processes these two actions will behave as asynchronous communication. Figure 6 specifies that the put and get actions communicate with the actions put_into_buffer and get_from_buffer, respectively.

The *encapsulation* mechanism is used to ensure that 'communicating halves' cannot occur in isolation, i.e. without their matching counterparts. This is realised by replacing isolated occurrences by deadlock actions δ using the encaps construction.

Having specified the actions, processes and specifications, the initial process is specified, usually consisting of a number of processes put in parallel. In μCRL this is specified by means of the init clause.

In the context of several communicating processes it is common to 'hide' certain actions, making them internal actions which cannot be observed. In process algebra, this is realised by replacing these by the 'silent step' τ; the corresponding μCRL construction is the hide function. In the running example of a one-datum buffer, the internal buffer action put and get are hidden (see Figure 7), which renders them unobservable.

3.2 Semantics

The semantics of μCRL is defined in terms of a transition system, consisting of states connected by actions. With computerised verification in mind, this operational semantics is the more practical choice compared to an axiomatic semantics, which use is limited to clarifying the intended meaning of a specification, or as a justification for more compact notations (for examples, see [7]).

A *transition system* consists of a number of *states*, one of which is designated the initial state, and a number of *transitions*, each of which connects two states and is labeled with an action. Performing the action brings the program from the one state into the other. More formally, a transition system $\mathcal{S}$ is defined as a tuple $\mathcal{S} = \langle S, Act, \rightarrow, s\rangle$, where S is the set of states, Act is a set of actions and $\rightarrow: S \times Act \times S$ is the transition relation.

A detailed rendering of the operational semantics of μCRL falls outside the scope of this report, but in order to understand the verification presented, an intuitive understanding of the semantics is helpful.

```
act   put_into_buffer, get_from_buffer: Naturals
comm  put_into_buffer | put = cin
      get_from_buffer | get = cout
```

Figure 6: Communication

---

init(**hide**({put,
          get },
          encaps({ put_into_buffer, get_from_buffer, put, get },
                 Buffer || Environment)))

Figure 7: Initialisation

Therefore, an example transition system is described, together with some notions of equivalence of transition systems. For a complete definition of the μCRL semantics, the reader is referred to [7], while for definitions from a classical process algebra point of view the canonical texts [3, 12] are recommended. A definition of μCRL 'by example' is given in Figure 8 where for a number of simple specification fragments the corresponding transition system is given.

These fragments are explained as follows: (a) either the action free_tid or the action free_dialogue_id is performed; (b) first a free_tid and than a free_dialogue_id is performed; (c) a free_tid and a free_dialogue_id are performed interleaved (where both actions can be the first, after which the other action is performed); (d) a free_tid is performed after zero or more times a free_dialogue_id.

A more sophisticated example of transition system is provided by the running example of the one-datum buffer. Figure 9 show transition systems for the buffer and its environment, as well as for the parallel composition of the two. In order to enforce a finite transition system, the one-datum buffer is restricted to a finite domain of two natural numbers. The figure makes explicit how parallel composition is expressed as the 'Cartesian product' of the buffer and the environment.

Closer examination of the environment reveals that it abuses the buffer, in that it possibly tries to put a second bit into the one-bit buffer. However, the encapsulation ensures that isolated communication actions, depicted by dashed transitions in Figure 9, are deadlocked, indicated by the dashed transitions, rendering part of the transition system unreachable from the initial state, effectively blocking the environment's abusive behaviour. The one and only course of action left is that

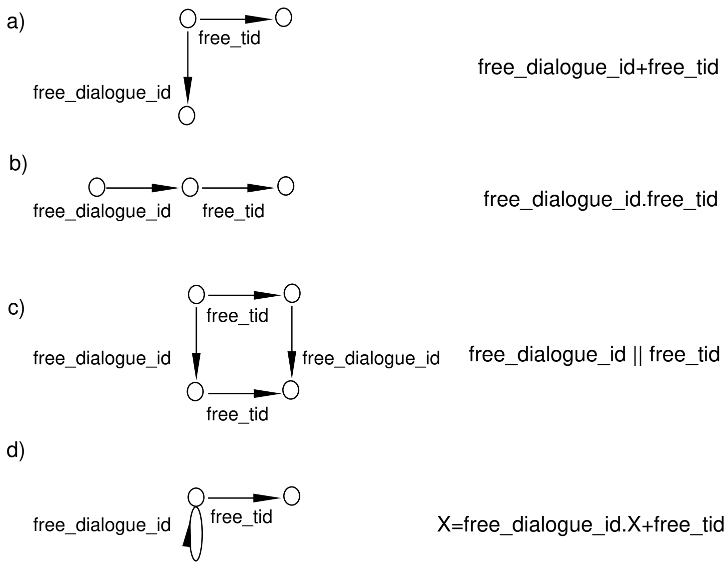


Figure 8: μCRL semantics by example

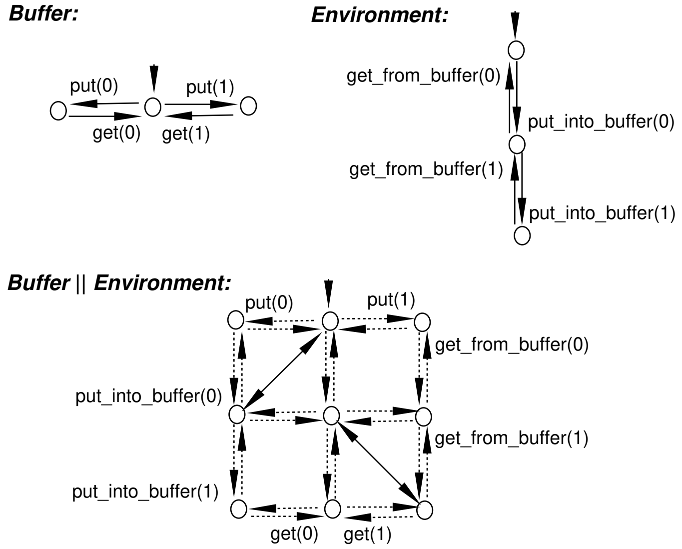


Figure 9: Transition system for the one-datum buffer

the environment repeatedly puts and gets a 0 bit.

Given two specifications, the question whether these are equivalent boils down to constructing their transition systems and verifying whether these are equivalent. Equivalence of transition systems is defined in numerous ways, but in the context of this text, it is sufficient to understand the notions of strong bisimulation and weak bisimulation.

Loosely spoken, two transition systems are *strongly bisimular* if a relation can be defined between the states of the two systems with the property that in each pair of related states the same actions result in related states. Weak bisimulation is defined similarly, ignoring the silent τ steps.

Formally, suppose $\mathcal{S}_1 = \langle S_1, Act_1, \rightarrow_1, s_1 \rangle$ and $\mathcal{S}_2 = \langle S_2, Act_2, \rightarrow_2, s_2 \rangle$ are transition systems. A *strong bisimulation* is a relation $R : S_1 \times S_2$ with the following properties:

- $s_1 R s_2$ and $s_1 \xrightarrow{a}_1 s_1' \Rightarrow s_2 \xrightarrow{a}_2 s_2'$ and $s_1' R s_2'$, for some $s_2' \in S_2$
- $s_1 R s_2$ and $s_2 \xrightarrow{a}_2 s_2' \Rightarrow s_1 \xrightarrow{a}_1 s_1'$ and $s_1' R s_2'$, for some $s_1' \in S_1$

A *weak bisimulation* is a relation $R : S_1 \times S_2$ with the following properties:

- $s_1 R s_2$ and $s_1 \xrightarrow{a}_1 s_1' \Rightarrow a = \tau$ and $s_1' R s_2$ or $s_2 \xrightarrow{\tau^*}_2 \cdot \xrightarrow{a}_2 \cdot \xrightarrow{\tau^*}_2 s_2'$ and $s_1' R s_2'$, for some $s_2' \in S_2$ and
- $s_1 R s_2$ and $s_2 \xrightarrow{a}_2 s_2' \Rightarrow a = \tau$ and $s_1 R s_2'$ or $s_1 \xrightarrow{\tau^*}_1 \cdot \xrightarrow{a}_1 \cdot \xrightarrow{\tau^*}_1 s_1'$ and $s_1' R s_2'$, for some $s_1' \in S_1$ and

Systems that are strong bisimular are also weak bisimular, but not vice versa. The notions are exemplified in Figure 10.

In conclusion, μCRL specifications are equivalent with respect to some bisimulation relation iff the state spaces associated are equivalent. It is this notion of equivalence that plays a key part in the verification presented in this report. In the next section, it is described how these steps are supported by software tools.

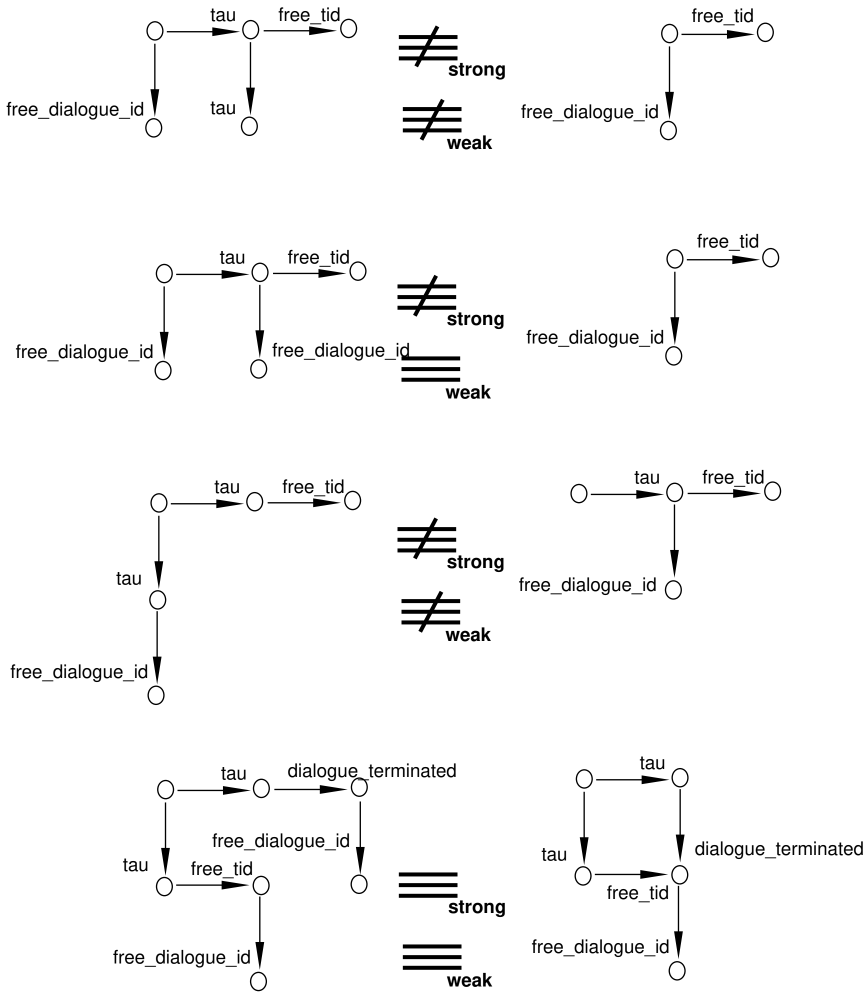


Figure 10: bisimulations by example

3.3 Tool support

In order to prove or disprove equivalence of two μCRL specifications for each of the two a transition system needs to be generated, after which bisimulation equivalence is to be checked. Both of these steps are mechanised.

State space generation consists of two steps. First, the μCRL specification is syntactically checked and linearised, i.e. it is converted to a format that is more suitable for automated processing. Second, the resulting specification is processed by the instantiator tool, which actually generates the state space. Both these tools, *mcrl* and *instantiator*, are freely available [4]; for the underlying principles, the reader is referred to [8].

The actual verification is supported by Cæsar/Aldébaran [6, 9] a package featuring efficient equivalence checking and reduction for a variety of equivalence relations, such as strong bisimulation and weak bisimulation checking, checking of temporal formulae, deadlock checking and state space visualisation.

4. Specifying TCAP

Before we are able to formally verify that the original TCAP and the optimised version are in some form equivalent, we need to have a formal specification of both protocols. Although the original TCAP is thoroughly specified by ITU [11], this specification is only partly described in a formal language, viz. the standard Specification and Description Language (SDL) standardised by the CCITT [10], whereas the other part is described in informal text. With our tools for verification in mind, we decided to translate the specification given in SDL directly into the specification language μCRL. For the optimised TCAP we had no specification at hand, only an implementation. Together with the author of this implementation, we derived a specification of it, which as a side-effect resulted in a better understanding of the implementation.

It is important to note that the writing of the specification cannot be seen as separate from the actual verification, i.e. the bisimulation checking. More than once, differences between the original and the optimised TCAP were found to be originating from deficiencies in the specification, as opposed to errors in the optimisation. This section aims at describing the final specification. In Section 7 the interaction between specification and verification will receive attention, it can be considered as the history of the specification of this section.

4.1 From SDL *to* μCRL

The specification of the state machines occurring in this protocol are all given in SDL. Only a few basic elements of this language are used in the specification, which all map easily to μCRL primitives.

As an example, consider the *Idle* state of the *Transaction State Machine* (TSM). The SDL specification (see Figure 11) should be interpreted as follows: whenever the state machine TSM is in the idle state, it waits until it receives a begin message, which is supposed to be sent by the TCO state machine. Here we distinguish between a *begin received* (begin_rec) and a *begin transaction* (begin_trans) message. Furthermore, any other message sent to TSM and received in this state is just discarded.

4.2 Modelling the communication

In the SDL specification for most messages both a sender and a receiver are specified, but sometimes a state machine and sometimes a layer is mentioned as the receiving/sending party. Carefully checking the SDL specification reveals that we can be more specific than just mentioning the layer, since for every message there is only one state machine sending and only one state machine receiving this message[3]. We take advantage of this by specifying a communication buffer (or channel) for every state machine and for the messages to and from the environment, i.e. the tc-user (user) and the network (sccp). Reading from and writing to a channel is denoted by prefixing the channel name by r_ and s_, e.g. r_tsm(begin_rec) indicates an attempt to read begin_rec from state machine TSM.

[3] Being very precise, this should read 'at most one', since tr_p_abort and tr_notice_ind are sent, but never received (see Section 4.5).

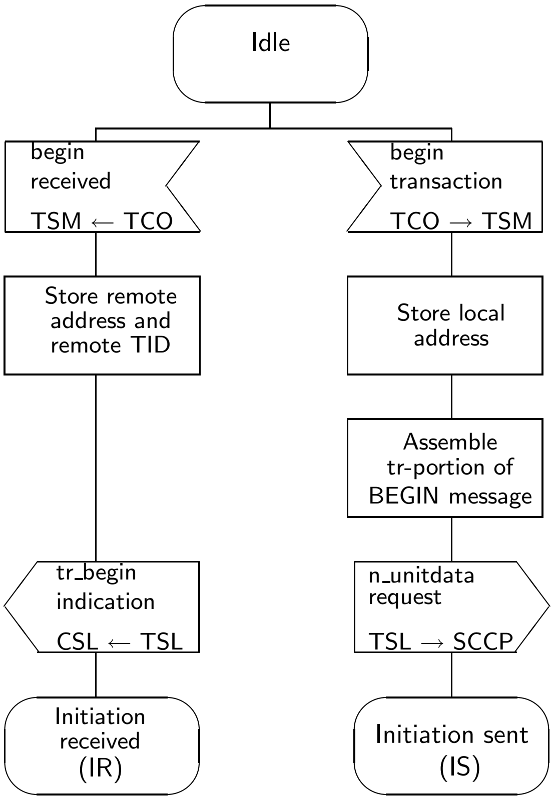


Figure 11: SDL specification of the Idle state of TSM

The semantics of SDL enforces that all messages that are sent from one process to another also actually arrive at the other process. Nothing is said about the capacity to store messages that have arrived. However, the information is provided that messages that have arrived in a certain state are discarded if they cannot be read in that state. We have not tried to model this behaviour in μCRL, since we regard a message that arrives in the wrong state as an error of the protocol. The ultimate consequence of not specifying the removal of these messages is a deadlock situation, the presence of which can be verified by Cæsar/Aldébaran. However, the proven absence of deadlocks supports the view that the communication model chosen is not too restrictive.

We choose to model the communication by means of a one-datum buffer, i.e. a channel that can only contain one message at a time. In Section 3 we have described how such a one-datum buffer is specified in μCRL. This one-datum buffer forces state machines to block whenever a channel already contains a message. Therefore, our specification might be too restrictive with respect to the concurrency of the state machines.

4.3 Sequence of actions and non-deterministic choices

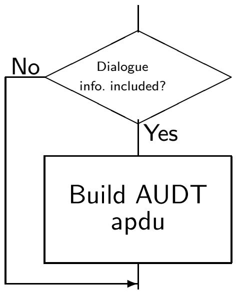
In our SDL example, after receiving a message, several actions are performed. These actions refer to some program code, but are left abstract and their implementation is left up to the programmer. We follow this approach by directly translating the actions into μCRL actions, viz. store_remote_tid, store_local_address, assemble_begin_message. After the initial sequence of actions a message is sent, respectively a tr_begin indication (tr_begin_ind) to the layer CSL or an n_unitdata request which contains the assembled *begin* message (n_begin_req). At this point the SDL specification is rather sloppy, but after carefully studying the document, one has to conclude that sending a message after assembling a message indicates that the assembled message is sent. In both branches we jump to the next state after sending the message, viz. *Initiation received* (IR), and *Initiation sent* (IS). If we recall that in μCRL the dot was used to represent sequential actions and the plus is used for non-deterministic choice, then it is easily seen that the SDL fragment of Figure 11 translates to:

```
proc tsm_Idle=
        r_tsm(begin_rec).
              store_remote_tid.
              s_dha(tr_begin_ind).
              tsm_IR +
        r_tsm(begin_trans).
              store_local_address.
              assemble_begin_message.
              s_sccp(n_begin_req).
              tsm_IS
```

4.4 Tests and empty transitions

Several tests occur in the SDL specification of TCAP, basically checking the contents of a message for the occurrence of a certain field or specific data item. We have chosen not to specify the contents of the messages and therefore we are unable to test the contents of specific fields in the messages. Instead, we specify all tests to be non-deterministic choices, which implies that we expect anything to be possible with respect to the contents of a message. This is a safe assumption, but it might introduce more non-determinism than is actually possible.

A test in SDL is denoted by a diamond with the property to test written in it. The diamond has two alternatives, denoted 'Yes' and 'No'. We translate all diamonds with the '+' operator of μCRL, where the alternatives are the arguments of the test. See Figure 12 for an example of an SDL test with its μCRL translation.

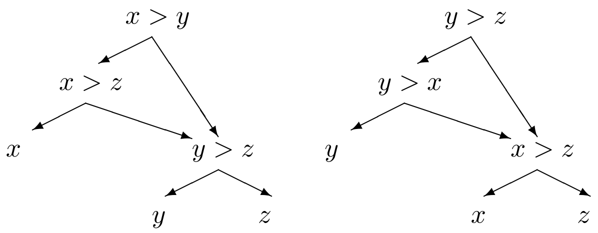


build_audt_apdu + tau

Figure 12: Translation of: 'building an audt' if the 'dialogue info' is included in a message

In many cases the arguments consist of different sequences of actions that join later on. In particular it occurs that an action is only performed whenever a tests succeeds and otherwise we already jump to the consecutive action, which is translated by the corresponding τ action in μCRL, such as in Figure 12. Note that, by this transformation, we lose the information of what is tested. Moreover, the '+' is a commutative and associative operator, thus we lose the order in which the tests should be performed as well.

This is best clarified with an abstract example, in no way related to the TCAP verification. Consider a simple algorithm that computes the maximum of three numbers x, y, and z, by using the test $n > m$ for numbers n and m. Clearly we can perform the test in several ways. For example we could first test $x > y$ and, if this succeeds, $x > z$ after which by success the value x is presented and by failure the test $z > y$ is performed. Another way of performing the test is to first compare x and z and consecutively compare y. In a picture we depict these two possibilities by



In μCRL the resulting specifications are $\tau\cdot(\mathsf{max}(x) + \tau\cdot(\mathsf{max}(y) + \mathsf{max}(z))) + \tau\cdot(\mathsf{max}(y) + \mathsf{max}(z))$ and $\tau\cdot(\mathsf{max}(x) + \tau\cdot(\mathsf{max}(x) + \mathsf{max}(z))) + \tau\cdot(\mathsf{max}(x) + \mathsf{max}(z))$, which are not equivalent. So, data abstraction can introduce differences.

This, however, only affects tests that are directly performed after each other, for tests followed by actions we obtain the same possible sequences of actions. As an example consider the SDL specification[4] of the DHA machine in the *Idle* state when a tr_uni_ind message is received (see Figure 13). The two

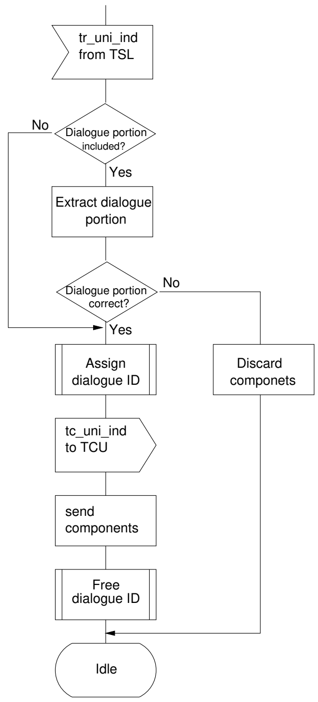


Figure 13: SDL specification of DHA in Idle state receiving tr_uni_ind

tests are followed by actions and as can be seen in the μCRL specification, we have to copy some of the actions to two different places. The crucial point is, however, that the 'discard_components' is always preceded by an 'extract_dialogue_portion'.

```
r_dha(tr_uni_ind).
 ( assign_dialogue_id.
   s_user(tc_uni_ind).
```

[4]The test 'version 1 supported' has been left out, since the result of this test should always be 'Yes'.

```
      send_components.
      free_dialogue_id +
    extract_dialogue_portion.
      ( discard_components +
        assign_dialogue_id.
          s_user(tc_uni_ind).
          send_components.
          free_dialogue_id
      )
  ).dha_Idle
```

In the next section we return to the test issue and discuss which consequences it has for the specification of the optimisation and why it does not harm to translate tests like this in these particular circumstances.

4.5 Checking the TCAP *specification*

With the above described translation, the μCRL specification is easily obtained from the given SDL specification. The translation is carried out by hand, a process during which small typeset errors in the specification were detected: TCM written instead of TSM, ABRT meaning abort *etc.* There is a severe risk of introducing such and severer mistakes oneself by performing the manual translation (resulting in approximately 650 lines of μCRL code). Carefully checking the resulting specification using the tool sets of μCRL and Cæsar/Aldébaran is therefore necessary, thus eliminating syntax errors and deadlock situations.

Using these tools we detected that the message tr_p_abort can be sent from the *Idle* state of DHA, but there is no state in which this message can be received. Feedback from the implementor of the protocol taught us that the test 'version 1 supported' preceding the sending of this message, always succeeds, hence we removed the test and the 'No' alternative from the specification, such that sending the tr_p_abort is no longer an option.

In the *Idle* state of TCO the message tr_notice_ind can be send to the CSL layer, but nowhere in the protocol this message can be received. We decided to remove the possibility to send this message from the specification, i.e. act as if the message is just discarded as soon as it has been sent.

Finally, it was discovered by code inspection that in the implementation handling the internal local_abort message had erroneously been shifted from the active state to the *Initiation Received* state of the protocol. It was this error that would have severely hampered the correctness of the optimisation in practice, more than any other error found!

5. Specifying optimised TCAP

The SDL specification of the original TCAP served as a basis for its μCRL counterpart. For the optimised TCAP such a specification was not available. A well-written ERLANG program served here as the basis for a μCRL specification. However, the program is much more detailed than an SDL specification and abstracting to μCRL is therefore a non-trivial task. A full week has been spent in describing the program on an abstract level.

In discussions with the implementor of the program, we had to find out which state machines were implemented in the optimised version and which implicit assumptions were made for the optimisation.

In the implementation several tests and actions had been combined, resulting in performing only one test. Our idea of specifying tests as non-deterministic choices turned out to be problematic for these tests (see Section 7). The actions described in the original TCAP and those in the optimised TCAP had some dissimilarities because of implementation-specific choices. Manually we had to check that the semantics of the actions in the original TCAP and in the optimised TCAP were equivalent.

5.1 Abstraction of the optimised TCAP

As explained in Section 2, the TCAP protocol consists of two layers, a component layer and a transaction layer. The component layer communicates with the tc-user and the transaction layer communicates with the network (Figs. 1 and 2). The layers communicate and synchronise with each other by means of messages that are prefixed with tr_.

The main idea of the optimisation of TCAP is to get rid of the messages in between the layers. The messages are exchanged between the state machine DHA in the components layer and the two state machines TCO and TSM in the transaction layer. Both DHA and TSM consist of four states, viz. *Idle*, *Initiation Sent*, *Initiation Received* and *Active*. One of the observations of the implementor of the optimised TCAP was that not only the names of the states are the same, but the behaviour of the state machines is similar as well. Even more, the state machines seem to follow each others state, i.e. whenever DHA goes to *Idle* then TSM goes to *Idle* as well, whenever TSM goes to *Active*, DHA goes to *Active* as well, *etc.* Because of this synchronisation behaviour, the implementor decided to merge DHA and TSM into a new state machine with four states, basically merging the states of the separate machines.

This optimisation idea is best explained by a specification fragment. Consider the situation in the original TCAP where TCO and DHA are in the *idle* state (Figure 14). Whenever a tc_uni_req is received

```
proc tco_Idle=                      proc dha_Idle=
       r_tco(tr_uni_req).                  r_user(tc_uni_req).
          assemble_uni_message.               (build_audt_apdu + tau).
          s_sccp(n_uni_req).                  request_components.
          tco_Idle+                           process_components.
          ⋮                                   assemble_tsl_data.
                                              s_tco(tr_uni_req).
                                              free_dialogue_id.
                                              dha_Idle+
                                              ⋮
```

Figure 14: Sequentialising the specification (original)

by DHA in this state, a tr_uni_req is sent to TCO. The effect of the message is that TCO sends an n_uni_req to the network. The idea behind the optimisation is that instead of sending this tr_uni_req to TCO, the state machine DHA is sending the n_uni_req directly to the network (see Figure 15).

With some additional small changes to the TCO state machine, it was claimed that messages from the tc-user and from the network result in the same observable behaviour when TCAP is regarded as a black box.

In a process of discussion and feedback we obtained the 'specification' as given in Figure 16 from the implementor, describing the two new state machine together with all messages. Actions have not been specified in this first approach. This picture translates easily to μCRL, where we only have three communication channels, two to the outside world and one for communication between the two state machines. Again we choose to specify the channels with a one-datum buffer, although ERLANG has unbounded queues as its communication paradigm and the implementation is such that a generic server collects all messages from the outside world (tc-user and network) and these internal messages are communicated to the state machines by call-back functions. From the point of view of the specification, this implies that at most one message at a time is received and can be sent.

```
proc dhatsm_Idle=
    r_user(tc_uni_req).
      (build_audt_apdu + tau).
      request_components.
      process_components.
      assemble_tsl_data.
      assemble_uni_message.
      s_sccp(n_uni_req).
      free_dialogue_id.
      dhatsm_Idle+
    ⋮
```

Figure 15: Sequentializing the specification (optimisation)

5.2 Combining tests and actions
The SDL specification of the original TCAP protocol contains several tests and actions that are rather implementation-specific. For example, analysing a message's 'dialogue portion' is done in several steps, first checking whether this portion is included in the message, if so, extracting this, and finally checking if the dialogue portion is correct (e.g. Figure 13). In a language where one explicitly has to allocate memory for storing the dialogue portion and where one explicitly has to free the memory afterward, this seems a logical sequence of tests and actions. However, in ERLANG with its dynamic memory allocation and garbage collection, this sequence has been replaced by a combination of a test and an action, viz. extract a 'dialogue portion' from the message (which might be *undefined* if it does not exist) and fail if this portion is incorrect, after which the garbage collector automatically removes the allocated memory and even discards the received message. Also, in ERLANG one has a *case* construct, which encourages programmes to combine sequences of *if-then-else* constructs in one *case* construct. The specification language μCRL lacks a *case* construct, hence we have to translate the ERLANG cases to *if-then-else* statements, or similar to the previous approach, to non-deterministic choices. Care should be taken in this translation, since from a purely operational point, performing tests in a different order results in non-equivalent specifications. Therefore, we decided to follow the original tests and actions as much as possible in the specification of the optimised TCAP.

6. Specifying the environment
The specification of TCAP describes how incoming messages are treated and which outgoing messages may occur under which circumstances. Since the informal specification [11] leaves it open in which order a tc-user sends its messages and what the resulting messages should be, our first ambition was not to specify an environment, i.e. to allow any possible sequence of messages from the tc-user and the network. Soon it turned out that we would be unable to prove equivalence of original and optimised TCAP in such a general environment. Modelling behaviour which is far beyond normal use, for instance, a tc-user starting to communicate by sending a tc_end_req, introduces problems with respect to the equivalence we want to prove (we mention an infinite state space as one of the problems). Since differences between the optimised TCAP and its original specification are uninteresting for those sequences of messages that cannot occur in reality, we tried to isolate all possible realistic sequences. A drawback of this approach is that these sequences are not specified. Hence, their construction is based on our and the implementor's experiences and intuition.

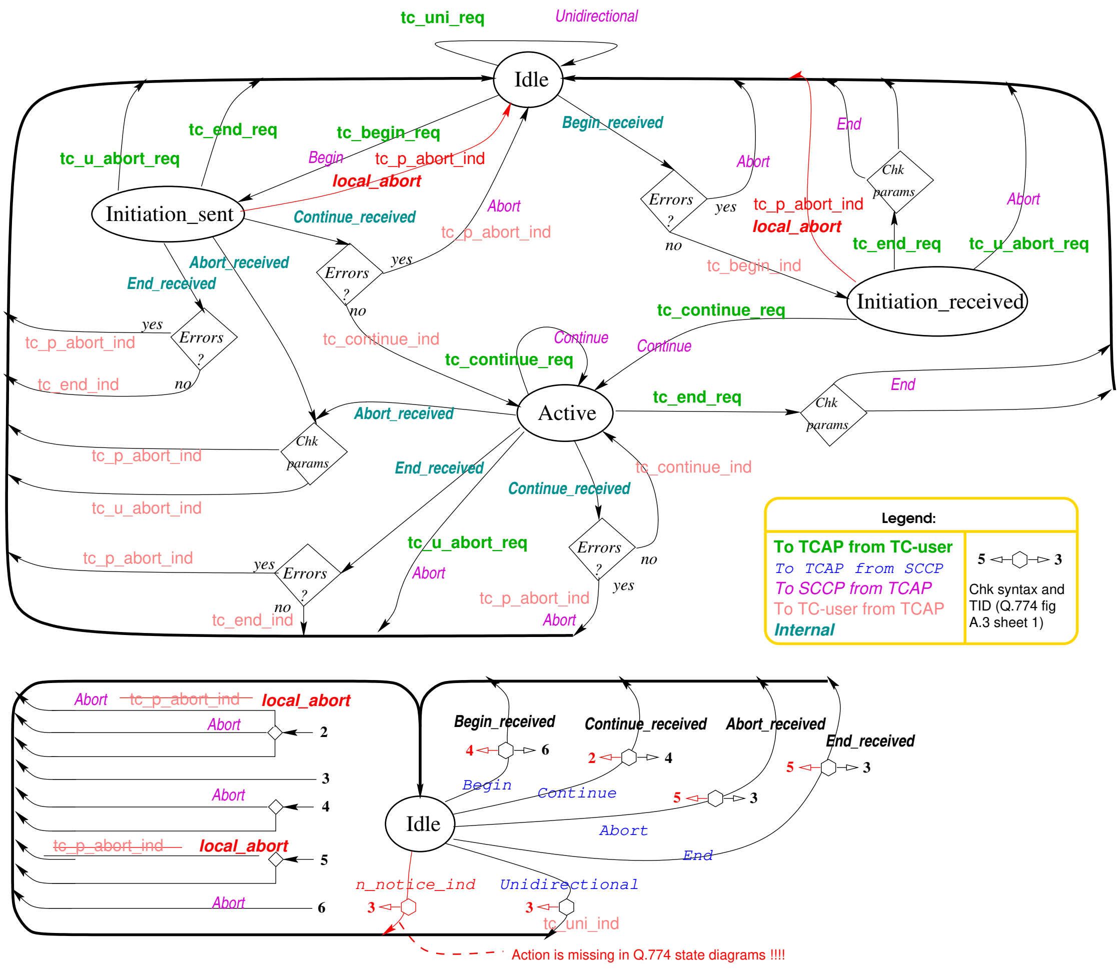


Figure 16: Initial specification of optimised TCAP

6.1 Two tc-users talking

The TCAP protocol is one layer in a stack of several protocols where, roughly speaking, lower layers take care of message delivery and time-outs, and a higher layer is an application that uses TCAP. One connection involves two protocol stacks, connected by a network, thus there are two tc-users, two TCAPs and two sets of lower protocol layers. So, the environment of one instance of TCAP consists of the tc-user on top of it, the network layers below it, the actual network, the network layers at the other side of the connection, the TCAP at the other side and, finally, the tc_user at the other side. To keep things manageable, a more abstract environment was used, consisting of the two parts of the environment observable by the local TCAP: the tc-user and the SCCP network layer.

We, naively formalised the environment as a tc-user party and a network party (the SCCP) that both could either send a begin message or receive a begin message, responding to this by changing state (the engaged state) in which no new begin messages can be received any longer (see Figure 17).

In this formalisation it has been made explicit that after termination, the protocol regains its idle state, i.e. is open for other connections. However, numerous differences found by Aldébaran indicated that differences occurred after termination of a connection. Consulting the implementor of the optimisation brought clarity: the assumption of TCAP repeatedly establishing connections was

```
proc tc_user =                          proc sccp =
    s_user(tc_begin_req).                   s_sccp(n_begin_req).
      tc_user_engaged+                        sccp_engaged+
    r_user(tc_begin_ind).                   r_sccp(n_begin_ind).
      tc_user_engaged                         sccp_engaged

proc tc_user_engaged =                  proc sccp_engaged =
    s_user(tc_continue_req).                s_sccp(n_continue_req).
      tc_user_engaged+                        sccp_engaged+
    s_user(tc_p_abort_req).                 s_sccp(n_p_abort_req).
      tc_user+                                sccp+
    s_user(tc_end_req).                     s_sccp(n_end_req).
      tc_user+                                sccp+
    r_user(tc_continue_ind).                r_sccp(n_continue_ind).
      tc_user_engaged+                        sccp_engaged+
    r_user(tc_abort_ind).                   r_sccp(n_abort_ind).
      tc_user+                                sccp+
    r_user(tc_end_ind).                     r_sccp(n_end_ind).
      tc_user                                 sccp
```

Figure 17: Iterative two-process environment (tc_user process)

incorrect in the context in which TCAP was verified.

TCAP, as a complete protocol, creates a new transaction ID for every new connection. Messages are all labelled with a transaction ID, except for the begin message, which causes a new ID to be generated. Hence, a running transaction cannot be disturbed by a new begin message or a message from another connection.

Thus, either a tc_begin_req or an n_begin_ind starts a session, depending on whether the local or the remote TCAP initialises the connection. After receiving one of those *begin* requests, any other begin request initiates in fact a new connection. We are only interested in one session and therefore, we may safely assume that after the initial *begin* request, no other *begin* message will follow, which is depicted in Figure 18 for the tc-user side (the SCCP side is specified in the same way).

Observations like this are useful for two purposes. First they drastically reduce the size of the state space, and, even more important, they exclude situations in which the original and optimised protocol differ, and that are irrelevant in the sense that these do not occur in normal use. For example, in situations where both an n_begin_ind and a tc_begin_req are received, the original and the optimised protocol end up in non-equivalent states.

We could have tried to also model the assignment of transaction IDs and therewith a more precise representation of the protocol. In that case we had to model an extra process administering the IDs and whenever a new transaction is demanded, this administering process assigns a new ID and creates new processes for all state machines. These state machines are only given the messages that are determined for them. We have chosen not to model this more complicated behaviour. On the one hand this extra modelling is outside the scope of this verification study, since the focus is on verifying the merge of transition machines. On the other hand modelling this behaviour would (if done correctly) imply dynamic spawning of processes, which is not supported by the language μCRL.

The environment of Figure 18 still contains the problem that we have two processes that can both start sending a begin message. We could avoid this by starting the tc-user in engaged mode and SCCP in active mode and vice versa. However, we need a stronger synchronisation property. Whenever a

```
proc tc_user =
    s_user(tc_begin_req).
      tc_user_engaged+
    r_user(tc_begin_ind).
      tc_user_engaged

proc tc_user_engaged =
    s_user(tc_continue_req).
      tc_user_engaged+
    s_user(tc_p_abort_req).
      delta+
    s_user(tc_end_req).
      delta+
    r_user(tc_continue_ind).
      tc_user_engaged+
    r_user(tc_abort_ind).
      delta+
    r_user(tc_end_ind).
      delta
```

Figure 18: One-connection two-process environment (tc_user process)

message is sent to TCAP, the environment waits for a response and reacts on that. This idea boils down to using only one process for the environment, having two states, the state in which a begin message is expected and a state in which the begin has been sent and the parties are communicating (Figure 19).

With the latter environment we have two problems left. First, we are left with the situation in which a connection is established and immediately disconnected from the other side. It turned out that this behaviour was unrealistic (the transaction ID of the other side would not have matched) and a continue message had always to be sent as a response first. Adding to the environment that the other party always responses with either a continue message or an abort message solves this problem. Second, there are messages that are discarded and hence no response to the environment is generated. In the next section this problem is discussed in more detail.

The basic ideas of the environment are presented in Figure 19, but some details are left out. For instance, there is also a unidirectional message, which is sent by one party, where after the connection is terminated. We refer to Appendix I for the complete specification of the environment.

```
proc environment =
    s_user(tc_begin_req).
      r_sccp(n_begin_req).
      engaged+
    s_user(n_begin_ind).
      r_user(tc_begin_ind).
      engaged

proc engaged =
    s_user(tc_continue_req).
      r_sccp(n_continue_req).
      engaged+
    s_sccp(n_continue_ind).
      r_user(tc_continue_ind).
      engaged+
    ⋮
```

Figure 19: One-connection one-process environment

6.2 The no_message *message*

Messages that are received by TCAP are subjected to a format check. We have not specified this format, including source and destination addresses, message IDs, message specific data and much more. However, it is of importance to specify that a certain message might be rejected because of an erroneous contents. Therefore, we replaced all SDL tests for correctness of the message with non-deterministic choices in μCRL. This represents the continuously present possibility of receiving an erroneous message.

The action performed when a message is of the wrong shape, depends on the state of the process and the contents of the message. If the source address of the message can still be extracted from the message, an *abort* could be sent in return. However, when the sender is unclear, the message need be discarded. In the latter case the sender receives no response.

The environment waiting for the response on a certain message will deadlock in case no return messages is sent. Since specifying 'not getting a response, is impossible within μCRL we decided to use a trick to overcome the problem that this typical kind of deadlock could not be distinguished from ordinary deadlocks. In those cases where we were sure that no message was returned, we added the sending of a special return message, the *no_message* message. This no_message message is kept by the environment, which then terminates. This could have been specified by having the environment enter a deadlock state, but in order to be able to distinguish between an erroneous deadlock and a recognised termination, the latter has been specified by means of an infinite idle loop.

Clearly, this solution is unsatisfactory, since we changed the actual specification by adding meta knowledge about the specification. A misinterpretation of us would imply that we have checked a different protocol than the one we claim we have checked.

7. Checking equivalence

In theory, verifying correctness of the optimised TCAP protocol consists of specifying both the original and the optimisation in μCRL, generating a state space for each, using the *instantiator* from the μCRL tool set, and checking bisimulation equivalence of the two state spaces with the Cæsar/Aldébaran tool set. In practice, however, verification appeared to be more complex, for several reasons. First, the specifications were hardly accurate from the start. Apart from the usual minor mistakes, the specification process was repeatedly hampered by a limited understanding of matter by the 'specifier'. Second, the bisimulation semantics chosen for comparing the two specifications appeared to be stricter than needed; no alternative was available. Third, the specification language μCRL misses a language construct needed for a natural specification of the TCAP protocol. Fourth and final, the initial optimisation was not correct. The verification process brought to light several minor errors that needed to be fixed for the verification to be able to proceed. Instead of verification as a concluding step in the development, it appeared to be a recurring step, repeatedly performed after each iteration of the processes of specification and, even, implementation, providing essential feedback to each of these.

The Cæsar/Aldébaran tool set is, basically, only used for checking bisimulation equivalence. Its use in locating the differences found is, by its nature as operating on state spaces, limited. The output in case of a mismatch consists of a common path to diverging state (for example, see Figure 20). Tracing differences back to a location in the μCRL specification needs to be done manually and can be problematic, due to μCRL and Cæsar/Aldébaran being independent tools. The simulation facilities offered by Cæsar/Aldébaran are only useful to some extent, in that these allow one to 'step through' the state space, following actions from state to state, but the drawback is that these force one into a very restricted view, analysing differences between states, obscuring the view from more global patterns, such as the 'parallel diamonds' described in the next section. The key in isolating differences is in projecting out parts of the state space by studying scenarios.

Differences found fall in three categories. Some were rather trivial errors like typing errors in specification that could be fixed instantaneously by consulting the designer of the TCAP optimisation. More interresting are specification shortcomings that rooted in insufficient knowledge of crucial aspects of the protocol, such as the environment and the communication model of the internal bus. The final

```
LTSs org_omin.aut and opt_omin.aut are not related modulo strong bisimulation.
Diagnostic sequences generated by aldebaran:

sequence 1:
initial states
(S1 = 0, S2 = 0)
"c_sccp(n_begin_ind)"
(S1 = 5, S2 = 5)
"assign_local_tid"
(S1 = 8, S2 = 8)
"cs_tsm(begin_rec)"
(S1 = 11, S2 = 11)
"cr_tsm(begin_rec)"
(S1 = 14, S2 = 14)
"store_remote_tid"
(S1 = 18, S2 = 18)
"extract_dialogue_portion"
(S1 = 30, S2 = 26)
"set_application_mode"
(S1 = 22, S2 = 22)
"assign_dialogue_id"
(S1 = 26, S2 = 30)
"c_user(tc_begin_ind)"
(S1 = 34, S2 = 38)
"c_sccp(n_end_ind)"
(S1 = 35, S2 = 39)
"i"
(S1 = 37, S2 = 9)
Only org_omin.aut can do a "discard_received_message"-transition from these states

sequence 2:
initial states
(S1 = 0, S2 = 0)
Only opt_omin.aut can do a "discard_received_message"-transition from these states

sequence 3:
initial states
(S1 = 0, S2 = 0)
Only opt_omin.aut can do a "i"-transition from these states
```

Figure 20: Comparing state spaces with Aldébaran

```
proc Bus=sum(m:Bus_message,s-_bus(m).r-_bus(m).Bus)
```

Figure 21: Asynchronous communication

category consists of relevant differences between the original and the optimisation.

7.1 Refining the environment

As explained in Section 6, the context in which TCAP operates is specified in the environment, which is the only part of the specification common to both the original and the optimised TCAP. This environment represents all possible situations the protocol can expect, so in order to be able to study the protocol in certain well-specified situations, the specification of the environment needs to be restricted, by 'commenting out' parts of the specification that are not within the scope of interest. In some cases, it is possible to restrict the TCAP protocol itself, but this is only feasible if the behaviour of interest can be isolated in both specifications. Often, restricting the environment, being the common part of the two, is the safer approach.

So, once the rough location of a difference has been found, it can be further tracked down by restricting the environment to exactly those situations where the difference is expected to occur, after which the newly generated restricted state space can be analysed in more detail. Usually, this requires some trial and error, but especially in tracking down more subtle differences this proved to be a useful technique.

As described in Section 6 the environment developed from a very general one (Figure 17), via a two-process restricted version (Figure 18), to a one-process environment (Figure 19) where the actual behaviour had to be refined even more. The environment finally chosen for verification purposes indeed presents a trustful representation of the assumptions underlying the correct use of TCAP. On the other hand, this implies that the validity of the verification is limited to 'ideal' situations, in which all users of the protocol abide by the rules. In practice, this is indeed the case, as the network layers TCAP is sandwiched in between, i.e. the user and the SCCP, have their own protocols.

As a consequence, issues related to fault-tolerance, such as what happens in erratic circumstances as an SCCP line loosing messages, or whether the optimised TCAP performs better or worse in this respect, fall definitely out of the scope of this study. However, the appropriateness of the model-checking verification approach chosen for addressing fault-tolerance is limited. Fault-tolerance, being an issue with statistical underpinnings, is probably better addressed in a decent test traject than in the verification preceding it.

7.2 Refining the communication model

Concurrently with exploring the TCAP environment, the internal communication model was refined. It would be a blatant over-simplification to state that these two refinement were carried out in isolation, since each of these were developed in small trial and error steps, each of which was thought of as being final. Instead of the communication model being implicit, as was the case with the environment, the problem with the communication model was that it was explicitly specified in existing specifications, but that the semantics of these constructions was implicitly.

The initial problem was that the native μCRL specification primitive is synchronous communication, whereas the TCAP communication is essentially asynchronous. However, synchronous communication being the simplest of the two, it can be used to specify asynchronous communication, as illustrated in Figure 21. What remains to be figured out is the precise nature of asynchronous communication.

The fragment of Figure 21 corresponds to a one-datum buffer, but this is not the only option. Other possibilities are a queue, bounded or unbounded, or a number of one-datum buffers that can be accessed at random. Experiments demonstrated that each possible choice results in different TCAP behaviour; moreover, unbounded queues even resulted in an infinite state spaces.

Thorough inspection of the ERLANG implementation revealed there was no such thing as one internal bus [2]. Instead, each of the three state machines has its own private buffer in which messages directed to it are posted. At certain point, it was concluded that these internal busses correspond to one-datum buffers, and this was taken as a leading assumption in further verification, but later, after studying the SDL manuals, it was concluded that these should be really unbounded queues. Although this does not render invalid the verification presented here, since a one-datum buffer is a very limited case of an unbounded queue, it does pose an important side mark in the final conclusion that can be drawn from the verification of TCAP (Section 10).

7.3 Differences in non-determinism

Having fixed most of the minor specification errors and settled upon a reasonably accurate models of environment and internal communication, Aldébaran still signalled non-trivial differences between the original TCAP and the optimisation. It soon appeared that most of the remaining differences could be categorised of differences of non-determinism. As is not uncommon for optimisation in general, the TCAP optimisation was more deterministic than the original in that it fixed the order of certain actions where the original left the order unspecified. Intuitively, this is not erroneous but, strictly spoken, the optimisation differs from the original and that is the conclusion Aldébaran draws.

As an example of where non-determinism arises we refer to Figure 14 in Section 5. In the two specification fragments, each taken from a different state machine, the two state machines synchronise, in that the second automata receives an internal message (tr_uni_req) sent by the first. After the one state machine has sent the message and the other state machine has received it, the two are free to proceed. However, the optimised TCAP merges the two state machines in a straightforward manner, with the side-effect of the two courses of actions being fixed in one interleaving (Figure 15).

In the state space, this type of difference can be recognised in the occurrence of 'parallel diamonds' in the original that are missing in the optimisation. By careful direct manipulation in the state space visualisation tool from Cæsar/Aldébaran these patterns can be made explicit manually, though this is only feasible if the state space is not too large. For example, see Figure 22, where the transitions and states that are missing in the optimisation are dashed. However, it is not feasible to check by

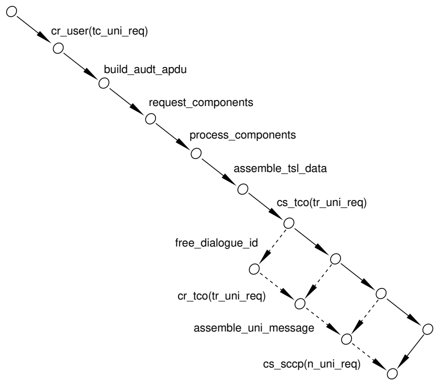


Figure 22: parallel diamonds

hand whether all differences between the two state spaces are indeed of this type. The solution is in translating these graphical patterns back to the μCRL specification.

Having located a μCRL fragment where the optimisation fixes a course of actions, the verification is proceeded by replacing the fragment by a fragment where all non-deterministic alternatives are made explicit (for an example, see Figure 23). If, the resulting spaces are equivalent, then it can be safely concluded that the optimisation has a lower degree of non-determinism than the original.

```
proc Optimization2_Idle=
    r_user_external(tc_uni_req).
      (build_audt_apdu + tau).
      request_components.
      process_components.
      assemble_tsl_data.
      ( free_dialogue_id.
          assemble_uni_message.
          s_sccp_external(n_uni_req)+
        assemble_uni_message.
          ( free_dialogue_id.
              s_sccp_external(n_uni_req)+
            s_sccp_external(n_uni_req).
              free_dialogue_id)).
          Optimization2_Idle+
        ...
```

Figure 23: Intermediate specification

So, the differences in non-determinism have been verified in two steps. First, an intermediate specification is written which is obviously a more non-determinstic version of the optimisation and, second, this intermediate specification is proven equivalent to the original. The verification of the first step can be considered a trivial case of theorem proving as opposed to the model-checking approach of the second. It must be stressed that this approach allowed the verification to proceed thanks to the small size of the transition system generated. For systems of a higher order of complexity, it is not realistic to expect that this works; the approach is not scalable. The full intermediate specification is given in Appendix III.

After this elimination of non-determinism differences, it became obvious that most of the differences had been eliminated, but not all. The remaining differences appeared to be relevant differences that really had implications for the correctness of the optimisation.

The first relevant difference is in, again, non-determinism. This time, however, the optimisation is more non-deterministic than the original. Consider the fragments in Figure 24. After a local abort has been read, an internal message is sent to the state machine DHA, which sends a message to the tc-user. Meanwhile, TSM returns a tsm_is_idle message to TCO, resulting in a free_tid action. The latter action can only be performed by TCO *after* the discard_received message has been performed. Compare this to the optimisation fragment in Figure 24.

Here, the discard_received_message action can be freely interleaved with the free_tid action. As a first solution the problem was tackled in a way similar to the non-determinism differences mentioned above, but after consulting the TCAP optimisation designer this proved to be erroneous, due to the very nature of the free_tid and discard_received_message which will be explained in the next section.

For the time being it is sufficient to know that the action free_tid and discard_received message are *not* supposed to be alternating freely. That is, the discard action must go first and the optimisation is wrong in that it allows the free_tid first. In order to correct this bug, the action that follow the local_abort message are moved to the state machine that sends the local_abort. What remains is the receiving of the local abort, for synchronisation purposes (see Figure 26)

7.4 Differences in concurrency

In the original protocol it is very well possible that a message is read from, say, the network while the previous message has not been fully processed by the system, that is, the corresponding message to the user has not been sent yet. The optimised protocol is more strict in this, in that it requires one message to be fully processed before a new message can be read.

This difference in behaviour can be explained by the fact that the original protocol has more internal 'storage space' than the optimised protocol, viz. three internal busses versus two, and the difference should not be considered a real problem, but the real problem is to isolate this difference in concurrency from other, possibly harmful, differences. The answer lies in reverting to a less concurrent environment.

Initially, the environment was specified as two independent processes, one for the TCAP user and one for the network. In this environment, it is very well possible that the user sends another message before the network has received the first message. For this concurrent and realistic environment, the optimisation is not correct (cf. Section 6).

The less concurrent environment that was subsequently used puts user and network process rigorously in sync by merging them into one process. After a user message has been sent, a corresponding network message is read, under the assumption that all possible combinations are known. However, there is one snag in this: the approach of explicitly specifying what messages are received when, includes specifying when no message is received at all. The latter cannot be specified in the version of μCRL supported by current tools. A small trick has been used to be able to specify these absence of messages nevertheless, i.e. if the TCAP protocol refrains from sending a message, it explicitly sends a no_message message. The other way round, if the environment receives such a message it infers that no message has been sent.

The second difference in concurrency is more subtle in the sense that internal actions are concerned, so that this difference is not visible to the outside world. In the original TCAP, there are situations where sending a message as a response to another message is followed by some internal processing actions. As an example, see Figure 27, where a fragment of the optimised TCAP is shown. Here, it is explicit how the sending of a tc_end_ind message to the user may be followed by a send_components action. Summarising, the internal processing extends beyond the processing visible to the outside world.

The mere difference in internal and external behaviour is not the problem here, although it makes things more complex. Take a look at the optimised fragment that corresponds to the one in Figure 28.

```
proc tco_Idle=
        r_sccp(n_abort_ind).
            (s_tsm(local_abort) + tau).
            discard_received_message +
        r_tco(tsm_is_idle).
            free_tid+

proc tsm_IS=
        r_tsm(local_abort).
            s_dha(tr_p_abort_ind).
            s_tco(tsm_is_idle).
            tsm_Idle+

proc dha_IS=
        r_dha(tr_p_abort_ind).
            s_user(tc_p_abort_ind).
            dialogue_terminated.
            free_dialogue_id.
            dha_Idle+
```

Figure 24: A problematic difference in non-determinism(original fragment)

```
proc tco_Idle=                              proc dhatsm_IS=
        r_sccp(n_abort_ind).                        r_dha(local_abort).
            (s_tsm(local_abort) + tau).                 s_user(tc_p_abort_ind).
            discard_received_message +                  dialogue_terminated.
                                                        free_dialogue_id.
                                                        free_tid.
                                                        dhatsm_Idle+
```

Figure 25: A problematic difference in non-determinism (optimisation fragment)

Though at first sight the two fragments are equivalent, a closer look reveals that the original TCAP is able to proceed with reading a new message right after the tc_continue message has been sent, whereas the optimisation first needs to finish the send_components action.

Experiments taught that the difference stuck upon is not as easy to isolate as the earlier differences in concurrency. Actually, a satisfying solution has not been found yet, since the only way of making the two equivalent requires that both the original and the optimisation are changed, by swapping the sending of the tc_continue_ind message and the send_components action, making the sending of the message the final action. This means that internal and external processing are synchronised, solving the concurrency problem. Although this 'hack' can be defended by noting that the swappings do not influence the protocol in any meaningful way, it remains a weak point in the verification.

7.5 Evaluation

The verification appeared to be more than just a final step in the TCAP optimisation life-cycle. More than once, verification had more than trivial consequences for the μCRL specifications, which needed some thoughtful rewriting for the verification to proceed.

- The environment of TCAP, modelling the SCCP and user protocol layers, was refined more than once, gradually revealing the presumptions underlying the correct use of TCAP. However, as a consequence fault-tolerance fell out of scope.
- The environment was restricted to a less concurrent one, in order to be able to verify the less concurrent optimisation.
- The specification of the optimisation was 'unfolded' in order to localise differences in concurrency.

```
proc tco_Idle=                              proc dhatsm_IS=
        r_sccp(n_abort_ind).                        r_dha(local_abort).
            (s_tsm(local_abort).                        dhatsm_Idle+
            discard_received_message.
            s_user(tc_p_abort_ind).
            dialogue_terminated.
            free_dialogue_id.
            free_tid + tau)+
```

Figure 26: The bug fix

```
proc tsm_IS=
    r_tsm(continue_rec).
        store_remote_tid.
        s_dha(tr_continue_ind).
        tsm_A+
```

```
proc dha_IS=
    r_dha(tr_continue_ind).
        (extract_dialogue_portion + tau).
        s_user(tc_continue_ind).
        (send_components + tau).
        dha_A+
```

Figure 27: concurrency (read in parallel with actions after sending)

- In one situation, the optimisation is more non-deterministic than the original. This difference, which appeared to be a real error will be addressed in Section 8.

The first two of these translate into preconditions bounding the validity of the verification. Although these are not thought of as severe limitations, it is essential that these play a crucial role in the final evaluation of this study.

The third requires some more justification, since it results in an intermediate specification that is *not* bisimulation-equivalent to the actual optimisation, though it is thought of as equivalent in an intuitive sense. The foundation for this intuitive 'equivalence' is to be found in practice. It needs to be verified whether the proposed rewritings are sound, and these are issues for which the TCAP implementor is to be consulted. This feedback is described in the next section.

8. Feedback to TCAP

We have already stressed before that checking whether the optimised version of TCAP was equivalent to the original one, has been a process of interaction and iteration. In the same line, the final result of the verification was not a 'yes' or 'no', but a modified specification of the optimisation and a list of assumptions under which this specification was equivalent to the original TCAP. The assumptions had to be checked carefully by the implementor and thereafter, the implementation could be slightly adjusted to meet the new specification. Changing the code is a matter of a few minutes, whereas more work had to be performed to check that the assumptions hold.

8.1 Handling the right message in the wrong state

The principal error found by the verification process (cf. Section 7) was the deficiency of a local_abort message with corresponding code in the *Active* state of optimised TCAP. It turned out that the code for this message was wrongly placed, viz. in the *Initiation Received* state. The presence of capturing

```
proc dhatsm_IS=
    r_tsm(continue_rec).
      store_remote_tid.
      (extract_dialogue_portion + tau).
      (s_user(tc_continue_ind).
      (send_components + tau).
      dhatsm_A +
      ⋮
```

Figure 28: concurrency (actions after sending before reading)

this local_abort in the wrong state is not as harmful as the absence of capturing it in the *Active* state. Consulting the implementor of the optimisation revealed that this error was not just another error in the μCRL specification or in the informal specification from Figure 16, since it had been replicated in the ERLANG implementation.

8.2 Incorrect increase in non-determinism

It appeared that in the original TCAP the actions free_tid and discard_received_message are always executed in the order mentioned here, while in some cases the optimisation allowed these to be executed in the reverse order.

Initially, it was assumed that this was another occurrence of a harmless difference in non-determinism, but when the implementor had explained the very meaning of these actions, the difference was concluded a relevant difference, though not necessarily an error.

The action free_tid is an act of garbage collection and discard_received_message is an act of process termination, after which no other action can be executed. It is essentially that garbage is collected before the process is terminated, since swapping the two would imply that no garbage is collected at all, resulting in a memory leak.

However, the Erlang implementation posed no problems in this respect, since the language ERLANG has its own implicit garbage collection mechanism (see [2]). So, any memory not freed explicitly by TCAP will be freed by Erlang eventually. Nevertheless, it makes sense to have corrected this shortcoming in the specification of the optimisation, to ensure a smooth implementation traject in any language other than ERLANG.

8.3 Checking order of tests and actions

In Sect. 5.2 we have already treated at length the problem of combining tests and actions induced by the difference in implementation language. In addition, the verification process brought forward that the specifications both had several sequences of actions that were more deterministic than the corresponding sequence in the other specification. As a result, the original specification and optimised specification differ in various aspects, which were felt irrelevant. However, the harmlessness of these 'rewritings' had to be validated by hand.

This validation boils down to checking independence of the pairs of action that were swapped in the verification process, i.e. checking whether the order in which these are executed is irrelevant. Superficially spoken, this checking is superfluous, since the original TCAP allows these actions to alternate freely, but there is a snag in this observation.

The complication that plays a role here, is that actions that are equal in the abstract view adhered to in the specifications, are often not identical at the lower level of detailed design. As argued in Section 5, the verification presented here focusses on concurrency problems, instead of on the details of data manipulation. Though this preassumption is felt of as valid when used to compare two traces of identically-named actions, rearranging actions is a different story, since there might be hidden interdependencies lost in the abstraction.

The actions involved were only few, as depicted in Figure 29. Careful checking by hand turned out that the precise order of these actions is not important for the present implementation.

8.4 Evaluation

Evaluating the differences found by the verification required understanding of TCAP at a lower level of abstraction than the level of the μCRL specification. Without knowledge of the details of the ERLANG implementation it was impossible to establish whether differences were specification artefacts, introduced by the language and tools used, or relevant errors.

It appeared that the 'unfolded rewritings' were indeed intuitively correct, justifying the approach chosen to locate differences in non-determinism. However, the feedback to TCAP also shed light onto a subtle error residing at a low level op abstraction, i.e. the non-commutativity of process killing and garbage collection. Though the bug located is no real bug in the context of ERLANG it is essential

s_user(tc_p_abort_ind)	discard_received_message
s_user(tc_p_abort_ind)	free_tid
dialogue_terminated	discard_received_message
dialogue_terminated	s_sccp(n_abort_req)
dialogue_terminated	free_tid
free_dialogue_id	s_sccp(n_uni_req)
free_dialogue_id	s_sccp(n_end_req)
free_dialogue_id	s_sccp(n_abort_req)
free_dialogue_id	free_tid
free_dialogue_id	discard_received_message
free_dialogue_id	assemble_uni_message
free_dialogue_id	assemble_abort_message

Figure 29: commutative pairs of actions

that it is corrected in the specification, in order to facilitate implementation in languages like C.

9. Evaluating μCRL

Just as μCRL has been used to verify the optimisation of TCAP, this same TCAP has been used to verify μCRL and the tools associated, including Aldébaran. More precise, this case study was used to assess the expressive strength of the language μCRL, and the appropriateness of the tools that were used to generate state spaces and check bisimulation equivalence.

The language μCRL appeared to be fit for the task at hand, to a large extent. It was felt that the message-driven structure of the protocol fitted neatly into the primitives offered, as can be verified from the specification fragments offered in Section 3 and the appendices, that let themselves be understood in a natural way. Complexities arose not in the specification of messages received, but in the specification of the messages *not* received.

It was explained in Section 3 that the only way to specify a properly reacting environment is the introduction of an awkward-looking no-message message. The natural solution would have been to revert to a time-out, i.e. to wait for a message for a specified amount of time, and to consider a message that does not arrive in time as not sent. The version of μCRL fully supported by tools, however, is a timeless variant of process algebra, although its superset as described in [7] does include time primitives.

Timed μCRL facilitates the explicit specification of the time after which certain actions are to happen by the '$^c t$' primitive. As an example, consider Figure 30 where it is specified that a read action is to be performed within *timeout* time steps. Though a semantics of timed μCRL has been defined, as are concepts as timed bisimulation, tool support is non-existent as yet.

The state spaces generated for TCAP appeared no real match for the high-performance instantiator tool, developed for μCRL, designed for the generation of huge state spaces containing millions of states, which is more than Cæsar/Aldébaran can handle. In the context of verification of optimisation, where issues as concurrency and non-determinism are heavily stressed, the emphasis is on the action performed, and the order in which they are performed, as opposed to the precise functionality of each

---

sum (t: Time,read^{c}t $\lhd$ t $\leq$ timeout $\rhd \delta^c 0$)

Figure 30: Time out in timed μCRL

specification		states	transitions
original	generated	958	2012
	reduced	187	358
intermediate	generated	829	1981
	reduced	187	358
optimised	generated	462	822
	reduced	159	266

Table 1: The sizes of the state spaces generated

and every action. As a consequence, the data acted upon by the various actions are left unspecified, yielding a state space that is manageable by all means, i.e. consisting of several hundreds of states (see Table 1).

The notion of weak bisimulation, accepted on intuitive grounds, could be simply checked by the Aldébaran tool. More problematic was it to trace differences found back to the μCRL specification. Given the fact that the two are separate tools, this presented the human verifier with a puzzle that appeared complex at times. However, the non-determinism difference appeared to be too subtle to be adequately covered by a concept as strict as bisimulation.

It required a good deal of effort to isolate the non-determinism difference, as was explained in Section 5, but although the specification was not so large as to make this manual checking impossible, the need for tool support was felt. Existing approaches that touch the problem experienced in this verification study are to be looked for among formalisations of the the notion of 'more non-deterministic than' such as [1] and the partial order reduction used to prune the state space in model checking [13] have been located, but the question whether this research could really play a role here has remained unanswered as yet.

A radically different alternative to the bisimulation-checking approach utilised in this verification study is to capture the TCAP behaviour in a theory in temporal logic, and to verify validity of this theory in each of the two state spaces, interpreted as a temporal model. In a technical sense, this is a feasible approach, given the fact that the Cæsar/Aldébaran supports validation of temporal formulae. The catch is that it requires a precise knowledge of the requirements to be verified, which appeared rather problematic in the study at hand. The one and only thing known about the TCAP protocol beforehand, was that the original and the optimisation are to be equivalent, and only in the course of verification the subtleties of TCAP were made explicit.

10. Conclusions and further work

Proving equivalence of the original TCAP specification and its optimised design turned out to be feasible by formally specifying both in μCRL, generating state spaces using the μCRL tool set, and checking weak bisimulation equivalence using the Cæsar/Aldébaran tool set. These techniques on their own are well established, but actually using them in real-world verification studies is not common; often, theorem proving or model-checking is used. As indicated at many places throughout this report, the verification process has been a process of interaction. From the several steps in this interaction, we have learned different things.

When transforming the SDL specification into a μCRL specification, we found several minor errors in the former specification. We noticed that two messages could be sent, but were never caught. By looking at the implementation, we noticed that the specification was in some sense language dependent, which had consequences for the specification of the optimised version, which was constructed from an implementation. Although we were able to express the part we were interested in in μCRL we would have liked to be able to express more details (such as timing and dynamic creation of processes) in μCRL.

Obtaining the specification of the optimised implementation turned out to be a time consuming task. The diagram of Figure 16 was obtained within a few days, but omitting the actions and only specifying the messages in such a drawing, is only the beginning towards a specification that can be used for our verification task. Several times after we thought to have modelled the implementation correctly, we found we had misunderstood a certain aspect. We clearly lacked a tool that could abstract from the ERLANG code and produce something closely corresponding to the μCRL specification we now have built by hand. The consequent and uniform way in which protocols are implemented in ERLANG, would make it possible to build such a tool, which is now further investigated within Ericsson.

The specification of the environment turned out to be a very tricky point. We had to restrict the possible sequences of messages and therefore we pre-assumed certain behaviour of the protocol, but it is hard to distinguish whether this restriction is to severe.

The main effort in the comparison has led in constructing the right specification for the optimised TCAP. Much effort has been put in localising differences between the original TCAP and consecutive versions of the optimised TCAP specification. Ample tool support was provided for this task, such that small changes of the specification had great impact on the work that had to be re-done for localisation of differences.

The absence of a mechanism in μCRL to detect that a process is in a state in which the environment cannot receive any message from this process, gave rise to a the no-message message solution. Except for the necessity of changing the specification at several places, this solution has as its main drawback that knowledge about the behaviour of the protocol is hard-coded in the protocol itself.

Both specifications turned out to be more deterministic than the other for some sequences of actions. Although not hard to realise, we had no tool support for identifying 'more deterministic' sequences, such that we had to expand the least deterministic sequences by hand.

Given the fact that the combination of tests and actions were specified implementation dependent, it would probably have been better to transform the tests to μCRL tests, instead of non-deterministic choices. This would have caused more specification work and an enormous increase of the state space.

The implementation of the optimised TCAP was not correct, which underpins the usefulness of our verification task. At least one serious error could be detected in the code. Several smaller mistakes had already been eliminated by the process of formalising the design. For about three man months work, Ericsson now possesses a specification of an optimised version of TCAP, which can be used in future implementations of the SS No. 7 protocol stack. Verification as part of the development and/or optimisation pays off and is more valuable than applying the techniques in a final and separate step.

Now that the specifications have been proved equivalent, we plan to investigate a refinement of the communications model and a generalisation of the environment. Interesting to know is in which respect the one-datum buffer plays a crucial role and how a more general environment influences the equivalence of the specifications.

ACKNOWLEDGEMENTS

Hans Nilsson gave us the opportunity to verify his ideas and entrusted his software to us. We are thankful for his open ear and valuable feedback during the verification process, without which the project would have broken down.

Judi Romijn read an earlier version of this report. Her comments led to a number of substantial improvements in the text.

References

1. L. Aceto. On relating concurrency and nondeterminism. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proc. Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 376–402. Springer, Berlin, 1991.
2. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in Erlang.* Prentice Hall, 2nd edition, 1996.
3. J. C. M. Baeten and J. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.
4. CWI, http://www.cwi.nl/∼mcrl/mutool.html. *The μCRL home page.*
5. Erlang, http://www.erlang.org/. *Open Source Erlang.*
6. J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (Cæsar/Aldébaran development package): A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*, pages 437–440. Springer Verlag, Aug. 1996.
7. J. F. Groote. The syntax and semantics of timed μCRL. Technical Report SEN-R9709, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, June 1997.
8. J. F. Groote and B. Lisser. *Tutorial and reference guide for the μCRL toolset verson 1.0.* CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, february 1999.
9. INRIA Rhône-Alpes, http://www.inrialpes.fr/vasy/pub/cadp/. *Cæsar/Aldébaran Development Package.*
10. International Telecommunication Union, http://www.itu.int. *CCITT specification and description language (SDL)*, 03/93 edition, March 1993.
11. International Telecommunication Union, http://www.itu.int. *Signalling System No. 7 - Transaction capabilities procedures*, 03/93 edition, March 1993.
12. R. Milner. *a calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science.* Springer Verlag, 1980.
13. D. Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Orders Methods in Verification*, DIMACS, pages 233–257, Princeton, NJ, USA, 1996. American Mathematical Society.

Appendix I
TCAP original

% %
% The specification of the original TCAP protocol
% %

% Datatype Bool

```
sort  Bool
func  T,F:→Bool
map   or,and:Bool#Bool→Bool
      not:Bool→Bool
var   b:Bool
rew   or(T,b)=T
      or(b,T)=T
      or(b,F)=b
      or(F,b)=b
      and(T,b)=b
      and(b,T)=b
      and(b,F)=F
      and(F,b)=F
      not(T)=F
      not(F)=T
      not(not(b))=b
```

% Datatypes for the natural numbers

```
sort  Nat
func  0:→Nat
      S: Nat → Nat
```

```
map   eq: Nat # Nat → Bool
var   x,y:Nat
rew   eq(0,0)=T
      eq(S(x),0)=F
      eq(0,S(x))=F
      eq(S(x),S(y))=eq(x,y)
```

% Datatype for the messages to SCCP

```
sort  sccp_message
func  n_begin_req, n_begin_ind,
      n_continue_req, n_continue_ind,
      n_uni_req, n_uni_ind,
      n_end_req, n_end_ind,
      n_abort_req, n_abort_ind,
      n_no_message :→ sccp_message
map   eq: sccp_message # sccp_message → Bool
      i:sccp_message → Nat
var   m1,m2:sccp_message
rew   eq(m1,m2)=eq(i(m1),i(m2))
      i(n_begin_ind)=0
      i(n_begin_req)=S(i(n_begin_ind))
      i(n_continue_ind)=S(i(n_begin_req))
      i(n_continue_req)=S(i(n_continue_ind))
      i(n_uni_ind)=S(i(n_continue_req))
      i(n_uni_req)=S(i(n_uni_ind))
      i(n_end_ind)=S(i(n_uni_req))
      i(n_end_req)=S(i(n_end_ind))
      i(n_abort_ind)=S(i(n_end_req))
      i(n_abort_req)=S(i(n_abort_ind))
      i(n_no_message)=S(i(n_abort_req))
```

% Datatype for the messages to tc_user

```
sort  user_message
func  tc_uni_req, tc_uni_ind,
      tc_begin_req, tc_begin_ind,
      tc_end_req, tc_end_ind,
      tc_continue_req, tc_continue_ind,
      tc_u_abort_req, tc_u_abort_ind,
      tc_p_abort_ind,
      tc_no_message :→ user_message
map   eq: user_message # user_message → Bool
      i:user_message → Nat
var   m1,m2:user_message
rew   eq(m1,m2)=eq(i(m1),i(m2))
      i(tc_uni_req)=0
      i(tc_begin_req)=S(i(tc_uni_req))
      i(tc_end_req)=S(i(tc_begin_req))
```

```
        i(tc_continue_req)=S(i(tc_end_req))
        i(tc_u_abort_req)=S(i(tc_continue_req))
        i(tc_uni_ind)=S(i(tc_u_abort_req))
        i(tc_begin_ind)=S(i(tc_uni_ind))
        i(tc_end_ind)=S(i(tc_begin_ind))
        i(tc_continue_ind)=S(i(tc_end_ind))
        i(tc_u_abort_ind)=S(i(tc_continue_ind))
        i(tc_p_abort_ind)=S(i(tc_u_abort_ind))
        i(tc_no_message)=S(i(tc_p_abort_ind))
```

% Datatype for the messages to TCO

```
sort    TCO_message
func    tr_uni_req, tr_begin_req, tr_end_req, tr_continue_req,
        tr_u_abort_req, tr_notice_req, tr_p_abort_req,
        tsm_is_idle :→ TCO_message
map     eq: TCO_message # TCO_message → Bool
        i:TCO_message → Nat
var     m1,m2:TCO_message
rew     eq(m1,m2)=eq(i(m1),i(m2))
        i(tr_uni_req)=0
        i(tr_begin_req)=S(i(tr_uni_req))
        i(tr_end_req)=S(i(tr_begin_req))
        i(tr_continue_req)=S(i(tr_end_req))
        i(tr_u_abort_req)=S(i(tr_continue_req))
        i(tr_notice_req)=S(i(tr_u_abort_req))
        i(tr_p_abort_req)=S(i(tr_notice_req))
        i(tsm_is_idle)=S(i(tr_p_abort_req))
```

% Datatype for the messages to DHA

```
sort    DHA_message
func    tr_uni_ind, tr_begin_ind, tr_end_ind, tr_continue_ind,
        tr_u_abort_ind, tr_notice_ind, tr_p_abort_ind :→ DHA_message
map     eq: DHA_message # DHA_message → Bool
        i:DHA_message → Nat
var     m1,m2:DHA_message
rew     eq(m1,m2)=eq(i(m1),i(m2))
        i(tr_uni_ind)=0
        i(tr_begin_ind)=S(i(tr_uni_ind))
        i(tr_end_ind)=S(i(tr_begin_ind))
        i(tr_continue_ind)=S(i(tr_end_ind))
        i(tr_u_abort_ind)=S(i(tr_continue_ind))
        i(tr_notice_ind)=S(i(tr_u_abort_ind))
        i(tr_p_abort_ind)=S(i(tr_notice_ind))
```

% Datatype for the messages to TSM

```
sort  TSM_message
func  begin_trans, begin_rec,
      end_trans, end_rec,
      continue_trans, continue_rec,
      abort_trans, abort_rec,
      local_abort :→ TSM_message
map   eq: TSM_message # TSM_message → Bool
      i:TSM_message → Nat
var   m1,m2:TSM_message
rew   eq(m1,m2)=eq(i(m1),i(m2))
      i(begin_trans)=0
      i(end_trans)=S(i(begin_trans))
      i(continue_trans)=S(i(end_trans))
      i(abort_trans)=S(i(continue_trans))
      i(local_abort)=S(i(abort_trans))
      i(begin_rec)=S( i(local_abort))
      i(end_rec)=S(i(begin_rec))
      i(continue_rec)=S(i(end_rec))
      i(abort_rec)=S(i(continue_rec))



% Actions

act   s_user, s-_user, cs_user,
      r_user, r-_user, cr_user: user_message
      s_sccp, s-_sccp, cs_sccp,
      r_sccp, r-_sccp, cr_sccp: sccp_message
      s_tco, s-_tco, cs_tco,
      r_tco, r-_tco, cr_tco : TCO_message
      s_tsm, s-_tsm, cs_tsm,
      r_tsm, r-_tsm, cr_tsm: TSM_message
      s_dha, s-_dha, cs_dha,
      r_dha, r-_dha, cr_dha: DHA_message

      assemble_abort_message
      assemble_begin_message
      assemble_continue_message
      assemble_end_message
      assemble_tsl_data
      assemble_uni_message
      assign_dialogue_id
      assign_local_tid
      build_aare_apdu
      build_aarq_apdu
      build_abort_message
      build_abort_apdu
      build_audt_apdu
      dialogue_terminated
      discard_components
      discard_received_message
```

```
        extract_dialogue_portion
        free_dialogue_id
        free_tid
        idle
        process_components
        request_components
        send_components
        set_application_mode
        store_local_address
        store_new_local_address
        store_remote_tid
```

% Asynchronous communications

```
comm    s-_user | s_user = cs_user
        r-_user | r_user = cr_user
        s-_sccp | s_sccp = cs_sccp
        r-_sccp | r_sccp = cr_sccp
        s-_tco | s_tco = cs_tco
        r-_tco | r_tco = cr_tco
        s-_tsm | s_tsm = cs_tsm
        r-_tsm | r_tsm = cr_tsm
        s-_dha | s_dha = cs_dha
        r-_dha | r_dha = cr_dha
```

% Processes

% Synchronous communications

```
proc    INTtco=sum(m:TCO_message,s-_tco(m)· r-_tco(m)· INTtco)
        INTtsm=sum(m:TSM_message,s-_tsm(m)· r-_tsm(m)· INTtsm)
        INTdha=sum(m:DHA_message,s-_dha(m)· r-_dha(m)· INTdha)
        EXTuser=sum(m:user_message,s-_user(m)· r-_user(m)· EXTuser)
        EXTsccp=sum(m:sccp_message,s-_sccp(m)· r-_sccp(m)· EXTsccp)
```

% State machine TCO

```
proc    tco_Idle=
            ( r_sccp(n_uni_ind)·
                (discard_received_message·
                    s_user(tc_no_message)+
                    s_dha(tr_uni_ind))+
            r_sccp(n_begin_ind)·
                (discard_received_message·
                    s_user(tc_no_message)+
                assemble_abort_message·
                    s_sccp(n_abort_req)·
```

```
            discard_received_message +
        assign_local_tid·
            (build_abort_message·
                s_sccp(n_abort_req) +
            s_tsm(begin_rec)))+
    r_sccp(n_continue_ind)·
        (discard_received_message·
                s_user(tc_no_message)+
        assemble_abort_message·
                s_sccp(n_abort_req)·
                discard_received_message+
        assemble_abort_message·
                s_sccp(n_abort_req)·
                s_tsm(local_abort)·
                discard_received_message+
        s_tsm(continue_rec))+
    r_sccp(n_end_ind)·
        (s_tsm(local_abort)·
                discard_received_message +
        discard_received_message·
                s_user(tc_no_message) +
        s_tsm(end_rec))+
    r_sccp(n_abort_ind)·
        (s_tsm(local_abort)·
                discard_received_message +
        discard_received_message·
                s_user(tc_no_message) +
        s_tsm(abort_rec))+
    r_tco(tsm_is_idle)·
        free_tid+
    r_tco(tr_uni_req)·
        assemble_uni_message·
        s_sccp(n_uni_req)+
    r_tco(tr_begin_req)·
        s_tsm(begin_trans)+
    r_tco(tr_continue_req)·
        s_tsm(continue_trans)+
    r_tco(tr_end_req)·
        s_tsm(end_trans)+
    r_tco(tr_u_abort_req)·
        s_tsm(abort_trans)
    )· tco_Idle
```

% State machine TSM in states Idle, IS, IR and A

```
proc  tsm_Idle=
    r_tsm(begin_rec)·
        store_remote_tid·
        s_dha(tr_begin_ind)·
        tsm_IR+
```

```
    r_tsm(begin_trans)·
        store_local_address·
        assemble_begin_message·
        s_sccp(n_begin_req)·
        tsm_IS

tsm_IR=
    r_tsm(continue_trans)·
        store_new_local_address·
        assemble_continue_message·
        s_sccp(n_continue_req)· tsm_A+
    r_tsm(end_trans)·
        (assemble_end_message·
            s_sccp(n_end_req) +
            s_sccp(n_no_message))·
        s_tco(tsm_is_idle)·
        tsm_Idle+
    r_tsm(abort_trans)·
        assemble_abort_message·
        s_sccp(n_abort_req)·
        s_tco(tsm_is_idle)·
        tsm_Idle

tsm_IS=
    r_tsm(continue_rec)·
        store_remote_tid·
        s_dha(tr_continue_ind)·
        tsm_A+
    r_tsm(end_rec)·
        s_dha(tr_end_ind)·
        s_tco(tsm_is_idle)·
        tsm_Idle+
    r_tsm(abort_rec)·
        (s_dha(tr_u_abort_ind)+ s_dha(tr_p_abort_ind))·
        s_tco(tsm_is_idle)·
        tsm_Idle+
    r_tsm(local_abort)·
        s_dha(tr_p_abort_ind)·
        s_tco(tsm_is_idle)·
        tsm_Idle+
    r_tsm(end_trans)·
        s_sccp(n_no_message)·
        s_tco(tsm_is_idle)·
        tsm_Idle+
    r_tsm(abort_trans)·
        s_tco(tsm_is_idle)·
        tsm_Idle

tsm_A=
    r_tsm(continue_rec)·
        s_dha(tr_continue_ind)·
```

```
            tsm_A+
        r_tsm(continue_trans)·
            assemble_continue_message·
            s_sccp(n_continue_req)·
            tsm_A+
        r_tsm(end_rec)·
            s_dha(tr_end_ind)·
            s_tco(tsm_is_idle)·
            tsm_Idle+
        r_tsm(end_trans)·
            (assemble_end_message·
                s_sccp(n_end_req)+
                s_sccp(n_no_message))·
            s_tco(tsm_is_idle)·
            tsm_Idle+
        r_tsm(abort_rec)·
            (s_dha(tr_u_abort_ind)+ s_dha(tr_p_abort_ind))·
            s_tco(tsm_is_idle)·
            tsm_Idle+
        r_tsm(local_abort)·
            s_dha(tr_p_abort_ind)·
            s_tco(tsm_is_idle)·
            tsm_Idle+
        r_tsm(abort_trans)·
            assemble_abort_message·
            s_sccp(n_abort_req)·
            s_tco(tsm_is_idle)·
            tsm_Idle
```

% State machine DHA in states Idle, IS, IR and A

```
proc  dha_Idle=
        r_user(tc_uni_req)·
            (build_audt_apdu + tau)·
            request_components·
            process_components·
            assemble_tsl_data·
            s_tco(tr_uni_req)·
            free_dialogue_id·
            dha_Idle+
        r_user(tc_begin_req)·
            (set_application_mode· build_aarq_apdu + tau)·
            request_components·
            process_components·
            assemble_tsl_data·
            assign_local_tid·
            s_tco(tr_begin_req)·
            dha_IS+
        r_dha(tr_uni_ind)·
            (assign_dialogue_id·
```

```
            s_user(tc_uni_ind)·
            send_components·
            free_dialogue_id +
        extract_dialogue_portion·
            (discard_components·
                s_user(tc_no_message)+
            assign_dialogue_id·
                s_user(tc_uni_ind)·
                send_components·
                free_dialogue_id))· dha_Idle+
    r_dha(tr_begin_ind)·
        (extract_dialogue_portion·
            (build_abort_apdu·
                discard_components·
                s_tco(tr_u_abort_req)·
                dha_Idle+
            set_application_mode·
                assign_dialogue_id·
                s_user(tc_begin_ind)·
                (send_components + tau)·
                dha_IR) +
        assign_dialogue_id·
            s_user(tc_begin_ind)·
            (send_components + tau)·
            dha_IR)

dha_IR=
    r_user(tc_continue_req)·
        (build_aare_apdu + tau)·
        request_components·
        process_components·
        assemble_tsl_data·
        s_tco(tr_continue_req)·
        dha_A+
    r_user(tc_end_req)·
        (s_tco(tr_end_req)·
            dialogue_terminated·
            free_dialogue_id·
            dha_Idle +
        (build_aare_apdu + tau)·
            request_components·
            process_components·
            assemble_tsl_data·
            s_tco(tr_end_req)·
            dialogue_terminated·
            free_dialogue_id·
            dha_Idle)+
    r_user(tc_u_abort_req)·
        (build_aare_apdu + build_abort_apdu + tau)·
        s_tco(tr_u_abort_req)·
        dialogue_terminated·
```

```
        free_dialogue_id·
        dha_Idle

dha_IS=
    r_user(tc_end_req)·
        s_tco(tr_end_req)·
        dialogue_terminated·
        dha_Idle+
    r_user(tc_u_abort_req)·
        s_tco(tr_u_abort_req)·
        dialogue_terminated·
        dha_Idle+
    r_dha(tr_end_ind)·
        (extract_dialogue_portion + tau)·
        (s_user(tc_end_ind)·
            (send_components + tau) +
        discard_components·
            s_user(tc_p_abort_ind))·
        dialogue_terminated·
        free_dialogue_id·
        dha_Idle+
    r_dha(tr_continue_ind)·
        ((extract_dialogue_portion + tau)·
            (send_components + tau)·
            s_user(tc_continue_ind)·
            dha_A +
        discard_components·
            s_user(tc_p_abort_ind)·
            build_abort_apdu·
            s_tco(tr_u_abort_req)·
            dialogue_terminated·
            free_dialogue_id·
            dha_Idle
        )+
    r_dha(tr_u_abort_ind)·
        (s_user(tc_u_abort_ind)+ s_user(tc_p_abort_ind))·
        dialogue_terminated·
        free_dialogue_id·
        dha_Idle+
    r_dha(tr_p_abort_ind)·
        s_user(tc_p_abort_ind)·
        dialogue_terminated·
        free_dialogue_id·
        dha_Idle

dha_A=
    r_user(tc_continue_req)·
        request_components·
        process_components·
        assemble_tsl_data·
        s_tco(tr_continue_req)·
```

```
        dha_A+
    r_user(tc_end_req)·
        (s_tco(tr_end_req)+
        request_components·
            process_components·
            assemble_tsl_data·
            s_tco(tr_end_req))·
        dialogue_terminated·
        free_dialogue_id·
        dha_Idle+
    r_user(tc_u_abort_req)·
        (build_abort_apdu + tau)·
        s_tco(tr_u_abort_req)·
        dialogue_terminated·
        free_dialogue_id·
        dha_Idle+
    r_dha(tr_end_ind)·
        (s_user(tc_end_ind)·
            (send_components + tau) +
        discard_components·
            s_user(tc_p_abort_ind))·
        dialogue_terminated·
        free_dialogue_id·
        dha_Idle+
    r_dha(tr_continue_ind)·
        ((send_components + tau)·
            s_user(tc_continue_ind)·
            dha_A +
        discard_components·
            s_user(tc_p_abort_ind)·
            (build_abort_apdu + tau)·
            s_tco(tr_u_abort_req)·
            dialogue_terminated·
            free_dialogue_id·
            dha_Idle)+
    r_dha(tr_u_abort_ind)·
        (s_user(tc_u_abort_ind)+ s_user(tc_p_abort_ind))·
        dialogue_terminated·
        free_dialogue_id·
        dha_Idle+
    r_dha(tr_p_abort_ind)·
        s_user(tc_p_abort_ind)·
        dialogue_terminated·
        free_dialogue_id·
        dha_Idle
```

% Successful termination

```
proc  terminate=
        idle· terminate
```

% Environment

```
proc  environment =
          s_user(tc_begin_req)·
              (r_sccp(n_begin_req)+ r_sccp(n_no_message)·
                  terminate)·
              (s_sccp(n_continue_ind)·
                  (r_user(tc_continue_ind)·
                      user_engaged+
                  r_user(tc_no_message)·
                      terminate+
                  r_sccp(n_abort_req)·
                      terminate)·
              s_sccp(n_abort_ind)·
                  (r_user(tc_p_abort_ind)+ r_user(tc_u_abort_ind)+ r_user(tc_no_message))·
                  terminate)+
          s_sccp(n_begin_ind)·
              (r_user(tc_begin_ind)+ r_user(tc_no_message)·
                  terminate +
              r_sccp(n_abort_req)·
                  terminate)·
              (s_user(tc_continue_req)·
              (r_sccp(n_continue_req)·
                  sccp_engaged+
              r_sccp(n_no_message)·
                  terminate)·
              s_user(tc_u_abort_req)·
                  (r_sccp(n_abort_req)+ r_sccp(n_no_message))·
                  terminate) +
          s_user(tc_uni_req)·
              (r_sccp(n_uni_req)+ r_sccp(n_no_message))·
              terminate+
          s_sccp(n_uni_ind)·
              (r_user(tc_uni_ind)+ r_user(tc_no_message))·
              terminate


      user_engaged =
          s_user(tc_continue_req)·
              (r_sccp(n_continue_req)·
                  engaged+
              r_sccp(n_no_message)·
                  terminate)+
          s_sccp(n_continue_ind)·
              (r_user(tc_continue_ind)·
                  user_engaged+
              r_user(tc_no_message)·
                  terminate+
              r_sccp(n_abort_req)·
```

```
            terminate)+
        s_user(tc_end_req)·
            (r_sccp(n_end_req)·
                terminate+
            r_sccp(n_no_message)·
                terminate)+
        s_user(tc_u_abort_req)·
            (r_sccp(n_abort_req)·
                terminate+
            r_sccp(n_no_message)·
                terminate)+
        s_sccp(n_abort_ind)·
            (r_user(tc_u_abort_ind)·
                terminate+
            r_user(tc_p_abort_ind)·
                terminate+
            r_user(tc_no_message)·
                terminate)


sccp_engaged =
        s_user(tc_continue_req)·
            (r_sccp(n_continue_req)·
                engaged+
            r_sccp(n_no_message)·
                terminate)+
        s_sccp(n_continue_ind)·
            (r_user(tc_continue_ind)·
                sccp_engaged+
            r_user(tc_no_message)·
                terminate+
            r_sccp(n_abort_req)·
                terminate)+
        s_sccp(n_end_ind)·
            (r_user(tc_end_ind)·
                terminate+
            r_user(tc_p_abort_ind)·
                terminate+
            r_user(tc_no_message)·
                terminate)+
        s_user(tc_u_abort_req)·
            (r_sccp(n_abort_req)·
                terminate+
            r_sccp(n_no_message)·
                terminate)+
        s_sccp(n_abort_ind)·
            (r_user(tc_u_abort_ind)·
                terminate+
            r_user(tc_p_abort_ind)·
                terminate+
            r_user(tc_no_message)·
```

```
                terminate)

    engaged =
        s_user(tc_continue_req)·
            (r_sccp(n_continue_req)·
                engaged+
            r_sccp(n_no_message)·
                terminate)+
        s_sccp(n_continue_ind)·
            (r_user(tc_continue_ind)·
                engaged+
            r_user(tc_no_message)·
                terminate+
            r_sccp(n_abort_req)·
                terminate)+
        s_user(tc_end_req)·
            (r_sccp(n_end_req)·
                terminate+
            r_sccp(n_no_message)·
                terminate)+
        s_sccp(n_end_ind)·
            (r_user(tc_end_ind)·
                terminate+
            r_user(tc_p_abort_ind)·
                terminate+
            r_user(tc_no_message)·
                terminate)+
        s_user(tc_u_abort_req)·
            (r_sccp(n_abort_req)·
                terminate+
            r_sccp(n_no_message)·
                terminate)+
        s_sccp(n_abort_ind)·
            (r_user(tc_u_abort_ind)·
                terminate+
            r_user(tc_p_abort_ind)·
                terminate+
            r_user(tc_no_message)·
                terminate)

% Initial process

init

    % Internal communications are hidden

    hide ({
            cs_tco, cr_tco,
            cs_tsm, cr_tsm,
            cs_dha, cr_dha},
```

% Isolated communications are deadlocked

encap ({ s-_user, s_user, r-_user, r_user,
s-_sccp, s_sccp, r-_sccp, r_sccp,
s-_tco,s_tco, r-_tco,r_tco,
s-_tsm,s_tsm, r-_tsm,r_tsm,
s-_dha,s_dha, r-_dha,r_dha},

% State machines, channels and environment run in parallel

tco_Idle ‖ tsm_Idle ‖ dha_Idle ‖
INTtco ‖ INTtsm ‖ INTdha ‖ EXTuser ‖ EXTsccp ‖
environment))

Appendix II
TCAP optimised

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% The specification of the optimised TCAP protocol
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %


% The specification of datatypes, actions, communications and environment
% is identical to those of the original in Appendix I


% State machine TCO

proc  tco_Idle=
        (r_sccp(n_uni_ind)·
            (discard_received_message·
                s_user(tc_no_message) +
            tau·
                (assign_dialogue_id·
                    s_user(tc_uni_ind)·
                    send_components·
                    free_dialogue_id +
                extract_dialogue_portion·
                    (discard_components·
                        s_user(tc_no_message) +
                    assign_dialogue_id·
                        s_user(tc_uni_ind)·
                        send_components·
                        free_dialogue_id)))+
        r_sccp(n_begin_ind)·
            (discard_received_message·
                s_user(tc_no_message) +
```

```
    assemble_abort_message·
        s_sccp(n_abort_req)·
        discard_received_message +
    assign_local_tid·
        (build_abort_message·
            s_sccp(n_abort_req) +
        s_tsm(begin_rec))) +
r_sccp(n_continue_ind)·
    (discard_received_message·
        s_user(tc_no_message) +
    assemble_abort_message·
        s_sccp(n_abort_req)·
        discard_received_message +
    assemble_abort_message·
        s_sccp(n_abort_req)·
        s_tsm(local_abort)·
        s_user(tc_p_abort_ind)·
        dialogue_terminated·
        discard_received_message·
        free_dialogue_id·
        free_tid+
    s_tsm(continue_rec)+
    discard_received_message·
        s_user(tc_no_message) +
    assemble_abort_message·
        s_sccp(n_abort_req)·
        discard_received_message) +
r_sccp(n_end_ind)·
    (s_tsm(local_abort)·
        s_user(tc_p_abort_ind)·
        dialogue_terminated·
        discard_received_message·
        free_dialogue_id·
        free_tid+
    s_tsm(end_rec)+
    discard_received_message·
        s_user(tc_no_message))+
r_sccp(n_abort_ind)·
    (s_tsm(local_abort)·
        s_user(tc_p_abort_ind)·
        dialogue_terminated·
        discard_received_message·
        free_dialogue_id·
        free_tid+
    s_tsm(abort_rec)+
    discard_received_message·
        s_user(tc_no_message))
)· tco_Idle
```

% State machine DHATSM with states Idle, IS, IR, A
% is a merge of the original automata DHA and TSM

```
proc  dhatsm_Idle=
          r_user(tc_uni_req)·
              (build_audt_apdu + tau)·
              request_components·
              process_components·
              assemble_tsl_data·
              assemble_uni_message·
              s_sccp(n_uni_req)·
              free_dialogue_id·
              dhatsm_Idle+
          r_user(tc_begin_req)·
              (set_application_mode· build_aarq_apdu + tau)·
              request_components·
              process_components·
              assemble_tsl_data·
              assign_local_tid·
              store_local_address·
              assemble_begin_message·
              s_sccp(n_begin_req)·
              dhatsm_IS+
          r_tsm(begin_rec)·
              store_remote_tid·
              (extract_dialogue_portion·
                  (build_abort_apdu·
                      discard_components·
                      assemble_abort_message·
                      s_sccp(n_abort_req)·
                      free_tid·
                      dhatsm_Idle +
                  set_application_mode·
                      assign_dialogue_id·
                      s_user(tc_begin_ind)·
                      (send_components + tau)·
                      dhatsm_IR) +
              assign_dialogue_id·
                  s_user(tc_begin_ind)·
                  (send_components + tau)·
                  dhatsm_IR)

      dhatsm_IR=
          r_user(tc_continue_req)·
              (build_aare_apdu + tau)·
              request_components·
              process_components·
              assemble_tsl_data·
              store_new_local_address·
              assemble_continue_message·
              s_sccp(n_continue_req)·
```

```
        dhatsm_A+
    r_user(tc_end_req)·
        (build_aare_apdu + tau)·
            request_components·
            process_components·
            assemble_tsl_data·
            (assemble_end_message·
                free_tid·
                dialogue_terminated·
                free_dialogue_id·
                s_sccp(n_end_req)+
            tau·
                dialogue_terminated·
                free_dialogue_id·
                s_sccp(n_no_message)·
                free_tid)·
            dhatsm_Idle+
    r_user(tc_u_abort_req)·
        (build_aare_apdu + build_abort_apdu + tau)·
        assemble_abort_message·
        s_sccp(n_abort_req)·
        free_tid·
        dialogue_terminated·
        free_dialogue_id·
        dhatsm_Idle

dhatsm_IS=
    r_user(tc_end_req)·
        free_tid·
        dialogue_terminated·
        dhatsm_Idle+
    r_user(tc_u_abort_req)·
        free_tid·
        dialogue_terminated·
        dhatsm_Idle+
    r_tsm(end_rec)·
        (extract_dialogue_portion + tau)·
        (s_user(tc_end_ind)·
            (send_components + tau) +
        discard_components·
            s_user(tc_p_abort_ind))·
        dialogue_terminated·
        free_dialogue_id·
        dhatsm_Idle+
    r_tsm(continue_rec)·
        store_remote_tid·
        tau·
        ((extract_dialogue_portion + tau)·
            (send_components + tau)·
            s_user(tc_continue_ind)·
            dhatsm_A +
```

```
        discard_components·
            s_user(tc_p_abort_ind)·
            build_abort_apdu·
            s_sccp(n_abort_req)·
            free_tid·
            dialogue_terminated·
            free_dialogue_id·
            dhatsm_Idle)+
    r_tsm(abort_rec)·
        (tau·
            (s_user(tc_u_abort_ind)+ s_user(tc_p_abort_ind))·
            dialogue_terminated·
            free_dialogue_id·
            free_tid+
        tau·
            s_user(tc_p_abort_ind)·
            dialogue_terminated·
            free_dialogue_id·
            free_tid)·
        dhatsm_Idle+
    r_tsm(local_abort)·
        dhatsm_Idle

dhatsm_A=
    r_user(tc_continue_req)·
        request_components·
        process_components·
        assemble_tsl_data·
        store_new_local_address·
        assemble_continue_message·
        s_sccp(n_continue_req)·
        dhatsm_A+
    r_user(tc_end_req)·
        (request_components·
                process_components·
                assemble_tsl_data+
        tau)·
        (assemble_end_message· s_sccp(n_end_req) + s_sccp(n_no_message))·
        free_tid·
        dialogue_terminated·
        free_dialogue_id·
        dhatsm_Idle+
    r_user(tc_u_abort_req)·
        (build_abort_apdu + tau)·
        assemble_abort_message·
        s_sccp(n_abort_req)·
        free_tid·
        dialogue_terminated·
        free_dialogue_id·
        dhatsm_Idle+
    r_tsm(end_rec)·
```

```
        (s_user(tc_end_ind)·
            (send_components + tau) +
        discard_components·
            s_user(tc_p_abort_ind))·
        dialogue_terminated·
        free_dialogue_id·
        free_tid·
        dhatsm_Idle+
    r_tsm(continue_rec)·
        ((send_components + tau)·
            s_user(tc_continue_ind)·
            dhatsm_A +
        discard_components·
            s_user(tc_p_abort_ind)·
            (build_abort_apdu + tau)·
            assemble_abort_message·
            s_sccp(n_abort_req)·
            free_tid·
            dialogue_terminated·
            free_dialogue_id·
            dhatsm_Idle)+
    r_tsm(abort_rec)·
        (tau·
            (s_user(tc_u_abort_ind)+s_user(tc_p_abort_ind))·
            dialogue_terminated·
            free_dialogue_id·
            free_tid+
        (tau·
            s_user(tc_p_abort_ind))·
            dialogue_terminated·
            free_dialogue_id·
            free_tid)·
        dhatsm_Idle+
    r_tsm(local_abort)·
        dhatsm_Idle
```

% Successful termination

proc terminate=
idle· terminate

% Initial process

init

% Internal communications are hidden

hide ({ cs_tsm, cr_tsm},

```
% Isolated communications are deadlocked

encap  ({ s-_user, s_user, r-_user, r_user,
          s-_sccp, s_sccp, r-_sccp, r_sccp,
          s-_tsm,s_tsm, r-_tsm,r_tsm},

% State machines, channels and environment run in parallel

tco_Idle || dhatsm_Idle ||
INTtsm || EXTuser || EXTsccp ||
environment ))
```

Appendix III
TCAP rewritten

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% The specification of the optimised TCAP protocol
% rewritten to cover differences in non-determinism
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %


% The specification of datatypes, actions, communications and environment
% is identical to those of the original in Appendix I

% State machine TCO

proc  tco_Idle=
          (r_sccp(n_uni_ind)·
              (discard_received_message·
                  s_user(tc_no_message) +
              tau·
                  (assign_dialogue_id·
                      s_user(tc_uni_ind)·
                      send_components·
                      free_dialogue_id +
                  extract_dialogue_portion·
                      (discard_components·
                          s_user(tc_no_message) +
                      assign_dialogue_id·
                          s_user(tc_uni_ind)·
                          send_components·
                          free_dialogue_id)))+
          r_sccp(n_begin_ind)·
              (discard_received_message·
                  s_user(tc_no_message) +
```

```
    assemble_abort_message·
        s_sccp(n_abort_req)·
        discard_received_message +
    assign_local_tid·
        (build_abort_message·
            s_sccp(n_abort_req) +
        s_tsm(begin_rec))) +
r_sccp(n_continue_ind)·
    (discard_received_message·
        s_user(tc_no_message) +
    assemble_abort_message·
        s_sccp(n_abort_req)·
        discard_received_message +
    assemble_abort_message·
        s_sccp(n_abort_req)·
        (s_tsm(local_abort)·
            (discard_received_message·
                (s_user(tc_p_abort_ind)·
                    (dialogue_terminated·
                        (free_dialogue_id·
                            free_tid +
                        free_tid·
                            free_dialogue_id)+
                    free_tid·
                        dialogue_terminated·
                        free_dialogue_id)+
                free_tid·
                    s_user(tc_p_abort_ind)·
                    dialogue_terminated·
                    free_dialogue_id) +
            s_user(tc_p_abort_ind)·
                (discard_received_message·
                    (dialogue_terminated·
                        (free_dialogue_id·
                            free_tid +
                        free_tid·
                            free_dialogue_id)+
                    free_tid·
                        dialogue_terminated·
                        free_dialogue_id) +
                dialogue_terminated·
                    (discard_received_message·
                        (free_dialogue_id·
                            free_tid +
                        free_tid·
                            free_dialogue_id)+
                    free_dialogue_id·
                        discard_received_message·
                        free_tid ))))+
    s_tsm(continue_rec) +
    discard_received_message·
```

```
          s_user(tc_no_message) +
     assemble_abort_message·
          s_sccp(n_abort_req)·
          discard_received_message)+
r_sccp(n_end_ind)·
          (s_tsm(local_abort)·
          (discard_received_message·
               (s_user(tc_p_abort_ind)·
                    (dialogue_terminated·
                         (free_dialogue_id·
                              free_tid +
                         free_tid·
                              free_dialogue_id)+
                    free_tid·
                         dialogue_terminated·
                         free_dialogue_id)+
               free_tid·
                    s_user(tc_p_abort_ind)·
                    dialogue_terminated·
                    free_dialogue_id) +
          s_user(tc_p_abort_ind)·
               (discard_received_message·
                    (dialogue_terminated·
                         (free_dialogue_id·
                              free_tid +
                         free_tid·
                              free_dialogue_id)+
                    free_tid·
                         dialogue_terminated·
                         free_dialogue_id) +
               dialogue_terminated·
                    (discard_received_message·
                         (free_dialogue_id·
                              free_tid +
                         free_tid·
                              free_dialogue_id)+
                    free_dialogue_id·
                         discard_received_message·
                         free_tid )))+
          s_tsm(end_rec)+
          discard_received_message·
               s_user(tc_no_message))+
r_sccp(n_abort_ind)·
     (s_tsm(local_abort)·
          (discard_received_message·
               (s_user(tc_p_abort_ind)·
                    (dialogue_terminated·
                         (free_dialogue_id·
                              free_tid +
                         free_tid·
                              free_dialogue_id)+
```

```
                        free_tid·
                            dialogue_terminated·
                            free_dialogue_id)+
                    free_tid·
                        s_user(tc_p_abort_ind)·
                        dialogue_terminated·
                        free_dialogue_id) +
                s_user(tc_p_abort_ind)·
                    (discard_received_message·
                        (dialogue_terminated·
                            (free_dialogue_id·
                                free_tid +
                            free_tid·
                                free_dialogue_id)+
                        free_tid·
                            dialogue_terminated·
                            free_dialogue_id) +
                    dialogue_terminated·
                        (discard_received_message·
                            (free_dialogue_id·
                                free_tid +
                            free_tid·
                                free_dialogue_id)+
                        free_dialogue_id·
                            discard_received_message·
                            free_tid )))+
            s_tsm(abort_rec)+
            discard_received_message·
                s_user(tc_no_message))
        )· tco_Idle
```

% State machine DHATSM with states Idle, IS, IR, A
% is a merge of the original automata DHA and TSM

```
proc  dhatsm_Idle=
        r_user(tc_uni_req)·
            (build_audt_apdu + tau)·
            request_components·
            process_components·
            assemble_tsl_data·
            (free_dialogue_id·
                assemble_uni_message·
                s_sccp(n_uni_req)+
            assemble_uni_message·
                (free_dialogue_id· s_sccp(n_uni_req)+
                s_sccp(n_uni_req)· free_dialogue_id))·
            dhatsm_Idle+
        r_user(tc_begin_req)·
            (set_application_mode· build_aarq_apdu + tau)·
            request_components·
```

```
        process_components·
        assemble_tsl_data·
        assign_local_tid·
        store_local_address·
        assemble_begin_message·
        s_sccp(n_begin_req)·
        dhatsm_IS+
    r_tsm(begin_rec)·
        store_remote_tid·
        (extract_dialogue_portion·
            (build_abort_apdu·
                discard_components·
                assemble_abort_message·
                s_sccp(n_abort_req)·
                free_tid·
                dhatsm_Idle+
            set_application_mode·
                assign_dialogue_id·
                s_user(tc_begin_ind)·
                (send_components + tau)·
                dhatsm_IR)+
        assign_dialogue_id·
            s_user(tc_begin_ind)·
            (send_components + tau)·
            dhatsm_IR)

dhatsm_IR=
    r_user(tc_continue_req)·
        (build_aare_apdu + tau)·
        request_components·
        process_components·
        assemble_tsl_data·
        store_new_local_address·
        assemble_continue_message·
        s_sccp(n_continue_req)·
        dhatsm_A+
    r_user(tc_end_req)·
        (build_aare_apdu + tau)·
            request_components·
            process_components·
            assemble_tsl_data·
            (assemble_end_message·
                (s_sccp(n_end_req)·
                    (free_tid· dialogue_terminated· free_dialogue_id+
                    dialogue_terminated·
                        (free_dialogue_id· free_tid+
                        free_tid· free_dialogue_id ))+
                    dialogue_terminated·
                        (free_dialogue_id· s_sccp(n_end_req)· free_tid+
                        s_sccp(n_end_req)·
                            (free_dialogue_id· free_tid+free_tid· free_dialogue_id)))+
```

```
        dialogue_terminated·
            (tau·
                (free_tid· free_dialogue_id+free_dialogue_id· free_tid )·
                s_sccp(n_no_message)+

            free_dialogu%e_id·
                (assemble_end_message· %
                    s_sccp(n_end_req)+ %
                s_sccp(n_no_message))·
                free_tid+

            assemble_end_message·
                (free_dialogue_id· s_sccp(n_end_req)· free_tid+
                s_sccp(n_end_req)·
                    (free_dialogue_id· free_tid+free_tid· free_dialogue_id)))+

            tau·
                (free_tid· dialogue_terminated· free_dialogue_id+
                dialogue_terminated·
                    (free_dialogue_id· free_tid+ free_tid· free_dialogue_id))·
                s_sccp(n_no_message))·
        dhatsm_Idle+

r_user(tc_u_abort_req)·
    (build_aare_apdu + build_abort_apdu + tau)·
        (dialogue_terminated·
            (assemble_abort_message·
                (free_dialogue_id·
                    s_sccp(n_abort_req)·
                    free_tid+
                s_sccp(n_abort_req)·
                    (free_dialogue_id·
                        free_tid+
                    free_tid·
                        free_dialogue_id))+
            free_dialogue_id·
                assemble_abort_message·
                s_sccp(n_abort_req)·
                free_tid)+
        assemble_abort_message·
            (dialogue_terminated·
                (s_sccp(n_abort_req)·
                    (free_dialogue_id·
                        free_tid+
                    free_tid·
                        free_dialogue_id)+
                free_dialogue_id·
                    s_sccp(n_abort_req)·
                    free_tid)+
            s_sccp(n_abort_req)·
```

```
                (dialogue_terminated·
                    (free_tid·
                        free_dialogue_id+
                    free_dialogue_id·
                        free_tid)+
                free_tid·
                    dialogue_terminated·
                    free_dialogue_id)))·
        dhatsm_Idle

dhatsm_IS=
    r_user(tc_end_req)·
        (free_tid· dialogue_terminated+
        dialogue_terminated· free_tid)·
        dhatsm_Idle+
    r_user(tc_u_abort_req)·
        (free_tid· dialogue_terminated+
        dialogue_terminated· free_tid)·
        dhatsm_Idle+
    r_tsm(end_rec)·
        (extract_dialogue_portion + tau)·
        (s_user(tc_end_ind)·
            (send_components + tau) +
        discard_components·
            s_user(tc_p_abort_ind))·
        dialogue_terminated·
        free_dialogue_id·
        dhatsm_Idle+
    r_tsm(continue_rec)·
        store_remote_tid·
        tau·
        ((extract_dialogue_portion + tau)·
            (send_components + tau)·
            s_user(tc_continue_ind)·
            dhatsm_A +
        discard_components·
            s_user(tc_p_abort_ind)·
            build_abort_apdu·
            (assemble_abort_message·
                (dialogue_terminated·
                    (free_dialogue_id· s_sccp(n_abort_req)· free_tid+
            s_sccp(n_abort_req)·
                (free_tid· free_dialogue_id+free_dialogue_id· free_tid))+
            s_sccp(n_abort_req)·
                (dialogue_terminated·
                    (free_dialogue_id· free_tid+free_tid· free_dialogue_id)+
                free_tid· dialogue_terminated· free_dialogue_id))+

            dialogue_terminated·
                (free_dialogue_id· assemble_abort_message· s_sccp(n_abort_req)· free_tid+
                assemble_abort_message·
```

```
                (free_dialogue_id· s_sccp(n_abort_req)· free_tid+
                s_sccp(n_abort_req)·
                    (free_tid· free_dialogue_id+free_dialogue_id· free_tid)))·
        dhatsm_Idle)+

    r_tsm(abort_rec)·
        (tau·
            ((s_user(tc_u_abort_ind)+ s_user(tc_p_abort_ind))·
                (free_tid· dialogue_terminated· free_dialogue_id+
            dialogue_terminated·
                (free_tid· free_dialogue_id+free_dialogue_id· free_tid))+
            free_tid·
                (s_user(tc_u_abort_ind)+ s_user(tc_p_abort_ind))·
                dialogue_terminated·
                free_dialogue_id) +
        tau·
            (s_user(tc_p_abort_ind)·
                (free_tid· dialogue_terminated· free_dialogue_id+
                dialogue_terminated·
                    (free_tid· free_dialogue_id+free_dialogue_id· free_tid))+
            free_tid·
                s_user(tc_p_abort_ind)·
                dialogue_terminated·
                free_dialogue_id))·
                dhatsm_Idle+
    r_tsm(local_abort)·
        dhatsm_Idle


dhatsm_A=
    r_user(tc_continue_req)·
        request_components·
        process_components·
        assemble_tsl_data·
        assemble_continue_message·
        s_sccp(n_continue_req)·
        dhatsm_A+
    r_user(tc_end_req)·
        (request_components·
            process_components·
            assemble_tsl_data + tau)·
        (assemble_end_message·
            (s_sccp(n_end_req)·
                (free_tid· dialogue_terminated· free_dialogue_id+
                dialogue_terminated·
                    (free_dialogue_id· free_tid+free_tid· free_dialogue_id))+
        dialogue_terminated·
            (free_dialogue_id· s_sccp(n_end_req)· free_tid+
            s_sccp(n_end_req)·
                (free_dialogue_id· free_tid+free_tid· free_dialogue_id)))+
```

```
    dialogue_terminated·
        (s_sccp(n_no_message)·
            (free_tid· free_dialogue_id+free_dialogue_id· free_tid)+
        free_dialogue_id·
            (assemble_end_message·
                s_sccp(n_end_req)+
            s_sccp(n_no_message))·
            free_tid+

        assemble_end_message·
            (free_dialogue_id· s_sccp(n_end_req)· free_tid+
            s_sccp(n_end_req)·
                (free_dialogue_id· free_tid+free_tid· free_dialogue_id)))+

    s_sccp(n_no_message)·
        (free_tid· dialogue_terminated· free_dialogue_id+
        dialogue_terminated·
            (free_dialogue_id· free_tid+ free_tid· free_dialogue_id)))·

    dhatsm_Idle+

r_user(tc_u_abort_req)·
    (build_abort_apdu + tau)·
    (assemble_abort_message·
        (dialogue_terminated·
        (free_dialogue_id·
            s_sccp(n_abort_req)·
            free_tid+
        s_sccp(n_abort_req)·
            (free_tid· free_dialogue_id+free_dialogue_id· free_tid))+

        s_sccp(n_abort_req)·
            (dialogue_terminated·
                (free_tid· free_dialogue_id+free_dialogue_id· free_tid)+
            free_tid· dialogue_terminated· free_dialogue_id))+

    dialogue_terminated·
        (assemble_abort_message·
            (free_dialogue_id·
                s_sccp(n_abort_req)·
                free_tid+
            s_sccp(n_abort_req)·
                (free_tid· free_dialogue_id+free_dialogue_id· free_tid))+

        free_dialogue_id·
            assemble_abort_message·
            s_sccp(n_abort_req)·
            free_tid))·
    dhatsm_Idle+
```

```
r_tsm(end_rec)·
    (s_user(tc_end_ind)·
        (((send_components + tau)·
            (dialogue_terminated·
                (free_dialogue_id· free_tid+free_tid· free_dialogue_id)+
            free_tid·
                dialogue_terminated·
                free_dialogue_id))+

        free_tid·
            (send_components + tau)·
            dialogue_terminated·
            free_dialogue_id)+

        discard_components·
            (s_user(tc_p_abort_ind)·
                (free_tid·
                    dialogue_terminated·
                    free_dialogue_id+
                dialogue_terminated·
                    (free_tid· free_dialogue_id+free_dialogue_id· free_tid))+
                free_tid·
                    s_user(tc_p_abort_ind)·
                    dialogue_terminated·
                    free_dialogue_id)+

        free_tid·
            (discard_components·
                    s_user(tc_p_abort_ind)·
                    dialogue_terminated·
                    free_dialogue_id+

            s_user(tc_end_ind)·
                    (send_components + tau)·
                    dialogue_terminated·
                    free_dialogue_id))·

    dhatsm_Idle+
r_tsm(continue_rec)·
    ((send_components + tau)·
        s_user(tc_continue_ind)·
        dhatsm_A +
    discard_components·
        s_user(tc_p_abort_ind)·
        (build_abort_apdu + tau)·
        (assemble_abort_message·
            (dialogue_terminated·
                (free_dialogue_id·
                    s_sccp(n_abort_req)·
                        free_tid+
                s_sccp(n_abort_req)·
```

```
                    (free_tid· free_dialogue_id+
                    free_dialogue_id· free_tid))+
                s_sccp(n_abort_req)·
                    (dialogue_terminated·
                            (free_tid· free_dialogue_id+
                            free_dialogue_id· free_tid)+
                            free_tid· dialogue_terminated· free_dialogue_id))+
                dialogue_terminated·
                    (free_dialogue_id·
                        assemble_abort_message·
                        s_sccp(n_abort_req)·
                        free_tid+
                    assemble_abort_message·
                        (s_sccp(n_abort_req)·
                            (free_tid· free_dialogue_id+
                            free_dialogue_id· free_tid)+
                        free_dialogue_id· s_sccp(n_abort_req)· free_tid)))·
    dhatsm_Idle)+

r_tsm(abort_rec)·
    (tau·
        ((s_user(tc_u_abort_ind)+ s_user(tc_p_abort_ind))·
        (free_tid· dialogue_terminated· free_dialogue_id+
        dialogue_terminated·
            (free_tid· free_dialogue_id+free_dialogue_id· free_tid))+
        free_tid·
            (s_user(tc_u_abort_ind)+ s_user(tc_p_abort_ind))·
            dialogue_terminated·
            free_dialogue_id) +

    tau·
        (s_user(tc_p_abort_ind)·
        (free_tid· dialogue_terminated· free_dialogue_id+
            dialogue_terminated·
            (free_tid· free_dialogue_id+free_dialogue_id· free_tid))+
        free_tid·
            s_user(tc_p_abort_ind)·
            dialogue_terminated·
            free_dialogue_id))·
    dhatsm_Idle+

r_tsm(local_abort)·
    dhatsm_Idle
```

% Successful termination

proc terminate=

```
        idle· terminate
```

% Initial process

init

% Internal communications are hidden

hide ({cs_tsm, cr_tsm},

% Isolated communications are deadlocked

encap ({ s-_user,s_user, r-_user,r_user,
s-_sccp,s_sccp, r-_sccp,r_sccp,
s-_tsm,s_tsm, r-_tsm,r_tsm},

% State machines, channels and environment run in parallel

tco_Idle ∥ dhatsm_Idle ∥
INTtsm ∥ EXTuser ∥ EXTsccp ∥
environment))