

OSEK/VDX

Conformance Testing Methodology

Version 2.0

April 16th, 1999

This document is an official release and replaces all previously distributed documents. The OSEK group retains the right to make changes to this document without notice and does not accept any liability for errors. All rights reserved. No part of this document may be reproduced, in any form or by any means, without permission in writing from the OSEK/VDX steering committee.

What is OSEK/VDX?

OSEK/VDX is a joint project of the automotive industry. It aims at an industry standard for an open-ended architecture for distributed control units in vehicles.

A real-time operating system, software interfaces and functions for communication and network management tasks are thus jointly specified.

The term OSEK means "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" (Open systems and the corresponding interfaces for automotive electronics).

The term VDX means „Vehicle Distributed eXecutive“. The functionality of OSEK operating system was harmonized with VDX. For simplicity OSEK will be used instead of OSEK/VDX in the document.

OSEK partners:

Adam Opel AG, BMW AG, Daimler-Benz AG, IIT University of Karlsruhe, Mercedes-Benz AG, Robert Bosch GmbH, Siemens AG, Volkswagen AG.

GIE.RE. PSA-Renault (Groupement d'intérêt Economique de Recherches et d'Etudes PSA-Renault).

Motivation:

- High, recurring expenses in the development and variant management of non-application related aspects of control unit software.
- Incompatibility of control units made by different manufacturers due to different interfaces and protocols.

Goal:

Support of the portability and reusability of the application software by:

- Specification of interfaces which are abstract and as application-independent as possible, in the following areas: real-time operating system, communication and network management.
- Specification of a user interface independent of hardware and network.
- Efficient design of architecture: The functionality shall be configurable and scaleable, to enable optimal adjustment of the architecture to the application in question.
- Verification of functionality and implementation of prototypes in selected pilot projects.

Advantages:

- Clear savings in costs and development time.
- Enhanced quality of the control units software of various companies.
- Standardized interfacing features for control units with different architectural designs.
- Sequenced utilization of the intelligence (existing resources) distributed in the vehicle, to enhance the performance of the overall system without requiring additional hardware.
- Provides absolute independence with regards to individual implementation, as the specification does not prescribe implementation aspects.

OSEK conformance testing

OSEK conformance testing aims at checking conformance of products to OSEK specifications. Test suites are thus specified for implementations of OSEK operating system, communication and network management.

Work around OSEK conformance testing is supported by the MODISTARC project sponsored by the Commission of European Communities. The term MODISTARC means "Methods and tools for the validation of OSEK/VDX based DISTRIBUTED ARChitectures".

This document has been drafted by MODISTARC members:

Bernd Büchs	Adam Opel AG
Wolfgang Kremer	BMW AG
Stefan Schmerler	FZI
Franz Adis	FZI
Benoit Caillaud	INRIA
Yves Sorel	INRIA
Dirk John	IIT, Karlsruhe University
Robert France	Motorola
Barbara Ziker	Motorola
Jean-Emmanuel Hanne	Peugeot Citroën S.A.
Samuel Boutin	Renault S.A.
Eric Brodin	Sagem SA
Gerhard Goeser	Siemens Automotive SA
Patrick Palmieri	Siemens Automotive SA
Didier Stunault	Thomson-CSF Detexis

Remark by the authors

This document is inspired by the paper published at the first OSEK Workshop [1], which sets the development framework for an OSEK conformance test process. This document also takes account of the work conducted since by the OSEK conformance testing group.

Table of Contents

1.	Scope of Conformance Testing	5
1.1.	Motivations	5
1.2.	Definitions and intentions	5
1.3.	Conformance testing concerns	7
1.4.	Rules of test suites definition	7
1.5.	Other limitations	8
1.6.	Work programme	9
2.	Definition of Test Suites	11
2.1.	Definition process	11
2.2.	OS test suites	14
2.2.1.	Test configurations	14
2.2.2.	Services and variants	14
2.2.3.	Definition method for test suites	16
2.3.	COM and NM test suites	17
2.3.1.	Test configurations	17
2.3.2.	Services and variants	18
2.3.3.	Event management	22
2.3.4.	Definition method for test suites	29
2.4.	Strategy for OSEK modules integration and testing	30
2.4.1.	OSEK integration and conformance testing	30
2.4.2.	Conformance testing of a complete OSEK implementation	30
3.	Test architecture for COM and NM	32
3.1.	Description of the test architecture	32
3.2.	Rules of UT, LT and TMP specification	33
3.3.	TMP mechanisms	35
3.4.	Example of UT specification	37
4.	Methods of test suite generation	39
4.1.	Generation of OS test suites	39
4.1.1.	Generation method and supporting tool	39
4.1.2.	Test suite example	40
4.1.3.	Test generation tool	41
4.2.	Generation of COM and NM test suites	44
4.2.1.	Generation method	44
4.2.2.	Impact of test architecture	46
4.2.3.	Test suite example	47
4.2.4.	Test generation tools	49
5.	TTCN overview	51
5.1.	Declarations	51
5.2.	Constraints	52
5.3.	Dynamic behaviour	53
6.	Abbreviations	55
7.	References	56

1. Scope of Conformance Testing

1.1. Motivations

The OSEK project has led to the specification of three standards defining an Operating System and Communication and Network Management services and protocols ([2], [3], [4]). Its purpose is to define basic modules on which the distributed applications of future automotive systems will be based. Standardisation of these modules will significantly reduce the systems' development costs and schedules, while creating a fully open market for interchangeable OSEK components or hardware/software systems integrating these components.

These goals cannot be achieved until a conformance procedure exists that allows the products claiming the OSEK/VDX label to be qualified. Short of such a procedure, there is a high risk that many incompatible implementations will arise, forcing either the ECU suppliers to procure from a single software source, or the car manufacturers to procure from a single ECU supplier. The goal of the conformance procedure is to prevent potential conformance conflicts or to act as an arbitrator and settle disputes, where appropriate.

1.2. Definitions and intentions

The objectives of the OSEK conformance process are to determine whether an OSEK implementation complies or not with the OSEK specification. The purpose is therefore first to establish, based on the specification, a list of conformance rules applicable to implementations, which should ideally guarantee that the implementations will react in accordance with the specification to a given series of events, under a given set of circumstances. These rules are standardised by **test suites** that the implementations must pass to achieve OSEK qualification.

From the conformance testing perspective, the implementation to be tested is seen as a black box whose sole external interfaces are accessible. Therefore, conformance tests cannot check the complete OSEK functionality of implementations.

The external interfaces are the APIs and OPDUs defined in the specification. OSEK module access points are presented in Figure 1 below. COM and NM make use of both APIs and OPDUs while OS only makes use of APIs.

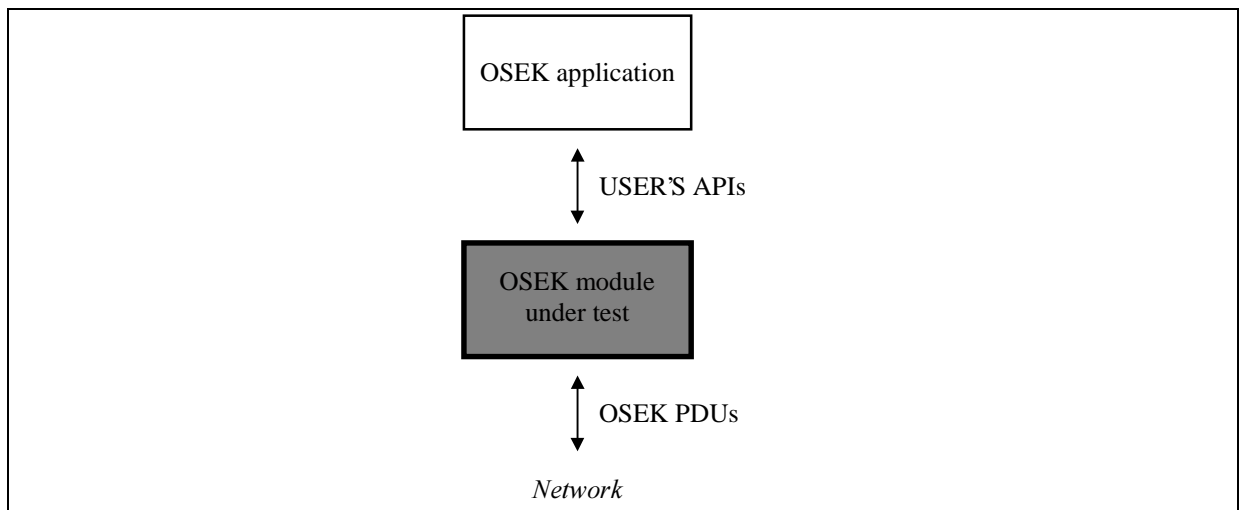


Figure 1 OSEK module interfaces

A test suite then defines which actions and verifications a tester must conduct on the APIs and OPDUs to carry out the conformance tests. This definition's abstraction level is the same as that of the OSEK specification. It does not presuppose nor recommend any kind of test platform implementation.

In addition to strict observance of the OSEK specification, the test suites shall reflect the OSEK specification intentions and objectives. The only point of interest here is what the OSEK customer is expecting from the conformance tests, whether he is ECU or system integrator. In this respect, the intentions of OSEK are expressed in terms of portability of applications and interoperability of ECUs. These two notions are detailed below.

- Portability is the ability to move an application from one OSEK environment to another OSEK environment with no need to make changes to this application. The "OSEK environment" depends on three factors which are the micro-controller, the ECU and the OSEK implementation itself. The OSEK implementation in turn comprises three elements: the OS, COM and NM. Different degrees of portability can therefore be defined, with the minimum level corresponding to changing a single element, while the maximum level corresponds to changing all elements. The test suites will only address portability of applications, not OSEK inter-module portability. This implies that APIs and services for user applications are correctly implemented.
- Interoperability is the ability for two applications or more installed in different ECUs to exchange data, which assumes that the exchange protocols defined in the OSEK specification are being complied with. Interoperability only concerns COM and NM.

The selected tests shall be those and only those which are required to achieve both these objectives.

1.3. Conformance testing concerns

The conformance process shall run throughout the life cycle of an OSEK system. Each stage can give rise to conflicts or to errors, and suitable test procedures are therefore necessary. The development cycle can be broken down into three stages:

- development of OSEK software,
- integration of the OSEK software to the target equipment,
- interconnection of equipment to validate the system.

In the first phase, the purpose is to validate source code. Conformance tests can help to verify whether software being developed is compliant with the specification. But they are not capable of performing a full validation.

In the second phase, the purpose is to test one or more OSEK components installed in the target equipment and verify OSEK compliance of the object code. It is the main goals of conformance testing.

In the third phase, i.e. system validation, the user's applications are in operational use and running the tests would have a disruptive effect. Any interoperability problems which could be detected shall be resolved by other means, such as using protocol analysers. Going back to phase 2 can however be required to analyse the behaviour of a suspect ECU.

1.4. Rules of test suites definition

The test suites shall strictly reproduce what is written in the specification, and only that, in keeping with the above-defined objectives. The test suites are independent from the design options of the implementation to be tested. They shall add nothing which could reduce without justification the scope of compliant implementations.

Conversely, any detail of the specification which can be considered as specific to an implementation or a category of implementations shall be dismissed. As the saying goes:

"All what behaves like OSEK is OSEK"

The test suites are independent from the tested implementation environment, in particular as regards the type of processor for the OS or the bus protocol (CAN, VAN, J1850...) for COM. The specification of the network data exchanged during a test is provided in OSEK PDU format.

The test suites shall cover all the specification variants, such as OS or COM classes or NM optional services. The compatible variants of each suite are clearly identified to allow the tests applicable to each implementation to be selected.

The specification elements to be covered by tests fall into three classes.

- APIs.
APIs define procedural calls available to application. The purpose is to check that the interface has been implemented correctly.

- **Services.**
 Services define the function performed by the API, such as a task activation. The specification defines two categories of services, i.e.: generation services and run-time services. Only run-time services are addressed by conformance tests. Generation services supply constructors intended to automate OSEK application generation. They are usually implemented by specific off-line tools of the supplier's production line. And validation of such tools is out of the scope of OSEK conformance testing.
 Testing conformance of a service amounts to checking that the API produces the expected information and return status. It allows to check behaviour rules stated in the specification inasmuch as those behaviours are observable through API calls. For example, OS scheduling rules can be verified by activating several tasks and calling *GetTaskState* to check the states of the different tasks. Behaviour verification often implies execution of a combination of services.
 The specification defines standard status and extended status. Both of them will be checked. Although extended status are generally used for application debugging and not installed in operational software, they will be tested so that applications can be validated with compliant OSEK software from the beginning.
 Conformance tests will check all status, inclusive of error reports. They will also check that the service can be called up under all the conditions defined by the specification, at task level or at interrupt level, for example.
- **Protocols.**
 Protocols describe network behaviours. They only concern COM and NM. They are specified by state/event automata described by graphs or SDL diagrams. Protocol observability is allowed through the OPDUs transmitted and received by the implementation. Conformance tests of protocols shall cover all the transitions of the automata inasmuch as they are observable, whether they express nominal behaviours or error cases.

Two other types of tests can also be meaningful to an OSEK implementation user. However they are out of the scope of conformance testing and they will not be dealt with in the current project:

- Tests of capacity parameters, for example the number of tasks supported by the OS or the number of network connections supported by COM,
- Tests of performance parameters, for example task switching times or message transit times in the communication layers.

1.5. Other limitations

It is not the purpose of the conformance tests to validate an OSEK software. The intended objectives are not to detect design or coding errors but to check that the implementation is consistent with the specification. Implementators shall validate their software using conventional resources before presenting it for conformance tests.

In the same way, the conformance tests definition does not aim at checking that the OSEK specification is consistent, bug-free, and has been designed to cover all possible situations. In this task, the specification is considered as a reliable basis. However, any discrepancies which could be discovered incidentally will be forwarded to the specification groups to be impacted to the OSEK standards.

Conformance tests have no claim to completeness. It will always be impossible to check all possible combinations of events and situations predicted by the specification. The goal is to achieve through tests as wide a coverage of the various functions as possible. Portability or interoperability troubles may still arise owing to special scheduling of events or combination of parameter values not covered by the tests.

The conformance tests are "black box" tests. Only events corresponding to the specification interfaces (APIs, OPDUs) can be checked. Only the data associated with these events are observable. For example, checking the state of a protocol automaton requires the existence of an interface primitive providing this state. Otherwise, the test can only assume that the state went to the expected value based on the events it could observe, such as the OPDUs transmitted by the implementation.

1.6. Work programme

This document forms part of the MODISTARC programme which has been designed to cover the whole life-cycle of the conformance testing activity. Actually, as every specification the test suites definition must be consolidated through implementation and validation of the implementation. Therefore, the role of MODISTARC is firstly to define the OSEK test suites and subsequently to realise the first implementation of the specified suites and validate the implementation against OSEK prototypes. More precisely, the MODISTARC programme consists of the four following phases:

- Conformance testing methodology. The goals are to define the relevant methods for checking OSEK software conformance. This includes methods that will be employed throughout the life cycle for definition, implementation and validation of the test suites. The issues are represented by the present document.
- Test suites definition. The goals are to specify the OSEK conformance test suites according to the rules established in the present document. The issues are three documents defining the test suites for OS, COM and NM. They are established from the associated specification document and they will become the official OSEK conformance standards after the validation.
- Implementation work. This phase includes all implementation work required to validate the test suites specification. It comprises the development of conformance tools implementing the test suites and the development of OSEK prototypes to be used in the validation phase.

As test suites are defined in an implementation-independent style, some implementation choices have been decided for the test tools. For instance, the hardware environment is PC and network protocols are checked via a CAN network.

The OSEK prototypes include a PC implementation and three ECUs. Each one will implement different OSEK specification variants to allow different configurations of the suites to be assessed.

- Conformance test campaign. This phase will start with conformance assessment of the four OSEK prototypes using the PC tools. Then, the various prototypes will be interconnected on a test platform and a distributed application will be installed and executed to demonstrate the fulfilment of the two main objectives stated before, that is portability and interoperability of OSEK software once conformance has been established.

The intended test application is a virtual application using OSEK services and protocols, and built as a symmetrical application. Therefore, portability will be evaluated by checking that the OSEK API's behaviour is the same in the different targets. Interoperability will be demonstrated through the data exchanges between the remote parts of the symmetrical application.

2. Definition of Test Suites

2.1. Definition process

The conformance tests shall check that the APIs, services and protocols of the specification are implemented correctly.

As concerns APIs, the tests shall check that the implementation complies with the syntax defined in the specification. This check is carried out automatically by a compilation tool when linking the implemented API with an OSEK compliant software that makes use of the APIs. Provided there is a conformance software for OSEK services which respects the OSEK API, linking that software with OSEK implementation will automatically check syntax compatibility between specified OSEK API and implemented OSEK API. Therefore, it is not necessary to define specific test suites to verify API conformance.

As concerns services and protocols, the various parameters which will impact definition of the test suites have to be first identified:

- the list of variants,
- the list of APIs accessible as a function of the variants,
- the list of specification parameters which the user shall define before executing the tests (such as network addresses)
- the list of constraint parameters whose value will influence test selection for a given implementation (such as maximum number of tasks)

Subsequently, conformance test definition is a two-stage process:

- definition of the test purposes,
- definition of the test cases

Definition of the test purposes results from analysis of the specification. Definition determines what can and what must be tested. The method used consists in reading the specification and extracting checkable assertions. The assertions are established from the specification's text, tables or figures and from the SDL diagrams for protocols. Then, the complete assertions are analysed to remove redundancies. The result is a table containing for each assertion:

- a sequence number used as a reference for test suite traceability,
- the description of the test purpose comprising one or two sentences extracted from the specification,
- the variants of the specification to which the purpose applies,
- reference to the specification paragraph allowing traceability to be provided against the specification.

The complete test purposes of an OSEK module makes up the **test plan**.

Definition of the test cases consists in specifying the sequence of interactions between the tester and the implementation which will allow one or more test purposes to be verified. The method is derived from ISO 9646 [5] which was drafted to check conformance of the ISO communication protocols. The general principles apply to OSEK conformance tests, including those of the OS.

The test cases are organised per a hierarchical classification. Each sequence is identified by a path in this classification. For example, OS/function/test number.

To each test case is associated a list of requirements defining:

- the utilisation conditions: task level, ISR, ...
- the applicable variants: OS, COM class, ...
- the capacity requirements: number of required tasks,...

The complete test cases for an OSEK module make up the **test suite**.

The test suite and all needed information to implement and execute the tests make up the **test procedure**. Extra information concerns elements of test architecture, parameterization of the test suite, selection of applicable test cases...

A test case comprises three parts:

- a preamble which places the implementation in the test execution conditions,
- the sequence of interactions corresponding to the test purposes,
- a postamble which returns the implementation to initial condition.

Specification of test cases requires the definition of one or more initial states from which the preambles and postambles will be established. The most used initial states are those obtained after OS, COM or NM initialisation respectively.

The purpose of this method is to increase the flexibility of use of conformance tools. The tests with the same initial test are independent and it is possible to access one of them without repeating the previous tests. Further, as the initial and end states are the same, a test can be replayed as many times as necessary to analyse a conformance fault. Conversely, the preambles and postambles extend the duration of each test when a complete test is run. There must be a sufficient number of initial states for the test suites length to be acceptable.

The interactions sequence which describes a test case is made of transmissions to the implementation and receptions from the implementation. In the most simple case, transmission corresponds to activation of an API and reception corresponds to the status code and parameters returned by the API. In the case of protocols, these can be transmissions and receptions of PDUs.

A reception interaction is followed by a verification of the received information which allows checking whether the test ran correctly.

Execution of a test case results in a **verdict** which may have one of the following values:

- **PASS**: the test purpose(s) have been achieved,
- **FAIL**: a conformance fault has been detected,
- **INCONCLUSIVE**: the test purposes have not been achieved but no compliance fault has been detected. This only occurs in case of COM and NM, when message(s) expected from the implementation have not been received by the tester. The problem can result from network failure or other environmental events and the tester cannot certify a compliance fault.

INCONCLUSIVE verdict may also be set when the test cannot be executed for various reasons, e.g. non implemented variant of the specification.

Execution of a test suite produces two reports:

- the **conformance tests report** providing the list of the test cases that have effectively been processed and the verdict produced by every one,

- the **conformance report** giving the global status of conformance. The implementation will be declared compliant if all tests produce a PASS verdict. Tests giving INCONCLUSIVE verdicts for which environment failures are suspected must be passed again until they return a PASS verdict.

2.2. OS test suites

2.2.1. Test configurations

As the conformance testing is agreed to be a black box testing, the only interface of the OS module is the OS API defined in the OS specification. This can be seen in the left part of Figure 2. The right part shows how the tested OS is embedded into the conformance tester to run the test suites.

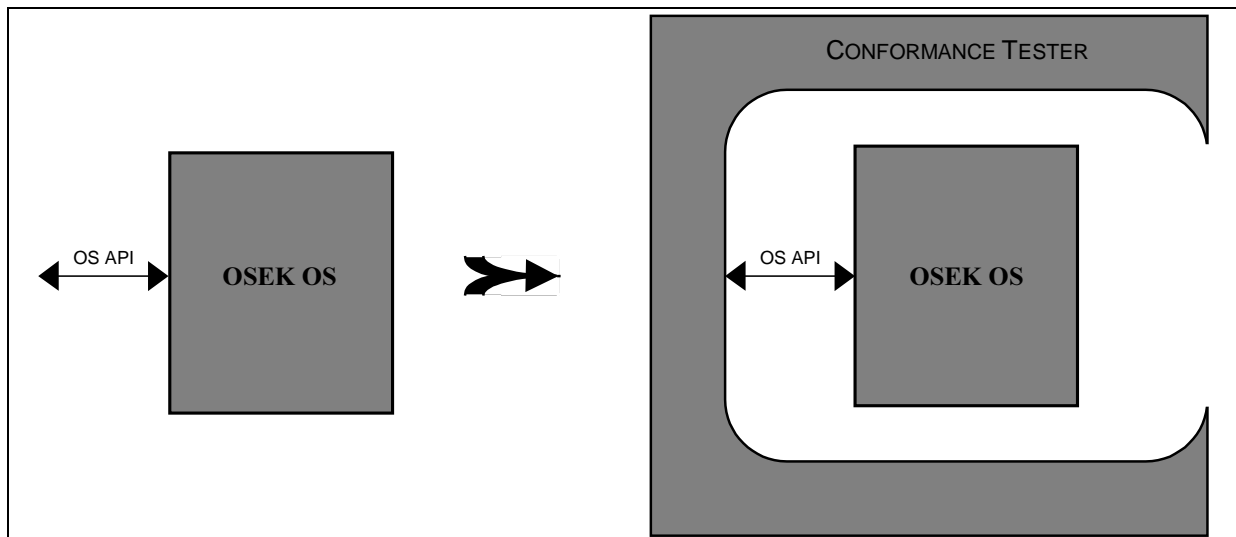


Figure 2 Conformance testing configuration for OS

The conformance tester consists of an OSEK application which makes dedicated calls to the API and compares the returned values to the values prescribed in the specification. To compile and generate such an application it is necessary to configure the system (tester and OS). Configuration implies amongst others determination of

- conformance class needed
- task attributes (priority, basic/extended, preemptive/non-preemptive, ...)
- resources, events, alarms needed
- ...

The OSEK OS specification [2] recommends the usage of OIL (OSEK Implementation Language) for system generation.

2.2.2. Services and variants

The OS API consists of the services presented in Table 1. Constructional elements for object declaration and generation are not considered since they are not used at run time.

OS-API Services	Service call
Task management services <ul style="list-style-type: none"> – Transfer task into <i>ready</i> state – Terminate calling task – Terminate calling task and activate succeeding task – Call scheduler – Get currently active task – Get state of a task 	<i>ActivateTask</i> <i>TerminateTask</i> <i>ChainTask</i> <i>Schedule</i> <i>GetTaskId</i> <i>GetTaskState</i>
Interrupt handling services <ul style="list-style-type: none"> – Enter interrupt service routine (ISR) – Leave interrupt service routine (ISR) – Enable interrupts – Disable interrupts – Query interrupt status 	<i>EnterISR</i> <i>LeaveISR</i> <i>EnableInterrupt</i> <i>DisableInterrupt</i> <i>GetInterruptDescriptor</i>
Resource management services <ul style="list-style-type: none"> – Get resource and enter critical section – Release resource and leave critical section 	<i>GetResource</i> <i>ReleaseResource</i>
Event control services <ul style="list-style-type: none"> – Set event of extended task – Clear Event – Get event mask of a task – Wait for setting of an event 	<i>SetEvent</i> <i>ClearEvent</i> <i>GetEvent</i> <i>WaitEvent</i>
Alarms services <ul style="list-style-type: none"> – Read alarm base characteristics – Occupy and set relative alarm – Occupy and set absolute alarm – Cancel alarm – Get alarm value 	<i>GetAlarmBase</i> <i>SetRelAlarm</i> <i>SetAbsAlarm</i> <i>CancelAlarm</i> <i>GetAlarm</i>
Operating system execution control services <ul style="list-style-type: none"> – Get current application mode – Start operating system – Shut down operating system 	<i>GetActiveApplicationMode</i> <i>StartOS</i> <i>ShutdownOS</i>
Hook routines <ul style="list-style-type: none"> – Called if OS service returns an error – Called at task switch before entering context of new task – Called at task switch after leaving context of old task – Called after start-up – Called before shutdown 	<i>ErrorHook</i> <i>PreTaskHook</i> <i>PostTaskHook</i> <i>StartupHook</i> <i>ShutdownHook</i>

Table 1 OS-API services

All services except the event control services shall be implemented in all conformance classes. The event control services shall be implemented in conformance classes ECC1 and ECC2 only.

There are many variants for the OS services because of different conformance classes (Table 2) and different scheduling policies (Table 3) that are defined in the specification. Each OS service may have a different behaviour for each variant. The assessment of one of these variants is done statically before generation of the OSEK application. The conformance test must cover all possible variants.

OS conformance classes	Description
BCC1	only basic tasks, limited to one request per task and one task per priority, while all tasks have different priorities
BCC2	like BCC1, plus more than one task per priority possible and multiple requesting of tasks allowed
ECC1	like BCC1, plus extended tasks
ECC2	like BCC2, plus extended tasks without multiple requesting admissible

Table 2 OS conformance classes

OS scheduling policies	Description
non-preemptive	Task switches are only performed via one of a selection of explicitly defined system services (explicit points of rescheduling)
full-preemptive	Tasks may be rescheduled at any instruction by the occurrence of trigger conditions pre-set by the operating system
mixed-preemptive	Full-preemptive and non-preemptive scheduling principles are to be used for execution of different tasks on the same system

Table 3 OS scheduling policies

2.2.3. Definition method for test suites

The following means will be used to describe and specify the test suites:

- State-/Activity-Charts for a high-level description of the sequence of the test suite
- C-Code for generation of the executable of the test suite
- OIL [8] for configuration of the test suite
- TTCN [5] as recommended by ISO 9646 for test suite description

For details about generation of test suites see Chapter 4.1.

2.3. COM and NM test suites

2.3.1. Test configurations

According to the black box testing principle, a test suite is made of a sequence of interactions between the tester and the OSEK implementation. It is therefore necessary to firstly identify the OSEK interfaces that will be made available to a conformance tester for running the test suites.

Configuration for COM

The potential interfaces of a COM module are those defined in the COM specification [3] and presented in the left part of Figure 3:

- the COM API intended to OSEK applications,
- the OS API, as for instance Alarm services to manage the protocol timers,
- the COM-IL API intended to OSEK/NM modules,
- the Network API at the interface between Interaction Layer and Network Layer,
- the DLL/COM API at the interface between Network Layer and Data Link Layer or between Interaction Layer and Data Link Layer in implementations without the optional network layer,
- the DLL/NM API offering special data link services to NM modules such as the Window management functionality,
- the OSEK PDUs sent onto or received from the network by the COM module.

As regards conformance testing, only interfaces that comply with the objectives of application portability and network interoperability must be kept. This excludes internal interfaces and interfaces with other OSEK modules (OS, NM). Relevant interfaces for conformance testing are presented in the right part of Figure 3, namely:

- the COM API, mandatory to enable application portability,
- the OSEK PDUs, mandatory to enable network interoperability.

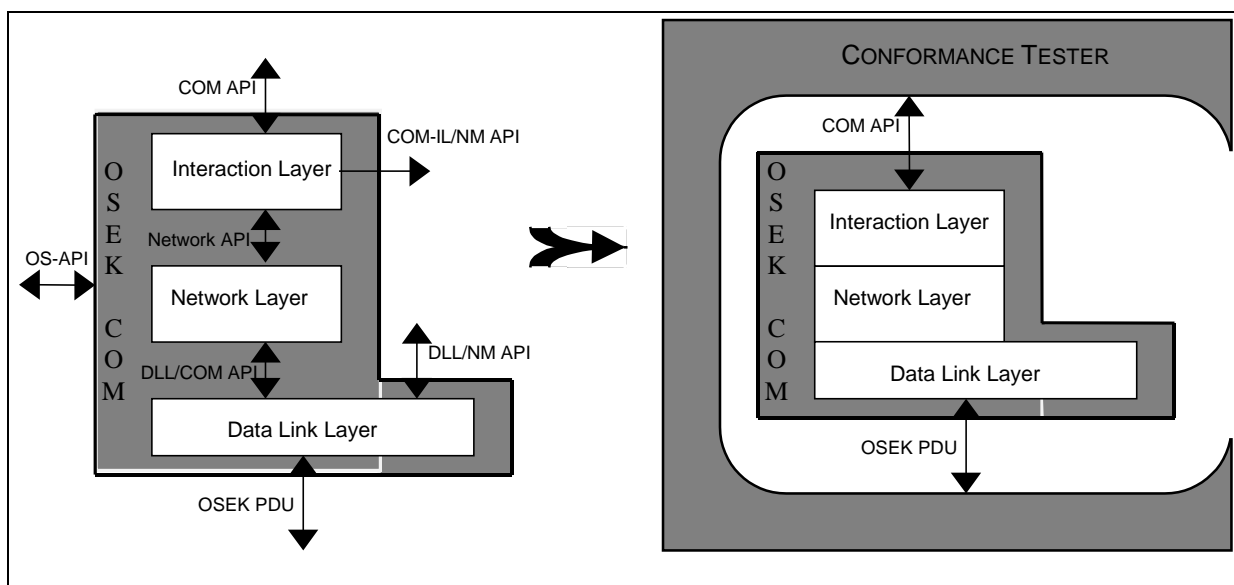


Figure 3 Conformance testing configuration for COM

Configuration for NM

The interfaces of an NM module as defined by the NM specification are those shown in left part of Figure 4:

- the NM API intended to OSEK applications and offering data exchange capabilities to application via the NM infrastructure,
- the OS API,
- the COM-IL/NM API (indirect NM),
- the COM-DLL/NM API.

However, it is more convenient to test NM protocols via a real network. Implementing the protocol tester in an external equipment makes it independent of the NM module environment and it allows to keep the same practical test methods as for COM protocols. Therefore, the lower NM interface should be the OSEK PDUs exchanged by NM rather than the DLL/NM API. Consequently, the necessary Data Link services are to be added to the NM module in the test configuration as shown in the middle part of Figure 4. In case of indirect NM, the test environment must also provide the COM-IL/NM services. They can be either simulated by the conformance tester or generated by an OSEK/COM module incorporated in the tester.

Finally, the same justification as in COM can be put forward to exclude the OS interface from NM conformance testing objectives. As a consequence, the NM conformance tester will only access the NM API and the OSEK PDUs as presented in the right part of Figure 4 and additionally the COM-IL/NM API in case of indirect NM.

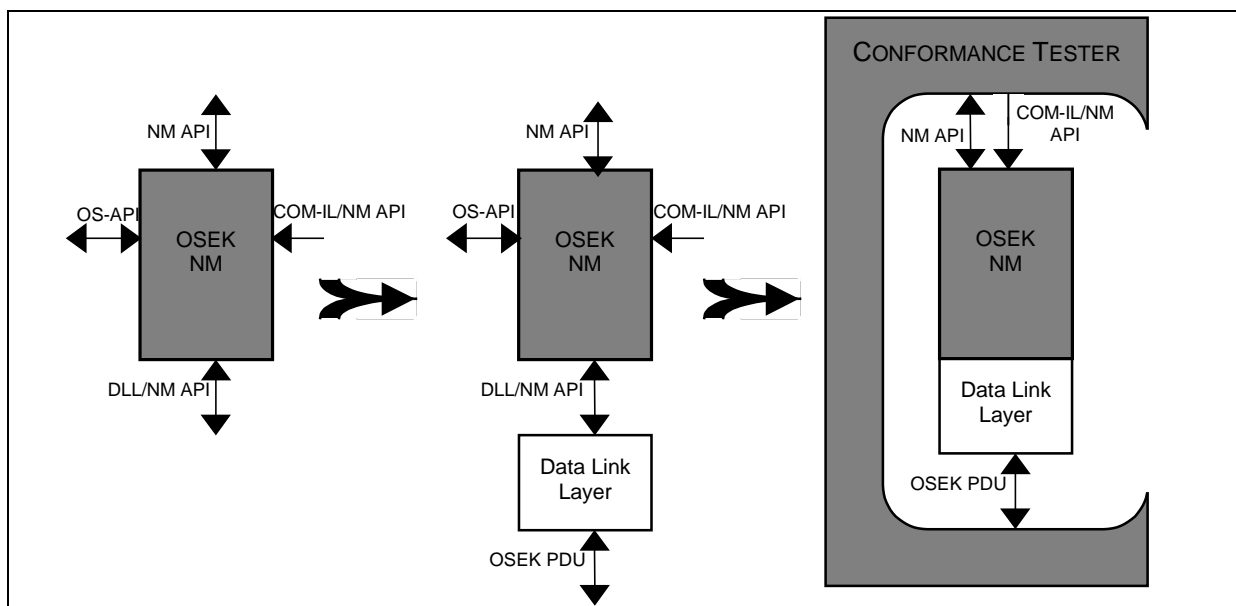


Figure 4 Conformance testing configuration for NM

2.3.2. Services and variants

COM-API services

OSEK/COM implementations shall be compliant with one in the four conformance classes defined in the specification. Conformance classes are upward compatible. Therefore, conformance tests shall be organised in order to provide a series of test cases adapted to each

conformance class. Due to upward compatibility, implementations must pass test cases of their conformance class and of the lower classes.

Conformance class of implementation	Tests to be executed			
	CCC0	CCC1	CCC2	CCC3
CCC0	X			
CCC1	X	X		
CCC2	X	X	X	
CCC3	X	X	X	X

Table 4 Tests to be executed according to conformance class

The COM API consists of the services presented below. The lowest conformance class at which the services must be implemented is also specified.

COM-API Services	Service Call	Conformance class
COM start-up		
- Start of COM module	<i>StartCOM</i>	CCC0
Data transfer		
- Update and send static message object	<i>SendMessage</i>	CCC0
- Receive static message object	<i>ReceiveMessage</i>	CCC0
- Update and send dynamic message object	<i>SendMessageTo</i>	CCC2
- Receive dynamic message object	<i>ReceiveMessageFrom</i>	CCC2
- Set message resource as busy	<i>GetMessageResource</i>	CCC0
- Set off message resource from busy	<i>ReleaseMessageResource</i>	CCC0
- Get the current status of the message	<i>GetMessageStatus</i>	CCC0

Table 5 COM-API services

Services related to system object declaration and generation will not be checked in conformance tests since they are not used at run time. However, an implicit verification will be performed through the test application implementation. Indeed, both the test application and the OSEK/COM will make use of system objects that will be compiled to generate executable code and will be run during the test campaign.

Moreover, the COM services include a lot of capabilities. Messages can be either unqueued (not buffered) or queued (buffered), withcopy (local copy in application task) or without copy (direct access to message object). Transmission triggering can be direct (at user's request) periodic or mixed (direct + periodic). Reception or end of transmission can be signalled through task activation or event setting. An alarm may be sent to warn of non reception of a periodic message or to warn of non transmission.

Notifications of (non) message transmission or of (non) message reception can be seen as pieces of information sent by the COM module to applications. They can therefore be assimilated to indication services supplied by the COM module. These indication services, as presented in the table below, are not new services but a means to formalise notification

actions that should be performed by the implementation. They can be implemented with OS task or event activation mechanisms, or they can solely be notified by setting the message status. The implementation option is selected on a per message basis.

COM indication service	Service name	Conformance class
- Indication of message transmission	<i>Tx_ind</i>	CCC1
- Indication of no message transmission	<i>NoTx_ind</i>	CCC1
- Indication of message reception	<i>Rx_ind</i>	CCC1
- Indication of no message reception	<i>NoRx_ind</i>	CCC1

Table 6 COM indication services

Messages can be transferred between local tasks or through the network. In case of network-wide transmission, the *SendMessage/ReceiveMessage* calls imply the utilisation of one of the network protocols, UUDT (unsegmented) or USDT (segmented). *SendMessageTo/ReceiveMessageFrom* are only used for network transmissions with USDT. Those services offer the possibility either to send the same message object to different receivers or to receive a message from different senders.

All message characteristics are set statically before generation of the OSEK application. The conformance tests must be able to verify all the possibilities. Different messages and the associated parameters will therefore be defined and implemented in the conformance test application in order to be able to cover the different underlying options of the COM-API. To sum up the possible options and the associated minimum conformance class are as follows :

- withoutcopy/unqueued (CCC0), withcopy/unqueued (CCC0) or queued (CCC3) message,
- direct (CCC0), periodic (CCC1) or mixed (CCC1) transmission,
- application signalling mode (task activation, event setting) (CCC1),
- local (CCC0), UUDT (CCC0) or USDT (CCC2) transfer.

NM-API services

The NM specification defines two main variants that will determine the test suite's selection for an NM implementation:

- Direct Management,
- Indirect Management.

Direct NM services

The services of direct management can be broken down into core services required in every implementation and optional services. They are described in Table 7 below.

Direct Management Services	Service Call	Core	Optional
Configuration management			
- Select the indication sensitivity	<i>SelectDeltaConfig</i>		X
- (Re)start the configuration management	<i>InitConfig</i>		X
- Make current configuration available	<i>GetConfig</i>	X	
- Comparison of two configurations	<i>CmpConfig</i>		X
Operating mode management			
- Start of NM, i.e. transition to NM mode 'NMon'	<i>StartNM</i>	X	
- Stop of NM, i.e. transition to NM mode 'NM Shutdown' and finally to 'NMoff'	<i>StopNM</i>	X	
- Transition to NM mode 'NMpassive' without network-wide notification	<i>SilentNM</i>		X
- Transition to NM mode 'NMactive' after a previous call of SilentNM	<i>TalkNM</i>		X
- Transition to a new operating mode (e.g. BusSleep, Awake)	<i>GotoMode</i>		X
- Select the indication sensitivity	<i>SelectDeltaStatus</i>		X
- Get status information (network, node)	<i>GetStatus</i>		X
- Comparison of two states	<i>CmpStatus</i>		X
Data Field management			
- Transmit data	<i>TransmitRingData</i>		X
- Read received data	<i>ReadRingData</i>		X

Table 7 Core and optional services of Direct NM

The list of test cases applicable to a given implementation will depend on optional services actually implemented. The NM test suite will therefore specify the needed options for each test case.

System generation services have been removed from Table 7 since only run-time services will be tested. However, they define a way to notify NM events to application during network operation and they can therefore be assimilated to indication services as presented in the table below. For the sake of clarity, a name has been assigned to each service.

Indication services are not new services but a means to formalise notification actions that should be performed by the implementation. They can be implemented with OS task or event activation mechanisms, or they can solely be notified by setting the network status or the configuration status.

Constructional element	NM indication service	Service name
<i>InitIndDeltaConfig</i>	- Indication of configuration change	<i>Config_ind</i>
<i>InitIndDeltaStatus</i>	- Indication of status change	<i>Status_ind</i>
<i>InitIndRingData</i>	- Indication of ring data reception	<i>RingData_ind</i>

Table 8 Indication services of direct NM

Indirect NM services

The services of indirect management can be broken down into core services required in every implementation and optional services. They are described in Table 9 below. Indirect NM is based on monitoring of OSEK/COM message transmissions by the local node or message receptions by the remotes nodes. The protocol includes two exclusive options called One Global Timeout and One Timeout per Message.

Indirect Management Services	Service Call	Core	Optional
Configuration management			
- Select the indication sensitivity	<i>SelectDeltaConfig</i>		X
- (Re)start the configuration management	<i>InitConfig</i>		X
- Make current configuration available	<i>GetConfig</i>	X	
- Comparison of two configurations	<i>CmpConfig</i>		X
Operating mode management			
- Start of NM, i.e. transition to NM mode 'NMon'.	<i>StartNM</i>	X	
- Stop of NM, i.e. transition to NM mode 'NM Shutdown' and finally to 'NMoff'	<i>StopNM</i>	X	
- Transition to a new operating mode (e.g. BusSleep, Awake)	<i>GotoMode</i>		X
- Select the indication sensitivity	<i>SelectDeltaStatus</i>		X
- Get status information (network, node)	<i>GetStatus</i>		X
- Comparison of two states	<i>CmpStatus</i>		X
Processing of monitoring information (1)			
- Indication of monitored message transmission/reception	<i>I_MsgTransfer.ind</i>	X	
- Indication of monitoring time-out expiry	<i>I_MsgTimeout.ind</i>	X (2)	

(1) services called by the COM-IL to deliver monitoring information

(2) service implemented only in One Timeout per Message option

Table 9 Core and optional services of indirect NM

2.3.3. Event management

The concept of black box testing implies an event-driven specification of the test suites. In protocol specifications, the events are arranged in specification inputs and specification

outputs which will be henceforth called inputs and outputs for simplification. The main event categories are presented in the table below.

Event type	Input	Output	Interface
local service	procedure call	procedure return	user
network service	request	indication, confirmation	user
OPDUs	reception	transmission	network
Timer	expiry	start, stop	OS

Table 10 Protocol event categories

Service events

User services have been split into local services and network services:

- Local services are used to obtain local information maintained by the COM or NM module. Usually, execution of local services does not impact the operational behaviour of the COM/NM implementation (e.g. *GetStatus*). The requested information is provided in output parameters when the called procedure returns to the calling application.
- Network services are used to transmit or modify network-wide information. The service is not completed when the called service procedure returns to application. A uniform interaction model can be defined to describe the interactions between the tested implementation and the conformance tester. It consists of three primitives called request, indication and confirmation. The request primitive is the procedure call done to request a service execution. Indication and confirmation primitives represent call-backs from the tested implementation to notify the end of service execution to the conformance tester. They can exist or not depending on the requested service. An indication corresponds to a remote notification and a confirmation to a local notification. Depending on the service and on implementation choices the notification can be implemented by task activation or event signalling, or it can be implicitly known by polling methods.

For instance, the full model of interaction applies to message transfer when notification of transmission and reception are implemented:

- request for transmission: *SendMessage*,
- indication of reception: *Rx_ind*,
- confirmation of transmission: *Tx_ind*.

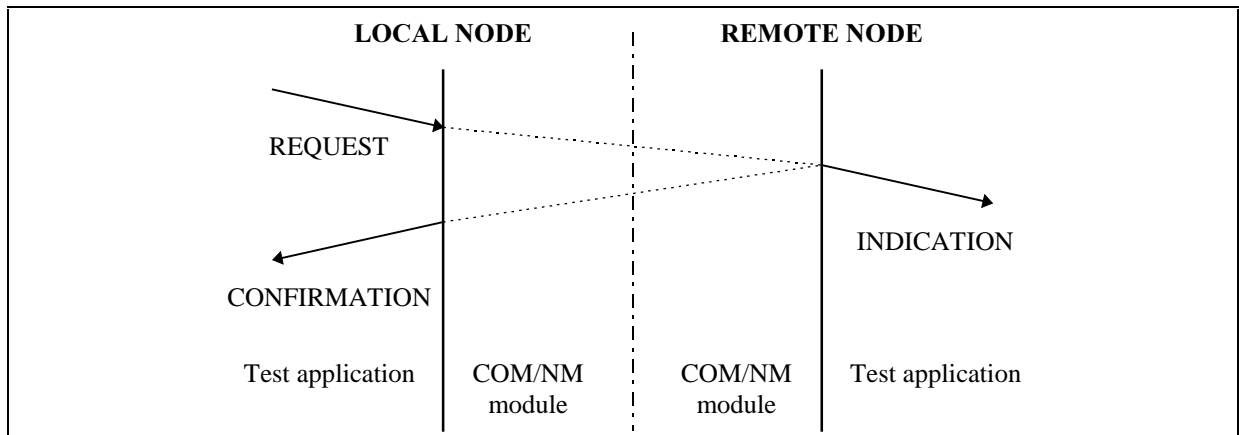


Figure 5 Service interaction model

Timer events

The OSEK specification makes use of two types of timers:

- periodic timers to trigger periodic protocol tasks,
- waiting timers defining the maximum allowed time for a specification's input event to occur.

Timer events are in principle neither controllable nor observable, since the COM and NM interfaces with OS are not visible from a conformance test application. Nevertheless, the time-outs can be most of the time artificially triggered by the tests:

- for a periodic time-out, the tester has to wait for enough time to observe the output(s) generated by the time-out,
- for a waiting time-out, the tester has not to send the expected input and wait for enough time to observe the output(s) generated by the time-out.

The conformance test specification will not claim a precise measurement of timer values. However, some tests will be designed for checking that measured values meet the implemented values with a certain margin of error.

COM events

The COM events simulated or processed during conformance tests execution are listed in the following tables.

The COM-API services are described in section 2.3.2. The related events are listed in the following table according to the event classification presented above.

Local events of COM-API	Network events of COM-API		
Procedure call	Request	Indication	Confirmation
<i>ReceiveMessage</i>	<i>StartCOM</i>		<i>none</i>
<i>ReceiveMessageFrom</i>	<i>SendMessage</i>	<i>Rx_ind, NoRx_ind</i>	<i>Tx_ind, NoTx_ind</i>
<i>GetMessageResource</i>	<i>SendMessageTo</i>	<i>Rx_ind (*)</i>	<i>Tx_ind, NoTx_ind</i>
<i>ReleaseMessageResource</i>			
<i>GetMessageStatus</i>			

(*) periodic/mixed transfer is not allowed with *SendMessageTo*, so *NoRx_ind* cannot happen

Table 11 COM-API events

The COM-OPDUs and the associated protocol fields are described below. UUDT transfers make only use of one type of frame. In USDT transfers, the data receiving protocol must send flow control frames to stop or resume the data transfer. The first flow control is sent after the first data frame. Then data are sent per blocks of consecutive frames. USDT protocol also allows to send unsegmented data within a single frame.

The specification defines two formats of protocol encoding called normal addressing and extended addressing. The latter includes an additional byte of information called extended address.

COM OPDUs	Fields	Definition
<i>UUDT Frame</i>	Data	UUDT data transfer
<i>SingleFrame</i>	PCI, DL, Data	USDT data frame for unsegmented transfer
<i>FirstFrame</i>	PCI, XDL-DL, Data	USDT's first data frame of segmented transfer
<i>ConsecutiveFrame</i>	PCI, SN, Data	USDT's other frames of segmented transfer
<i>FlowControl Frame</i>	PCI, FS, BS _{max} , ST _{min}	Sent by USDT receiver to stop/resume transfer

Table 12 COM OPDUs

Legend:

- Data: user data.
- PCI: Protocol Control Information identifying the type of USDT frame.
- XDL, DL: (eXtended) Data Length = user data length. In segmented transfer, XDL-DL ≤ 4095.
- SN: Sequence Number identifying the data segment of segmented transfer.
- FS: Flow Status = clear to send if sender can resume data transfer, or wait if sender must stop data transfer.
- BS_{max}: maximum Block Size = number of ConsecutiveFrames sent without intermediate flow control. $1 \leq BS_{max} \leq 255$. 0 means that no flow control is performed.
- ST_{min}: minimum Separation Time between successive ConsecutiveFrames.

For segmented transfers, the conformance tests must be able to check whether the sending and receiving protocols are correctly implemented and therefore a tester must:

- verify the sequence of OPDUs transmitted by the sending protocol and generate the appropriate flow controls,

- generate appropriate data frames to the receiving protocol and verify that it transmits correct flow controls.

The tester must also support the two types of encoding formats.

The protocol timers managed by the COM network layer are described below. They are implemented only in USDT protocol. The related time-out events will be simulated in conformance tests.

Timers	Side	Definition
<i>As</i>	sender	Time-out for data frame transmission by the DLL
<i>B1</i>	sender	Time-out for flow control reception after FirstFrame transmission
<i>B2</i>	sender	Time-out for flow control reception after ConsecutiveFrame transmission
<i>D2</i>	sender	Time-out between two successive flow control receptions
<i>Ar</i>	receiver	Time-out for flow control transmission by the DLL
<i>C</i>	receiver	Time-out between flow control transmission and ConsecutiveFrame reception
<i>D1</i>	receiver	Time-out between two successive ConsecutiveFrame receptions

Table 13 Time-out events of COM network protocol

NM events

The NM events simulated or processed during conformance tests execution are listed in the following tables.

Direct NM events

The API services of direct NM are described in section 2.3.2. The related events are listed in the following table according to the event classification presented above.

Local events of direct NM API	Network events of direct NM API		
Procedure call	Request	Indication (1)	Confirmation (1)
<i>SelectDeltaConfig</i>	<i>StartNM</i>	<i>Config_ind</i>	<i>Status_ind</i>
<i>GetConfig</i>	<i>StopNM</i>	<i>Config_ind</i>	<i>Status_ind</i>
<i>CmpConfig</i>	<i>InitConfig</i>	<i>Config_ind</i>	<i>Status_ind</i>
<i>SelectDeltaStatus</i>	<i>SilentNM</i>	<i>Config_ind</i>	<i>Status_ind</i>
<i>GetStatus</i>	<i>TalkNM</i>	<i>Config_ind</i>	<i>Status_ind</i>
<i>CmpStatus</i>	<i>GotoMode</i>	<i>Status_ind</i>	<i>Status_ind</i>
<i>ReadRingData</i>	<i>TransmitRingData</i>	<i>RingData_ind</i>	

(1) events that modify the status of a node may lead to a status change indication inside the node and to a configuration change indication inside remote nodes

Table 14 Events of direct NM API

The direct NM specification defines seven OPDU types (also called NMPDUs) described below.

OPDUs of direct NM			Definition
Type	Sleep.ind	Sleep.ack	
<i>Ring message</i>	<i>cleared set set</i>	<i>cleared cleared set</i>	Message transmitted when the local node belongs to the logical ring
<i>Alive message</i>	<i>cleared set</i>	<i>don't care don't care</i>	Message transmitted to ask for registration to the logical ring
<i>Limphome message</i>	<i>cleared set</i>	<i>don't care don't care</i>	Message transmitted during failure recovery period

Table 15 OPDUs of direct NM

The protocol timers managed by direct NM are described below. The related time-out events will be simulated in conformance tests.

Timers of direct NM	Definition
T_{Typ}	Typical time interval between two ring messages
T_{Max}	Maximum time interval between two ring messages
T_{Error}	Time interval between two ring messages with NMLimphome identification
$T_{WaitBusSleep}$	Time the NM waits before transmission in NMbussleep

Table 16 Time-out events of direct NM protocol

Timer T_{Tx} (delay to repeat a NM message's transmission request when it was rejected by the DLL) will not be simulated since it appears as implementation dependent. Such a functionality can be present or not depending on the DLL implementation.

Indirect NM events

The API services of indirect NM are described in section 2.3.2. The related events are listed in the following table according to the event classification presented above. Monitoring services activated by the COM-IL/NM API can be assimilated as local services.

Local events of indirect NM API	Network events of indirect NM API		
Procedure call	Request	Indication (1)	Confirmation
<i>SelectDeltaConfig</i>	<i>StartNM</i>		<i>Status_ind</i>
<i>GetConfig</i>	<i>StopNM</i>		<i>Status_ind</i>
<i>CmpConfig</i>	<i>InitConfig</i>		<i>Status_ind</i>
<i>SelectDeltaStatus</i>	<i>GotoMode</i>		<i>Status_ind</i>
<i>GetStatus</i>		<i>Config_ind</i>	
<i>CmpStatus</i>			
<i>I_MsgTransfer.ind</i>			
<i>I_MsgTimeout.ind</i>			

(1) In indirect NM, API calls are of local scope. Configuration indications result from monitoring of the COM messages by the NM protocol.

Table 17 Events of indirect NM API

As it is based on COM message monitoring, the indirect NM protocol does not implement any specific OPDU.

In option One Global Time-out, the indirect NM makes use of a periodic timer called Time-out for OBServation and described below.

Timer of indirect NM	Definition
<i>TOB</i>	Period of network status update

Table 18 Time-out event of indirect NM protocol

Network faults

The conformance tests must be able to verify IUT behaviour in case of faulty network. In NM specification, two types of network faults are identified and therefore must be processed by implementations:

- "transmission error": IUT can receive but cannot transmit,
- "fatal bus error": IUT can neither receive nor transmit (e.g. CAN bus off).

In COM specification, only the transmission error is processed. It causes timers As (data sending side) or Ar (data reception side) to expire. In case of fatal bus error, COM exchanges are suspended by the NM module through the D_Offline call to COM/DLL.

Generation of network faults by the tester requires the development of adapted hardware depending on the physical network (CAN, VAN...). If such hardware cannot be made available, network faults shall be at least simulated by adhoc software. Otherwise, it will not be possible to perform a meaningful conformance testing, since a large part of the COM and NM protocols is devoted to network failure recovery. Error simulation software must be incorporated at the interface between the COM/DLL and the network drivers.

2.3.4. Definition method for test suites

As for the OS, the COM and NM test suites will be specified in TTCN language. TTCN is an ISO standard [6] especially designed for describing conformance tests of communication protocols. TTCN has been widely used in the telecommunication area. TTCN's main features are presented later in this document.

2.4. Strategy for OSEK modules integration and testing

2.4.1. OSEK integration and conformance testing

The modularity and scalability of the OSEK specification enables a progressive integration of OSEK software in target equipment. This can be done in three steps:

1. OS module,
2. COM module which makes use of OS services,
3. NM module which makes use of OS and COM services.

As well, separate test suites are going to be specified for each of the three modules. As there is no correlation between interfaces to be tested by each of them, the three test suites can be considered as independent of each other. Therefore, they will enable in turn a progressive conformance assessment of the OSEK implementation during integration. The three steps of integration and conformance testing are shown in figure below.

1. OS module integration and OS conformance testing.
2. COM module integration with certified OS, then COM conformance testing.
3. NM module integration with certified OS and COM, then NM conformance testing.

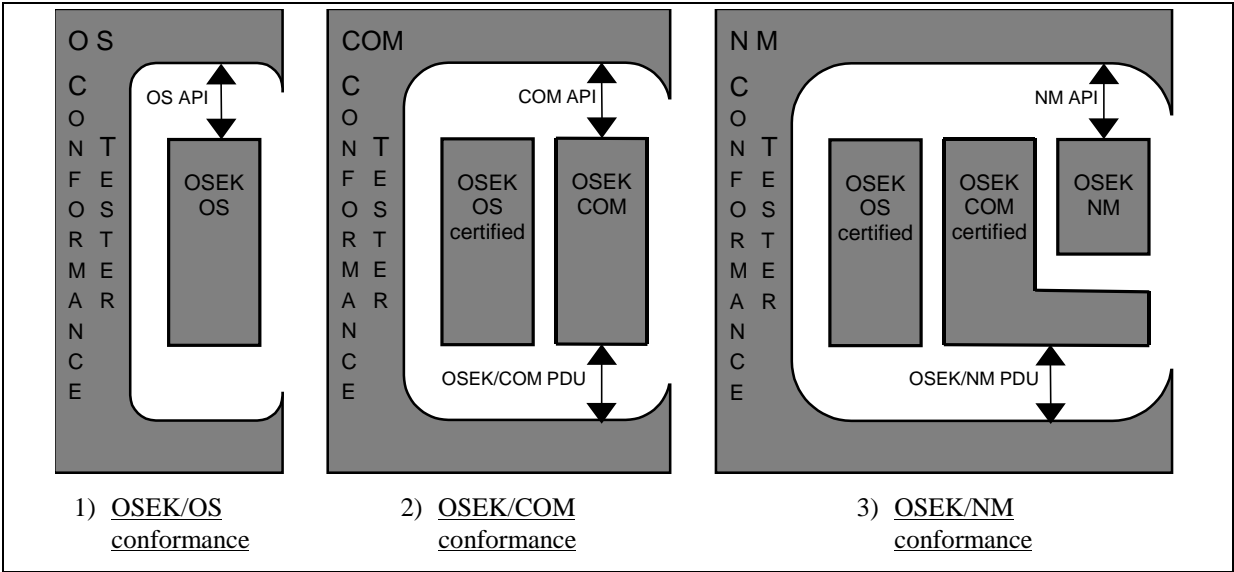


Figure 6 Progressive integration and conformance testing of OSEK modules

2.4.2. Conformance testing of a complete OSEK implementation

For both the OSEK developer and the end-user, it is essential that conformance should be verified once all OSEK modules have been integrated in the final equipment and no further software modification will be made. Again, thanks to modularity and independence of the test suites, the conformance tests can be executed against the complete OSEK implementation.

They can also be replayed in the life cycle of the product to verify non regression after any modification.

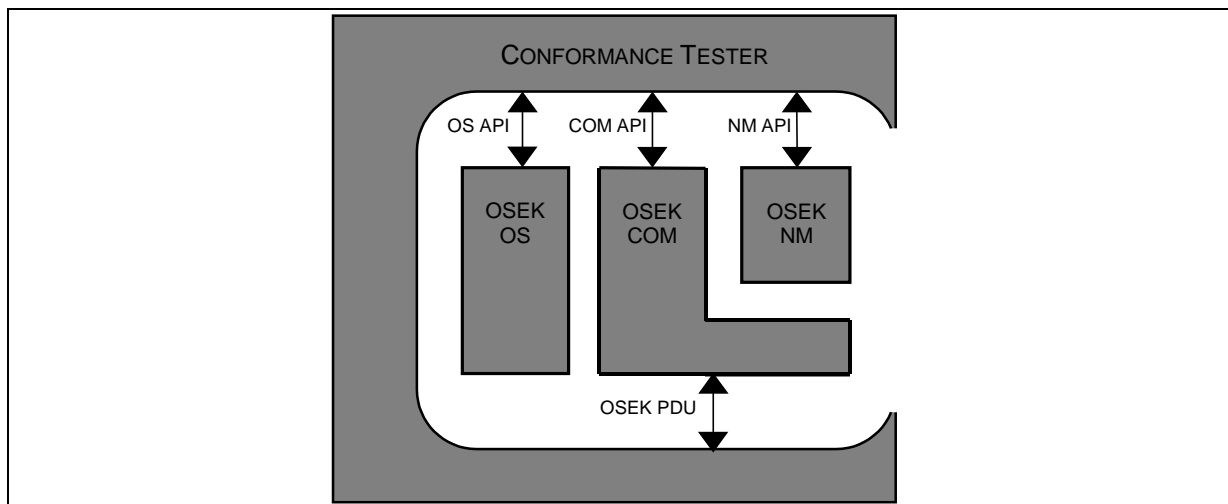


Figure 7 Conformance testing configuration for complete OSEK implementation

In some products, OSEK/COM and OSEK/NM implementations can be associated to a non OSEK OS and despite this, their conformance shall be evaluated. The modularity of the test suites also enables to cope with this situation as presented in Figure 8 below:

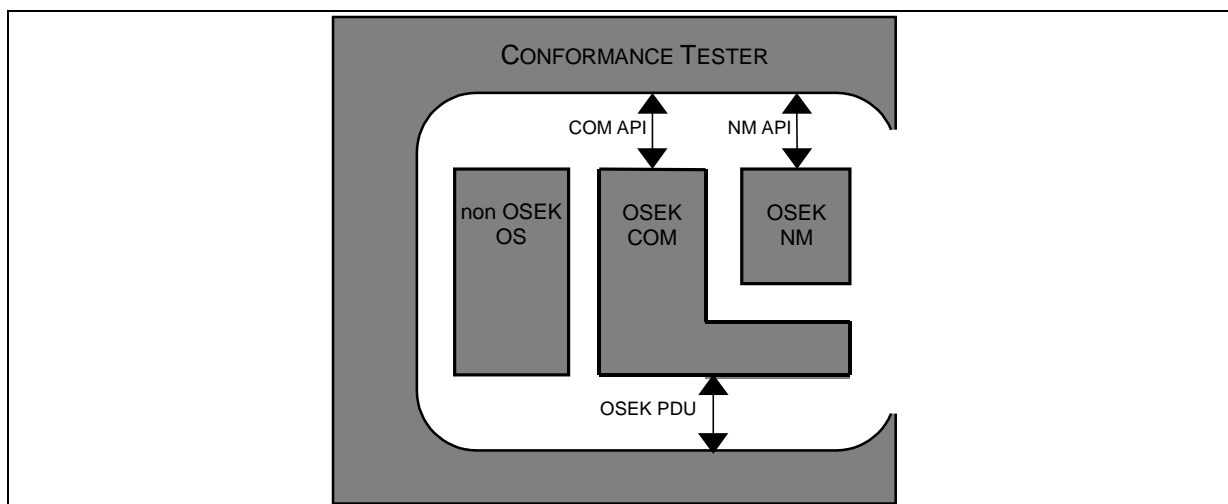


Figure 8 Conformance testing configuration for implementations without OSEK OS

3. Test architecture for COM and NM

3.1. Description of the test architecture

COM and NM services can be split into two categories:

- Local services such as sending a message to another local task. API calls are entirely processed inside the tested implementation's equipment.
- Network services such as sending a message to another equipment. They require data exchanges with a remote COM or NM implementation using the OSEK protocols.

The first ones are tested through local procedures and the same techniques are employed as in OS conformance testing. The latter involve an external equipment playing the role of the remote OSEK implementation. It is called Test Equipment while the equipment containing the OSEK implementation is called Equipment under Test. The OSEK implementation to be tested is itself called Implementation Under Test (IUT).

To test OSEK protocols, the conformance tester needs to access two interfaces of the IUT:

- The network interface for exchanging OSEK PDUs via the interconnection network.
- The service interface for exchanging service information via OSEK APIs.

Therefore, the conformance tester consists of two distinct modules:

- The Upper Tester (UT) which communicates with IUT's upper interface through APIs. It is implemented at the top of the IUT and in the Equipment Under Test.
- The Lower Tester (LT) which communicates with IUT's lower interface through PDUs. It is implemented in a Test Equipment connected to the IUT via the physical network.

This architecture is the so-called Coordinated Test Architecture of ISO 9646. Indeed, the respective actions of Upper and Lower Testers shall be Coordinated during a test case execution. The co-ordination is performed by a specific protocol called Test Management Protocol (TMP) which forms part of the test procedure specification. TMP data are exchanged between LT and UT by the means of TM_PDUs (Test Management PDUs).

The test architecture and the interactions between UT, LT and IUT are illustrated by the following figure. In actual implementations, TM_PDUs are transferred using the data transmission services of the IUT. They are therefore encapsulated in the data part of OSEK PDUs. The IUT does not interpret TM_PDUs but it only passes them from UT to network and conversely.

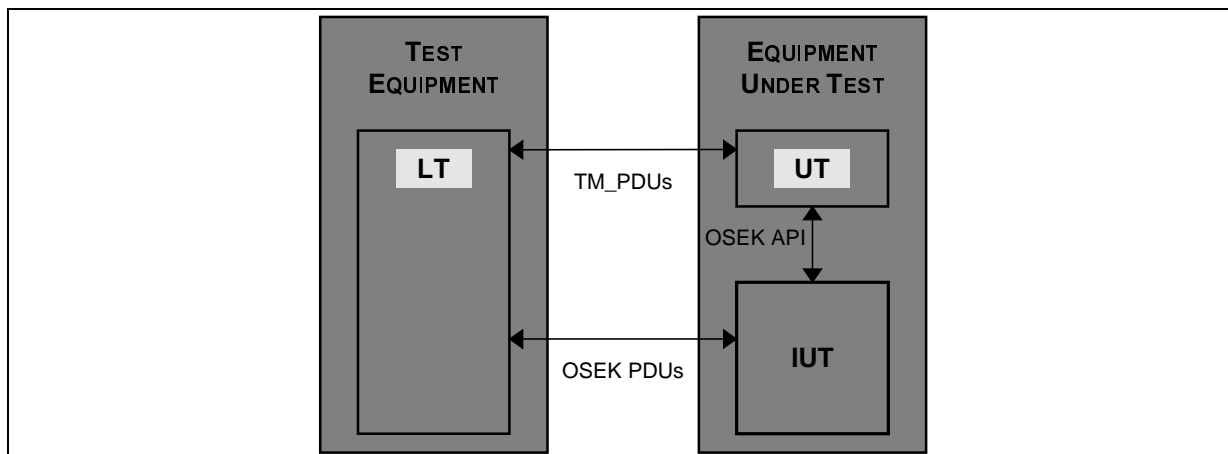


Figure 9 Principle of test architecture

3.2. Rules of UT, LT and TMP specification

The rules governing the definition of LT, UT, and TMP are twice:

- The UT shall be generic and as simple as possible. Since the UT is implemented in the same equipment as the IUT, it needs to be customised by each implementator according to target specific constraints. At least it must be recompiled and linked to IUT software. Therefore the maximum functionality of conformance tools is transferred to LT. "Generic and simple" also means that UT specification must be as far as possible independent of the selected test cases for a given IUT.
- The TMP shall use the minimum services and protocols of the IUT. Using IUT services is requested to transfer TM_PDUs and the simplest protocols shall be used to do it in order not to disturb the test execution and not to duplicate much of conformance tests in TM_PDU transfer operations. Specific test cases need to be added to the test suite for the purpose of testing IUT's ability to transfer TM_PDUs. Such tests also aim to verify that the UT has been correctly customised. They shall be executed at the beginning of the test campaign.

Execution of the test suite is therefore entirely driven by the LT. The LT is in charge of performing all the actions specified in the test case. It controls the operation of the UT in ways necessary to run the tests selected for the IUT. It analyses the test results and computes the verdicts.

The role of UT is limited to interpretation and execution of LT commands. The UT shall also return to LT all data collected from the IUT at the API level. It never calls API services on its own except commands enabling network communications such as StartCOM and StartNM. The TMP is a two-way protocol operating as follows:

- from LT to UT, it conveys API calls and parameters the UT must then send to the IUT,
- from UT to LT, it conveys information returned by API calls and indications of OS events or task activation originating from the IUT.

As stated before, conformance testing requires the simulation of network faults that can be either generated by hardware means or simulated by software. In case of software simulation, a TM_PDU will be sent by the LT to notify the equipment under test of the type of fault (transmission error, fatal bus error) and its duration. This TM_PDU is not transmitted to the UT. It must be interpreted by the network driver of the Equipment Under Test which will

simulate the requested fault. Return to a non faulty behaviour will be ordered by another TM_PDU.

Since the UT is independent of the executed test case and operates as an application protocol implemented at the top of the IUT, it will be specified in SDL like COM and NM protocols. The specification will be incorporated in the OSEK conformance standards as part of the test procedure. According to OSEK recommendations, OIL [8] could be used to define configuration parameters of UT implementations.

The TMP is a simple point-to-point send/receive protocol. A predefined DLL connection is assigned to each direction of transmission. Depending on tester's implementation, TM_PDUs can be sent or receive:

- either at the COM-API level using the SendMessage/ReceiveMessage interface. TM_PDUs are exchanged on UUDT connections mapped upon the predefined DLL connections,
- or at the DLL-API level using the OSEK D_UUData service or any convenient DLL driver interface.

Either of those options can be chosen for NM conformance. They lead to the two possible architectures of the equipment under test presented below. In principle, COM conformance will use the first possibility since the COM-API is always present.

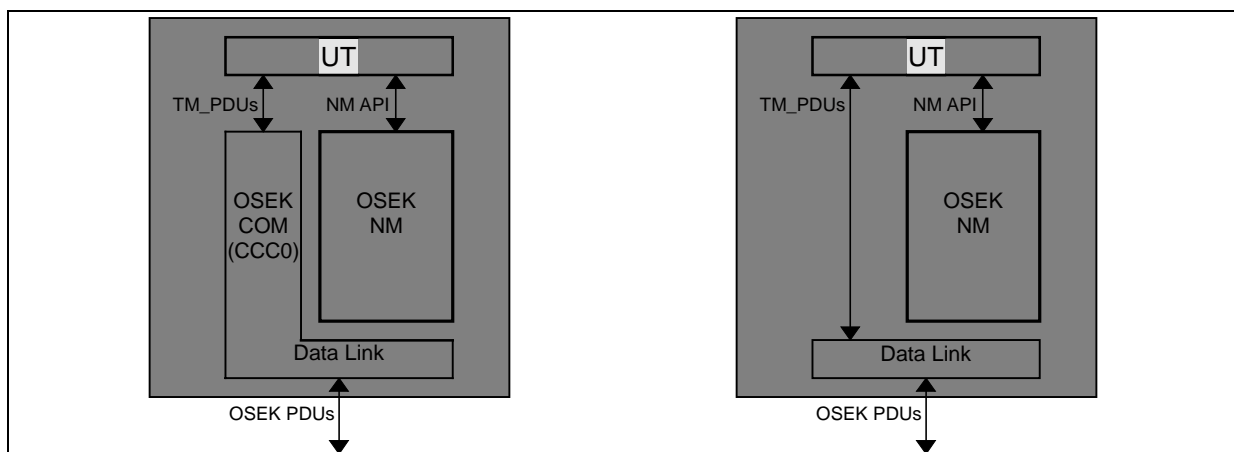


Figure 10 'Equipment under test' architectures for NM conformance

Overall design rules can be set up to specify the TMP and the UT. They depend on the type of service requested by LT or on the type of event occurring at the UT/IUT interface. The table below specifies the TM_PDUs exchanged by LT and UT according to the event classification presented in section 2.3.3:

API event	TM_PDU		
	from	to	contents
Request	LT	UT	API to be called and parameters
	UT	LT	Status returned by API call (1)
Indication, confirmation	UT	LT	Type of event and parameters (2)
Local service	LT	UT	API to be called and parameters
	UT	LT	Status and data returned by API call (1)
None	LT	UT	Configuration parameters

(1) The TM_PDU can be sent out or not, depending on information given in message from LT to UT.

(2) The TM_PDU can be sent out or not, depending on some configuration parameter.

Table 19 TMP and UT specification rules

3.3. TMP mechanisms

Figure 11 and Figure 12 show the required protocol mechanisms to order the UT to send or receive application messages. These communication scenarii will be run at the start of a test campaign in order to verify the correct operation of the UT. They describe basic actions the UT will have then to perform in any test case.

The scenarii assume a task is activated within the UT each time a message is received (Message_ind event). A TMP task is activated upon TM_PDU reception and a "test" task upon reception of test case's OPDUs. The actions performed on each task activation are as follows:

- (1): the UT acquires the TM_PDU and executes the requested SendMessage,
- (2): the UT sends a TM_PDU to notify the LT of a message arrival,
- (3): the UT acquires the TM_PDU and executes the requested ReceiveMessage. Then it sends the message back to LT with a SendMessage.

Four primitives have been defined to describe LT actions:

- SendOPDU(*Msg*) to transmit the test case message *Msg*,
- ReceiveOPDU(*Msg*) to receive the test case message *Msg*,
- SendTM_PDU[*command*], to transmit a *command* to UT via a TM_PDU,
- ReceiveTM_PDU[*event*], to receive notification of an IUT *event* via a TM_PDU.

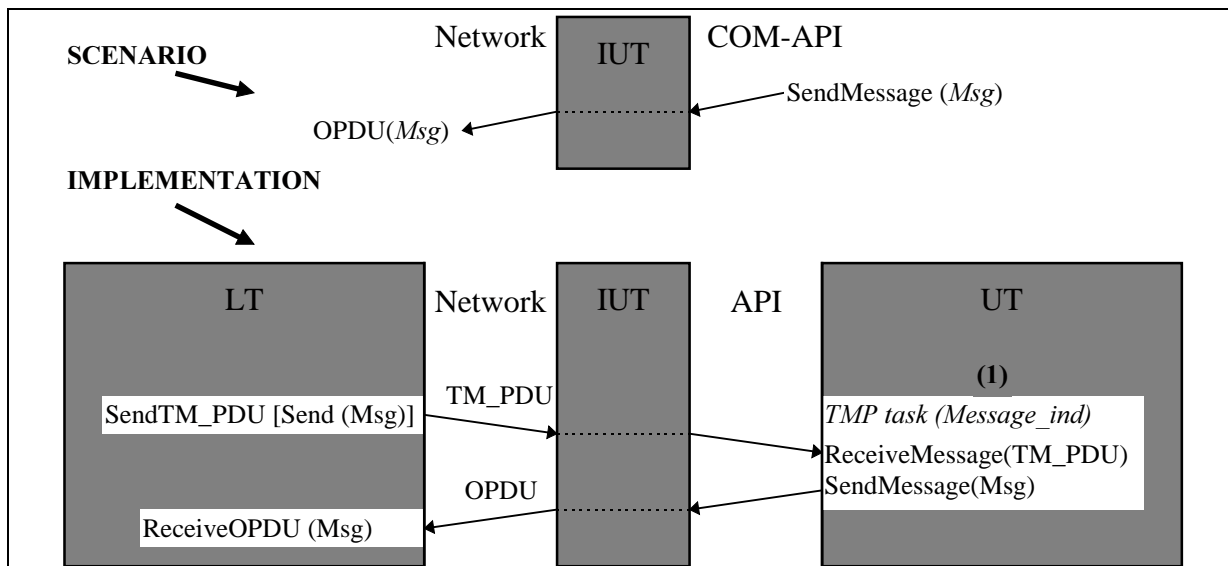


Figure 11 TMP mechanisms for message sending by UT

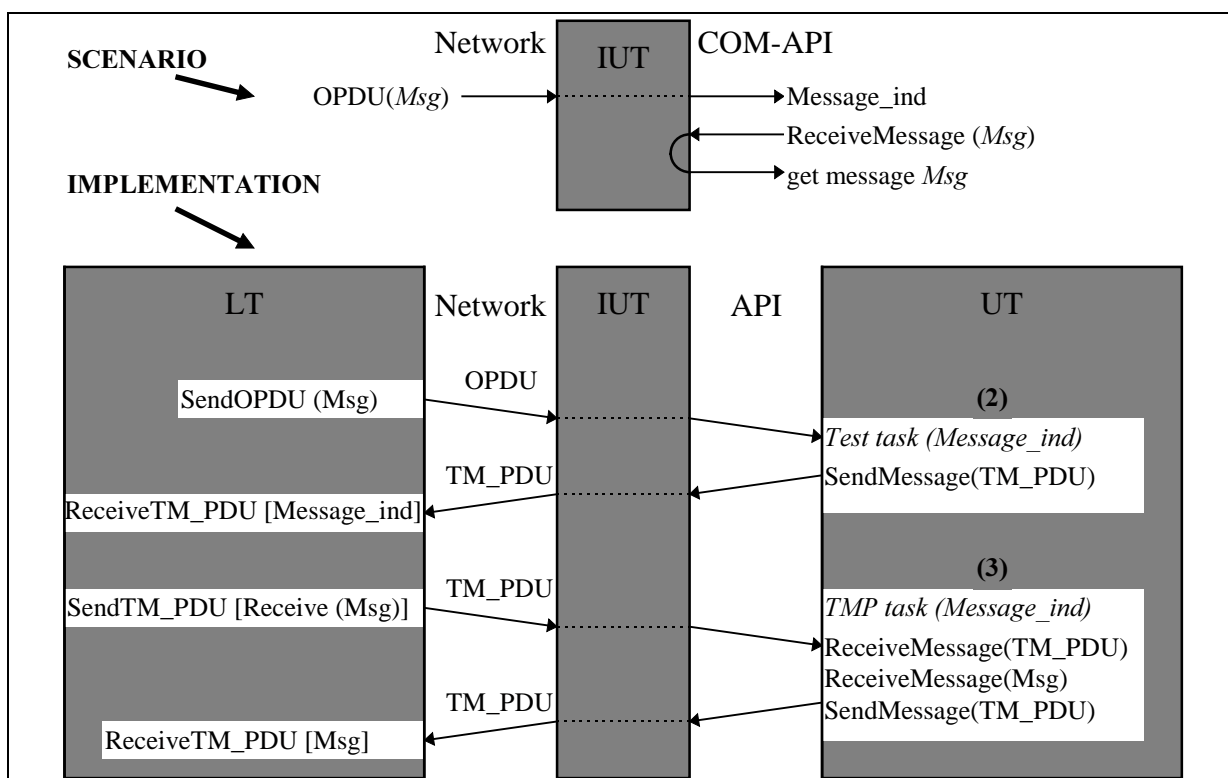


Figure 12 TMP mechanisms for message reception by UT

The message reception protocol can be simplified as presented in Figure 13 to reduce the protocol overhead. The simplified protocol will be used in almost all situations. The full protocol is only required when the LT does not want the UT to execute a `ReceiveMessage` on message reception. This happens for instance when testing implementation of reception FIFO mechanisms within the Interaction Layer.

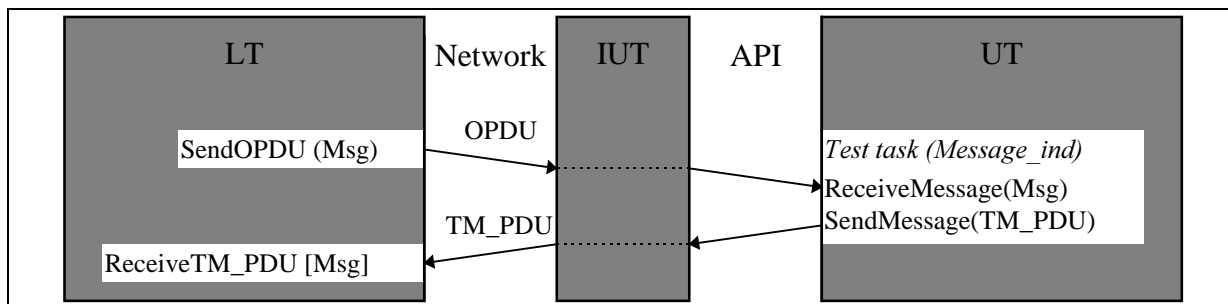


Figure 13 Simplified protocol for message reception by UT

3.4. Example of UT specification

Let us consider the following specification of a connection establishment and release protocol. The SDL specification is presented below. The protocol comprises three phases:

- Establishment request from network by *estab_m* PDU. If the request is accepted, the IUT generates the *estab_i* indication to application and the *ack_m* acknowledgement PDU to network. If not accepted, the IUT sends out an *err_m* PDU.
- Establishment response from application with *conn_r* request. The IUT transmits a *conn_m* PDU to the network.
- Connection release from network by *rel_m* PDU. The IUT generates a *rel_i* indication to application.

To test IUT conformance from a remote LT, the TMP shall be designed so as to return IUT's indications to LT and transmit LT requests to UT. TMP actions are as follows:

- the *estab_i* and *rel_i* indications are returned in the *estab_p* and *rel_p* TM_PDUs,
- the *conn_r* request is transmitted by the *conn_p* TM_PDU.

Therefore, the UT behaviour consists of:

- sending *estab_p* (resp. *rel_p*) TM_PDU on *estab_i* (resp. *rel_i*) event reception,
- sending *conn_r* request to IUT on *conn_p* TM_PDU reception.

The related SDL specification is given below.

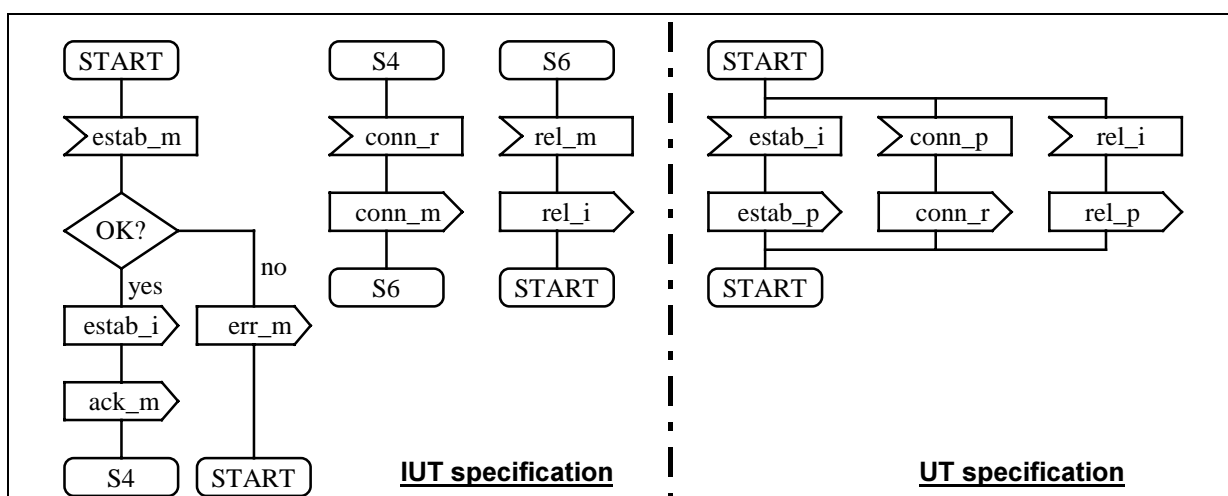


Figure 14 SDL specification of IUT and UT

The Figure below shows the scenario of a successful establishment and release and the associated test sequence to be observed when the UT is connected to IUT.

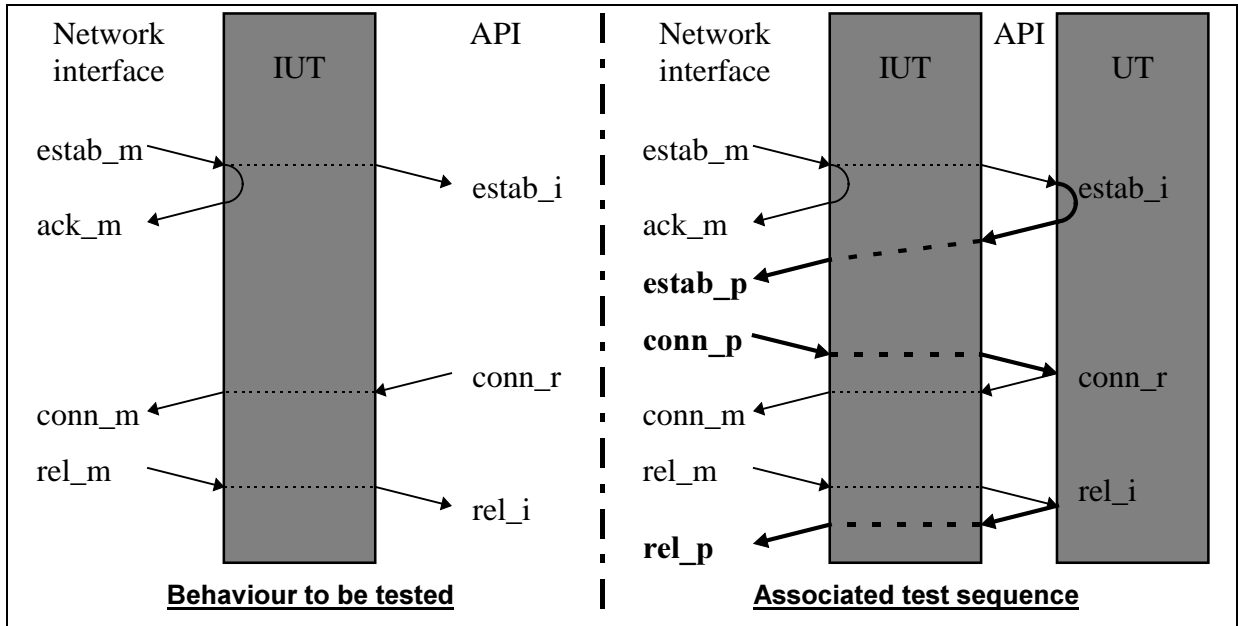


Figure 15 IUT scenario and associated test sequence

4. Methods of test suite generation

4.1. Generation of OS test suites

4.1.1. Generation method and supporting tool

The test suite compares the actual state of an OSEK system with its specified state. As the internal structure of the OSEK OS is not specified, the test suite will be implemented based on its API specification.

The implementation under test will be treated as a black box; its internal structure is not taken into account. Thus, this test suite can only show if the actual behaviour of the OSEK OS corresponds to the OSEK OS specification. Though, it's not possible with this method to detect in case of a failure what exactly caused this failure.

Black box testing also means that intrusions of the tested implementation are not allowed. Therefore the only interface the test suite is able to use is the API defined in the OSEK OS specification. Thus the test suite will be an OSEK application which uses the API services to ensure that each API service is called so much until every error described in the specification will be provoked at least one time.

Corresponding to the OSEK OS services the test suite will be grouped into the following test groups:

- Task management
- Interrupt handling
- Resource management
- Event control
- Alarms
- Operating system execution control
- Hook Routines

In addition the different variants have to be taken into account. This leads to a further grouping corresponding to the Conformance Classes (BCC1, BCC2, ECC1, ECC2) and to the Scheduling Policies (non-preemptive, full-preemptive, mixed-preemptive).

According to this different divisions the test cases will be grouped in the following order:

1. OS service group
2. Conformance class
3. Scheduling policy

The test cases will be created based on the API specification supported by the Classification-Tree Method. There are several tools that support this method e. g. CTE by ATS GmbH. This method has several benefits:

- It supports test case determination from unit to system testing
- Syntax-directed, graphical editor, that allows the user to comfortably create and modify classification trees in an object-oriented way.
- Automatic test coverage checking to ensure that all system entities are tested.

4.1.2. Test suite example

Figure 16 shows an example for the Classification-Tree of test group Resource Management. Table 20 shows the resulting test cases and their description.

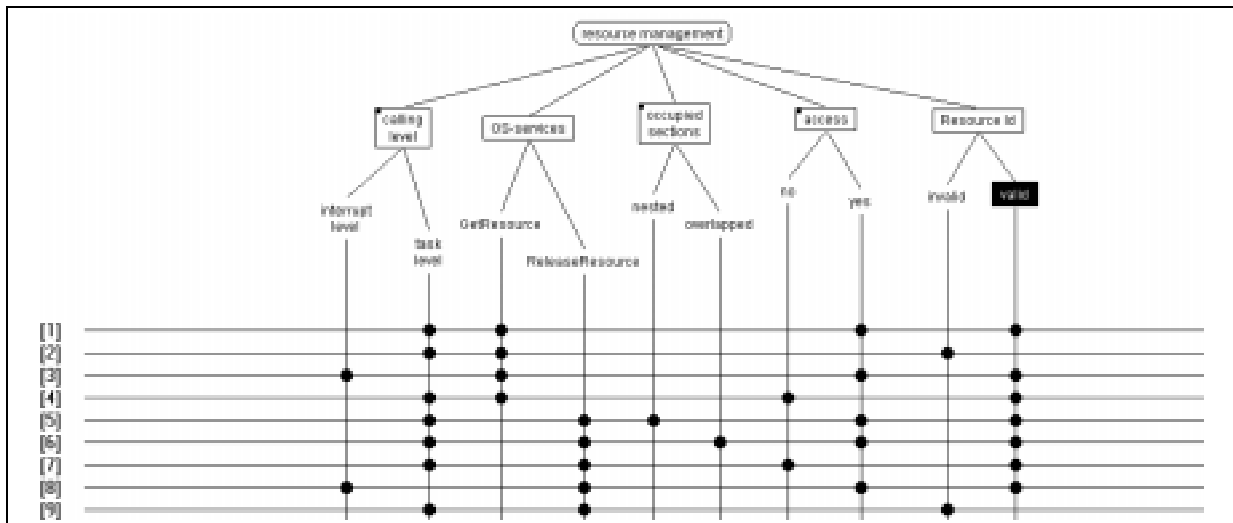


Figure 16 Classification-Tree of test group Resource Management

Test cases	Description
Test case 1	no error message (E_OK): because referred resource is not occupied
Test case 2	an error message (E_ID:): because the resource identifier is invalid
Test case 3	an error message (E_CALLEVEL): because the call is not allowed at the interrupt level
Test case 4	error message (E_Access): because the calling task has no access to the resource
Test case 5	no error message (E_OK): because at least two resources should be occupied and released in a nested way
Test case 6	at least two resources should be occupied and released in an overlapping way. The OS should not allow to release resources in an overlapping way.
Test case 7	error message (E_NOFUNC): the task wants to release a resource which is not accessible or not occupied
Test case 8	error message (E_CALLEVEL): because call is not allowed at interrupt level
Test case 9	error message (E_ID): because the resource identifier is invalid

Table 20 Test cases of test group Resource Management

From these test cases a test suite is derived and specified in that way that all test cases and all errors defined in the OS specification are called at least once. The development of an appropriate test suite will be supported by State- and Activity-Charts as described in the following chapter. The implementation of the resulting test suite is an OSEK application written in ANSI-C. It is so specified, that it requires only few resources and its source code is as compact as possible. A possible test suite for the Resource Management is proposed in Table 21. It consists of two tasks. TASK1 has priority 1. It is allowed to get resource1 and resource2 but not resource3. TASK2 has priority 0. It is allowed to get resource2 and resource3. The resource99 is not defined.

Steps	Description
Step 1	At system start, TASK2 is running.
Step 2	TASK2 gets resource3. That leads to no error (E_OK) (test case 1).
Step 3	TASK2 activates TASK1.
Step 4	Only at the non pre-emptive scheduling: TASK2 calls the scheduler.
Step 5	TASK1 is running.
Step 6	TASK1 gets resource1. That leads to no error (E_OK) (test case 1).
Step 7	TASK1 gets resource99. That leads to an error (E_ID), because resource4 is not defined (test case 2).
Step 8	TASK1 is interrupted by a ISR, which tries to get a resource1. That leads to an error (E_CALLEVEL) (test case 3).
Step 9	TASK1 gets resource2. That leads to no error (E_OK) (test case 1).
Step 10	TASK1 releases resource1. That leads to an error, because resource1 and resource2 are overlapped, but there is not error message (test case 6).
Step 11	TASK1 releases resource2. That leads to no error (E_OK) (test case 5).
Step 12	TASK1 releases resource2 again. That leads to an error (E_NOFUNC) (test case 7).
Step 13	TASK1 is interrupted by a ISR, which calls ReleaseResource(resource1). That leads to an error (E_CALLEVEL) (test case 8).
Step 14	TASK1 releases resource1. That leads to no error (E_OK) (test case 5).
Step 15	TASK1 releases resource99. That leads to an error (E_ID) (test case 9).
Step 16	TASK1 get resource3. That leads to an error. Because TASK1 is not allowed to get resource3 (E_ACCESS) (test case 4).
Step 17	TASK1 terminates itself. TASK2 is running.
Step 18	TASK2 releases resource3. That leads to no error (E_OK) (test case 5).
Step 19	TASK2 terminate itself, so no task will run.

Table 21 Possible test suite of test group Resource Management

4.1.3. Test generation tool

The generation of the test suite shall produce among other things a TTCN description. This implies the use of SDL (Specification and Description Language) because of several tools that can semi-automatically generate TTCN test suites from a SDL specification. But there are some restrictions regarding the use of SDL for specification of the OSEK OS:

- It is not possible to model all OS requirements completely, because the specification or description of non-functional requirements and constraints of a system is not supported.
- Algorithms are awkward to formulate in SDL, e. g. each interrupts requires a channel to the involved components.
- Inputs at the same time are stored in random order in the input queue.

Therefore the use of SDL does not seem to be practicable. Anyway, a TTCN description will be produced for documentation purposes and to keep interfaces for other tools, later on.

As OS test sequences are rather state-oriented than protocol-based as it is the case with NM and COM, it seems to be more appropriate to model the test suite using a CASE-tool which is based on State- and Activity-Charts like for instance Statemate by i-Logix Inc. State-/Activity-Charts are used to create an executable specification of a system. They allow to easily create a graphical model that represents the intended functions and the behaviour of the system. Compared with a textual description or C-Code, a graphical model has the benefit that it is easier to understand and to debug. Its behaviour is much more comprehensible and hence errors can rapidly be discovered and eliminated. There are also analysis tools to verify that the model meets the needed requirements.

This method comprises the following views of a system.

- **Statecharts** describe the timing behaviour of a system and control events and conditions that cause changes in the system's operation. Statecharts are evolved from state-transition diagrams, additionally they permit hierarchical states, concurrent or parallel processes and timing.
- **Activity-charts** represent the functional partitioning of a system and the data and control interfaces between the functional units. Each activity may be connected to a Statechart which models its behaviour. Activities can be further decomposed into smaller functions. The relationships between the functional and behavioural views can be checked for consistency.

Once a model has been created, Statemate's Check Model tool can be used to verify that it is complete and consistent. Statemate also provides a wide array of debugging and monitoring capabilities to quickly and easily verify that a model is working correctly. Therefore, it is possible to ensure that all states will be executed and each activity will be activated at least once during simulation. In addition, the simulation tool permits to connect user-defined code to a model's internal activities. In this way, each time this activity will be activated, the user-defined code will be executed.

The test suite, i. e. the sequence described in Table 21, will be modelled using State- and Activity-Charts. Each step will be described as a state in a Statechart, while each OSEK API service will be represented as an event (Figure 17). In each step of the test suite one API service is called and its result is compared to the value specified for this situation. Calling an API service call is modelled as the activation of the corresponding activity. At this point debugging and model checking tools can be used to test for completeness and correctness of the test suite as far as this is possible. This will verify that all states of the test suite model will be reached and thus all test cases will be executed.

For automatic code generation during simulation of the test suite, user-defined code will be attached to the model. This user code is triggered by the events which correspond to the OSEK API services and will be executed automatically by the simulator tool each time the event is generated, i. e. each time the corresponding API service is called. This API service call and some code to check the service's return value against its specified value will be written into the test suite's source file (ANSI C). In this way the code for the test suite will be generated during simulation of the corresponding Statemate model.

In a further step the generated code will be optimized as far as code size is concerned. This shall ensure that the test suite can be used even on platforms with low resources (ECU, ...).

Beside code generation of the test suite, an OIL file for configuration of the OSEK OS and a TTCN description of the test suite for documentation and as interface to other tools will be created. The OIL file will contain all application specific information needed. It will be up to the testing person to complete it with OSEK implementation specific matters (ECU type, ...)

There are two points which are important to point out:

- All tools mentioned above will only be used for the specification and generation of the test suite. They will not be needed for the conformance test itself. In other words, it will not be necessary for the OS implementator to acquire any of the tools to do conformance testing.
- Code will be generated during simulation of the test suite by the attached user code functions. This will produce an absolutely flat and efficient code. An additional optimization process will ensure that the generated code will be as small and resource saving as possible. As Statemate's code generator won't be used the code will be comparable to hand-written code.

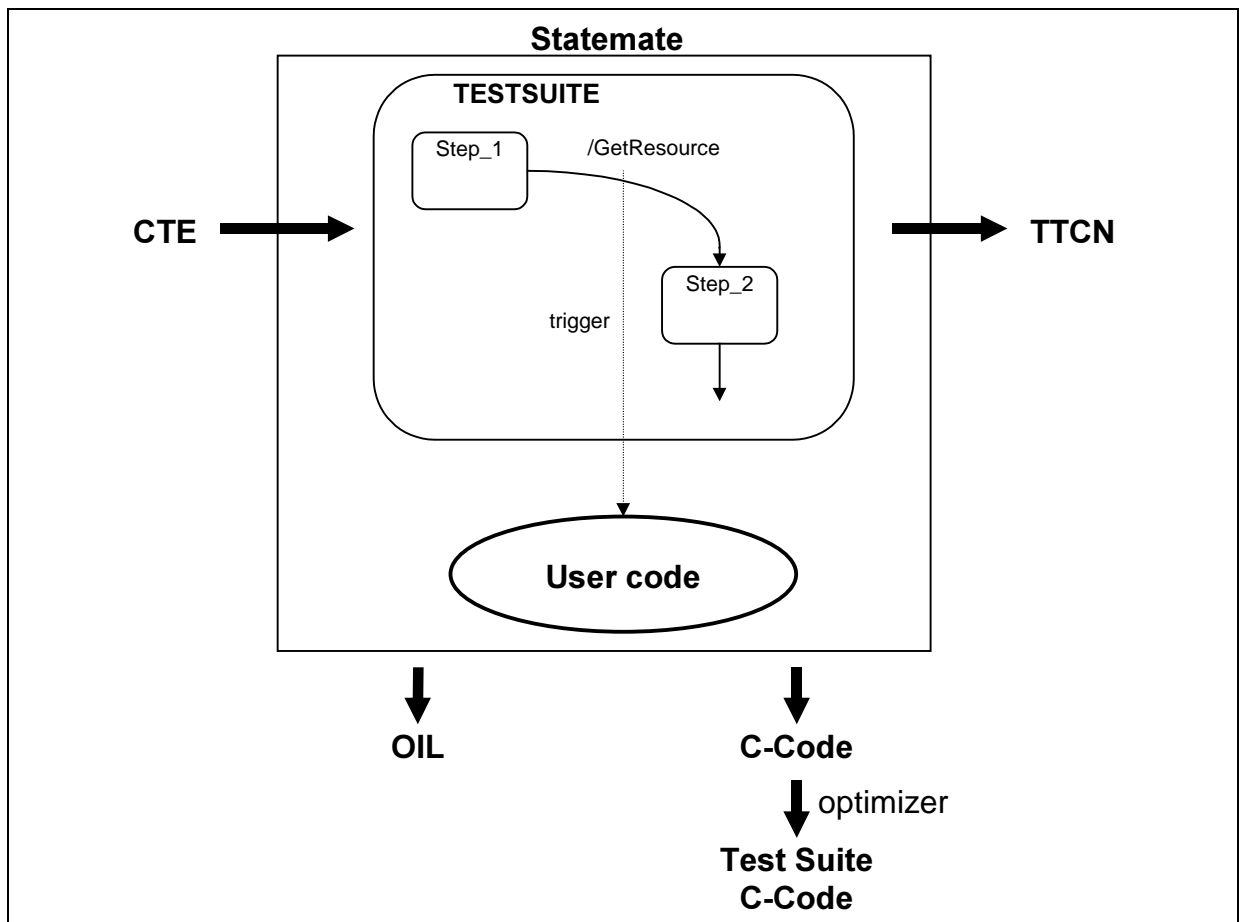


Figure 17 Modelling the test suite in Statemate

4.2. Generation of COM and NM test suites

4.2.1. Generation method

The generation of test suites for COM and NM will be mainly based on the SDL diagrams attached to the specification. However, the text and figures of the specification will be conscientiously analysed to derive checkable assertions according to the general principles presented in § 2.1. The assertions will be compared to the tests obtained from the SDL diagram analysis in order to generate a complete list of test purposes.

As concerns NM, the protocol is also defined by state/transition graphs. This form is the official representation. It will supersede the SDL definition in case of conflict.

The SDL diagrams give a graphical representation of the specification. They specify the protocol automata in a hierarchical manner. Each automaton is represented by an SDL process whose internal structure is a decision tree comprising:

- At the first level: the list of possible states of the automaton.
- At the second level: the list of events that may happen in a given state. Events can be external such as a user API call or internal such as an inter-layer API call.
- At the third level: the actions performed by the protocol when receiving a given event in a given state. Within the sequence of actions, the test of protocol variables may lead to subdivisions of the decision tree. The last action normally sets the new state of the protocol automaton.

The sequence of actions includes:

- assignments of protocol variables or of output event parameters,
- tests of protocol variables or of event parameters,
- sending of events. Again, events can be external (to the environment and hence to the conformance tester) or internal (to the same or another automaton),
- subroutines which may in turn include assignments, tests, sending of events and subroutines.

The definition method of the conformance tests aims at covering all branches of the specification tree. Whenever possible, a test purpose is specified for each complete and different branch (not for each segment). The following rules will be observed:

- Try and cover all protocol states (specification level 1),
- Try and cover all events specified in a given state and those events only (specification level 2)

Remark:

The assertion above means that most of robustness tests are excluded from the scope of OSEK conformance testing. The conformance tests do not aim to verify that the implementation should accept all possible events in every state if it is not specified that way. From the conformance point of view, unspecified events are considered as impossible. The tests cannot predict the implementation behaviour in such a situation, whether it should be "do not care and ignore that event" or "handle it as an error".

The tests aim to a static coverage of the specification. They will not check all possible sequences of events. For instance, let us suppose the specification defines two input

events Event1 and Event2. Execution of the test suite will arbitrarily lead to send either Event1 then Event2 or Event2 then Event 1, not both of them unless they are correlated by the use of the same protocol variables.

- Try and cover all branches of the action tree (specification level 3). Again, the coverage is static. Let us consider the following specification where two consecutive tests lead to define four branches B1, B2, B'1, B'2 (Fig. a). Four execution paths are possible B1B'1, B2B'2, B1B'2 and B2B'1, but two are sufficient to cover the specification as for example B1B'1 and B2B'2. Only two test purposes will be specified although the implementation could be designed as in Fig. b and in this case, partially covered by the conformance tests.

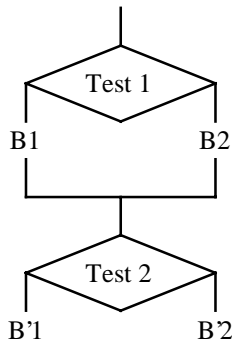


Fig. a: specification

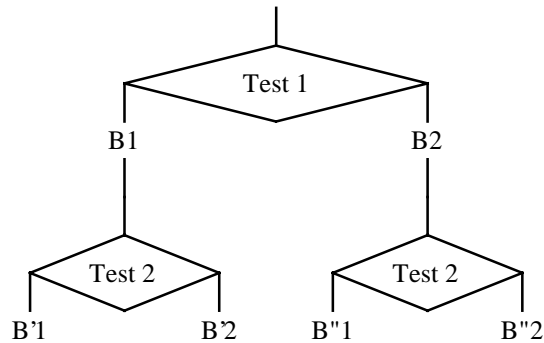


Fig. b: implementation

In the same manner, the subroutine are covered only once, not at each call, although they could be expanded each time into the main code of the implementation.

The test purposes are however meaningful only if the results can be observed by the conformance tester. Observable outputs consist of information returned by API calls or of PDUs transmitted by the protocol. Two branches of the specification tree can be differentiated for the same input only if they produce different outputs. The test purposes should be selected accordingly. In the figure below where V1 and V2 are non observable internal variables, two tests can be defined for Input1 and only one for Input 2. For Input 2, the test purpose will simply be "Send Input 2 and observe that nothing happens".

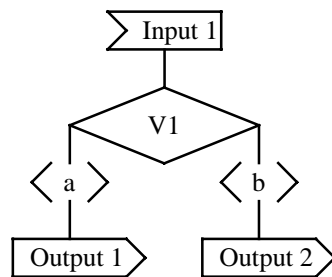


Fig. a: testable

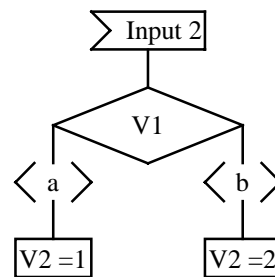


Fig. b: not testable

4.2.2. Impact of test architecture

The test architecture for protocols is composed of a UT and a LT. As the UT implements a generic behaviour independent of the test case being executed, specifying a test suite amounts to specify the associated LT behaviour. Therefore, a test case specification is made up of the sequence of PDUs, including TM_PDUs, exchanged by the LT and the IUT during test execution. The specification shall also define the verifications the LT has to perform on PDUs originating from the IUT.

Therefore, the test generation process shall not take into consideration the sole SDL specification of IUT, but rather the SDL resulting from the combination of IUT and UT specifications. After the combination, IUT APIs become internal interfaces of the IUT + UT set. API events will be ignored in test case definitions because they are not directly accessible to LT. Since they are notified to LT through TM_PDUs, they will be replaced by TM_PDU receptions.

To generate the combined specification, the SDL processes defining the IUT and the UT shall be associated two by two in order to remove their interactions and generate a unique process. The transformation rules are in most cases quite trivial. Starting from IUT specification, they consist in replacing API-level inputs and outputs by the corresponding TM_PDUs exchanged with the LT, i.e.:

- replacing API inputs by incoming TM_PDUs the LT needs to send out to cause generation of that inputs by the UT,
- replacing API outputs by outgoing TM_PDUs the LT will receive as a notification of that outputs.

For example, such rules can be applied to the SDL specifications of IUT and UT presented in Figure 14. The result is shown in Figure 18:

- the *conn_r* API request is replaced by the *conn_p* TM_PDU,
- the *estab_i* and *rel_i* indication events are replaced by the *estab_p* and *rel_p* TM_PDUs.

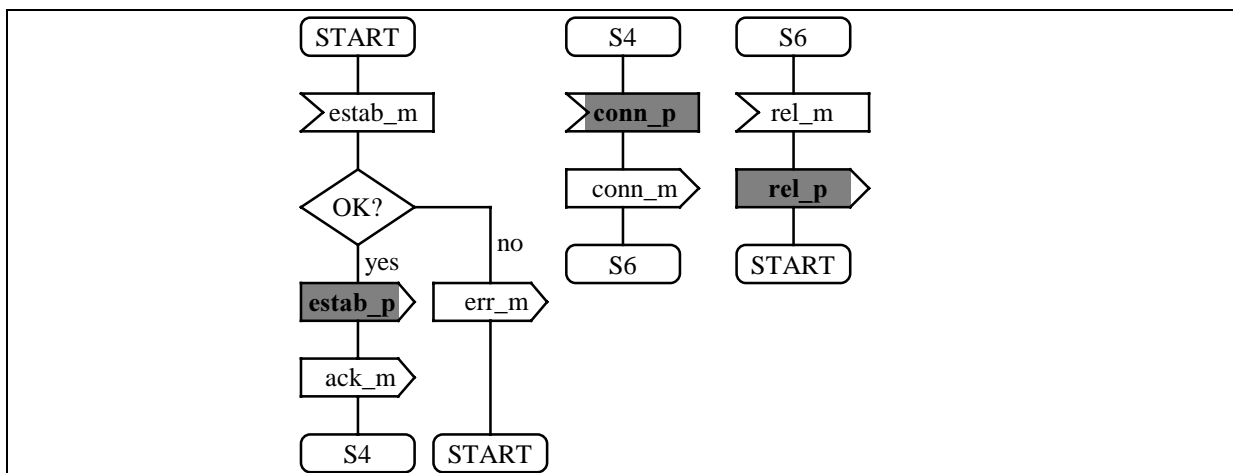
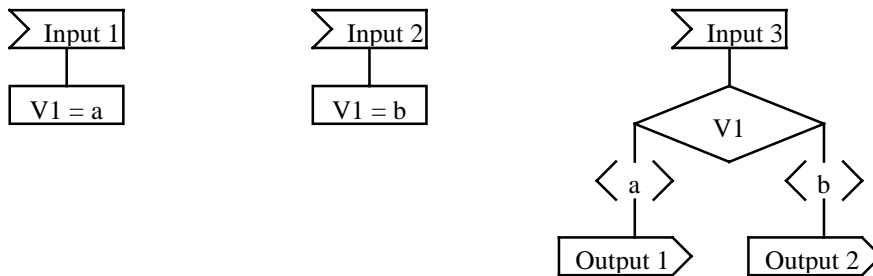


Figure 18 Combined specification of IUT and UT

4.2.3. Test suite example

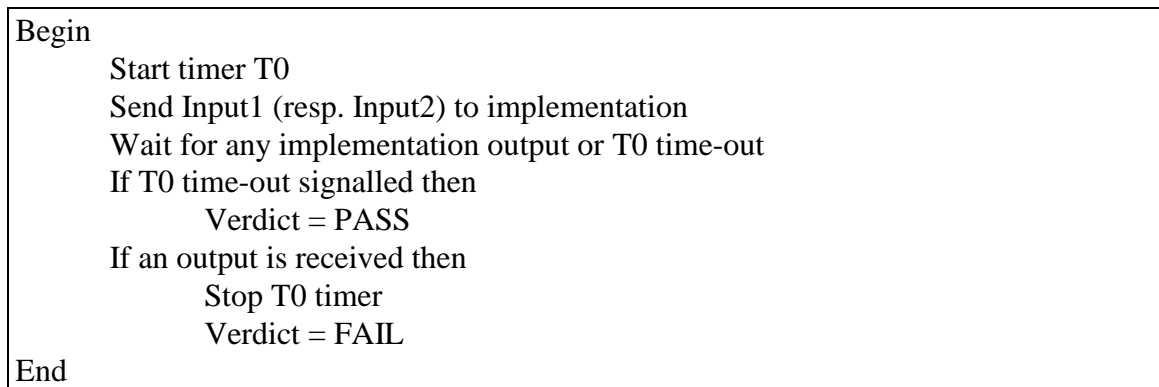
Let us consider the following specification of UT + IUT.



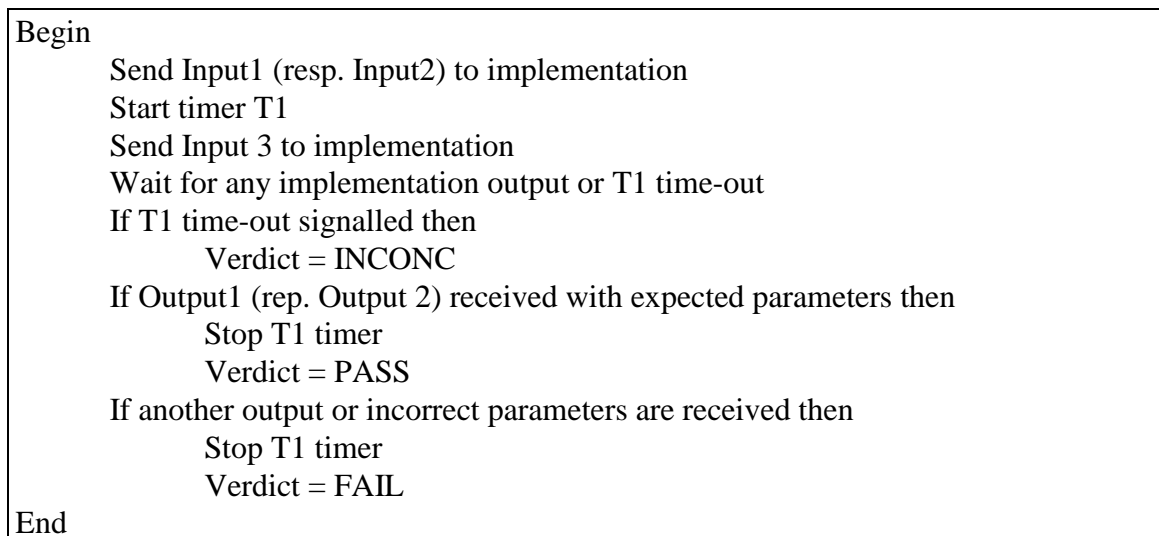
The list of test purposes can be defined as follows:

1. *"Send Input 1 and observe nothing"*
2. *"Send Input 2 and observe nothing"*
3. *"Send Input 3 when V1 = a and observe Output 1"*
4. *"Send Input 3 when V1 = b and observe Output 2"*

The test cases associated to purposes 1 and 2 are as follows:



The test cases associated to purposes 3 and 4 will look like:



In the above sequence, the first statement "Send Input1 (resp. Input2)" represents the preamble of the test case. It serves to place the implementation in the appropriate state allowing the test purpose to be observed. Furthermore these test cases allow to implicitly check that V1 was correctly initialised on Input1 or Input2. Nevertheless, the test cases for those inputs shall be kept, so that if an unspecified event is received, the tests can determine whether the source of error is Input1, Input2 or Input3.

This example also shows a general method for managing implementation's output events. A timer is started before sending the input that should normally cause the output. The tester then waits for that output up to the timer expiry. When timer expires, verdict is set to INCONC because in actual implementations, the tester cannot determine whether lack of reception is due non conformance or to transfer problems (e.g. network failure) between the implementation and the tester.

As a more complete example, let us consider again the SDL specification of connection handling protocol shown in Figure 14. A test purpose can be defined as follows:

"when conn_p is sent to IUT in state S4, a conn_m PDU will eventually be answered"

This test purpose can be formalised by the following automaton where ! and ? are standard notations representing an output and an input respectively.

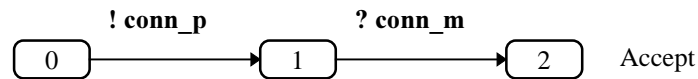


Figure 19 details the abstract test case using the same notations. The test case contains a preamble that brings the specification in a state in which *conn_p* can be received. After the test case has succeeded, a postamble brings the specification in its initial state (provided it can be reached at all). Unexpected receptions yield FAIL verdicts while time-outs lead to INCONC. The tree subdivision at S1 node aims to enable reception of *estab_p* and *ack_m* in any order.

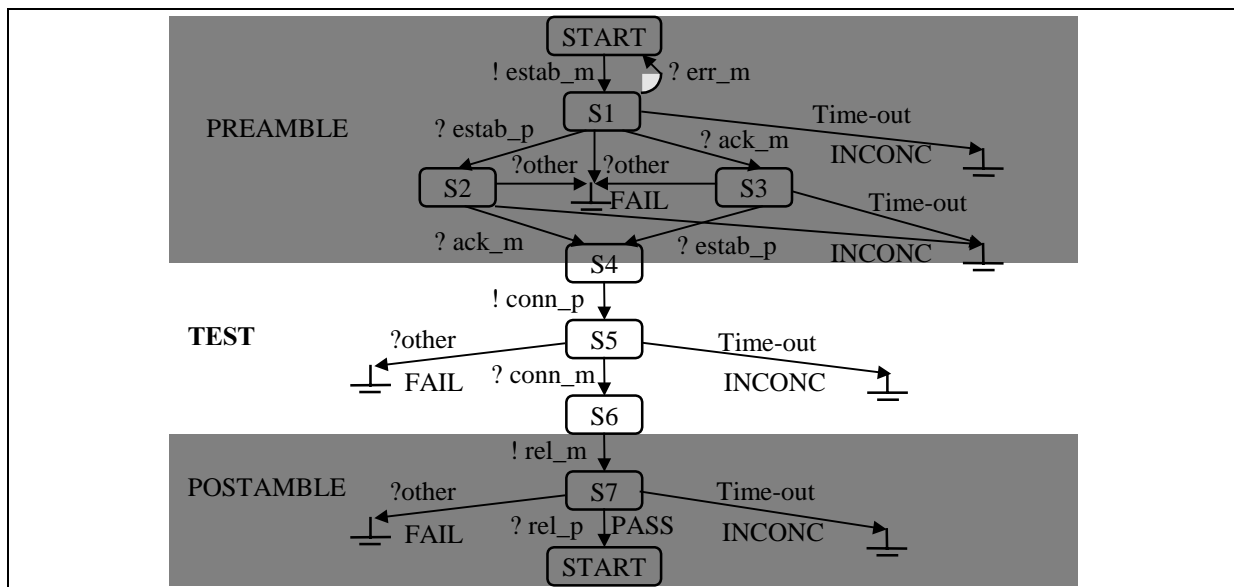


Figure 19 Test case for connection handling protocol

4.2.4. Test generation tools

The generation of test suites from the SDL specifications will be achieved with the help of a case tool. The same environment has been already employed to develop the SDL specifications of OSEK COM and NM. It incorporates a TTCN toolset allowing to develop TTCN test cases and comprising an editor, an analyser, a simulator and a code generator. This environment will be used to generate syntactically correct specifications of the COM and NM test suites.

The toolset also allows a combined simulation of TTCN tests and SDL specifications. In such simulations, the TTCN test cases are executed against the SDL specification which plays the role of the IUT. In the OSEK context, TTCN versus SDL simulations will have two objectives:

- to validate test cases,
- to verify the ratio of specification coverage by the test suite. This information is provided by the tool after each simulation. Traceability information is also supplied about the SDL branches covered by the tests. It will be exploited to eliminate redundancies and improve the completeness of the test suites.

As a conclusion, Figure 20 describes the complete test generation process that is anticipated to support the OSEK COM and NM conformance. TTCN test cases are first written and compiled to eliminate syntax and semantics errors (step 1). Then, they can be simulated against the SDL specification and therefore test suite conformance to the OSEK specification can be verified (step 2). As the TTCN specification only represents the Lower Tester part of the test architecture, the OSEK/SDL specification needs to be associated with the SDL of Upper Tester to simulate the complete test architecture.

The next step (3) is the implementation of conformance tests:

- The Upper Tester is developed from the SDL specification by the tool provider and delivered to the IUT implementator. The latter has in turn to adapt the software to the target environment.
- The Lower Tester is developed from the TTCN specification by the tool provider. The main part is automatically generated by the code generator provided by the case tool. Adaptation is also required to install the software in tester's target environment.

It should be pointed out that the development tools mentioned above will only be used for the specification and generation of the test suites. They will not be needed for conformance tests execution. In other words, it will not be necessary for OSEK implementators to acquire any of the case tools to do conformance testing.

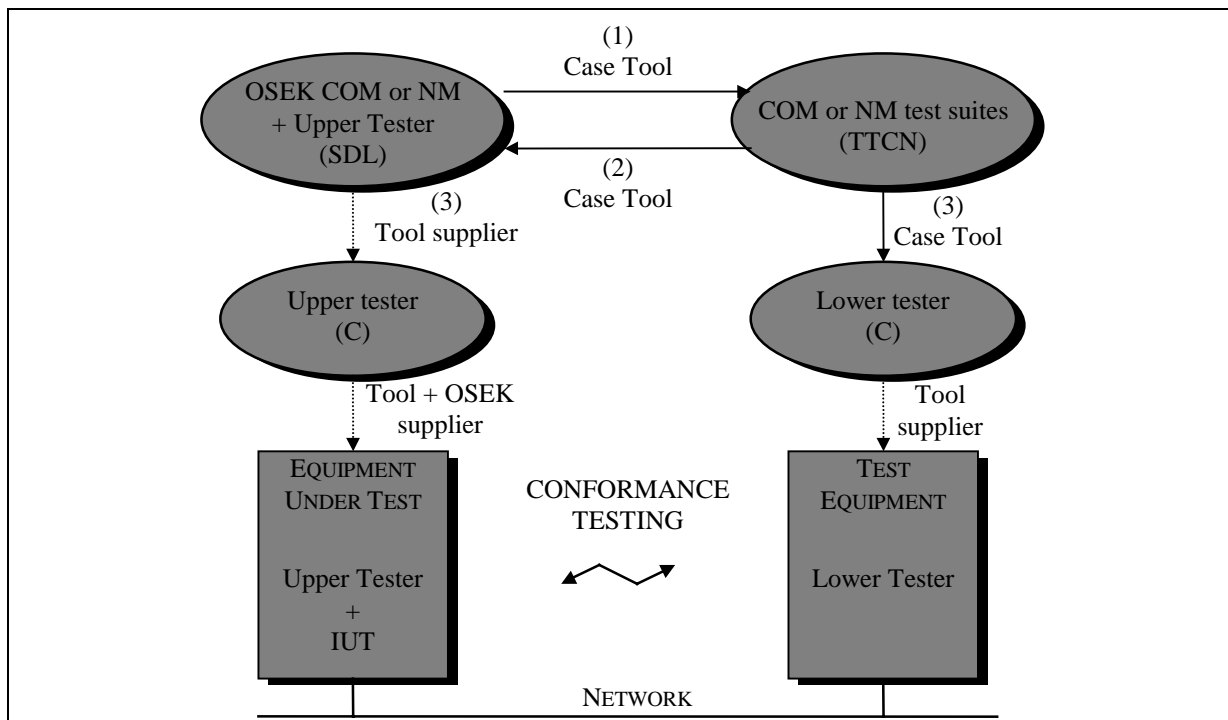


Figure 20 Test suites generation and implementation process

5. TTCN overview

The three OSEK test suites for OS, COM and NM will be specified in TTCN language. TTCN [6] is the standardised test notation for the description of OSI conformance tests. It combines a tree notation for dynamic test behaviour description with a tabular representation of the language constructs. TTCN has two representations: a graphical form suitable for human readability and a textual form for automatic processing of test suites. Both forms are strictly equivalent.

TTCN test suites are structured in several parts: declarations, constraints and dynamic behaviour. The following sections aim at providing an overall description of the language. A complete definition can be found in document [6]. The structure of TTCN test suites is detailed throughout examples. For protocols, they are drawn from the connection handling scenario presented before.

5.1. Declarations

The declaration part begins with the description of a test architecture which is formalised by the definition of Points of Control and Observation (PCO). The PCOs specify the conformance tests' access points to the IUT:

- in OS conformance, the OS-API will be the only PCO,
- in COM and NM conformance, the TTCN test cases will specify the exchange of PDUs between target equipment and tester according to the coordinated test architecture principle detailed before. The test architecture comprises two PCOs placed at the lower interface of the IUT:
 - one for OSEK frame exchanges between LT and IUT,
 - one for TM_PDU exchanges between LT and UT.

The PCO definition includes the declaration of a name, a type and a role which can be either UT for Upper Tester or LT for Lower Tester.

PCO Declarations			
PCO Name	PCO Type	Role	Comments
TMP	DataBus_PCO	LT	TMP access point
Detailed Comments :			

Table 22 PCO declaration

The declaration part also includes data-type and operation declarations which can be external (any implementation dependent language) or internal. Internal declarations are specified in a TTCN specific syntax or in ASN.1, the ISO standard of data presentation. Data types allow to describe the format of data exchanged by the tester and the target equipment. Within OSEK conformance, data types will define:

- The formats of APIs controlled by the OS tester. API procedures are called ASP (Abstract Syntax Primitives) in TTCN notation.
- The formats of PDUs exchanged by the COM and NM testers, including OPDUs and TM_PDUs,

ASP Type Definition		
ASP Name	: ActivateTask	
PCO Type	: API_PCO	
Comments	: task activation procedure	
Parameter Name	Parameter Type	Comments
TaskID	TaskType	Task reference
Detailed Comments	:	

PDU Type Definition		
PDU Name	: estab_m	
PCO Type	: DataBus_PCO	
Comments	: connection establishment request	
Field Name	Field Type	Comments
ConNum	ConNumType	Connection Number
PCI	PCI_Type	Protocol Control Information
Detailed Comments	:	

Table 23 ASP and PDU type declarations

The declaration part of a TTCN test suite also aims at defining internal data of the tester:

- Internal variables enable the specification of complex tests, e.g. sequence number for identifying PDUs of segmented data transfer. They can be declared with a test case limited or global scope.
- Test suite parameters allow to specify IUT's parameters that will be used to select the applicable test cases or to enable the communication between tester and IUT, such as addressing information. Through parameterization, test suites can be made generic or general to several implementations of a family of protocols. Parameters are usually defined in associated parameter files.
- Timers allow to set time-outs for IUT answers to tester's stimuli. Very much like parameters, the duration of these timers can be defined in external parameter files.

Timer Declarations			
Timer Name	Duration	Unit	Comments
ack_t	10	sec	Wait for PDU acknowledgement
Detailed Comments	:		

Table 24 Timer declaration

5.2. Constraints

TTCN constraint declarations specify the values of ASP parameters and PDU fields used by the tester in send or receive operations:

- In send operations, they define the actual values assigned to ASP parameters or to PDU fields.
- In receive operations, they define the values to be matched by the fields of received PDUs or by the parameters of received ASPs.

Constraints can be either declared in constraint tables or directly specified inside dynamic behaviour descriptions.

ASP Constraint Declaration		
Constraint Name : Activate_10		
PDU Type : ActivateTask		
Derivation Path :		
Comments : activate task number 10		
Parameter Name	Parameter Value	Comments
TaskID	10	Task reference
Detailed Comments :		

PDU Constraint Declaration		
Constraint Name : estab_m0		
PDU Type : estab_m		
Derivation Path :		
Comments : establishment request for connection number 10		
Field Name	Field Value	Comments
ConNum	10	Connection Number
PCI	EstabPCIValue	Protocol Control Information
Detailed Comments :		

Table 25 ASP and PDU constraint declarations

5.3. Dynamic behaviour

Dynamic behaviours of the tester are specified within test cases and test steps. They consist in a tree-like structure, describing sets of sequences of interactions with the IUT or internal events (timers related events). Interactions with the IUT are either an ASP send or an ASP receive in OS conformance, or either a PDU send or a PDU receive in COM and NM conformance. Timers can be set and reset. Diagnostics can be produced (FAIL, PASS or INCONC) at any place in test cases. Complex behaviours can be expressed with the help of usual control structures : conditional, loop, alternative choice and test step (procedure) call.

The table below provides the TTCN translation of the test case specified in Figure 19.

The hierarchy of tests is represented by the successive indentation levels of the behaviour description. Statements at the same level like $L? estab_p$, $L? ack_m$, ... represent the possible choices a this level. In $L? estab_p$

- L specifies the PCO at which the event will occur. It can be omitted if there is only one PCO.
- $?$ stands for a tester input. Conversely $!$ indicates an output.
- $estab_p$ is the data type identifier of the input/output. It must have been previously defined in a data type declaration.

A time-out is associated to each expected input of the tester, such as ack_t for ack_m PDU. The START, CLEAR and TIMEOUT statements are used to manage timers.

The constraint column specifies the actual values of inputs/outputs which must have been previously defined in a TTCN constraint table.

The possible verdicts are shown in the verdict column. The parentheses of (PASS) stand for a temporary verdict.

Test Case Dynamic Behaviour					
Test Case Name :					
Group :					
Purpose :					
Default :					
Comments :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		L! estab_m, START err_t, START ack_t, START estab_t	estab_m0		
2		L? estab_p, CLEAR estab_t, CLEAR err_t	estab_i1		
3		L? ack_m, CLEAR ack_t			
4	L1	L! conn_p, START conn_t	conn_r2		
5		L? conn_m, CLEAR conn_t	conn_m3	(PASS)	
6		L! rel_m, START rel_t	rel_m4		
7		L? rel_p, CLEAR rel_t	rel_i5	PASS	
8		? TIMEOUT rel_t		FAIL	
9		? TIMEOUT conn_t		FAIL	
10		? TIMEOUT ack_t		FAIL	
11		L? ack_m, CLEAR ack_t, CLEAR err_t			
12		L? estab_p, CLEAR estab_t	estab_i1		
13		GOTO L1			
14		? TIMEOUT estab_t		FAIL	
15		L? err_m, CLEAR estab_t, CLEAR ack_t, CLEAR err_t		INCONC	
16		? TIMEOUT estab_t		FAIL	
17		? TIMEOUT ack_t		FAIL	
18		? TIMEOUT err_t		FAIL	

Table 26 Test case dynamic behaviour

6. Abbreviations

API	Application Programming Interface
COM	Communication
DLL	Data Link Layer
ECU	Electronic Control Unit
IL	Interaction Layer
ISO	International Standard Organization
ISR	Interrupt Service Routine
IUT	Implementation Under Test
LT	Lower Tester
NM	Network Management
OPDU	OSEK Protocol Data Unit
OS	Operating System
PDU	Protocol Data Unit
PCO	Point of Control and Observation
SDL	Specification and Description Language
TMP	Test Management Protocol
TM_PDU	Test Management - Protocol Data Unit
TTCN	Tree and Tabular Combined Notation
UT	Upper Tester

7. References

- [1] OSEK/VDX Certification Procedure - F. Kaag, J. Minuth, K.J. Neumann, H. Kuder - Proceedings of the 1st International Workshop on Open Systems in Automotive Networks - October 1995.
- [2] OSEK/VDX Operating System - Version 2.0 Revision 1 - 15 October 1997
- [3] OSEK/VDX Communication - Version 2.1 - revision 1- 17th June 1998.
- [4] OSEK Network Management - Concept and Application Programming Interface- Version 2.50 - 31th of May 1998.
- [5] ISO/IEC 9646-1 - Information technology, Open Systems Interconnection, Conformance testing methodology and framework, *part 1 : General Concepts*, 1992.
- [6] ISO/IEC 9646-3 - Information technology, Open Systems Interconnection, Conformance testing, methodology and framework, *part 3 : The Tree and Tabular Combined Notation (TTCN)*, 1992.
- [7] OSEK/VDX - Overall Glossary - 23 September 1997
- [8] OSEK/VDX - System Generation - OIL: OSEK Implementation Language - Version 2.0 - 16 December 1997