OSEK/VDX

Conformance Testing Methodology

Version 1.0

December 19th, 1997

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, IIIT 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
Didier Stunault	Dassault Electronique
Stefan Schmerler	FZI
Franz Adis	FZI
Benoit Caillaud	INRIA
Yves Sorel	INRIA
Dirk John	IIIT, 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

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

0. Preface 5

1. Scope of Conformance Testing 6
   1.1. Motivations 6
   1.2. Definitions and intentions 6
   1.3. Conformance testing concerns 8
   1.4. Rules of test suites definition 8
   1.5. Other limitations 10
   1.6. Work programme 10

2. Definition of Test Suites 12
   2.1. Definition process 12
   2.2. OS test suites 15
      2.2.1. Test configurations 15
      2.2.2. Services and variants 15
      2.2.3. Definition method for test suites 17
   2.3. COM and NM test suites 18
      2.3.1. Test configurations 18
      2.3.2. Services and variants 20
      2.3.3. Event management 25
      2.3.4. Definition method for test suites 30
   2.4. Test suites for complete OSEK implementation 31
      2.4.1. Test configurations 31
      2.4.2. Services and variants 32

3. Test architecture for COM and NM 33
   3.1. Description of the test architecture 33
   3.2. Rules of UT, LT and TMP specification 34
   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 48

5. TTCN overview 52
   5.1. Declarations 52
   5.2. Constraints 53
   5.3. Dynamic behaviour 55

6. Abbreviations 57

7. References 58

0. Preface

This document defines a Conformance Testing Methodology that is going to be applied in Modistarc project. It is a draft because of unstable state of OSEK Communication and Network Management specifications. However, the specification evolution should not affect the methodology concepts but only the conformance tests resulting from application of this methodology to OSEK specifications.

This document will also evolve throughout the Modistarc project according to outcomes of the conformance development process. Modistarc activities which will bring inputs to this document are the definition of Conformance Test Suites and the Test Campaign.

At the end of Modistarc, this document will be the reference document for OSEK Conformance testing methodology.

A glossary of terms used in this document will be provided in the next version of the OSEK Overall Glossary [7].

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 in principle the APIs and OPDUs defined in the specification. APIs may comprise user's APIs and inter-module APIs corresponding to interfaces between OSEK modules. OSEK module access points are presented in Figure 1 below. Some or all of them may be present depending on the type of module (OS, COM, NM).

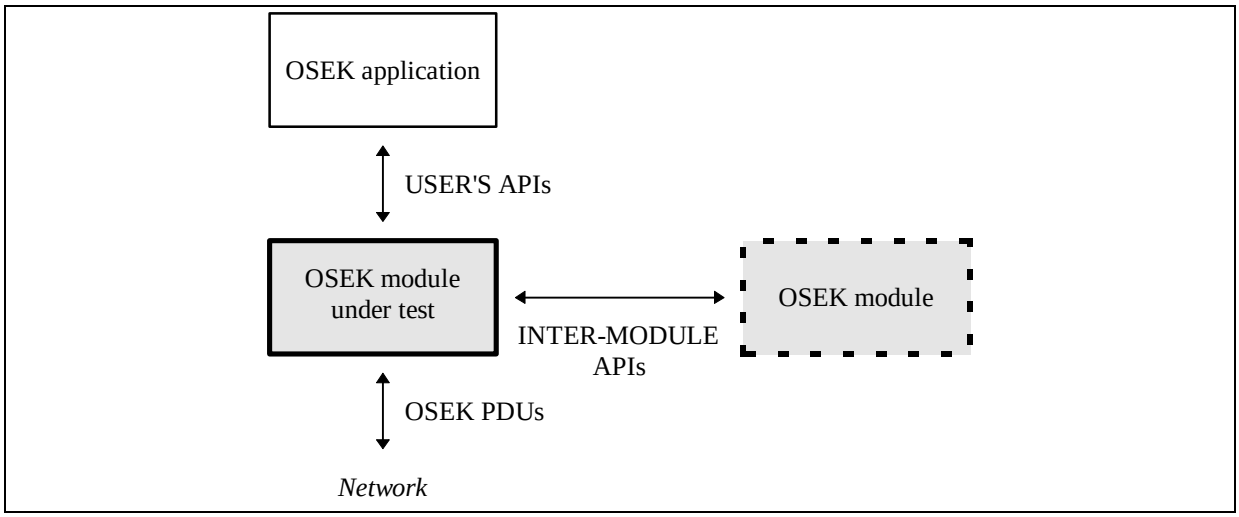


Figure 1 OSEK module interfaces

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. Wherever possible, the test suites shall address the latter. This implies that OSEK APIs and services 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 an ECU,
- interconnection of ECUs to validate the system.

In the first phase, the purpose is to test an isolated OS, COM or NM component, whose all interfaces are accessible.

In the second phase, the purpose is to test an assembly of the three OSEK components. Only the external interfaces of the assembly are visible. For example, the interfaces between NM and the link layer of COM are no longer accessible. The purpose, in this phase, is to check OSEK compliance when all software products are integrated in the target hardware environment. The concerned tests are sub-assemblies or adaptations of isolated component tests.

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.

As a summary, two types of configurations will be taken into consideration by the conformance tests:

- test of an isolated OSEK component for the purpose of certifying software before integration to an ECU,
- test of a complete OSEK assembly, for the purpose of certifying a software in its operational environment.

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 application or inter-module procedural calls. 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, state tables 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:

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

Capacity and performance parameters will not be verified by conformance tests. However, a list of meaningful parameters will be given for reference. Decision to do measurements and supply the obtained values to customers is left to the **implementor**s.

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. The **implementor** shall validate his 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.

However, the data provided by the tested implementation must be significant enough for the test to be pronounced passed. In some cases, it is necessary to add monitoring services or to allow access to internal interfaces defined in the specification, such as the interfaces between communications layers. As a consequence, such so-called testability services shall be offered by the candidate implementations for the tests to be run. They will be defined in the following.

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. ECUs will be particularly employed to validate the approach to what is called above the "OSEK assembly conformance".

- 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 all available 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 a 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 user's implementation will automatically check syntax compatibility between OSEK API and implemented 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 and of the test configuration (isolated or integrated module),
- 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 configurations: isolated module, assembly test,
- the applicable variants: OS 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 a number of initial states from which the preambles and postambles will be established. For communication, for example, the initial states could be:

- COM not initialised (connections not established) for the connection handling test cases,
- COM initialised (connections established) for the data transfer test cases.

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 the behaviour of the implementation still complies with the specification. This case occurs when several behaviours are possible owing to tester's impossibility to control some of the implementation events. For instance, a transport connection establishment can be

accepted or refused by the implementation according to the OSEK specification. The decision may depend on internal variables. If the request is refused, a test case trying to check the acceptance protocol will not reach its objectives and lead to an inconclusive verdict.

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 or INCONCLUSIVE verdict. However, inconclusive verdicts will be reported and stressed. Tests may be passed again to analyse such contingencies and try to either fix the problem or give a justification for the 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.

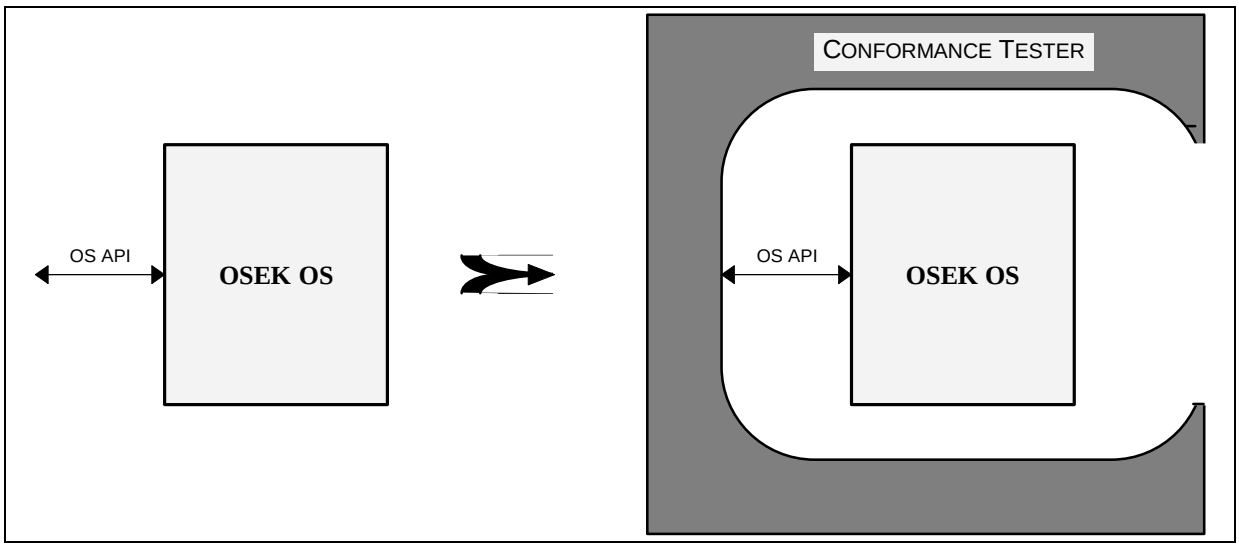


Figure 2 Conformance testing configuration for OS

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	
– Transfer task into *ready* state	*ActivateTask*
– Terminate calling task	*TerminateTask*
– Terminate calling task and activate succeeding task	*ChainTask*
– Call scheduler	*Schedule*
– Get currently active task	*GetTaskId*
– Get state of a task	*GetTaskState*
Interrupt handling services	
– Enter interrupt service routine (ISR)	*EnterISR*
- Leave interrupt service routine (ISR)	*LeaveISR*
- Enable interrupts	*EnableInterrupt*
- Disable interrupts	*DisableInterrupt*
– Query interrupt status	*GetInterruptDescriptor*
Resource management services	
– Get resource and enter critical section	*GetResource*
– Release resource and leave critical section	*ReleaseResource*
Event control services	
– Set event of extended task	*SetEvent*
– Clear Event	*ClearEvent*
– Get event mask of a task	*GetEvent*
– Wait for setting of an event	*WaitEvent*
Alarms services	
– Read alarm base characteristics	*GetAlarmBase*
– Occupy and set relative alarm	*SetRelAlarm*
– Occupy and set absolute alarm	*SetAbsAlarm*
– Cancel alarm	*CancelAlarm*
– Get alarm value	*GetAlarm*
Operating system execution control services	
– Get current application mode	*GetActiveApplicationMode*
– Start operating system	*StartOS*
– Shut down operating system	*ShutdownOS*
Hook routines	
– Called if OS service returns an error	*ErrorHook*
– Called at task switch before entering context of new task	*PreTaskHook*
– Called at task switch after leaving context of old task	*PostTaskHook*
– Called after start-up	*StartupHook*
– Called before shutdown	*ShutdownHook*

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 description and specification of the test suites the following means will be used:

- 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

CAUTION!

All what concerns COM in this section refers to COM specification 2.0 Draft 1.5 [3]. The contents will be updated according to the future COM specification 2.1 and its inevitable impacts to the NM specification [4].

Nevertheless, this section aims at presenting a generic methodology which should not be influenced too much by the new specifications. In principle, modifications to this text should only affect the OSEK interface descriptions and the lists of associated test events.

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 Transport API at the interface between Interaction Layer and Transport Layer,
- the DLL/COM API at the interface between Transport Layer and Data Link Layer or between Interaction Layer and Data Link Layer in implementations without the optional transport 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, a selection should be made among the available COM interfaces to keep only those that comply with the objectives of portability and interoperability. The selection leads to consider as relevant the interfaces presented in the right part of Figure 3, namely:

- the COM API, mandatory to enable application portability,
- the DLL/NM API, mandatory to enable NM module portability,
- the OSEK PDUs, mandatory to enable network interoperability.

Note that links between COM Transport and NM have now disappeared from the specification. Transport errors are processed by internal procedures of the Interaction Layer and the T-Error service will not be validated by the conformance tests.

The OS API has been removed to leave users the possibility to check conformance of COM modules that adapt to non OSEK operating systems. In that case, the objectives are focused on network service portability and interoperability, and the maxim "All what behaves like OSEK is OSEK" prevails over full portability including OS. Moreover, the COM specification provides with a list of the required OS services, but it does not specify when and how those services are

to be used. For example to start a protocol timer, a *StartTimer()* procedure is called rather than the equivalent *SetRelAlarm()* of the OS specification. In some other cases, it is impossible to establish a clear correlation between protocol functions and OS services.

The Transport and DLL/COM APIs correspond to internal services. They are hidden from applications and it is not necessary to implement them as specified. Even, for code optimisation and efficiency reasons it is likely that most COM modules will not implement a clear-cut separation between COM layers. Therefore, conformance to Transport and DLL/COM APIs will not be checked.

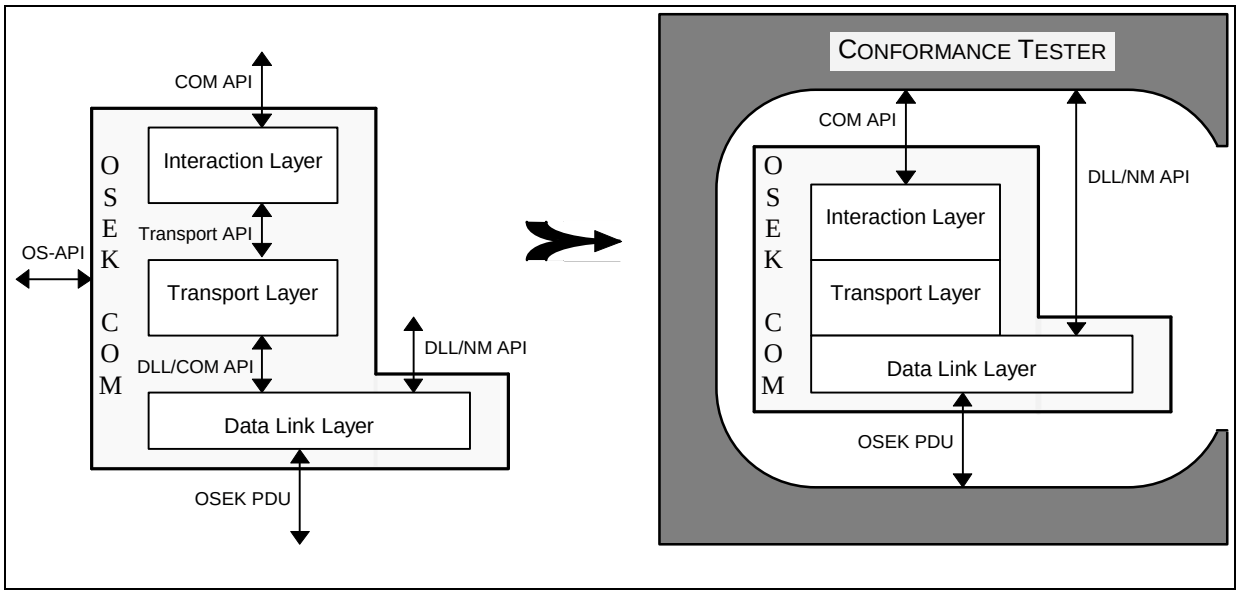


Figure 3 Conformance testing configuration for COM

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 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.

Finally, the same justification as in COM can be put forward to exclude the interface with OS 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.

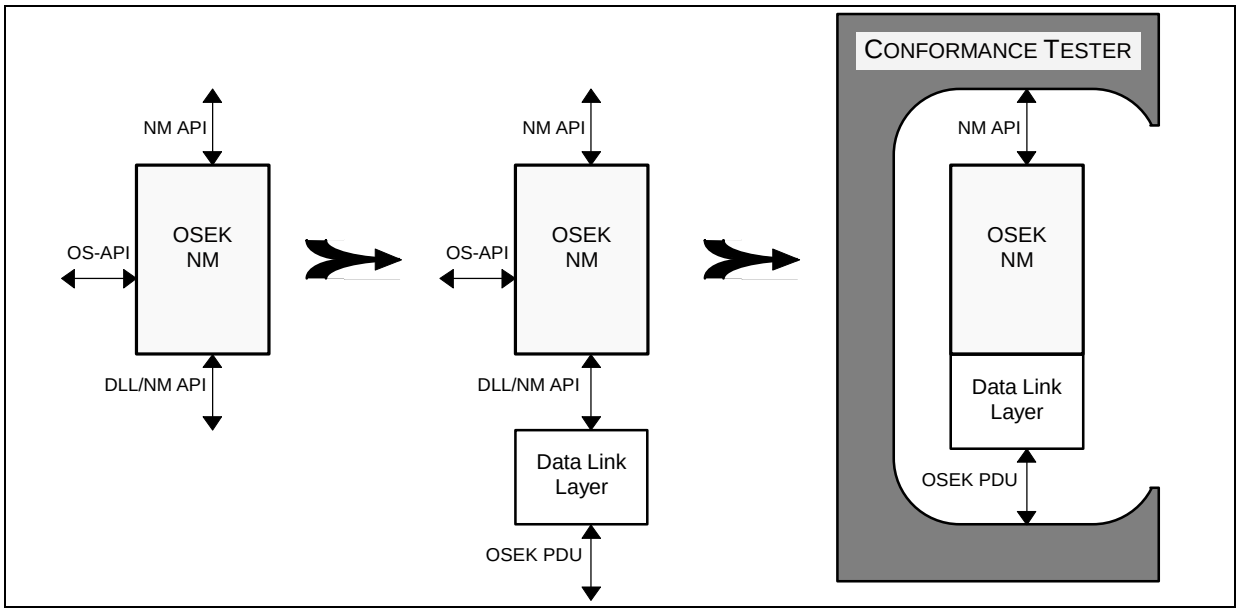


Figure 4 Conformance testing configuration for NM

2.3.2. Services and variants

COM-API services

The COM API consists of the four services presented below. 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, the test application software will encompass system objects that will be compiled to generate executable tests and will be run during the test campaign.

COM-API Services	Service Call
COM start-up - Start of COM module	 *StartCOM*
Data transfer - Update and send message object - Receive message object - Get status of last message transmission/reception	 *SendMessage* *ReceiveMessage* *GetMessageStatus*

Table 4 COM-API services

All services shall be implemented.

The COM variants shall meet the conformance classes defined in the specification and characterised by the following table. Parameters that decide of the conformance class are:

- the maximum size of message reception FIFOs and
- the presence or absence of the Transport Layer.

COM conformance classes	Max. FIFO size	Transport layer
CCC0	0[(1)], 1	no
CCC1	0[(1)], 1	yes
CCC2	≥ 1	yes

1) Message object is overwritten each time a new message is received

Table 5 COM conformance classes

Actually, the four COM services hide a lot of different options. Messages can be either state messages (not buffered) or event messages (buffered). Reception can be signalled through task activation or event setting. An alarm may be sent to warn of non reception of a periodic message.

Notifications of message reception or non 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
- Indication of message reception	*Message_ind*
- Indication of no message reception	*MessageNotReceived_ind*

Table 6 COM indication services

Messages can be transferred between local tasks or through the network. In case of network-wide transmission, the *SendMessage* call implies the utilisation of one in the four COM transport protocols, AUDT, ASDT, UUDT, USDT according to the transport connection characteristics. UUDT and USDT connections are point-to-point or multipoint and pre-established before start-up. AUDT and ASDT connections are point-to-point and dynamically established either at start-up by the *StartCOM* service or upon the first call to *SendMessage.*

It should be noted that UUDT connections are equivalent to Data Link connections. The CCC0 class means that all connections are of UUDT type.

All the options 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 connections 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 are as follows:

- state or event message,
- size of message and of reception FIFO,
- mode of reception signalling (task activation, event setting),
- type of message (state, event),
- local or network transfer,

- connection parameters (transport protocol, topology),
- message/connection association,
- connection establishment at start-up or before first message transmission.

Remark:
CCC1 and CCC2 require the implementation of all the four data transfer services, although they are independent of each other and could be implemented or not. Variants implementing part of these services will not be accepted. An OSEK compliant implementation must be strictly compatible with the COM conformance classes.

NM-API services

The NM specification defines two main variants that will determine the selection of test suites 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			
- Make current configuration available	*GetConfig*	X	
- Comparison of two configurations	*CmpConfig*		X
Operating mode management			
- Start of NM, i.e. transition to NM mode 'NMon'.	*StartNM*	X	
- Stop of NM, i.e. transition to NM mode 'NM Shutdown' and finally to 'NMoff'	*StopNM*	X	
- Transition to NM mode 'NMpassive' without network-wide notification	*SilentNM*		X
- Transition to NM mode 'NMactive' after a previous call of SilentNM	*TalkNM*		X
- Transition to a new operating mode (e.g. BusSleep, Awake)	*GotoMode*		X
- Get status information (network, node)	*GetStatus*		X
Data Field management			
- Transmit data	*TransmitRingData*		X
- Read received data	*ReadRingData*		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
InitIndDeltaConfig	- Indication of configuration update	*Config_ind*
InitIndDeltaStatus	- Indication of status information update	*Status_ind*
InitIndGotoMode	- Indication of network operating mode change	*GotoMode_ind*
InitIndRingData	- Indication of ring data reception	*RingData_ind*

Table 8 Indication services of direct NM

Indirect NM services

The table of indirect management services is supplied below. There is no implementation variant. All services are mandatory.

Indirect Management Services	Service Call	Core
Operating mode management		
- Start of NM, i.e. transition to NM mode 'NMon'.	*StartNM*	X
- Stop of NM, i.e. transition to NM mode 'NM Shutdown' and finally to 'NMoff'	*StopNM*	X
- Get status information	*GetStatus*	X
- Indication of status information update	*Status_ind*	X
Configuration management		
- Make current configuration available	*GetConfig*	X
- Indication of configuration update	*Config_ind*	X

Table 9 Services of indirect NM

DLL/NM -API services

The required DLL/NM services of a COM module depend on the type of NM they are intended to. Three possible variants are authorised:

- no service,
- Direct NM variant,
- Indirect NM variant.

The services of the last two options are listed in Table 10 below.

DLL Services for NM	Service Call	Direct NM	Indirect NM
Data transfer			
- Request to send window data	*D_WindowData.req*	X	
- Indication of window data reception	*D_WindowData.ind*	X	
- Confirmation of window data transmission	*D_WindowData.con*	X	
- Indication of data reception	*D_UUData.ind*		X
- Confirmation of data transmission	*D_UUData.con*		X
Data link management			
- Initialisation of communication hardware	*D_Init*	X	X
- Blocking of the user communication	*D_Offline*	X	X
- Re-enabling of the user communication	*D_Online*	X	X
- Get data link status information	*D_GetLayerStatus*	X	X
- Error indication	*D_Error.ind*	X	X

Table 10 DLL/NM services for Direct and Indirect NM

The DLL is responsible for signalling errors occurring in the DLL or in the physical layer. Most of those errors are implementation dependent and not identified in the OSEK specification. Only two of them are processed by the NM protocol:

- "message not transmitted" carried out by the *D_WindowData.con* or *D_UUData.con* primitives,
- "bus off" transmitted by the *D_Error.ind* primitive.

These errors need to be simulated in order to verify the complete functionality of the NM protocol.

Remarks on DLL specification (document [3]):

- *The DLL specification defines a* D_GetConStatus *service which is not requested by the NM specification. This service has been removed from the table above.*
- *The DLL specification does not specify* D_UUData.ind *and* D_UUData.con *as part of the DLL/NM interface. It is on that point inconsistent with the indirect NM specification. In the COM specification, the D_UUData API is placed at the interface with the Transport Layer. It is therefore assumed that COM implementations of the Indirect NM variant will encompass mechanisms allowing to provide this API to both the Transport Layer and the Indirect NM.*
- *The C syntax does not authorise a dot (".") character in C labels. API names like D_UUData.ind,... D_Error.ind cannot be implemented in C ! The names must be changed.*

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 (appli,...)
network service	request	indication, confirmation	user (appli,...)
OPDUs	reception, acknowledgement (DLL)	transmission	network
Timer	expiry	start, stop	OS

Table 11 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. Execution of local services does not impact the operational behaviour of the COM/NM implementation. 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 for OPDU exchange at the DLL level:

- request for transmission: *D_WindowData.req,*
- indication of reception: *D_WindowData.ind,*
- confirmation of transmission (= acknowledgement received): *D_WindowData.con.*

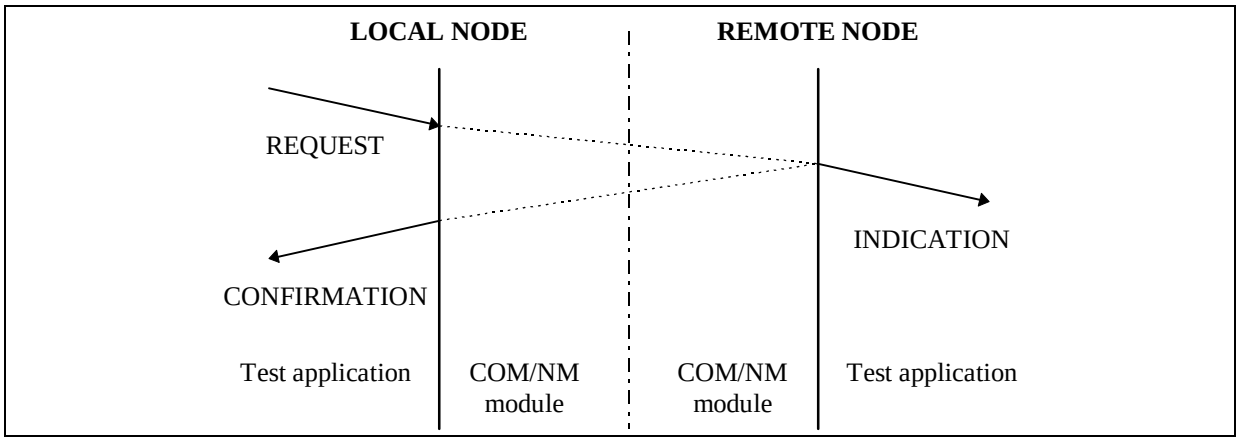


Figure 5 Service interaction model

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

To be completed when the COM specification is released.

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
ReceiveMessage *GetMessageStatus*	*StartCOM* *SendMessage*	*Message_ind* *MessageNotReceived_ind*	*none*

Table 12 COM-API events

The COM-OPDUs are described below as well as the DLL events managed by the COM transport protocol to transmit and receive OPDUs. A *none* in the "Acknowledgement" column means that the transport protocol automata of the OSEK specification do not deal with the D_UUData.con primitive. The associated treatments are considered as implementation dependent and therefore not taken into account by conformance tests (see also the remark below).

COM OPDUs	Definition
To be completed according to future COM specs	

DLL events of COM		
Transmission	Reception	Acknowledgement
D_UUData.req	*D_UUData.ind*	*none*

Table 13 COM OPDUs and associated DLL events

The protocol timers managed by the COM transport automata are described below. The related time-out events will be simulated in conformance tests.

Timers of COM transport	Definition
To be completed according to future COM specs	

Table 14 Time-out events of COM transport protocol

Remark:
According to the current specification, the conformance tests will not check protocol errors other than transport time-outs. Primitives like D_Error.ind (Link Layer) or T_Error.ind (Transport) are internally dealt with by the Transport Layer and the Interaction Layer respectively. Therefore they are not accessible to conformance tests.

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
Procedure call
GetConfig
CmpConfig
GetStatus
ReadRingData

Network events of direct NM API		
Request	Indication	Confirmation
StartNM	*Config_ind*	*none*
StopNM	*Status_ind*	
SilentNM		
TalkNM		
GotoMode	*GotoMode_ind*	
TransmitRingData	*RingData_ind*	

Table 15 Events of direct NM API

The direct NM specification defines seven OPDU types (also called NMPDUs) described below as well as the DLL events managed by the direct NM protocol to transmit and receive OPDUs.

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

DLL events of direct NM		
Transmission	Reception	Acknowledgement
D_WindowData.req	*D_WindowData.ind*	*D_WindowData.con*

Table 16 OPDUs of direct NM and associated DLL events

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 17 Time-out events of direct NM protocol

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.

Local services of indirect NM	Network services of indirect NM		
Procedure call	Request	Indication	Confirmation
GetStatus *GetConfig*	*StartNM* *StopNM*	*Config_ind* *Status_ind*	*none*

Table 18 Events of indirect NM API

The indirect NM specification does not use specific OPDUs. The principle of indirect NM is to monitor some of the OPDUs transmitted or received by the COM module. For this, it manages the following COM/DLL events:

DLL events of indirect NM		
Transmission	Reception	Acknowledgement
	D_UUData.ind	*D_UUData.con*

Table 19 DLL events of indirect NM

The indirect NM makes use of a periodic timer called Time-out for OBservation and described below.

Timer of indirect NM	Definition
TOB	Period of network status update

Table 20 Time-out event of indirect NM protocol

DLL/NM events

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

Two types of network faults need to be generated to check DLL's capability to notify the higher layers of these events:

- "transmission error" notified by *D_WindowData.con* or *D_UUData.con*,
- "bus off" transmitted by *D_Error.ind*.

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, it will not be possible to check the full DLL/NM services. However, to enable a meaningful NM conformance testing, network faults shall be at least simulated by adhoc software since a large part of the NM protocol is devoted to network failure recovery. This capability shall be offered by the DLL software connected to the NM module in the NM conformance testing configuration (see Figure 4).

Local events of DLL/NM	Network events of DLL/NM API		
Procedure call	Request	Indication	Confirmation
D_GetLayerStatus	*D_Init*	*D_Error.ind*	
	D_Offline		
	D_Online		
	D_WindowData.req	*D_WindowData.ind*	*D_WindowData.con*
	(*)	*D_UUData.ind*	*D_UUData.con*

Table 21 Events of DLL/NM API

(*) Access to *D_UUData.req* is not necessary. The COM messages monitored by Indirect NM are sent via the COM/*SendMessage* API. Therefore, the *SendMessage* call will also be used the same way by conformance tests to verify the occurrence of a *D_UUData.con* after a PDU transmission.

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. Test suites for complete OSEK implementation

2.4.1. Test configurations

In case of complete OSEK implementation, only the external interfaces of the three OSEK module assembly are accessible:

- the OS API,
- the COM API,
- the NM API,
- the OSEK PDUs.

The resulting test configuration is presented in Figure 6 below:

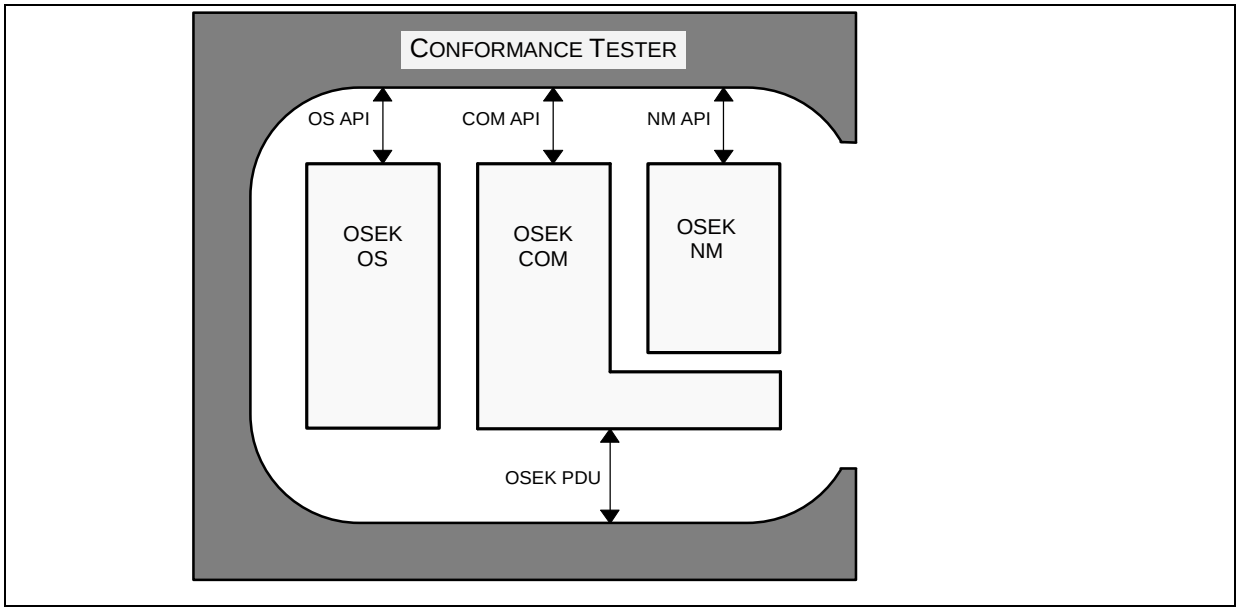


Figure 6 Conformance testing configuration for complete OSEK implementation

In some ECUs, COM and NM implementations can be associated to non OSEK OS and despite this, their OSEK conformance shall be evaluated. A specific test configuration without the OS API is defined to comply with this situation. The remaining interfaces available to the conformance tester are as follows:

- the COM API,
- the NM API,
- the OSEK PDUs.

The resulting test configuration is presented in Figure 7 below:

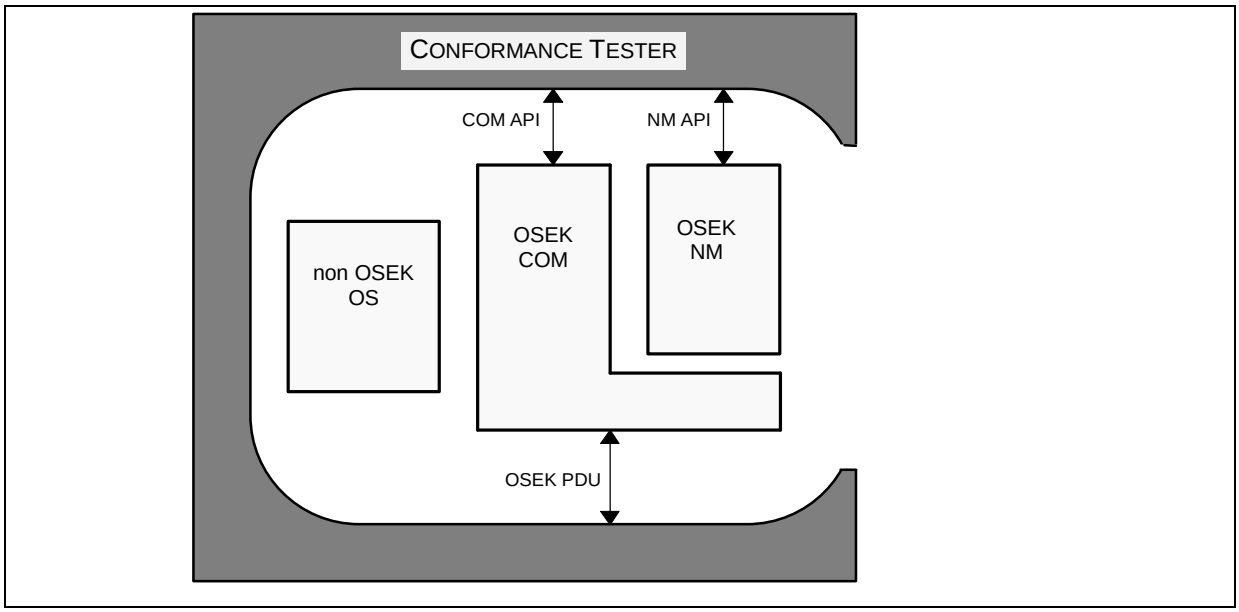


Figure 7 Conformance testing configuration for implementations without OSEK OS

2.4.2. Services and variants

The tested services and variants of a complete OSEK implementation are those defined in section 2.3.2 for each available API, i.e. OS API (for configuration with OSEK OS), COM API and NM API.

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.

To test OSEK protocols, the conformance tester needs to access two interfaces of the Implementation Under Test (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 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.

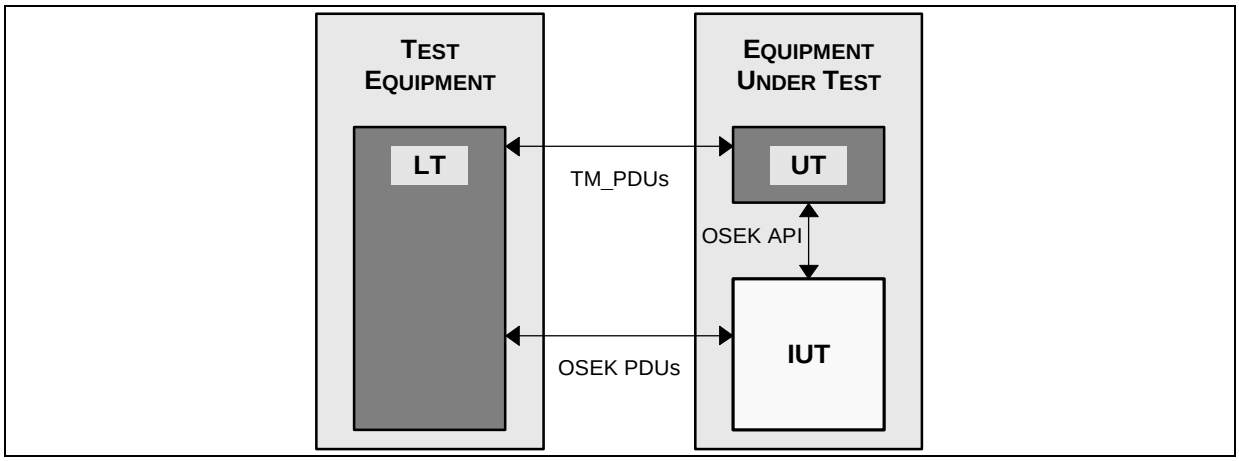


Figure 8 Principle of test architecture

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 **implementor** 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, NM 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, bus off) and its duration. This TM_PDU is not transmitted to the UT but interpreted by the DLL which will simulate the requested fault until the first DLL reinitialisation (*D_Init*) performed by the NM module after end of the fault period.

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.

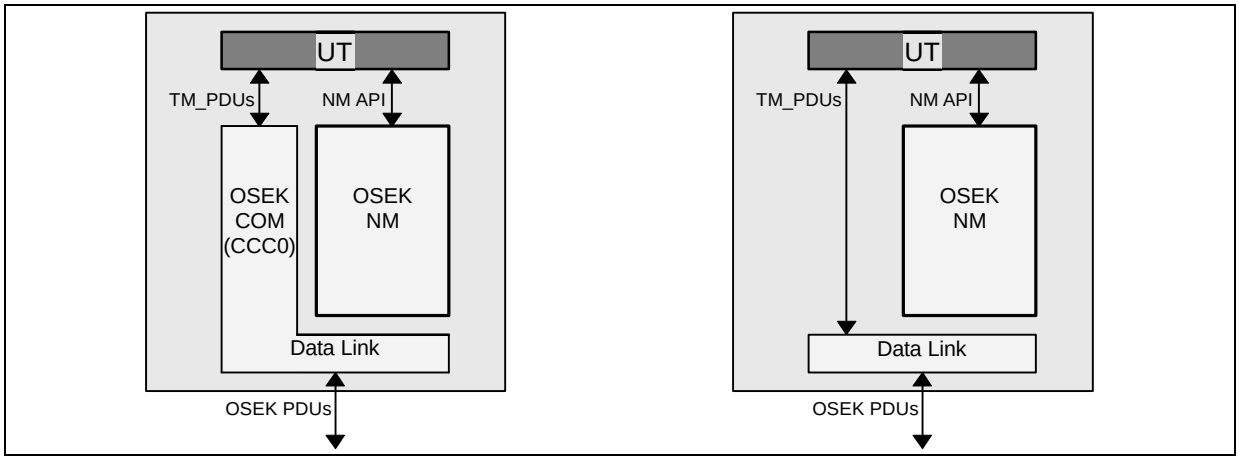


Figure 9 'Equipment under test' architectures for NM conformance

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			Remarks
	from	to	contents	
Request	LT	UT	API to be called and parameters	
	UT	LT	Status returned by API call	if error status
Indication, confirmation	UT	LT	Type of event and parameters	
Local service	LT	UT	API to be called and parameters	
	UT	LT	Status and data returned by API call	

Table 22 TMP and UT specification rules

3.3. TMP mechanisms

Figure 10 and Figure 11 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 Receive-Message. 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.

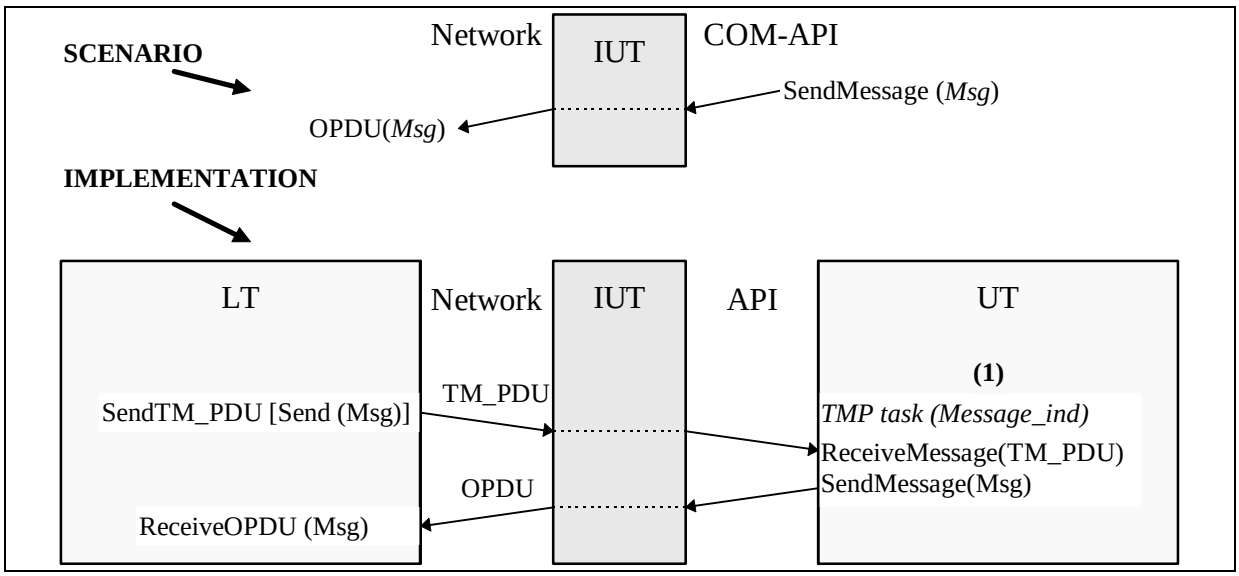


Figure 10 TMP mechanisms for message sending by UT

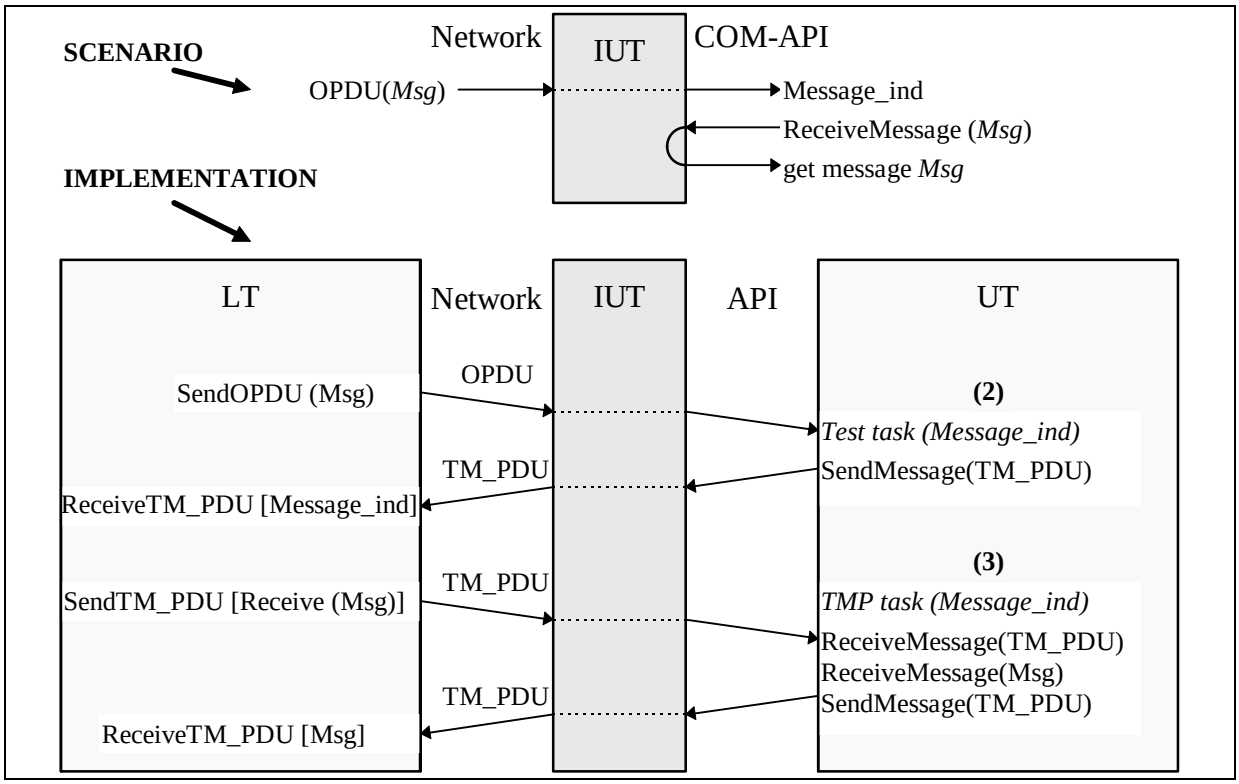


Figure 11 TMP mechanisms for message reception by UT

The message reception protocol can be simplified as presented in Figure 12 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.

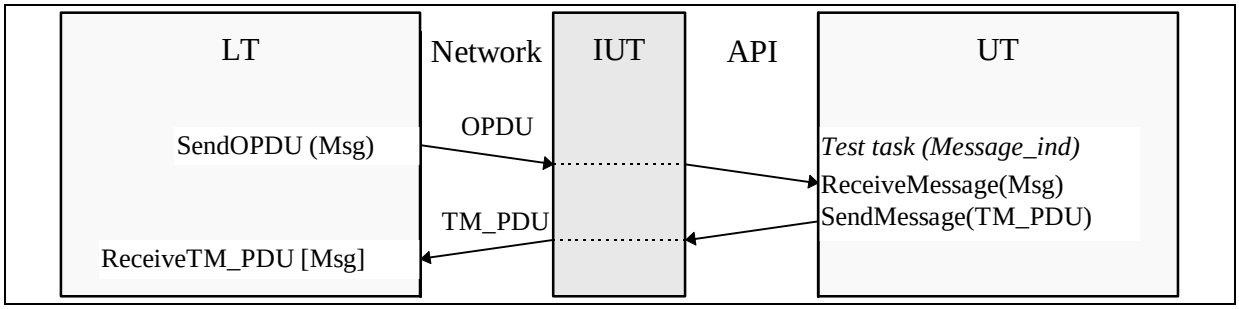


Figure 12 Simplified protocol for message reception by UT

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.

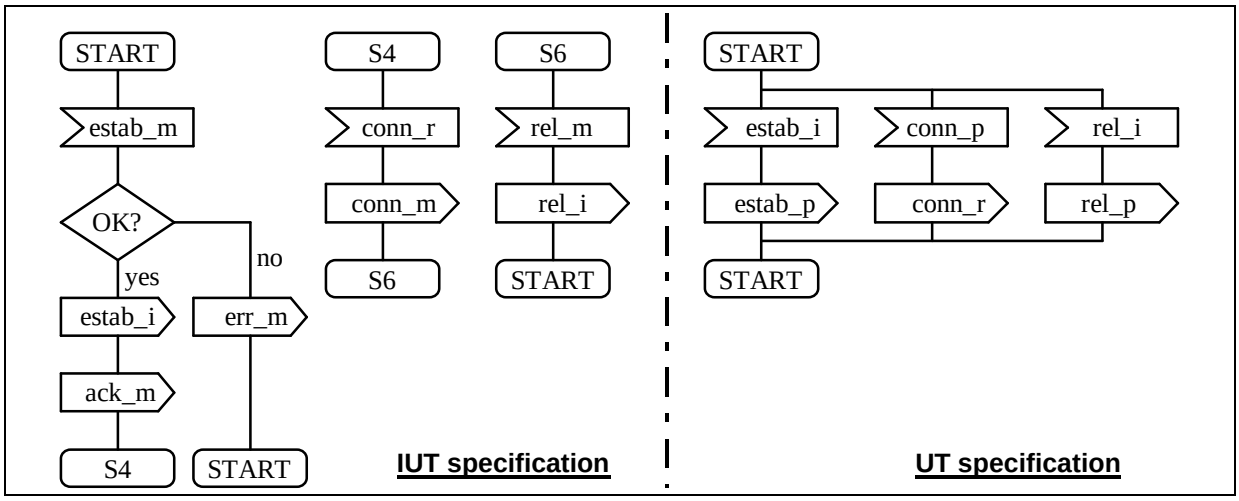


Figure 13 SDL specification of IUT and UT

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.

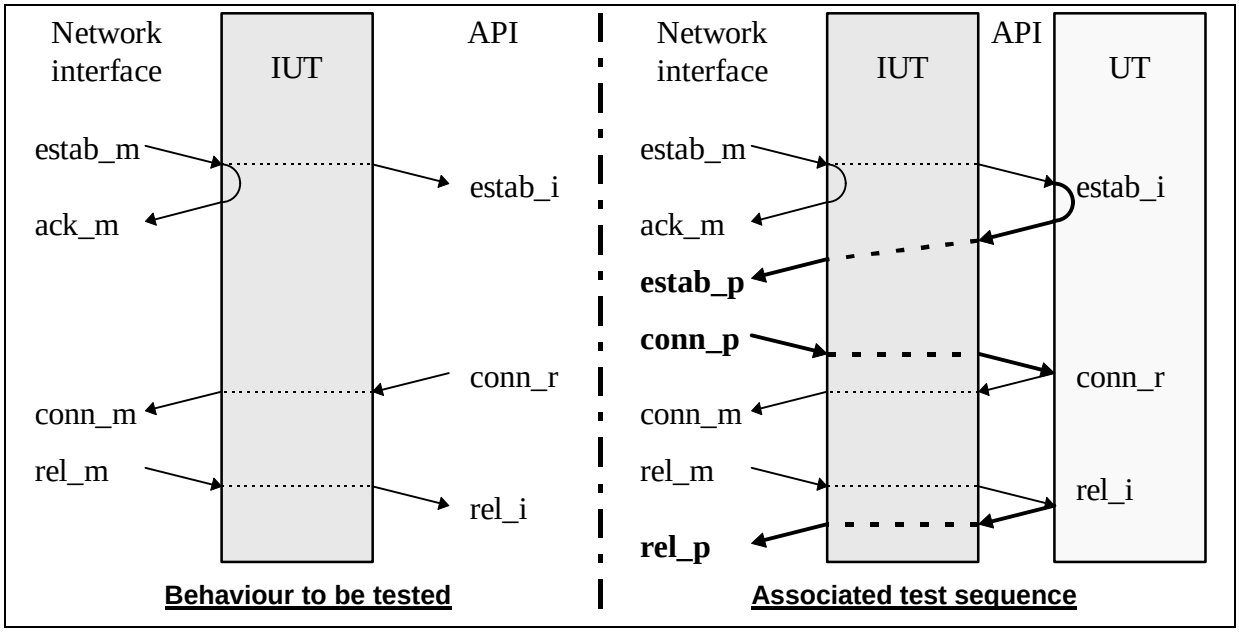


Figure 14 IUT scenario and associated test sequence

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 15 shows an example for the Classification-Tree of test group Resource Management. Table 23 shows the resulting test cases and their description.

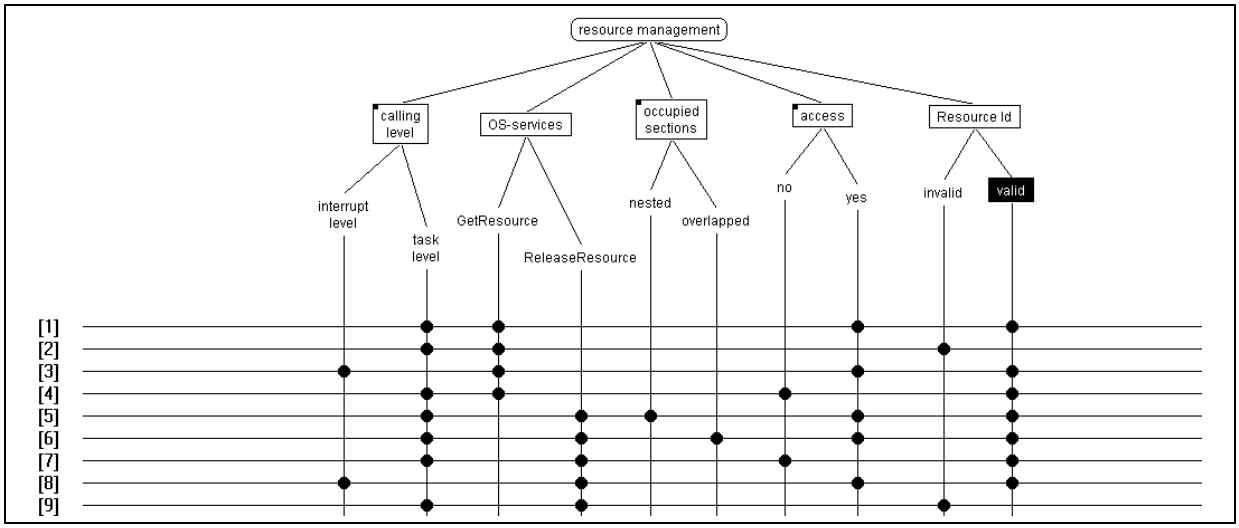


Figure 15 Classification-Tree of test group Resource Management

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 shoud 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 23 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 24. 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 case5**).
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 24 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 use a CASE-tool which is based on State- and Activity-Charts like for instance Statemate by i-Logix Inc to model the test suite. 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 24, will be modelled with 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 activity in an Activity-chart (Figure 16). In each step of the test suite one API service is called and its result is compared to the value specified for this situation. An API 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.

In addition there will be an activity which simulates the behaviour of the OSEK OS. This will not be a complete model of the OSEK OS, but it will produce the return and status values of the API services. This makes it possible to use intentionally a non-compliant OS and check the test suites reaction.

For automatic code generation during simulation of the test suite, user-defined code will be attached to the activities representing the OSEK API which is executed if the corresponding activity is activated, i. e. if the corresponding API service is called. This API service call will be written out together with a routine which will check the return value against its specified value. 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 implementor 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.**

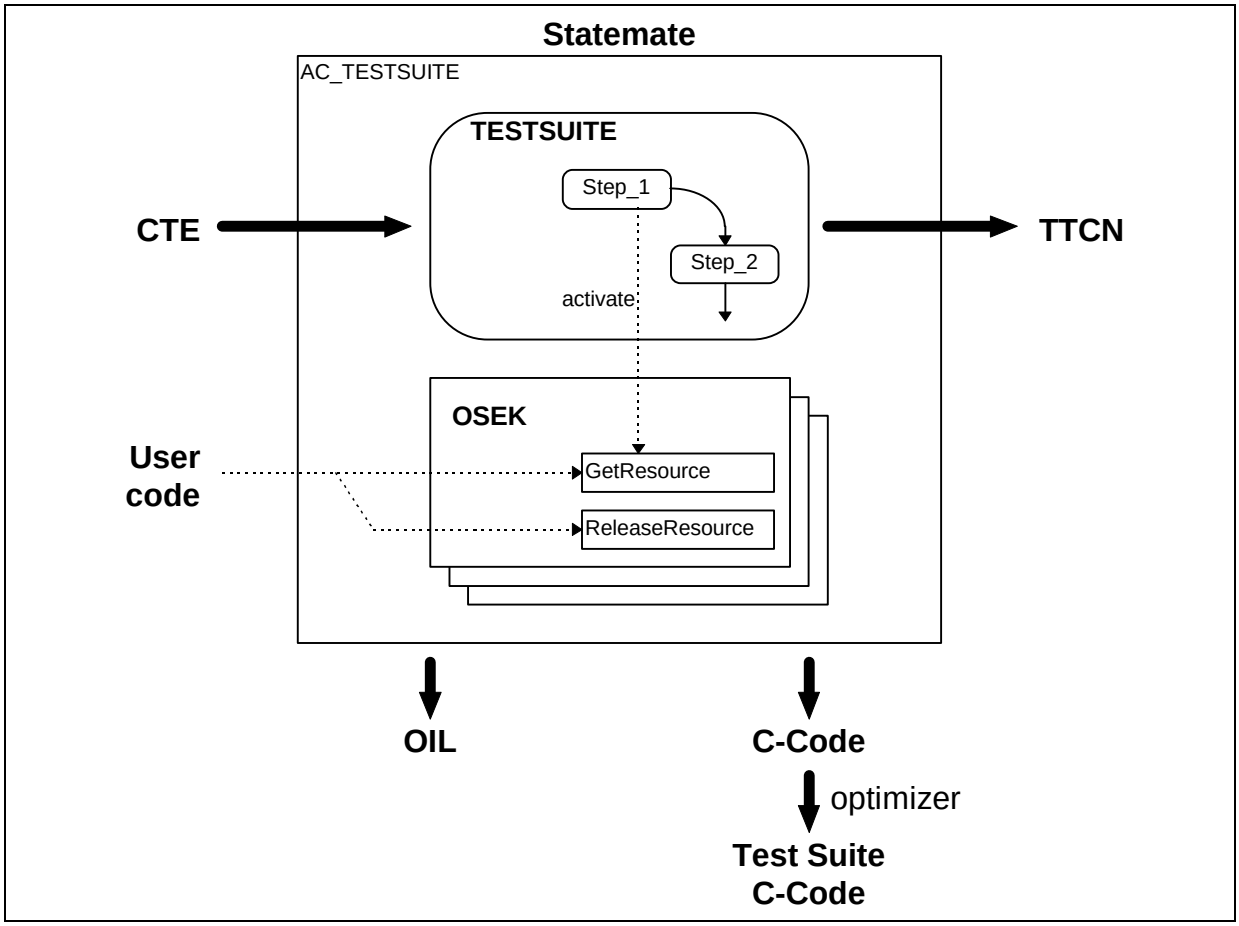


Figure 16 Modelling the test suite in Statemate

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 COM, the protocol definition in state table form will not be taken into consideration since this representation is redundant with the SDL definition.

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 an API call or internal such as a connect event sent by the connection handling automaton to a data transfer automaton,
- 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 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.

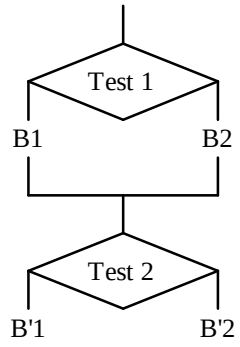


Fig. a: specification

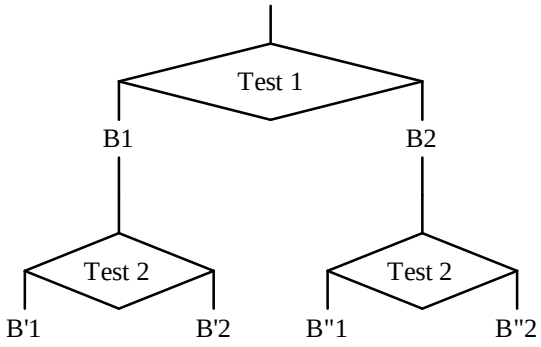


Fig. b: implementation

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".

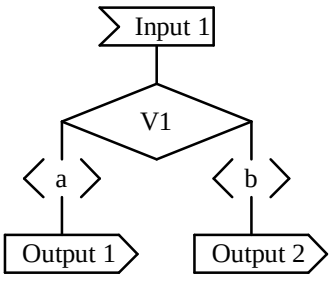


Fig. a: testable

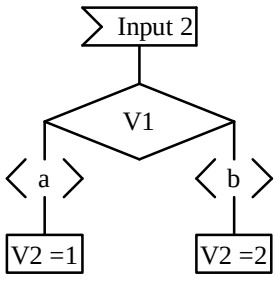


Fig. b: not testable

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_PDUs 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 13. The result is shown in Figure 17:

- 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.

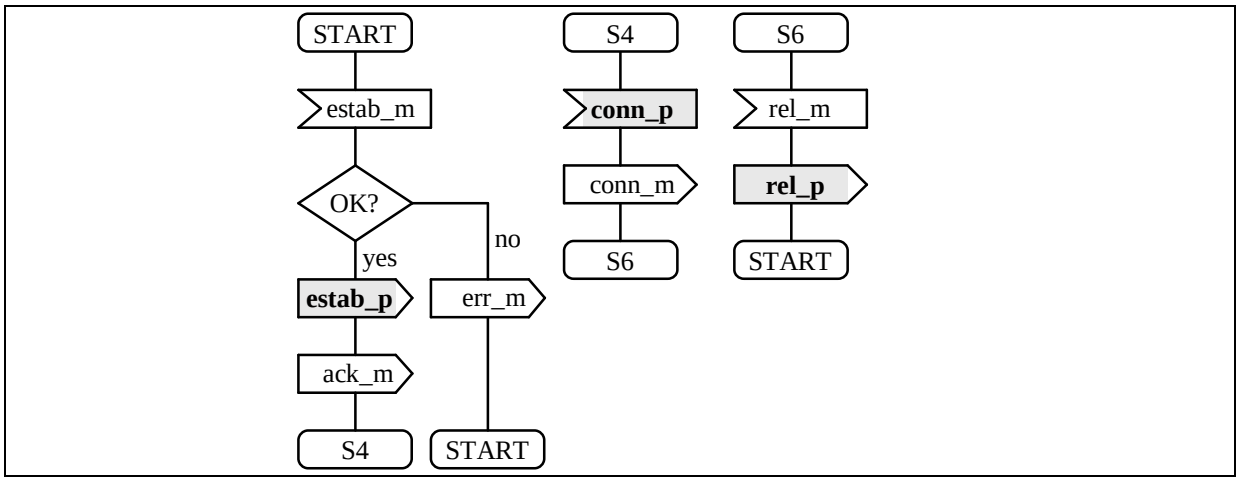
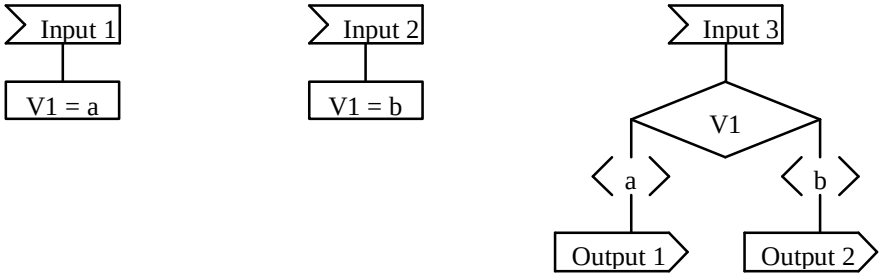


Figure 17 Combined specification of IUT and UT

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:

```
Begin
        Start timer T0
        Send Input1 (resp. Input2) to implementation
        Wait for any implementation output or T0 time-out
        If T0 time-out signalled then
                Verdict = PASS
        If an output is received then
                Stop T0 timer
                Verdict = FAIL
End
```

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

```
Begin
        Send Input1 (resp. Input2) to implementation
        Start timer T1
        Send Input 3 to implementation
        Wait for any implementation output or T1 time-out
        If T1 time-out signalled then
                Verdict = FAIL
        If Output1 (rep. Output 2) received with expected parameters then
                Stop T1 timer
                Verdict = PASS
        If another output or incorrect parameters are received then
                Stop T1 timer
                Verdict = FAIL
End
```

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.

4.2.4. Test generation tools

The generation of test suites from the SDL specifications will be achieved with the help of two tools, the SDT environment from Telelogic and the TGV tool from INRIA.

The SDT environment has been already employed to develop the SDL specifications of OSEK COM and NM. It incorporates a TTCN toolset which allows to develop TTCN test cases and comprises 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 TTCN environment also includes a tool providing for a guided generation of test cases from the SDL specifications. At each step of a test case's definition the tool proposes the possible interactions permitted by the SDL specification and the user can select one of them. This functionality will be used to interactively and iteratively design tests that comply to the OSEK specification by construction. 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 hand-written 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.

The TGV tool aims to automatically generate test cases from SDL specifications. The principle is to compute a test case from an SDL specification and a test purpose:

- The test purpose characterizes the abstract property of the specification that needs to be tested. It is formalised by a finite automaton expressed in SDL. Test purposes are used to select a test case from all possible behaviours of the specification.
- The specification is represented by a transition system which is computed by the verification tool of the SDL toolset. It describes the set of the possible behaviours of the specification. This transition system is translated into an adhoc format. Then, as testing only considers traces of observable interactions, internal actions are discarded and the graph is determinized. The resulting graph represents the observable behaviour of the specification on which TGV's main algorithm can be applied.

The output is a test case which is given by a graph in an ad hoc format which is then translated in TTCN. (A C translation is under study in order to produce executable test cases). This tool will essentially be used to validate the completeness of individual test cases produced with

SDL environment. Its main role will be to verify that all non-compliant behaviours leading to FAIL verdicts are specified in TTCN descriptions.

The principle of test generation with TGV is illustrated in Figure 18.

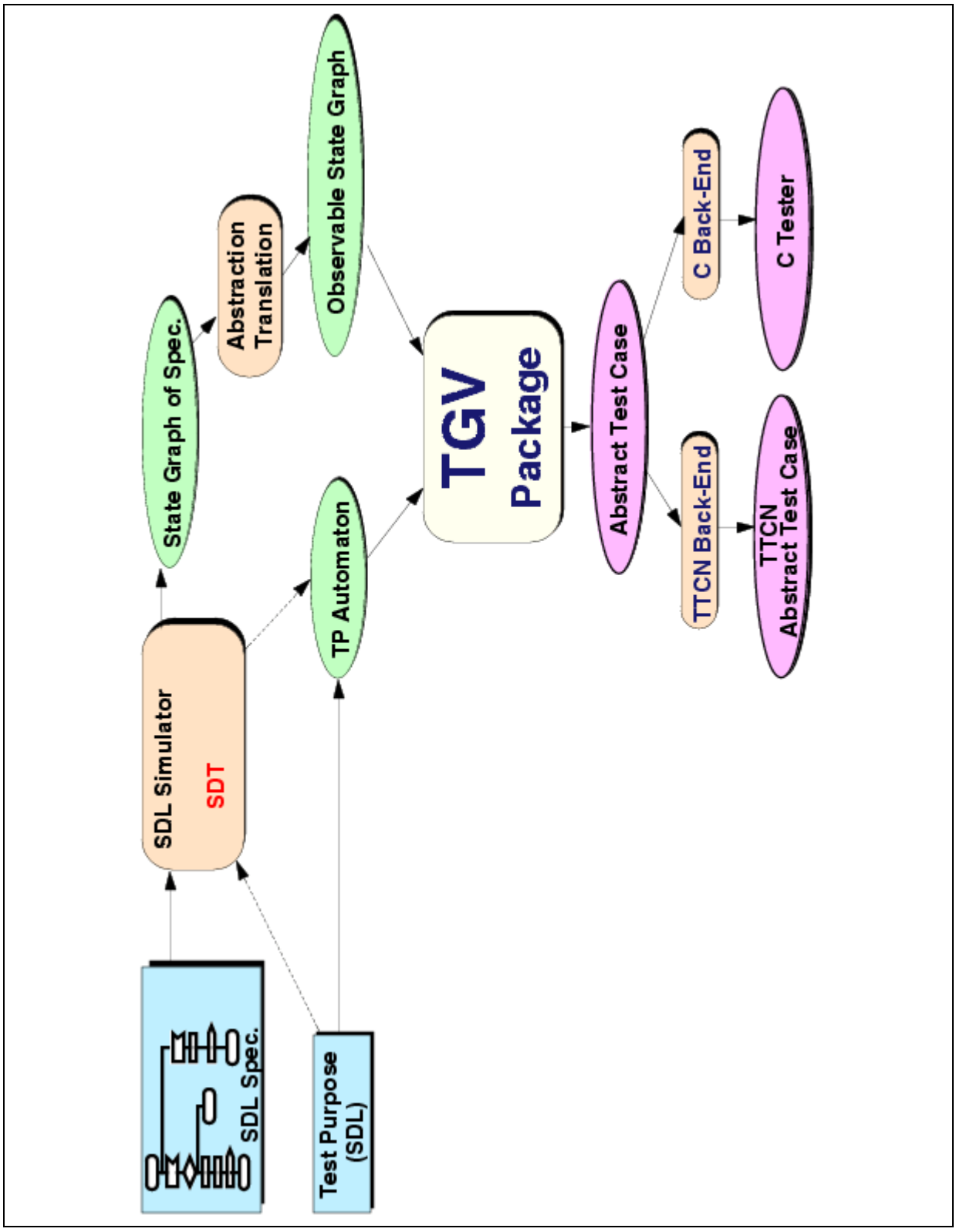


Figure 18 Test generation with TGV

The obtained results can be described through an example. Let us consider again the SDL specification of connection handling protocol shown on Figure 13. 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.

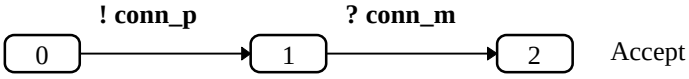


Figure 19 details the abstract test case generated by TGV using the same notations. The test case contains a preamble that brings the specification in a state in which *conn_p* can be received. After the test case has succeeded, a postamble brings the specification in its initial state (provided it can be reached at all). Then operations on timers are inserted so that the liveness of the tester is ensured.

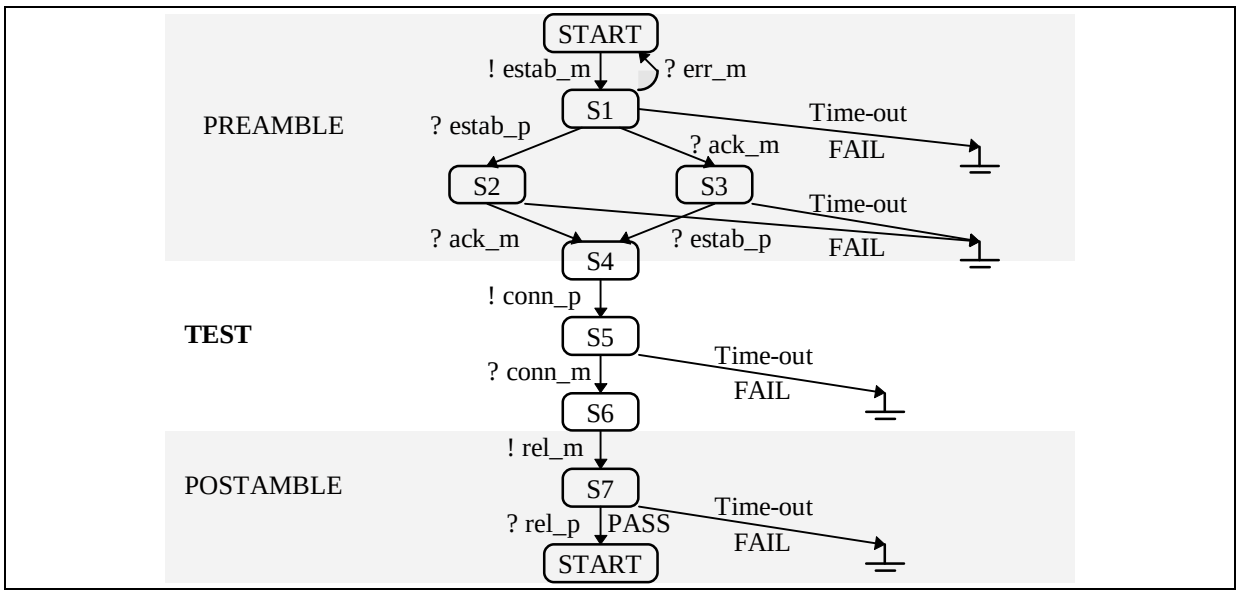


Figure 19 Test case generated by TGV

An outstanding feature of this sequence is the tree subdivision at S1 node allowing reception of *estab_p* and *ack_m* in any order. The reason is that PDUs are transmitted through FIFO channels using different priorities. Since *estab_p* and *ack_m* are respectively transmitted by a UT task and an IUT task, the order of their actual emissions on the data bus cannot be predicted and the two possibilities shall be authorised.

As a conclusion, Figure 20 describes the complete test generation process that is anticipated to support the OSEK COM and NM conformance. The TGV method combined with the guided generation of SDT aim at designing a test suite that conform to the specification (step 1). With SDT, the TTCN test suites can be simulated against the SDL specification and therefore validated (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 and the Lower Tester from the TTCN specification. Both can be

produced with the help of code generation tools of the SDL environment. The so-generated C code needs to be adapted by implementors to each target environment:

- UT adaptation to the equipment under test by each IUT developer,
- LT adaptation to the test equipment by the test tool supplier.

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 the conformance test itself. In other words, it will not be necessary for OSEK implementors to acquire any of the tools to do conformance testing.

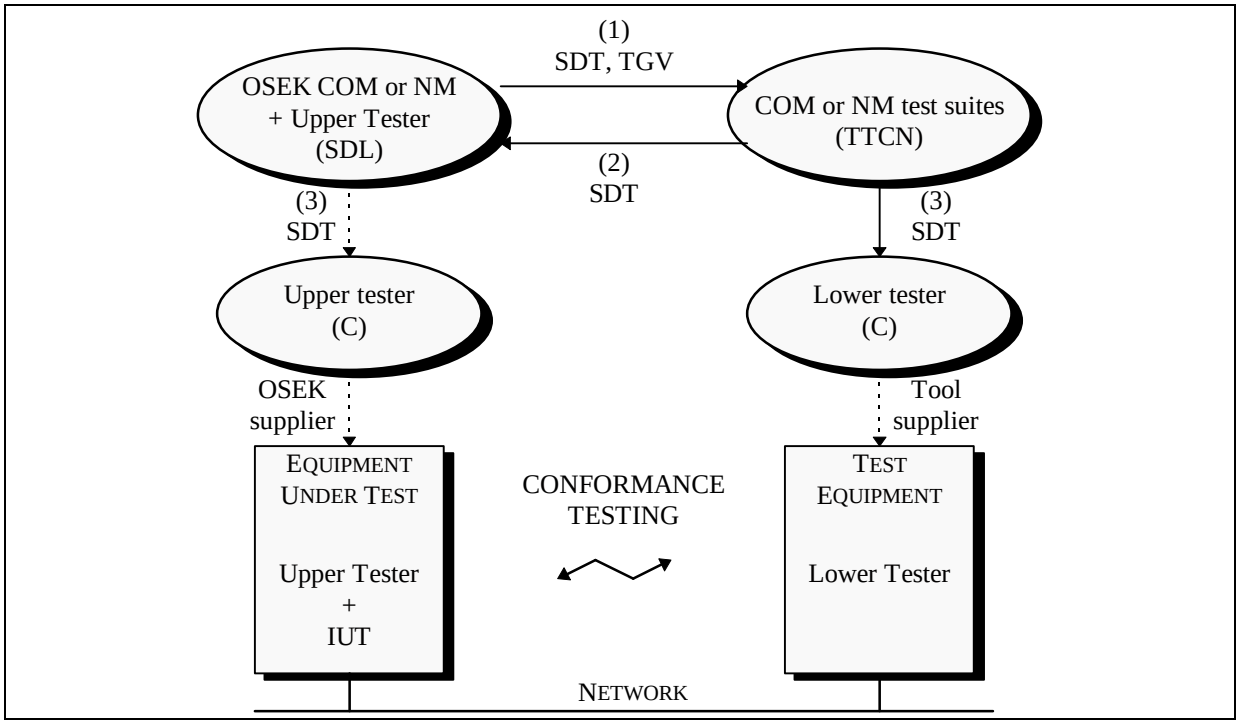


Figure 20 Test suites generation and implementation process

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 IUT and LT according to the coordinated test architecture principle detailed before. Again, the test architecture comprises only one PCO placed at the lower interface of the IUT.

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
L	DLL-API	LT	DLL services access point
Detailed Comments :			

Table 25 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 IUT. 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 : OS-API Comments : task activation procedure		
Parameter Name	Parameter Type	Comments
TaskID	TaskType	Task reference
Detailed Comments :		

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

Table 26 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, especially when the protocol uses integers for identifying PDUs (sliding window protocols, etc.). 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 can be defined in the test suite itself or externally in 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 internally or externally.

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

Table 27 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 : OS-API 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 : DLL-API Derivation Path : Comments : establishment request for connection number 10		
Field Name	Field Value	Comments
ConNum PCI	10 EstabPCIValue	Connection Number Protocol Control Information
Detailed Comments :		

Table 28 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 or PASS) 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 29 Test case dynamic behaviour

6. Abbreviations

API	Application Programming Interface
COM	Communication
DLL	Data Link Layer
ECU	Electronic Control Unit
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 - DRAFT - Version 2.0 - Draft 1.5 - 1997 05 14 *(To be updated)*

[4] OSEK Network Management - Concept and Application Programming Interface- Version 2.0 - 4th of April 1997

[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