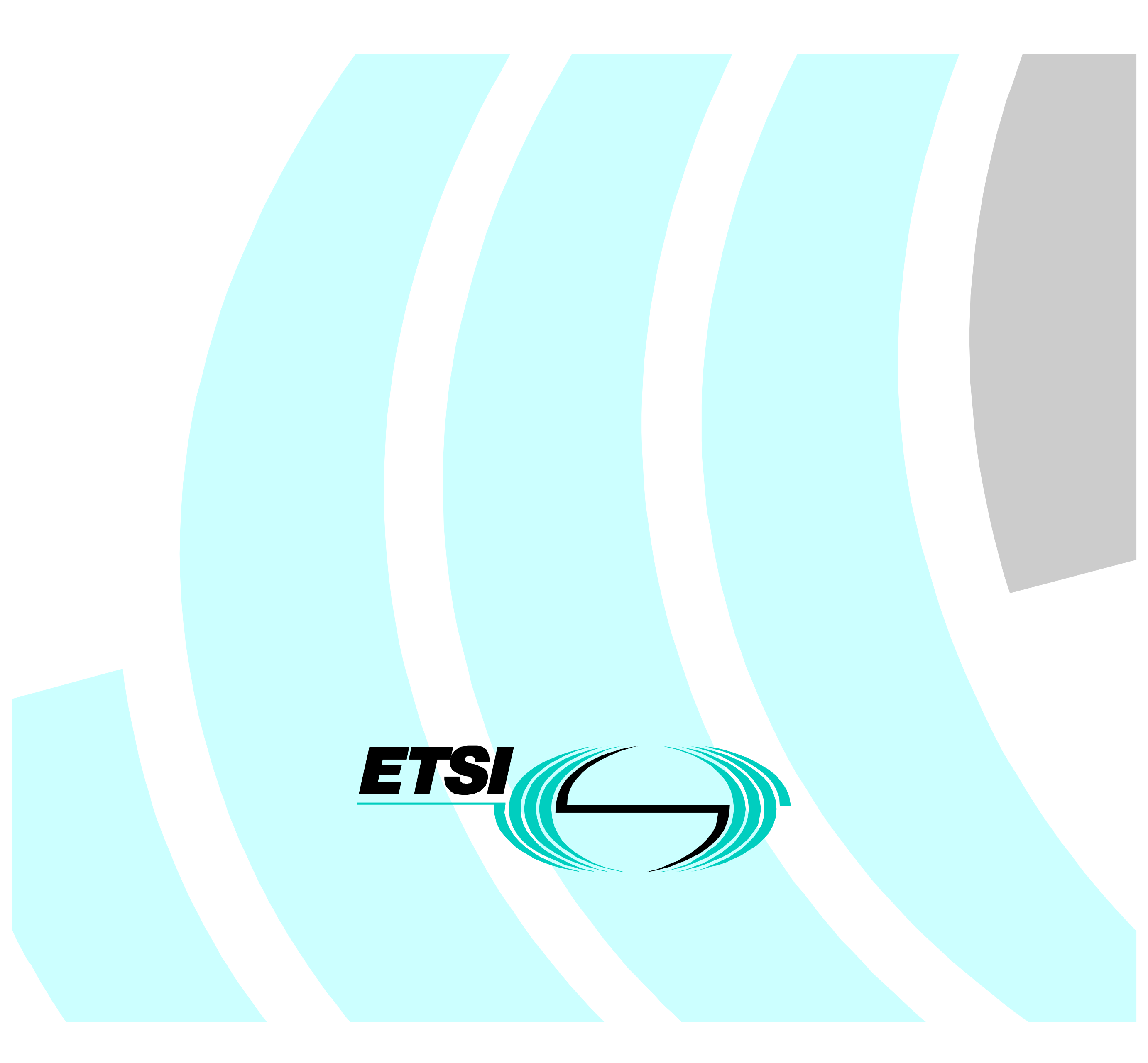
ETSI TR 101 873-3 V1.1.1 (2001-01)

Technical Report

**Methods for Testing and Specification (MTS);
The Tree and Tabular Combined Notation version 3;
Part 3: TTCN-3 MSC Presentation Format**



Reference

DTR/MTS-00063-3

Keywords

ASN.1, methodology, MSC, MTS, testing, TTCN

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

Individual copies of the present document can be downloaded from:
http://www.etsi.org

The present document may be made available in more than one electronic version or in print. In any case of existing or perceived difference in contents between such versions, the reference version is the Portable Document Format (PDF). In case of dispute, the reference shall be the printing on ETSI printers of the PDF version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at http://www.etsi.org/tb/status/

If you find errors in the present document, send your comment to:
editor@etsi.fr

Copyright Notification

No part may be reproduced except as authorized by written permission.
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 2001.
All rights reserved.

Contents

Intellectual Property Rights 7
Foreword....7
Introduction....7
1 Scope....9
2 References....9
3 Abbreviations....9
4 Overview....9
5 TSC language concepts....12
5.1 TSC document....12
5.1.1 Parameterization of TSC documents....13
5.1.2 Implicit and explicit typing for components and ports....13
5.1.2.1 Implicit typing for components and ports....13
5.1.2.2 Explicit typing for components and ports....13
5.1.3 Importing other TSC documents....14
5.1.4 Variable declarations....14
5.1.5 Data definitions....14
5.1.6 Data parenthesis declaration....14
5.2 Control part....14
5.2.1 Parameterization of TSCs....15
5.2.2 Control variables....16
5.2.3 References to test cases....16
5.2.4 Control actions....16
5.2.5 Alternative expressions....17
5.3 Test cases....17
5.4 Instances....18
5.4.1 Test component instances....18
5.4.2 Explicit port instances....18
5.4.3 Implicit port instances....19
5.5 Test component handling....20
5.6 Port handling....21
5.6.1 Handling of explicit ports....21
5.6.2 Handling of implicit ports....22
5.7 Configurations....23
5.8 Data....24
5.8.1 Declaring data....24
5.8.2 Global data....24
5.8.3 Static data....24
5.8.4 Dynamic data....25
5.8.4.1 Component variables....25
5.8.4.2 Bindings....25
5.8.4.3 Component initiation....25
5.8.4.4 Data in messages....25
5.8.4.5 Action boxes....25
5.8.4.6 Component and port types....25
5.8.5 Guarding conditions....26
5.8.6 Setting conditions....26
5.9 Timers....26
5.10 Asynchronous communication....27
5.10.1 Messages....27
5.10.2 Receiving any message....29
5.10.3 Receiving from any port....29
5.10.4 Trigger message....29
5.11 Synchronous communication....29

5.11.1 Call, getreply, catch, and timeout ... 30
5.11.2 Getcall, reply, and raise ... 32
5.11.3 Any handling ... 33
5.12 Behaviour ... 34
5.12.1 Sequential behaviour ... 34
5.12.2 Alternative behaviour ... 34
5.12.3 Interleaved behaviour ... 36
5.12.3.1 Co-regions ... 36
5.12.4 Loops ... 37
5.12.5 Functions ... 37
5.12.6 Defaults ... 37
5.12.7 Return statement ... 38
5.12.8 Action Boxes ... 39
5.13 Verdicts ... 39
5.14 High-level TSC (HTSC) ... 40
5.15 Hyper_TSC ... 41
5.16 Partial TSC ... 44
5.17 Hybrid_TSC ... 44

Annex A: Used subset of and extensions to MSC ... 46

A.1 Overview ... 46
A.1.2 Test specific extensions ... 47
A.1.2.1 TSC document ... 47
A.1.2.1.1 New document sections ... 47
A.1.2.1.2 Component and port instances ... 47
A.1.3 TSC heading ... 48
A.1.4 Test components ... 48
A.1.4.1 MTC as keyword ... 48
A.1.4.2 Self as keyword ... 48
A.1.5 Messages ... 48
A.1.6 Trigger ... 49
A.1.7 Check ... 49
A.1.8 Control flow ... 49
A.1.9 Test verdicts within conditions ... 50
A.1.10 Timer ... 50
A.1.11 Create to test components ... 50
A.1.12 Start to test components ... 50
A.1.13 Return for functions ... 51
A.1.14 Stop on test components ... 51
A.1.14.1 Special meaning of stop on the MTC instance ... 51
A.1.14.2 Stop within an operand of an in-line expression ... 51
A.1.15 Clear, start and stop to ports ... 52
A.1.16 In-line expressions ... 52
A.1.16.1 Propagation of messages to the environment ... 52
A.1.16.2 New interleave in-line expression ... 52
A.1.17 HTSC ... 53
A.1.18 Hybrid TSCs ... 53
A.1.19 Extensions to the data part ... 53
A.1.19.1 Declaring Data ... 53
A.1.19.2 Static Data ... 54
A.1.19.3 Dynamic Data ... 54
A.1.19.4 Bindings ... 54
A.1.20 Hyper TSCs ... 55
A.1.21 Ports ... 55
A.1.22 Default ... 55

Annex B: The TSC forms ... 56

B.1 Overview ... 56
B.1.1 An example ... 56
B.1.2 Form aspect: vertical split vs. no vertical split ... 57
B.1.3 Form aspect: horizontal split vs. no horizontal split ... 58

B.1.4 Form aspect: explicit vs. implicit port representation ... 60
B.1.5 Form aspect: hybrid vs. not hybrid ... 61
B.1.6 Form aspect: partial vs. complete ... 62
B.1.7 Summary of TSC forms ... 62

Annex C: Subset of the graphical syntax of TSC ... 63

C.1 Meta-Language for TSC/gr ... 63

C.2 Conventions for the syntax description ... 64

C.3 Relation between TTCN-3 and TSC Files ... 64

C.4 The TSC/gr production rules ... 65
C.4.1 Test Sequence Chart document ... 65
C.4.2 Groups ... 68
C.4.3 Test Sequence Chart ... 68
C.4.4 Environment and Ports ... 70
C.4.5 Basic TSC ... 72
C.4.5.1 Instances (Component and Port Instances) ... 72
C.4.5.2 Messages ... 74
C.4.5.3 Control Flow ... 75
C.4.5.4 Special Messages ... 78
C.4.5.5 Environment and Ports ... 78
C.4.5.6 Conditions ... 80
C.4.5.7 Timers ... 81
C.4.5.8 Actions ... 82
C.4.5.9 Defaults ... 82
C.4.5.10 Instance creation ... 83
C.4.5.11 Instance stop ... 83
C.4.6 Structural concepts ... 83
C.4.6.1 Co-regions ... 83
C.4.6.2 In-line expressions ... 84
C.4.6.3 TSC references ... 84
C.4.7 High-level TSC (HTSC) ... 86

Annex D: Mapping TSC to TTCN-3 ... 88

D.1 Description ... 88
D.1.1 TSC documents ... 88
D.1.1.1 TSCDocument ... 88
D.1.1.2 TSCDocumentHead ... 88
D.1.1.2.1 TTCN3DataDefinitionList ... 89
D.1.1.2.2 TTCN3DataDefinition ... 89
D.1.1.3 ControlPartArea ... 89
D.1.1.3.1 TSCControlReferenceList ... 90
D.1.1.3.2 TSCFunctionorGroupReferenceList ... 90
D.1.1.3.3 TSCFunctionorGroupRef ... 90
D.1.1.3.4 TSCControlReferenceArea ... 90
D.1.1.4 TestcasePartArea ... 91
D.1.1.4.1 TSCTestcaseorGroupReferenceList ... 91
D.1.1.4.2 TSCTestcaseReferenceArea ... 91
D.1.1.5 FunctionPartArea ... 91
D.1.1.5.1 TSCFunctionReferenceArea ... 92
D.1.1.6 NamedAltPartArea ... 92
D.1.1.6.1 TSCNamedAltorGroupReferenceList ... 92
D.1.1.6.2 TSCNamedAltReferenceArea ... 92

Annex E: An INRES example in TSC ... 93

E.1 TSC document ... 93

E.2 TSCs for sequential test case ... 94
E.2.1 First version ... 94
E.2.2 Second version ... 96

E.2.3 Third version ... 98
E.3 TSCs for concurrent test case ... 99
E.4 INRES example in TTCN-3 core language ... 108
History ... 115

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://www.etsi.org/ipr).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Report (TR) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

Introduction

The Message Sequence Chart (MSC) presentation format for TTCN-3 is a graphical format based on the ITU-T Recommendation Z.120 [2].

This presentation format uses a subset of Message Sequence Charts with test specific extensions and extensions of general nature. This MSC profile will be in the following referred to as Test Sequence Charts (TSCs). The majority of extensions are textual extensions only. Graphical extensions are defined to ease the readability of TSC documents. All graphical extensions have either substitutes within standard MSC or are optional (i.e. need not to be used). This allows established MSC tools (with some modifications to the textual syntax) to be used for the graphical definition of TTCN-3 test cases using TSCs. The used subset of and extensions to MSC are described in annex A.

The core language of TTCN-3 is defined in ES 201 873-1 [1] and provides a full text-based syntax, static semantics and operational semantics as well as defining the use of the language with ASN.1. The TSC presentation format provides an alternative way of displaying the core language.

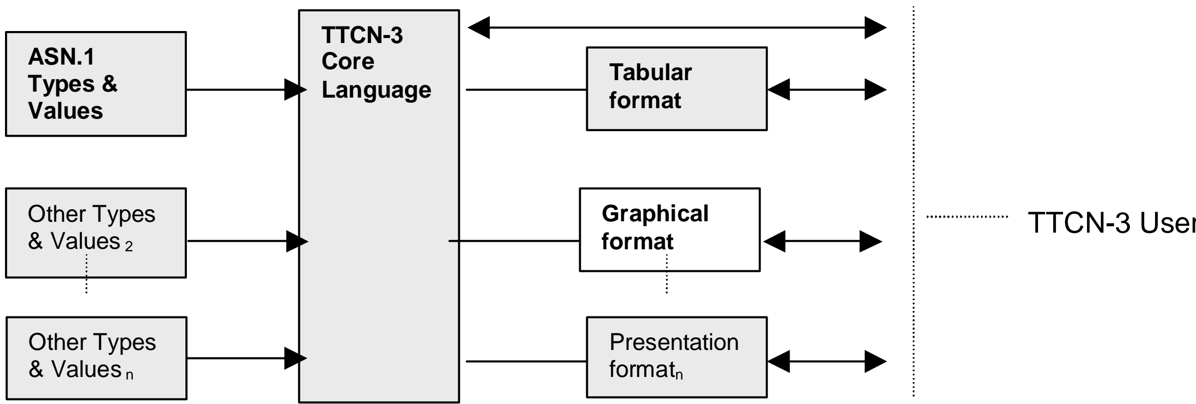


Figure 1: User's view of the core language and the various presentation formats

The core language may be used independently of the TSC presentation format. However, the TSC format cannot be used without the core language. Use and implementation of the TSC presentation format shall be done on the basis of the core language.

The present document describes how different forms (styles) of Message Sequence Charts (MSC) can be used for describing test specifications. In doing so, we introduce new language concepts that are necessary for the definition of test specifications.

The present document defines:

- the language features of TSC;
- the guidelines for the use of TSC;
- the used subset of and extensions to MSC.

Together, these characteristics form the TSC presentation format.

1 Scope

The present document defines the Message Sequence Chart (MSC) presentation format for the TTCN-3 core language as defined in ES 201 873-1 [1]. This presentation format uses a subset of Message Sequence Charts as defined in [2] with test specific extensions and extensions of general nature.

The present document is based on the core TTCN-3 language defined in ES 201 873-1 [1]. It is particularly suited to display tests as TSCs. It is not limited to any particular kind of test specification.

The specification of other formats is outside the scope of the present document.

2 References

For the purposes of this Technical Report (TR) the following references apply:

[1] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 1: TTCN-3 Core Language".

[2] ITU-T Recommendation Z.120 (1999): "Message Sequence Chart (MSC)".

[3] ISO/IEC 9646-3 (1994): "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)".

3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ASN.1 Abstract Syntax Notation One
BNF Backus-Nauer Form
CATG Computer Aided Test Generation
CORBA Common Object Request Broker Architecture
ETS Executable Test Suite
ETSI European Telecommunication Standards Institute
HTSC High-level Test Sequence Chart
MSC Message Sequence Chart
MTC Master Test Component
PTC Parallel Test Component
SUT System Under Test
TSC Test Sequence Chart
TTCN Tree and Tabular Combined Notation

4 Overview

According to the OSI conformance testing methodology defined in ISO/IEC 9646-3 [3], testing normally starts with the development of a test purpose, defined as follows.

A prose description of a well-defined objective of testing, focusing on a single conformance requirement or a set of related conformance requirements as specified in the appropriate OSI specification.

Having developed a test purpose an abstract test suite is developed that comprises of one or more abstract test cases. An abstract test case defines the actions necessary to achieve part (or all) of the test purpose.

Applying these terms to Message Sequence Charts (MSCs) we can define two categories for their usage:

1) Test Purposes – Typically, a MSC specification that is developed as a use-case or as part of a system specification. For example, figure 2 illustrates a simple MSC describing the interaction between instances representing the System Under Test (SUT) and its environment. Such MSC specifications can represent many different behaviours. Therefore, one or more abstract test cases may be required to ensure that the SUT conforms to the specification. Note that the inclusion of SUT instances is optional, and that both the SUT and Environment can be defined using more than one instance. Figure 2 illustrates the typical configuration used during the development of test purposes.

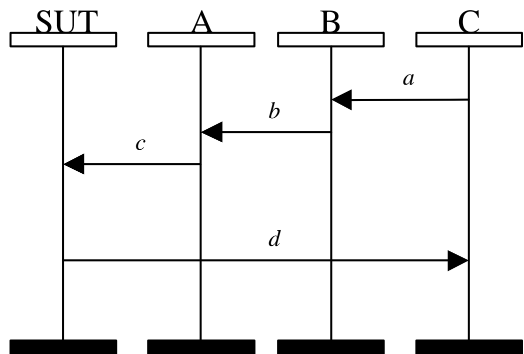


Figure 2: MSC illustrating how the SUT interacts with its environment

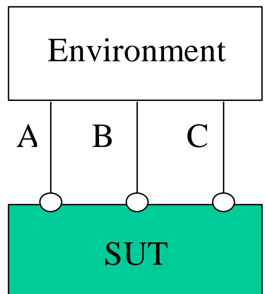


Figure 3: Illustrates the architecture that is normally represented by a test purpose

2) Abstract Test Cases (or Test Suite) – Typically, a MSC written solely for the purpose of describing the behaviour of a test case. For example, figure 3 illustrates a simple MSC defining the interactions between different elements of the test configuration e.g. instances can represent test components, ports mapped to the SUT and ports connected to other test components. Figure 3 illustrates the test configuration used by the MSC specification.

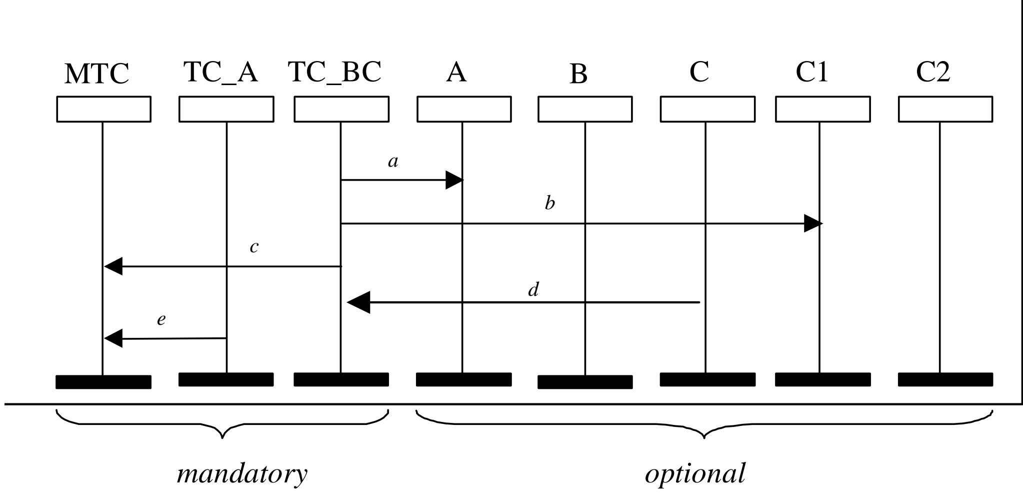


Figure 4: A MSC illustrating both the behaviour and configuration of a test case

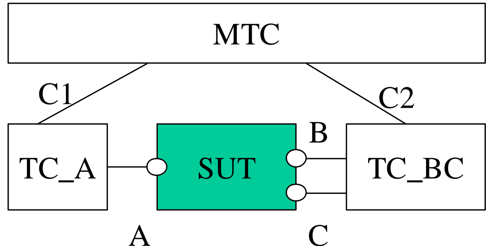


Figure 5: Configuration used by the above MSC

In identifying these two categories of MSC usage two distinct areas of work can be defined see figure 6:

a) Develop a mapping from MSC test purposes to TTCN-3 test cases (or TSC test cases – see next item). However, it is perceived that such a mapping would be non-trivial, and would involve the development of Computer Aided Test Generation (CATG) techniques.

b) Develop Test Sequence Charts (TSC) being a test specific profile of MSC for the purpose of specifying abstract test cases (and test suites) together with a mapping from TSC test cases to TTCN-3.

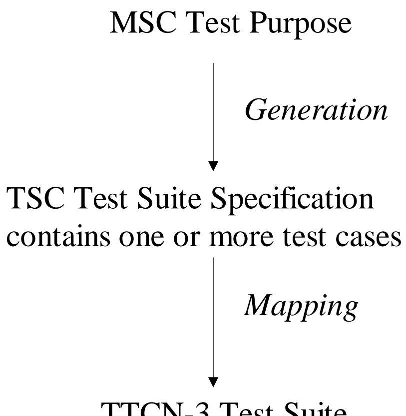


Figure 6: Diagram illustrating the different levels of abstraction for sequence charts

NOTE: Item b) is the focus adopted during the development of the present document.

5 TSC language concepts

This clause explains the language concepts of TSC and indicates their use for the specification of test suites in TSC.

5.1 TSC document

The Test Sequence Chart document defines a test suite and the associated collection of test sequence charts, which again define traces of test events. An example is illustrated in figure 7.

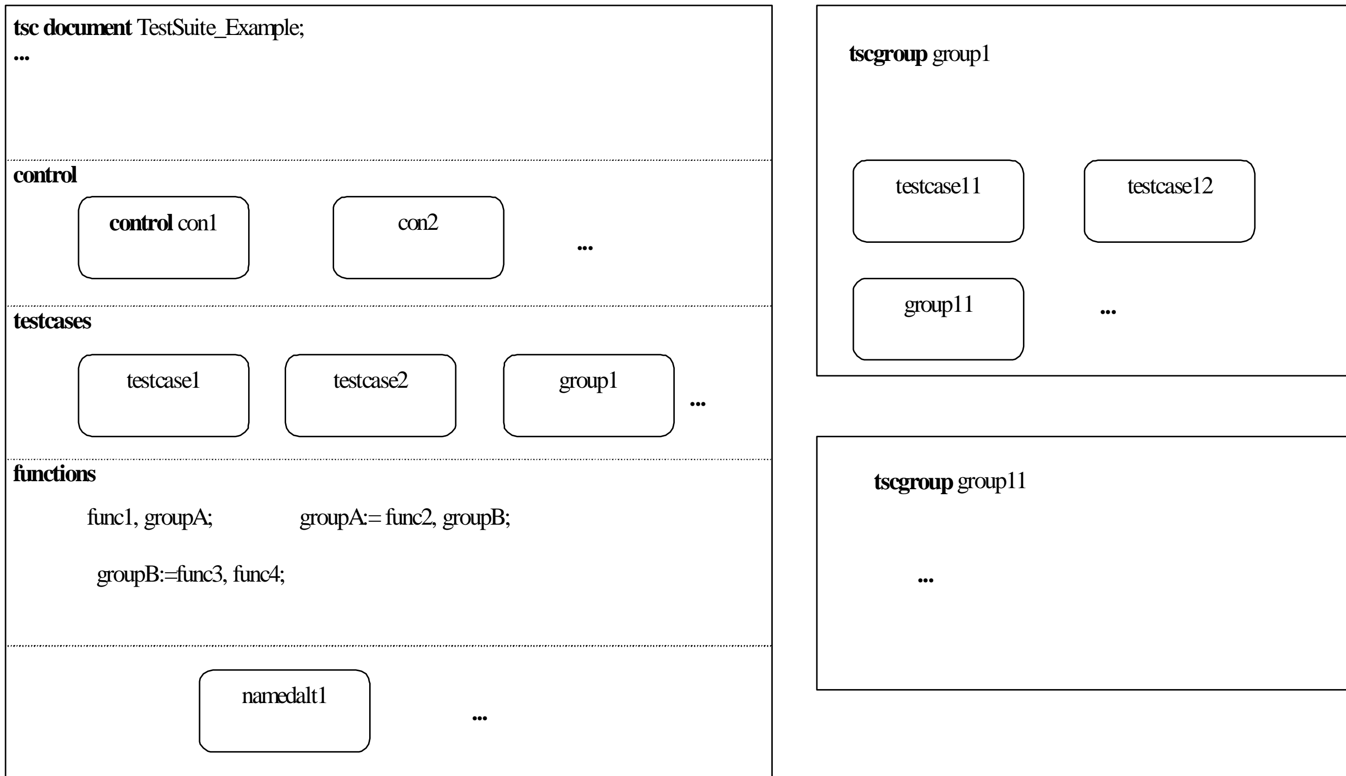


Figure 7: TSC document

The TSC document may contain a collection of Test Sequence Charts (TSCs), together with any declarations needed for the test suite. Within a TSC document TSCs are grouped according to their function.

The control part, identified using the **control** keyword, refers to one or more control TSCs, which are defined using the HTSC notation. These control HTSCs may refer to other control HTSCs, or to TSCs representing test cases. The root control HTSC is indicated by the keyword **control** followed by the HTSC identifier.

The test cases part, identified using the **test cases** keyword, contains TSCs that define the interaction between the ports and components contained in the TSC document.

The functions part, identified using the **functions** keyword, contains test behaviour patterns, which are reused by the test case part.

The named alternatives part, identified using the keyword **named alts,** contains the named alternatives used in TSCs.

TSCs within the TSC document may be grouped providing a hierarchical structure. A TSC groups references a TSC group document, denoted using the keyword **tscgroup,** which in-turn references TSCs or other TSC groups.

TSC references of each part of the TSC document can be denoted either using graphical reference symbols or in the form of a list of identifiers. The only constraint is that the two representations cannot be mixed within one area.

A TSC document is like a MSC document except of the more fine-grained differentiation into control, test cases, components and functions TSC parts (as opposed to defining and utility MSC parts only).
Whenever there is no special support for the different TSC document areas, the keywords **control**, **test cases**, **functions**, and **named alts** should be put into comment symbols, which serve like the separation line as proposed above.

5.1.1 Parameterization of TSC documents

A TSC document parameter list defines a set of values that are supplied by the test environment at run-time. During testing execution these values shall be treated as constants. For example:

```
tscdocument TestSuiteExample (integer TS_par1, boolean TS_par2);
```

The data language used with TSC defines the form given to TSC document formal parameters. The syntax given to formal parameters is taken from TTCN-3.

5.1.2 Implicit and explicit typing for components and ports

Ports facilitate communication between test components, and between test components and the test system interface. Where, a TSC port instance can support message-based events, procedure-based events or a mixture of both. Test components then use these ports to communication with other test components or the test system interface.

Where, a port type defines the method of communication that can pass through a port (e.g. message-based, procedure-based or mixed), including the direction of communication (e.g. in, out or inout).

TSC supports both implicit and explicit typing for components and ports as described in [1].

5.1.2.1 Implicit typing for components and ports

Implicit typing of components and ports is defined using the keyword **implicit_typing** within the TSC document header. For example:

```
tscdocument TestSuiteExample implicit_typing;
```

Implicit typing means that any typing of ports or components is determined from the message interactions between ports and component instances.

5.1.2.2 Explicit typing for components and ports

Explicit typing of components and ports is defined using the keyword **explicit_typing** within the TSC document header. This means that all port or component types must be given either in the TSC document header, or within instance headers. For example:

```
tscdocument TestSuiteExample explicit_typing;
data {
    type port MyPortType= {...};
    type component MyMTCType= {...};
}
```

5.1.3 Importing other TSC documents

TSC documents may be imported using an import clause in the document declaration part or by an import statement within the test case reference. The syntax for the import clause follows that defined in [1].

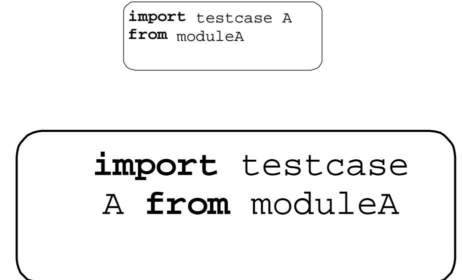


Figure 8: Example of an imported test case

5.1.4 Variable declarations

Component variables are declared within the TSC document header as part of the declaration contained with the data definition part, which follows TTCN-3 syntax as defined in [1]. For example:

```
tscdocument ExampleTestSuite explicit_typing;
data {
    type component MyMTC { ...; var MyType1 a, b; var MyPTCType PTC1;... }
    type component MyPTC1 { ...; var MyType2 c; var MyType3 d; ...}
}
```

5.1.5 Data definitions

As described in clause 6.8 the TSC notation can be parameterized with a data language of the users choice. In doing so, data definitions from the chosen language are given within the TSC document header. For example, if TSC is parameterized with TTCN-3 data types and values, a TSC document header could look as follows:

```
tscdocument ExampleTestSuite (SuiteParType par1);
language TTCN-3 data {
    type integer SuiteParType;
    type enumerated SeqNo { zero (0), one (1) };
    type record MyMessageType { integer field1, SeqNo field2 };
    template MyMessageType MyTemplateType { 1 , ? };
}
```

5.1.6 Data parenthesis declaration

Delimiters denoting the start and finish of data strings are declared in the TSC document header (see also clause 6.8).

5.2 Control part

The control part of a TSC document may reference one or more High-level TSCs (HTSC). Together, all HTSCs referenced within the control part of a TSC document define the execution order (and possibly repetitions) of actual test cases. One of the control HTSC can be distinguished from other HTSCs using the **control** keyword. This control HTSC represents the root of the control structure for the test suite, and may refer to other HTSCs that are also contained within the control part of the TSC document. Figure 9 illustrates a simple example of a controlling High-level Test Sequence Chart (HTSC).

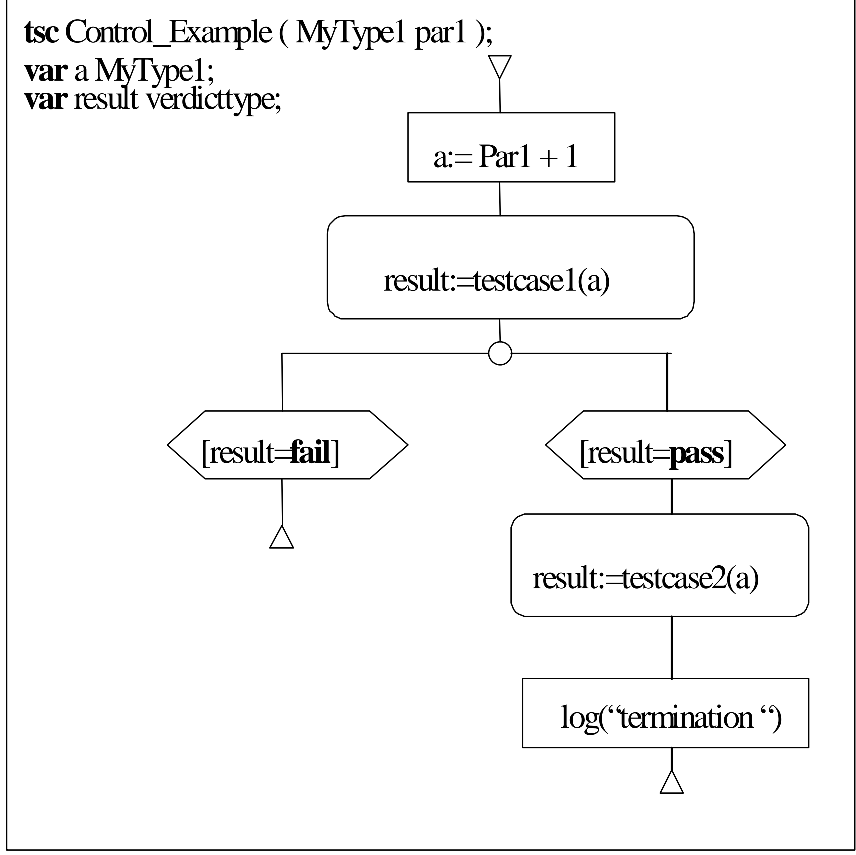


Figure 9: Example of a control HTSC

Because all the HTSCs contained within the control part of the TSC document represent a single thread of control, parallel composition is not allowed within a control HTSC, but alternative expressions with guarded conditions are allowed. Also, all reference expressions are not supported within control HTSCs.

A Control HTSC is similar to a High-level MSC with the addition of formal parameters, local variables, action boxes containing

A Control HTSC is similar to a High-level MSC with the addition of formal parameters, local variables, action boxes containing data expressions or log operations, guarded conditions, and references to TSC test cases that return values. Some restrictions have been applied to the usage of MSC reference expressions.

5.2.1 Parameterization of TSCs

A control HTSC parameter list defines a set of values that are supplied by the test environment at run-time. During testing execution these values shall be treated as constants. For example:

```
control ControlExample (integer TS_par1, boolean TS_par2);
```

The data language used with TSC defines the form given to control TSC formal parameters. In the example illustrated above the syntax given to formal parameters is taken from TTCN-3 (see also clause 6.8).

5.2.2 Control variables

A controlling HTSC may declare local variables. These variables are different from component variables, as declared within a TSC document header (see also clause 6.8.4.1), because they are not bound to a single component instance, and they are not globally visible. Instead controlling variables are declared within the TSC header and are only visible within the HTSC they are declared.

For example:

```
control control_example
var integer a:=0;
var verdicttype result1,result2;
```

As with other data definitions the form given for control variable declarations follows that of the data language used with TSC. For example, the variable declarations illustrated above are defined using TTCN-3 syntax.

5.2.3 References to test cases

A control HTSC may refer to TSC test cases referenced within the test case part of a TSC document using the reference symbol. In addition to simple TSC references, a reference symbol may also contain a binding expression.

Binding expressions allow the values returned by TSC test cases to be assigned to a local control variable. The left-hand side of a binding expression is the identifier of the local control variable, and the right-hand side of the expression is the test case reference.

For example, figure 10 illustrates a reference to a TSC test case whose value is assigned to the local control variable.

```
result:=testcase(a)
```

Figure 10: Example of a test case reference binding expression

A timeout on test execution may be defined either using the time constraint notation of MSC or using the keyword **duration**. If the test case fails to complete within the given time constraint an **error** verdict is returned.

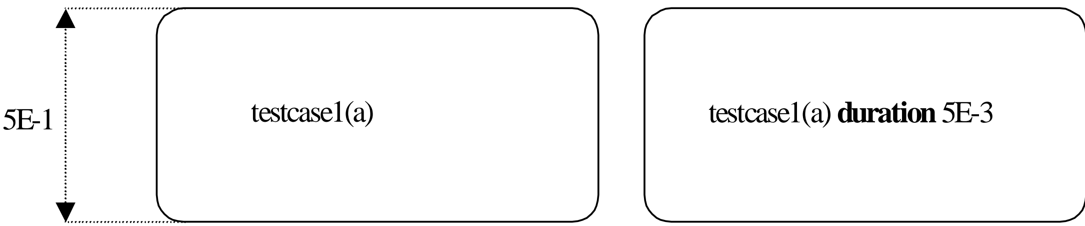


Figure 11: Time supervision of test case execution

5.2.4 Control actions

Action boxes can be used within control HTSCs to introduce expressions, which operate over local control variables, or to include log statements:

1) a log statement can be introduced into an action box using the TTCN-3 log statement;

2) simple binding expressions that operate over local control variables or parameters can be given in action boxes contained within a control HTSC. For example, figure 12 illustrates two typical TTCN-3 expressions in which a local control variable (on the right-hand side) is bound to an expression involving a formal parameter (on the left-hand side). A semi-colon separates multiple expressions.

```
varA:=par1;
varB:=par1+1;
```

Figure 12: Example of an action box containing control expressions

5.2.5 Alternative expressions

Alternative expressions allow choices to be introduced into a control HTSC, where each alternative represents a different thread of control actions. Within a control HTSC all alternative branches must be guarded using guarding conditions (see clause 6.8.5). For example, figure 9 illustrates an alternative with two possible choices both of which are guarded by Boolean conditions. The order in which alternative choices is from top to bottom or from left to right, respectively. If none of the guarding conditions evaluates to true during run-time then control execution is stopped.

5.3 Test cases

Test cases (see [1]) are represented using Test Sequence Charts (TSCs). A TSC represents the flow of test events between test component instances, port instances and/or the environment.

A test case declaration is represented by a TSC within a box. The test case identifier is used as the identifier for the TSC. The component type identifier given after the **runs on** keyword within TTCN-3, is represented in TSC as the type of the MTC test component instance, contained inside the instance head symbol (for further details please refer to clause 6.5). The type declaration for the test system interface is put into the TSC header with the **system** keyword (taking the TTCN-3 syntax).

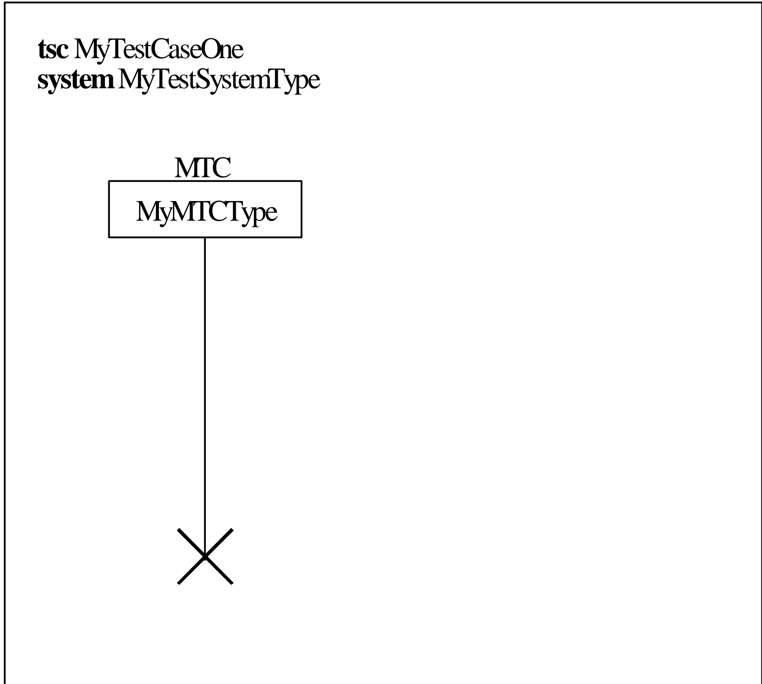


Figure 13: TSC for a test case with MTC

The result of an executed test case always is a value of type **`verdicttype`**.

A test case is treated like an MSC except of the addition of the system clause to the header information.

5.4 Instances

TSC instances represent either test components or ports (see [1]). In order to differentiate the two kinds of TSC instances, different graphical symbols are used. TSC instances contained in the TSCs referenced from the control, test cases, components, or functions part must only be chosen from the list of instances declared in the instance list of the TSC document.

5.4.1 Test component instances

Figure 14 shows a test component instance. The instance head symbol is an unfilled box, on top of which the identifier of the test component and inside of which optionally the type of the test component is given. The head symbol is followed by the instance line (represented as a solid line). The events on this instance line are totally ordered from top to bottom. The end of a behaviour description for a test component is indicated by the instance end symbol, which is a filled box.

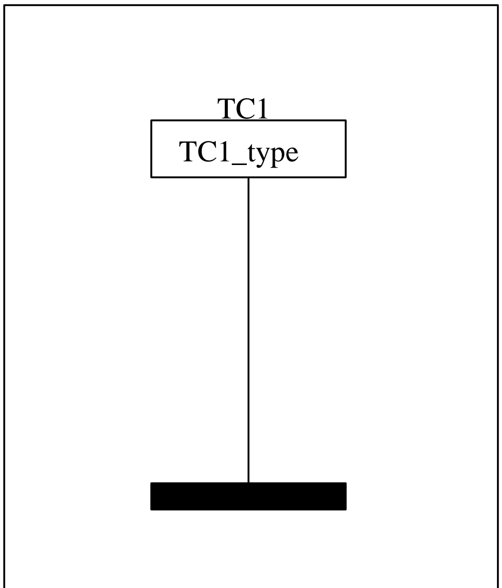


Figure 14: Test component instance

There is a special test component instance , denoted using the **mtc** keyword, which represents the main test component of a test case (see [1]). It has not to be declared explicitly as it is given implicitly. The concrete MTC type for a test case is taken from the **runs on** and/or the type information in the **mtc** instance head.

A test component instance is treated like an instance in MSC. Test component instances may use the following TSC constructs: synchronous and asynchronous communication concepts, in-line expressions, co-regions, conditions, action boxes, timers, instance creation, TSC references, stop and special messages for test component and port handling (see clauses 6.5 and 6.6).

5.4.2 Explicit port instances

Figure 15 shows an explicit port instance. The instance head symbol is an unfilled box with dashed lines, on top of which the identifier of the port and inside of which optionally the port type is given. The head symbol is followed by an port instance line (represented as a dashed line). The end of the behaviour description for a port is indicated by the port instance end symbol, which is an empty, unfilled box with dashed lines. Also to maintain backward compatibility with existing MSC tools TSC allows solid lines to be used for representing port instances. In such cases the **port** keyword is placed before the port name to distinguish them from control instances.

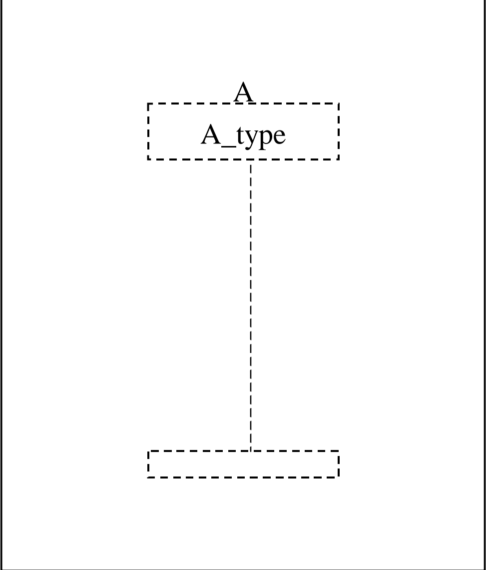


Figure 15: Explicit port instance

A port instance is graphically treated like an instance in MSC except of the specific port head, port instance line and port end symbols. In addition, the semantics of a port instance resembles the semantics of ports in TTCN-3: being a FIFO queue in the receiving direction. Port instances may use synchronous and asynchronous communication TSC constructs (indicated by an **async** and **sync** keyword respectively. In the instance head – a port instance without an explicit type definition and without this differentiation is assumed to be **async**) and may share in-line expressions and TSC references with test component instances. Furthermore, they may have special messages for test port handling (see clause 6.6).

Whenever there is no support for the special port instance symbol, the MSC instance symbol should be used with the keyword **port** in front of the port type, instead.

5.4.3 Implicit port instances

Figure 16 shows an implicit port instance. In fact, there are no graphical symbols for a port that is represented implicitly. It is rather a specific kind of denoting messages that are exchanged at a port. In case of a sending port, the message denotation is prefixed with the port identifier and the special symbol ">", whereas in case of a receiving port the message denotation is postfixes with ">" and the port identifier. Optionally, the port type can be given after the port identifier separated by a "**:**". The example given in figure 16 represents that message *m* is exchanged at port *A* of type *A_type*.

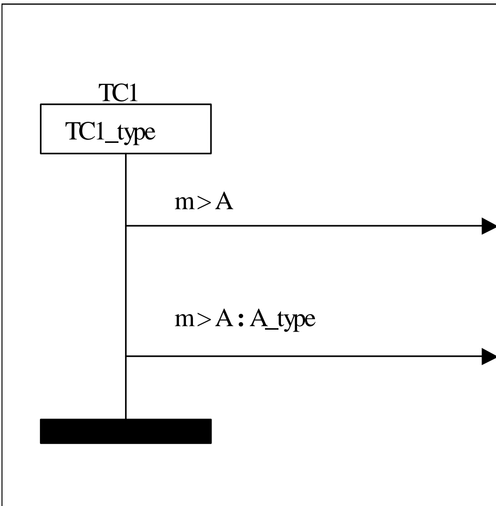


Figure 16: Implicit port instance (for mapped ports)

If two test component instances have ports which are connected to each other, the name of the sending port is given as prefix and the name of the receiving port as postfix to the message. In the case that the port names are equal, the postfix can be omitted.

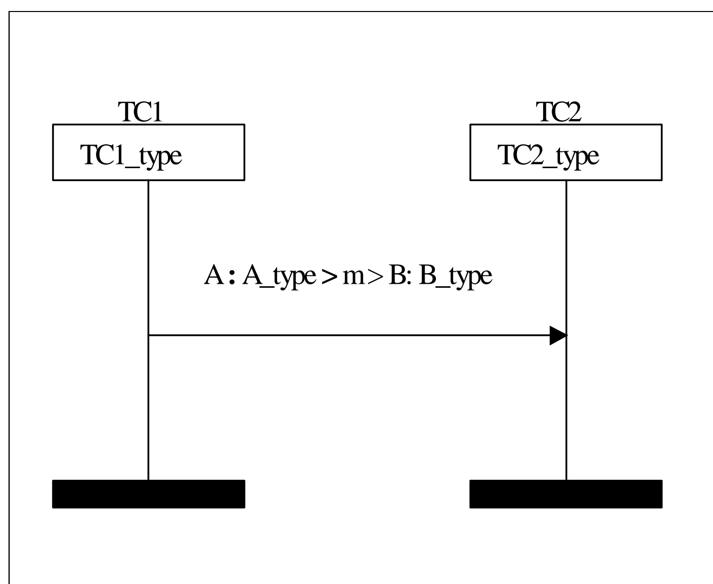


Figure 17: Implicit port instance (for connected ports)

5.5 Test component handling

The creation and stop of a test component is shown in the left-hand side of figure 18. The creation of a test component (see [1]) is represented by a dashed line that is connected to the instance head symbol. Whenever the create line is directed to a component instance no information shall be attached to it. Otherwise, the identifier of the test component and "**create**" are attached to the create symbol.

The stop of a test component, i.e. the termination of a test component (see [1]) is represented by a cross symbol attached to the instance line.

The right-hand side of figure 18 shows the graphical representation of connect, map, start and done operations for test components. Connect and/or map operations (see [1]) are given within an action box attached to the instance line of that test component that performs the connect or map operation. The syntax for the operations is taken from TTCN-3.

The starting of a test component (see [1]) is represented by a dashed arrow connected to the instance line of the started test component. The identifier of the test component and "**start**" are attached to the arrow symbol. Whenever the behaviour function of the test component is described implicitly (i.e. without an explicit TSC reference), the function identifier shall be given in parenthesis in addition.

The done operation (see [1]) is represented by a guarding condition indicated by a left and right brackets **[ ]** (optionally, the keyword **when** can be used for a guarding condition as it is done in MSC). The guarding condition has to be put at the beginning of an operand of an in-line expression, a branch of an HTSC, or a whole TSC. This reflects the typical use of the done operation in TTCN-3 within flow control statements like if-else and loop statements (see [1]).

creation and stop:

TC1

TC1_type

connect, map, start and done:

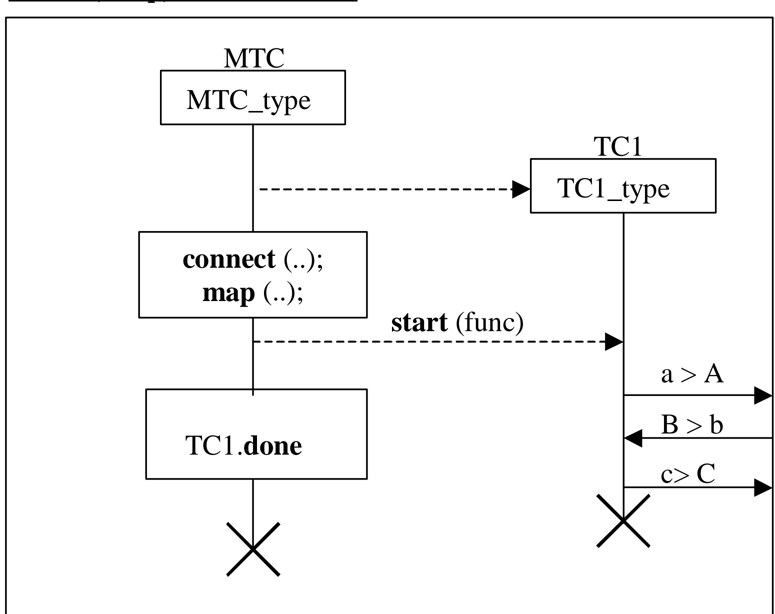


Figure 18: Test component handling

The treatment of done and start operations for test components are extensions to MSC. Whenever no support for this is given, action boxes have to be used instead.

5.6 Port handling

The representation of communication operations (see [1]) in TSC depends on the selected form for representing ports.

5.6.1 Handling of explicit ports

The left-hand side of figure 19 represents the operations for controlling communication ports, which are explicitly represented as port instances (see [1]). A dashed arrow pointing at the porting instance, upon which the operation is to be performed, is represented by a dashed arrow. The type of port operation (i.e. start, clear, and stop) is attached to this arrow. The syntax for the port operations is defined with TTCN-3.

The right-hand side of figure 19 represents a sending operation (see [1]) and receiving operation (see [1]). Sending and receiving operations are represented with a solid arrow having the port instance as destination or, respectively, as source. The value of information that is sent or received is attached to the arrow. The representation of this information is described in further details in clauses 6.10 and 6.11.

clear, start and stop:

A
A_type
clear
start
stop

send and receive:

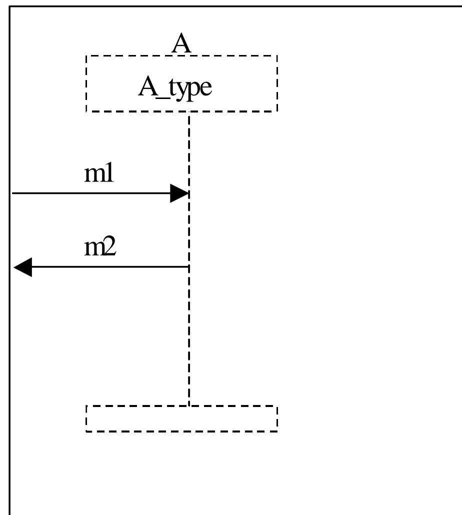


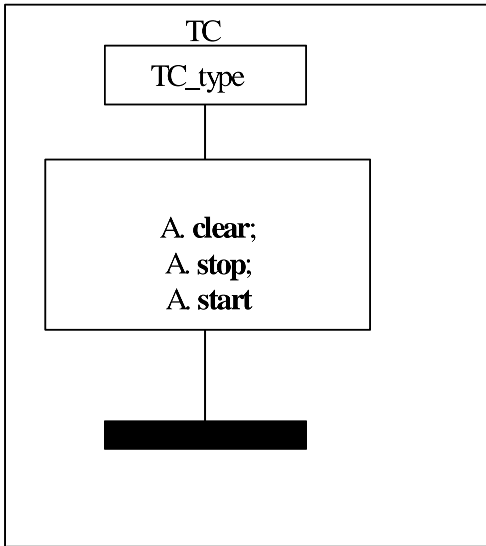
Figure 19: Handling of explicit ports

5.6.2 Handling of implicit ports

The left-hand side of figure 20 represents the operations for controlling communication ports that are implicitly represented (see [1]). In this case an action box containing the port operations (i.e. start, clear and stop), whose syntax is defined within TTCN-3.

The right-hand side of figure 20 represents a sending operation (see TTCN-3 [1]) and receiving operation (see [1]). Sending and receiving operations are represented with a solid arrow having the test component instance that performs the operation as source or, respectively, as destination. In case of a sending port the port identifier is attached to the arrow as a prefix (separated with the special symbol ">") to the message denotation. In case of a receiving port, the port identifier is attached to the arrow as a postfix (separated with the special symbol "<") to the message denotation. The representation of this information is described in further details in clauses 6.10 and 6.11. Optionally, the port type can be given in the prefix after the port identifier separated by a "**:**".

clear, start and stop:



send and receive:

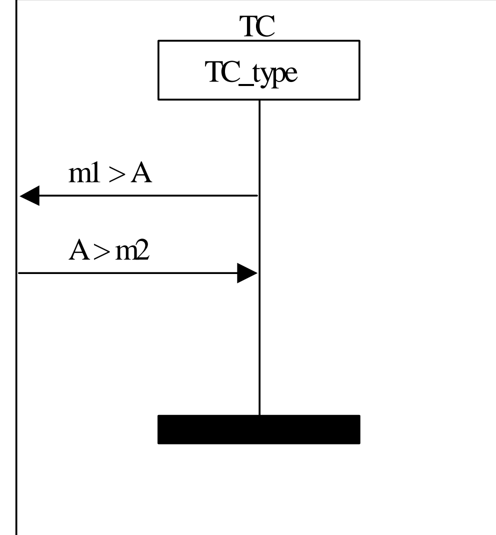


Figure 20: Handling of implicit ports

5.7 Configurations

Test configurations (see [1]) can be graphically represented only if the ports are represented explicitly by port instances:

1) a dashed line to represent the association of a test component with a port instance and;

2) a solid line to represent the connection between two port instances.

The association symbol is always attached to a test component instance and a port instance. The connection symbol is always attached to two port instances. It is used only if the ports of a test component are represented as separate port instances (see figure 21).

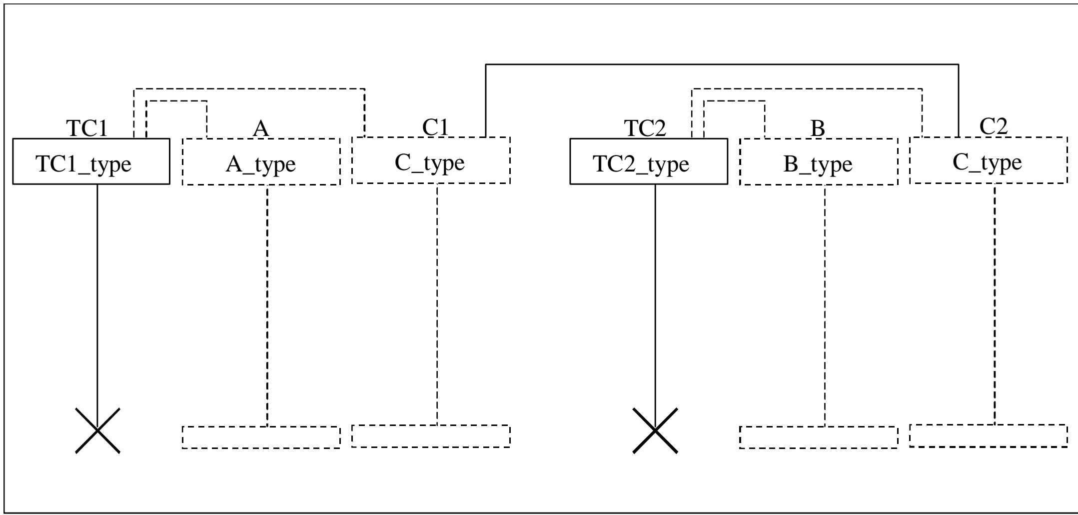


Figure 21: Representation of configuration (1)

Two connected ports may be represented by a single multi-port instance. The head of a multi-port instance is split into several parts by dashed vertical lines, thus allowing the specification of several port identifiers/types. In this case, the association of test component instances and port instances is used only (see figure 22).

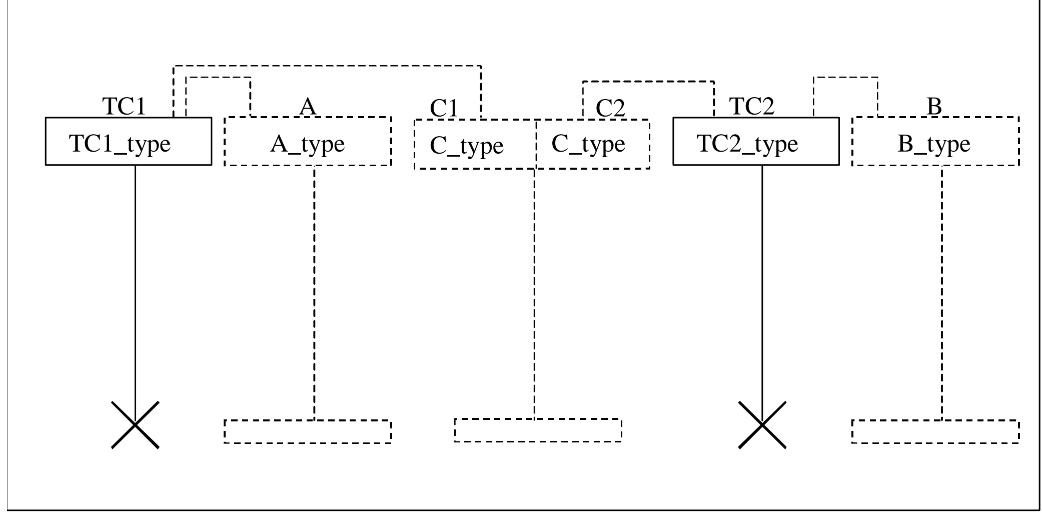


Figure 22: Representation of configuration (2)

The graphical symbols for associations and connections are new to MSC.

Whenever there is no support for these two symbols, the configurations cannot be represented graphically. Instead, connect and map operations are placed inside an action box.

5.8 Data

The approach adopted for incorporating data into Test Sequence Charts (TSC) has been taken according to MSC [2]. MSC introduced the novel concept of data parameterization allowing users to select the data language of their own choice. Consequently, TSC reuses this concept, with some minor modifications and extensions, as the basis for including TTCN-3 data types and values into the TSC language.

However, it must be noted that TSC should not necessarily be restricted to TTCN-3 data types or values. And in general, it should be possible to parameterize the TSC language with other data languages e.g. Java or C. This will be worked out in a later version of TSC.

Data is incorporated into TSC in a number of places, such as document parameters, control variables, verdicts, component and port instance headers, message passing, timers, action boxes, and TSC references. Data is used in two distinguishable ways: statically, such as in the parameterization of a TSC diagram, or dynamically, such as in the acquisition of a value through a message receipt. TSC has a number of points in the definition where a string representing some aspect of data usage is required, such as expressions or type declarations. In order to define the semantics of TSC with data a number of functions are required that extract the required information from such strings to interface to the TSC data concepts.

In clause 6.1 we define a set of semantic functions that are required for the use of TTCN-3 data types and values in TSC.

5.8.1 Declaring data

The declaration of data mostly takes place in a TSC document header clause 6.1, the only exceptions being:

- port and component types, which can be declared within instance headers;
- message types, which can be declared on a message arrow;
- control variables, which are declared local within control TSCs;
- global and component verdict variables, which are declared implicitly.

All declarations are given within the data definition part of the TSC document header. All types and values are taken directly from the TTCN-3 core notation.

5.8.2 Global data

In general TSC does not support global data. However, implicitly TSC assumes that a verdict variable is declared for each test case and test component, and that address variables exist for each component. Values for verdict variables are defined through the mechanisms given in clause 6.12.8, whereas access to address variables is given in clause 6.5.

5.8.3 Static data

Optionally, both TSC documents and TSCs can define formal parameter lists. A corresponding TSC reference must define a list of actual parameters whose scope is the TSC body. These parameters are treated as constants, whose values are determined at compile time. Consequently, when a parameter appears in the body, it must only be used in expressions that reference its value, hence cannot be modified dynamically.

5.8.4 Dynamic data

5.8.4.1 Component variables

Variables are owned by single component instances. This means that only the instance owning a variable can define its value through the use of bindings or expressions contained within action boxes or message parameters.

5.8.4.2 Bindings

A binding could be considered as an assignment as found in many programming languages. However, if wildcards are used, permitting under specification, these bindings become simple expressions. Again the notion of a binding is taken from MSC.

A binding consists of an expression part and a pattern part that are connected by a bind symbol. The bind symbol has left and right form both of which are equivalent, but which permit more natural reading of a binding associated with a message. Figure 23 illustrates a simple message interaction in which the variable x, owned by instance TC1, is assigned to an expression involving the variable y, owned by instance TC2. In all cases the value of a variable should be defined before it can be used.

```
tscdocument example1
explicit_typing;
type component TC1_Type { .. var x:integer; ..}
type component TC2_Type { .. var y:integer; ..}
```

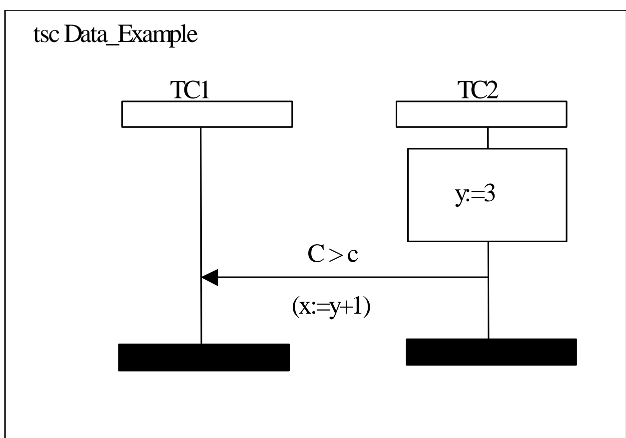


Figure 23: Example of a variable binding

5.8.4.3 Component initiation

A parameter list may be passed to a created component via the start message.

5.8.4.4 Data in messages

Dynamic data is permitted in messages via their parameter lists. A list of parameters may be bindings or expressions.

5.8.4.5 Action boxes

Data can appear inside action boxes as a semi-colon separated list of TTCN-3 statements. Such statements are permitted to use variables, which are owned by the component instance on which the action is placed, or formal parameters.

5.8.4.6 Component and port types

The introduction of component and port types is an extension to the notion of instance kind in MSC. Component and port types define the type and direction of data that can be communicated by a component over a port. If explicit typing is used (see clause 6.1.2.2) then all communication events must be consistent with associated component and port types.

5.8.5 Guarding conditions

Guarding conditions in TSC are used to guard choices in alternatives and for the definition of upper bounds in loops. The Boolean expression placed within a guarding condition is put into brackets. However, in order to keep compatibility with MSC, the guard can also be preceded with the keyword **when**.

5.8.6 Setting conditions

Setting conditions in TSC are used to set test verdicts. Test verdicts can have one of five values with the keywords **pass**, **fail**, **inconc**, **none**, and **error** (see clause 6.13).

5.9 Timers

As illustrated in figure 24 the setting, stopping and timeout of a timer are using different graphical symbols. These events are used to represent the start, stop, and read timer operations and the timeout event (see [1]).

The graphical symbol for a timer looks like an hour-glass. It is attached to the test component instance line with a line (in the case of the start timer operation) or with an arrow pointing at the test component instance (in the case of a timeout event). The stop timer operation is represented by a cross, which is attached to the test component instance line. The graphical symbols are associated with further information like the timer identifier or the timer duration.

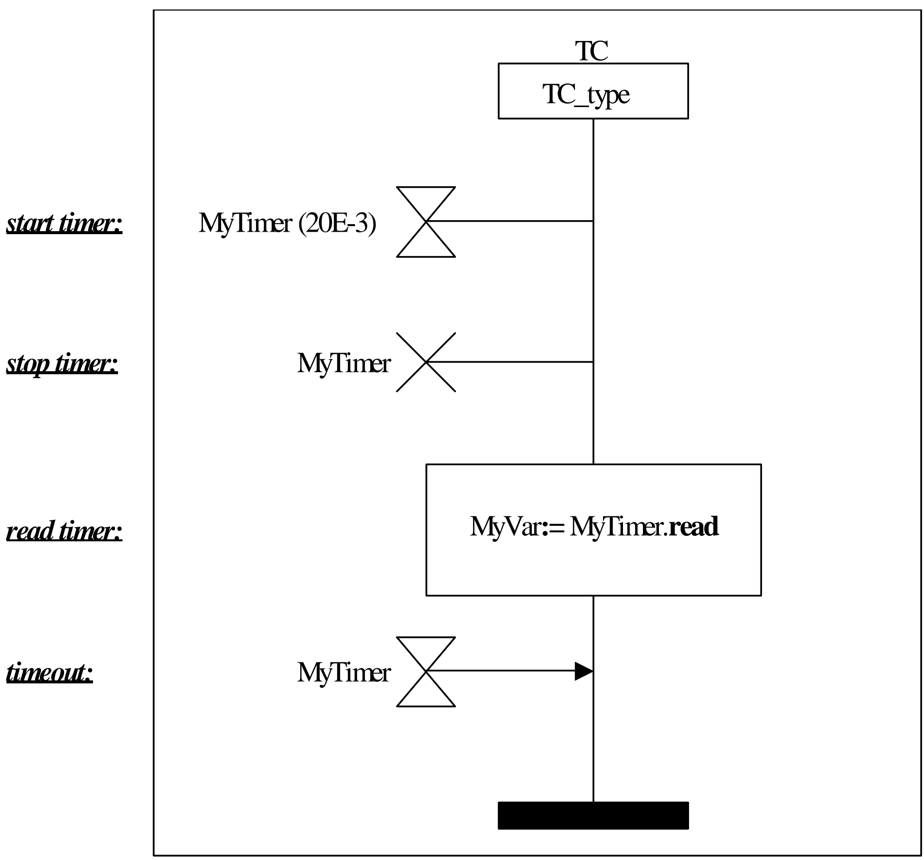


Figure 24: Timer representation

The graphical symbols for timer operations can be combined to give a more compact representation for the timer handling as shown in figure 25.

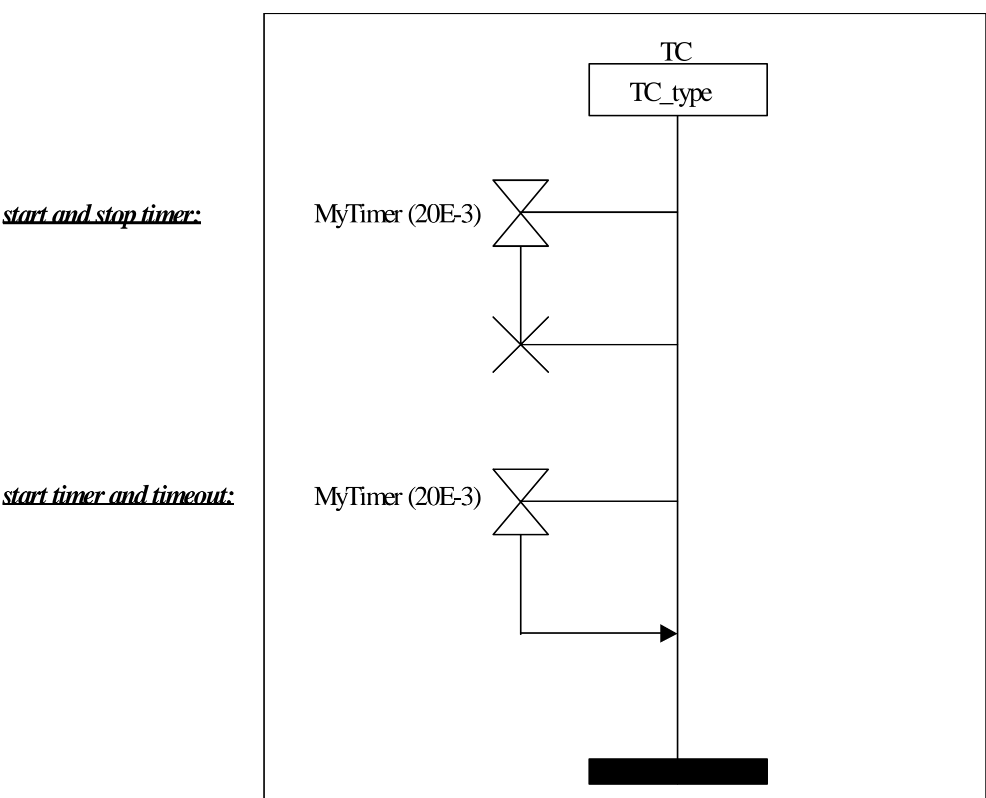


Figure 25: Condensed timer representation

A timer in TSC is treated like a timer in MSC except of the addition of the read timer operation.

5.10 Asynchronous communication

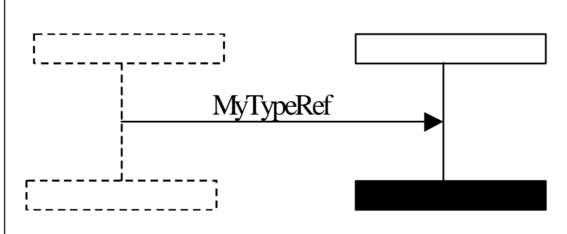
Asynchronous communication comprises of the **send**, **receive**, **check** and **trigger** operations (see [1]). In general, a message arrow represents the sending and receiving of a message from or to a component instance. The sending of a message by a component to it respective port represents a **send** event, and the receipt of a message by component from a port represents a **receive** event.

In TSC, messages are communicated via ports each representing a First-In First-Out (FIFO) buffer. This means that messages cannot overtake each other.

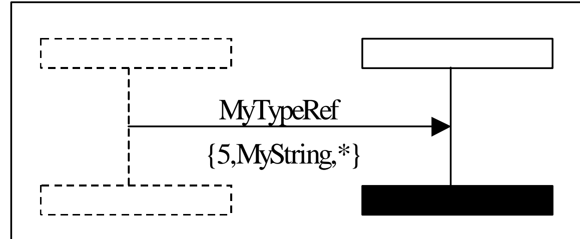
5.10.1 Messages

In TTCN-3 a message must have a value. Therefore, the type of each message must be declared within the TSC document header. The value of messages is defined using templates or in-line templates (see [1]). A template is declared within the TSC document header. In TSC the type of a message is given either by the message name or implicitly through its template reference, if an in-line template is not used. Figure 26 illustrates the various combinations of message usage within TSC.

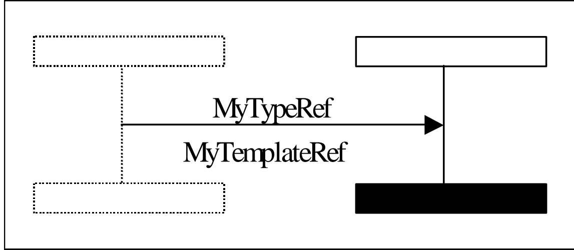
Type Only



Type and In-line



Type and Template



Template Only

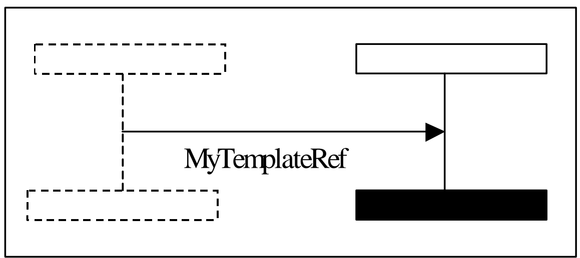


Figure 26: Combinations of type and template usage for TSC messages

In the case where no template is given the message assumes that any value can be received. When receiving a message the **value** keyword can be used to store its value in a local variable. Figure 27 illustrates a receive operation in which the value of the message is stored in variable x.

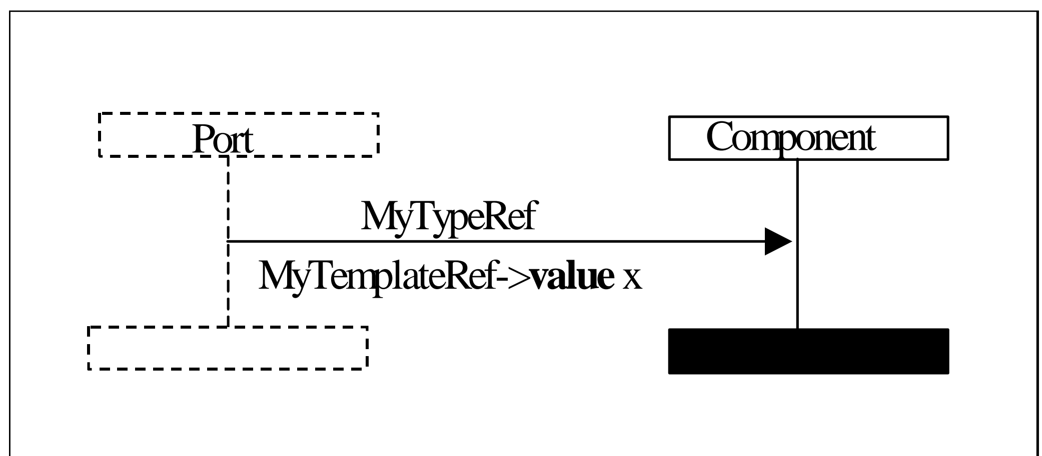


Figure 27: Storing the value of a received message

5.10.2 Receiving any message

Graphically a message arrow labelled with the **any** keyword shows the receipt of any message. Figure 28 illustrates an example of the **any** keyword.

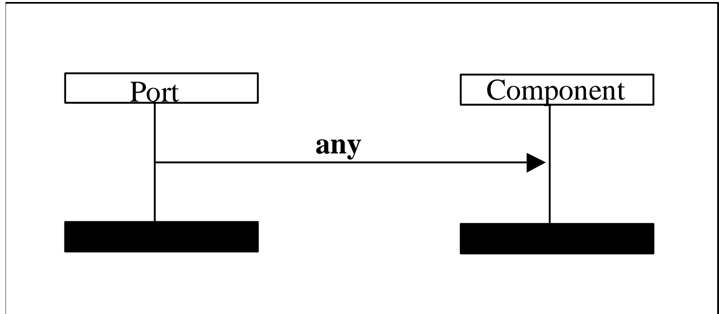
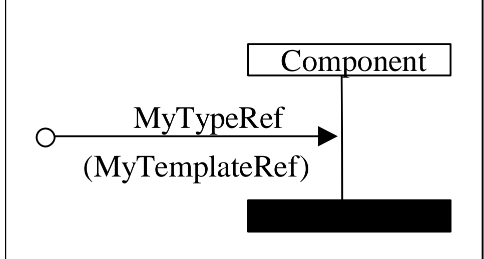


Figure 28: Example receipt of any message

5.10.3 Receiving from any port

In general the receipt of messages from any port is illustrated using a message coming from a found symbol. Figure 29 illustrates two examples of receiving a message from any port. On the left-hand side a message is shown coming from a found symbol. This mechanism is used for the no vertical split. On the right-hand side the **any** keyword is used as an implicit port.



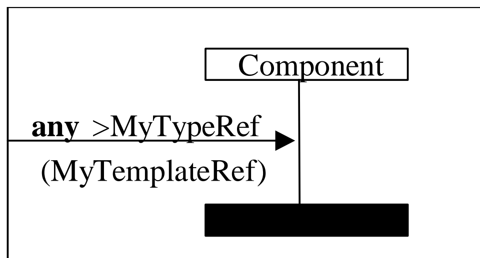


Figure 29: Examples of receiving a message from any port

By labelling a message using the **any** keyword it is possible to receive any message from any port.

5.10.4 Trigger message

A message arrow with the **trigger** keyword placed above the line represents the trigger operation. The template for the trigger operation is placed under the line as defined within TTCN-3.

5.11 Synchronous communication

Synchronous communication comprises the send operations **call**, **reply**, **raise** and the receive operations **getcall**, **getreply**, **catch** (see [1]). Synchronous communication operations are declared within an TSC document in the declaration part. Their use is represented as events of the respective test component instance.

5.11.1 Call, getreply, catch, and timeout

The call operation is graphically represented by a message arrow starting from a component instance. Above the message arrow, the keyword **call** followed by the signature is placed. The signature template is placed underneath the message arrow. The call operation may be followed by a start timer operation represented by the start timer symbol together with the timer duration. The call symbol is attached to a subsequent blocking area on the test component. The blocking area is graphically represented by a tall thin rectangle with dashed vertical border lines. The blocking area finishes at the reception of a getreply, exception or timeout.

The getreply operation is graphically represented by a message arrow pointing towards a component instance. Above the message arrow, the keyword **getreply** followed by the signature is placed. The signature template is placed underneath the message arrow. Underneath the signature template, the binding of parameter values to variables is placed. The binding begins with - > (see [1]). The getreply symbol is attached to the end of the blocking area (started by a call message) on the test component.

The catch operation is graphically represented by a message arrow pointing towards a component instance. Above the message arrow, the keyword **catch** followed by the signature is placed. The signature template is placed underneath the message arrow. Underneath the signature template, the binding of parameter values to variables is placed. The binding begins with - > (see [1]). The catch symbol is attached to the end of the blocking area (started by a call message) on the test component. Multiple exceptions within alternative expressions are allowed.

The timeout is represented by a timeout symbol like in MSC. The timeout symbol is attached to the end of the blocking area on the test component.

In case of a non-blocking call [1], the call operation is not attached to a subsequent blocking area. Instead, a normal instance line replaces the blocking area.

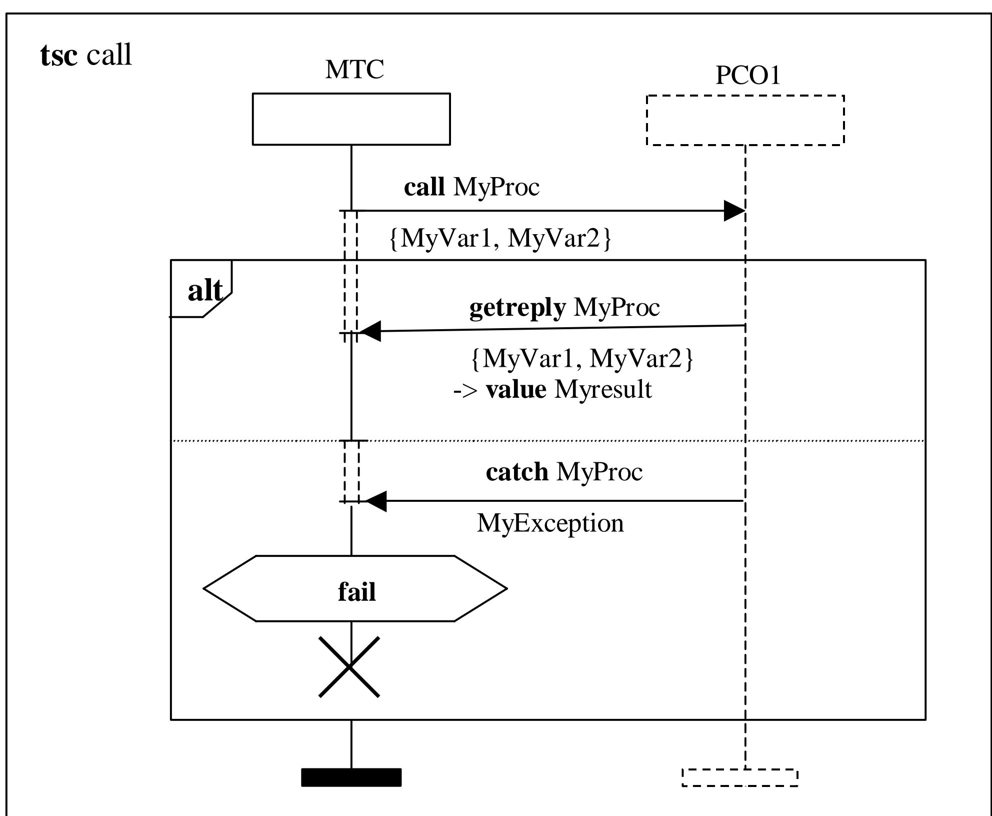


Figure 30: Blocking call with reply and exception

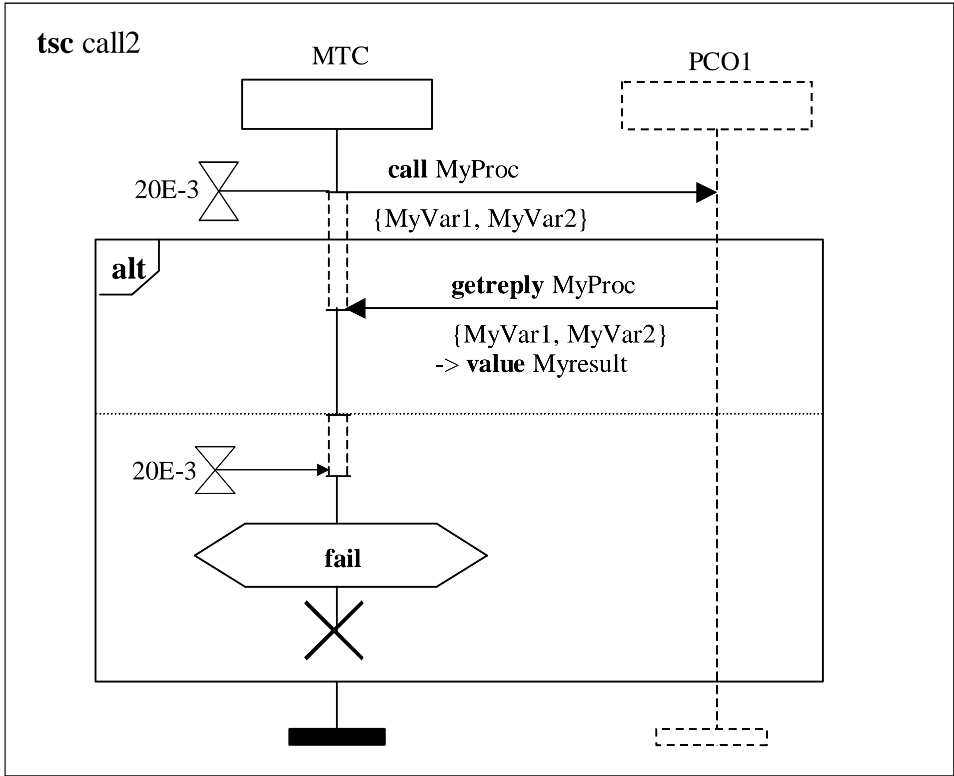


Figure 31: Blocking call with reply and timeout

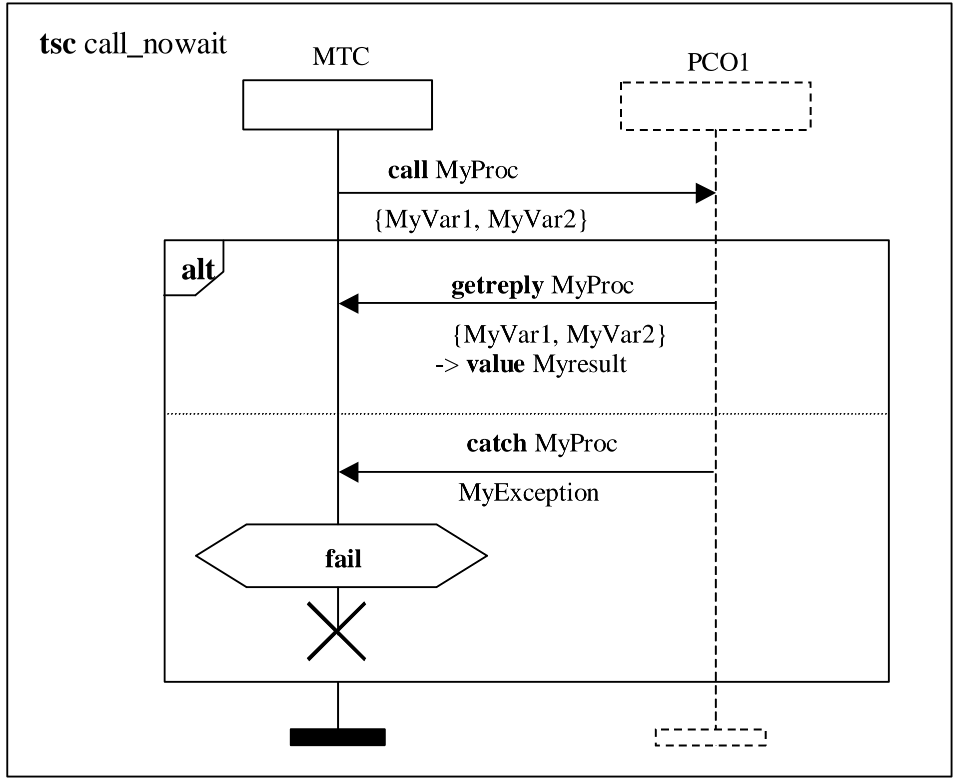


Figure 32: Non-blocking call with reply and exception

5.11.2 Getcall, reply, and raise

The getcall operation is graphically represented by a message arrow pointing towards a component instance. Above the message arrow, the keyword **getcall** followed by the signature is placed. The signature template is placed underneath the message arrow. Underneath the signature template, the binding of parameter values to variables is placed. The binding begins with - > (see [1]). The getcall symbol is attached to a subsequent activation region on the test component. The activation region is graphically represented by a tall thin black filled rectangle. The activation region finishes at the reception of a reply or raise.

The reply operation is graphically represented by a message arrow starting from a component instance. Above the message arrow, the keyword **reply** followed by the signature is placed. The signature template is placed underneath the message arrow. The reply symbol is attached to the end of the activation region (started by a getcall message) on the test component.

The raise operation is graphically represented by a message arrow starting from a component instance. Above the message arrow, the keyword **raise** followed by the signature is placed. The signature template or an expression is placed underneath the message arrow. The **raise** symbol is attached to the end of the activation region (started by a getcall message) on the test component.

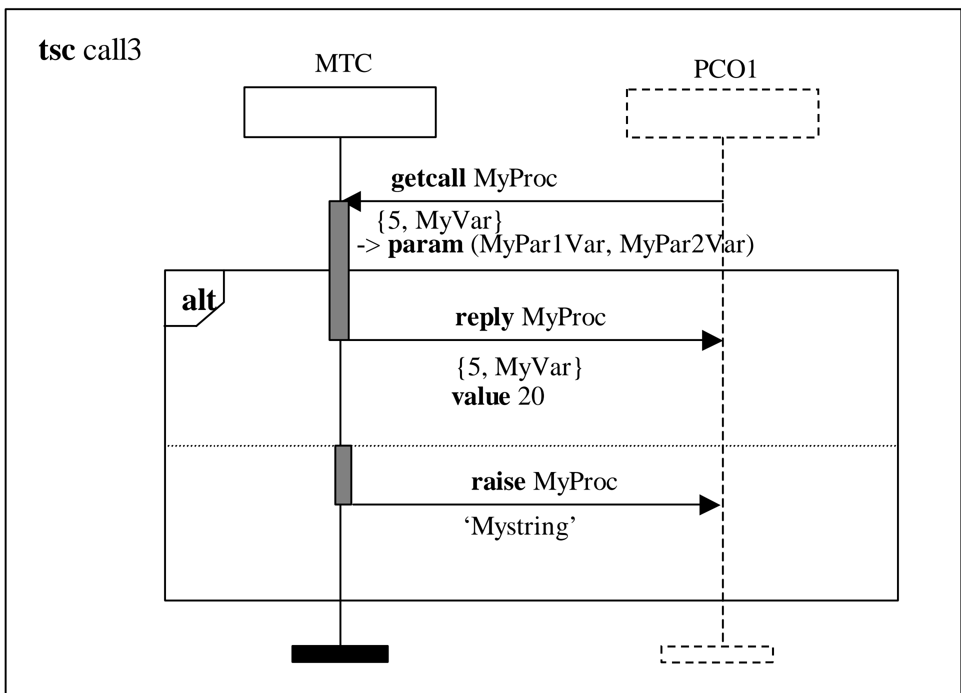


Figure 33: Incoming call with reply and exception

5.11.3 Any handling

Accepting any call is represented by a message arrow with **getcall any** above.

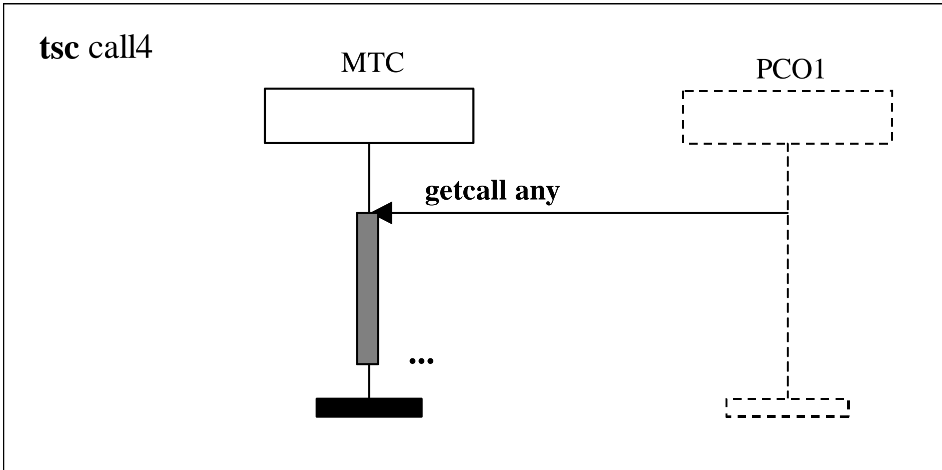


Figure 34: Incoming any call

Getcall on any port is represented by a found message with **getcall** followed by the signature above and the signature template underneath.

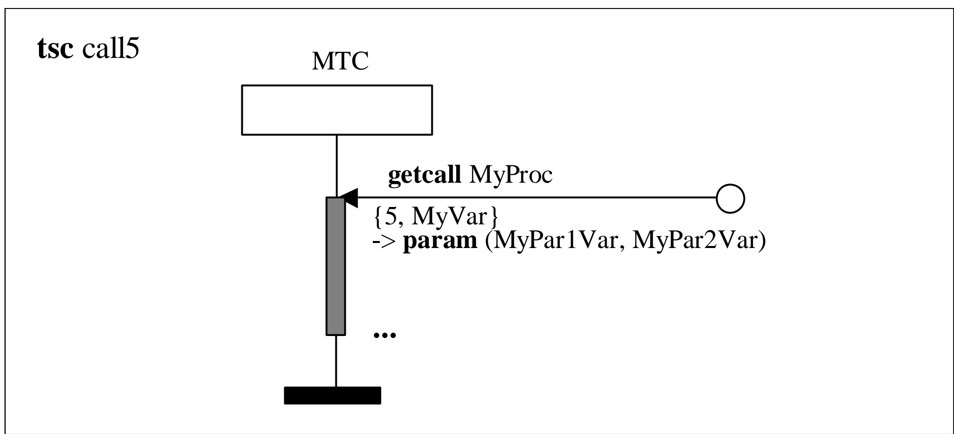


Figure 35: Incoming call from any port

Get any reply from a call is represented by a message arrow with **getreply** followed by the signature and the keyword **any** underneath.

Get any reply from any call is a message arrow with **getreply** followed by the keyword **any** above and the keyword **any** underneath.

Get a reply from any port is represented by a found message with **getreply** followed by the signature above and the optional template underneath.

In all cases above, if the port is presented explicitly then the message arrows are drawn from the port instance (in the case that a concrete port is used). If the ports are represented implicitly, then the port is put in front of getcall and getreply (separated by a ">" symbol).

Catch any exception is represented by a message arrow with the keyword **catch any**. Catch on any port is represented by a found message with the keyword **catch** above.

5.12 Behaviour

The behavioural program statements (see [1]) cover sequential, alternative, interleaved, default behaviour, and the return statement. Their representation in TSC is discussed in the following clauses.

5.12.1 Sequential behaviour

Sequential behaviour (see [1]) is implicitly represented as subsequent behaviour on the instance axis of the test component, which performs the sequential behaviour. For example, the order in which events are placed on the instance axis is the order in which they occur.

Figure 36 represents that the test component TC1 sends in sequence the messages a and b.

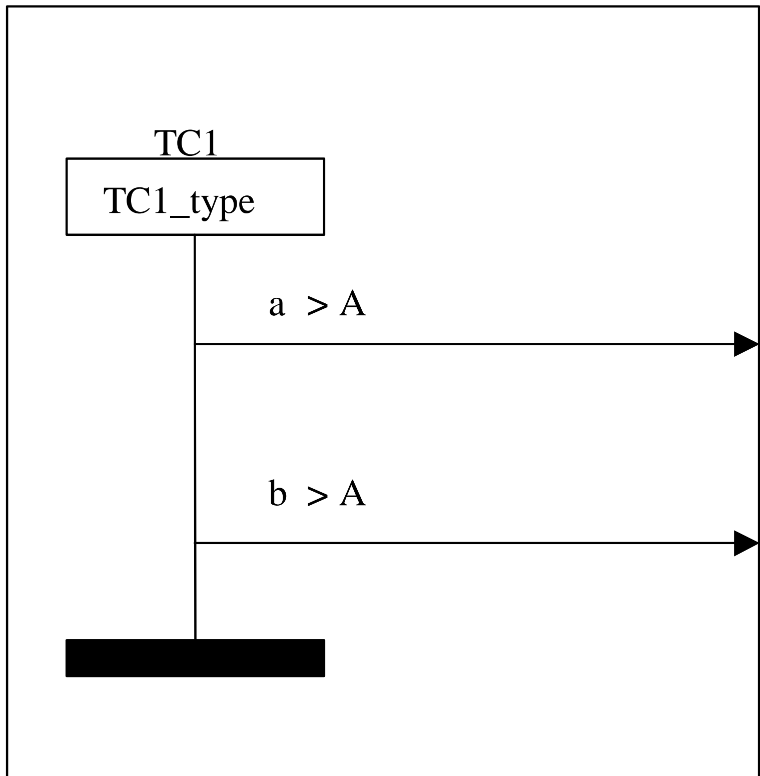


Figure 36: Representation of sequential behaviour

Sequential behaviour in TSC is treated like sequential behaviour in MSC.

5.12.2 Alternative behaviour

Alternative behaviour (see [1]) is represented in the form of an alternative in-line expression (alternatives can also be represented by an HTSC with choice and guards or, in the HyperTSC form with the enhanced graphical means to represent alternatives. The exception in-line expression may be used as a shorthand notation for an alternative in-line expression with the exception part as the first operand and the rest of TSC as the second operand. The alternative in-line expression is represented in a box, indicating in the left upper corner with the keyword **alt** that it is an alternative expression. The alternatives are separated by a dotted line. Guarded alternatives use guarding conditions. The else case is guarded with an **else** condition. In order to keep compatibility with MSC, the **otherwise** keyword can be used instead. Named alternatives are represented by TSC references with a prefix [**expand**], which refer to the TSC representing the named alternative. A TSC that is used as a named alternative, referenced within an **expand**, must have an alternative expression on top level.

Within an alternative in-line expression, only local conditions are allowed, but alternative in-line expressions may be non-local, i.e. covering multiple components. Each component has either only operands with receiving events or with sending events as first events or an operand may be empty. Sending events must be guarded with deterministic guarding conditions whereas receiving events may be guarded with guarding conditions. The interpretation of an alternative in-line expression across multiple components differs from MSC. Non-local alternative in-line expressions covering multiple components are interpreted as independent local alternative in-line expressions for each component. We allow non-local alternative in-line expressions as a convenient shorthand notation and a possibility to avoid gates.

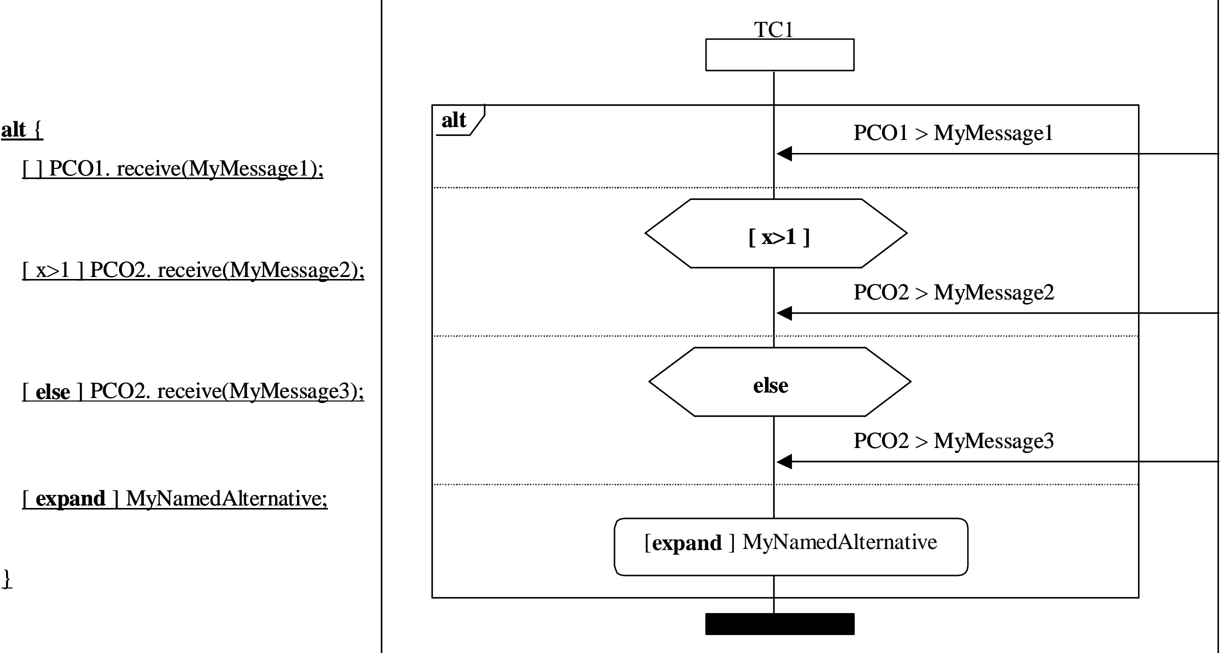


Figure 37: Representation of alternative behaviour (with implicit ports)

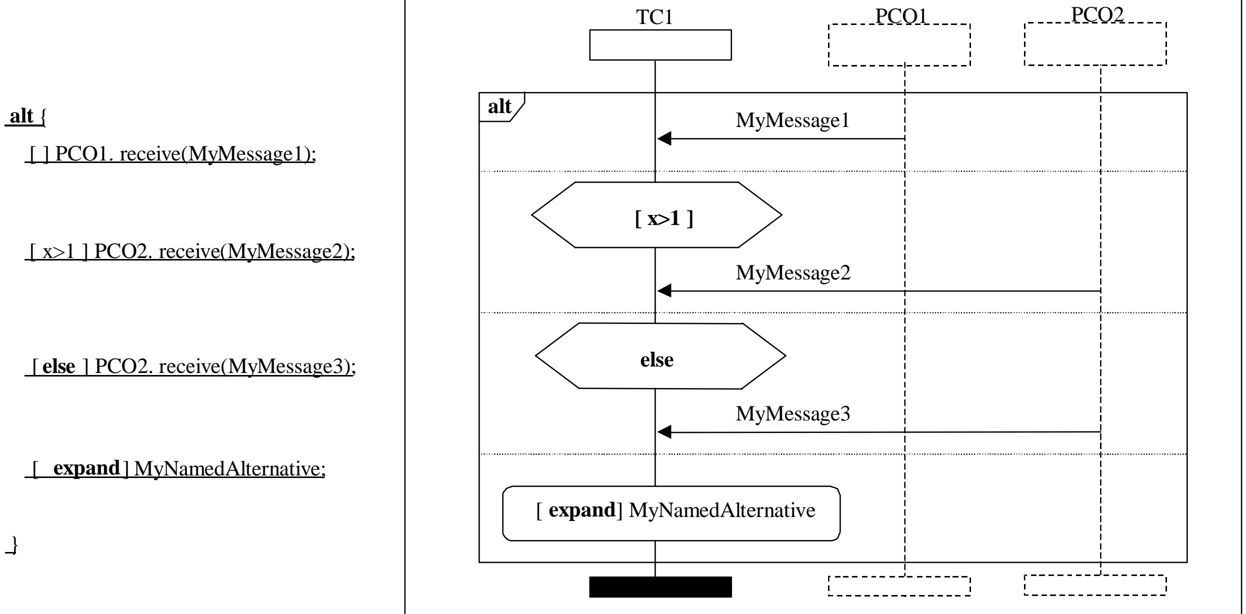


Figure 38: Representation of alternative behaviour (with explicit ports)

Alternative behaviour is either represented within alternative in-line expressions as it is done in MSC or within HyperTSC, which is an extension to MSC. In addition for user convenience, brackets and the keyword **otherwise** can be used within guarding condition.
Alternatives get an additional semantic interpretation: they are interpreted from top to bottom within alternative in-line expressions and from left to right within HyperTSC. Whenever this is ambiguous, a circled number can be put into the TSC reference box of a HyperTSC to indicate the priority level for that alternative. The alternatives are evaluated from smallest to highest numbers.

Whenever there is no support for HyperTSC, alternative in-line expressions can be used only.

5.12.3 Interleaved behaviour

The **`interleave`** statement of [1] allows the specification of interleaved occurrences of sequences preserving the deterministic ordering of sending events. This statement is represented in TSC with an in-line expression having in the upper left side box the keyword **int**.

In agreement with TTCN-3, interleave expressions are not allowed within operands of alternative expressions and interleave expressions may not be guarded. Therefore, initial events for each operand must be a reception statement, i.e. receive, trigger, getcall, getreply, catch, check.

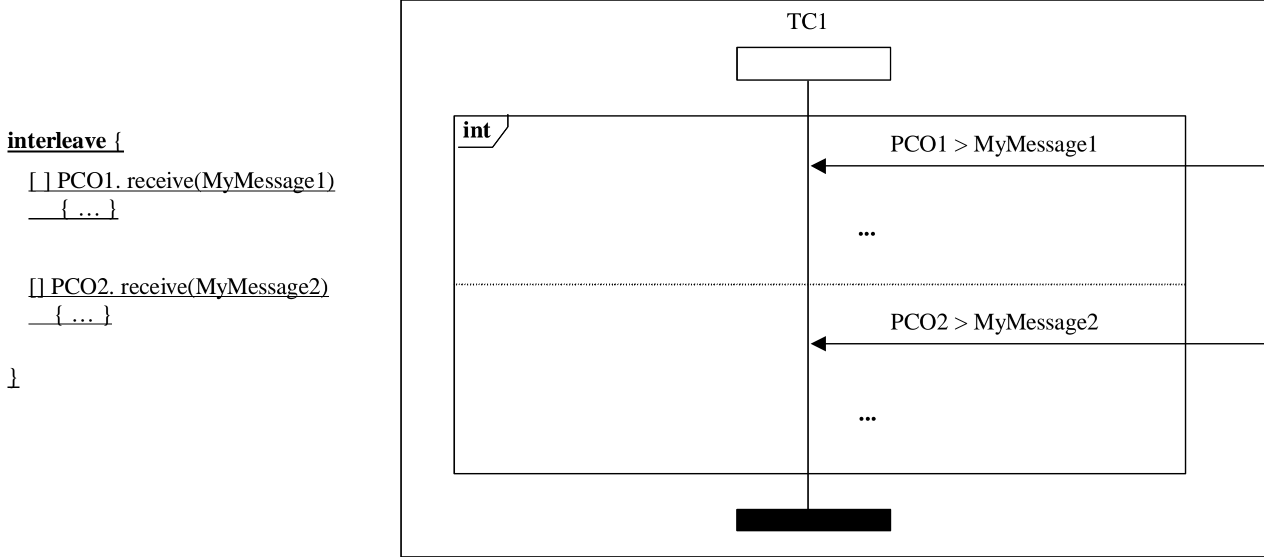


Figure 39: Representation of interleaved behaviour

The interleave in-line expression is an extension to MSC. It is like any other MSC in-line expression except that it uses the keyword **int** to indicate that it represents interleaved behaviour. Simple interleaved behaviour can be represented in TSC by the use of co-regions.

5.12.3.1 Co-regions

Co-regions may be used in TSC instead of interleave in-line expressions for the simple case where only reception events are interleaved, i.e. only reception events (receive, trigger, getcall, getreply, catch, and check) may be placed within a co-region.

5.12.4 Loops

The **for**, **while**, **do while** statement of (see [1]) allows the specification of cyclic behaviour. In TSC, we support the **for** and **while** statement only. These statements are represented in TSC with a loop in-line expression having in the upper left side box the keyword **loop** (loops can also be represented by an HTSC with cyclic connection lines and guards or, in the HyperTSC form with the enhanced graphical means to represent loops).

In TSC, loops must have a lower boundary. If the lower boundary is not reached or if the lower boundary is greater than the upper boundary then an error verdict is given. Within loop in-line expressions, only local conditions are allowed, but loop in-line expressions may be non-local.

A loop exits, if the upper boundary is reached or if a guard becomes false.

The interpretation of in-line loop expressions across multiple components differs from MSC. Similarly to in-line alternative expressions, non-local loop in-line expressions covering multiple components are interpreted as independent local loop in-line expressions for each component. We allow non-local loop in-line expressions as a convenient shorthand notation and a possibility to avoid gates.

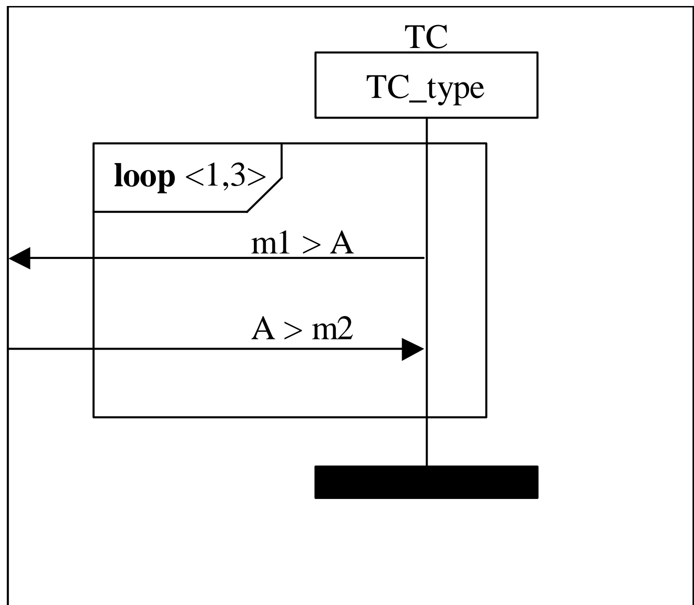


Figure 40: Loop behaviour

5.12.5 Functions

Functions are used in TTCN-3 to express test behaviour or to structure computation in a module. Functions may return a value. [1]. In TSC, functions are represented by means of TSC references. Binding expressions allow the values returned by TSC test cases to be assigned to a local variable. Note, that values cannot be returned by a function (TSC reference) crossing multiple components.

5.12.6 Defaults

Defaults (see [1]) are represented by separate TSCs and activated and deactivated by referring to those TSCs within a special graphical symbol for defaults. The graphical symbol for representing defaults is a parallelogram, which is attached to the test component instance line. Inside the graphical symbol, the keyword **activate** or **deactivate** followed by the default behaviour in parenthesis are given (as it is done in the TTCN-3 syntax for the activate and deactivate operations). Named alternatives may take parameters. A list of named alternatives may also be given to the activate operation. In this case, the named alternatives are expanded in the order given.

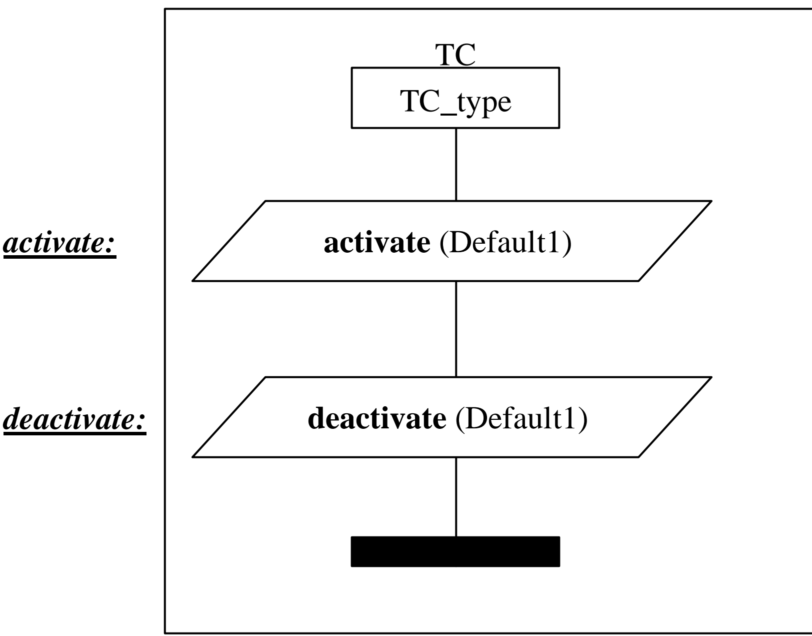


Figure 41: Default representation

The default symbol is an extension to MSC for the representation of TTCN-3 defaults. Whenever the new graphical symbol is not available, an alternative representation would be the action box having the activate or deactivate operation inside.

5.12.7 Return statement

Return statements (see [1]) for behaviour functions (i.e. those with a **runs on** keyword in the function header) (see note) are represented by the instance stop symbol. If a return value is associated, an additional dashed arrow starting at the instance line of the returning test component is used. The **return** keyword and the return value are attached to the arrow symbol. The type identifier for the return is given after the **return** keyword of the TSC header.

NOTE: Evaluation functions are not represented graphically in TSC, but are contained in the TSC data part with their textual TTCN-3 definition.

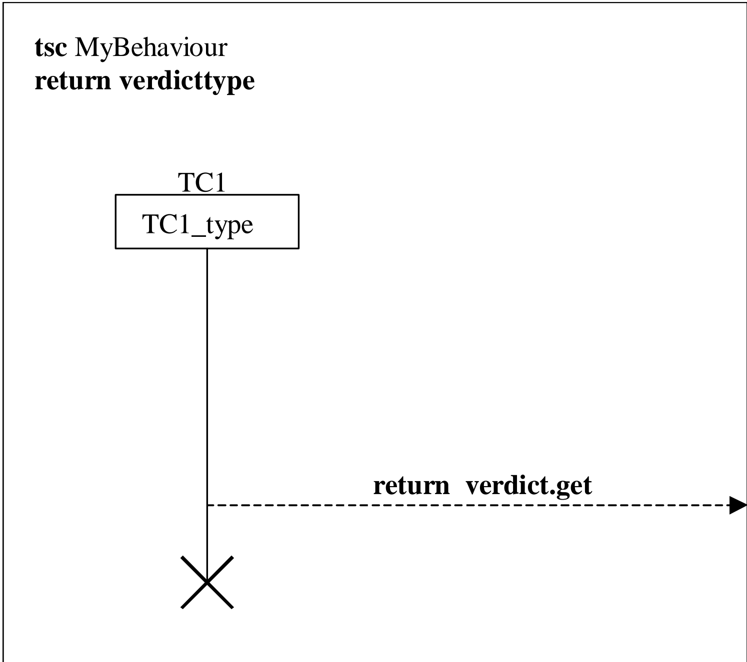


Figure 42: Return representation

The treatment of return values for behaviour functions is an extension to MSC: both the additional keyword **return** in the header information as well as the dashed arrow for the return value extend MSC. If this extension is not supported an action box for the return of a value is used instead.

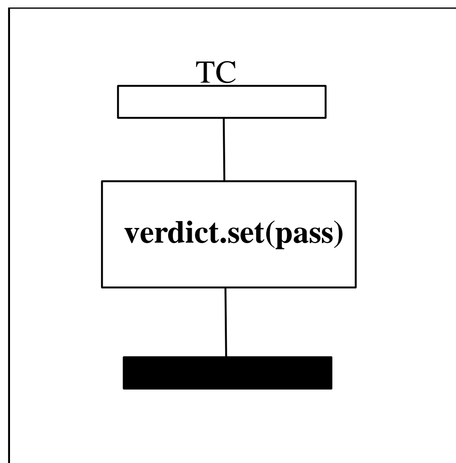
5.12.8 Action Boxes

Apart from assignments, action, boxes may contain TTCN-3 statements like the connect, map or log operation.

5.13 Verdicts

Verdict operations allow test components to set and retrieve local verdict values. Each test component maintains its own local verdict, which is created for each test component at the time of instantiation. All verdicts can have one of five values **pass**, **fail**, **inconc**, **none**, and **error** where, **inconc** means an inconclusive verdict. In addition to local verdicts there is a global verdict for each test case. The value of this verdict is returned by the test case upon termination. The value of the global verdict is defined by the values of all test components within a test case (i.e. MTC and every PTC). The rules for determining the global verdict is determine using the overwriting rules defined in [1].

A local verdict can be set in one of two ways: (1) using the verdict operation **verdict.set** (value), or (2) using a local condition labelled with one of the following keywords **pass**, **fail**, **inconc**, and **none**. Figure 43 illustrates the setting of a local verdict variable.



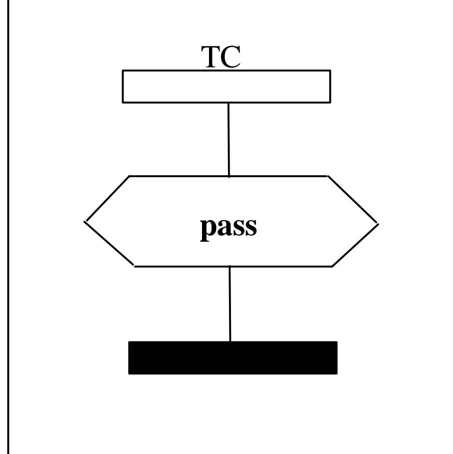


Figure 43: Operations that can be used to set a local verdict

The value of a local verdict variable can be retrieved using the operation **verdict.get**. Figure 44 illustrates the assignment of the local verdict to a local variable.

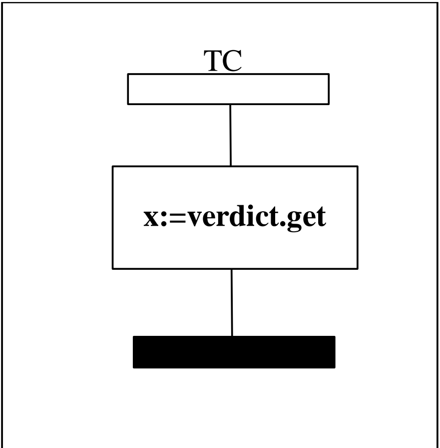


Figure 44: An example of how the local verdict variable can be retrieved

5.14 High-level TSC (HTSC)

High-level TSCs are used in the control part of the TSC document (see clause 6.2) or may be used for representing alternative or cyclic behaviour (see clause 6.15) within test cases. HTSCs provide a means to graphically define how a set of TSCs can be combined. HTSCS allow the description of sequential, alternative and parallel composition. Guarding conditions (see clause 6.8.5) are used to guard choices and loops in HTSCs. HTSCs used for the control part of the TSC document have the following extensions to MSC [2]: formal parameters, local variables, action boxes containing data expressions , guarded conditions, and references to TSC test cases that return values. Some restrictions have been applied to the usage of MSC reference expressions. For test case modelling, HTSC contains few extensions concerning the parallel composition: The parallel frame (rectangle) may be omitted in case where it is redundant and no ambiguities may occur. In order to explicitly indicate the connection of ports between different reference symbols in HTSCs, a double line arrow with the port names above can be used. A dashed double line arrow can be used to indicate a create/start operation between references.

NOTE: the same connection and create/start construct can be used also between TSC references contained in plain TSCs.

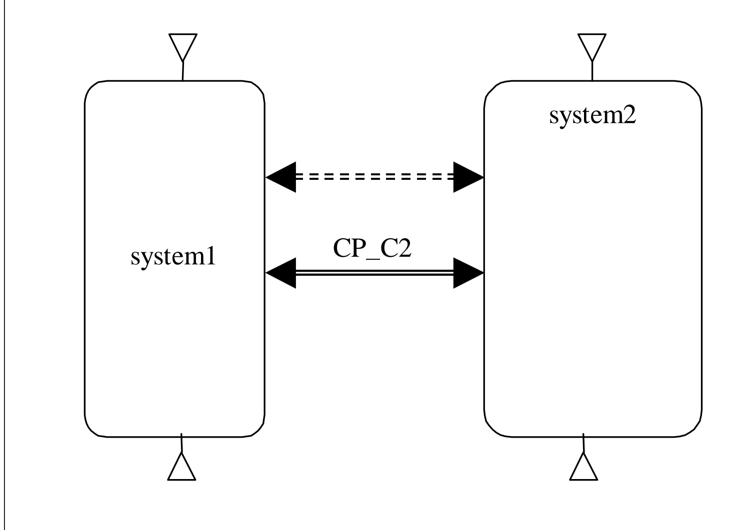


Figure 45: HMSC with parallel composition and connection, create/start symbol

5.15 Hyper_TSC

The most obvious and straightforward way to represent TTCN-3 test cases by TSC diagrams is to use in-line operator expressions for alternatives, iterations etc. Depending on the test verdict, the representation of alternatives by means of the exception operator may be more suitable (see clause 6.12.2). Practice has shown that apart from simple cases using such in-line expressions may lead to diagrams, which are not so easy to read and to understand. As a consequence, it is not clear whether such a TSC format would mean a progress with respect to the TTCN-3 notation or even a step back. In particular, in-line operator expressions obscure the message flow of the "standard" cases (pass verdict) by mixing it with alternative parts. As a rule, in-line expressions should be used only in a very limited manner and should be restricted to a few alternatives or loops.

In more complex situations, HTSCs are much more transparent since they abstract from details and focus on the compositional structure. However, if standard HTSCs are used instead of in-line expressions it has the immediate drawback that the representation appears in a fairly indirect manner. Standard HTSCs consist of TSC references, which point to TSC definitions by means of reference names. In order to overcome this deficiency, an expanded form of TSC references within HTSCs is admitted. This is a real extension to the MSC language. If a "mixed" representation is allowed where some HTSC references appear in expanded form and some not, a very flexible notation is obtained (see figure 46). In particular, it provides a means to single out the normal (pass-) case. The TSC references referring to the pass-case may be shown in expanded form, the others as non-expanded TSC references. An immediate generalization of this idea is to allow also representations where inconclusive- and fail-cases are singled out by changing the roles.

A test case representation by means of HTSCs where parts of the TSC references appear in expanded form still has the disadvantage that the non-expanded TSC references do not provide sufficient information without looking at the corresponding explicit TSC definitions in the TSC document. This is particularly inconvenient in case where many small TSC definitions are used, which is just typical for test case descriptions. To make HTSCs applicable to test cases, an additional extension is necessary. It seems to be more appropriate to interpret the TSC referencing mechanism in a hypertext-like manner assuming a corresponding tool support where the TSC references can be expanded within the embedding HTSC or possibly also in a separate window. The TSC references, which can be expanded may be indicated by underlining the text, by coloured text or by a variation of the line width of the symbol lines.

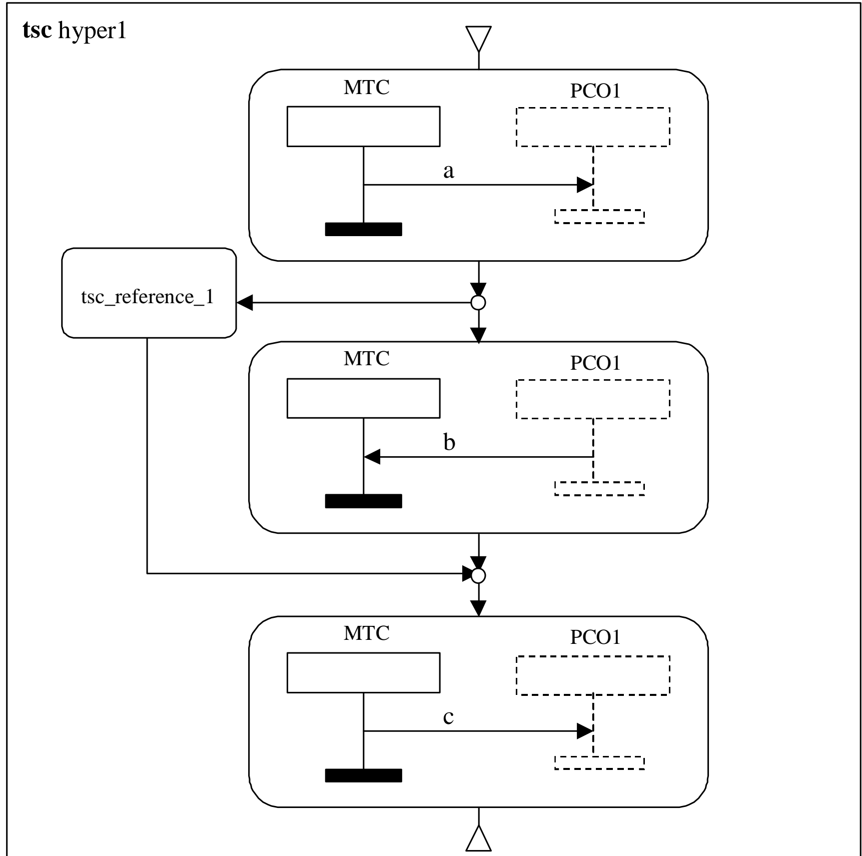


Figure 46: Combined use of Hyper TSC and TSC references

Eventually, such a coherent expanded representation of a whole path should be possible not only in a separate window but also in in-line-form within the HTSC itself. This is advantageous in particular, to show the pass-case in a coherent manner since in case of many alternatives, a splitting is very disturbing. Therefore, a further extension of HTSCs is introduced which somehow may be viewed also as a unification of HTSC and BTSC (basic TSC). Several expanded TSC references may be combined to one coherent expanded TSC reference. As a consequence, the connection points have to be shifted to the border line of the resulting TSC reference. This procedure is illustrated in figure 47.

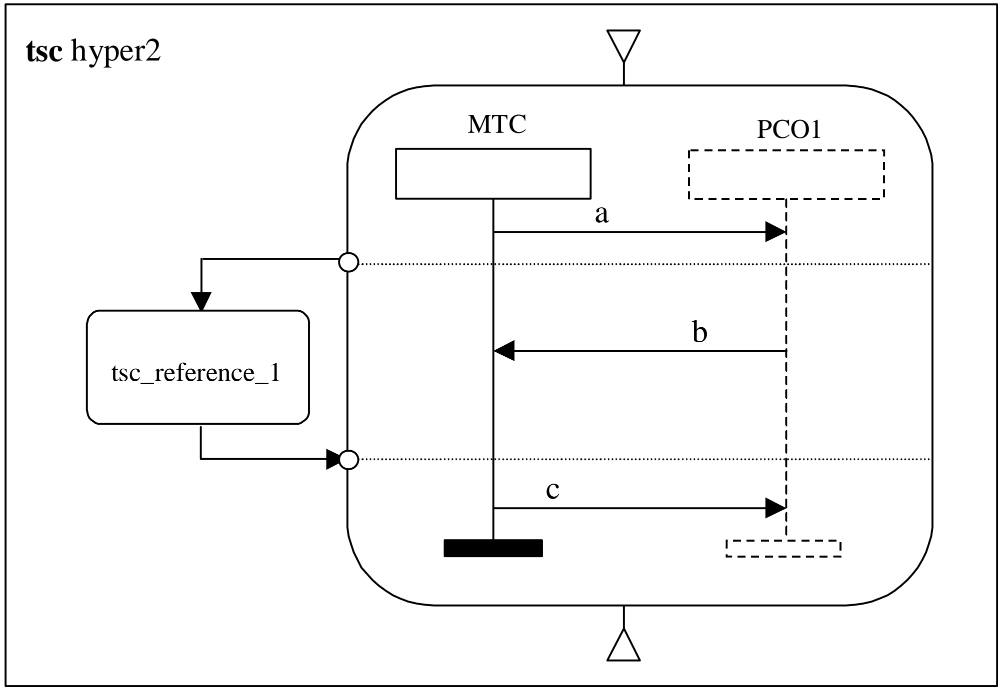


Figure 47: Condensed representation of Hyper TSC

The dashed lines are used in an obvious manner as separators to indicate the clauses to which the branching refers. If the branching refers to the reception of an event also a more compact notation with the same semantics may be used where the separation line is a continuation of the message input.

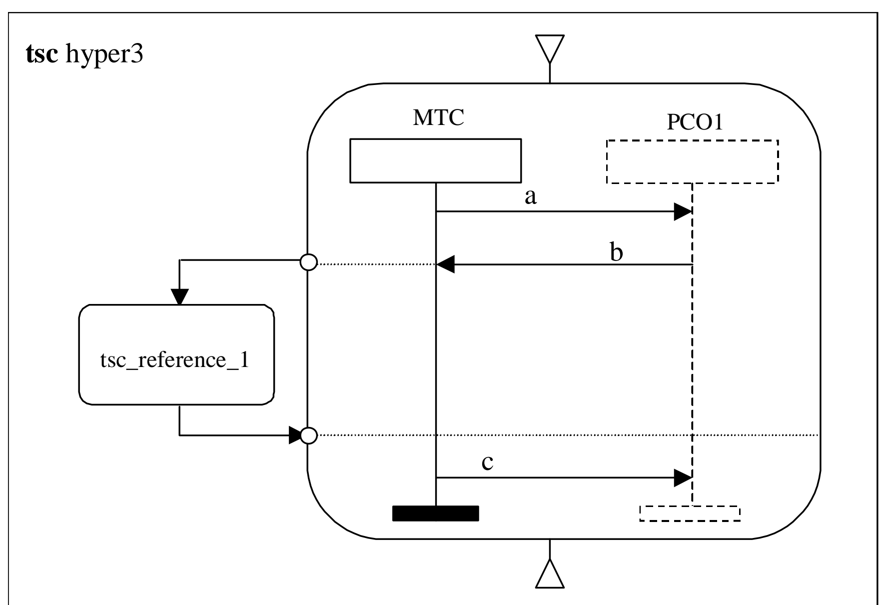


Figure 48: Shorthand for sectioning in Hyper TSC

The concept of TSC reference expansion is not restricted to HMSCs. It may be used also in any basic TSCs containing MSC references.

It is important to note that the extension for Hyper TSC merely refers to the graphical representation and does not imply any semantics changes.

A tag will be used in TSC to indicate whether a TSC shall be represented in expanded or non-expanded form.

5.16 Partial TSC

The analogy with hypertext in HyperTSC leads to another extension of TSC references. Since the TSC reference name normally does not provide much information about the underlying TSC definition it should be admitted to write pure comment text together with the test verdict into the TSC reference symbol instead of the TSC reference name thereby, the TTCN-3 syntax for comments must be used. Please consider figure 49 , which contains a comment only instead of a concrete reference.

It should be noted, however, that this deviates slightly from the MSC standard: The TSC references do not contain a name but an arbitrary text. Subsequently, two cases may be distinguished: the TSC reference is undefined or a corresponding TSC reference definition is provided. In the first case only the comment text together with the test verdict is assigned to the TSC reference symbols. In the later case, a TSC reference identifier may be specified explicitly in front of the comment text or otherwise a default identifier is created automatically.

In case where TSC reference definitions are provided the TSC references may be expanded either within the diagram or in a separate window depending on the special situation. The other way round, expanded TSC references may be closed. As an additional feature, complete paths in the HTSC may be expanded and shown in expanded form as a coherent TSC in a separate window.

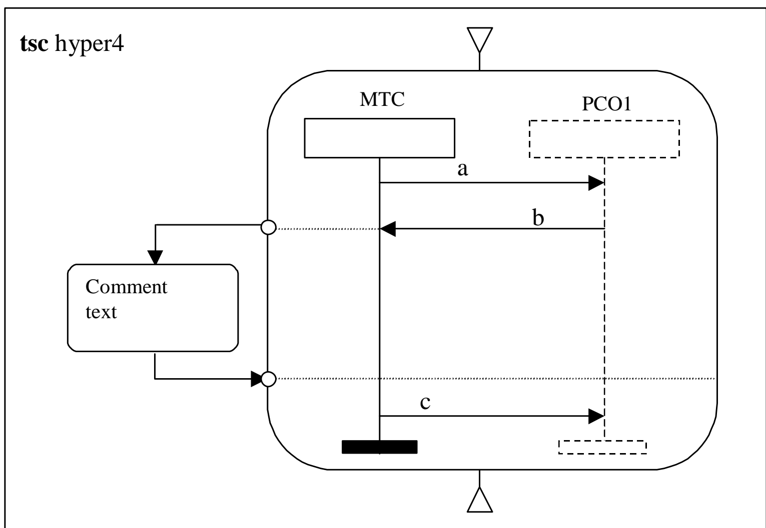


Figure 49: Partial TSC (including commented TSC reference)

5.17 Hybrid_TSC

Since TSC can be viewed as a basis for creating TTCN-3 test descriptions, some of the TSC reference definitions may be provided as well in form of TTCN-3 descriptions instead of TSC diagrams. A TSC containing such a hybrid description shall be denoted as Hybrid TSC. In a Hybrid TSC, each TSC reference may be defined either in form of a TSC diagram or in form of a TTCN-3 description.

The choice between both descriptions is left completely to the user. There are two extreme cases of Hybrid TSCs:

1) all TSC references are defined in form of TTCN-3 descriptions; or

2) all TSC references are defined in form of TSC diagrams, which is the normal HyperTSC case.

In practice probably, a mixture is most important where the main path of the test case (pass case) is described in a coherent manner in form of an expanded TSC diagram while the TSC references describing the side cases (inconclusive/fail) refer to TTCN-3 descriptions. Such a hybrid representation may be particularly useful for documentation purposes since it provides a complete test case description with a visualization of the main paths in form of the TSC format. In case of multiple components, the TSC reference containing a TTCN-3 description must refer to one single component.

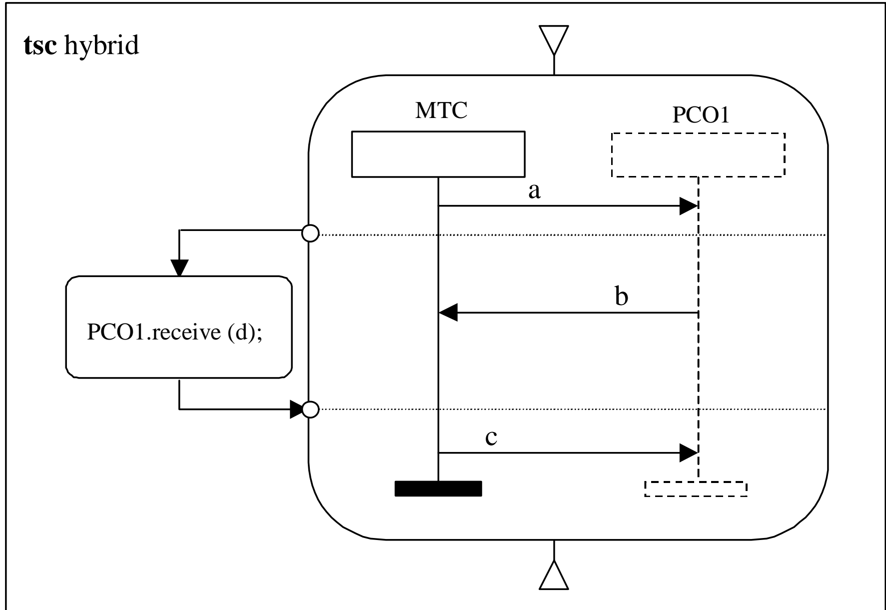


Figure 50: Hybrid TSC (including TTCN-3 core notation)

Annex A: Used subset of and extensions to MSC

A.1 Overview

Table A.1: Existing MSC constructs with possible extensions/modifications

Feature	Used	Textual changes	Graphical changes	Static semantics changes	Dynamic semantics changes	Section
MSC Document	yes	yes	yes	yes	no	
Basic MSC						
MSC	yes	yes	no	yes	yes	
Instance	yes	yes	yes	yes	yes	
Message	yes	yes	no	yes	yes	
Control Flow	yes	yes	no	yes	yes	
Environment	yes	no	no	no	no	
Gates	no					
General Ordering	no					
Condition	yes	yes	no	yes	yes	
Timer	yes (without parameters)	yes	no	yes	yes	
Action	yes	yes	no	yes	yes	
Instance Creation	yes	yes	no	yes	no	
Stop	yes	no	no	no	yes	
Data Concepts						
Declaring Data	yes	yes	yes	yes	yes	
Static Data	yes	yes	no	no	no	
Dynamic Data	yes	yes	yes	yes	yes	
Bindings	yes	yes	no	no	no	
Assumed Data Types	TTCN-3 data					
Time Concepts						
Time Constraints	yes (for references in control TSC only)	no	no	no	no	
Time Measurements	no					
Structural Concepts						
Co-region	yes	no	no	yes	no	
In-line Expression	yes	yes	no	no		
MSC Reference Expression	no					
MSC Reference	yes	yes	no	yes	no	
Instance Decomposition	no					
High-Level MSC	yes	yes	yes	yes	yes	

Table A.2: TSC specific constructs

Feature	Related to existing MSC Construct	Textual changes	Graphical changes	Static semantics changes	Dynamic semantics changes	Section
Port	Instance	yes	yes	yes	yes	
Start to Test Component	Instance Creation	yes	no	yes	yes	
Return to Function	Instance Creation	yes	no	yes	yes	
Clear to Port	Instance Creation	yes	no	yes	yes	
Start to Port	Instance Creation	yes	no	yes	yes	
Stop to Port	Instance Creation	yes	no	yes	yes	
Trigger	Message	yes	no	yes	yes	
Check	Message	yes	no	yes	yes	
Call	Control Flow	no	no	yes	no	
Getreply	Control Flow	yes	yes	yes	no	
Catch	Control Flow	yes	yes	yes	yes	
Getcall	Control Flow	yes	no	yes	no	
Reply	Control Flow	yes	yes	yes	no	
Raise	Control Flow	yes	yes	yes	yes	
Timer in Calls	Timer	yes	yes	yes	yes	
Functions	MSC References	yes	no	yes	yes	
Defaults	Action	yes	yes	yes	yes	
Hybrid TSC	HMSC	yes	no	no	no	
HyperTSC	HMSC	yes	no	no	no	

A.1.2 Test specific extensions

A.1.2.1 TSC document

Meaning
The TSC document defines the associated collection of TSCs.

A.1.2.1.1 New document sections

Syntax
The TSC document contains document sections for control, test case and functions. This implies the use of new keywords (textual extension) and additional separator lines (graphical extension). This need not to be used as it eases the reading only and comments can be used instead as a workaround. TSC allows document parameterization, i.e. a document head with parameters (textual extension).

Static Semantics
The TSC document must contain references to defined TSCs only. Incomplete TSC specifications containing just comments are also allowed. Control references may refer to an HTSC only.

Dynamic Semantics
No change.

A.1.2.1.2 Component and port instances

Syntax
Component and port instances are differentiated (by keywords) This implies a new syntax for instance declaration. It eases readability, but can be realized also with just comments or naming convention as a workaround.

Static Semantics
Port instances or corresponding message prefixes are allowed for declared ports only.

Dynamic Semantics
FIFO order for CPs (= connected ports) and no event structure for PCOs (= mapped ports) is assumed.

A.1.3 TSC heading

Meaning
The TSC head provides the TSC name together with parameter lists.

Syntax
The TSC head contains a system and return keyword (textual extension).

Static Semantics
Return value and return type have to be compatible. Only those ports declared in the test system interface shall be used as mapped ports. Test system interface and used messages/calls on mapped ports have to be compatible.

Dynamic Semantics
Return becomes a separate control event in the trace semantics.

A.1.4 Test components

Meaning
A test case is executed on test components. Each test case contains exactly one MTC.

A.1.4.1 MTC as keyword

Syntax
MTC/mtc is used as a new keyword (textual extension).

Static Semantics
MTC/mtc may be used only as identifier for "real" instances. There has to be exactly one mtc per test case.

Dynamic Semantics
Stop on an mtc instance has a special meaning, i.e. disrupting all other components.

A.1.4.2 Self as keyword

Syntax
SELF/self is used as a new keyword (textual extension). It can be used whenever the test component identifier is not known. Alternatively, the test component identifier can be empty in such cases.

Static Semantics
Self may be used only as identifier for "real" instances. There may be at most one self per TSC.

Dynamic Semantics
No change.

A.1.5 Messages

Meaning
Messages describe the asynchronous communication in test cases, i.e. send and receive events at components and ports.

Syntax
Messages are represented by solid arrows. On top of the message, the message type can be given, (optionally) with a port prefix in case of implicit port representation. Below the message arrow the message template can be given either by referring to a named message template or by giving an in-line template definition (in parenthesis).

Static Semantics
Template and message/call/etc. types have to match. Port type and message/call/etc. types have to match. Prefixes are allowed only for communication events.

Dynamic Semantics
For CPs (= connected ports) a FIFO order is defined. No event structure is defined for PCOs (= mapped ports).

A.1.6 Trigger

Meaning
The trigger operation filters messages with certain matching criteria from a stream of received messages on a given incoming port.

Syntax
In TSC, trigger is defined as a special message with keyword trigger (textual extension to the text syntax of messages).

Static Semantics
The trigger message must always point at a real instance.

Dynamic Semantics
Trigger is interpreted as a separate control event in the trace semantics. Sent messages/calls etc. which are not yet received at that port and which do not match, are deleted from that port. This is indicated by a special control event: delete.

A.1.7 Check

Meaning
The check operation allows read access to the top element of message–based and procedure-based incoming port queues without removing the top element from the queue.

Syntax
In TSC, check is defined as a special message with keyword check (textual extension to the text syntax of messages).

Static Semantics
The check message must always point at a real instance.

Dynamic Semantics
Check is interpreted as a separate control event in the trace semantics. It becomes only then part of a trace if it matches successfully to the top element of the port.

A.1.8 Control flow

Meaning
Control flow comprises procedure based (synchronous) communication mechanisms defined by means of calls and replies.

Syntax
TSC uses special keywords for getcall, getreply, catch, raise, reply and special binding to variables for parameters and return values of a call (textual extension to the text syntax of call messages). For the reply messages getreply, catch, reply and raise, solid message arrows are used instead of dashed arrows since they are already distinguished from call messages by keywords (graphical extension). As a workaround dashed message arrows may be used.

Static Semantics
Getcall, getreply and catch must point at real (component) instances only. The corresponding port has to be synchronous or mixed and must be type compatible. Call, raise and reply must point at port instances only. The port has to be synchronous or mixed and must be type compatible. Method areas and suspension areas are attached only to "real" (component) instances.

Dynamic Semantics
Getreply, catch are received at the end of a suspension region or after a non-blocking call, which matches the getreply and catch. Raise, reply are received at the end of a method symbol or after a non-blocking call, which matches the raise and reply.

A.1.9 Test verdicts within conditions

Meaning
Test verdicts within conditions denote the result of a test case.

Syntax
TSC employs verdict keywords together with a special handling of their meaning within conditions. However, as any identifier is allowed within MSC conditions, this does not really impact the syntax of MSC.

Static Semantics
The verdict keywords can be used only in conditions and action boxes, where the verdict of a component is set or read.

Dynamic Semantics
Verdicts can only become worse relative to an instance. Components report implicitly back their verdict to the mtc when stopping.

A.1.10 Timer

Meaning
In TSC, unnamed timers are used to supervise call operations.

Syntax
A timer start without timer identifier is directly attached to the beginning of a suspension region. A corresponding timeout is directly attached to the end of a suspension region (graphical extension: extension to the graphical placement of symbols). A semi-optimal workaround would be to place the start/timeout timer nearby the begin/end of a suspension region.

Static Semantics:
No other timer events are allowed between start and timeout of the unnamed timer.

Dynamic Semantics:
Timer start at the beginning of a suspension region and the end event of a suspension region become events in the trace. An end event can be a reply, exception, or unnamed timeout event. No other event can be between start and end.

A.1.11 Create to test components

Meaning
The create operation is used to dynamically create all test components except the MTC.

Syntax
The TSC create construct uses the textual TTCN-3 syntax instead of the textual MSC syntax (textual extension).

Static Semantics
The create arrow may point only to "real" instances (but not to the MTC) or to MSC references or to the environment. The referenced instance type must have been declared in the declaration part of the TSC document.

Dynamic Semantics
Create just becomes an event in the trace semantics.

A.1.12 Start to test components

Meaning
By means of the start operation the execution of a component's behaviour is started.

Syntax
Graphically, a dashed arrow represents the start operation. This implies a new use of dashed line messages (graphical extension). As a workaround, standard message arrows can substitute dashed line arrows. The TSC start construct uses the textual TTCN-3 syntax.

Static Semantics
The start arrow may point only to "real" instances (but not to the MTC) or to MSC references or to the environment. The referenced instance type must have been declared in the declaration part of the TSC document. Start may only happen after a create. On the created instance, no event is allowed between create and start.

Dynamic Semantics
Start becomes just an event in the trace semantics.

A.1.13 Return for functions

Meaning
The return operation terminates execution of a function. The return may be optionally associated with a return value.

Syntax
Graphically, a dashed arrow represents the return operation. This implies a new use of dashed line messages (graphical extension). As a workaround, standard message arrows can substitute the dashed line arrow. The TSC start construct uses the textual TTCN-3 syntax.

Static Semantics
The return arrow may point only to "real" instances (the calling instance) or to the environment. The referenced instance type must have been declared in the declaration part of the TSC document.

Dynamic Semantics
Return becomes just an event in the trace semantics.

A.1.14 Stop on test components

Meaning
The stop test component operation explicitly stops the execution of the test component in which the stop is called.

A.1.14.1 Special meaning of stop on the MTC instance

Syntax
At least, this implies a semantic change to cover the disruption of other test components (those which have been created but are still running when the MTC stops). Possibly, it would be even better to have a specific symbol for this stop behaviour with disrupt semantics. Either we require the use of that symbol on MTC instances or we use a pre-processing step to substitute "old" stop symbols on MTC instances with this new symbol and only then apply the semantics.

Static Semantics:
No change.

Dynamic Semantics
The stop event on an MTC initiates a disrupt for all other active components, i.e. only other stop events are possible (this does of course not forbid to have further events belonging to the next test case).

A.1.14.2 Stop within an operand of an in-line expression

Syntax
In TSC, a stop of a test component is allowed also within a section of an in-line expression (graphical extension). As a workaround, in some cases, the sections of the in-line expression may be chosen differently.

Static Semantics
Stop is allowed only for real instances.

Dynamic Semantics
No change.

A.1.15 Clear, start and stop to ports

Meaning
Clear removes the content of an incoming port queue. Start starts listening and gives access to a port. Stop stops listening and disallows sending operations at a port.

Syntax
Clear/start/stop are special messages to a port. This implies a new use of dashed line messages and new text syntax for those arrows (graphical and textual extension). As a workaround, dashed line arrows can be substituted by standard message arrows.

Static Semantics
Clear/Start/Stop may be sent only to declared ports.

Dynamic Semantics
Clear/start/stop are always pointing to port instances. Ports of a component are implicitly started at the start of this component. A start for a started port is the null operation. Start, clear and stop become control events in the trace semantics. A receive and a send event on a stopped port is not possible. A start for a stopped port enables again the receive and send events.

A.1.16 In-line expressions

Meaning
In TSC, in-line expressions are used to define alternative and interleaved behaviour.

A.1.16.1 Propagation of messages to the environment

Syntax
Out- and in-messages point at, respectively come from the in-line expression frame only. This is a graphical syntax change to MSC, but in fact just an extension to MSC tools. Note that in MSC-96 this propagation of messages to the next higher environment was allowed, contrary to MSC.

Static Semantics
No change.

Dynamic Semantics
Events coming from or pointing at the in-line expression frame are propagated to the environment (i.e. recursively to the next higher environment).

A.1.16.2 New interleave in-line expression

Syntax
In TSC, a new keyword int in the left top corner of an in-line expression frame is introduced (textual extension).

Static Semantics
The interleaving in-line expression must always include a component ("real") instance. It is not allowed within alt expressions and must not include loop expression, activate/deactivate, stop, return and MSC references.

Dynamic Semantics
Interleaving in-line expressions denote full interleaving with "non-interruptible" parts (the receive {send, calculation, etc.} * sequences). The non-interruptible parts are defined by the TTCN-3 semantics.

A.1.17 HTSC

Meaning
HTSCs (High Level TSCs) define the possible composition of TSCs. HTSCs are used for the description of the module control part.

Syntax
In TSC, variable declarations are allowed within HTSCs (textual extension to the HMSC header). Action boxes are included in HTSCs (graphical extension). As a (non-ideal) workaround, a reference to a TSC including the action box may be used. Value returning TSC references are introduced (textual extension).

These syntax extensions could be even generic. It is problematic, however, for weak sequential composition, which is fortunately not an issue for TSC, since every test case (mtc instance) has an unambiguous start and end event.

Static Semantics
These extensions may be used for control TSCs only.

Dynamic Semantics
There is implicitly a control instance, which hosts the variables. The sequential composition of test cases does not have to consider weak composition as stopping/terminating the mtc imposes stopping all running test components and only then the next test case can be invoked.

A.1.18 Hybrid TSCs

Meaning
Within hybrid TSCs, some of the TSC references may point to TSC reference definitions, which are provided in form of TTCN-3 descriptions instead of TSC diagrams.

Syntax
MSC references point at "legal" TTCN-3 code fragments (tool issue).

Static Semantics
The code fragments must be consistent (to the environment).

Dynamic Semantics
A code to TSC mapping may be used to handle the semantics for the code fragment or alternatively, the TSC graphics is mapped to TTCN-3 code with a subsequent evaluation of the whole.

A.1.19 Extensions to the data part

A.1.19.1 Declaring Data

Meaning
Declaration of types and variables used within a TSC.

Syntax
A number of changes have been made to the syntax used for declaring data within TSC:

- Instance, message and timer declarations have been removed from the document header. For TSC, the declaration of component instance variables is now given within the TTCN-3 data definition string;
- Introduced the concepts of implicit and explicit typing for instances;
- HTSCs can now contain local variable declarations;
- Message names now represent message types, and parameter part now is prefixed with a message template;
- Addition of verdicts;
- Value returning TSC references.

Static Semantics
HTSCs can only contained variable declarations if it is referenced from the control part of a TSC document.
Message names and parameters static rules have changed.
Explicit typing imposes TTCN-3 semantics on the use of component instance events and ports.
Value returning TSC references.

Dynamic Semantics
Addition of verdicts.
Addition of local variables for control HTSCs.
Value returning TSC references.

A.1.19.2 Static Data

Meaning
Parameterization of TSCs constructs.

Syntax
Formal parameter lists now follow TTCN-3 syntax.

Static Semantics
No change.

Dynamic Semantics
No change.

A.1.19.3 Dynamic Data

Meaning
Dynamic data refers to the assignment and reassignment of variables.

Syntax
Local variables, declared within control HTSCs, can be assigned values within High-level TSCs.
TSCs references can return values.
Verdicts.
Wildcards are not need for TTCN-3 parameterization.

Static Semantics
As above.

Dynamic Semantics
TSC has implicit verdict variables, one for each test case and one for each component. The dynamic rules for these variables are defined by TTCN-3.
Control variables define a possible execution order for control HTSCs.

A.1.19.4 Bindings

Meaning
Bindings are treated as assignments.

Syntax
TSC references can return values.

Static Semantics
No change.

Dynamic Semantics
No change.

A.1.20 Hyper TSCs

Meaning
In HperTSCs, some of the TSC references may appear in expanded form whereby the TSC referencing mechanism is interpreted in a hypertext-like manner (tool issue). Several expanded TSC references may be combined to one coherent expanded TSC reference with the connection points being shifted to the borderline of the TSC reference.

Syntax
HyperTSCs admit that only comments are contained in TSC references. The TSC reference name is implicitly defined by the hyperlink (graphical extension, in particular, tool issues).

Static Semantics
No change.

Dynamic Semantics
No change.

A.1.21 Ports

Meaning
In TSC, communication is effected between the components within the test system and between the components and the test system interface via communication ports.

Syntax
For ports a special event structure differing from "real" instances may be defined: FIFO/LIFO, no order. In TSC we take FIFO for CPs and no event structure for PCOs. The explicit port representation uses a new graphical symbol, i.e. a dashed instance (graphical extension). The implicit port representation uses a message prefix. Therefore, this extension can just be reflected as a textual extension with message prefix and keyword port, async, etc. in the instance header (workaround).

Static Semantics
Port instances are allowed for declared ports only.

Dynamic Semantics
An event structure is assumed according to the definition. In TSC this implies FIFO order for CPs (= connected ports) and no event structure for PCOs (= mapped ports).

A.1.22 Default

Meaning
A default behaviour is an extension to an alt statement or a single receive operation which is defined in a special manner. A default behaviour has to be activated before it is used and may again be deactivated. This construct may be used quite generally, e.g. for exception handling or for operations like "onhook" in telecommunication.

Syntax
A new graphical symbol with new keywords (activate/deactivate) is introduced for the default construct. A workaround for the new default symbol is an action box.

Static Semantics
The default construct may be only attached to component ("real") instances. For the default behaviour, which is activated or deactivated, a corresponding TSC must be defined.

Dynamic Semantics
The default behaviour defines additional alternative events.

Annex B: The TSC forms

B.1 Overview

TSC can be used in different forms to represent TTCN-3 test cases. Using a number of simple TSC examples, each illustrating a different form, we explain those forms.

TSC forms are either for test purpose specifications (these are not considered in the present document) or for test cases. TSC Forms for test cases can be characterized by the following aspects:

Vertical vs. no vertical split,
i.e. one test component per TSC or combined view on all test components in a TSC.

Horizontal vs. no horizontal split,
i.e. one TSC per function or combined view on the complete behaviour in a TSC.

Explicit vs. implicit port representation,
i.e. port representation with special port instances or port representation as annotations to send and receive operations.

Hybrid vs. no hybrid form,
i.e. references referring to TSC definitions or to TTCN-3 specification parts.

Partial vs. complete form,
i.e. TSC references may contain comments only or all TSC references refer to TSCs.

A TSC specification can make use of hyper facilities. A **HyperTSC** contains in TSC references hyperlinks to the defining TSC and supports different views on a TSC to make the test specification more readable and more transparent.

B.1.1 An example

A simple example is taken to express the various forms of TSC specifications . The small example consists of an SUT that has two interfaces *A* and *B*. The test behaviour is to send a message *a* at interface *A* and to receive subsequently a message *b* at interface *B*. The test behaviour is realized by two components *TC1* and *TC2*, which are mapped to the SUT interfaces *A* and *B*, respectively. The are co-ordinated via port *C* by exchanging message *c*.

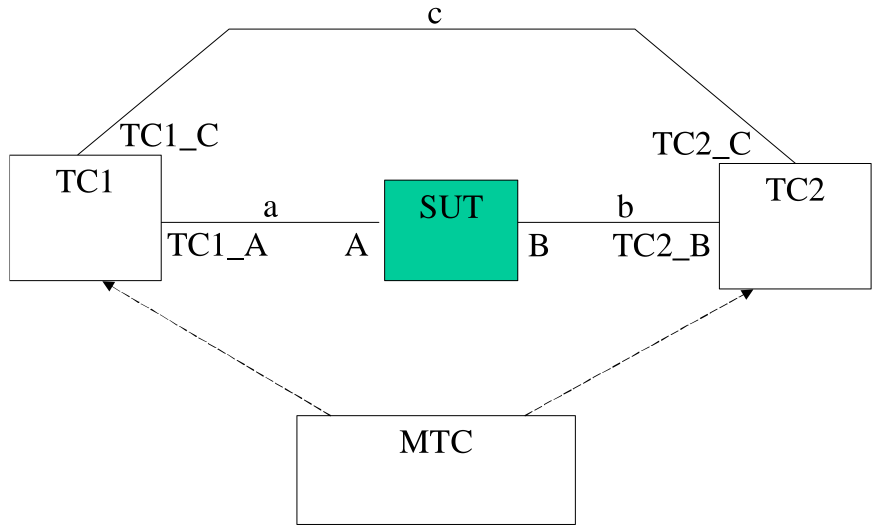


Figure B.1: Simple example for explanation of TSC forms

Those parts of the TTCN-3 example specification that are essential to discuss the various TSC forms, are given below:

```
MTC:         var TC1_type TC1 = create(TC1_type);
             var TC2_type TC2 = create(TC2_type);
             map(TC1.TC1_A,sut.A); map(TC2.TC2_B,sut.B); connect(TC1.TC1_C,TC2.TC2_C);
             TC1.start(behaviour1);
             TC2.start(behaviour2);
             all component.done;
             verdict.set(pass);

behaviour1: TC1_A.send(a);
             TC2_C.send;
             verdict.set(pass);

behaviour2: TC2_C.receive;
             TC2_B.receive(b);
             verdict.set(pass);
```

B.1.2 Form aspect: vertical split vs. no vertical split

In a vertical split form, there are TSCs per test component, which represent the behaviour of the test components. For example, there is a TSC *vertical_split_mtc* for the master test component (indicated by the keyword **mtc**), a TSC *behaviour1* for test component *TC1*, and a TSC *behaviour2* for test component *TC2*. Per TSC, there is a TSC instance representing the test component. The TSC instance head contains the identifier of the test component, the keyword **mtc** and its type. In the mid of figure 2, there is an unnamed instance of type *TC1_type*, which acquires the test component identifier from the create message of the mtc. A test component is started with the special start message (indicated by the dashed arrow symbol). Afterwards, the behaviour of the test component is represented. In the example, message a is sent via port *TC1_A* and message *c* via port *TC1_C*. Finally, a test verdict is assigned (in the example, *pass* is assigned) and the test component terminates.

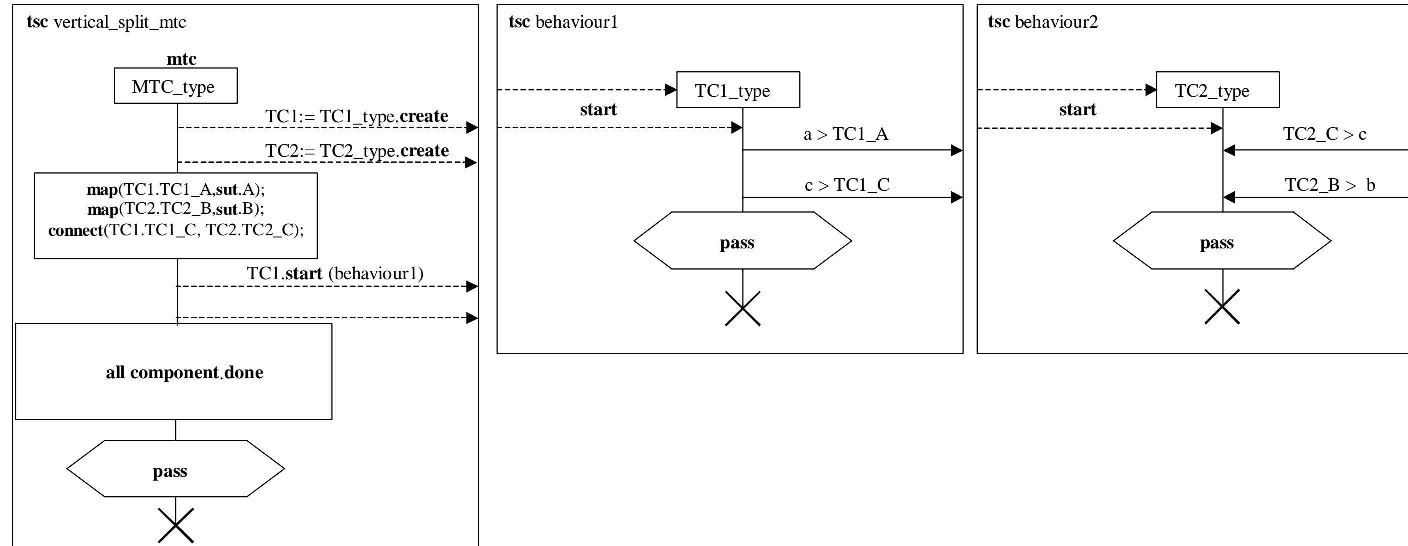


Figure B.2: Vertical split

Without a vertical split, one TSC contains all test components constituting the test case behaviour. In this view, the communication between *TC1* and *TC2* with message *c* transferred from *TC1_C* to *TC2_C* becomes apparent. In addition, the order between test events at different test components is visible. The identifier for the behaviour function of a test component instance is contained in its start operation.

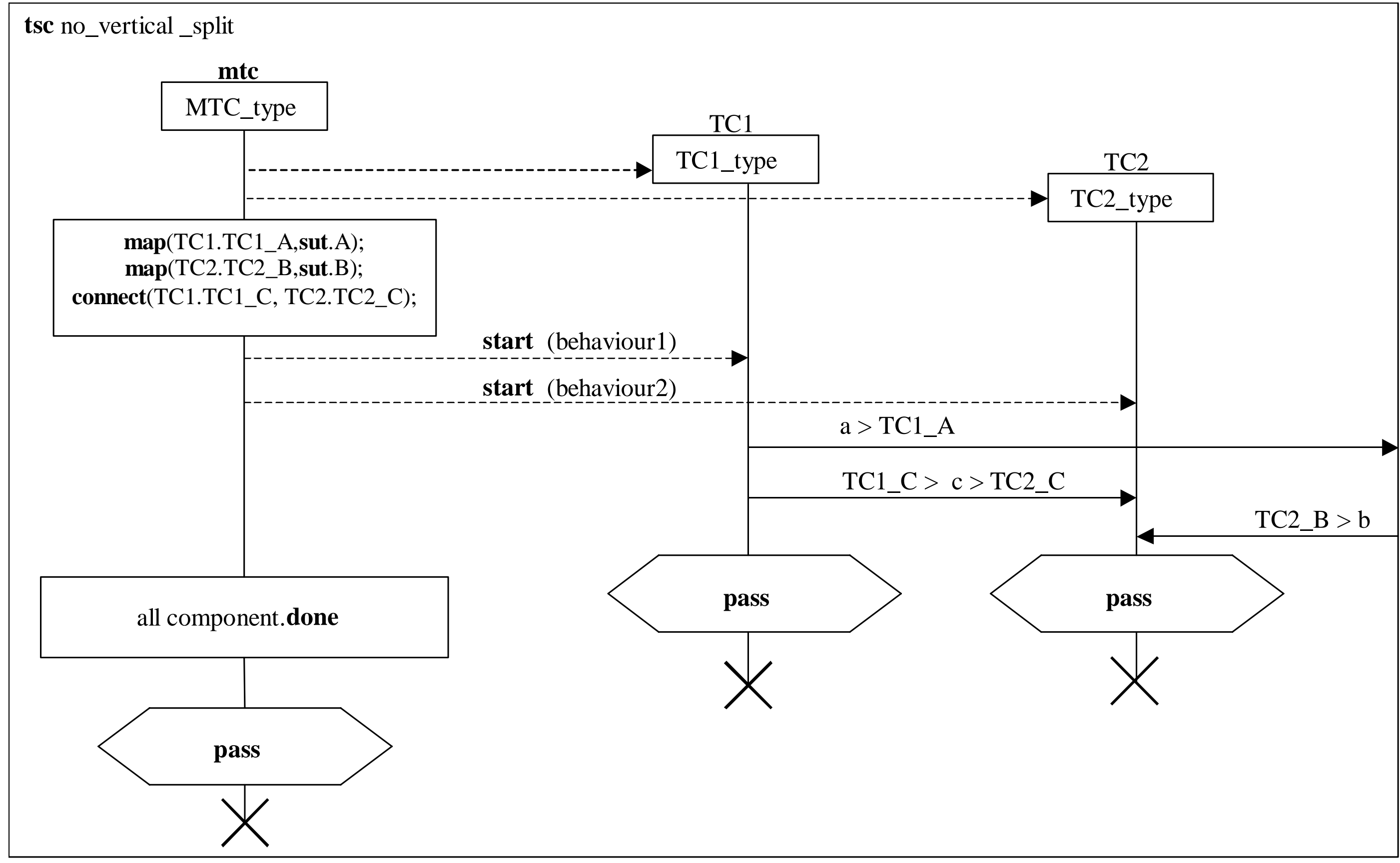


Figure B.3: No vertical split

B.1.3 Form aspect: horizontal split vs. no horizontal split

In the horizontal split form, separate TSCs per function called in the behaviour of a test component are defined. Let us assume, that in the example the behaviour of the test component uses an additional function *func1* to perform the sending and receiving of messages. A separate TSC *func1* is defined for this function (right-hand side of figure B.4). The test component refers to this TSC by means of a TSC reference (a box with round corners containing the TSC identifier it refers to).

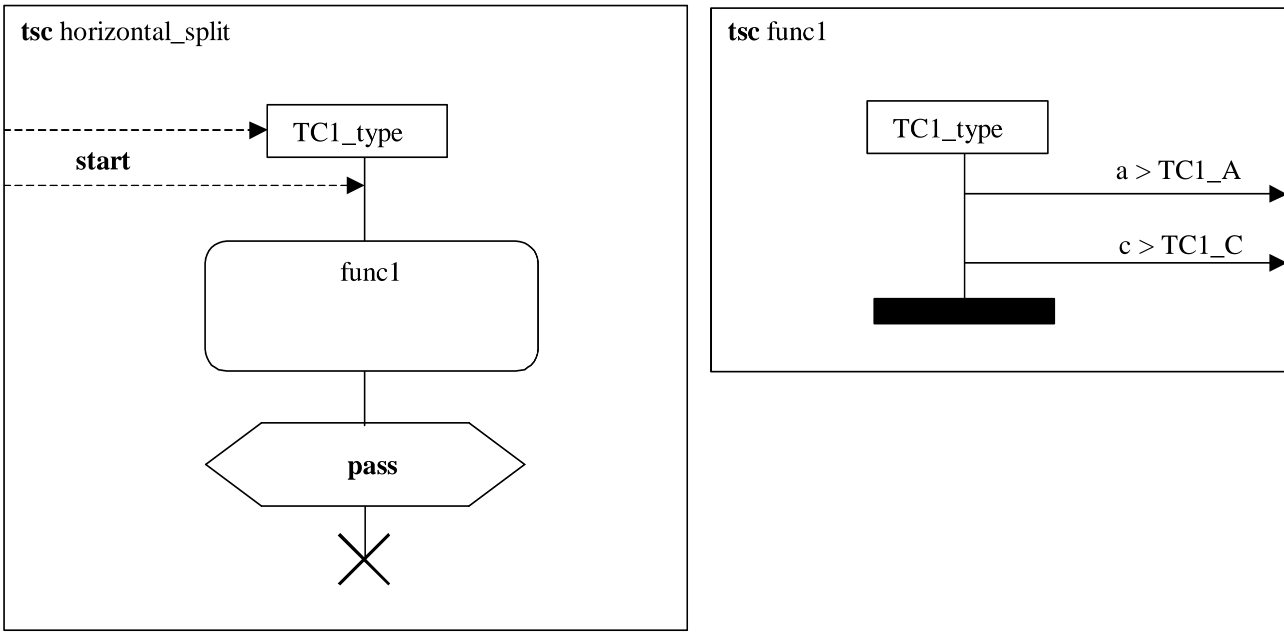


Figure B.4: Horizontal split

Without horizontal split, the test behaviour of functions is represented in-line at the TSC instance, which represents the test component that performs a given function. In this form, the information on the structuring of test behaviour into functions is lost! Whenever this information shall be kept, a Hyper TSC (i.e. one where the TSC reference to a function is expanded) shall be used instead (see clause B.1.15). In the example, the test behaviour of the function *func1* is just represented in-line at the instance.

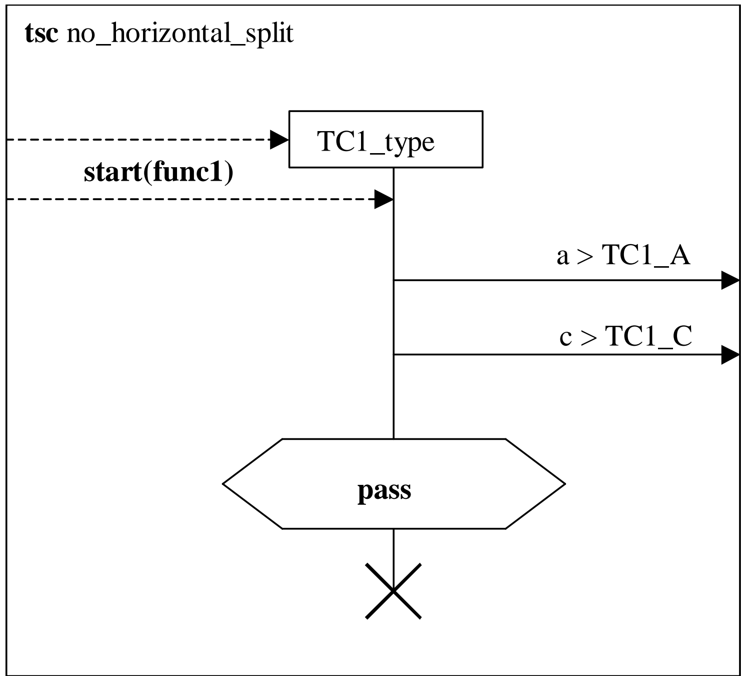


Figure B.5: No horizontal split

B.1.4 Form aspect: explicit vs. implicit port representation

Communication ports can be represented in TSC differently. Either, a port is represented by a special port instance (indicated by dashed instance head, line and end symbols) or it is represented as a prefix to the message.

Figure B.6 contains an explicit port representation for the ports *TC1_A* of type *A_type* and *TC1_C* of type *C_type*. The instance head of port instances contains the port type. Messages, which are sent or received at a port, are represented by incoming or, respectively, outgoing messages of the port instance. For example, message *a* that is sent by *TC1* to port *TC1_A* is represented by a message from instance *TC1* to instance *TC1_A*.

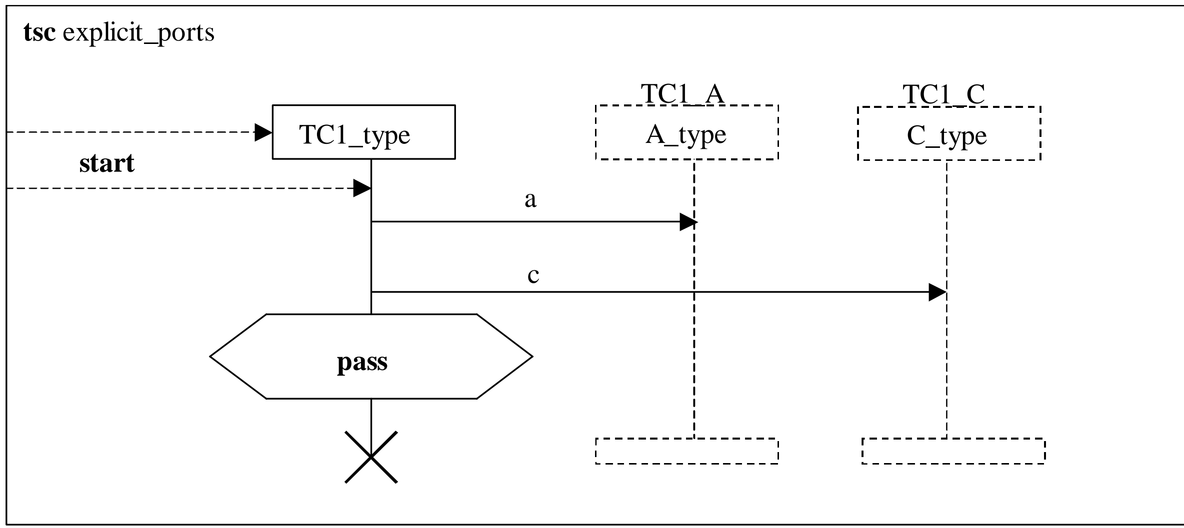


Figure B.6: Explicit ports

Figure B.7 contains an implicit port representation, where the prefix in front of a message indicates the port to which a message is sent or from which a message is received. The prefix is separated from the message with a ">" symbol. For example in the figure, TC1*_A > a* indicates that message *a* is sent to port *TC1_A*. Optionally, the port type can be given in addition such as it is done with *TC1_C : C_type > c*.

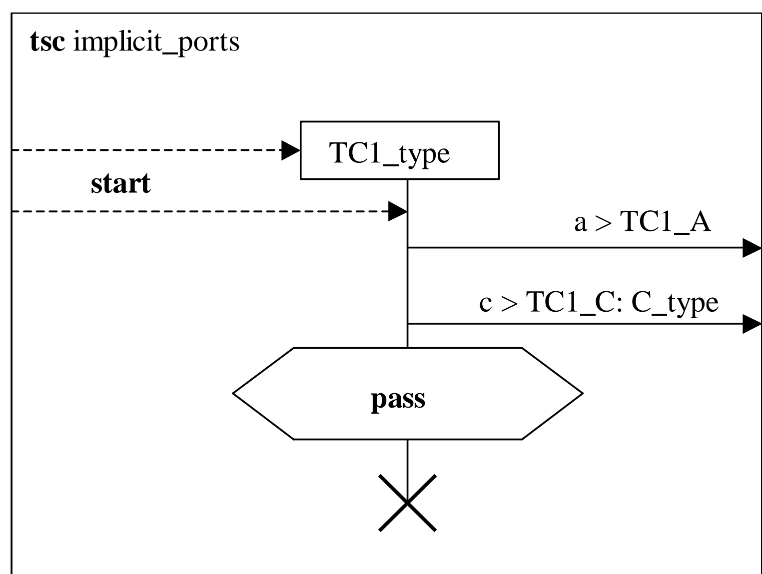


Figure B.7: Implicit ports

A combination of different port representation can also be used. In figure B.8 , port *TC1_A* is represented explicitly and port *TC1_C* is given implicitly.

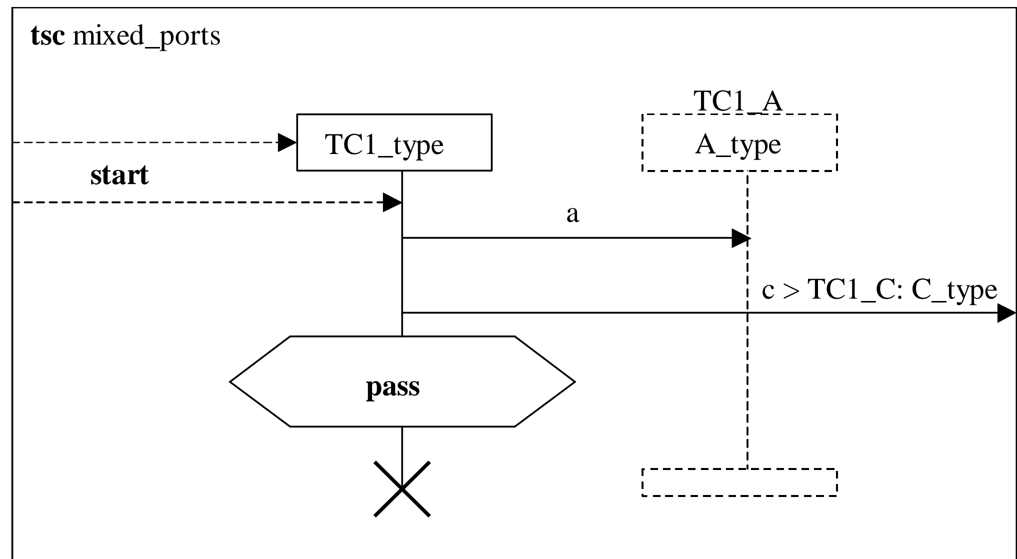


Figure B.8: Combination of different port representation

A tag will be used in TSC to indicate whether a port shall be represented in TSC/gr explicitly or implicitly.

B.1.5 Form aspect: hybrid vs. not hybrid

In hybrid TSC, TTCN-3 code can be used directly inside of TSC references. In figure B.9, the test behaviour is given textually in terms of TTCN operations and not graphically.

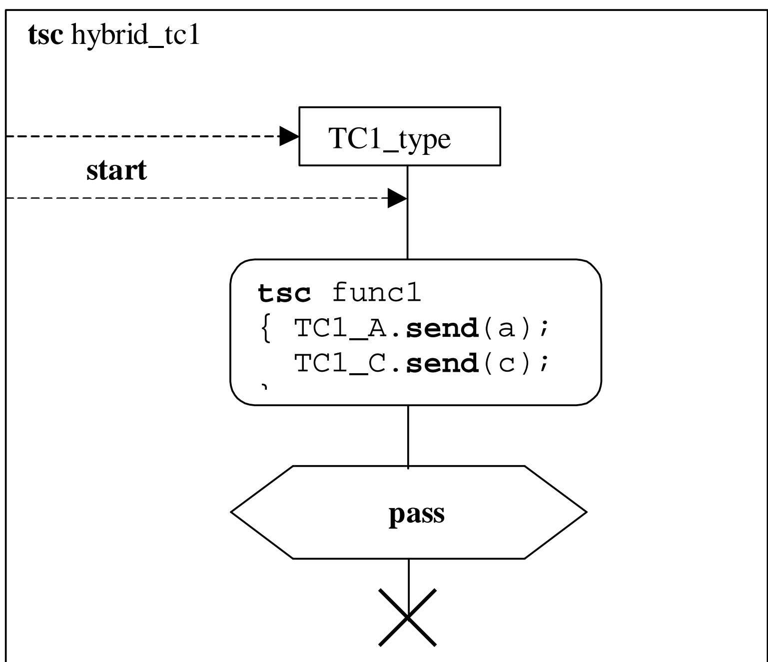


Figure B.9: Hybrid TSC

The figures given in the previous clauses have not been hybrid.

A tag will be used in TSC to indicate whether a TSC is defined in TSC/gr or in TTCN-3. In fact, both the TSC/gr and the TTCN-3 code could be used within a TSC definition and either of the two or both can be shown.

B.1.6 Form aspect: partial vs. complete

TSC specifications may be partial only. A TSC may contain inside a TSC reference a comment only (as it is done in figure B.10). The support of partial TSC specifications is of particular importance for the step-wise development of test suites within TSC.

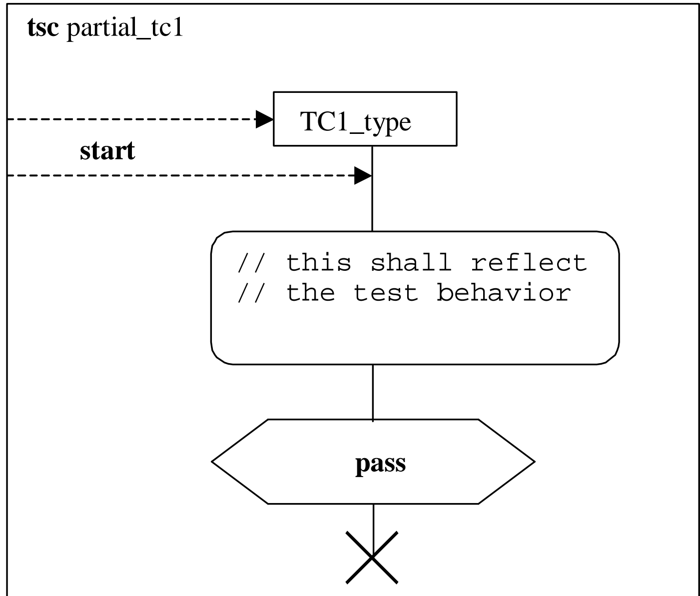


Figure B.10: Partial TSC

Complete TSC specifications contain TSC identifiers or TTCN-3 code within TSC references. The figures given in the previous clauses are complete ones.

B.1.7 Summary of TSC forms

The five different form aspects of TSC result in several different forms for representing test suites in TSC.

The first three aspects, i.e. vertical vs. no vertical split, horizontal vs. No horizontal split, and explicit vs. implicit port representation, relate to the graphical setting of test specifications in TSC. The hybrid vs. not hybrid aspect relates to the combined use of TSC and TTCN-3. The partial vs. complete form aspect relates to the level of completeness of a TSC specification.

Since the graphical representation of test cases in TSC is the main focus of the work, the present document concentrates on the vertical vs. no vertical split, horizontal vs. no horizontal split, and explicit vs. implicit port representation only.

Annex C: Subset of the graphical syntax of TSC

This grammar definition for TSC/gr is only partial. It does not cover:

- HyperTSC;
- configuration symbols between test component instances and port instances;
- connection symbols between TSC references;
- stop symbol within operands of in-line expressions.

The graphical syntax definition of TSC/gr uses grammar rules of TTCN-3 and of MSC. Whenever TTCN-3 core notation production rules are used (for textual annotation of graphical elements) they are referred to as **production_rule_name**. Those rules are contained in annex A of [1]. Whenever MSC graphical production rules are used, they are referred to as < **production_rule_name** >. Those rules are taken from [2], clause 1. Please note that rules taken from MSC may have different textual syntax for annotated graphical symbols and may have a modified set of derived productions within TSC. New rules and rules where the graphical syntax of MSC is changed have no prefix, i.e. **production_rule_name**.

C.1 Meta-Language for TSC/gr

The graphical syntax for TSC/gr is defined on the basis of the graphical syntax of MSC [2]. The graphical syntax definition uses a meta-language, which is explained in clause 1.3.4 of [2]:

"The graphical syntax is not precise enough to describe the graphics such that there are no graphical variations. Small variations on the actual shapes of the graphical terminal symbols are allowed. These include, for instance, shading of the filled symbols, the shape of an arrow head and the relative size of graphical elements. Whenever necessary the graphical syntax will be supplemented with informal explanation of the appearance of the constructions.
The meta-language consists of a BNF-like notation with the special meta-constructions: ***contains***, ***is followed by***, ***is associated with***, ***is attached to, above*** and ***set.*** These constructs behave like normal BNF production rules, but additionally they imply some logical or geometrical relation between the arguments. The ***is attached to*** construct behaves somewhat differently as explained below. The left-hand side of all constructs except ***above*** must be a symbol. A symbol is a non-terminal that produces in every production sequence exactly one graphical terminal. We will consider a symbol that ***is attached to*** other areas or that ***is associated with*** a text string as a symbol too. The explanation is informal and the meta-language does not precisely describe the geometrical dependencies".

See [2] for more details.

C.2 Conventions for the syntax description

Table C.1 defines the meta-notation used to specify the grammar for TSC. It is identical to the meta-notation used by TTCN-3, but different from the meta-notation used by MSC. In order to ease the readability, the correspondence to the MSC meta-notation is given in addition and differences are indicated.

Table C.1: The Syntactic Meta-Notation

Meaning	TTCN-3	TSC	MSC	Differences
is defined to be	::=	::=	::=	
abc followed by xyz	abc xyz	abc xyz	abc xyz	
Alternative	|	|	|	
0 or 1 instances of abc	[abc]	[abc]	[abc]	
0 or more instances of abc	{abc}	{abc}	{abc}*	X
1 or more instances of abc	{abc} +	{abc} +	{abc} +	
Textual grouping	(...)	(...)	{...}	X
the non-terminal symbol abc	Abc	abc (for a TSC non-terminal) or <abc> (for a MSC non-terminal) or abc (for a TTCN non-terminal)	<abc>	X
a terminal symbol abc	**abc** or "abc"	**abc** or "abc"	**abc** or <name> or <character string>	X

C.3 Relation between TTCN-3 and TSC Files

A TTCN-3 Module has as graphical representation a TSC Document. The test case, functions and named alts contained in a TTCN-3 Module have as graphical representations Test Sequence Charts.

The mapping function from TSC to TTCN-3 core notation defines the underlying core notation for TSC documents and Test Sequence Charts. The **with** attribute within the TTCN-3 core notation is used to reference to the TSC Document and the Test Sequence Charts, respectively. The relation between TSC document and TTCN-3 core notation module is done via the identical identifier of the TSC document and the TTCN-3 module. The **related to** attribute within TSC allows to reference to related system specifications, requirement specifications, etc.

This relation is shown in figure C.1.

```
module MyModule
{
:
const integer MyConstant := 1;
type record MyMessageType { … };
:
function MyTestStep(){ … }
    with display "tsc MyTestStep";
:
testcase MyTestCase(){ … }
    with display "tsc MyTestCase";
:
}
with display "tscdocument MyModule";
```

Figure C.1: Relating TTCN-3 Core Notation to TSC

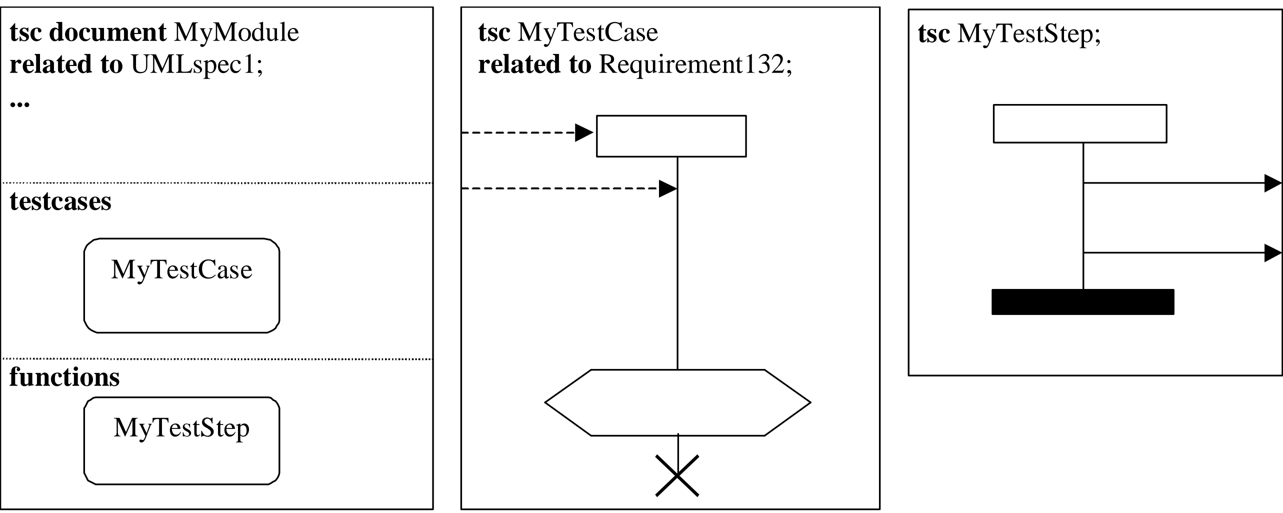


Figure C.2: Relating TSC to TTCN-3 Core Notation

It is a technical issue whether the core notation and the TSC presentations of a TTCN-3 Module are stored in one file or in several files. This is not defined here.

C.4 The TSC/gr production rules

C.4.1 Test Sequence Chart document

```
TSCDocumentArea ::=
        <frame symbol>
        contains TSCDocumentHead
        [ is followed by ControlPartArea ]
        [ is followed by TestcasePartArea ]
        [ is followed by FunctionPartArea ]
        [ is followed by NamedAltPartArea ]
;

TSCDocumentHead ::=
        tscdocument TTCN3ModuleId [ ModuleParList ]
        [ related to  Identifier { "," Identifier } ]
        [ explicit_typing | implicit_typing ]
        [ WithStatement ] [ ";"]
        [ [ language TTCN-3 ] data TTCN3DataDefinitionPart]
;
```

The **related to** identifier list may refer for example system specifications, requirements specification, etc., which relate to the TSC document.

The **explicit_typing** keyword indicates that declarations needed within this TSC document are contained in the module definition part. The **implicit_typing** keyword indicates that the declarations have to be generated from the TSCs contained in this TSC document. If no keyword is used the default is **explicit_typing**.

```
TTCN3DataDefinitionPart ::=
        "{"  TTCN3DataDefinitionList "}"
;

TTCN3DataDefinitionList ::=
        { TTCN3DataDefinition [ WithStatement ] }
;
```

```
TTCN3DataDefinition ::=
        TypeDef | ConstDef |
        TemplateDef | FunctionDef | SignatureDef |
        ImportDef | TSCGroupDef | ExtFunctionDef |
        ExtConstDef
;
```

Functions defined within TTCN3DataDefinitions shall only use basic statements or function references to functions defined within TTCN3DataDefinitions or to external functions.

```
TSCGroupDef ::=
        GroupKeyword GroupIdentifier
        BeginChar
            TTCN3DataDefinitionList
        EndChar
;

ControlPartArea ::=
        control
        (TSCControlReferenceList [ ";" TSCFunctionGroupDefList ]
        | is followed by (TSCControlReferenceArea *) set )
        is followed by <separator area>
;

TSCControlReferenceList ::=
        TSCControlReference { ","TSCControlReference }
;

TSCControlReference ::=
        ( [ control ] FunctionIdentifier ) | GroupIdentifier
;

TSCFunctionGroupDefList ::=
        TSCFunctionGroupDef { ";" TSCFunctionGroupDef }
;

TSCFunctionGroupDef ::=
        GroupIdentifier ":=" TSCFunctionorGroupReferenceList
;

TSCFunctionorGroupReferenceList::=
        TSCFunctionorGroupRef { ","TSCFunctionorGroupRef }
;

TSCFunctionorGroupRef ::=
        FunctionIdentifier | GroupIdentifier
;

TSCControlReferenceArea ::=
        <msc reference symbol>
        contains ( [ control ] FunctionIdentifier | GroupIdentifier )
;
```

TSC references used in the ControlPartArea shall refer to HTSCs representing TTCN-3 control functions or to groups that contain other groups or references to control HTSC. Only one control HTSC reference can be prefixed with **control**.

```
TestcasePartArea ::=
        testcases
        ( TSCTestcaseorGroupReferenceList [ ";" TSCTestcaseGroupDefList ]
        | is followed by ( TSCTestcaseReferenceArea *) set )
        is followed by <separator area>
;

TSCTestcaseorGroupReferenceList::=
        TSCTestcaseorGroupRef { ","TSCTestcaseorGroupRef }
;

TSCTestcaseorGroupRef ::=
        TestcaseIdentifier | GroupIdentifier
;

TSCTestcaseGroupDefList::=
        TSCTestcaseGroupDef { ";"TSCTestcaseGroupDef }
;
```

```
TSCTestcaseGroupDef ::=
        GroupIdentifier ":=" TSCTestcaseorGroupReferenceList
;

TSCTestcaseReferenceArea::=
        <msc reference symbol>
        contains (TestcaseIdentifier | GroupIdentifier )
;
```

TSC references used in the TestcasePartArea shall refer to TSCs representing TTCN-3 test cases or to groups containing other groups or test cases only.

```
FunctionPartArea ::=
        functions
        ( TSCFunctionorGroupReferenceList [ ";"TSCFunctionGroupDefList]
        | is followed by (TSCFunctionReferenceArea *) set )
;

TSCFunctionReferenceArea::=
        <msc reference symbol>
        contains (FunctionIdentifier | GroupIdentifier )
;
```

TSC references used in the FunctionPartArea shall refer to TSCs representing TTCN-3 functions that include behaviour statements or to groups containing other groups or functions only. Data functions are not graphically represented in TSC.

```
NamedAltPartArea ::=
        named alts
        ( TSCNamedAltorGroupReferenceList [ ";"TSCNamedAltGroupDefList]
        | is followed by (TSCNamedAltReferenceArea *) set )
;

TSCNamedAltGroupDefList ::=
        TSCNamedAltGroupDef { ";" TSCNamedAltGroupDef }
;

TSCNamedAltGroupDef ::=
        GroupIdentifier ":=" TSCNamedAltorGroupReferenceList
;

TSCNamedAltorGroupReferenceList::=
        TSCNamedAltorGroupRef { ","TSCNamedAltorGroupRef }
;

TSCNamedAltorGroupRef ::=
        NamedAltIdentifier | GroupIdentifier
;

TSCNamedAltReferenceArea::=
        <msc reference symbol>
        contains (NamedAltIdentifier | GroupIdentifier )
;
```

TSC references used in the NamedAltPartArea shall refer to TSCs representing named alts or to groups containing other groups or named alts only.

C.4.2 Groups

```
TSCGroup ::=
        TSCControlGroup | TSCTestcaseGroup |
        TSCFunctionGroup | TSCNamedAltGroup
;

TSCControlGroup ::=
        <msc symbol>
        contains TSCGroupHeading
        is followed by ( TSCControlReferenceArea *) set
;

TSCGroupHeading::=
        tscgroup GroupIdentifier
;

TSCTestcaseGroup ::=
        <msc symbol>
        contains TSCGroupHeading
        is followed by ( TSCTestcaseReferenceArea *) set
;

TSCFunctionGroup ::=
        <msc symbol>
        contains TSCGroupHeading
        is followed by ( TSCFunctionReferenceArea *) set
;

TSCNamedAltGroup ::=
        <msc symbol>
        contains TSCGroupHeading
        is followed by ( TSCNamedAltReferenceArea *) set
;
```

C.4.3 Test Sequence Chart

```
TSCDiagram ::=
        BasicTSCDiagram | HTSCDiagram | HyTSCDiagram
;

BasicTSCDiagram ::=
        <msc symbol>
        contains TSCHeading <msc body area>
;

HTSCDiagram ::=
        <msc symbol>
        contains HTSCHeading ( HTSCExprArea+ ) set
;
```

The use of several HTSC expression areas within one HTSC is a shorthand notation for the parallel expression of those HTSC expression areas. This shall not be used for HTSCs representing control functions.

```
HyTSCDiagram ::=
        <msc symbol>
        contains TSCHeading TSCHyperBodyArea
;

TSCHeading ::=
        tsc TSCHead [ <end> ]
;

TSCHead ::=
            ( TestcaseIdentifier [ "(" [ TestcaseFormalParList ] ")" ]
              SystemSpec)
        |   ( FunctionIdentifier [ "(" [ FunctionFormalParList ] ")" ]
                  [ReturnType] )
        |   ( NamedAltIdentifier [ "("[ NamedAltFormalParList ] ")"] )
;
```

Local definitions shall be used only for TSCs representing test cases and functions but not for TSCs representing named alts. Please note that all variable and timer definitions for test cases and functions shall be part of the component type definition referenced in the runs on and/or the instance head of a test component.

```
<msc symbol> ::=
        <frame symbol>

;

<frame symbol> ::=
        [rectangle]

;

<msc body area> ::=
        ( <instance layer> <text layer>
           <event layer> <connector layer> ) set
;

<instance layer> ::=
        ( <instance area>* ) set
;

<text layer> ::=
        ( <text area>* ) set
;

<event layer> ::=
        <event area> | <event area> above <event layer>
;

<connector layer> ::=
        ( <message area>* | <incomplete message area>* |
          TSCSpecialArea* |
         <method call area>* | <incomplete method call area>*
         <reply area>* | <incomplete reply area>*) set
;

<event area> ::=
        <instance event area>
    |   <shared event area>
    |   <create area>
;

<instance event area> ::=
        (<message event area>
    |   <method call event area>
    |   <reply event area>
    |   <timer area>
    |   <concurrent area>
    |   <method area>
    |   <suspension area>
    |   <action area>
    |   <default area>)
;

<shared event area> ::=
    |   <condition area>
    |   <msc reference area>
    |   <in-line expression area>
;
```

C.4.4 Environment and Ports

Ports represent the interface between a test component and its environment. They are either explicitly represented by port instance or implicitly within communication messages to the environment (represented as communication to and from the frame symbol of a TSC).
There is a special TSC management port, which is implicitly given and which allows to gate the TTCN-3 configuration operations create and start and to gate the TTCN-3 behaviour statement return.

Communication to and from the frame symbol of an in-line expression is a shorthand notation for a communication to the environment.

```
TSCIn-linePortArea ::=
        TSCIn-lineOutPortArea | TSCIn-lineInPortArea |
        TSCIn-lineSpecialOutPortArea | TSCIn-lineSpecialInPortArea  |
        TSCIn-lineOutCallPortArea | TSCIn-lineInCallPortArea |
        TSCIn-lineOutReplyPortArea | TSCIn-lineInReplyPortArea
;

TSCIn-lineOutPortArea ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol> | <found message symbol> ) ]
;

TSCIn-lineInPortArea ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol> | <found message symbol> )  ]
;

TSCIn-lineOutCallPortArea ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol>| <found message symbol> ) ]
;

TSCIn-lineInCallPortArea ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol> | <found message symbol> ) ]
;

TSCIn-lineOutReplyPortArea ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol> | <found message symbol> ) ]
;

TSCIn-lineInReplyPortArea ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol> | <found message symbol> ) ]
;

TSCIn-lineCreateOutPortArea ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to <createline symbol>]
;

TSCIn-lineCreateInPortArea ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to <createline symbol> ]
;

<def port area> ::=
        ( <def out port area> |<def in port area>|
        <def order out port area> | <def order in port area>
        <def out call port area> | <def in call port area>
        <def out reply port area> | <def in reply port area>
        <def create out port area> | <def create in port area>
        <def special out port area> | <def special in port area> )
        is attached to <msc symbol>
;
```

```
<def out port area> ::=
        <void symbol>
        is attached to  ( <message symbol> | <found message symbol> )
;
```

The <message symbol> or <found message symbol> must have its arrow head end attached to the <def out port area>.

```
<def in port area> ::=
        <void symbol>
        is attached to <message symbol>
;
```

The <message symbol> must have its open end attached to the <def in port area>.

```
<def out call port area> ::=
        <void symbol>
        is attached to ( <message symbol> | <found message symbol> )
;
```

The <message symbol> or <found message symbol> must have its arrow head end attached to the <def out call port area>.

```
<def in call port area> ::=
        <void symbol>
        is attached to <message symbol>
;
```

The <message symbol> must have its open end attached to the <def in call port area>.

```
<def out reply port area> ::=
        <void symbol>
        is attached to ( <message symbol> | <found message symbol> )
;
```

The <message symbol> or <found message symbol> must have its arrow head end attached to the <def out reply port area>.

```
<def in reply port area> ::=
        <void symbol>
        is attached to <message symbol>
;
```

The <message symbol> or <lost message symbol> must have its open end attached to the <def in reply port area>.

```
<def create out port area> ::=
        <void symbol>
        is attached to <createline symbol>
;
```

The <createline symbol> must have its arrow head end attached to the <def create out port area>.

```
<def create in port area> ::=
        <void symbol>
        is attached to <createline symbol>
;
```

The <createline symbol> must have its open end attached to the <def create in port area>.

```
<def special out port area> ::=
        <void symbol>
        is attached to TSCSpecialMessageSymbol
;
```

The TSCSpecialMessageSymbol must have its arrow head end attached to the <def create out port area>.

```
<def special in port area> ::=
        <void symbol>
        is attached to TSCSpecialMessageSymbol
;
```

The TSCSpecialMessageSymbol must have its open end attached to the <def create in port area>.

C.4.5 Basic TSC

C.4.5.1 Instances (Component and Port Instances)

```
<instance area> ::=
        <instance fragment area> [ is followed by <instance area> ]
;

<instance fragment area> ::=
        <instance head area> is followed by <instance body area>
;

<instance head area> ::=
        InstanceHeadSymbol
        is associated with InstanceHead
        contains InstanceType
        [is attached to <createline symbol>]
        [is attached to ( TSCPortInstanceHeadArea * ) set]
;
```

Only instance head areas of port instances shall be attached to each other. The createline symbol shall be attached to component instance only.

```
TSCPortInstanceHeadArea ::=
        InstanceHeadSymbol
        is associated with PortInstanceHead
        contains PortInstanceType
;

InstanceHead ::=
        ComponentInstanceHead | PortInstanceHead
;

ComponentInstanceHead::=
        ComponentIdentifier | mtc | MTC
;

PortInstanceHead ::=
        Port
;

InstanceType ::=
        ComponentInstanceType | PortInstanceType
;

ComponentInstanceType ::=
         ComponentTypeIdentifier
;

PortInstanceType ::=
        [port]  PortTypeIdentifier
;

InstanceHeadSymbol ::=
        <instance head symbol> |
        PortInstanceHeadSymbol
;

<instance head symbol> ::=

;
```

The <instance head symbol> can be used for test component instances and for port instances. In the latter case, the keyword **port** (see InstanceType) shall be used in addition.

```
PortInstanceHeadSymbol  ::=

;
```

The PortInstanceHeadSymbol shall be used for port instances only.

```
<instance body area> ::=
        InstanceAxisSymbol
        is followed by ( InstanceEndSymbol | <stop symbol> )
;

InstanceAxisSymbol ::=
        <instance axis symbol> |
        PortInstanceAxisSymbol
;

PortInstanceAxisSymbol ::=

;
```

The PortInstanceAxisSymbol shall be used for port instances only.

```
<instance axis symbol> ::=
         ( <instance axis symbol1> | <instance axis symbol2> )
        is attached to ( <event area>* ) set
;

<instance axis symbol1> ::=

;

<instance axis symbol2> ::=

;

InstanceEndSymbol ::=
        <instance end symbol> |
        PortInstanceEndSymbol
;

PortInstanceEndSymbol ::=

;
```

The PortInstanceEndSymbol shall be used for port instances only.

```
<instance end symbol> ::=

;
```

C.4.5.2 Messages

```
<message event area> ::=
        <message out area> | <message in area> |
        TSCSpecialOutArea | TSCSpecialInArea
;
```

Message shall never be attached to two port instances.

```
<message out area> ::=
        <message out symbol>
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
;

<message out symbol> ::=
        <void symbol>
;

<void symbol> ::=       .
;

<message in area> ::=
        <message in symbol>
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
;

<message in symbol> ::=
        <void symbol>
;

<message area> ::=
        ( <message symbol> [ above it  TSCMessage_PortsAndType ] )
        [ is associated with [ “[“ ] TSCTemplate [ “]" ] ]
        is attached to (<message start area> | <message end area>)
;

TSCMessage_PortsAndType ::=
        [ TSCSourcePort ] [ Type | any ] [ TSCDestinationPort ]
;

TSCSourcePort::=
        [ ( Port [ “:"  PortTypeIdentifier “>" ] ) | any ]
;

TSCDestinationPort::=
         [“>" Port [ “:"  PortTypeIdentifier] ]
;

<message start area> ::=
        <message out area> | <def in port area> | <in-line port area>
;

<message end area> ::=
        <message in area> | <def out port area> | <in-line port area>
;

<message symbol> ::=
        ──────────────►
;

<incomplete message area> ::=
        <found message area>
;

<found message area> ::=
        ( <found message symbol>  [ above ( Type | any )] )
        [ is associated with [ “[“ ] TSCTemplate [ “]" ] ]
        [ is associated with  Port ]
        is attached to <message end area>
;
```

At least the type, any or the template shall be given.

```
TSCTemplate ::=
        ( TemplateInstance | any ) [ FromClause ] [ PortRedirect ]
;

<found message symbol> ::=

;
```

C.4.5.3 Control Flow

```
<method call area> ::=
        ( <message symbol>
        above ( call | getcall | check ) [ TSCSignatureType | any ] )
        is associated with
        ( TSCCallSendTemplate | TSCCallReceiveTemplate )
        is attached to ( <method call start area> | <method call end area> )
        is attached to InstanceAxisSymbol
        [is attached to (<method area>)]
;
```

Receive templates and any shall be used only in combination with getcall and check. Method areas shall be attached only to test component instances.

See [1] for detailed information on the use of call, getcall and check operation.

```
TSCSignatureType ::=
        Signature [ [ ":" ] DerivedDef  ":=" ]
;

TSCCallSendTemplate ::=
        ( ArrayValueOrAttrib | FieldSpecList  ) [ ToClause ]
;

TSCCallReceiveTemplate ::=
        [ ( ArrayValueOrAttrib | FieldSpecList  ) ]
        [ FromClause ] [( PortRedirectWithParam | PortRedirect ) ]
;
```

Port redirect shall be used only in combination with check.

```
<method call start area> ::=
        ( <call out area> | <def in port area> | <in-line port area> )
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
        [ is attached to <suspension symbol> ]
;
```

The suspension symbol can only be used on component instances. Calls without suspension symbol represent nowait calls. Suspension regions shall be attached only to test component instances.

```
<method call end area> ::=
        ( <call in area> | <def out port area> | <in-line port area> )
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
;

<reply area> ::=
        ( <message symbol>
        above ( raise | reply | getreply | catch | check )
            [ Signature | TSCSignatureType | any ] )
        is associated with
        ( TSCReplySendTemplate | TSCReplyReceiveTemplate )
        is attached to ( <reply start area> <reply end area>)
;
```

Signature shall be used only for raise. Receive templates and any shall be used only in combination with getreply, catch, and check.

See [1] for detailed information on the use of raise, reply, getreply, catch, and check operation.

```
TSCReplySendTemplate ::=
        ( ArrayValueOrAttrib | FieldSpecList  ) [ ReplyValue ] [ ToClause ]
;
```

Reply value shall be used only in combination with reply.

```
TSCReplyReceiveTemplate ::=
        [ ( ArrayValueOrAttrib | FieldSpecList  ) ] [ ValueMatchSpec ]
        [ FromClause ] [( PortRedirectWithParam | PortRedirect ) ]
;
```

Port redirect shall be used only in combination with catch or check.

```
<reply start area> ::=
        ( <reply out area> | <def in port area> | <in-line port area> )
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
        is attached to <method symbol>
;
```

The method symbol can only be used on component instances.

```
<reply end area> ::=
        ( <reply in area> | <def out port area> | <in-line port area> )
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
        [ is attached to <suspension symbol> ]
;
```

For non-blocking calls, the suspension symbol is not attached.

```
<incomplete method call area> ::=
        <found method call area>
;

<found method call area> ::=
        ( <found message symbol> above getcall [ TSCSignatureType | any ] )
        [ is associated with TSCCallReceiveTemplate ]
        [ is associated with Port]
        is attached to <method call end area>
;
```

Either the signature template or an any shall be given. The port identifier shall not be used in combination with a signature template.

```
<incomplete reply area> ::=
        <found reply area>
;

<found reply area> ::=
        <found message symbol> is associated with <msg identification>
        [ is associated with ( <instance name> | <port name> ) ]
        is attached to <reply end area>
;

<method call event area> ::=
        (<call out area> | <call in area>)
;

<call out area> ::=
        <call out symbol>
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
;

<call out symbol> ::=
        <void symbol>
;
```

```
<call in area> ::=
        <call in symbol>
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
;

<call in symbol> ::=
        <void symbol>
;

<reply event area> ::=
        (<reply out area> | <reply in area>)
;

<reply out area> ::=
        <reply out symbol>
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
;

<reply out symbol> ::=
        <void symbol>
;

<reply in area> ::=
        <reply in symbol>
        is attached to InstanceAxisSymbol
        is attached to <message symbol>
;

<reply in symbol> ::=
        <void symbol>
;

<method area> ::=
        <method symbol>
        is attached to <instance axis symbol>
        [contains <method event layer>]
;

<method event layer>::=
        <method event area> | <method event area> above <method event layer>
;

<method event area>::=
        <event area>
;

<method symbol> ::=

;

<suspension area> ::=
        <suspension symbol>
        is attached to <instance axis symbol>
        [contains <suspension event layer>]
;

<suspension event layer>::=
        <method invocation area>
        | <method invocation area> above <suspension event layer>
;

<method invocation area> ::=
        <method start area>
        is followed by <method area>
        is followed by <method end area>
;

<method start area> ::=
        <call in area> | <found method call area>
;
```

```
<method end area> ::=
        <reply out area> | <lost reply area>
;

<suspension symbol> ::=

;
```

C.4.5.4 Special Messages

```
TSCSpecialOutArea  ::=
        <message out symbol>
        is attached to <instance axis symbol>
        is attached to TSCSpecialMessageSymbol
;

TSCSpecialInArea ::=
        <message in symbol>
        is attached to <instance axis symbol>
        is attached to TSCSpecialMessageSymbol
;

TSCSpecialArea ::=
        TSCSpecialMessageSymbol
        is associated with TSCSpecialMessages
        is attached to ( TSCSpecialStartArea  | TSCSpecialEndArea )
;

TSCSpecialMessages ::=
        ( [ ComponentIdentifier "." ] StartKeyword [ "(" FunctionInstance ")" ] ) |
        ( [ ComponentType "." ] CreateKeyword ) |
        ( [ ComponentIdentifier "." ] ReturnKeyword [ "(" Expression ")" ] ) |
        ( [ PortOrAll "." ]  PortClearOp ) |
        ( [ PortOrAll "." ] PortStartOp ) |
        ( [ PortOrAll "." ] PortStopOp )
;

TSCSpecialStartArea ::=
        TSCSpecialOutArea  | <def in port area> | <in-line port area>
;

TSCSpecialEndArea ::=
        TSCSpecialEndArea | <def out port area> | <in-line port area>
;

TSCSpecialMessageSymbol ::=

;
```

C.4.5.5 Environment and Ports

```
<in-line port area> ::=
        ( <in-line out port area> | <in-line in port area> |
        <in-line create out port area> | <in-line create in port area> |
        <in-line out call port area> | <in-line in call port area> |
        <in-line out reply port area> | <in-line in reply port area> )
        [ is associated with <port identification> ]
;

<in-line out port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol>| <found message symbol> ) ]
        [ is attached to ( <message symbol> | <lost message symbol> ) ]
;
```

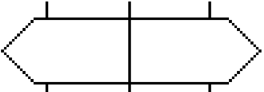
```
<in-line in port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol> | <lost message symbol>) ]
        [ is attached to ( <message symbol> | <found message symbol>) ]
;

<in-line out call port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol>| <found message symbol> ) ]
        [ is attached to ( <message symbol> | <lost message symbol> ) ]
;

<in-line in call port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol> | <lost message symbol>) ]
        [ is attached to ( <message symbol> | <found message symbol>) ]
;

<in-line out reply port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol>| <found message symbol> ) ]
        [ is attached to ( <message symbol> | <lost message symbol> ) ]
;

<in-line in reply port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to ( <message symbol> | <lost message symbol>) ]
        [ is attached to ( <message symbol> | <found message symbol>) ]
;

<in-line create out port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to <createline symbol>]
        [ is attached to <createline symbol> ]
;

<in-line create in port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
        [ is attached to <createline symbol>]
        [ is attached to <createline symbol> ]
;

<in-line order port area> ::=
        <in-line order out port area> | <in-line order in port area>
;

<in-line order out port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
;

<in-line order in port area> ::=
        <void symbol>
        is attached to <in-line expression symbol>
;

<def port area> ::=
        ( <def out port area> |<def in port area>|
        <def out call port area> | <def in call port area>
        <def out reply port area> | <def in reply port area>
        <def create out port area> | <def create in port area>
        <def special out port area> | <def special in port area>)
        is attached to <msc symbol>
;

<def out port area> ::=
        <void symbol>
        is attached to ( <message symbol> | <found message symbol> )
;
```

```
<def in port area> ::=
        <void symbol>
        is attached to  <message symbol>
;

<def out call port area> ::=
        <void symbol>
        is attached to ( <message symbol> | <found message symbol> )
;

<def in call port area> ::=
        <void symbol>
        is attached to <message symbol>
;

<def out reply port area> ::=
        <void symbol>
        is attached to ( <message symbol> | <found message symbol> )
;

<def in reply port area> ::=
        <void symbol>
        is attached to <message symbol>
;

<def create out port area> ::=
        <void symbol>
        is attached to <createline symbol>
;

<def create in port area> ::=
        <void symbol>
        is attached to <createline symbol>
;
```

C.4.5.6 Conditions

```
<condition area> ::=
        <condition symbol>
        contains <condition text> [<shared>]
        is attached to ( <instance axis symbol>* ) set
;

<condition symbol> ::=

;
```

If a shared <condition> crosses an <instance axis symbol> which is not involved in this condition the <instance axis symbol> is drawn through:



```
<condition text> ::=
        <condition name list> |
        when ( <condition name list> | "(" BooleanExpression ")" ) |
        "[" BooleanExpression "]" |
        otherwise | else |
        pass |  PASS |  inconc |  INCONC |
        fail |  FAIL |  none |  NONE |  error |  ERROR
;
```

C.4.5.7 Timers

```
<timer area> ::=
        ( <timer start area> | <timer stop area> | <timeout area> )
;

<timer start area> ::=
        <timer start area1> | <timer start area2>
;

<timer start area1> ::=
        <timer start symbol>
        is associated with [ TimerRef  ]  [ TimerValue ]
        is attached to <instance axis symbol>
        [is attached to (<restart symbol> | <timer stop symbol2>
        | <timeout symbol3> ) ]
;

<timer start area2> ::=
        <restart symbol>
        is associated with [ TimerRef  ] [ TimerValue ]
        is attached to <instance axis symbol>
        is attached to <timer start symbol>
        [is attached to ( <timer stop symbol2> | <timeout symbol3> ) ]
;

<timer start symbol> ::=
        <start symbol1> | <start symbol2>
;

<start symbol1> ::=
;

<start symbol2> ::=
;

<restart symbol> ::=
;

<timer stop area> ::=
        <timer stop area1> | <timer stop area2>
;

<timer stop area1> ::=
        <timer stop symbol1> [ is associated with TimerRef  ]
        is attached to <instance axis symbol>
;

<timer stop area2> ::=
        <timer stop symbol2> [ is associated with TimerRef  ]
        is attached to <instance axis symbol>
        is attached to ( <timer start symbol> | <restart symbol> )
;

<timer stop symbol> ::=
        <timer stop symbol1> | <timer stop symbol2>
;

<timer stop symbol1> ::=
;
```

<timer stop symbol2> ::=

;

<timeout area> ::=
<timeout area1> | <timeout area2>
;

<timeout area1> ::=
<timeout symbol> [***is associated with*** TimerRef]
[(<parameter list>)]
is attached to <instance axis symbol>
;

<timeout symbol> ::=
<timeout symbol1> | <timeout symbol2>
;

<timeout symbol1> ::=

;

<timeout symbol2> ::=

;

<timeout area2> ::=
<timeout symbol3> [***is associated with*** TimerRef]
is attached to <instance axis symbol>
is attached to (<timer start symbol> | <restart symbol>)
;

<timeout symbol3> ::=

;

C.4.5.8 Actions

<action area> ::=
<action symbol>
is attached to <instance axis symbol>
contains ({ FunctionStatement }+)
;

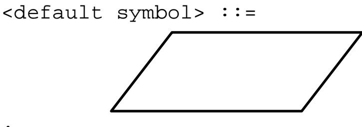
<action symbol> ::=

;

C.4.5.9 Defaults

<default area> ::=
<default symbol>
is attached to <instance axis symbol>
contains ([**activate** | **deactivate**]"(" FunctionIdentifier ")")
;

```
<default symbol> ::=
```



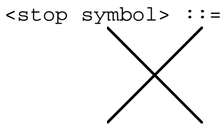
```
;
```

C.4.5.10 Instance creation

```
<create area> ::=
        <createline symbol>
        [ is associated with [ VariableRef ":=" ]
                     ComponentType "." CreateKeyword]
        is attached to
        ( (<instance axis symbol> | <def create in port area> )
        ( <instance head area> | <def create out port area> )) set
;

<createline symbol> ::=
        — — — ►
;
```

C.4.5.11 Instance stop

```
<stop symbol> ::=
```



```
;
```

C.4.6 Structural concepts

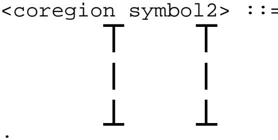
C.4.6.1 Co-regions

```
<concurrent area> ::=
        <coregion symbol>
        is attached to <instance axis symbol>
        contains <coevent layer>
;

<coregion symbol> ::=
        <coregion symbol1> | <coregion symbol2>
;

<coevent layer> ::=
        <coevent area> |
        <coevent area> above <coevent layer>
;

<coevent area> ::=
        ( <message event area> |
         <incomplete message area> | <action area> |
         <timer area> | <create area> )*
;
```

Only receive events are allowed in a coregion.

```
<coregion symbol1> ::=
;
```

```
<coregion symbol2> ::=
```



```
;
```

C.4.6.2 In-line expressions

```
<in-line expression area> ::=
        ( <loop area> | <alt area> | <int area> )
        [is attached to <msc symbol>]
;

<loop area> ::=
        <in-line expression symbol> contains
        ( loop [ <loop boundary> ] <operand area> )
        is attached to ( InstanceAxisSymbol * ) set
;

<alt area> ::=
        <in-line expression symbol> contains
        ( alt <operand area>
        ( is followed by <separator area>
           is followed by <operand area> )* )
        is attached to ( InstanceAxisSymbol * ) set
;

<int area> ::=
        <in-line expression symbol> contains
        ( int <operand area>
        ( is followed by <separator area>
           is followed by <operand area> )* )
        is attached to ( InstanceAxisSymbol * ) set
;

<in-line expression symbol> ::=

;

<operand area> ::=
        ( <event layer> )* set
;

<separator area> ::=
        <separator symbol>
;

<separator symbol> ::=

;
```

C.4.6.3 TSC references

```
<msc reference area> ::=
        <msc reference symbol>
        [ is attached to <time interval area> ]
        contains ( TSCReference ) set
        is attached to ( InstanceAxisSymbol * ) set
        is attached to ( TSCConnectionArea * ) set
        is attached to ( TSCConnectionArea * ) set
;
```

```
TSCReference ::=
        ( [ VariableRef ":=" ]
            ( TestcaseIdentifier | FunctionIdentifier | ([ expand ] NamedAltIdentifier ))
            [ "(" [FunctionFormalParList] ")" ] [ duration FloatValue ] )
        | FunctionBody
        | Comment
;
```

An assignment shall be used only for references to test cases, i.e. within control HTSCs. A time interval or duration shall be used only for references to test cases, i.e. within control HTSCs.

```
<msc reference symbol> ::=
;

<time interval area> ::=
        <int symbol>
        is associated with FloatValue
        is followed by ( <cont interval> | <interval area 2>)
;

<interval area 2> ::=
        <int symbol>
        [is associated with <time interval>]
;

<cont interval> ::=
        <cont int symbol>
        is associated with <interval name>
;

<int symbol> ::=
        {<int symbol 1> | <int symbol 2> }
        is attached to <msc reference symbol>
;

<int symbol 1> ::=
;

<int symbol 2> ::=
;

<cont int symbol> ::=
;
```

C.4.7 High-level TSC (HTSC)

```
HTSCHeading ::=
        tsc
        ( FunctionIdentifier [ "(" [FunctionFormalParList] ")" ]
           [ ReturnType] )
        { VarInstance } [ <end> ]
;
```

The HTSC, which defines the control part for the module (indicated with the **control** keyword within the control area of the TSC document) shall have neither formal parameters nor a return type. Only those HTSCs that are referenced within the control part of the TSC document shall have local variables.

```
HTSCExprArea ::=
        (<text layer> <start area> HTSCNodeExpressionArea*
        <hmsc end area>* ) set
;

<start area> ::=
        <hmsc start symbol> is followed by ( <alt op area>+ ) set
;

<hmsc start symbol> ::=
        ▽
;

<hmsc end area> ::=
        <hmsc end symbol> is attached to ( <hmsc line symbol>+) set
;

<hmsc end symbol> ::=
        △
;

<hmsc line symbol> ::=
        <hmsc line symbol1> | <hmsc line symbol2>
;

<hmsc line symbol1> ::=
        |
;

<hmsc line symbol2> ::=
        ↓
;

<alt op area> ::=
        <hmsc line symbol>
        is attached to ( HTSCNodeArea | <hmsc end symbol> )
;

HTSCNodeExpressionArea ::=
        HTSCNodeArea is followed by ( <alt op area>+) set
        is attached to ( <hmsc line symbol>+) set
;

HTSCNode area ::=
        <hmsc reference area>
    |   <connection point symbol>
    |   <hmsc condition area>
    |   <par expr area>
    |   HTSCActionArea
;
```

```
<hmsc reference area> ::= <msc reference symbol>
        contains <msc ref expr> [time <time interval> ]
;

<connection point symbol> ::=

;

<hmsc condition area> ::=
        <condition symbol> contains <condition text>
;

<par expr area> ::=
        <par frame symbol>
        is attached to <hmsc line symbol>
        contains ( HTSCExprArea +) set
;
```

Parallel expressions are not allowed for HTSCs representing a control function.

```
<par frame symbol> ::=
        <frame symbol>
;

HTSCActionArea  ::=
        <action symbol>
        is attached to <hmsc line symbol>
        contains { ( Assignment | LogStatement )
                [ ";"( Assignment | LogStatement ) ] }
;
```

Actions shall be used only within a HTSC representing a control function.

Annex D: Mapping TSC to TTCN-3

This annex defines the mapping of TSC/gr to TTCN-3 core notation [1].

D.1 Description

A denotational style for describing the mapping of TSC/gr syntax onto TTCN-3 syntax is used. For each TSC/gr production rule one or more functions, denoted using "[[" and "]]", each labelled using a permissible syntax for the production is defined. These functions may also carry parameters. The definition of each function is given by the rules on the right-hand side, which may in-turn invoke other mapping functions. All rules are terminated with a semi-colon. For example:

```
 [[ group identifier <body> ]] par1 := if par1 then
                                       group identifier  "{"
                                               [[ <body> ]]
                                       "}" [ ";" ]
;
```

In this example a single function is defined that is labelled with the syntax "**group** identifier <body>", which takes a single boolean parameter "*par1*" as an argument. If the parameter is *true* the function returns the TTCN-3 syntax for a simple group definition, otherwise it returns nothing. This function also calls another mapping function for the <body> production rule, which returns the syntax for the body production. Note that the inclusion of the semi-colon is optional.

D.1.1 TSC documents

D.1.1.1 TSCDocument

```
 [[ <frame symbol>
    contains TSCDocumentHead
    [ is followed by ControlPartArea ]
    [ is followed by TestcasePartArea ]
    [ is followed by FunctionPartArea ]
    [ is followed by NamedAltPartArea ]
 ]] ::=
        module [[TSCDocumentHead ]]
        [[ TestcasePartArea ]]
        [[ FunctionPartArea ]]
        [[ NamedAltPartArea ]]
        [[ ControlPartArea ]]
        "}"
        with "{"
                [ removeWithKeyword ( TSCDocumentHead.WithStatement ) ]
            display "ETSI TTCN-3 TSC v1.0",
            display TSCDocument
        "}" [  ";" ]
 ;
```

Where, the *removeWithKeyword* function allows the merger of with statements by simply removing the **with** keyword.

D.1.1.2 TSCDocumentHead

```
[[  tscdocument TTCN3ModuleId [ ModuleParList ]
    [ related to  Identifier { "," Identifier } ]
    explicit_typing [WithStatement] <end>
    [ TTCN3DataDefinitionPart]
    [ <parenthesis declaration> ]
 ]] ::=
        TTCN3ModuleId  [ ModuleParList ]
        "{"
        [[TTCN3DataDefinitionPart]] (explicit_typing)
;
[[  tscdocument TTCN3ModuleId [ ModuleParList ]
    [ related to  Identifier { "," Identifier } ]
```

```
    implicit_typing  [WithStatement] <end>
    [ TTCN3DataDefinitionPart]
    [ <parenthesis declaration> ]
 ]] ::=
        TTCN3ModuleId  [ ModuleParList ]
        "{"
        [[TTCN3DataDefinitionPart]] (implicit_typing)
;
The 'related to' and 'parenthesis declaration' productions are not mapped to TTCN-3.
TTCN3DataDefinitionPart
[[  BeginChar
    TTCN3DataDefinitionList
    EndChar
]] (typing) ::=
         [[ TTCN3DataDefinitionList  ]]
         if typing == implicit_typing then
             derive( PortDef ) derive ( ComponentDef )
;
```

The *derive* function determines the typing of ports or components from the message interactions between ports and component instances.

D.1.1.2.1 TTCN3DataDefinitionList

```
[[ { TTCN3DataDefinition [ WithStatement ] } ]] ::=
        { [[ TTCN3DataDefinition ]] [WithStatement ] [ ";" ] }
;
```

D.1.1.2.2 TTCN3DataDefinition

```
[[  TypeDef | ConstDef | ExtConstDef
    TemplateDef | FunctionDef | SignatureDef |
    ImportDef | TSCGroupDef | ExtFunctionDef |
]] ::=
        TypeDef | ConstDef | ExtConstDef  |
        TemplateDef | FunctionDef | SignatureDef |
        ImportDef | [[ TSCGroupDef ]] | ExtFunctionDef
;
TSCGroupDef
[[  group GroupIdentifier
    BeginChar
    TTCN3DataDefinitionList
    EndChar
]] ::=  group GroupIdentifier
        "{"
        [[ TTCN3DataDefinitionList ]]
        "}"
;
```

D.1.1.3 ControlPartArea

```
[[  control
    TSCControlReferenceList [ ";" TSCFunctionGroupDefList ]
    is followed by <separator area>
]] ::=
         [[ TSCControlReferenceList ]] ([[ TSCFunctionGroupDefList ]], defs_only)
         [[ TSCControlReferenceList ]] ([[ TSCFunctionGroupDefList ]], control_only)
;
[[  control
    is followed by (TSCControlReferenceArea *) set
    is followed by <separator area>
]] ::=
for all members of the (TSCControlReferenceArea *) set do [[ TSCControlReferenceArea ]] (defs_only)
for all members of the (TSCControlReferenceArea *) set do [[ TSCControlReferenceArea ]]
(control_only)
;
```

Where, *defs_only* and *control_only* are distinct flags to indication type definitions only or control definition respectively.

D.1.1.3.1 TSCControlReferenceList

```
 [[ TSCControlReference { ","TSCControlReference }]] (e,f) ::=
        {  [[TSCControlReference]] (e, f )  }
;
TSCControlReference
[[ control  FunctionIdentifier ]] (e,f) ::=
         if f == control_only then
            control
            "{"
            [[ process_tsc_diagram(FunctionIdentfier) ]]
            "}" [ ";" ]
;
[[ FunctionIdentifier ]]  (e,f)  ::=
         if f == defs_only then
            [[ process_tsc_diagram (FunctionIdentfier) ]] [ ";" ]
;
[[ GroupIdentifier ]]  (e,f) ::=
        if f == defs_only then
            group GroupIdentifier
            "{"
            [[ get_group_def(GroupIdentfier,e) ]] (e)
            "}" [ ";"]
;
```

Where,

- *process_tsc_diagram* is a function which processes the TSC diagram with the given identity,
- *get_group_def* is a function that returns the list of group or function identifiers associated with the group.

D.1.1.3.2 TSCFunctionorGroupReferenceList

```
[[ TSCFunctionorGroupRef { ","TSCFunctionorGroupRef } ]] e ::=
        [[ TSCFunctionorGroupRef ]] (e)  { [ ";" ] [[ TSCFunctionorGroupRef ]] (e) }
;
```

D.1.1.3.3 TSCFunctionorGroupRef

```
[[_GroupIdentifier_]] e ::=     group GroupIdentifier
        "{"
        [[ get_group_def(GroupIdentfier,e) ]] (e)
        "}" [ ";"]
;
[[ FunctionIdentifier ]] e::=  function [[ process_tsc_diagram (FunctionIdentfier) ]] [ ";" ]
;
```

D.1.1.3.4 TSCControlReferenceArea

```
[[  { <msc reference symbol> contains
    control FunctionIdentifier
]] f ::=
            if f == control_only then
                control
                "{"
                [[ process_tsc_diagram (FunctionIdentfier) ]]
                "}" [ ";" ]
;
[[  <msc reference symbol> contains
    FunctionIdentifier

]] f ::=
            if f == defs_only then
                [[ process_tsc_diagram (FunctionIdentfier) ]] [ ";" ]
;
```

```
[[  <msc reference symbol> contains
    GroupIdentifier
]] f ::=
            if f == defs_only then
                group GroupIdentifier
                “{“
                for all members group TSC retunred by  (get_TSC_Diagram(GroupIdentfier) do
                    [[ TSCControlReferenceArea ]] (f)
                “}" [ “;"]
;
```

Where, *get_TSC_Diagram* is a function that returns the TSC with the given identifier.

D.1.1.4 TestcasePartArea

```
[[  testcases
    ( TSCTestcaseorGroupReferenceList [ “;" TSCTestcaseGroupDefList ]
    is followed by <separator area>
]] ::=
            [[ TSCTestcaseorGroupReferenceList ]] ([[ TSCTestcaseGroupDefList ]])
;
[[  testcases is followed by ( TSCTestcaseReferenceArea *) set )
    is followed by <separator area>
]] ::=
for all members of the (TSCTestcaseReferenceArea *) set do [[ TSCTestcaseReferenceArea ]]
;
```

D.1.1.4.1 TSCTestcaseorGroupReferenceList

```
[[TSCTestcaseorGroupRef { “,"TSCTestcaseorGroupRef } ]] e ::=
        [[ TSCTestcaseorGroupRef ]] (e) { [ “;"]  [[ TSCTestcaseorGroupRef ]] (e)  }
;
TSCTestcaseorGroupRef
[[_GroupIdentifier_]] e ::=     group GroupIdentifier
        “{“
        [[ get_group_def(GroupIdentfier,e) ]] (e)
        “}" [ “;"]
;
[[ TestCaseIdentifier ]] e::=  testcase [[ process_tsc_diagram (TestCaseIdentfier) ]] [ “;" ]
;
```

D.1.1.4.2 TSCTestcaseReferenceArea

```
[[  <msc reference symbol> contains GroupIdentifier
]] ::=
            group GroupIdentifier
            “{“
            for all members group TSC retunred by  (get_TSC_Diagram(GroupIdentfier) do
                    [[ TSCTestcaseReferenceArea ]]
            “}" [ “;"]
;
[[  <msc reference symbol> contains TestcaseIdentifier
]] ::=
            testcase  [[ process_tsc_diagram(TestcaseIdentfier) ]]
            [ “;"]
;
```

D.1.1.5 FunctionPartArea

```
[[  functions
    ( TSCFunctionorGroupReferenceList [ “;" TSCFunctionGroupDefList ]
    is followed by <separator area>
]] ::=
            [[ TSCFunctionorGroupReferenceList ]] ([[ TSCFunctionGroupDefList ]])
;
[[  functions is followed by ( TSCFunctionReferenceArea *) set )
    is followed by <separator area>
]] ::=
for all members of the (TSCFunctionReferenceArea *) set do [[ TSCFunctionReferenceArea ]]
;
```

D.1.1.5.1 TSCFunctionReferenceArea

```
[[  <msc reference symbol> contains GroupIdentifier
]] ::=
            group GroupIdentifier
            "{"
            for all members group TSC retunred by  (get_TSC_Diagram(GroupIdentfier) do
                    [[ TSCFunctionReferenceArea ]]
            "}" [ ";"]
;
[[  <msc reference symbol> contains FunctionIdentifier
]] ::=
            function [[process_tsc_diagram(FunctionIdentfier) ]]
            [ ";"]
;
```

D.1.1.6 NamedAltPartArea

```
[[  named alts
    ( TSCNamedAltorGroupReferenceList [ ";" TSCNamedAltGroupDefList ]
    is followed by <separator area>
]] ::=
            [[ TSCNamedAltorGroupReferenceList ]] ([[ TSCNamedAltGroupDefList ]])
;
[[  named alts is followed by ( TSCNamedAltReferenceArea *) set )
    is followed by <separator area>
]] ::=
for all members of the (TSCNamedAltReferenceArea *) set do [[ TSCNamedAltReferenceArea ]]
;
```

D.1.1.6.1 TSCNamedAltorGroupReferenceList

```
[[TSCNamedAltorGroupRef { ","TSCNamedAltorGroupRef } ]] e ::=
        [[ TSCNamedAltorGroupRef ]] (e) { [ ";"]  [[ TSCNamedAltorGroupRef ]] (e) }
;
TSCNamedAltorGroupRef
[[_GroupIdentifier_]] e ::=     group GroupIdentifier
        "{"
        [[ get_group_def(GroupIdentfier,e) ]] (e)
        "}" [ ";"]
;
[[NamedAltIdentifier ]] e ::=  named alt [[ process_tsc_diagram (NamedAltIdentfier) ]] [ ";" ]
;
```

D.1.1.6.2 TSCNamedAltReferenceArea

```
[[  <msc reference symbol> contains GroupIdentifier
]] ::=
            group GroupIdentifier
            "{"
            for all members group TSC retunred by  (get_TSC_Diagram(GroupIdentfier) do
                    [[ TSCNamedAltReferenceArea ]]
            "}" [ ";"]
;
[[  <msc reference symbol> contains NamedAlteIdentifier
]] ::=
            named alt [[ process_tsc_diagram(NamedAltIdentfier) ]]
            [ ";"]
;
```

Annex E: An INRES example in TSC

The TSC example provided here uses the TTCN-3 specification given in annex A which provides sequential and concurrent test cases for the INRES protocol. The sequential test cases define identical test behaviour but reflect the various specification styles of TTCN-3 core notation. These styles can also be reflected in TSC. See annex B for further information.

E.1 TSC document

The TSC document for the INRES example includes the declaration for the module and declares the various TSCs for test cases and functions, which are presented in subsequent clauses.

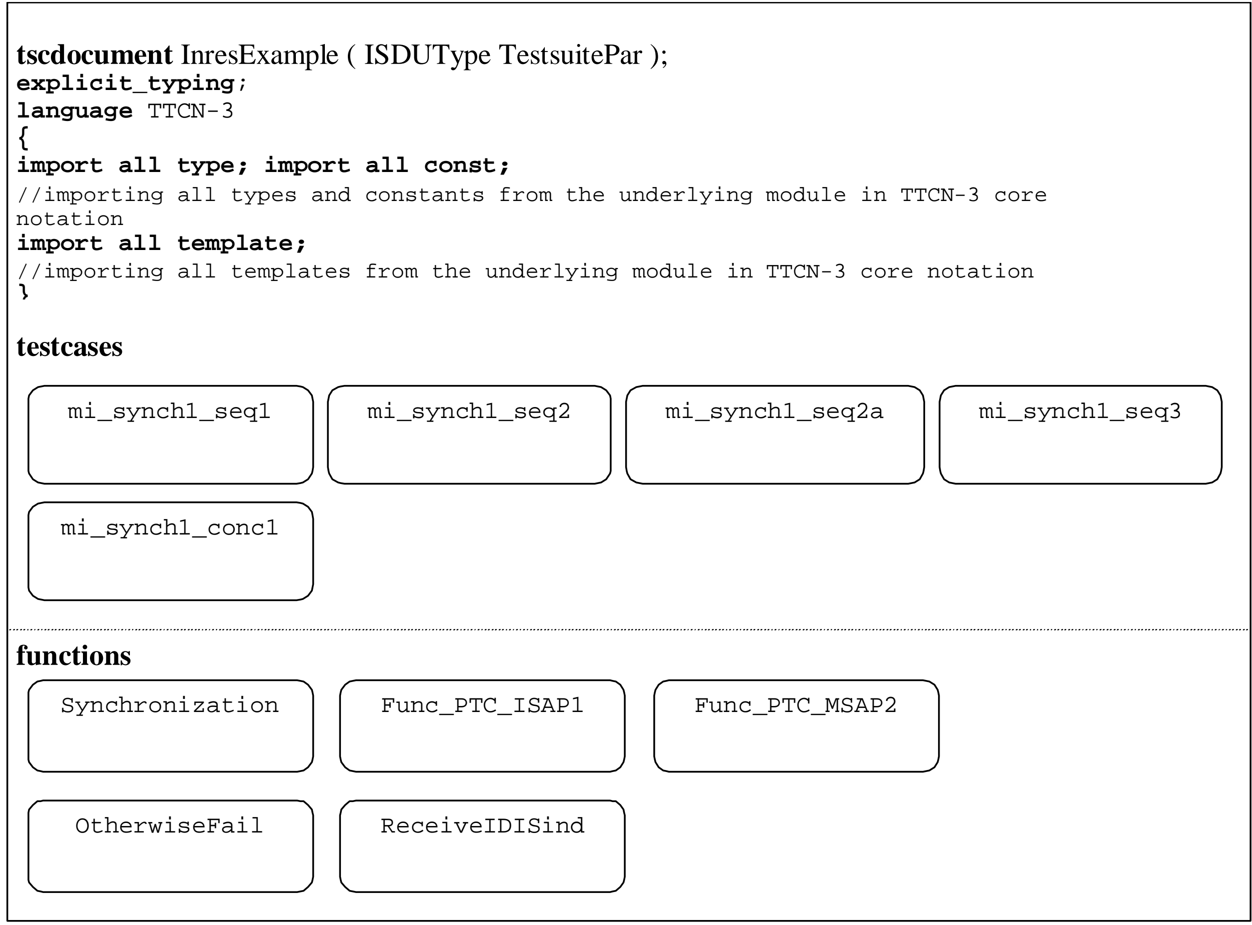


Figure E.1: TSC document for the INRES Example

E.2 TSCs for sequential test case

E.2.1 First version

The sequential test cases use the test component MTC and two port instances ISAP1 and MSAP2. After the default activation, the main test sequence leading to PASS is represented. An Hyper-TSC is used to represent alternatives to the PASS path. The TTCN-3 core notation description of this test case has a stop operation and the test verdict within all branches of the "behaviour tree". In case of the TSC representation, this kind of description is not supported. Instead, the verdict assignment and stop is represented only once.

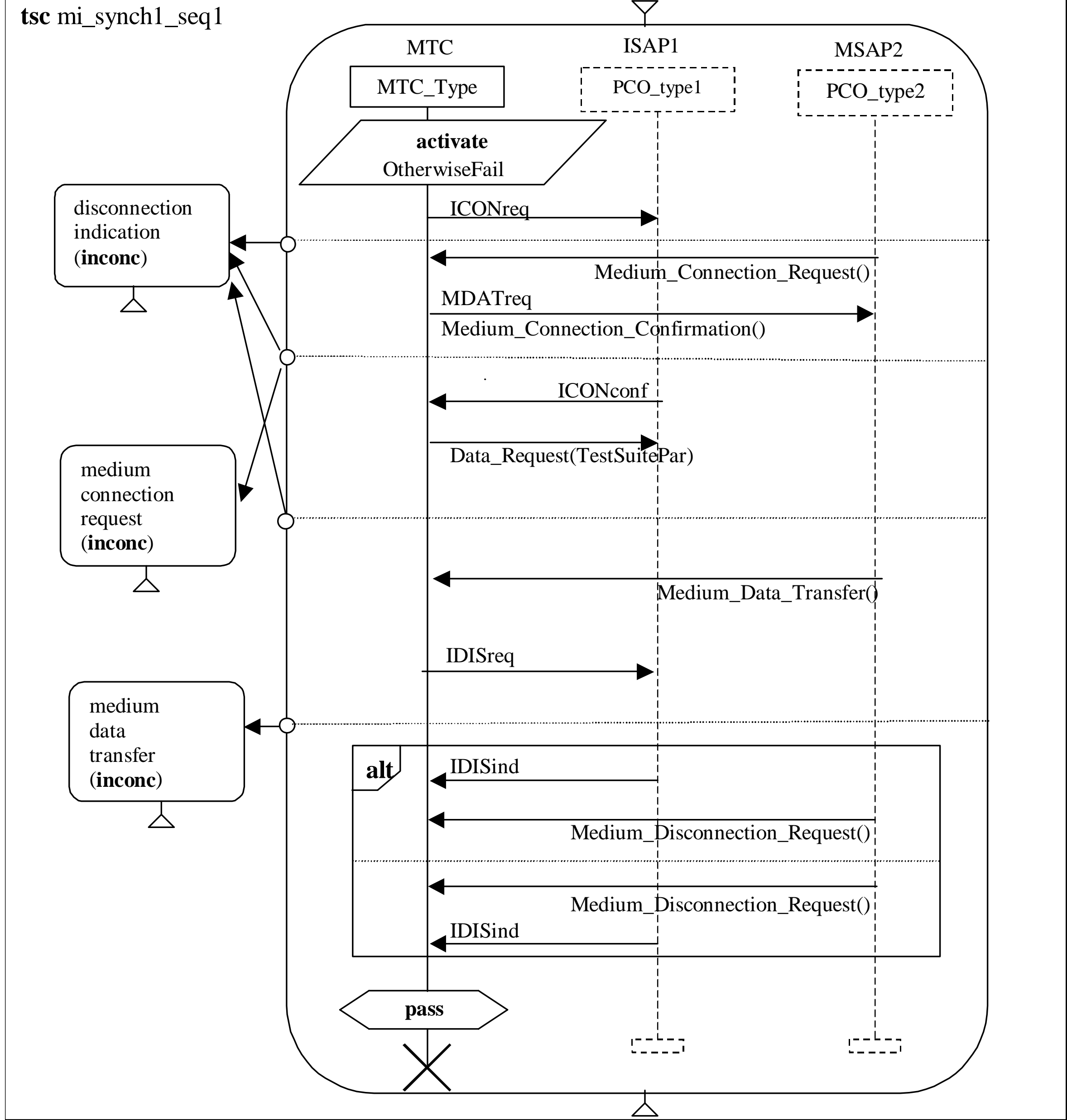


Figure E.2: INRES: sequential test case, first version

This TSC defines the default used throughout this example. Whenever any other message is received from ISAP1 or MSAP2, a fail verdict is assigned and the test case stops.

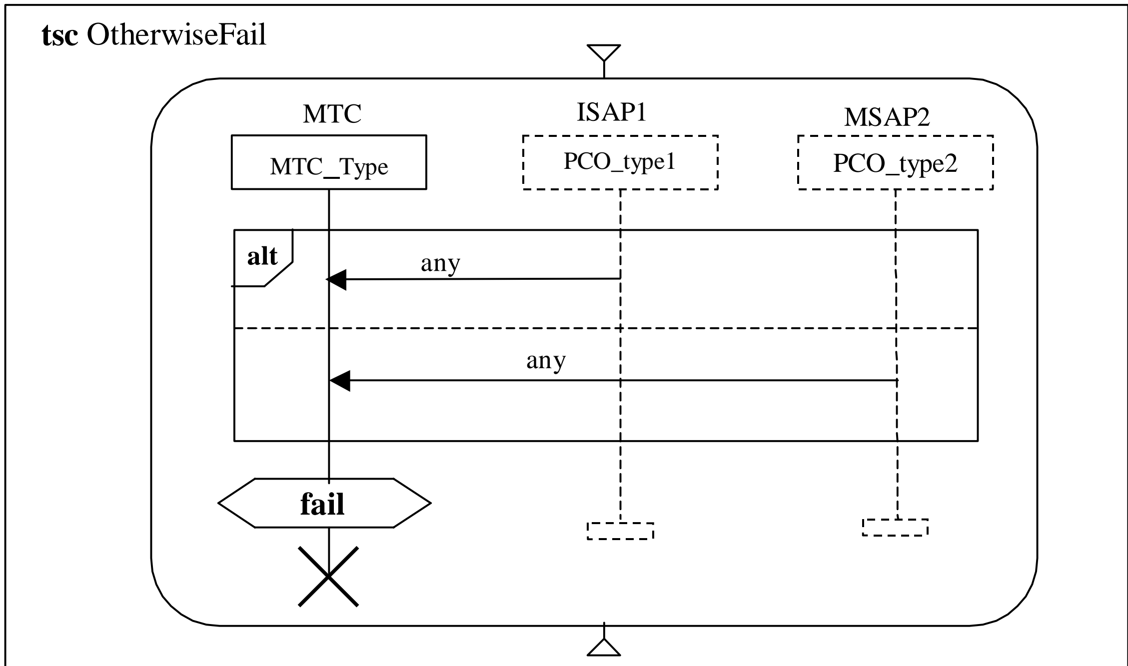


Figure E.3: INRES: sequential test case, default

E.2.2 Second version

The second version of the test case contains only minor modifications. This is mainly due to the fact, that the differences in the TTCN-3 core notation for the two test cases are not visible on TSC level, but rather it is transparent that the two reflect the same test behaviour.

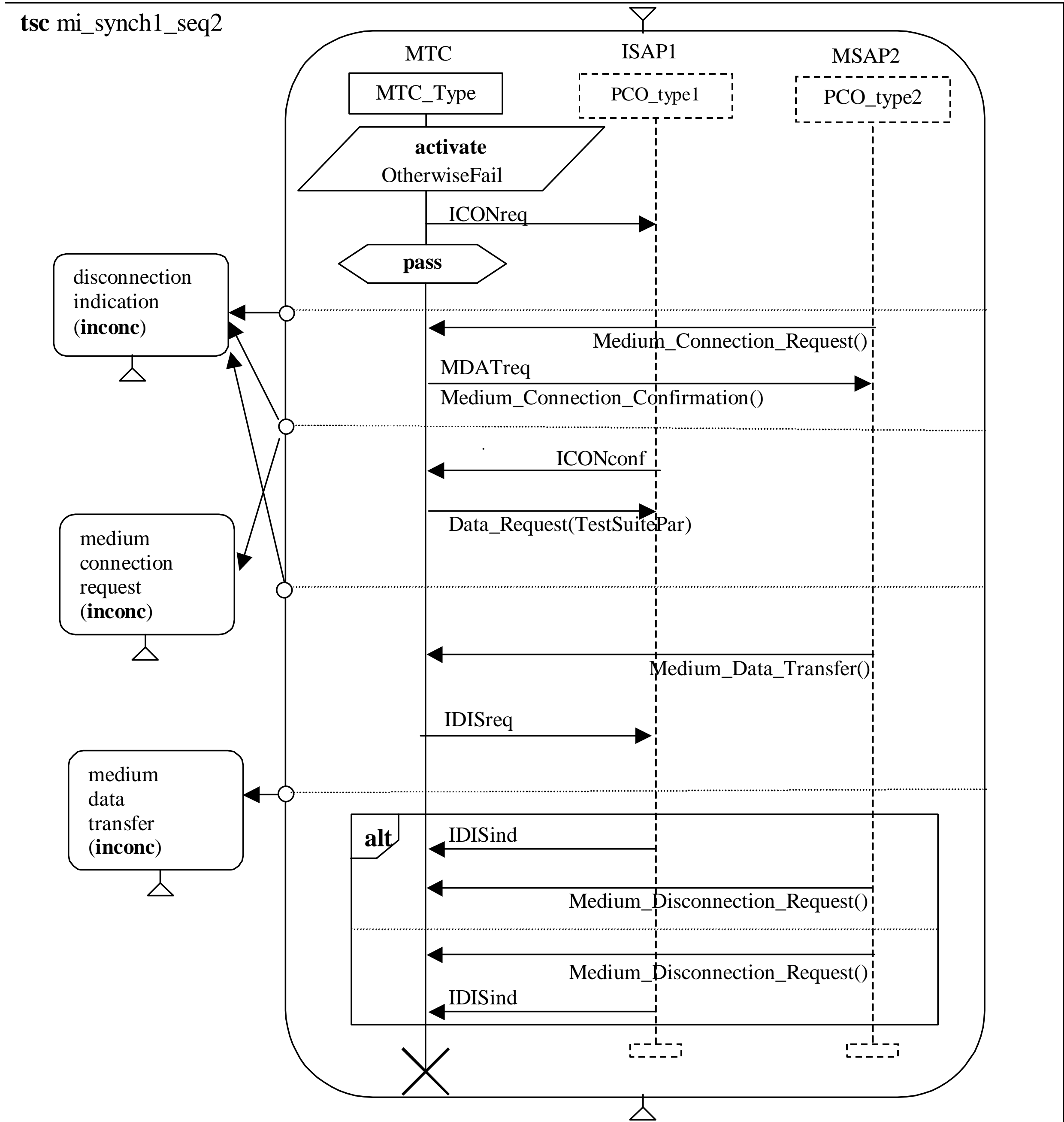


Figure E.4: INRES: sequential test case, second version

The above TSC can be represented even more condensed by using a co-region (which resembles the interleave operation of TTCN-3, but is more powerful) instead of the alternative expression at the bottom.

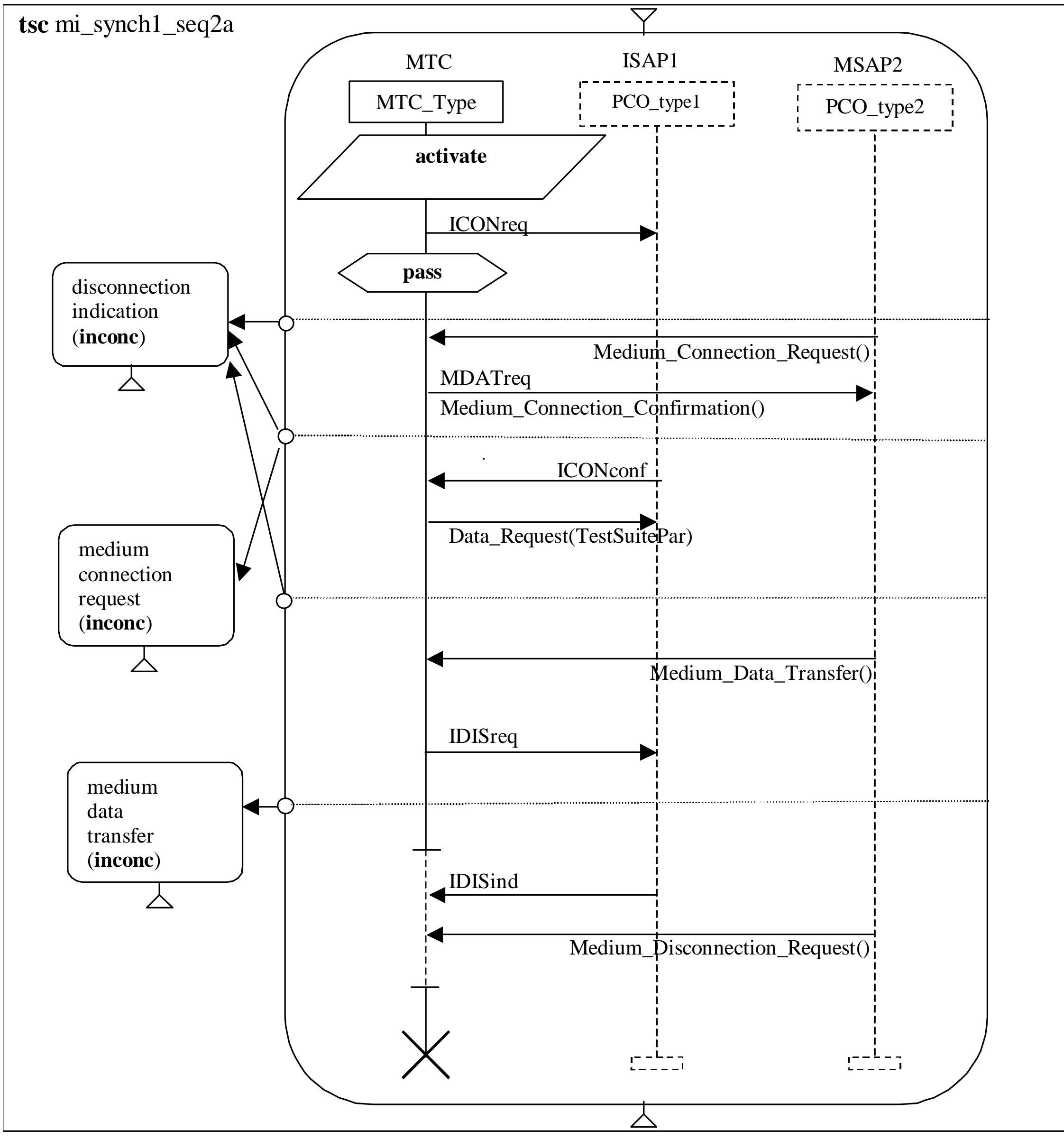


Figure E.5: INRES: sequential test case, second version (condensed)

E.2.3 Third version

Within the third case, named alternatives are used which are represented as TSC references in TSC.

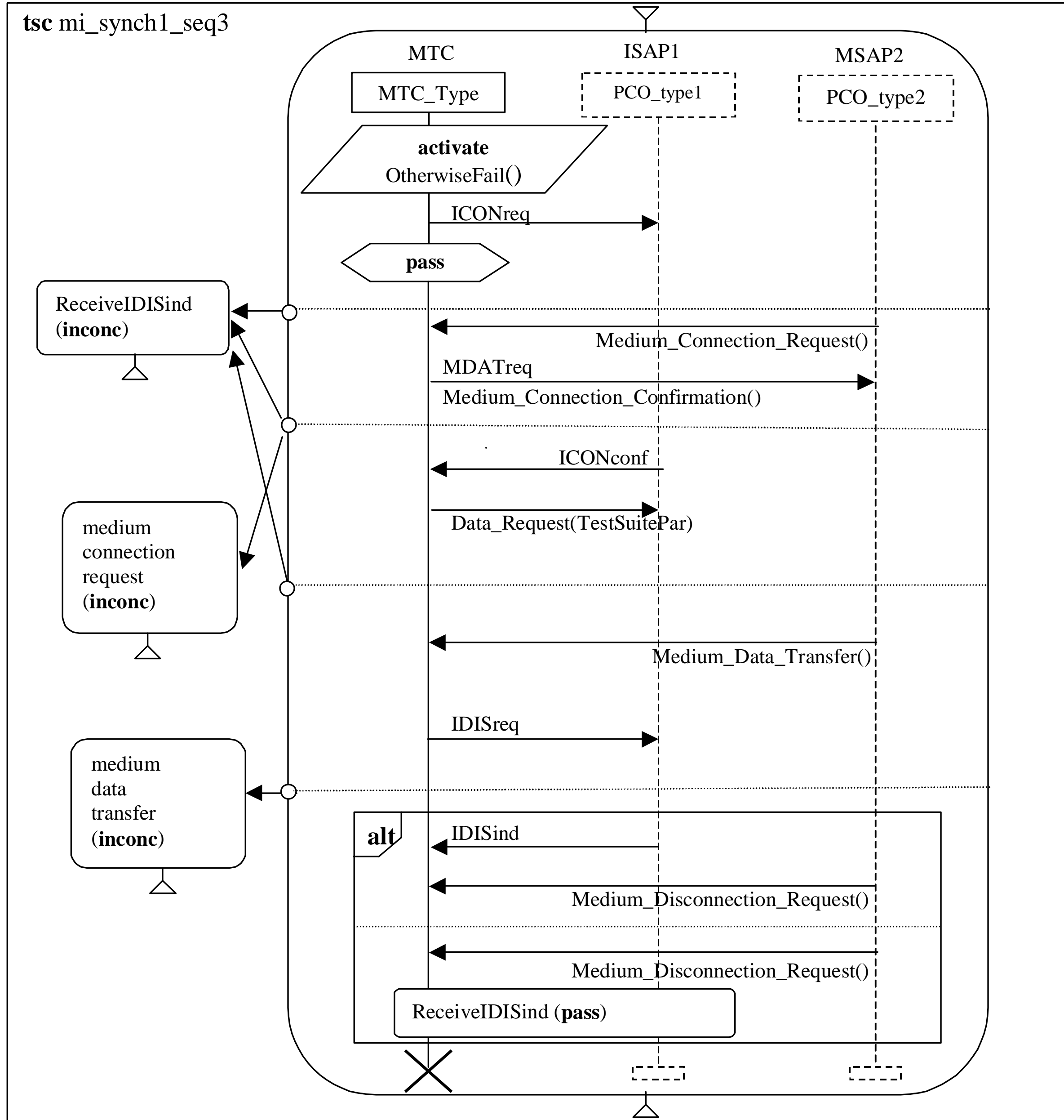


Figure E.6: INRES: sequential test case, third version

E.3 TSCs for concurrent test case

In figure E.7, the vertical split form is chosen for the representation of test components, i.e. for each test component there is one TSC. The TSC for PTC_ISAP1 is provided in figure E.12, the TSC for PTC_MSAP2 in figure E.14. Function "Synchronization" is presented using the horizontal split form. The corresponding TSC is provided in figure E.11.

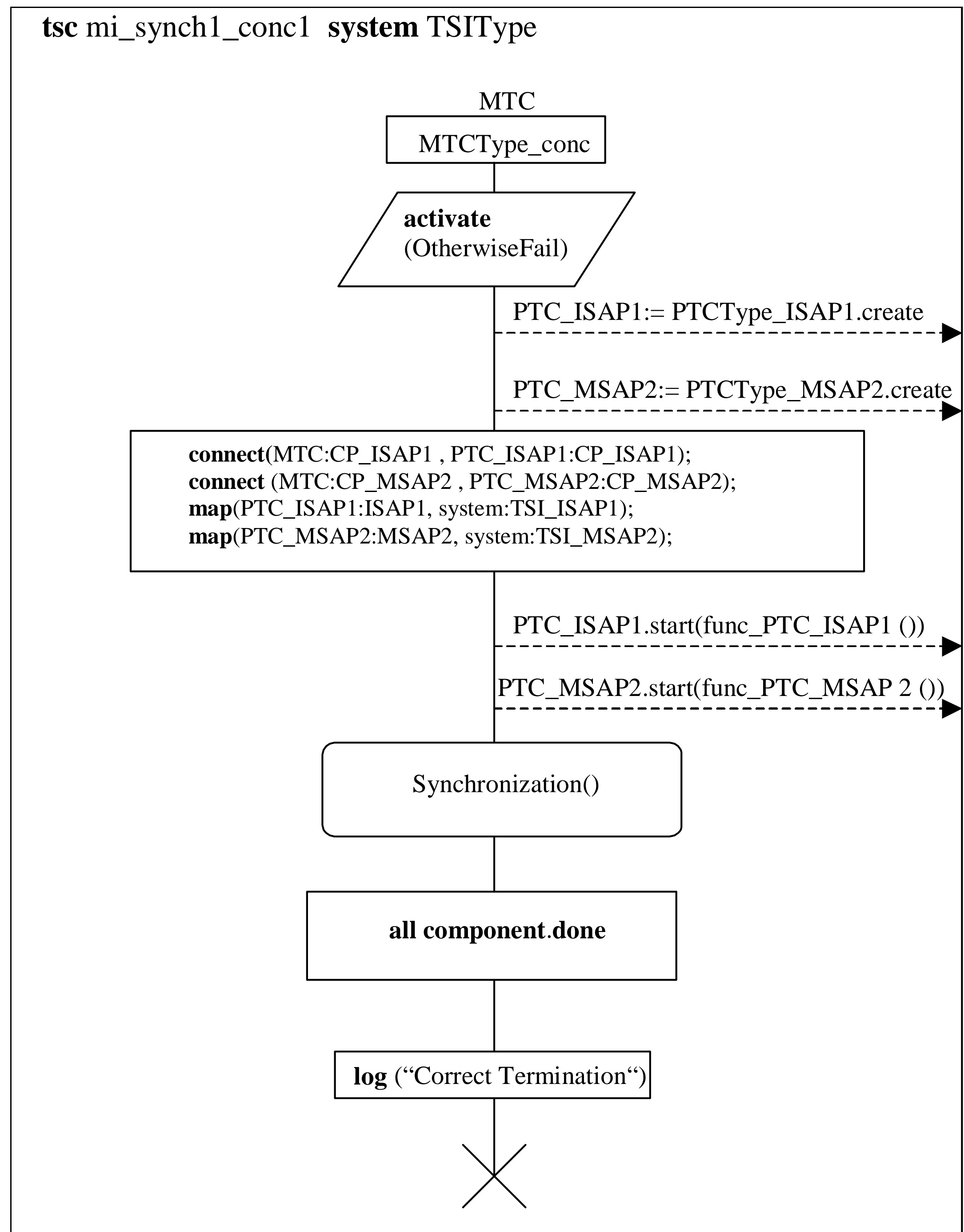


Figure E.7: INRES: concurrent test case (MTC, vertical split)

In figure E.8, the behaviour of the test components MTC, PTC_ISAP1 and PTC_MSAP2 is integrated within one TSC (no vertical split). For the behaviour definition , the following functions "Synchronization", "Func_PTC_ISAP1", and "Func_PTC_MSAP2" are used. Their definition is provided in figures E.11, E.13 and E.15, respectively.

In figure E.8, the message exchange between these functions is not shown explicitly, contrary to figure E.9. We take the view that the representation of the message exchange should be optional.

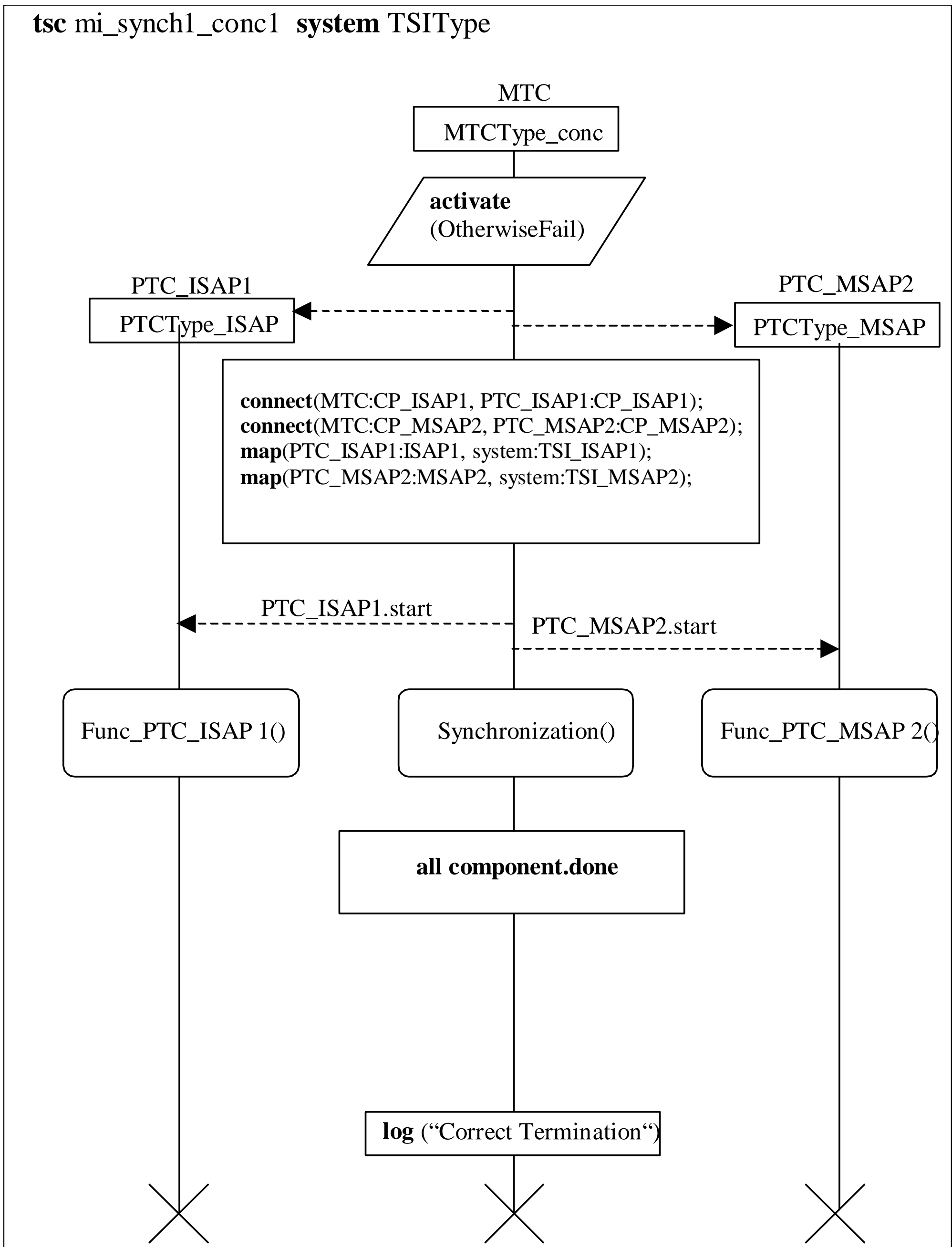


Figure E.8: INRES: concurrent test case (no vertical split)

In figure E.9, the same presentation is shown as in figure E.8, except that the message exchange between the functions is provided explicitly.

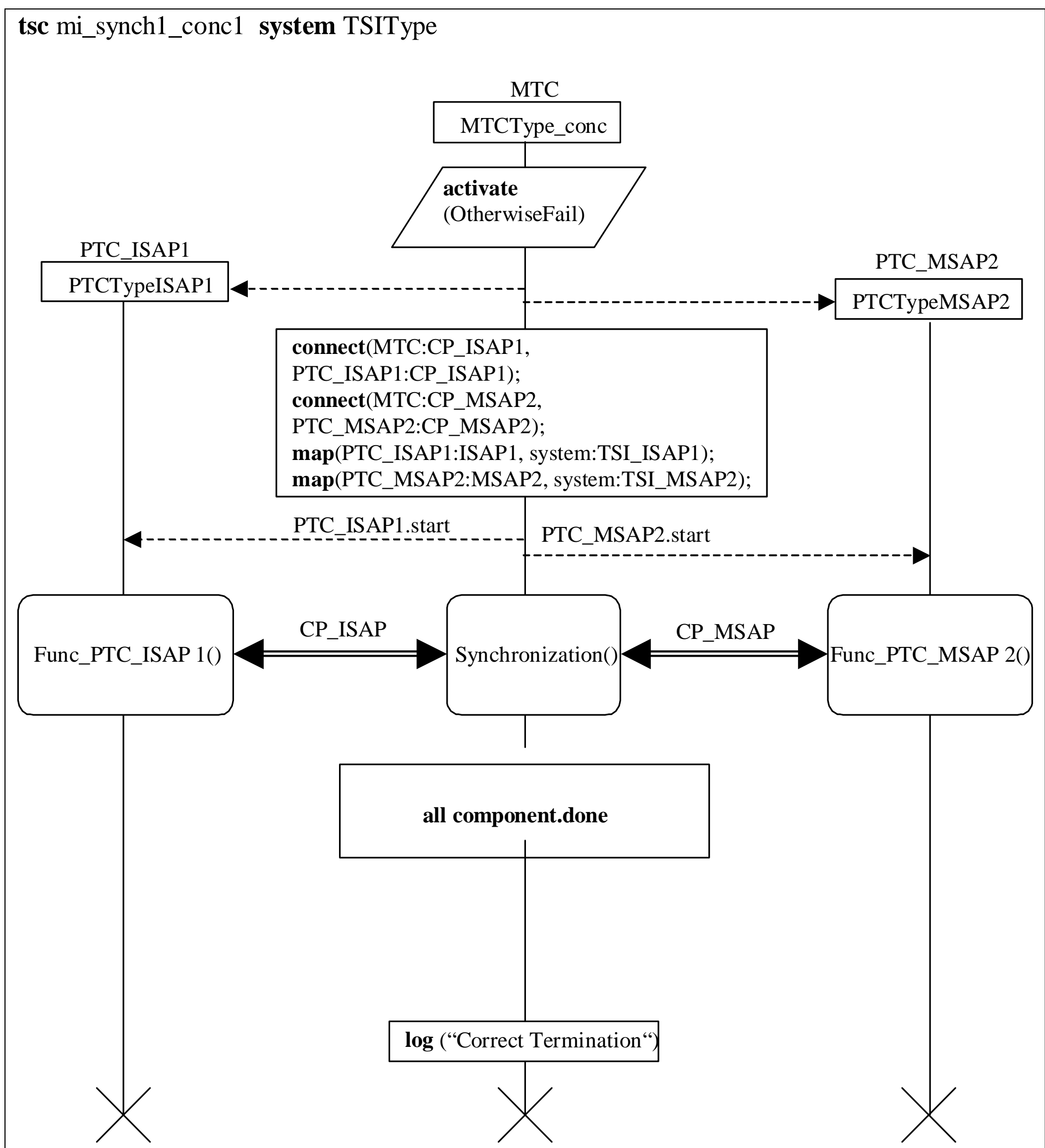


Figure E.9: INRES: concurrent test case (no vertical split, message exchange made explicit)

Figure E.10 is obtained from figure E.9 by expanding the MSC reference "Synchronization".

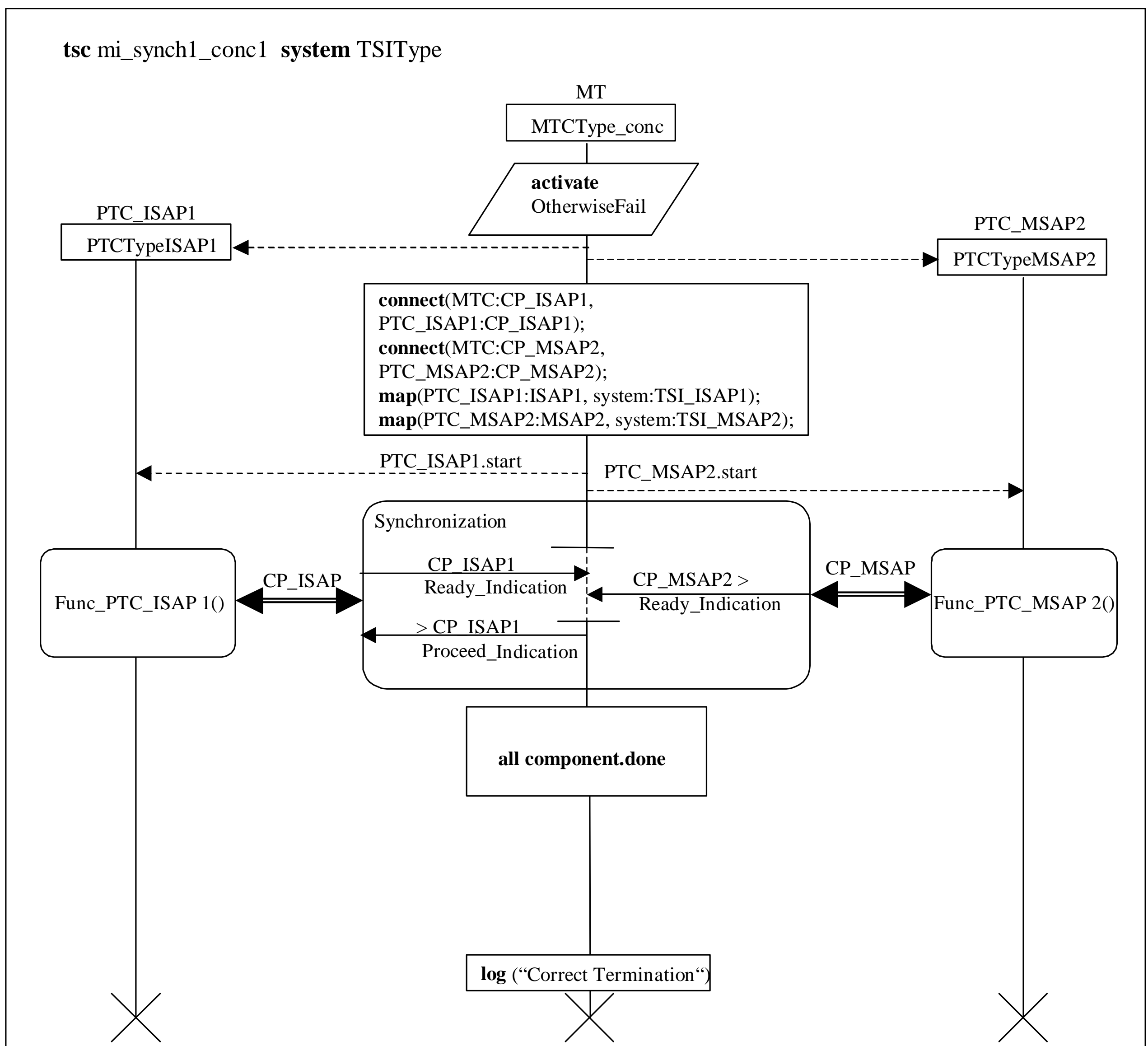


Figure E.10: INRES: concurrent test case (no vertical split with expanded function)

In figure E.11, the behaviour specification of function "Synchronization" is provided.

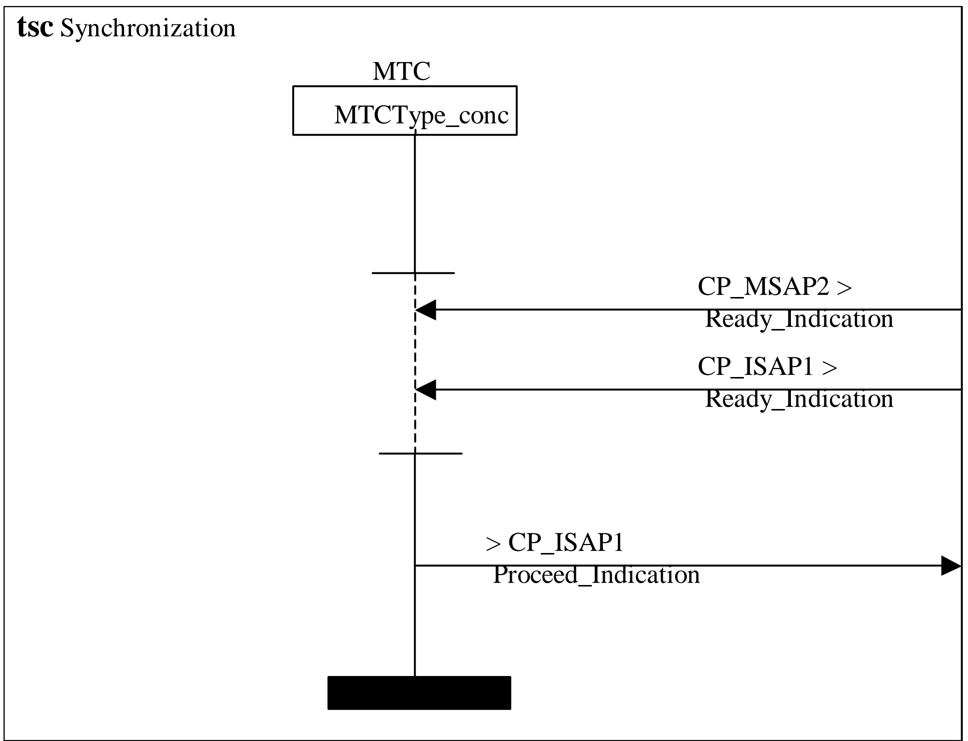


Figure E.11: INRES: concurrent test case (TSC Synchronization)

In figure E.12, the behaviour specification of PTC_ISAP1 is provided.

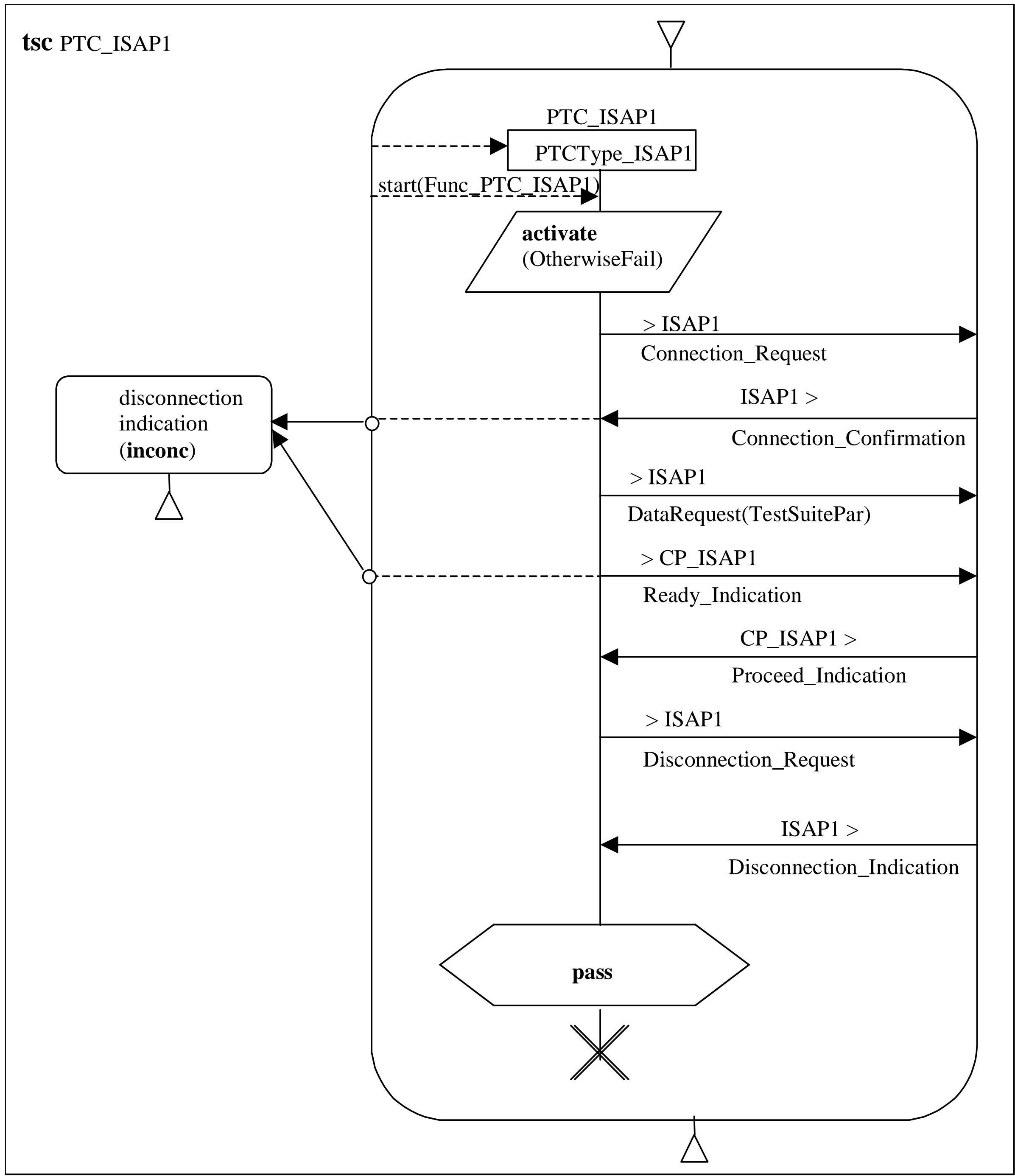


Figure E.12: INRES: concurrent test case (TSC PTC_ISAP1, vertical split)

In figure E.13, the behaviour specification of function "Func_PTC_ISAP1" is provided.

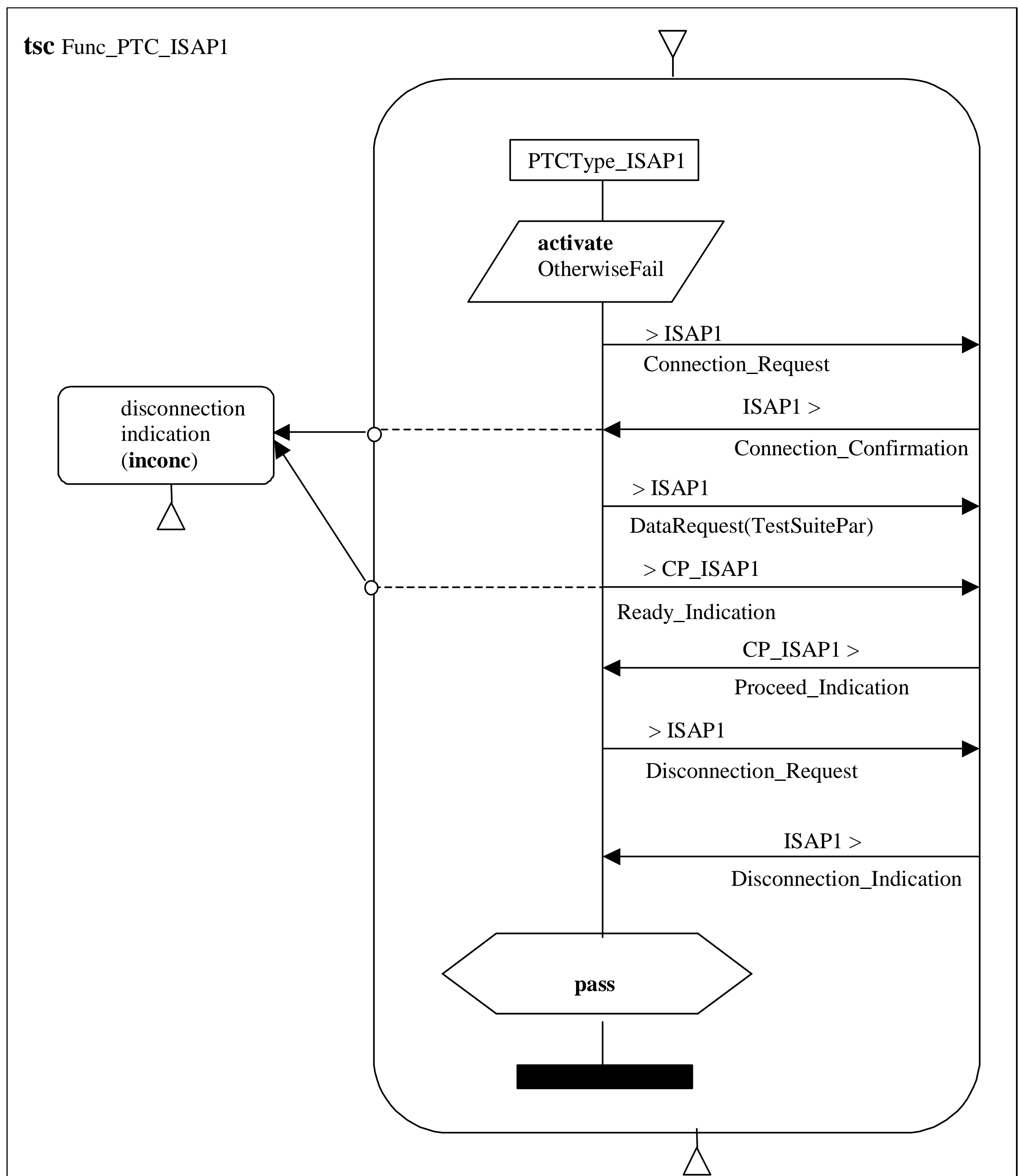


Figure E.13: INRES: concurrent test case (TSC Func_PTC_ISAP1)

In figure E.14, the behaviour specification of PTC_MSAP2 is provided.

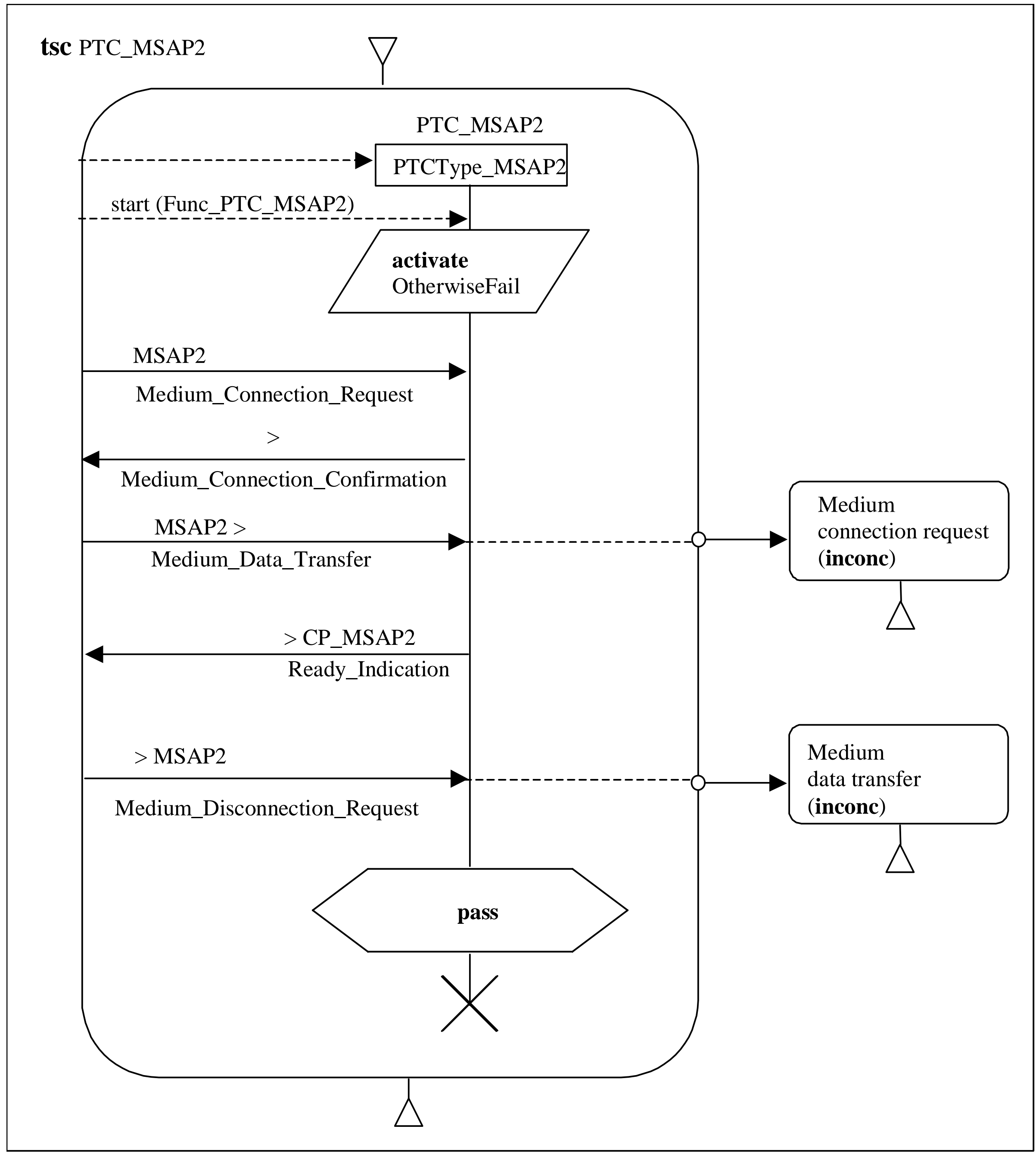


Figure E.14: INRES: concurrent test case (TSC PTC_MSAP2, vertical split)

In figure E.15, the behaviour specification of function "Func_PTC_MSAP2" is provided.

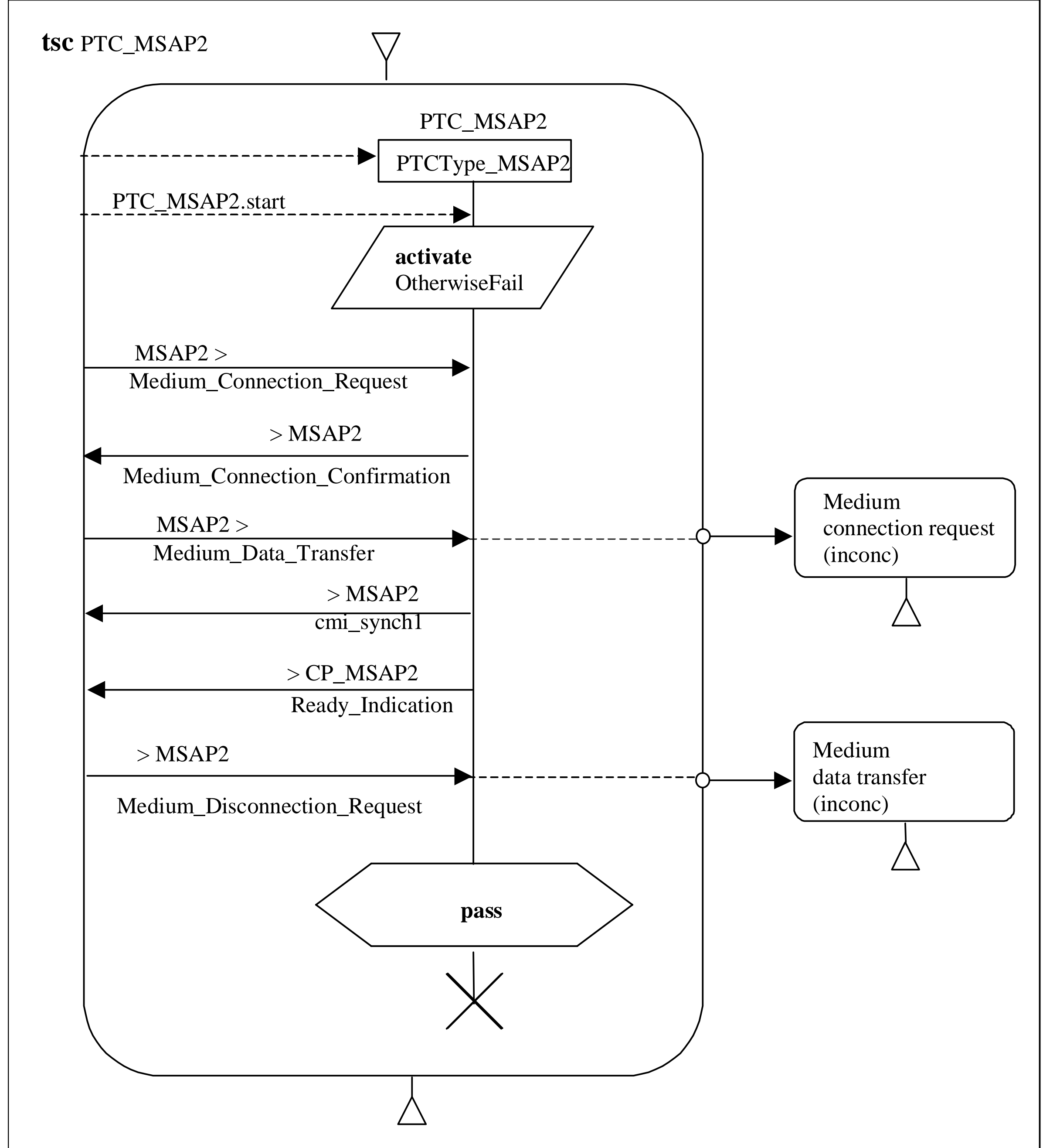


Figure E.15: INRES: concurrent test case (TSC Func_PTC_MSAP2)

In figure E.16, a merge of the TSCs in figures E.13, E.7 and E.15 is shown explicitly by means of an HMSC with parallel merge.

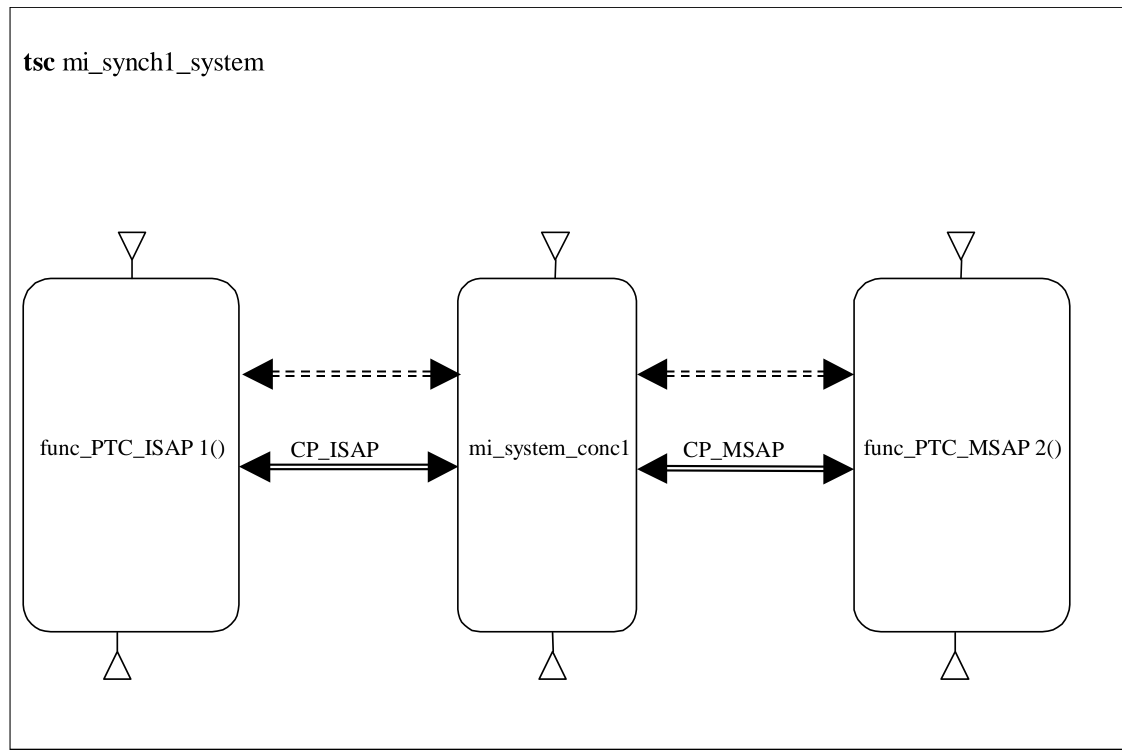


Figure E.16: INRES: concurrent test case (Test Case with TSC References)

E.4 INRES example in TTCN-3 core language

```
module InresExample ( ISDUType TestsuitePar ) {
//  General Type Definitions


group TypeAndConstantDefinitions {

    type integer ISDUType ;
    type enumerated Sequencenumber { zero(1), one(2) };
    type enumerated IPDUType { CR(1), CC(2), DR(3), DT(4), AK(5) };


    // Module Constants
    const Sequencenumber TestSuiteConst := 1;
}
//End TypeAndConstantDefinitions

// ASP Type Definitions for Medium Interface
group MediumServiceASPTypes {
    type record MDATreq {
        IPDUType iPDUType1 ,
        Sequencenumber sequencenumber2,
        ISDUType iSDUType3
    }
    type record MDATind {
        IPDUType iPDUType1,
        Sequencenumber sequencenumber2,
        ISDUType iSDUType3
    }

} with display "ASP Type Definitions";
//End MediumServiceASPTypes
```

```
// ASP Type Definitions for Inres Interface (Initiator side)
group InitiatorSideInresASPTypes {
    type record ICONreq {};
    type record IDATreq { ISDUType iSDUType1 };
    type record IDISreq {};
    type record ICONconf {};
    type record IDISind {}

} with display "ASP Type Definitions";
//End InitiatorSideInresASPTypes


// Template Definitions
group TemplateDefinitions {
    template IDATreq DataRequest (template ISDUType Par1):= {
        iSDUType1:= Par1
    };
    template MDATreq Medium_Connection_Confirmation :=  {
        iPDUType1:= CC,
        sequencenumber2:= *,
        iSDUType3:= *
    };
    template MDATind Medium_Connection_Request :=  {
        iPDUType1:= CR,
        sequencenumber2:= *,
        iSDUType3:= *
    };
    template MDATind Medium_Data_Transfer := {
        iPDUType1:= DT,
        sequencenumber2:= TestSuiteConst,
        iSDUType3:= TestSuitePar
    };
    template MDATind Medium_Disconnection_Request := {
        iPDUType1:= DR,
        sequencenumber2:= *,
        iSDUType3:= *
    }
    template MDATreq cmi_synch1 := {
        iPDUType1:= AK,
        sequencenumber2:= one,
        iSDUType3:= 55
    }
}
//End TemplateDefinitions

//named alternative used as default for the following test cases
named alt OtherwiseFail  {
        []  ISAP1.receive {
                verdict.set(fail);
                stop;
            }
    []  MSAP2.receive {
                verdict.set(fail);
                stop;
            }
} with display "default";

named alt ReceiveIDISind (verdict result) {
        []  ISAP1.receive( IDISind: {} )
            { verdict.set(result); }
} with display "test step"; // End ReceiveIDISind


// Configuration Definitions
group SequentialConfigurationDefinitions {
    // Port Type Definitions
    type port PCO_Type1 message {
        in ICONconf, IDISind;                   //received from SUT
        out ICONconf, IDATreq, IDISreq;     //send to SUT
    } with display "PCO Type, role := UT";
    type port PCO_Type2 message {
        in MDATind;           //received from SUT
        out MDATreq;              //send to SUT
```

```
    } with display "PCO Type, role = LT";
    // MTC Type Definition (is equal to Test System Interface)
    type component MTCType {
        PCO_Type1 ISAP1;
        PCO_Type2 MSAP2;
    }
} //End SequentialConfigurationDefinitions


group Sequential {
testcase mi_synch1_seq1() runs on MTCType
{
  activate (OtherwiseFail);     //Default activation
  ISAP1.send( ICONreq: {} );    //In-line template definition
  alt {
      [] MSAP2.receive( Medium_Connection_Request )      //use of a template
        {
            MSAP2.send( Medium_Connection_Confirmation );
            alt {
                 [] ISAP1.receive ( ICONconf: {} )
                    {
                      ISAP1.send ( Data_Request(TestSuitePar) );
                      alt {
                            [] MSAP2.receive ( Medium_Data_Transfer )
                                {
                                  ISAP1.send ( IDISreq: {} );
                                    alt {
                                    [] ISAP1.receive (IDISind: {} )
                                        { //PASS
                                            MSAP2.receive ( Medium_Disconnection_Request )
                                        }
                                    [] MSAP2.receive ( Medium_Disconnection_Request )
                                        { ISAP1.receive( IDISind: {} )
                                            //PASS
                                        }
                                    }
                                }
                            [] ISAP1.receive( IDISind: {} )
                                {
                                  verdict.set(inconc);
                                  stop
                                }
                          }
                    }
                [] MSAP2.receive( Medium_Connection_Request)
                    {
                      verdict.set(inconc);
                      stop;
                    }
                [] ISAP1.receive( IDISind: {} )
                    {
                      verdict.set(inconc);
                      stop
                    }
                }
        }
        [] ISAP1.receive( IDISind: {} )
        {
            verdict.set(inconc);
            stop;
        }
    }
    verdict.set(pass);
}
// End testcase mi_synch1_seq1
```

/* This is a slight transformation of the previous test case. It makes use of the 'sequential' character of TTCN-3. The stop operation is not copied to all branches of the 'behaviour tree'. Instead it is only executed at the end of the test case. Furthermore, the verdict is initialized with the verdict pass. During the test execution it can be overwritten by inconclusive or, in the default, by fail. The places where a pass has been assigned before are indicated by comments */

```
testcase mi_synch1_seq2 () runs on MTCType
{
    activate (OtherwiseFail);
    //Default activation
    ISAP1.send( ICONreq: {} );
    verdict.set(pass);
    alt {
    [] MSAP2.receive( Medium_Connection_Request )
       {
        MSAP2.send ( Medium_Connection_Confirmation );
        alt {
            []ISAP1.receive ( ICONconf: {} )
                {
                ISAP1.send ( Data_Request(TestSuitePar) );
                alt {
                    [] MSAP2.receive ( Medium_Data_Transfer )
                    {
                        ISAP1.send ( IDISreq: {} );
                        alt {
                            [] ISAP1.receive (IDISind: {} )
                                { //PASS
                                MSAP2.receive ( Medium_Disconnection_Request )
                                }
                            [] MSAP2.receive ( Medium_Disconnection_Request )
                                { ISAP1.receive( IDISind: {} )
                                //PASS
                        }
                    [] MSAP2.receive ( Medium_Data_Transfer )
                        { verdict.set(inconc) }
                        }
                    }
                    [] ISAP1.receive( IDISind: {} )
                        { verdict.set(inconc) }
                    }
                }
            []MSAP2.receive( Medium_Connection_Request )
                { verdict.set(inconc) }
            []ISAP1.receive( IDISind: {} )
                { verdict.set(inconc) }
            }
        }
    [] ISAP1.receive( IDISind: {} )
        { verdict.set(inconc) }
    }
    stop
}
//End testcase mi_synch1_seq2
```

/* This is a slight transformation of the previous test case. It shows the usage of named alternatives. A named alternative is a special "macro" construct which allows to extend the alternative construct by adding new alternatives. It is comparable to the TTCN-2 feature of tree attachment in sets of alternatives */

```
testcase mi_synch1_seq3 () runs on MTCType
{
    activate (OtherwiseFail);
    //Default activation
    ISAP1.send( ICONreq: {} );
    verdict.set(pass);
    alt {
        []MSAP2.receive( Medium_Connection_Request )
          {
            MSAP2.send ( Medium_Connection_Confirmation );
            alt {
                []ISAP1.receive ( ICONconf: {} )
                    {
                    ISAP1.send ( Data_Request(TestSuitePar) )
                    alt {
                        []MSAP2.receive ( Medium_Data_Transfer )
                          {
                            ISAP1.send ( IDISreq: {} );
                            alt {
                                []ISAP1.receive (IDISind: {} ) // PASS
                                    { MSAP2.receive ( Medium_Disconnection_Request ); }
                                []MSAP2.receive ( Medium_Disconnection_Request )
                                    { ReceiveIDISind(pass); }    // direct call of named alt
                                []MSAP2.receive ( Medium_Data_Transfer )
```

```
                                        { verdict.set(inconc); }
                                };
                              }
                            [expand]    ReceiveIDISind(inconc); // use of named alternative
                            };
                        }
                    []MSAP2.receive( Medium_Connection_Request )
                        { verdict.set(inconc); }
                    [expand]    ReceiveIDISind(inconc);
                    };
                }
            [expand]    ReceiveIDISind(inconc);
    };
    stop;
} // End testcase mi_synch1_seq3

}
//End group Sequential

group ConcurrentConfigurationDefinitions {

    // Port Type Definitions
    type port CP_Type message {
        inout CM;
    } with display "CP Type";

    // MTC Type Definition (is now different to Test System Interface)
    type component TSIType_conc {
        PCO_Type1 TSI_ISAP1;
        PCO_Type2 TSI_MSAP2;
    }

    type component MTCType_conc {
        CP_Type CP_ISAP1;
        CP_Type CP_MSAP2;
        timer MyTimer:= 5;
    }

    //PTCs
    type component PTCType_ISAP1 {
        PCO_Type1 ISAP1;
        CP_Type CP_ISAP1;
    }

    type component PTCType_MSAP2 {
        PCO_Type2 MSAP2;
        CP_Type CP_MSAP2;
    }
} //End ConcurrentConfigurationDefinitions

// CM Type Definitions
group CoordinationMessages {

    type record CM { charstring message1 };
} with display "CM Type Definitions";
// End CM Type Definitions

// CM Template Definitions
group CoordinationTemplateDefinitions {
    template CM Proceed_Indication := {
        message1:= "PROCEED"
    };

    template CM Ready_Indication := {
        message1:= "READY"
    };
} // End CM Template Definitions

// Now, we consider a concurrent version of the INRES Example
group Concurrent {

// Synchronization function
function Synchronization () runs on MTCType_conc
{
```

```
    interleave {
    [] CP_MSAP2.receive( Ready_Indication );
    [] CP_ISAP1.receive( Ready_Indication );
    }
    CP_ISAP1.send( Proceed_Indication );
} // End Synchronization function

function Func_PTC_ISAP1 () runs on PTCType_ISAP1
{
    activate (OtherwiseFail);  // Default activation
    ISAP1.send( Connection_Request );
    alt {
        [] ISAP1.receive( Connection_Confirmation )
            {
                ISAP1.send( Data_Request(TestSuitePar) );
                CP_ISAP1.send( Ready_Indication );
                alt {
                [] CP_ISAP1.receive( Proceed_Indication )
                    {
                        ISAP1.send( Disconnection_Request );
                        ISAP1.receive( Disconnection_Indication );
                        verdict.set( pass )
                    }
                [] ISAP1.receive( Disconnection_Indication)
                    {
                        verdict.set(inconc);
                    }
                }
            }
        [] ISAP1.receive( Disconnection_Indication)
            {
                verdict.set(inconc);
            }
    }
} //End Func_PTC_ISAP1

function Func_PTC_MSAP2 () runs on PTCType_MSAP2
{
    activate (OtherwiseFail);  // Default activation
    MSAP2.receive( Medium_Connection_Request );
    MSAP2.send( Medium_Connection_Confirmation );
    alt {
        [] MSAP2.receive( Medium_Data_Transfer )
           {
            MSAP2.send( cmi_synch1 );
            CP_MSAP2.send( Ready_Indication );
            alt {
                [] MSAP2.receive( Medium_Disconnection_Req )
                  {
                    verdict.set(pass);
                  }
                [] MSAP2.receive( Medium_Data_Transfer )
                  {
                    verdict.set(inconc);
                  }
              }
            }
        [] MSAP2.receive( Medium_Connection_Request )
            {
                verdict.set(inconc);
            }
    }
} //End Func_PTC_MSAP2

// Concurrent Testcase
testcase mi_synch1_conc1 () runs on MTCType_conc system TSIType
{
    activate (OtherwiseFail);  // Default activation
    verdict.set(pass);
    var PTCType_ISAP1 PTC_ISAP1:= PTCType_ISAP1.create;
    var PTCType_MSAP2 PTC_MSAP2:= PTCType_MSAP2.create;

    connect(PTC_ISAP1:CP_ISAP1,mtc:CP_ISAP1);
    connect(PTC_MSAP2:CP_MSAP2,mtc:CP_MSAP2);
    map(PTC_ISAP1:ISAP1, system:TSI_ISAP1);
    map(PTC_MSAP2:MSAP2, system:TSI_MSAP2);
```

```
    PTC_ISAP1.start(func_PTC_ISAP1());
    PTC_MSAP2.start(func_PTC_MSAP2());

    Synchronization();
    all component.done;
    log("Correct Termination");
} // End Concurrent Testcase
}
//End group Concurrent
// end  module InresExample
}
```

History

Document history		
V1.1.1	January 2001	Publication