Specification of Software Component Requirements Using the Trace Function Method

Colm Quinn, Sergiy Vilkomir and David Parnas
Software Quality Research Laboratory (SQRL)
Department of Computer Science and Information Systems
University of Limerick, Ireland
Email: colm.quinn@ul.ie

Srdjan Kostic
LMI Ericsson, Ireland
Radio house, Belfield Office Park
Clonskeagh, Dublin 4, Ireland
Email: srdjan.i.kostic@ericsson.com

***Abstract*— This paper describes the application of the Trace Function Method to specify the requirements of a software component. We illustrate the method on a software component of a telecommunications system that was developed by Ericsson. Beginning with incomplete informal descriptions, we analysed the requirements of the system and wrote a description that contains all pertinent information in one easily used reference document. The resulting documentation is more compact and complete than traditional software documentation and provides precise information that will be useful for testing and inspection.**

I. Introduction

The lack of precise, well-organised, and easily used documentation is one of the problems that has plagued the software industry since its birth. Much development time is consumed because documentation is incomplete, imprecise, inconsistent and poorly organised. This paper introduces a method by which complex components can be precisely summarised in useful ways. The method is an outgrowth of earlier research on mathematical specifications. In comparison to earlier work, this work emphasizes on readability. However, our goal is not to produce introductory material. Instead, we want to produce reference documents - documents in which someone with general familiarity with a system can quickly find detailed information. These detailed documents can also be used in testing and inspection [12].

The most basic task in documenting an existing system is deciding what information must be in the document and then finding it. It is not enough to simply write statements that describe the facts that one encounters; we describe a method that first raises questions. If those answers are easily answered, the task is easy. In other cases one must resort to experts and examination of the code to get the answers. The process of raising the questions is the key to producing a complete and consistent document.

In Section II we give the historical background to the method. In Section III we describe the system that we use in our case study. Section IV describes the Trace Function Method (TFM). A description of the case study using the TFM is presented in Section V with a formal specification in Appendix. Section VI presents our conclusions.

II. Historical Development

In the early seventies there was a great deal of research in the area of algebraic [3], [4], [5], [14] and axiomatic [11] specification with algebraic methods proving more popular. Based on this work, Bartussek and Parnas [2] developed a method known as Trace Assertion Method (TAM) that removed certain limitations that existed in the previous approaches. Further work continued with notable contributions from Hoffman [7], Iglewski, Madey and Stencel [8], Janicki and Sekerinski [9], Kubica [10], Stencel [13], Wang [15] and others. While the TAM had some advantages over the previous work, it was not designed to handle communication via global variables . Most importantly, it was counter-intuitive for most developers. In this paper we introduce a new approach, namely the Trace Function Method (TFM), which addresses these concerns. Like the later versions of TAM, TFM uses tabular expressions and the concept of traces of events to produce complete specifications and descriptions. This paper illustrates how to produce precise documentation using the TFM and shows that compact, easily used reference documentation can be distilled from a collection of unstructured imprecise documents. Simpler practical application of this method is presented in [1].

III. A Case Study: Software for Telecommunications System

A. The Case Study

We have used the TFM to document part of a telecommunications system developed by Ericsson. This product has seen extensive development, implementation and testing in both Sweden and Ireland, the system has gone through many versions. To date the product has been rolled out as part of 3G Radio Access Network in more than 20 3G mobile networks.

The Radio Network Controller (RNC) is part of the 3G providers network infrastructure. The RNC performs radio resource management as defined in the 3GPP specification document [16]. To communicate on the network, a mobile unit must first contact the RNC, establish a signalling connection and then request resources from the network. The RNC then handles all communication between the mobile unit and the providers network. The RNC has the power to tailor the network for best efficiency. It forms a central point for managing

Proceedings of the International Conference
on Software Engineering Advances (ICSEA'06)
0-7695-2703-5/06 $20.00 © 2006 IEEE

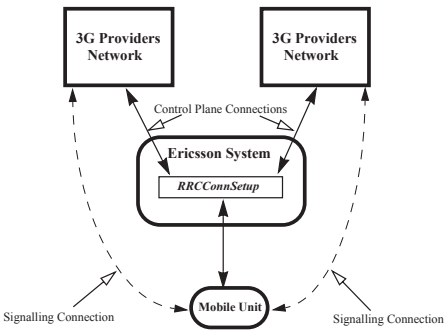

Fig. 1. System Communication

all connections, setting up new connections, dropping faulty connections and transferring connections from one node to another.

We have studied and documented the RRC Connection Establishment module (RRCConnEst), which is in charge of establishing an RRC connection for control signalling between a mobile unit and the system. A complete discussion of RRCConnEst can be found in the 3GPP specification [17].

For each mobile unit there must be one RRC signalling connection to the mobile unit. Figure 1 illustrates the connections that are established between the providers network (of which there may be one or more), the system, and the mobile unit.

RRCConnEst makes requests to other procedures in the system. Based on the arguments passed and the status of various data structures, the other procedure decide to grant or reject the request made. A message is sent back to RRCConnEst to indicate the result. After receiving a confirm message another request will be sent until an RRC connection is established. In the case of a rejection, there are several ways for the system to deallocate resources and retry. In addition, flags and global data structures are set. The flags, if set, change the standard flow of the confirm/reject messages. In addition the data structures mentioned above are global structures that can be read, and in some cases changed, by any other procedure in the system. These structures store information that is used by many procedures and certain configuration data that are set at start-up.

B. Detailed Description of RRCConnEst Operation

As stated above, RRCConnEst is specified as performing three possible actions:

- Successfully allocating resources and establishing a connection.
- Retrying a connection attempt.
- Terminating the connection attempt.

The successful connection setup process is logical, with many intermediate steps that relate to the setup of some section of the connection that must be established prior to full operation. In comparison, the retry mode has fewer intermediate

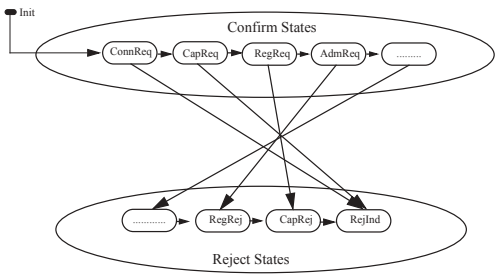

Fig. 2. System Modes

states but many distinct library functions[1] can cause it to be triggered. These intermediate steps relate to the release of various resources that have already been allocated in the earlier intermediate steps. As a result, all resources up to that point have to be de-allocated. This results in a sequence of steps that are the reverse of the order of the confirm steps. Figure 2 illustrates this situation in more detail.

A connection indicator (Init) is received. This results in the procedure sending a ConnRequest signal to check if it is allowed to establish a connection at this time. If confirmation is received , the next success state can be processed (CapReq). However, if a failure message is received, the RejInd state is reached and RejInd is sent to notify the caller that a connection cannot be established at this time. If, for example, we have reached AdmReq, the failure state will be ReqRej, followed by CapRej and RejInd states.

A set of messages link these steps. A strict ordering to the input sequences can be expected. Since the system has a one to one mapping between inputs and outputs, we can deduce the resulting outputs. Using this information, we list the seventeen confirm messages in the order we expect them (rrcConnRequest, capacityCfm, ..., rrcMsgDlCfm). We observe the following characteristics:

- Messages occur and are received in a fixed order in the majority of cases.
- It is possible that asynchronous messages can be received at any time. In these scenarios the procedure will release the connections.

In the case of the reject messages we have a list of seven conditions that may be reached (failSpRelease, ..., failReject). Each condition in turn is valid for a number of messages. Table VI shows this set. It should be noted, due to space constraints, that it is not possible to give all tables.

There are a number of override flags that influence the behaviour of the procedure. These indicate asynchronous unpredictable occurrences within the system as a whole. In the presence of success messages (e.g. CapacityCfm) the override flags can result in the system releasing instead of continuing connection setup as expected. To account for this, the standard

[1]Library functions are functions that can be invoked from outside the module.

messages as listed above are modified to include a separate flag element with each message.

The procedure under review uses the information received with each message to change various global data structures to reflect the new state of the system. These data structures are used later when the system tries to communicate with the mobile device to perform further services. This information is not discussed in this paper.

C. Previously available documentation

The existing extensive documentation was considered to be in a form better than industry standard and a suitable base to start our analysis. This documentation consisted of a conventional description of the operation of the procedure supplemented by graphical annotations based on Rational Rose sequence diagrams. The document was divided into sections by their logical function and appeared complete. We expected, and found, issues of inconsistency and interpretation of the natural language, but in general the quality of the documentation was high. However, the mathematical model behind the TFM approach forced us to try to answer questions about things that may have been considered "obvious" or unimportant by the developers. When we tried to find the answers to those questions, we found that the documentation was not as complete as was originally thought. Some information on data types and ranges was missing; other information was common "background knowledge". In addition, information about flags and other data was inadequate.

To compile a full list of the information required, a structured approach was required for the information-gathering phase. A transition table was used to show each mode[2] , its inputs, outputs and the next mode that was reached upon successful execution. Because of the tabular nature of the representation, unexplored and overlooked cases were exposed clearly and quickly. The final table, while bulky, provided a full specification of the component. However we needed to convert this table to one that could be easily used as a reference on the module. Further, we wanted to eliminate the rather arbitrary state representation information.

IV. Trace Function Method

The TFM is an interface specification method based on earlier algebraic, axiomatic and trace-based approaches. This approach appears to be more intuitive for the practitioner and is fully integrated with the other approaches being developed by the SQRL research group in the University of Limerick [1]. The TFM is intended for use in describing components with a hidden data structure that should not be mentioned in a specification (since it is subject to change). The TFM can describe components that communicate by means of global or shared variables if they are considered part of its interface with external software or hardware. Global variables are also used to invoke programs that are part of the interface.

We define an event as a change in the value of one or more global variables. The invocation of one of the interface procedure functions is also considered an event. The name of the procedure being invoked is treated as the value of an input variable.

An event descriptor describes values of all variables before and after the event. An abbreviated event descriptor only lists the values for variables that are changed/accessed during the event. We treat an input as a variable that influences the behaviour of the component to be specified and an output as a variable that is changed by the component.

A Trace is a sequence of event descriptors beginning immediately after the creation of the object being specified. Traces describe the history of a procedure or object. The use of traces allows us to eliminate any representation or model of the internal state information. For any deterministic component, the value of an output after an event can be described as a function of the trace that describes the sequence of events that affect that component.

A TFM specification consists of:

- A list of the input and output variables, including shared global variables and their types.
- A set of output function definitions, specifying the value of each of the output variables as a function of the trace of the components history.
- A set of auxiliary function definitions used in the output function definitions.

There is one output function for each variable; each of these has a domain that includes all possible histories and a range that comprises all possible output values. We define auxiliary functions to simplify the description of our output functions. This usually avoids repetition and improves readability.

The data structure and the syntax of invocable programs are usually described using the notion of the programming language. Of note is that we deal only with procedures with deterministic behaviour. The following notation is proposed:

- We denote a trace as T.
- A trace can be either empty ("_") or to contain one or more elements.
- Where there are more than one elements in a trace we use "." to denote the concatenation e.g. T = [e1.e2].
- X(T) is an output function where X is the function name that returns the expected output message as a result of the proceeding trace.

In addition there is a defined set of "standard" functions on traces:

- last(T) - the most recent element in the trace.
- rest(T) - the remainder of T after removing last(T).

For example, if T has four elements (e1.e2.e3.e4), e1 was the first event after initialization/creation, last(T) = [e4] and rest(T) = [e1.e2.e3].

[2]The number of states in such a system is far too large to enumerate. Consequently, we partition the states into classes of states, called modes [6]. If the partition is properly done,

one can then describe the mode transitions using tables that look just like state transition tables.

V. Application of the TFM to RRCConnEst

A partial formal TFM specification of the case study procedure is provided in Appendix. Because of space constraints and the confidential nature of the material, only a subset of the specification can be included in this paper. The tables presented are examples of the method used and illustrate the application of the TFM method. To give some perspective, the procedure provided by Ericsson is in the region of 20 thousand lines of code generated from a UML model. As a reference the original Ericsson specification was 145 pages and did not include all the information provided above. In comparison the complete TFM specification is 50 pages with all information in one source.

In this section we explain:

- The mapping between formal variables, functions etc. and the actual physical situation.
- How to read the TFM specification.

We define event descriptor classes to represent all possible values that the event descriptors may have. For our case study, an event is an input message. The event descriptor is described by listing all variables (both type and range) that are changed or accessed as a result of a message being received. Tables I and II in Appendix are examples of descriptions of these classes.

We denote the set of all 17-confirmation event descriptor classes by CONFIRM. A full definition of this set can be seen in Table V. We organise the reject descriptors that have similar behaviour into 8 groups (see Tables III and IV for a sample of 2 of the possible 8 groups). The set of these 8 groups is denoted as REJECT (see Table VI).

To describe the temporal order of these event descriptors, we define the functions conf and rej. The signature of function conf is:

- $conf : 1, 2, ..., 17 \rightarrow CONFIRM$

Given the sequence number of an event, the function returns the name of the event. A full definition can be seen in Table XI. Similarly we have a function rej with signature:

- $rej : 1, 2, ..., 8 \rightarrow REJECT$

A full definition is given in Table VII. Each group in turn has an associated set of event descriptor classes. We assume only one event from each group will be present on the trace, if at all.

We use these functions to describe elements simplify our specification. Confirmation messages appear in a trace in increasing order, for example: conf(1).conf(2).conf(3).conf(4). In contrast, reject messages appear on a trace in decreasing order, for example: conf(1).conf(2).conf(3).conf(4).rej(3).rej(2).rej(1).

To create trace specifications, we also need to refer to the number of a message knowing its name. We define the function:

- $num : CONFIRM \cup REJECT \rightarrow 1, 2, ...17$

num is the union of the inverse functions of functions conf and rej. It maps the name of an event descriptor class in $CONFIRM \cup REJECT$ to its representative number. The definition of this function is given in Table XII.

Any reject message can be the first reject message in the trace but only after specific confirmation messages. For example, rej(1) can be the first reject message only after conf(1) or conf(2).We denote prev-conf(rej(i)) as a function that returns for every rej(i) a set of the confirmation messages that must appear before rej(i) can appear as the first reject message. **P**(X) is a set of subsets of X.

- $prev\_conf : REJECT \rightarrow \mathbf{P}(CONFIRM)$

A full definition can be seen in Table VIII.

We denote the set of all 25 output messages as OUTPUT (Table IX). The function out (Table X) is defined with signature:

- $out : CONFIRM \cup REJECT \rightarrow \mathbf{P}(OUTPUT)$

This returns the output corresponding to an input event. To determine if the situation is normal or unexpected, function O(T) is used (see Table XIII). This is defined as:

- $O : TRACES \rightarrow \mathbf{P}(OUTPUT) \cup Error$

To find the output for any given situation, we need to use two tables. The table for function O(T) is used to decide if the situation is expected. For expected situations (valid inputs) the value of O(T) is out(last(T)) and we use the table for the out auxiliary function to find the corresponding output message. For unexpected situations (invalid inputs), O(T) generates an Error output message.

The function O(T) specifies an output for every possible trace. In each of the tables cells we have a predicate; the table cells are connected by a logical " $\wedge$ ". The conjunction describes the condition for each possible output. For every specific trace, we look for a corresponding condition and then lookup the output. Where several table cells in a column contain the same information, they are merged for ease of reading. Thus, for example, the first condition in the Table XIII shows that when $last(T) \in CONFIRM \wedge last(T) = conf(1) \wedge rest(T) = \_$ then $O(T) = out(last(T)$. This organization makes it easy to be sure that the table is complete and consistent. In combination with the auxiliary functions, the allowed behaviour of the system can be seen in a very compact and ordered form. This is in contrast to the original natural language and the Rational Rose model.

VI. Conclusions

To produce the formal document, several informal and disparate specification documents were reviewed and compiled into one source. The rigor necessary to compile this information revealed inconsistencies and gaps in the original specification and documentation. The resulting specification proved to be shorter, clearer and to have enough precision to be a useful input to the testing and inspection phases. The documentation is also demonstrably complete and consistent. We believe that the widespread availability of such documentation, and the availability of training in its use, would allow developers to work with confidence hence, more efficiently.

As a result of the work above, the method has evolved and been improved to cater for a larger range of problems. Simple issues like the handling of global variables and the communication protocol style of the component in question resulted in both the general method improving and specialisations been developed which refine this method for the target application. This documentation can be used as an everyday reference document for implementers. The use of output functions with supporting auxiliary functions leads to a structure that is easily readable and results in clear specification tables showing all pertinent information for the system in one document. During the research, many intermediate forms of documentation were attempted and analysed. The form presented above proved most suitable for the component under analysis. It should be noted that much of the time involved in applying the method was involved in gaining knowledge about the system. With a domain expert working on the design, some of the groupings and simplification that took large amounts of time would be second nature and much less time would be required.

ACKNOWLEDGEMENT

The authors wish to thank A. Leddy, J. McAuley, J. Prytherch and P. Martin of Ericsson, Ireland for their contributions to this work. This work was supported by the Science Foundation Ireland under SFI Grant 01/P1.2/C009.

REFERENCES

[1] R. Baber, D. Parnas, S. Vilkomir, P. Harrison, and T. O'Connor, Disciplined methods of software specifications: a case study. *Proc. of the Int'l Conf. on Information Technology Coding and Computing*, pp. 428–437, April 2005.

[2] W. Bartussek and D. Parnas, Using assertions about traces to write abstract specifications for software modules. *Proc. of 2nd Conf. of European cooperation in Informatics*, pp. 211–236, 1978.

[3] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, Abstract data-types as initial algebras and correctness of data representations. *Conf. on Computer Graphics, Pattern Recognition and Data Structure*, pp. 89–93, 1975.

[4] J. Guttag, Abstract data types and the development of data structures. *Proc. of the 1976 conference on data abstraction, definition and structure*, pp. 72–80, 1976.

[5] J. V. Guttag, E. Horowitz, and D. R. Musser, Some extensions to algebraic specifications. *ACM Conf. on Language Design for Reliable Software*, pp. 63–67, 1977.

[6] K. Heninger, Specifying software requirements for complex systems: new techniques and their application. *IEEE Trans. Software Eng.*, vol. SE-6, no. 1, pp. 2–13, Jan. 1980.

[7] D. Hoffman and Y. Wang, Executable prototypes of trace specifications. *Proc. of CIPS Edmonton*, pp. 202–220, 1987.

[8] M. Iglewski, J. Madey, and K. Stencel, On fundamentals of the Trace Assertion Method. *Techn. Rep. RR 94/09-6*, Universite du Quebec a Hull, Hull, Canada, 1994.

[9] R. Janicki, E. Sekerinski, Foundations of the Trace Assertion Method of module interface specification. *IEEE Trans. Software Eng.*, vol. 27, no. 7, pp. 577–598, July 2001.

[10] M. Kubica, TAM'97: the Trace Assertion Method of module interface specification. *Techn. Rep.*, Warsaw University, Institute of Informatics, Warsaw, Poland, 1997.

[11] B. H. Liskov and S. N. Zilles, Specification techniques for data abstractions. *IEEE Trans. Software Eng.*, vol. SE-1, no. 1, pp. 7–19, March 1975.

[12] D. Parnas, Inspection of safety critical software using function tables. *Proc. of IFIP World Congress*, pp. 270 - 277, 1994.

[13] K. Stencel, Refined simulation techniques for the Trace Assertion Method. *Techn. Rep. 95-17*, Warsaw University, Institute of Informatics, Warsaw, Poland, 1995.

[14] J. W. Thatcher, E. G. Wagner, and J. B. Wright, Data type specification: parameterization and the power of specification techniques. *ACM Trans. on Programming Languages and Systems*, pp. 711–732, 1982.

[15] Y. Wang, Specifying and simulating the externally observable behavior of modules. *Techn. Rep. 292*, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1994.

[16] 3GPP TS 23.002, 3rd generation Partnership Project, Technical Specification Group Services and Systems Aspects; Network architecture, 1999.

[17] 3GPP TS 25.331, 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Radio Resource Control (RRC); Protocol Specification, 1999.

APPENDIX
FORMAL SPECIFICATION

TABLE I
PGM(RUNREQ)

Variable	Type	Range
euCtxtRef	$< enum >$	%undefined%
propDelay	$< enum >$	%undefined%
rrcConnReqMsgPtr	$< enum >$	%undefined%
ueRef	$< enum >$	%undefined%

TABLE II
PGM(CAPACITYREQ)

Variable	Type	Range
reqId	$< int >$	%undefined%
ReqType	$< enum >$	%undefined%

TABLE III
UEUNREGISTRATION REJECT GROUP

Message Name	Flags
admissionRej	%none%
asynchforcedProcedureReset	%none%
asynch_forcedReleaseInd	%none%
cellParamsRC2CharCfm	rnhBarred
cellParamsRC2CharRej	%none%
rrcConnRequestInd	change_of_cell, EncodingInvalid, RRCLehFound
Timer	%none%

TABLE IV

FAILREJECT REJECT GROUP

Message Name	Flags
asynchforcedProcedur-eReset	%none%
asynch_forcedReleaseInd	%none%
registerServingUeC-txtCfm	rnhBarred
registerServingUeCtxtRej	%none%
rlibCapacityCfm	rnhBarred
rlibCapacityRej	%none%
rrcConnRequestInd	change_of_cell, EncodingInvalid, RRCLehFound
Timer	%none%

TABLE V

ELEMENTS OF SET CONFIRM

Message Name
adjacentIntraFreqCellsRsp
initialResourceCfm
admissionCfm
nbapRISetupRespInd
allocateDlChCodeNmCfm
registerServingUeCtxtCfm
capacityCfm
reserveAal2CepCfm
cellParamsCfm
rrcConnRequest
cellParamsRC2Cfm
rrcMsgDlCfm_1, rrcMsgDlCfm_2
ConnCfm
rrcMsgUlnd
fddIfhoSupp
spConfigCfm

TABLE VI

ELEMENTS OF SET REJECT

Message Name
celloDisconnect
failReject
DecreaseLoad
failSpRelease
DlChCodeRelease
failSpResources
failRadioLinkRlease
ueUnRegistration

TABLE VII

Reject AUXILIARY FUNCTION

i	rej(i)
1	failSpRelease
2	failRadioLinkRelease
3	celloDisconnect
4	releaseSpResources
5	DlChCodeRelease
6	DecreaseLoad
7	ueUnRegistration
8	failReject

TABLE VIII

prev_conf AUXILIARY FUNCTION

name	prev_conf(name)
rej(1)	conf(1), conf(2)
rej(2)	conf(3), conf(4)
rej(3)	conf(5), conf(6), conf(7)
rej(4)	conf(8), conf(9)
rej(5)	conf(10)
rej(6)	conf(10)
rej(7)	conf(11)
rej(8)	conf(12), conf(13), conf(14), conf(15), conf(16), conf(17)

TABLE IX

SET OF OUTPUT MESSAGES

Message Name
allocateDlChCodeNmReq
measControlReceivedInd_3
admissionReq
measControlReceivedInd_4
admissionDecreaseLoadInd
nbapRIDelRequestInd
adjacentIntraFreqCellsReq
nbapRIFreeCrnCldInd
CapacityReq
nbapRISetupRequestInd
cellParamsReq
netConnectReq
cellParamsRC2Req
reserveAal2CepReq
ConnRejectInd rrcMsgDlInd
deallocateDlChCodeNmInd
releaseResourceReq
deallocateDlChCpmInd
RRCConnEstInd
initialResourceReq
spConfigReq
messageRejectInd
ueUnregisterServingUeCtx-tInd
measControlReceivedInd_1
Error
measControlReceivedInd_2

TABLE X

out AUXILIARY FUNCTION

name	out(name)
rrcConnRequest	CapacityReq
capacityCfm	ueRegistrerServingCtxtReq
registerServingUeC-txtCfm	cellParamsRC2Req
cellParamsRC2Cfm	admissionReq
admissionCfm	cellParamsReq
cellParamsCfm	adjacentIntraFreqCellsReq
adjacentIntraFreqCellsRsp	allocateDlChCodeNmReq
allocateDlChCodeNmCfm	initialResourceReq
initialResourceCfm	reserveAal2CepReq
reserveAal2CepCfm	nbapRISetupRequestInd
nbapRISetupRespInd	netConnectReq
ConnCfm	spConfigReq
spConfigCfm	RRCConnEstInd
rrcMsgUlnd	measControlReceivedInd_1
rrcMsgDlCfm_1	measControlReceivedInd_2
fddIfhoSupp	rrcMsgDlInd
rrcMsgDlCfm_2	measControlReceivedInd_3. rrcMsgDlInd. measControlReceivedInd_4
failSpRelease	spConfigReq
failRadioLinkRelease	nbapRIFreeCrnCldInd. nbapRIDelRequestInd
celloDisconnect	messageRejectInd
DlChCodeRelease	deallocateDlChCodeNmInd. deallocateDlChCpmInd
releaseSpResources	releaseResourceReq
DecreaseLoad	admissionDecreaseLoadInd
ueUnRegistration	ueUnregisterServingUeCtxtInd
failReject	ConnRejectInd

TABLE XI

conf AUXILIARY FUNCTION

i	conf(i)
1	rrcConnRequest
2	capacityCfm
3	registerServingUeCtxtCfm
4	cellParamsRC2Cfm
5	admissionCfm
6	cellParamsCfm
7	adjacentIntraFreqCellsRsp
8	allocateDlChCodeNmCfm
9	initialResourceCfm
10	reserveAal2CepCfm
11	nbapRISetupRespInd
12	ConnCfm
13	spConfigCfm
14	rrcMsgUlnd
15	rrcMsgDlCfm_1
16	fddIfhoSupp
17	rrcMsgDlCfm_2

TABLE XII

num AUXILIARY FUNCTION

name	num(name)
rrcConnRequest	1
capacityCfm	2
registerServingUeCtxtCfm	3
cellParamsRC2Cfm	4
admissionCfm	5
cellParamsCfm	6
adjacentIntraFreqCellsRsp	7
allocateDlChCodeNmCfm	8
initialResourceCfm	9
reserveAal2CepCfm	10
nbapRISetupRespInd	11
ConnCfm	12
spConfigCfm	13
rrcMsgUlnd	14
rrcMsgDlCfm_1	15
fddIfhoSupp	16
rrcMsgDlCfm_2	17
failSpRelease	1
failRadioLinkRelease	2
celloDisconnect	3
releaseSpResources	4
DlChCodeRelease	5
DecreaseLoad	6
ueUnRegistration	7
failReject	8

TABLE XIII
$O(T)$ OUTPUT FUNCTION

T				O(T)
$last(T) \in CONFIRM \wedge$	$last(T) = conf(1) \wedge$	$rest(T) = \_$		$out(last(T))$
		$rest(T) \neq \_$		$Error$
	$last(T) \neq conf(1) \wedge$	$last(rest(T)) = conf(num(last(T)) - 1)$		$out(last(T))$
		$last(rest(T)) \neq conf(num(last(T)) - 1)$		$Error$
$last(T) \in REJECT \wedge$	$rnhBarred \wedge$	$last(T) = rej(2) \wedge last(rest(T)) = rej(4))$		$out(last(T)$
		$\neg last(T) = rej(2) \wedge last(rest(T)) = rej(4))$		$Error$
	$\neg rnhBarred \wedge$	$last(T) = rej(8) \wedge$	$last(rest(T)) \in prev\_conf(rej(8))$	$out(last(T))$
			$\neg last(rest(T)) \in prev\_conf(rej(8))$	$Error$
		$\neg last(T) = rej(8) \wedge$	$last(rest(T)) = rej(num(last(T))+1) \vee last(rest(T)) \in prev\_conf(rej(last(T)))$	$out(last(T))$
			$\neg last(rest(T)) = rej(num(last(T))+1) \vee last(rest(T)) \in prev\_conf(rej(last(T)))$	$out(last(T))$
$\neg(last(T) \in CONFIRM \vee last(T) \in REJECT)$				$Error$