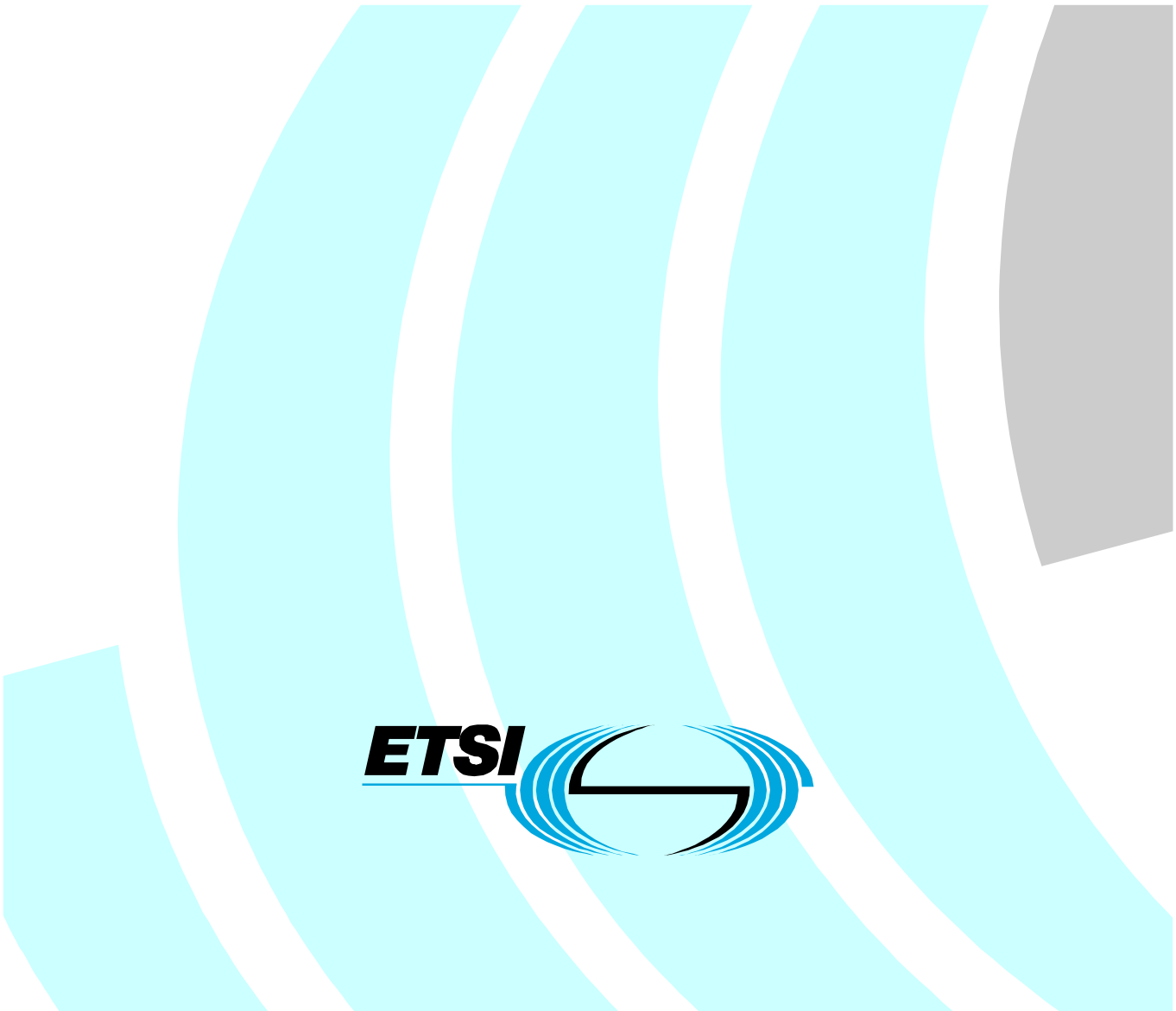
ETSI TR 102 043 V1.1.1 (2002-04)

Technical Report

Methods for Testing and Specification (MTS); The TTCN-3 Runtime Interface (TRI); Concepts and definition of the TRI



Reference
DTR/MTS-00074

Keywords
interface, 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://portal.etsi.org/tb/status/status.asp

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

Copyright Notification

Contents

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://webapp.etsi.org/IPR/home.asp).

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 present document is organized into two distinct parts, the first part describes the structure of a TTCN-3 test system implementation and the second part presents the proposed TTCN-3 Runtime Interface specification.

The first part will introduce the decomposition of a TTCN-3 test system into four main entities: Test Management (TM), a TTCN-3 Executable (TE), a SUT Adapter (SA), and a Platform Adapter (PA). In addition, the interaction between these entities, i.e. the corresponding interfaces, will be discussed.

The second part of the present document then specifies the TTCN-3 Runtime Interface (TRI). The interface is defined in terms of operations which are implemented as part of one entity and called by other entities of the test system. For each operation, the interface specification will define associated data structures, the intended effect on the test system, and any constraints on when the operation may be used. Notice that this interface specification only defines interactions between main test system entities.

1 Scope

The present document provides the specification of the runtime interface for TTCN-3 test system implementations. The TTCN-3 Runtime Interface provides a standardized adaptation for timing and communication of a test system to a particular processing platform and the system under test, respectively. The purpose of the present document is to define this interface as a set of operations independent of a target language.

This interface is defined to be compatible with the TTCN version 3 Core Language standard. Instead of advocating a particular implementation target language the present document uses the CORBA Interface Definition Language (IDL) to specify the TRI completely. Clauses 8 and 9 then present language mappings for this abstract specification to the target languages Java and C. A summary of the IDL based interface specification is provided in the annex.

The TRI can be considered to fulfil a similar purpose for TTCN-3 as the Generic Compiler Interface (GCI) did for TTCN version 2.

2 Compliance

The minimum required for a TRI compliant TTCN-3 test system is to adhere to the interface specification stated in the present document and one language mapping. For example, if a vendor supports Java, the TRI operation calls and implementations, which are part of the TTCN-3 executable, must comply with the IDL to Java mapping specified in the present document.

3 References

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

[1] ISO/IEC 9646-3 (2nd edition): "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular combined Notation (TTCN)".

[2] 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".

4 Definitions and abbreviations

4.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

explicit timer: timer that is declared in a TTCN-3 ATS and that can be manipulated by a user using TTCN-3 timer operations

implicit timer: system timer that is created by the TTCN-3 Executable to guard a TTCN-3 call or execute operation. Implicit timers are not accessible to the TTCN-3 user.

Platform Adapter (PA): entity that adapts the TTCN-3 Executable to a particular execution platform. The Platform Adapter creates a single notion of time for a TTCN-3 test system, and implements both, explicit and implicit, timers as well as external functions.

SUT Adapter (SA): entity that adapts the TTCN-3 communication operations with the SUT based on an abstract test system interface. It implements the real test system interface.

test event: either sent or received test data (message or procedure call) on a communication port that is part of the test system interface

Timer IDentification (TID): unique identification for explicit or implicit timer instances that is generated by the TTCN-3 Executable

TTCN-3 Executable (TE): part of a test system that deals with interpretation or execution of a TTCN-3 ETS

TTCN-3 Control Interface (TCI): currently a proprietary interface that specifies the interaction between Test Management and TTCN-3 Executable in a test system

Test Management (TM): entity that provides a user interface and administers the TTCN-3 test system

TTCN-3 Runtime Interface (TRI): interface that defines the interaction of the TTCN-3 Executable with the SUT and Platform Adapter in a test system

4.2 Definitions from ES 201 873-1

The following definitions are taken from ES 201 873-1 [2]:

communication port

Implementation eXtra Information for Testing (IXIT)

System Under Test (SUT)

test case

test system

4.3 Abbreviations

ATS	Abstract Test Suite
ETS	Executable Test Suite
IDL	Interface Definition Language
IXIT	Implementation Extra Information for Testing
MSC	Message Sequence Chart
MTC	Main Test Component
OMG	Object Management Group
PA	Platform Adapter
SA	SUT Adapter
SUT	System Under Test
TC	Test Control
TCI	TTCN-3 Control Interface
TE	TTCN-3 Executable
TEC	TTCN-3 Executable Control
TEQ	TTCN-3 Executable Queue
TID	Timer Identification
TL	Test Logging
TM	Test Management
TRI	TTCN-3 Runtime Interface
TSI	Test System Interface
TTCN-3	Testing and Test Control Notation 3

5 Void

6 General Structure of a TTCN-3 test system

A TTCN-3 test system can be thought of conceptually as a set of interacting entities where each entity corresponds to a particular aspect of functionality in a test system implementation. These entities manage test execution, interpret or execute compiled TTCN-3 code, realize proper communication with the SUT, implement external functions, and handle timing.

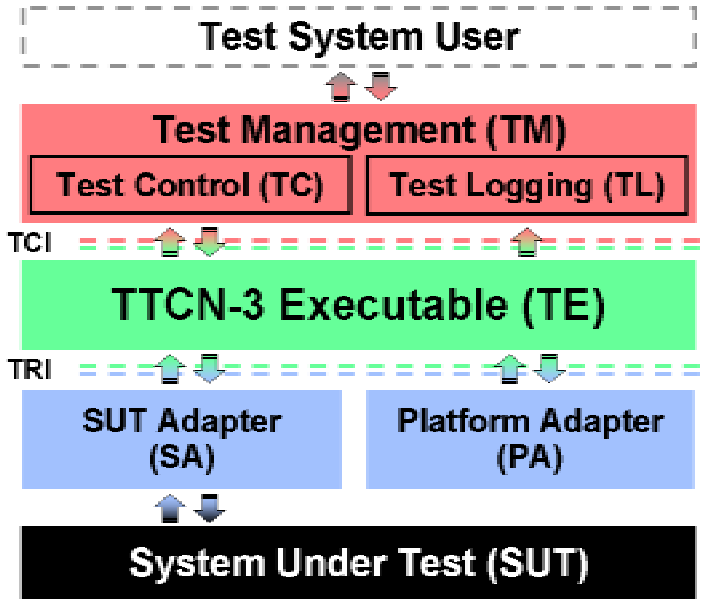


Figure 1: General Structure of a TTCN-3 Test System

6.1 Entities in a TTCN-3 test system

The structure of a TTCN-3 test system implementation is illustrated in figure 1. It should be noted that the further refinement of TM into smaller entities, as shown in this figure and used in the following clauses of the present document, is purely an aid to define TTCN-3 test system interfaces.

The part of the test system that deals with interpretation and execution of TTCN-3 modules, i.e. the Executable Test Suite (ETS), is shown as the TTCN-3 Executable (TE). This would typically correspond in a test system implementation either to the executable code produced by a TTCN-3 compiler or a TTCN-3 interpreter. It is here assumed that prior to a test system implementation the Abstract Test Suite (ATS), i.e. the original TTCN-3 source code, has been compiled into an executable format - the ETS.

The remaining part of the TTCN-3 test system, which deals with any aspects that cannot be concluded from information being present in the original ATS alone, can be decomposed into Test Management (TM), SUT Adapter (SA), and Platform Adapter (PA) entities. In general, these entities cover a test system user interface, test execution control, test event logging, as well as communication with the SUT and timer implementation.

6.1.1 Test Management (TM)

In the TM entity, we can distinguish between functionality related to test execution control and test event logging.

6.1.1.1 Test Control (TC)

The TC entity is responsible for overall management of the test system. After the test system has been initialized, test execution starts within the TC entity. The entity is responsible for the proper invocation of TTCN-3 modules, i.e. propagating module parameters and/or IXIT information to the TE if necessary. Typically, this entity would also implement a test system user interface.

6.1.1.2 Test Logging (TL)

The TL entity is responsible for maintaining the test log. It is explicitly notified to log test events by the TE. The TL entity has a unidirectional interface where any entity part of the TE may post a logging request to the TL entity. A TM internal interface may also be used to record test management information generated by the TC.

6.1.2 TTCN-3 Executable (TE)

The TE entity is responsible for the interpretation or execution of the TTCN-3 ATS. Conceptually, the TE can be decomposed into three interacting entities: a Control, Behaviour, and Queue entity. Such a decomposition of the TE functionality results in the structure shown in figure 2. The following clauses outline the responsibilities of each entity and also discuss the handling of timers at the TRI.

Notice again that the further refinement of the TE into smaller entities is purely a conceptual aid to define TTCN-3 test system interfaces - there is no requirement for this division to be reflected in implementations of the TRI.

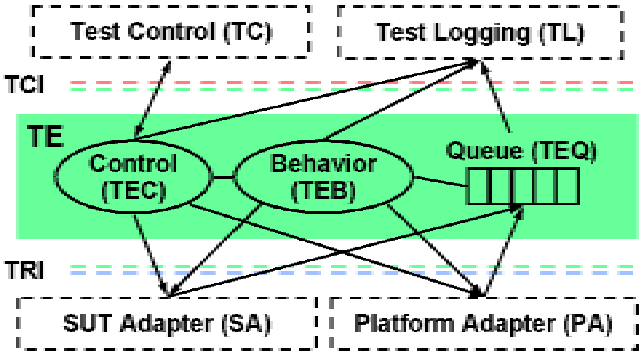


Figure 2: Interaction of TTCN-3 Executable with other entities of a TTCN-3 test system

6.1.2.1 TTCN-3 Executable Control (TEC)

The TEC entity handles the control part specified for TTCN-3 modules. This entity performs all the actions necessary to properly sequence the execution of test cases or functions as well as the collection and resolution of associated verdicts as defined in [2]. The TEC entity is also responsible for resetting adapters prior to the execution of a test case, i.e. to initialize adapters, unmap all test system interface ports and stop all timers from the previous test case execution. It queries the TM entity for module parameter values and sends logging information to it. It instructs the SA which SUT action operation is to be executed, and similarly the PA which external functions are to be executed or timers are to be started, stopped, queried, or read for operations being specified in the control part of a TTCN-3 module. A TE internal interfaces should handle, e.g. the starting of test cases in the TEB entity, or the retrieval of timeout events from the TEQ entity.

6.1.2.2 TTCN-3 Executable Behaviour (TEB)

The TEB entity handles the execution or interpretation of test cases and functions as defined in the corresponding TTCN-3 modules [2]. It is therefore responsible for controlling the sequencing and matching of test events, the logging of test events during test case execution, as well as the creation and removal of TTCN-3 test components. It also handles the TTCN-3 semantics of message and procedure based communication with the SUT, external function calls, SUT action operations, as well as timers, i.e. it instructs the SUT Adapter (SA) which message or procedure call is to be sent to the SUT or the Platform Adapter (PA) which external function is to be executed or which timers are to be started, stopped, queried, or read. Similarly, the TEB receives from TEQ entity incoming messages or procedure calls from the SUT as well as timeout events.

The TEB entity is responsible for the encoding and decoding of test data in communication operations with the SUT as specified in the executing TTCN-3 module. If no encoding has been specified for the module the test data's representation is tool specific. The TEB entity should implement all message and procedure based communication operations between test components, but only the TTCN-3 semantics of procedure based communication with the SUT, i.e. the possible blocking and unblocking of test component execution, guarding with implicit timers, and handling of timeout exceptions as a result of such communication operations. All procedure based communication operations with the SUT are to be realized and identified (in the case of a receiving operation) in the SA as they are most efficiently implemented in a platform specific manner. Note that timing of any procedure call operation, i.e. implicit timers, is implemented in the Platform Adapter (PA).

6.1.2.3 TTCN-3 Executable Queue (TEQ)

The TTCN-3 Executable is required to maintain its own port queues (distinct from those which may be available in the SA or PA) for input test events to perform snapshots for receiving operations as defined in [2]. Timeout events, which are generated by TTCN-3 timer, call timer, or test case timer implementations, are to be kept in a timeout list as specified in [1]. In figure 2, all of this functionality has been assigned to the TEQ entity. The purpose of the TEQ entity is to store events that the TE entity has been notified of by the SA or PA but which have not yet been processed. The TEQ entity has a unidirectional interface with the adapter entities. Only the adapter entities are allowed to initiate communication with the TEQ entity. Proper communication of test and timeout events to the TEB and TEC entities are achieved using TE internal interfaces. The TEQ entity may also send logging information to the TL entity.

6.1.2.4 Timers in the TTCN-3 Executable

Timers in the TE can be conceptually classified as explicit, that are TTCN-3 timers created and accessible within the TTCN-3 ATS, and implicit, that are timers created by the TE to guard TTCN-3 call or execute operations. Explicit as well as implicit timers are both created within the TE but implemented by the Platform Adapter (PA). This is achieved by generating a unique timer identification (TID) for any timer created in the TE. This unique TID should enable the TE to differentiate between different timers. The TID is to be used by the TE to interact with corresponding timer implementation in the PA.

Notice that it is the responsibility of the TE to implement the different TTCN-3 semantics for explicit and implicit timers correctly as defined in [2], e.g. the use of keywords `any` and `all` with timers only applies to explicit timers. In the PA all timers, i.e. implicit and explicit, are treated in the same manner.

6.1.3 SUT Adapter (SA)

The SA adapts message and procedure based communication of the TTCN-3 test system with the SUT to the particular execution platform of the test system. It is aware of the mapping of the TTCN-3 test component communication ports to test system interface ports and implements the real test system interface as defined in [2]. It is responsible to propagate send requests and SUT action operations from the TTCN-3 executable (TE) to the SUT, and to notify the TE of any received test events by enqueueing them in the port queues of the TE.

Procedure based communication operations with the SUT are implemented in the SA. The SA is responsible to distinguish between the different messages within procedure based communication (i.e. call, reply, and exception) and to propagate them in the appropriate manner either to the SUT or the TE. TTCN-3 procedure based communication semantics, i.e. the effect of such operation on TTCN-3 test component execution, are to be handled in the TE.

The SA has an interface with the TE which is used to send SUT messages (issued in TTCN-3 SUT action operations) to the SA and to exchange encoded test data between the two entities in communication operations with the SUT.

6.1.4 Platform Adapter (PA)

The PA implements TTCN-3 external functions and provides a TTCN-3 test system with a single notion of time. In this entity external functions are to be implemented as well as all timers. Notice that timer instances are created in the TE. A timer in the PA can only be distinguished by its timer identification (TID). Therefore, the PA treats both, explicit and implicit, timers in the same manner.

The interface with the TE enables the invocation of external functions and the starting, reading, and stopping of timers as well as the inquiring of the status of timers using their timer ID. The PA notifies the TE of expired timers.

6.2 Interfaces in a TTCN-3 Test System

As previously depicted in figure 1, a TTCN-3 test system has two interfaces, the TTCN-3 Control Interface (TCI) and the TTCN-3 Runtime Interface (TRI), which specify the interface between Test Management (TM) and TTCN-3 Executable (TE) entities, and TE, SUT Adapter (SA) and Platform Adapter (PA) entities, respectively.

The present document defines the TRI. The interaction of the TE with SA and PA will be defined here in terms of TRI operations. Although both interfaces, i.e. the TRI and the TCI, have to be defined for a complete implementation of an executable test suite (ETS) the specification and implementation of the TCI is currently considered to be proprietary.

6.3 Execution Requirements for a TTCN-3 Test System

Each TRI operation call shall be treated as an atomic operation in the calling entity. The called entity, which implements a TRI operation, shall return control to the calling entity as soon as its intended effect has been accomplished or if the operation cannot be completed successfully. The called entity shall not block in the implementation of procedure based communication. Nevertheless, the called entity shall block after the invocation of an external function implementation and wait for its return value. Notice that depending on the test system implementation failure to return from an external function implementation may result in the infinite blocking of test component execution, the TTCN-3 executable, the Platform Adapter, or even of the entire test system.

The execution requirements stated above can be realized in a tightly integrated test system implementation. Here, the entire TTCN-3 test system is implemented in a single executable or process where each test system entity is assigned at least one thread of execution. TRI operations can be implemented here as procedure calls.

Notice that a looser integration of a test system implementation is still possible, e.g. an implementation of a TTCN-3 test system with multiple SUT Adapters in a distributed computing environment. In this case only a small part of the SUT Adapter is tightly integrated with the remainder of the TTCN-3 test system whereas actual SA Adapters may be realized in separate processes. That small part of SA may then only implement a routing of information provided by TRI operations to the desired SUT Adapter processes, possibly being executed on remote hosts, and vice versa.

7 TTCN-3 Runtime Interface and Operations

This clause defines TRI operations in terms of when they are to be used and what their effect is intended to be in a TTCN-3 test system implementation. Also a set of abstract data types is defined which is then used for the definition of TRI operations. This definition also includes a more detailed description of the input parameters required for each TRI operation call and its return value.

7.1 Overview of the TRI

The TRI defines the interaction between the TTCN-3 Executable (TE), SUT Adapter (SA), and Platform Adapter (PA) entities within a TTCN-3 test system implementation. Conceptually, it provides a means for the TE to send test data to the SUT or manipulate timers, and similarly to notify the TE of received test data and timeouts.

The TRI can be considered to consist of two sub-interfaces, a triCommunication and a triPlatform interface. The triCommunication interface addresses the communication of a TTCN-3 ETS with the SUT which is implemented in the SA. The triPlatform interface represents a set of operations which adapt an ETS to a particular execution platform.

Table 1: Interface Overview

Interface	Direction (calling entity → called entity)	
Name	TE → SA or PA	SA or PA → TE
triCommunication	TE → SA	SA → TE
triPlatform	TE → PA	PA → TE

Both interfaces are bidirectional so that calling and called parts reside in the TE, SA, and PA entities of the test system.

Table 1 shows in more detail the caller/callee relationship between the respective entities. Notice that this table only shows interactions visible at the TRI. Internal communication between parts of the same entity are not reflected as the internal structure of the TE, SA, or PA may differ in a TTCN-3 test system implementation.

7.1.1 The triCommunication Interface

This interface consists of operations that are necessary to implement the communication of the TTCN-3 ETS with the SUT. It includes operations to initialize the Test System Interface (TSI), establish connections to the SUT, and handle message and procedure based communication with the SUT. In addition, the triCommunication interface offers an operation to reset the SUT Adapter (SA).

7.1.2 The triPlatform Interface

This interface includes all operations necessary to adapt the TTCN-3 Executable to a particular execution platform. The triPlatform interface offers means to start, stop, read a timer, enquire its status and to add timeout events to the expired timer list. In addition, it offers operations to call TTCN-3 external functions and to reset the Platform Adapter (PA). Notice that there is no differentiation between explicit and implicit timers required at the triPlatform Interface. Instead each timer shall be addressed uniformly with its Timer IDentifier (TID).

7.1.3 Correlation between TTCN-3 and TRI Operation Invocations

For some TTCN-3 operation invocations there exists a direct correlation to one TRI operation invocation (or possibly two in the case of TTCN-3 `execute` and `call` operations) which is shown in table 2. For all other TRI operation invocations there may be no direct correlation.

The shown correlation for TTCN-3 communication operations (i.e. `send`, `call`, `reply`, and `raise`) only holds if these operations are invoked on a test component port which is mapped to a TSI port. Nevertheless, this correlation holds for all such operation invocations if no system component has been specified for a test case, i.e. only the MTC test component is created for a test case and no other test components.

Table 2: Correlation between TTCN-3 and TRI Operation Invocations (*= if applicable)

TTCN-3 Operation Name	TRI Operation Name	TRI Interface Name
execute	triExecuteTestCase triStartTimer*	TriCommunication TriPlatform
map	triMap	TriCommunication
unmap	triUnmap	TriCommunication
send	triSend	TriCommunication
call	TriCall triStartTimer*	TriCommunication TriPlatform
reply	triReply	TriCommunication
raise	triRaise	TriCommunication
(sut.)action	triSUTactionInformal* triSUTactionTemplate*	TriCommunication TriCommunication
start (timer)	triStartTimer	TriPlatform
stop (timer)	triStopTimer	TriPlatform
read (timer)	triReadTimer	TriPlatform
running (timer)	triTimerRunning	TriPlatform
TTCN-3 external function	triExternalFunction	TriPlatform

Notice that all of the TRI operations listed in table 2 are called by the TE and that the TE may implement the invocation of these operations differently when evaluating a TTCN snapshot within the TTCN-3 ETS.

7.2 Error handling

Explicit error handling is specified only for TRI operations called by the TTCN-3 Executable (TE). The SA or PA reports the status of a TRI operation in the return value of a TRI operation. The status value can either indicate the local success (***TRI_OK***) or failure (***TRI_Error***) of the TRI operation. Therefore, the TE may react to an error which occurred either within the SA or PA and issue, e.g. a test case error.

For TRI operations called by the SA or PA no explicit error handling is required since these operations are implemented in the TE. Here, the TE is in control over the test execution in the case that an error occurs in such a TRI operation.

Notice that specific error codes as well as the detection and handling of errors in any of the test system entities are beyond the scope of the current TRI specification.

7.3 Data Interface

In the TRI operations only encoded test data shall be passed. The TTCN-3 Executable (TE) is responsible for encoding test data to be sent and decoding received test data in the respective TRI operations since encoding rules can be specified for or within a TTCN-3 module [2]. Notice that the TE is required to encode test data even if no encoding information has been provided in a TTCN-3 ATS. In this case the tool vendor has to define an encoding.

Instead of defining an explicit data interface for TTCN-3 and ASN.1 data types, the TRI standard defines a set of abstract data types. These data types are used in the following definition of TRI operations to indicate which information is to be passed from the calling to the called entity, and vice versa. The concrete representation of these abstract data types as well as the definition of basic data types are defined in the respective language mappings in clauses 8 and 9.

Notice that the values for any identifier data type shall be unique in the test system implementation where uniqueness is defined as being globally distinct at any point in time.

The following abstract data types are defined and used for the definition of TRI operations.

7.3.1 Connection

`TriComponentIdType` A value of type `TriComponentIdType` includes an identifier and the component type. The distinct value of the latter is the component type name as specified in the TTCN-3 ATS. This abstract type is mainly used to resolve TRI communication operations on TSI ports which have mappings to many test component ports.

`TriPortIdType` A value of type `TriPortIdType` includes a value of type `TriComponentIdType` to represent the component to which the port belongs, a port index (if present), and the port name as specified in the TTCN-3 ATS. The `TriPortIdType` type is mainly required to pass information about the TSI and connections to the TSI from the TE to the SA.

`TriPortIdListType` A value of type `TriPortIdListType` is a list of `TriPortIdType`. This abstract type is used for initialization purposes after the invocation of a TTCN-3 test case.

7.3.2 Communication

`TriMessageType` A value of type `TriMessageType` is encoded test data which either is to be send to the SUT or has been received from the SUT.

`TriAddressType` A value of type `TriAddressType` indicates a source or destination address within the SUT. This abstract type can be used in TRI communication operations and is an open type which is opaque to the TE [2].

`TriSignatureIdType` A value of type `TriSignatureIdType` is the name of a procedure signature as specified in the TTCN-3 ATS. This abstract type is used in procedure based TRI communication operations.

`TriParameterType` A value of type `TriParameterType` includes an encoded parameter and a value of `TriParameterPassingModeType` to represent the passing mode specified for the parameter in the TTCN-3 ATS.

`TriParameterPassingModeType` A value of type `TriParameterPassingModeType` is either *in*, *inout*, or *out*. This abstract type is used in procedure based TRI communication operations and for external function calls.

`TriParameterListType` A value of type `TriParameterListType` is a list of `TriParameterType`. This abstract type is used in procedure based TRI communication operations and for external function calls.

`TriExceptionType` A value of type `TriExceptionType` is an encoded type and value of an exception which either is to be send to the SUT or has been received from the SUT. This abstract type is used in procedure based TRI communication operations.

7.3.3 Timer

`TriTimerIdType` A value of type `TriTimerIdType` specifies an identifier for a timer. This abstract type is required for all TRI timer operations.

`TriTimerDurationType` A value of type `TriTimerDurationType` specifies the duration for a timer in seconds.

7.3.4 Miscellaneous

`TriTestCaseIdType` A value of type `TriTestCaseIdType` is the name of a test case as specified in the TTCN-3 ATS.

`TriFunctionIdType` A value of type `TriFunctionIdType` is the name of an external function as specified in the TTCN-3 ATS.

`TriActionTemplateType` A value of type `TriActionTemplateType` is the encoded type and value of a template used in a TTCN-3 SUT action operation.

`TriStatusType` A value of type `TriStatusType` is either ***TRI_OK*** or ***TRI_Error*** indicating the success or failure of a TRI operation.

7.4 Operation Descriptions

All operation definitions are defined using the Interface Definition Language (IDL). Concrete language mappings are defined in clauses 8 and 9.

For every TRI operation call all *in*, *inout*, and *out* parameters listed in the particular operation definition are mandatory. The value of an *in* parameter is specified by the calling entity. Similarly, the value of an *out* parameter is specified by the called entity. In the case of an *inout* parameter, a value is first specified by the calling entity but may be replaced with a new value by the called entity. Note that although TTCN-3 also uses *in*, *inout*, and *out* for signature definitions the denotations used in TRI IDL specification are not related to those in a TTCN-3 specification.

Operation calls should use a reserved value to indicate the absence of parameters which are defined as optional in the corresponding TRI parameter description. The reserved values for these types are defined in each language mapping and will be subsequently referred to as the `null` value.

All functions in the interface are described using the following template:

F.n.m	***Operation Name***	calling entity → called entity

Signature `IDL-Signature`

In Parameters Description of data passed as parameters to the operation from the calling entity to the called entity.

Out Parameters Description of data passed as parameters to the operation from the called entity to the calling entity.

InOutParameters Description of data passed as parameters to the operation from the calling entity to the called entity and from the called entity back to the calling entity.

Return Value Description of data returned from the operation to the calling entity.

Constraints Description of any constraints when the operation may be called.

Effect Behaviour required of the called entity before the operation may return.

7.5 Communication Interface Operations

- ***triSAReset*** **TE → SA**

Signature `TriStatusType triSAReset()`

In Parameters n.a.

Out Parameters n.a.

Return Value The return status of the triSAReset operation. The return status indicates the local success (*TRI_OK*) or failure (*TRI_Error*) of the operation.

Constraints This operation can be called by the TE at any time to reset the SA.

Effect The SA shall reset all communication means which it is maintaining, e.g. reset static connections to the SUT, close dynamic connections to the SUT, discard any pending messages or procedure calls.
The triResetSA operation returns ***TRI_OK*** in case the operation has been successfully performed, ***TRI_Error*** otherwise.

7.5.1 Connection Handling Operations

- ***triExecuteTestcase*** **TE → SA**

Signature `TriStatusType triExecuteTestcase(in TriTestCaseIdType testCaseId, in TriPortIdListType tsiPortList)`

In Parameters `testCaseId` identifier of the test case that is going to be executed.
`tsiPortList` a list of test system interface ports defined for the test system.

Out Parameters n.a.

Return Value The return status of the `triExecuteTestcase` operation. The return status indicates the local success (*TRI_OK*) or failure (*TRI_Error*) of the operation.

Constraints This operation is called by the TE immediately before the execution of any test case. The test case that is going to be executed is indicated by the `testCaseId`. `tsiPortList` contains all ports that have been declared in the definition of the system component for the test case, i.e. the TSI ports. If a system component has not been explicitly defined for the test case in the TTCN-3 ATS then the `tsiPortList` contains all communication ports of the MTC test component. The ports in `tsiPortList` are ordered as they appear in the respective TTCN-3 component declaration.

Effect The SA can set up any static connections to the SUT and initialize any communication means for TSI ports.
The `triExecuteTestcase` operation returns ***TRI_OK*** in case the operation has been successfully performed, ***TRI_Error*** otherwise.

- ***triMap*** **TE → SA**

Signature `TriStatusType triMap(in TriPortIdType compPortId, in TriPortIdType tsiPortId)`

In Parameters `compPortId` identifier of the test component port to be mapped.
`tsiPortId` identifier of the test system interface port to be mapped.

Out Parameters n.a.

Return Value The return status of the `triMap` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when it executes a TTCN-3 map operation.

Effect The SA can establish a dynamic connection to the SUT for the referenced TSI port.
The `triMap` operation returns ***TRI_Error*** in case a connection could not be established successfully, ***TRI_OK*** otherwise. The operation should return ***TRI_OK*** in case no dynamic connection needs to be established by the test system.

- ***triUnmap*** **TE → SA**

Signature
```
TriStatusType triUnmap(in TriPortIdType compPortId, in
TriPortIdType tsiPortId)
```

In Parameters
`compPortId` identifier of the test component port to be unmapped.
`tsiPortId` identifier of the test system interface port to be unmapped.

Out Parameters n.a.

Return Value The return status of the `triUnmap` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when it executes any TTCN-3 unmap operation.

Effect The SA shall close a dynamic connection to the SUT for the referenced TSI port.
The `triUnmap` operation returns ***TRI_Error*** in case a connection could not be closed successfully or no such connection has been established previously, ***TRI_OK*** otherwise. The operation should return ***TRI_OK*** in case no dynamic connections have to be established by the test system.

7.5.2 Message Based Communication Operations

- ***triSend*** **TE → SA**

Signature
```
TriStatusType triSend(in TriComponentIdType componentId,
                      in TriPortIdType      tsiPortId,
                      in TriAddressType     SUTaddress,
                      in TriMessageType     sendMessage)
```

In Parameters
`componentId` identifier of the sending test component.
`tsiPortId` identifier of the test system interface port via which the message is sent to the SUT Adapter.
`SUTaddress` (optional) destination address within the SUT.
`sendMessage` the encoded message to be send.

Out Parameters n.a.

Return Value The return status of the `triSend` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when it executes a TTCN-3 send operation on a component port, which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 send operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case.
The encoding of `sendMessage` has to be done in the TE prior to this TRI operation call.

Effect The SA can send the message to the SUT.
The `triSend` operation returns ***TRI_OK*** in case it has been completed successfully. Otherwise ***TRI_Error*** shall be returned. Notice that the return value ***TRI_OK*** does not imply that the SUT has received `sendMessage`.

- ***triEnqueueMsg*** **SA → TE**

Signature
```
void triEnqueueMsg (in TriPortIdType      tsiPortId,
                    in TriAddressType     SUTaddress,
                    in TriComponentIdType componentId,
                    in TriMessageType     receivedMessage);
```

In Parameters
`tsiPortId` identifier of the test system interface port via which the message is enqueued by the SUT Adapter.
`SUTaddress` (optional) source address within the SUT.
`componentId` identifier of the receiving test component.
`receivedMessage` the encoded received message.

Out Parameters n.a.

Return Value void

Constraints This operation is called by the SA after it has received a message from the SUT. It can only be used when `tsiPortId` has been either previously mapped to a port of `componentId` or has been referenced in the previous `triExecuteTestCase` statement.
In the invocation of a `triEnqueueMsg` operation `receivedMessage` shall contain an encoded value.

Effect This operation shall pass the message to the TE indicating the port of the component `componentId` to which the TSI port `tsiPortId` is mapped.
The decoding of `receivedMessage` has to be done in the TE.

7.5.3 Procedure Based Communication Operations

- ***triCall*** **TE → SA**

Signature
```
TriStatusType triCall(in TriComponentIdType   componentId,
                      in TriPortIdType        tsiPortId,
                      in TriAddressType       SUTaddress,
                      in TriSignatureIdType   signatureId,
                      in TriParameterListType parameterList)
```

In Parameters
`componentId` identifier of the test component issuing the procedure call.
`tsiPortId` identifier of the test system interface port via which the procedure call is sent to the SUT Adapter.
`SUTaddress` (optional) destination address within the SUT.
`signatureId` identifier of the signature of the procedure call.
`parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration.

Out Parameters n.a.

Return Value The return status of the `triCall` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when it executes a TTCN-3 call operation on a component port, which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 call operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case.
All *in* and *inout* procedure parameters contain encoded values. All *out* procedure parameters shall contain the distinct value of `null` since they are only of relevance in a reply to the procedure call but not in the procedure call itself.
The procedure parameters are the parameters specified in the TTCN-3 signature template. Their encoding has to be done in the TE prior to this TRI operation call.

Effect On invocation of this operation the SA can initiate the procedure call corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`.
The `triCall` operation shall return without waiting for the return of the issued procedure call (see note). This TRI operation returns ***TRI_OK*** on successful initiation of the procedure call, ***TRI_Error*** otherwise. No error shall be indicated by the SA in case the value of any *out* parameter is non-null. Notice that the return value of this TRI operation does not make any statement about the success or failure of the procedure call.
Note that an optional timeout value, which can be specified in the TTCN-3 ATS for a call operation, is *not* included in the `triCall` operation signature. The TE is responsible to address this issue by starting a timer for the TTCN-3 call operation in the PA with a separate TRI operation call, i.e. `triStartTimer`.

NOTE: This might be achieved for example by spawning a new thread or process. This handling of this procedure call is, however, dependent on implementation of the TE.

- ***triReply*** **TE → SA**

Signature
```
TriStatusType triReply(in TriComponentIdType   componentId,
                       in TriPortIdType        tsiPortId,
                       in TriAddressType       SUTaddress,
                       in TriSignatureIdType   signatureId,
                       in TriParameterListType parameterList,
                       in TriParameterType     returnValue)
```

In Parameters
`componentId` identifier of the replying test component.
`tsiPortId` identifier of the test system interface port via which the reply is sent to the SUT Adapter.
`SUTaddress` (optional) destination address within the SUT.
`signatureId` identifier of the signature of the procedure call.
`parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration.
`returnValue` (optional) encoded return value of the procedure call.

Out Parameters n.a.

Return Value The return status of the `triReply` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when it executes a TTCN-3 reply operation on a component port which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 reply operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case.
All *out* and *inout* procedure parameters and the return value contain encoded values. All *in* procedure parameters shall contain the distinct value of `null` since they are only of relevance to the procedure call but not in the reply to the call.
The `parameterList` contains procedure call parameters. These parameters are the parameters specified in the TTCN-3 signature template. Their encoding has to be done in the TE prior to this TRI operation call.
If no return type has been defined for the procedure signature in the TTCN-3 ATS, the distinct value `null` shall be passed for the return value.

Effect On invocation of this operation the SA can issue the reply to a procedure call corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`.
The `triReply` operation will return ***TRI_OK*** on successful execution of this operation, ***TRI_Error*** otherwise. No error shall be indicated by the SA in case the value of any *in* parameter or a non-defined return value is non-null.

- ***triRaise*** **TE → SA**

Signature
```
TriStatusType triRaise(in TriComponentIdType componentId,
                       in TriPortIdType      tsiPortId,
                       in TriAddressType     SUTaddress,
                       TriSignatureIdType signatureId,
                       in TriExceptionType   exception)
```

In Parameters
`componentId` identifier of the test component raising the exception.
`tsiPortId` identifier of the test system interface port via which the exception is sent to the SUT Adapter.
`SUTaddress` (optional) destination address within the SUT.
`signatureId` identifier of the signature of the procedure call which the exception is associated with.
`exception` the encoded exception.

Out Parameters n.a.

Return Value The return status of the `triRaise` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when it executes a TTCN-3 raise operation on a component port which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 raise operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case.
The encoding of the exception has to be done in the TE prior to this TRI operation call.

Effect On invocation of this operation the SA can raise an exception to a procedure call corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`.
The `triRaise` operation returns ***TRI_OK*** on successful execution of the operation, ***TRI_Error*** otherwise.

- ***triEnqueueCall*** **SA → TE**

Signature
```
void triEnqueueCall(in TriPortIdType        tsiPortId,
                    in TriAddressType       SUTaddress,
                    in TriComponentIdType   componentId,
                    in TriSignatureIdType   signatureId,
                    in TriParameterListType parameterList)
```

In Parameters
`tsiPortId` identifier of the test system interface port via which the procedure call is enqueued by the SUT Adapter.
`SUTaddress` (optional) source address within the SUT.
`componentId` identifier of the receiving test component.
`signatureId` identifier of the signature of the procedure call.
`parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration.

Out Parameters n.a.

Return Value void

Constraints This operation can be called by the SA after it has received a procedure call from the SUT. It can only be used when `tsiPortId` has been either previously mapped to a port of `componentId` or referenced in the previous `triExecute` statement.
In the invocation of a `triEnqueueCall` operation all *in* and *inout* procedure parameters contain encoded values. All *out* procedure parameters shall contain the distinct value of `null` since they are only relevant in the reply on the procedure call but not in the procedure call itself.

Effect The TE can enqueue this procedure call with the signature identifier `signatureId` at the port of the component `componentId` to which the TSI port `tsiPortId` is mapped. The decoding of procedure parameters has to be done in the TE.
No error shall be indicated by the TE in case the value of any *out* parameter is non-null.

- ***triEnqueueReply*** **SA → TE**

Signature
```
void triEnqueueReply(in TriPortIdType        tsiPortId,
                     in TriAddressType       SUTaddress,
                     in TriComponentIdType   componentId,
                     in TriSignatureIdType   signatureId,
                     in TriParameterListType parameterList,
                     in TriParameterType     returnValue)
```

In Parameters
`tsiPortId` identifier of the test system interface port via which the reply is enqueued by the SUT Adapter.
`SUTaddress` (optional) source address within the SUT.
`componentId` identifier of the receiving test component.
`signatureId` identifier of the signature of the procedure call.
`parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration.
`returnValue` (optional) encoded return value of the procedure call.

Out Parameters n.a.

Return Value void

Constraints This operation can be called by the SA after it has received a reply from the SUT. It can only be used when `tsiPortId` has been either previously mapped to a port of `componentId` or referenced in the previous `triExecute` statement.
In the invocation of a `triEnqueueReply` operation all *out* and *inout* procedure parameters and the return value contain encoded values. All *in* procedure parameters shall contain the distinct value of `null` since they are only of relevance to the procedure call but not in the reply to the call. If no return type has been defined for the procedure signature in the TTCN-3 ATS, the distinct value `null` shall be used for the return value.

Effect The TE can enqueue this reply to the procedure call with the signature identifier `signatureId` at the port of the component `componentId` to which the TSI port `tsiPortId` is mapped. The decoding of the procedure parameters has to be done within the TE.
No error shall be indicated by the TE in case the value of any *in* parameter or a non-defined return value is non-null.

- ***triEnqueueException*** **SA → TE**

Signature
```
void triEnqueueException (in TriPortIdType      tsiPortId,
                          in TriAddressType     SUTaddress,
                          in TriComponentIdType componentId,
                          in TriSignatureIdType signatureId,
                          in TriExceptionType   exception)
```

In Parameters
`tsiPortId` identifier for the test system interface port via which the exception is enqueued by the SUT Adapter.
`SUTaddress` (optional) source address within the SUT.
`signatureId` identifier of the signature of the procedure call which the exception is associated with.
`exception` the encoded exception.

Out Parameters n.a.

Return Value void

Constraints This operation can be called by the SA after it has received a reply from the SUT. It can only be used when `tsiPortId` has been either previously mapped to a port of `componentId` or referenced in the previous `triExecuteTestCase` statement.
In the invocation of a `triEnqueueException` operation `exception` shall contain an encoded value.

Effect The TE can enqueue this exception for the procedure call with the signature identifier `signatureId` at the port of the component `componentId` to which the TSI port `tsiPortId` is mapped.
The decoding of the exception has to be done within the TE.

7.5.4 Miscellaneous operations

- ***triSUTactionInformal*** **TE → SA**

Signature `TriStatusType triSUTactionInformal(in string description)`

In Parameters `description` an informal description of an action to be taken on the SUT.

Out Parameters n.a.

Return Value The return status of the `triSUTactionInformal` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when it executes a TTCN-3 SUT action operation, which only contains a string.

Effect On invocation of this operation the SA shall initiate the described actions to be taken on the SUT, e.g. turn on, initialize, or send a message to the SUT.
The `triSUTactionInformal` operation returns ***TRI_OK*** on successful execution of the operation, ***TRI_Error*** otherwise. Notice that the return value of this TRI operation does not make any statement about the success or failure of the actions to be taken on the SUT.

- ***triSUTactionTemplate*** **TE → SA**

Signature `TriStatusType triSUTactionTemplate(in TriActionTemplateType templateValue)`

In Parameters `templateValue` the encoded value of the action template.

Out Parameters n.a.

Return Value The return status of the `triSUTactionTemplate` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when it executes a TTCN-3 SUT action operation, which uses a template.
The encoding of the action template value has to be done in the TE prior to this TRI operation call.

Effect On invocation of this operation the SA shall initiate the actions to be taken on the SUT using the passed template value, e.g. turn on, initialize, or send a message to the SUT.
The `triSUTactionTemplate` operation returns ***TRI_OK*** on successful execution of the operation, ***TRI_Error*** otherwise. Notice that the return value of this TRI operation does not make any statement about the success or failure of the actions to be taken on the SUT.

7.6 Platform Interface Operations

- ***triPAReset*** **TE → PA**

Signature `TriStatusType triPAReset()`

In Parameters n.a.

Out Parameters n.a.

Return Value The return status of the `triPAReset` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation can be called by the TE at any time to reset the PA.

Effect The PA shall reset all timing activities which it is currently performing, e.g. stop all running timers, discard any pending timeouts of expired timers.
The `triResetSA` operation returns ***TRI_OK*** in case the operation has been performed successfully, ***TRI_Error*** otherwise.

7.6.1 Timer Operations

- ***triStartTimer*** **TE → PA**

Signature
```
TriStatusType triStartTimer(in TriTimerIdType       timerId,
                            in TriTimerDurationType
                            timerDuration)
```

In Parameters
`timerIdidentifier` of the timer instance.
`timerDuration` duration of the timer in seconds.

Out Parameters n.a.

Return Value The return status of the `triStartTimer` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when a timer needs to be started.

Effect On invocation of this operation the PA shall start the indicated timer with the indicated duration. The timer runs from the value zero (0.0) up to the maximum specified by `timerDuration`. Should the timer indicated by `timerId` already be running it is to be restarted. When the timer expires the PA will call the `triTimeout()` operation with `timerId`.
The `triStartTimer` operation returns ***TRI_OK*** if the timer has been started successfully, ***TRI_Error*** otherwise.

- ***triStopTimer*** **TE → PA**

Signature `TriStatusType triStopTimer(in TriTimerIdType timerId)`

In Parameters `timerId` identifier of the timer instance.

Out Parameters n.a.

Return Value The return status of the `triStopTimer` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when a timer is to be stopped.

Effect On invocation of this operation the PA shall use the `timerId` to stop the indicated timer instance. The stopping of an inactive timer, i.e. a timer which has not been started or has already expired, should have no effect.
The `triStopTimer` operation returns ***TRI_OK*** if the operation has been performed successfully, ***TRI_Error*** otherwise. Notice that stopping an inactive timer is a valid operation. In this case ***TRI_OK*** shall be returned.

- ***triReadTimer*** **TE → PA**

Signature
```
TriStatusType triReadTimer(in  TriTimerIdType       timerId,
                           out TriTimerDurationType elapsedTime)
```

In Parameters `timerId` identifier of the timer instance

Out Parameters `elapsedTime` value of the time elapsed since the timer has been started in seconds

Return Value The return status of the `triReadTimer` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation may be called by the TE when a TTCN-3 read timer operation is to be executed on the indicated timer (see clause 7.1.3).

Effect On invocation of this operation the PA shall use the `timerId` to access the time that elapsed since this timer was started. The return value `elapsedTime` shall be provided in seconds. The reading of an inactive timer, i.e. a timer which has not been started or already expired, shall return an elapsed time value of zero.
The `triReadTimer` operation returns ***TRI_OK*** if the operation has been performed successfully, ***TRI_Error*** otherwise.

- ***triTimerRunning*** **TE → PA**

Signature
```
TriStatusType triTimerRunning(in TriTimerIdType timerId,
                              out boolean running)
```

In Parameters `timer Id` identifier of the timer instance.

Out Parameters `running` status of the timer.

Return Value The return status of the `triTimerRunning` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation may be called by the TE when a TTCN-3 running timer operation is to be executed on the indicated timer (see clause 7.1.3).

Effect On invocation of this operation the PA shall use the `timerId` to access the status of the timer. The operation sets `running` to the boolean value `true` if and only if the timer is currently running.
The `triTimerRunning` operation returns ***TRI_OK*** if the status of the timer has been successfully determined, ***TRI_Error*** otherwise.

- ***triTimeout*** **PA → TE**

Signature
```
void triTimeout(in TriTimerIdType timerId)
```

In Parameters `timerId` identifier of the timer instance.

Out Parameters n.a.

Return Value `void`

Constraints This operation is called by the PA after a timer, which has previously been started using the `triStartTimer` operation, has expired, i.e. it has reached its maximum duration value.

Effect The timeout with the `timerId` can be added to the timeout list in the TE. The implementation of this operation in the TE has to be done in such a manner that it addresses the different TTCN-3 semantics for timers defined in [2] (see also clause 6.1.2).

7.6.2 Miscellaneous operations

- ***triExternalFunction*** **TE → PA**

Signature
```
TriStatusType triExternalFunction
     (in   TriFunctionIdType    functionId,
     inout TriParameterListType parameterList,
     out   TriParameterType     returnValue)
```

In Parameters `functionId` identifier of the external function.

InOut Parameters `parameterList` a list of encoded parameters for the indicated function. The parameters in `parameterList` are ordered as they appear in the TTCN-3 function declaration.

Out Parameters `returnValue` (optional) encoded return value.

Return Value The return status of the `triExternalFunction` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.

Constraints This operation is called by the TE when it executes a function which is defined to be TTCN-3 external (i.e. all non-external functions are implemented within the TE).
In the invocation of a `triExternalFunction` operation by the TE all *in* and *inout* function parameters contain encoded values. All *out* function parameters shall contain the distinct value of `null` since they are only of relevance in the return from the external function but not in its invocation. No error shall be indicated by the PA in case the value of any *out* parameter is non-null.

Effect For each external function specified in the TTCN-3 ATS the PA shall implement the behaviour.
On invocation of this operation the PA shall invoke the function indicated by the identifier `functionId`. It shall access the specified *in* and *inout* function parameters in `parameterList`, evaluate the external function using the values of these parameters, and compute values for *inout* and *out* parameters in `parameterList`. The operation shall then return encoded values for all *inout* and *out* function parameters, the distinct value of `null` for all *in* parameters, and the encoded return value of the external function.
If no return type has been defined for this external function in the TTCN-3 ATS, the distinct value `null` shall be used for the latter.
The `triExternalFunction` operation returns ***TRI_OK*** if the PA completes the evaluation of the external function successfully, ***TRI_Error*** otherwise.
Note that whereas all other TRI operations are considered to be non-blocking, the `triExternalFunction` operation is considered to be ***blocking***. That means that the operation shall not return before the indicated external function has been fully evaluated. External functions have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation.

8 Java Language Mapping

8.1 Introduction

This clause introduces the TRI Java language mapping. For efficiency reasons a dedicated language mapping is introduced instead of using the OMG IDL to Java language.

The Java language mapping for the TTCN-3 Runtime Interface defines how the IDL definitions described in clause 7 are mapped to the Java language. The language mapping is independent of the used Java version as only basic Java language constructs are used.

8.2 Names and Scopes

8.2.1 Names

Although there are no conflicts between identifiers used in the IDL definition and the Java language some naming translation rules are applied to the IDL identifiers.

- Java parameter identifiers shall start with a lower case letter, and subsequent part building the parameter identifier start with a capital letter. For example the IDL parameter identifier **`SUTaddress`** maps to `sutAddress` in Java.
- Java interfaces or class identifiers are omitting the trailing `Type` used in the IDL definition. For example the IDL type **`TriPortIdType`** maps to `TriPortId` in Java.

The resulting mapping conforms with the standard Java coding conventions.

8.2.2 Scopes

The IDL module **`triInterface`** is mapped to the Java package `org.etsi.ttcn3.tri`. All IDL type declarations within this module are mapped to Java classes or interface declarations within this package.

8.3 Type Mapping

8.3.1 Basic Type Mapping

Table 3 gives an overview on how the used basic IDL types are mapped to the Java types.

Table 3: Basic type mappings

IDL Type	Java Type
`boolean`	`org.etsi.ttcn.tri.TriBoolean`
`string`	`java.lang.String`

Other IDL basic types are not used within the IDL definition.

8.3.1.1 Boolean

The IDL **`boolean`** type is mapped to the interface `org.etsi.ttcn.tri.TriBoolean`, so that objects implementing this interface can act as holder objects.

The following interface is defined for `org.etsi.ttcn.tri.TriBoolean:`

```
// TriBoolean
package org.etsi.ttcn.tri;
public interface TriBoolean {
    public void    setBooleanValue(boolean value);
    public boolean getBooleanValue();
}
```

Methods

- `setBooleanValue(boolean value)`

  Sets this `TriBoolean` to the boolean value `value`.

- `getBooleanValue()`

  Returns the boolean value represented by this `TriBoolean`.

8.3.1.2 String

The IDL **`string`** type is mapped to the `java.lang.String` class without range checking or bounds for characters in the string. All possible strings defined in TTCN-3 can be converted to `java.lang.String`.

8.3.2 Structured Type Mapping

The TRI IDL description defines user defined types as native types. In the Java language mapping these types are mapped to Java interfaces. The interfaces define methods and attributes being available for objects implementing this interface.

8.3.2.1 TriPortIdType

`TriPortIdType` is mapped to the following interface:

```
// TRI IDL TriPortIdType
package org.etsi.ttcn.tri;
public interface TriPortId {
    public String         getPortName();
    public TriComponentId getComponent();
    public boolean        isArray();
    public int            getPortIndex();
}
```

Methods

- `getPortName()`

  Returns the port name as defined in the TTCN-3 specification.

- `getComponent()`

  Returns the component identifier that this `TriPortId` belongs to as defined in the TTCN-3 specification.

- `isArray()`

  Returns `true` if this port is part of an port array, `false` otherwise.

- `getPortIndex()`

  Returns the port index if this port is part of a port array starting at zero. If the port is not part of a port array, then -1 is returned.

8.3.2.2 TriPortIdListType

`TriPortIdListType` is mapped to the following interface:

```
// TRI IDL TriPortIdListType
package org.etsi.ttcn.tri;
public interface TriPortIdList {
    public int                  size();
    public boolean              isEmpty();
    public java.util.Enumeration getPortIds();
    public TriPortId            get(int index);
}
```

Methods

- `size()`

  Returns the number of ports in this list.

- `isEmpty()`

  Returns true if this list contains no ports.

- `getPortIds()`

  Returns an Enumeration over the ports in the list. The enumeration provides the ports in the same order as they appear in the list.

- `get(int index)`

  Returns the `TriPortId` at the specified position.

8.3.2.3 TriComponentIdType

`TriComponentIdType` is mapped to the following interface:

```
// TRI IDL TriComponentIdType
package org.etsi.ttcn.tri;
public interface TriComponentId {
    public String        getComponentId();
    public String        getComponentTypeName();
    public TriPortIdList getPortList();
    public boolean       equals(TriComponentId port);
}
```

Methods

- `getComponentId()`

  Returns a representation of this unique component identifier.

- `getComponentTypeName()`

  Returns the component type name as defined in the TTCN-3 specification.

- `getPortList()`

  Returns the component's port list as defined in the TTCN-3 specification.

- `equals(TriComponentId component)`

  Compares component with this `TriComponentId` for equality. Returns `true` if and only if both components have the same representation of this unique component identifier, `false` otherwise.

8.3.2.4 TriMessageType

`TriMessageType` is mapped to the following interface:

```
// TRI IDL TriMessageType
package org.etsi.ttcn.tri;
public interface TriMessage {
    public byte[]  getEncodedMessage();
    public void    setEncodedMessage(byte[] message);
    public boolean equals(TriMessage message);
}
```

Methods

- `getEncodedMessage()`

  Returns the message encoded according the coding rules defined in the TTCN-3 specification.

- `setEncodedMessage(byte[] message)`

  Sets the encoded message representation of this `TriMessage` to `message`.

- `equals(TriMessage message)`

  Compares message with this `TriMessage` for equality. Returns `true` if and only if have the same encoded representation, `false` otherwise.

8.3.2.5 TriAddressType

`TriAddressType` is mapped to the following interface:

```
// TRI IDL TriAddressType
package org.etsi.ttcn.tri;
public interface TriAddress {
    public byte[]  getEncodedAddress();
    public void    setEncodedAddress(byte[] address);
    public boolean equals(TriAddress address);
}
```

Methods

- `getEncodedAddress()`

  Returns the encoded address.

- `setEncodedAddress(byte[] address)`

  Set the encoded address of this `TriAddress` to `address`.

- `equals(TriAddress address)`

  Compares address with this `TriAddress` for equality. Returns `true` if and only if have the same encoded representation, `false` otherwise.

8.3.2.6 TriSignatureIdType

`TriSignatureIdType` is mapped to the following interface:

```
// TRI IDL TriSignatureIdType
package org.etsi.ttcn.tri;
public interface TriSignatureId {
    public String  getSignatureName();
    public void    setSignatureName(String sigName);
    public boolean equals(TriSignatureId sig);
}
```

Methods

- `getSignatureName()`

  Returns the signature identifier as defined in the TTCN-3 specification.

- `setSignatureName(String sigName)`

  Sets the signature identifier of this `TriSignatureId` to `sigName`.

- `equals(TriSignatureId sig)`

  Compares sig with this `TriSignatureId` for equality. Returns `true` if and only if both signatures have the same signature identifier, `false` otherwise.

8.3.2.7 TriParameterType

`TriParameterType` is mapped to the following interface:

```
// TRI IDL TriParameterType
package org.etsi.ttcn.tri;
public interface TriParameter {
    public String getParameterName();
    public void   setParameterName(String name);
    public int    getParameterPassingMode();
    public void   setParameterPassingMode(in mode);
    public byte[] getEncodedParameter();
    public void   setEncodedParameter(byte[] parameter);
}
```

8.3.2.8 Methods

- `getParameterName()`

  Returns the parameter name as defined in the TTCN-3 specification.

- `setParameterName(String name)`

  Sets the name of this `TriParameter` parameter to `name`.

- `getParameterPassingMode()`

  Returns the parameter passing mode of this parameter.

- `setParameterPassingMode(int mode)`

  Sets the parameter mode of this `TriParameter` parameter to `mode`.

- `getEncodedParameter()`

  Returns the encoded parameter representation of this `TriParameter`, or the null object if the parameter contains the distinct value null (see also clause 7.5.3).

- `setEncodedParameter(byte[] parameter)`

  Sets the encoded parameter representation of this `TriParameter` to parameter. If the distinct value null shall be set to indicate that this parameter holds no value, the Java null shall be passed as parameter (see also clause 7.5.3).

8.3.2.9 TriParameterPassingModeType

`TriParameterPassingModeType` is mapped to the following interface:

```
// TRI IDL TriParameterPassingModeType
package org.etsi.ttcn.tri;
public interface TriParameterPassingMode {
    public final static int TRI_IN    = 0;
    public final static int TRI_INOUT = 1;
    public final static int TRI_OUT   = 2;
}
```

Constants

- `TRI_IN`

  Will be used to indicate that a `TriParameter` is an in parameter.

- `TRI_INOUT`

  Will be used to indicate that a `TriParameter` is an inout parameter.

- `TRI_OUT`

  Will be used to indicate that a `TriParameter` is an out parameter.

8.3.2.10 TriParameterListType

`TriParameterListType` is mapped to the following interface:

```
// TRI IDL TriParameterListType
package org.etsi.ttcn.tri;
public interface TriParameterList {
    public int                  size();
    public boolean              isEmpty();
    public java.util.Enumeration getParameters();
    public TriParameter         get(int index);
    public void                 clear();
    public void                 add(TriParameter parameter);
}
```

Methods

- `size()`

  Returns the number of parameters in this list.

- `isEmpty()`

  Returns true if this list contains no parameters.

- `getParameters()`

  Returns an Enumeration over the parameters in the list. The enumeration provides the parameters in the same order as they appear in the list.

- `get(int index)`

  Returns the `TriParameter` at the specified position.

- `clear()`

  Removes all parameters from this `TriParameterList`.

- `add(TriParameter parameter)`

  Adds `parameter` to the end of this `TriParameterList`.

8.3.2.11 TriExceptionType

`TriExceptionType` is mapped to the following interface:

```
// TRI IDL TriExceptionType
package org.etsi.ttcn.tri;
public interface TriException {
    public byte[]  getEncodedException();
    public void    setEncodedException(byte[] message);
    public boolean equals(TriException exc);
}
```

Methods

- `getEncodedException()`

  Returns the exception encoded according to the coding rules defined in the TTCN-3 specification.

- `setEncodedMessage(byte[] exc)`

  Sets the encoded exception representation of this `TriException` to `exc`.

- `equals(TriException exc)`

  Compares exc with this `TriException` for equality. Returns `true` if and only if both exceptions have the same encoded representation, `false` otherwise.

8.3.2.12 TriTimerIdType

`TriTimerIdType` is mapped to the following interface:

```
// TRI IDL TriTimerIdType
package org.etsi.ttcn.tri;
public interface TriTimerId {
    public String  getTimerName();
    public boolean equals(TriTimerId timer);
}
```

Methods

- `getTimerName()`

  Returns the name of this timer identifier as defined in the TTCN-3 specification. In case of implicit timers the result is implementation dependent (see clause 6.1.2).

- `equals(TriTimerId timer)`

  Compares timer with this `TriTimerId` for equality. Returns `true` if and only if both timers identifiers represent the same timer, `false` otherwise.

8.3.2.13 TriTimerDurationType

`TriTimerDurationType` is mapped to the following interface:

```
// TRI IDL TriTimerDurationType
package org.etsi.ttcn.tri;
public interface TriTimerDuration {
    public double  getDuration();
    public void    setDuration(double duration);
    public boolean equals(TriTimerDuration duration);
}
```

Methods

- `getDuration()`

  Returns the duration of a timer as double.

- `setDuration(double duration)`

  Sets the duration of this `TriTimerDuration` to `duration`.

- `equals(TriTimerDuration duration)`

  Compares duration with this `TriTimerDuration` for equality. Returns `true` if and only if both have the same duration, `false` otherwise.

8.3.2.14 TriFunctionIdType

`TriFunctionIdType` is mapped to the following interface:

```
// TRI IDL TriFunctionIdType
package org.etsi.ttcn.tri;
public interface TriFunctionId {
    public String  toString();
    public String  getFunctionName();
    public boolean equals(TriFunctionId fun);
}
```

8.3.2.15 Methods

- `toString()`

  Returns the string representation of the function as defined in TTCN-3 specification.

- `getFunctionName()`

  Returns the function identifier as defined in the TTCN-3 specification.

- `equals(TriFunctionId fun)`

  Compares fun with this `TriFunctionId` for equality. Returns `true` if and only if both functions have the same function identifier, `false` otherwise.

8.3.2.16 TriTestCaseIdType

`TriTestCaseIdType` is mapped to the following interface:

```
// TRI IDL TriTestCaseIdType
package org.etsi.ttcn.tri;
public interface TriTestCaseId {
    public String  toString();
    public String  getTestCaseName();
    public boolean equals(TriTestCaseId tc);
}
```

Methods

- `toString()`

  Returns the string representation of the test case as defined in TTCN-3 specification.

- `getTestCaseName()`

  Returns the test case identifier as defined in the TTCN-3 specification.

- `equals(TriTestCaseId tc)`

  Compares tc with this `TriTestCaseId` for equality. Returns `true` if and only if both test cases have the same test case identifier, `false` otherwise.

8.3.2.17 TriActionTemplateType

`TriActionTemplateType` is mapped to the following interface:

```
// TRI IDL TriActionTemplateType
package org.etsi.ttcn.tri;
public interface TriActionTemplate {
    public String  toString();
    public byte[]  getEncodedTemplate();
    public boolean equals(TriActionTemplate actionTemplate);
}
```

Methods

- `toString()`

  Returns the string representation of the encoded `TriActionTemplate`.

- `getEncodedTemplate()`

  Returns the encoded template according the coding rules defined in the TTCN-3 specification.

- `equals(TriActionTemplate actionTemplate)`

  Compares `actionTemplate` with this `TriActionTemplate` for equality. Returns `true` if and only if have the same encoded representation, `false` otherwise.

8.3.2.18 TriStatusType

`TriStatusType` is mapped to the following interface:

```
// TriStatusType
package org.etsi.ttcn.tri;
public interface TriStatus {
    public final static int TRI_OK      =  0;
    public final static int TRI_ERROR   = -1;

    public String  toString();
    public int     getStatus();
    public void    setStatus(int status);
    public boolean equals(TriStatus status);
}
```

Methods

- `toString()`

  Returns the string representation of the status.

- `getStatus()`

  Returns the status of this `TriStatus`.

- `setStatus(int status)`

  Sets the status of this `TriStatus`.

- `equals(TriStatus status)`

  Compares status with this `TriStatus` for equality. Returns `true` if and only if they have the same status, `false` otherwise.

8.4 Constants

Within this Java language mapping constants have been specified. All constants are defined `public final static` and are accessible from every object from every package. The constants defined within this clause are not defined with the IDL section. Instead they result from the specification of the TRI IDL types marked as native.

The following constants can be used to determine the parameter passing mode of TTCN-3 parameters (see also clause 8.3.2.7).

```
org.etsi.ttcn.tri.TriParameterPassingMode.TRI_IN;
org.etsi.ttcn.tri.TriParameterPassingMode.TRI_INOUT;
org.etsi.ttcn.tri.TriParameterPassingMode.TRI_OUT;
```

For the distinct parameter value `null`, the encoded parameter value shall be set to Java `null`.

The following constants shall be used to indicate the local success of an method (see also clause 8.3.2.18)

```
org.etsi.ttcn.tri.TriStatus.TRI_OK;
org.etsi.ttcn.tri.TriStatus.TRI_ERROR;
```

8.5 Mapping of Interfaces

The TRI IDL definition defines two interfaces, the **`triCommunication`** and the **`triPlatform`** interface. As the operations are defined for different directions within this interface, i.e. some operations can only be called by the TTCN-3 Executable (TE) on the System Adapter (SA) while others can only be called by the SA on the TE. This is reflected by dividing the TRI IDL interfaces in two sub interfaces, each suffixed by the called entity.

Table 4: Sub Interfaces

Calling/Called	TE	SA	PA
TE	-	`TriCommunicationSA`	`triPlatformPA`
SA	`TriCommunicationTE`	-	-
PA	`TriPlatformTE`	-	-

All methods defined in this interfaces should behave as defined in clause 7.

8.5.1 Out and InOut Parameter Passing Mode

The following IDL types are used in `out` or `inout` parameter passing mode:

- `TriParameter`
- `TriParameterList`
- `TriBoolean`
- `TriTimerDuration`

In case they are used in `out` or `inout` parameter passing mode objects of the respective class will be passed with the method call. The called entity can then access methods to set the return values.

8.5.2 triCommunication - Interface

The **triCommunication** interface is divided into two sub interfaces, the triCommunicationSA interface, defining calls from the TE to the SA and the triCommunicationTE interface, defining calls from the SA to the TE.

8.5.2.1 triCommunicationSA

The triCommunicationSA interface is mapped to the following interface:

```
// TriCommunication
// TE -> SA
package org.etsi.ttcn.tri;
public interface TriCommunicationSA {
    // Reset Operation
    // Ref: TRI-Definition 7.5.1
    TriStatus triSAReset();

    // Connection handling operations
    // Ref: TRI-Definition 7.5.2
    public TriStatus triExecuteTestcase(TriTestCaseId
        testCaseId,TriPortIdList tsiPorts);
    // Ref: TRI-Definition 7.5.3
    public TriStatus triMap(TriPortId compPortId, TriPortId tsiPortId);
    // Ref: TRI-Definition 7.5.4
    public TriStatus triUnmap(TriPortId compPortId, TriPortId tsiPortId);

    // Message based communication operations
    // Ref: TRI-Definition 7.5.5
    public TriStatus triSend(TriComponentId componentId, TriPortId tsiPortId,
            TriAddress address, TriMessage sendMessage);

    // Procedure based communication operations
    // Ref: TRI-Definition 7.5.7
    public TriStatus triCall(TriComponentId componentId,
            TriPortId tsiPortId, TriAddress sutAddress,
            TriSignatureId signatureId, TriParameterList parameterList);

    // Ref: TRI-Definition 7.5.8
    public TriStatus triReply(TriComponentId componentId,
            TriPortId tsiPortId, TriAddress sutAddress,
            TriSignatureId signatureId, TriParameterList parameterList,
            TriParameter returnValue);

    // Ref: TRI-Definition 7.5.9
    public TriStatus triRaise(TriPortId tsitPortId, TriAddress sutAddress,
            TriSignatureId signatureId,
            TriException exception);

    // Miscellaneous operations
    // Ref: TRI-Definition 7.5.13
    public TriStatus triSutActionInformal(String description);

    // Ref: TRI-Definition 7.5.14
    public TriStatus triSutActionTemplate(TriActionTemplate templateValue);
}
```

8.5.2.2 triCommunicationTE

The triCommunicationTE interface is mapped to the following interface:

```
// TriCommunication
// SA -> TE
package org.etsi.ttcn.tri;
public interface TriCommunicationTE {
    // Message based communication operations
    // Ref: TRI-Definition 7.5.6
    public void triEnqueueMsg(TriPortId tsiPortId,
            TriAddress sutAddress, TriComponentId componentId,
            TriMessage receivedMessage);

    // Procedure based communication operations
    // Ref: TRI-Definition 7.5.10
    public void triEnqueueCall(TriPortId tsiPortId,
            TriAddress SUTaddress, TriComponentId componentId,
            TriSignatureId signatureId, TriParameterList parameterList );

    // Ref: TRI-Definition 7.5.11
    public void triEnqueueReply(TriPortId tsiPortId, TriAddress address,
            TriComponentId componentId, TriSignatureId signatureId,
            TriParameterList parameterList, TriParameter returnValue);
```

```
    // Ref: TRI-Definition 7.5.12
    public void triEnqueueException(TriPortId tsiPortId,
            TriAddress sutAddress, TriComponentId componentId,
        TriSignatureId signatureId, TriException exception);
}
```

8.5.3 triPlatform - Interface

The **`triPlatform`** interface is divided in two sub interfaces, the `triPlatformPA` interface, defining calls from the TE to the PA and the `triPlatformTE` interface, defining calls from the PA to the TE.

8.5.3.1 TriPlatformPA

The `triPlatformPA` interface is mapped to the following interface:

```
// TriPlatform
// TE -> PA
package org.etsi.ttcn.tri;
public interface TriPlatformPA {
    // Ref: TRI-Definition 7.6.1
    public TriStatus triPAReset();

    // Timer handling operations
    // Ref: TRI-Definition 7.6.2
    public TriStatus triStartTimer(TriTimerId timerId,
        TriTimerDuration timerDuration);

    // Ref: TRI-Definition 7.6.3
    public TriStatus triStopTimer(TriTimerId timerId);

    // Ref: TRI-Definition 7.6.4
    public TriStatus triReadTimer(TriTimerId timerId,
        TriTimerDuration elapsedTime);

    // Ref: TRI-Definition 7.6.5
    public TriStatus triTimerRunning(TriTimerId timerId,
        TriBoolean running);

    // Miscellaneous operations

    // Ref: TRI-Definition 7.6.7
    public TriStatus triExternalFunction(TriFunctionId functionId,
        TriParameterList parameterList, TriParameter returnValue);
}
```

8.5.3.2 TriPlatformTE

The `triPlatformTE` interface is mapped to the following Java interface:

```
// TriPlatform
// PA -> TE
package org.etsi.ttcn.tri;
public interface TriPlatformTE {
    // Ref: TRI-Definition 7.6.6
    public void triTimeout(TriTimerId timerId);
}
```

8.6 Optional Parameters

Clause 7.4 defines that a reserved value shall be used to indicate the absence of an optional parameter. For the Java language mapping the Java `null` value shall be used to indicate the absence of an optional value. For example if in the `triSend` operation the address parameter shall be omitted the operation invocation shall be `triSend(componentId, tsiPortId, null, sendMessage)`.

8.7 TRI Initialization

All methods are non-static, i.e. operations can only be called on objects. As the present document does not define concrete implementation strategies of TE, SA, and PA the mechanism how the TE, the SA, or the PA get to know the handles on the respective objects is out of scope of the present document.

Tool vendors shall provide methods to the developers of SA and PA to register the TE, SA, and PA to their respective communication partner.

8.8 Error Handling

Beside the error handling as defined in clause 7.2 no additional error handling is defined within this Java language mapping. In particular no exception handling mechanisms are defined.

9 ANSI C Language Mapping

9.1 Introduction

This clause defines the TRI ANSI-C language mapping for the abstract data types specified in clause 7.3. For basic IDL types, the mapping conforms to OMG recommendations.

9.2 Names and scopes

- C parameter identifiers shall start with a lower case letter, and subsequent part building the parameter identifier start with a capital letter. For example the IDL parameter `SUTaddress` maps to `sutAddress` in C.
- Abstract data type identifiers in C are omitting the trailing `Type` used in the IDL definition. For example the IDL type `TriPortIdType` maps to `TriPortId` in C.

Older C specifications have restricted the identifier uniqueness to the most significant 8 characters. Nevertheless, the recent ANSI-C specifications have moved this limitation to the 31 most significant characters. Aside from this issue, no naming or scope conflicts have been identified in this mapping.

9.2.1 Abstract Type Mapping

TRI ADT	ANSI C Representation	Notes and comments
TriActionTemplate	BinaryString	
TriAddress	BinaryString	
TriComponentId	typedef struct TriComponentId { BinaryString compInst; QualifiedName compType; } TriComponentId;	compInst is for component instance.
TriException	BinaryString	
TriFunctionId	QualifiedName	
TriMessage	BinaryString	
TriParameterList	typedef struct TriParameterList { TriParameter* parList[]; long int length; } TriParameterList;	No special values mark the end of parList[]. The length field shall be used to traverse this array properly.
TriParameter	typedef struct TriParameter { BinaryString par; TriParameterPassingMode mode; } TriParameter;	
TriParameterPassingMode	typedef enum { TRI_IN = 0, TRI_INOUT = 1, TRI_OUT = 2 } TriParameterPassingMode;	
TriPortIdList	typedef struct TriPortIdList { TriParameter* portIdList[]; long int length; } TriPortIdList;	No special values mark the end of portIdList[]. The length field shall be used to traverse this array properly.
TriPortId	typedef struct TriPortId { TriComponentId compInst; char* portName; long int portIndex; QualifiedName portType; void* aux; } TriPortId;	compInst is for component instance. For a singular (non-array) declaration, the portIndex value should be -1. The aux field is for future extensibility of TRI functionality.
TriSignatureId	QualifiedName	
TriStatus	long int #define TRI_ERROR -1 #define TRI_OK 0	All negative values are reserved for future extension of TRI functionality.
TriTestCaseId	QualifiedName	
TriTimerDuration	double	
TriTimerId	BinaryString	NOTE: Pending ETSI statement on timer and snapshot semantics may influence future representation.

9.2.2 ANSI C Type Definitions

C ADT	Type definition	Notes and comments

BinaryString	typedef struct BinaryString
	{
	  unsigned char* data;
	  long int bits;
	  void* aux;
	} BinaryString;
		data is a non-null-terminated string. bits is the number of bits used in data. bits value -1 is used to denote omitted value.
		The aux field is for future extensibility of TRI functionality.

QualifiedName	typedef struct QualifiedName
	{
	  char* moduleName;
	  char* objectName;
	  void* aux;
	} QualifiedName;
		The moduleName and objectName fields are the TTCN-3 identifiers literally.
		The aux field is for future extensibility of TRI functionality.

9.2.3 IDL Type Mapping

IDL type	ANSI C Representation	Notes and comments
Boolean	unsigned char	From OMG IDL to C++ mapping
String	char*	From OMG IDL to C++ mapping

9.2.4 TRI Operation Mapping

IDL Representation	ANSI C Representation
TriStatusType triSAReset()	TriStatus triSAReset()

TriStatusType triExecute
  (in TriTestCaseIdType testCaseId,
  in TriPortIdListType tsiPortList)
	TriStatus triExecute
	  (const TriTestCaseId* testCaseId,
	  const TriPortIdList* tsiPortList)

TriStatusType triMap
  (in TriPortIdType compPortId,
  in TriPortIdType tsiPortId)
	TriStatus triMap
	  (const TriPortId* compPortId,
	  const TriPortId* tsiPortId)

TriStatusType triUnmap
  (in TriPortIdType compPortId,
  in TriPortIdType tsiPortId)
	TriStatus triUnmap
	  (const TriPortId* compPortId,
	  const TriPortId* tsiPortId)

TriStatusType triSend
  (in TriComponentIdType componentId,
  in TriPortIdType tsiPortId,
  in TriAddressType SUTaddress,
  in TriMessageType sendMessage)
	TriStatus triSend
	  (const TriComponentId* componentId,
	  const TriPortId* tsiPortId,
	  const TriAddress* sutAddress,
	  const TriMessage* sendMessage)

void triEnqueueMsg
  (in TriPortIdType tsiPortId,
  in TriAddressType SUTaddress,
  in TriComponentIdType componentId,
  in TriMessageType receivedMessage)
	void triEnqueueMsg
	  (const TriPortId* tsiPortId,
	  const TriAddress* sutAddress,
	  const TriComponentId* componentId,
	  const TriMessage* receivedMessage)

TriStatusType triCall
  (in TriComponentIdType componentId,
  in TriPortIdType tsiPortId,
  in TriAddressType SUTaddress,
  in TriSignatureIdType signatureId,
  in TriParameterListType parameterList)
	TriStatus triCall
	  (const TriComponentId* componentId,
	  const TriPortId* tsiPortId,
	  const TriAddress* sutAddress,
	  const TriSignatureId* signatureId,
	  const TriParameterList* parameterList)

TriStatusType triReply
  (in TriComponentIdType componentId,
  in TriPortIdType tsiPortId,
  in TriAddressType SUTaddress,
	TriStatus triReply
	  (const TriComponentId* componentId,
	  const TriPortId* tsiPortId,
	  const TriAddress* sutAddress,
	  const TriSignatureId* signatureId,

IDL Representation	ANSI C Representation
in TriSignatureIdType signatureId, in TriParameterListType parameterList, in TriParameterType returnValue)	const TriParameterList* parameterList, const TriParameter* returnValue)
TriStatusType triRaise (in TriComponentIdType componentId, in TriPortIdType tsiPortId, in TriAddressType SUTaddress, in TriSignatureIdType signatureId, in TriExceptionType exception)	TriStatus triRaise (const TriComponentId* componentId, const TriPortId* tsiPortId, const TriAddress* sutAddress, const TriSignatureId* signatureId, const TriException* exception)
void TriEnqueueCall (in TriPortIdType tsiPortId, in TriAddressType SUTaddress, in TriComponentId componentId, in TriSignatureIdType signatureId, in TriParameterListType parameterList)	void TriEnqueueCall (const TriPortId* tsiPortId, const TriAddress* sutAddress, const TriComponentId* componentId, const TriSignatureId* signatureId, const TriParameterList* parameterList)
void TriEnqueueReply (in TriPortIdType tsiPortId, in TriAddressType SUTaddress, in TriComponentIdType componentId, in TriSignatureIdType signatureId, in TriParameterListType parameterList, in TriParameterType returnValue)	void TriEnqueueReply (const TriPortId* tsiPortId, const TriAddress* sutAddress, const TriComponentId* componentId, const TriSignatureId* signatureId, const TriParameterList* parameterList, const TriParameter* returnValue)
void TriEnqueueException (in TriPortIdType tsiPortId, in TriAddressType SUTaddress, in TriComponentIdType componentId, in TriSignatureIdType signatureId, in TriExceptionType exception)	void TriEnqueueException (const TriPortId* tsiPortId, const TriAddress* sutAddress, const TriComponentId* componentId, const TriSignatureId* signatureId, const TriException* exception)
TriStatusType triSUTActionInformal (in string description)	TriStatus triSUTActionInformal (const char* description)
TriStatusType triSUTActionTemplate (in TriActionTemplateType templateValue)	TriStatus triSUTActionTemplate (const TriActionTemplate* templateValue)
TriStatusType triPAReset()	TriStatus triPAReset()
TriStatusType triStartTimer (in TriTimerIdType timerId, in TriTimerDurationType timerDuration)	TriStatus triStartTimer (const TriTimerId* timerId, TriTimerDuration timerDuration)
TriStatusType triStopTimer (in TriTimerIdType timerId)	TriStatus triStopTimer (const TriTimerId* timerId)
TriStatusType triReadTimer (in TriTimerIdType timerId, out TriTimerDurationType elapsedTime)	TriStatus triReadTimer (const TriTimerId* timerId, TriTimerDuration* elapsedTime)
TriStatusType triTimerRunning (in TriTimerIdType timerId, out boolean running)	TriStatus triTimerRunning (const TriTimerId* timerId, unsigned char* running)
void triTimeout (in TriTimerIdType timerId)	void triTimeout (const TriTimerId* timerId)

IDL Representation	ANSI C Representation
TriStatusType triExternalFunction (in TriFunctionIdType functionId, inout TriParameterListType parameterList, out TriParameterType returnValue)	TriStatus triExternalFunction (const TriFunctionId* functionId, TriParameterList* parameterList, TriParameter* returnValue)

9.3 Memory Management

The called party is responsible for saving any data in its own scope if the data is to be used after function termination. In other words, the calling party does not guarantee that a pointer is valid after the function call.

9.4 Error handling

No error handling has been defined for this mapping.

10 Use Scenarios

This clause contains use scenarios that should help users of the TRI and tool vendors providing the TRI understand the semantics of the operations defined within the present document.

Three scenarios are defined in terms of Message Sequence Charts (MSC). A scenario consists of a TTCN-3 code fragment that uses TTCN-3 communication functions to the SUT as well as timer handling functions. The MSC shows the interactions between the TE, SA, and PA entities together with the SUT.

Please note that the TTCN-3 fragments are not complete, as the main objective of the fragments are the usage of dynamic behaviour. All of the presented scenarios use a common preamble sequence of TRI operations shown in figure 3.

Notice that the MSCs presented in this clause use message pairs to model each TRI operation. The MSC message triMap followed by triMapOK denotes, for example, that the TRI operation triMap has been invoked by the TE and it returns successfully from the SA. TRI operation calls are shown using abstract types and values, and are intended to serve only for illustration purposes. The concrete representation of these parameters in a particular target language is defined in the respective language mappings.

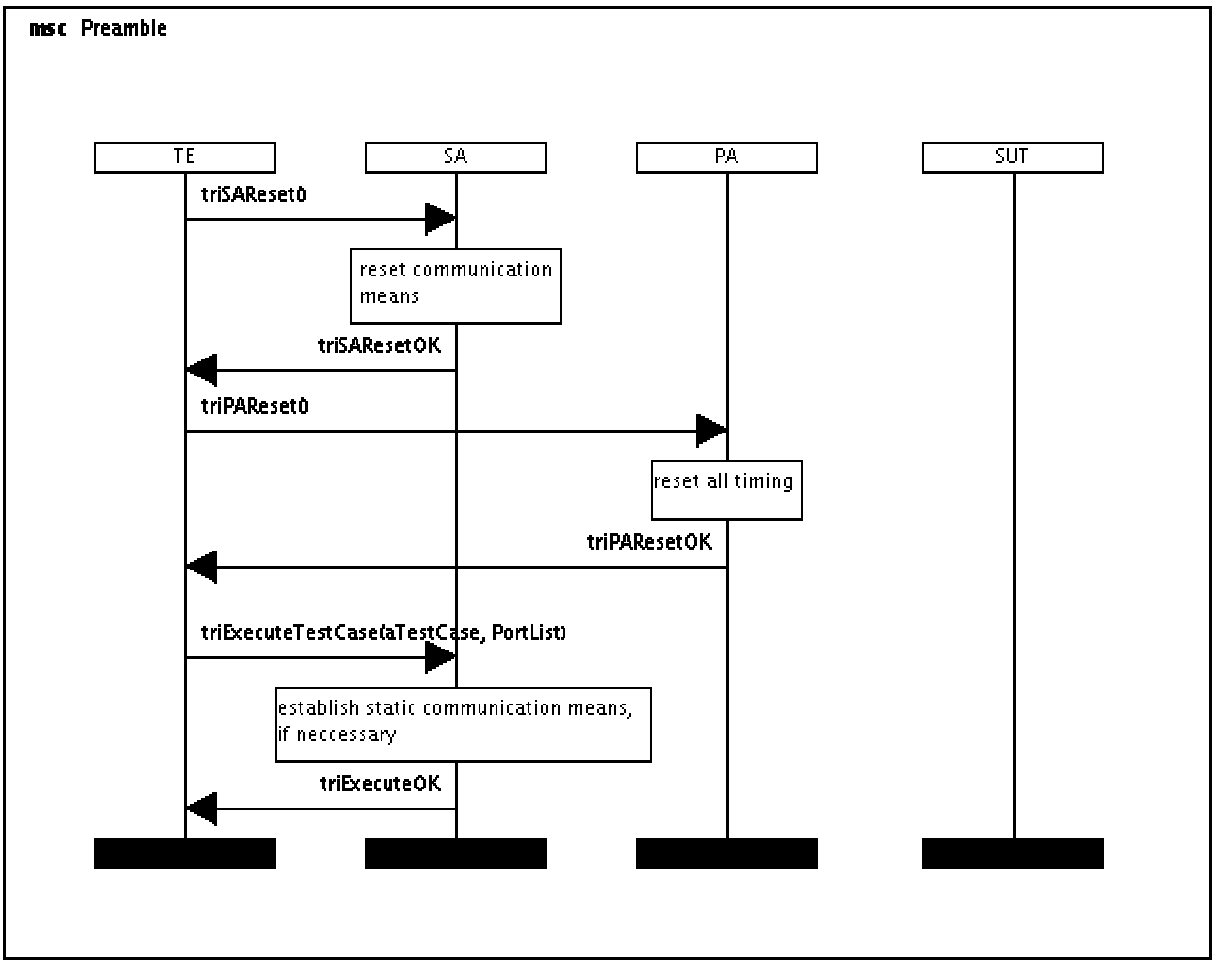


Figure 3: Common MSC Preamble

10.1 First Scenario

The first scenario shows some TTCN-3 timer operations, i.e. start and timer running, message based communication operations, i.e. send and receive, as well as connection handling operations, i.e. map and unmap.

10.1.1 TTCN-3 Fragment

```
module triScenario1
{
    external function MyFunction();

    type port PortTypeMsg message { inout integer }

    type component MyComponent {
    port PortTypeMsg MyPort;
    timer MyTimer
    }

    type component MyTSI {
    port PortTypeMsg PCO1;
    }

    testcase scenario1() runs on MyComponent system MyTSI
    {
    MyPort.clear;
    MyPort.start;
    MyTimer.start(2);

    map(MyComponent: MyPort, system: PCO1);
    MyPort.send (integer : 5);
    if (MyTimer.running)
```

```
    {
        MyPort.receive(integer:7);
    }
    else
    {
        MyFunction();
    }
    unmap(MyComponent: MyPort, system:PCO1);
    MyPort.stop;
    }


    control {
    execute( scenario1() );
    }

}
```

10.1.1.1 Message Sequence Chart

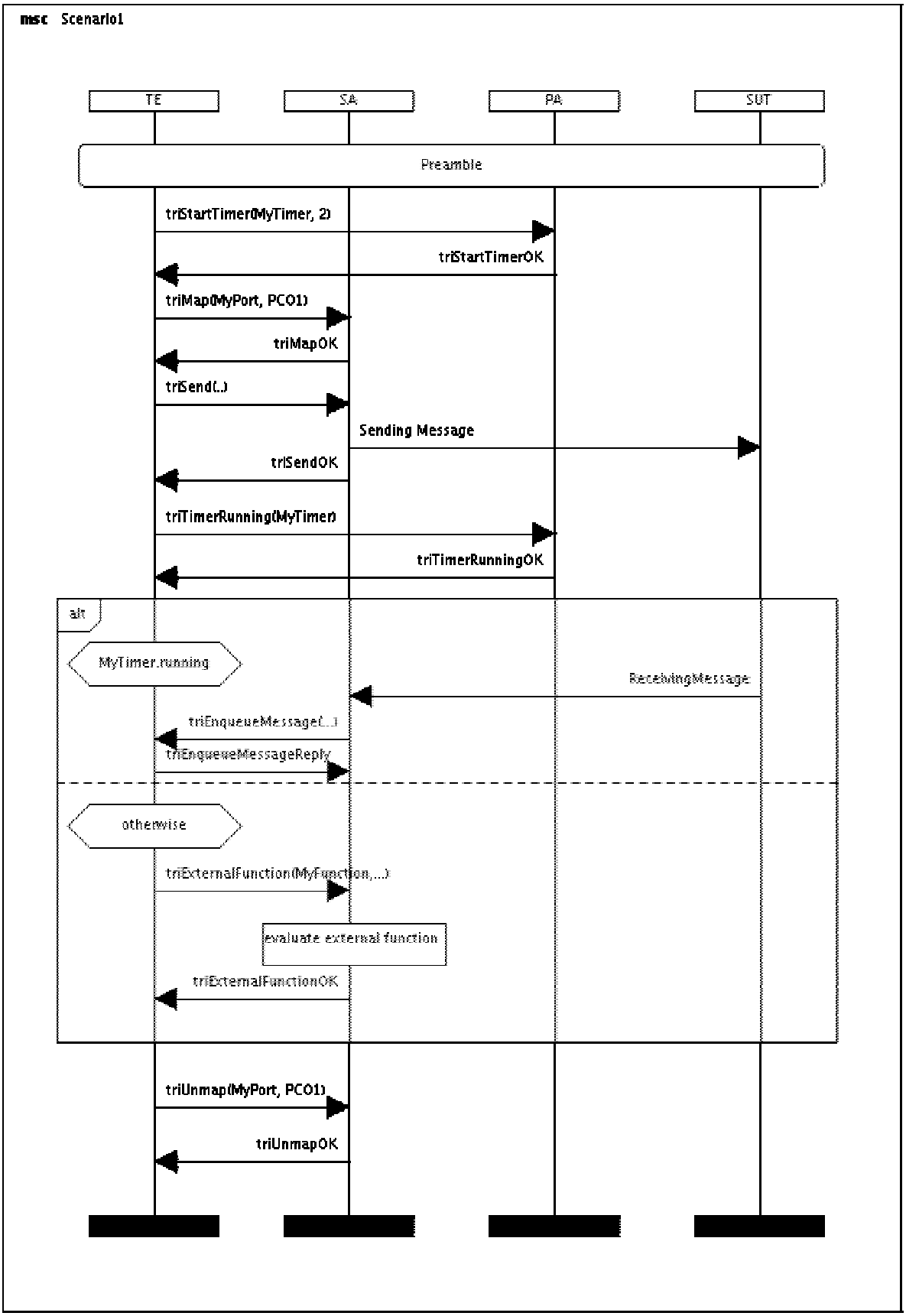


Figure 4: Use Scenario 1

10.2 Second Scenario

The second example shows a similar scenario which also uses timed procedure based communication operations which are initiated by the test component `MyComponent`. In this example `MyComponent` is assumed to run as the MTC.

10.2.1 TTCN-3 Fragment

```
module triScenario2
{

    signature MyProc ( in float par1, inout float par2)
    exception(MyExceptionType);

    type record MyExceptionType { FieldType1 par1, FieldType2 par2 }

    type port PortTypeProc procedure { out MyProc }

    type component MyComponent {
    port PortTypeProc MyPort;
    timer MyTimer = 7
    }

    testcase scenario2() runs on MyComponent
    {
    var float MyVar;

    MyPort.clear;
    MyPort.start;
    MyTimer.start;

      MyVar := MyTimer.read;

      if (MyVar>5.0) {
        MyPort.call (MyProc:{MyVar, 5.7}, 5);
        alt {
          [] MyPort.getreply(MyProc:{-,MyVar*5}) {}
          [] MyPort.catch (MyProc, MyExceptionType:* ) {}
          [] MyPort.catch (timeout) {}
        }
    }
MyTimer.stop;
    MyPort.stop;
    }

    control {
    execute( scenario2() );
    }

}
```

10.2.1.1 Message Sequence Chart

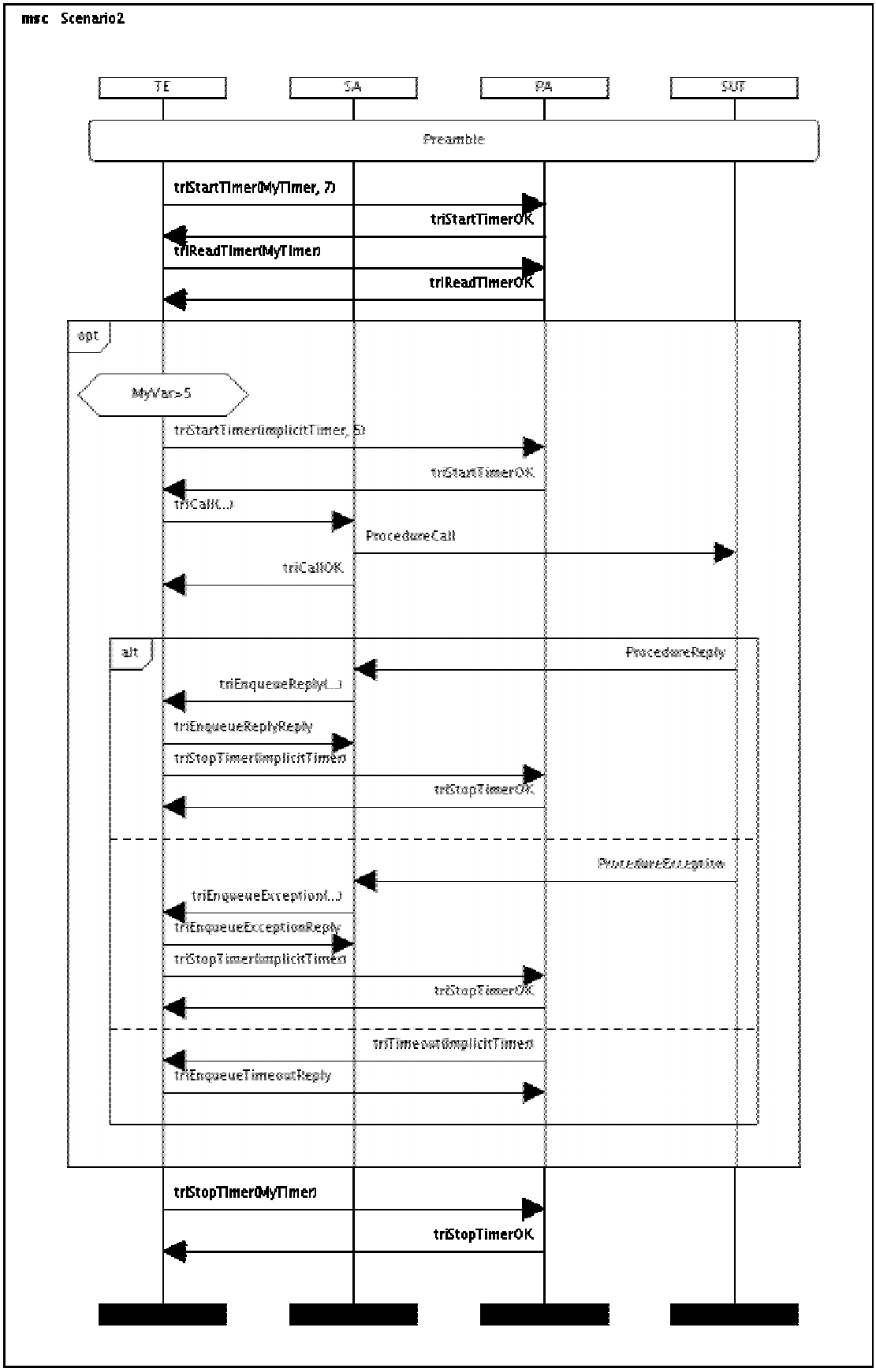


Figure 5: Use Scenario 2

10.3 Third Scenario

Use scenario 3 shows the reception of a procedure call as well as a reply and raising of an exception based on this received call. Again `MyComponent` is assumed to run as the MTC. `FieldType1`, `FieldType2`, `p1`, and `p2` are assumed to be defined elsewhere.

10.3.1 TTCN-3 Fragment

```
module triScenario3
{
    signature MyProc ( in float par1, inout float par2)
    exception(MyExceptionType);

    type record MyExceptionType { FieldType1 par1, FieldType2 par2 }

    type port PortTypeProc procedure { in MyProc }

    type component MyComponent {
    port PortTypeProc MyPort;
    timer MyTimer = 3
    }

   testcase scenario3(integer x) runs on MyComponent
    {
    MyPort.start;
    MyTimer.start;
    alt
    {
      [] MyPort.getcall(MyProc:{5.0, 6.0})
      {
          MyTimer.stop;
      }
      [x>5] MyTimer.timeout
      {
          MyPort.reply(MyProc:{-, 30.0});
      }
      [x<=5] MyTimer.timeout
      {
          MyPort.raise(MyProc, MyExceptionType:{p1, p2} );
      }
    }
    MyPort.stop;
    }

   control {
    execute( scenario3(4) );
   }
}
```

10.3.1.1 Message Sequence Chart

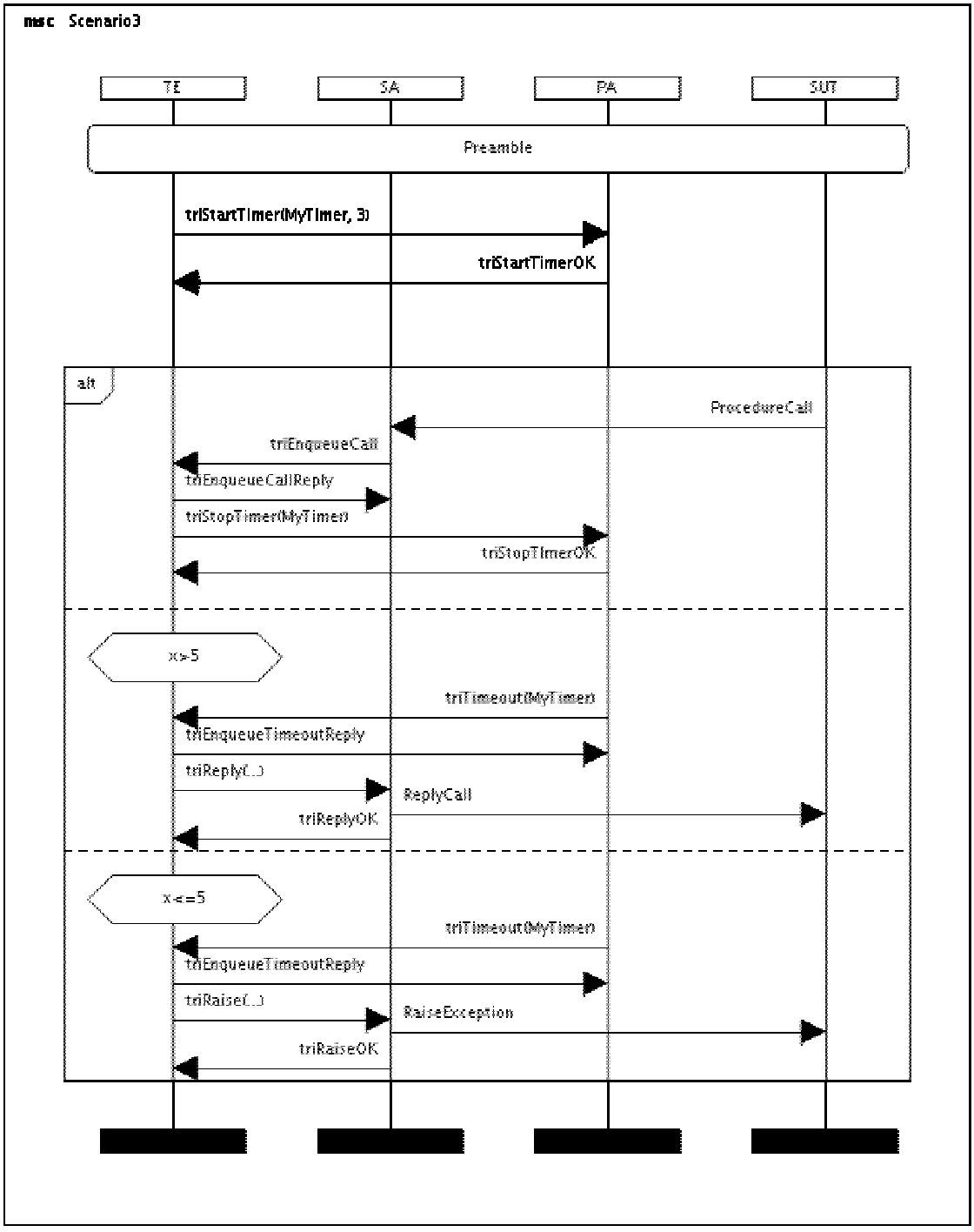


Figure 6: Use Scenario 3

Annex A:
IDL Summary

This clause summarizes the IDL definition of TRI operations as defined in clause 7.

```
// ********************************************************************
// Interface definition for the TTCN-3 Runtime Interface

// ********************************************************************

module triInterface
{

    //
    // ********************************************************************
    // Types
    // ********************************************************************
    //

    // Connection
    native TriPortIdType;
    typedef sequence<TriPortIdType> TriPortIdListType;
    native TriComponentIdType;

    // Communication
    native TriMessageType;
    native TriAddressType;
    native TriSignatureIdType;
    native TriParameterType;
    typedef sequence<TriParameterType> TriParameterListType;
    native TriExceptionType;

    // Timing
    native TriTimerIdType;
    native TriTimerDurationType;

    // Miscellaneous
    native TriFunctionIdType;
    native TriTestCaseIdType;
    native TriActionTemplateType;
    native TriStatusType;

    //
    // ********************************************************************
    // Interfaces
    // ********************************************************************
    //

    //
    // ********************************************************************
    // The communication interface (Ref: TRI-Definition: 7.5)
    // ********************************************************************
    //
    interface triCommunication
    {

    // Reset operation

    // Ref: TRI-Definition 7.5.1
    TriStatusType triSAReset();


    // Connection handling operations

    // Ref: TRI-Definition 7.5.2
    TriStatusType triExecuteTestCase(in , in TriTestCaseIdType testCaseId,              in
TriPortIdListType tsiPortList);

    // Ref: TRI-Definition 7.5.3
    TriStatusType triMap(in TriPortIdType compPortId, in TriPortIdType tsiPortId);
```

```
    // Ref: TRI-Definition 7.5.4
    TriStatusType triUnmap(in TriPortIdType compPortId, in TriPortIdType tsiPortId);


    // Message based communication operations

    // Ref: TRI-Definition 7.5.5
    TriStatusType triSend(in TriComponentIdType componentId, in TriPortIdType tsiPortId,
            in TriAddressType SUTaddress, in TriMessageType sendMessage);

    // Ref: TRI-Definition 7.5.6
    void triEnqueueMsg(in TriPortIdType tsiPortId , in TriAddressType SUTaddress,
        in TriComponentIdType componentId, in TriMessageType receivedMessage);


    // Procedure based communication operations

    // Ref: TRI-Definition 7.5.7
    TriStatusType triCall(in TriComponentIdType componentId, in TriPortIdType tsiPortId,
           in TriAddressType SUTaddress, in TriSignatureIdType signatureId,
           in TriParameterListType parameterList);

    // Ref: TRI-Definition 7.5.8
    TriStatusType triReply(in TriComponentIdType componentId, in TriPortIdType tsiPortId,
    in TriAddressType SUTaddress, in TriSignatureIdType signatureId,
    in TriParameterListType parameterList, in TriParameterType returnValue  );

    // Ref: TRI-Definition 7.5.9
    TriStatusType triRaise(in TriComponentIdType componentId, in TriPortIdType tsiPortId,       in
TriAddressType SUTaddress, in TriSignatureIdType signatureId,               in TriExceptionType
exception);

    // Ref: TRI-Definition 7.5.10
    void triEnqueueCall(in TriPortIdType tsiPortId, in TriAddressType SUTaddress,
    in TriComponentIdType componentId, in TriSignatureIdType signatureId,
    in TriParameterListType parameterList );

    // Ref: TRI-Definition 7.5.11
    void triEnqueueReply(in TriPortIdType tsiPortId, in TriAddressType SUTaddress,
        in TriComponentIdType componentId, in TriSignatureIdType signatureId,
    in TriParameterListType parameterList, in TriParameterType returnValue );

    // Ref: TRI-Definition 7.5.12
    void triEnqueueException(in TriPortIdType tsiPortId, in TriAddressType SUTaddress,
        in TriComponentIdType componentId, in TriSignatureIdType signatureId,
    in TriExceptionType exception);


    // Miscellaneous operations

// Ref: TRI-Definition 7.5.13
    TriStatusType triSUTactionInformal(in string description);

    // Ref: TRI-Definition 7.5.14
    TriStatusType triSUTactionTemplate(in TriActionTemplateType templateValue);

    };

    //
    // ************************************************************************
    // The platform interface (Ref: TRI-Definition: 7.6)
    // ************************************************************************
    //
    interface triPlatform
    {

    // Reset Operation

    // Ref: TRI-Definition 7.6.1
    TriStatusType triPAReset();


    // Timer handling operations

    // Ref: TRI-Definition 7.6.2
    TriStatusType triStartTimer(in TriTimerIdType timerId,
    in TriTimerDurationType timerDuration);
```

```
    // Ref: TRI-Definition 7.6.3
    TriStatusType triStopTimer(in TriTimerIdType timerId);

    // Ref: TRI-Definition 7.6.4
    TriStatusType triReadTimer(in TriTimerIdType timerId,                      out
TriTimerDurationType elapsedTime);

    // Ref: TRI-Definition 7.6.5
    TriStatusType triTimerRunning(in TriTimerIdType timerId, out boolean running);

    // Ref: TRI-Definition 7.6.6
    void triTimeout(in TriTimerIdType timerId);


    // Miscellaneous operations

    // Ref: TRI-Definition 7.6.7
    TriStatusType triExternalFunction(in TriFunctionIdType functionId,
        inout TriParameterListType parameterList,
        out TriParameterType returnValue);

    };
};
```

Annex B: Bibliography

OMG CORBA (V2.2): "The Common Object Request Broker: Architecture and Specification", Section 3, February 1998.

INTOOL CGI/NPL038 (V2.2): "Generic Compiler/Interpreter interface; GCI Interface Specification", Infrastructural Tools, December 1996.

History

Document history		
V1.1.1	April 2002	Publication