Final draft ETSI ES 203 119-9 V1.1.1 (2025-05)

Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 9: Test Runtime Interfaces

Reference

DES/MTS-203119-9v1.1.1

Keywords

interface, runtime, TDL, testing

Contents

Intellectual Property Rights

Foreword

This final draft ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS), and is now submitted for the ETSI Membership Approval Procedure (MAP).

The present document is part 9 of a multi-part deliverable. Full details of the entire series can be found in part 1 [1].

Modal verbs terminology

Introduction

The TDL language has been designed from the start with executability in mind. However, some of the constructs of TDL are abstract and adaptation to concrete implementation is required.

The present document provides a specification for an architecture of a test execution environment for TDL test descriptions and interfaces between the components. It provides a mapping of abstract TDL constructs to concrete implementation artifacts that are a prerequisite to produce executable TDL test descriptions. The described approach follows the commonly used 'separation of concerns' principle.

1 Scope

The present document specifies the architecture for the execution environment of TDL test descriptions and functional requirements for the components in the form of function declarations that will be provided by an implementation of the components and data types used as input and output parameters of the functions. The test executor component will interpret the elements of test descriptions according to operational semantics specified in [1].

The present document will be used for developing a code generator or interpreter for mapping abstract TDL constructs to code and the required test environment components that are specific to a chosen test execution platform.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found in the ETSI docbox.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents are necessary for the application of the present document.

[1] ETSI ES 203 119-1: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding, but are not required for conformance to the present document.

[i.1] ETSI: "TDL Open Source Project".

3 Definition of terms, symbols, abbreviations and conventions

3.1 Terms

For the purposes of the present document, the terms given in [1] and the following apply:

system adapter: entity that adapts the test executor communication operations with the SUT

test executor: application that carries out test behaviour

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the abbreviations given in [1] and the following apply:

SUT	System Under Test
TRI	Test Runtime Interfaces

3.4 Conventions

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

Meta-classes and predefined instances from the TDL meta-model [1] are typed in italic, e.g. *DataType*.

4 Test Execution Environment

4.1 Architecture

The execution environment for TDL test descriptions shall have modular architecture as shown on Figure 4.1.

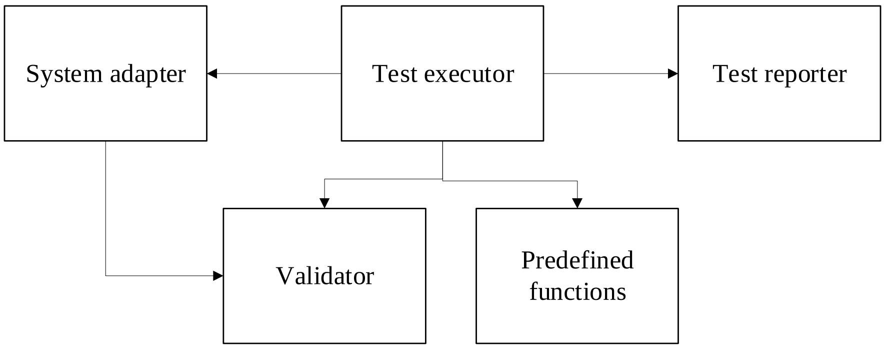

Figure 4.1: Architecture

The central component of the environment shall be the test executor. The test executor shall execute the behaviour of the elements in a test description according to the operational semantics specified in [1].

The test executor shall only execute the behaviours that occur on component instances of type 'Tester'. The execution of the behaviour of each component shall be independent of other components. This implies that the implementations of the runtime components shall be thread-safe in case the execution language supports multi-threading.

Additional requirements for the test executor are defined in clauses 4.2 and 4.3.

The specifics of the realization of the test executor and the mechanism for resolving the implementations of runtime interface components are outside the scope of the present document.

NOTE: It is a good practice to use a dependency injection mechanism for providing specific implementation for required interfaces.

The interfaces that should be realized and made available to the test executor are collectively called Test Runtime Interfaces (TRI). The following interfaces shall be implemented:

- 'System adapter' manages interactions between the test executor and the SUT;
- 'Validator' provides data matching functionality;

- 'Test reporter' implements test logging; and
- 'Predefined functions' provides implementation of TDL predefined functions.

Interface function declarations and implementation requirements are specified in clause 5.

4.2 Mappings

TDL model elements of the meta-classes *DataType* and *DataInstance* (and their sub-classes) may be mapped to specific objects or type definitions in the execution platform via *DataElementMapping* elements. The realization of the TDL test executor shall support two mechanisms for resolving the mappings.

In the first instance, the mappings shall refer to specific objects or type definitions in the execution platform programming language. The test executor shall resolve those mappings using language specific means and pass references to those objects to the runtime interface components.

In the second case, the mapping information shall be encapsulated in instances of 'Mapping' and the associated *DataType* or *DataInstance* shall be encapsulated in instances of 'Type' or 'Value' respectively.

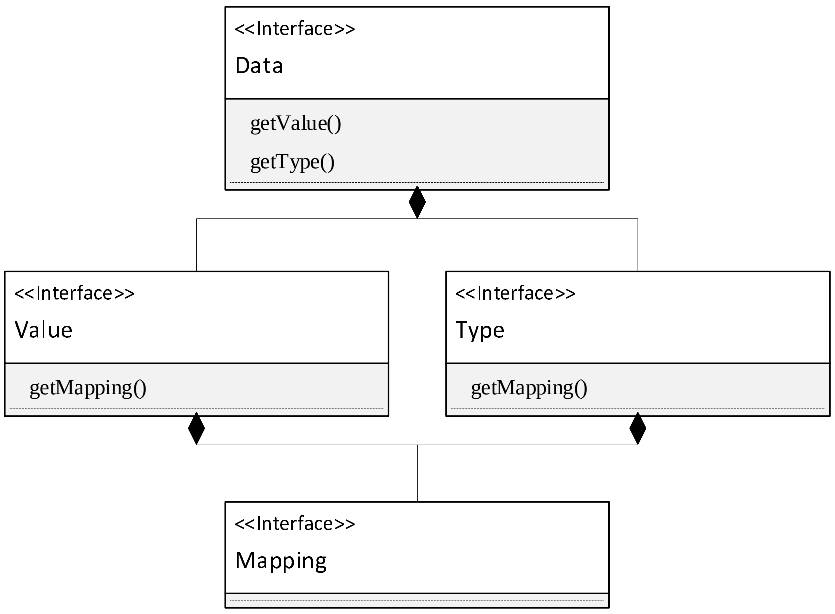

Figure 4.2: Encapsulated Data and Type

The choice of the mechanism shall be made by the concrete realization of the test executor based on the *MappingName* annotation applied to the mappings.

TDL model elements of the meta-classes *Action*, *Function* and *PredefinedFunction* shall be mapped to functions implemented in the execution language. The *MappingName* annotation shall be used to identify the appropriate mapping for the language.

4.3 Data Values

The test executor shall perform all data modifications (as specified by *MemberAssignment* and *ParameterBinding* elements) before passing the values to other components. Before parameter bindings are applied, the value instances shall be copied using language-specific means to adhere to the immutability principle of TDL data.

5 Test Runtime Interfaces

5.1 Notational Conventions

References to types and interfaces defined in the present document are typed in quotation marks, e.g. 'System adapter'.

Signatures of interface functions are defined using the following syntax:

- <function-name> (<parameters>): <return-type-declaration> <throws-declaration>

The <function-name> defines the name of the function. The <parameters> is a potentially empty set of comma-separated (,) parameter declarations in the following format:

- <parameter-name>: <type-declaration>

The <return-type-declaration> defines the type of object that the function shall return. The special keyword 'void' is used to indicate that the function does not return a value.

The <type-declaration> and <return-type-declaration> are specified in the following format:

- <type-name> [<multiplicity>] {<type-qualifiers>}

The <multiplicity> is specified using a single value that is either an integer or a special symbol * indicating unlimited multiplicity, or a range of a lower and an upper limit, e.g. 0..1. If omitted, then the multiplicity shall be 1.

The <type-qualifiers> may be used to indicate further specialization, e.g. that a multivalued return value is ordered.

The <throws-declaration> is a potentially empty set of comma-separated (,) exception types prefixed with the keyword 'throws'.

5.2 Basic Principles

The runtime interfaces are defined as sets of functions called by the test executor.

All the parameters to the functions are positional and the test executor shall pass the arguments to the functions in the order that they are defined in the function declarations. The order of items in multivalued arguments and return values shall be the same as it is in the TDL model. All objects passed as arguments or return values shall be immutable.

The test executor shall not pass undefined (or equivalent) values as arguments unless explicitly specified in the present document. Similarly, TRI components shall not return undefined values as function results. The components shall not raise exceptions other than specified in the present document or exceptions that are considered fatal in the programming language (runtime exceptions).

The programming language in which the components are implemented shall support the following:

- objects;
- instance functions;
- values of type text, number, Boolean, and enumerated types;
- collections;
- type inheritance; and
- enumerated types.

If the programming language does not support throwing (or raising) exceptions, then the exception object shall be returned as function result instead.

5.3 Overview

Clause 5.4 specifies the types for the objects that are used to pass information between the test executor and the TRI components. In addition to the types specified in the present document, the following basic types are used:

- 'Boolean' type indicates a truth value, that is true or false;
- 'Integer' type indicates a whole number;
- 'String' type indicates a textual value; and
- 'Object' type indicates a structured value (that may have a more specific type assigned depending on the implementation language).

The implementation language types used for the 'Boolean', 'Integer' and 'String' types shall correspond to the types mapped to the predefined TDL types *Boolean*, *Integer*, and *String*.

NOTE: For programming languages that may be used without explicit type definitions but that have typing support available, the implementation of the TRI should prefer including type definitions for interface types and functions.

The generalization declaration in the following clauses implies that the generalizing type shall transitively inherit all functions declared for the general type.

5.4 Types

5.4.1 Element

Description

The 'Element' type serves as super-type for other types. It corresponds to the *Element* meta-class.

Generalization

There is no generalization specified.

Functions

- getName(): String [0..1]
  Value of the 'name' property of the *Element* element.
- getAnnotations(): ElementAnnotation [0..*] {ordered}
  Objects corresponding to the 'annotation' property of the *Element* element.

5.4.2 ElementAnnotation

Description

The 'ElementAnnotation' type corresponds to the *Annotation* meta-class. The *AnnotationType* element shall be represented using its qualified name.

Generalization

There is no generalization specified.

Functions

- getKey(): String [1]
  Value of the 'qualifiedName' property of the *AnnotationType* element.

- getValue(): String [0..1]
  Value of the 'value' property of the *Annotation* element.

5.4.3 NamedElement

Description

The 'NamedElement' type serves as super-type for other types. It corresponds to *NamedElement* meta-class.

Generalization

- Element

Functions

- getQualifiedName(): String [1]
  Value of the 'qualifiedName' property of the *NamedElement* element.

5.4.4 Data

Description

The 'Data' type shall provide a data value and a data type for runtime interface functions where data is used.

The data value shall not be encoded. The value representation is implementation-specific, but it shall either be resolved by the runtime using the data mapping mechanism or it shall be an instance of the 'Value' type (see clause 5.4.9).

The data type shall describe the structure of the value. It shall either be resolved by the runtime using the data mapping mechanism or it shall be an instance of the 'Type' type (see clause 5.4.8).

NOTE: Both the value and the type may be represented by the same mechanism in order to simplify the implementation of the test executor.

The mapping mechanisms are specified in clause 4.2.

Generalization

There is no generalization specified.

Functions

- getValue(): Object [1]
  The object representing the value of the data.

- getType(): Object [1]
  The object representing the type of the data.

5.4.5 Argument

Description

The 'Argument' type extends 'Data' to provide a name of the parameter to which the data is assigned.

Generalization

- Data

Functions

- getParameterName(): String [1]
  Value of the 'name' property of the associated *Parameter* element.

5.4.6 Procedure

Description

The 'Procedure' type corresponds to the *ProcedureSignature* meta-class. The parameter objects shall be grouped by kind.

Generalization

- NamedElement

Functions

- getIn(): Parameter [0..*] {ordered}
  The parameter objects that correspond to *ProcedureParameter* elements whose kind is *ParameterKind In*.

- getOut(): Parameter [0..*] {ordered}
  The parameter objects that correspond to *ProcedureParameter* elements whose kind is *ParameterKind Out*.

- getException(): Parameter [0..*] {ordered}
  The parameter objects that correspond to *ProcedureParameter* elements whose kind is *ParameterKind Exception*.

5.4.7 Parameter

Description

The 'Parameter' type corresponds to the *Parameter* meta-class and its sub-classes. The type object is specified in clause 5.4.8.

Generalization

- Element

Functions

- getType(): Object [1]
  The object representing the type of the parameter.

5.4.8 Type

Description

The 'Type' type corresponds to the *DataType* meta-class and its sub-classes. Objects of this type shall be created when the *DataType* element does not have a suitable mapping specified.

Generalization

- NamedElement

Functions

- isStructure(): Boolean [1]
  Whether this type corresponds to a *StructuredDataType* element.

- isCollection(): Boolean [1]
  Whether this type corresponds to a *CollectionDataType* element.

- isEnum(): Boolean [1]
  Whether this type corresponds to an *EnumDataType* element.

- getMapping(): Mapping [1]
  The 'Mapping' object for this type chosen for the runtime.

- getParameters(): String [0..*]
  The names of *Member* elements if this type is a structure, undefined otherwise.

- getParameterType(parameterName: String): Type [1]
  The 'Type' of the parameter whose name equals 'parameterName', undefined if this type is not a structure.

- getItemType(): Type [1]
  The 'Type' of items in the collection if this type is a collection, undefined otherwise.

- getEnumLiterals(): Data [0..*]
  The 'Data' objects that correspond to *SimpleDataInstance* elements that are contained in this type, if this type corresponds to *EnumDataType*, undefined otherwise.

5.4.9 Value

Description

The 'Value' type corresponds to the *DataInstance* or the *DataUse* meta-class and their sub-classes. Objects of this type shall be created when the corresponding element does not have a suitable mapping specified.

For primitive 'Value' objects (that is, values that are not a structure nor a collection), the 'getValue' function shall return either a primitive value in a format specific to the implementation language or an instance of *SpecialValue*.

Generalization

- Element

Functions

- isStructure(): Boolean [1]
  Whether this value corresponds to a *StructuredDataInstance* element.

- isCollection(): Boolean [1]
  Whether this value corresponds to a *CollectionDataInstance* element.

- getMapping(): Mapping [1]
  The 'Mapping' object for this value chosen for runtime.

- getValue(): Object [1]
  The actual value of this 'Value' object if this object represents primitive data, undefined otherwise.

- getParameters(): String [0..*]
  The names of 'Member' elements of the 'Type' of this 'Value' if this value is a structure, undefined otherwise.

- getParameter(parameterName: String): Data [1]
  The 'Data' containing the type and value of the parameter whose name equals 'parameterName', undefined if this type is not a structure.

- getItems(): Data [0..*]
  The 'Data' objects contained in this 'Value' if the value is a collection, undefined otherwise.

5.4.10 SpecialValue

Description

The 'SpecialValue' type is an enumerated type and corresponds to sub-classes of the *SpecialValueUse* meta-class. The 'SpecialValue' object shall be used as a value for a 'Value' object.

Generalization

There is no generalization specified.

Functions

There are no functions specified.

5.4.11 Mapping

Description

The 'Mapping' type corresponds to the *DataResourceMapping*, *DataElementMapping* or *ParameterMapping* meta-classes. The value of the predefined *MappingName* annotation (when applied to a resource mapping) shall be used to identify if the 'Mapping' is applicable in a given context.

In case the mapping represents a data element mapping, the names of the contained parameter mappings shall match the ones returned by 'getParameters' function of the 'Value' or 'Type' object that contains this mapping.

Generalization

- Element

Functions

- getMappingName(): String [1]
  The name applied to the associated *DataResourceMapping* element via the predefined *MappingName* annotation.

- isResource(): Boolean [1]
  Whether this mapping corresponds to a *DataResourceMapping* element.

- isParameter(): Boolean [1]
  Whether this value corresponds to a *ParameterMapping* element.

- getUri(): String [1]
  The value of element specific property defining the URI of the mapping.

- getResource(): Mapping [1]
  The 'Mapping' corresponding to *DataResourceMapping* for this data element mapping, undefined if this is a parameter or resource mapping.

- getParameterMapping(parameterName: String): Mapping [1]
  The 'Mapping' of the parameter whose name equals 'parameterName', undefined if this is a parameter or resource mapping.

5.4.12 GateReference

Description

The 'GateReference' type corresponds to the *GateReference* meta-class.

Generalization

There is no generalization specified.

Functions

- getGate(): Element [1]
  The object representing the gate instance.

- getGateType(): NamedElement [1]
  The object representing the gate type.

- getGateTypeKind(): GateTypeKind [1]
  The object representing the gate type kind.

- getComponent(): Element [1]
  The object representing the component instance.

- getComponentType(): NamedElement [1]
  The object representing the component type.

- getComponentRole(): ComponentInstanceRole [1]
  The object representing the component role.

5.4.13 GateTypeKind

Description

The 'GateTypeKind' type is an enumerated type and corresponds to the *GateTypeKind* meta-class.

Generalization

There is no generalization specified.

Functions

There are no functions specified.

5.4.14 ComponentInstanceRole

Description

The 'ComponentInstanceRole' type is an enumerated type and corresponds to the *ComponentInstanceRole* meta-class.

Generalization

There is no generalization specified.

Functions

There are no functions specified.

5.4.15 Connection

Description

The 'Connection' type corresponds to the *Connection* meta-class.

Generalization

- Element

Functions

- getEndPoints(): GateReference [2]
  'GateReference' objects that correspond to *GateReference* elements contained in the corresponding *Connection* element.

5.4.16 Verdict

Description

The 'Verdict' type corresponds to the *SimpleDataInstance* meta-class whose type is the predefined type *Verdict*.

Generalization

- NamedElement

Functions

There are no functions specified.

5.4.17 StopException

Description

Objects of the 'StopException' type shall be returned (or thrown) by an implementation to indicate that the test execution should be stopped.

Generalization

There is no generalization specified.

Functions

- getMessage(): String [0..1]
  The informal reason for stopping.

- getVerdict(): Verdict [0..1]
  The final verdict of the test execution.

5.4.18 ValidationFailedException

Description

Objects of the 'ValidationFailedException' type shall be returned (or thrown) by an implementation of the 'Validator' interface to indicate that the match operation has failed.

Generalization

There is no generalization specified.

Functions

- getMessage(): String [0..1]
  The informal reason for the validation failure.

5.5 System Adapter

5.5.1 Overview

The system adapter component provides the mechanism for communicating with the System Under Test (SUT). This is usually a protocol stack implementation or an adapter for a user interface. An implementation may support either single or multiple concurrent connections and either message- or procedure-based communication (or both), depending on the testing needs.

Functions specific to message-based communication are specified in clauses 5.5.3 and 5.5.4. Functions specific to procedure-based communication are specified in clauses 5.5.5, 5.5.6 and 5.5.7. Clause 5.5.5 specifies the handling of a procedure call when the test executor is the caller. Clauses 5.5.6 and 5.5.7 specify the handling of a procedure call when the test executor is the callee (the server).

An implementation shall be able to handle receiving multiple calls (by the test executor) for the same incoming data (see clause 5.5.5). This implies that incoming data chunks shall be stored in a message queue in an encoded form to facilitate repeated decoding attempts.

NOTE: A message queue could be implemented as a first-in-first-out stack.

5.5.2 Configure connections

Signature

- configure(connections: Connection [1..*]): **void**

Description

Prepare the adapter for all 'connections' configured for an upcoming test description execution. This function shall be called before any test behaviour is executed.

NOTE: This function should be used for performing any required initialization of the component.

5.5.3 Send message

Signature

- send(message: Data, connection: Connection): **void**

Description

The implementation shall encode the 'message' data and send it to the SUT over the specified 'connection' using protocol- or adapter-specific means.

5.5.4 Receive message

Signature

- receive(expectedMessage: Data [0..1], connection: Connection): Data **throws** ValidationFailedException

Description

The execution of the function code shall block until data is available at the top of the incoming message queue for the specified 'connection'.

NOTE 1: The exact mechanism of blocking is language- and implementation-dependent. However, the blocking should be done in a manner that does not prevent the execution of concurrent code.

If the 'expectedMessage' is specified, then the implementation shall attempt to decode incoming data using the type of the specified 'expectedMessage' and match the data against the value of the 'expectedMessage'. An instance of 'Validator' may be used for matching. If the data matches the provided 'expectedMessage' then it is decoded and returned, otherwise the function shall throw a 'ValidationFailedException' specifying the result of the validation.

The decoded 'Data' object returned from the function shall use the same mapping mechanism as the 'message' argument. That is, if the 'expectedMessage' is a mapped object then the returned object shall be mapped object as well (see clause 4.2).

If the 'expectedMessage' is not specified, then the data shall be returned without validation and the data may be in encoded form. The returned data shall be passed on to the 'Test reporter' component by the test executor. The specific representation format for the data is implementation-specific.

The implementation shall only consider one (the oldest) chunk of incoming data at a time. If the received data matches the 'expectedMessage' or the 'expectedMessage' is undefined, then the message shall be considered handled and the data is removed from the incoming message queue.

NOTE 2: The implementation of the test executor should include a mechanism for interrupting the code that is blocked in waiting for input. The interruption mechanism is implementation-dependent.

5.5.5 Call procedure

Signature

- call(operation: Procedure, arguments: Argument [0..*], expectedReturn: Data [0..1], expectedException: Data [0..1], connection: Connection): Data **throws** ValidationFailedException

Description

The implementation shall encode the specified 'arguments' and call a remote procedure specified by the 'operation' over the specified 'connection' and block until a matching reply (either exception or response) is received.

Either 'expectedReturn' or 'expectedException' shall be provided, but not both (see clause 9.4.8 in [1]). The implementation shall use either of the provided data for decoding and matching and shall return a corresponding 'Data' object upon success.

Rules for processing and decoding incoming data and for function execution and return values described in clause 5.5.4 shall apply.

5.5.6 Receive procedure call

Signature

- receiveCall(operation: Procedure, expectedArguments: Argument [0..*], connection: Connection): Data [0..*] **throws** ValidationFailedException

Description

The execution of the function code shall block until the call to 'operation' is received with matching 'expectedArguments' from the specified 'connection'. The implementation shall use 'expectedArguments' for decoding and matching and shall return corresponding 'Data' objects upon success.

Rules for processing and decoding incoming data and for function execution and return value described in clause 5.5.4 shall apply.

5.5.7 Reply to procedure call

Signature

- replyCall(operation: Procedure, reply: Data [0..1], exception: Data [0..1], connection: Connection): **void**

Description

The implementation shall encode the 'reply' or 'exception' data and send it to the SUT as response to the 'operation' over the specified 'connection' using protocol- or adapter-specific means. Either 'reply' or 'exception' shall be provided, but not both (see clause 9.4.8 in [1]).

5.6 Validator

5.6.1 Overview

Component for providing data validation and verdict management.

5.6.2 Match data

Signature

- matches(expected: Data, actual: Data): **void throws** ValidationFailedException

Description

The implementation shall perform implementation-specific comparison of the 'actual' data received from the SUT and the 'expected' data specified in the test description. The function implementation shall return normally when the match succeeds. The failure to match shall be indicated by throwing a 'ValidationFailedException' specifying the result of the validation.

5.6.3 Set verdict

Signature

- setVerdict(verdict: Verdict): **void**

Description

The implementation shall update the current verdict of the test description execution according to provided 'verdict' and the rules described in clause 9.4.4 in [1].

5.6.4 Get verdict

Signature

- getVerdict(): Verdict

Description

The implementation shall return the current verdict of the test description execution.

5.7 Test Reporter

5.7.1 Overview

Component for providing environment-specific test logging functionality.

5.7.2 Comment

Signature

- comment(body: String): **void**

Description

The implementation shall log the comment associated with a model element. This function shall be called before the execution of the associated element.

5.7.3 Test objective reached

Signature

- testObjectiveReached(uri: String, description: String): **void**

Description

The implementation shall log the test objective associated with a behaviour element. This function shall be called after the execution of the associated behaviour.

5.7.4 Behaviour started

Signature

- behaviourStarted(id: String, kind: String, properties: Object [0..*]): **void**

Description

The implementation shall log the start of a behaviour execution. The 'kind' parameter shall hold the name of the behaviour element meta-class. The 'id' shall be a locally unique identifier of the behaviour. The 'properties' shall be specific to the behaviour and are implementation-specific.

5.7.5 Behaviour completed

Signature

- behaviourCompleted(id: String): **void**

Description

The implementation shall log the completion of a behaviour execution. The 'id' shall be a locally unique identifier of the behaviour and it shall match an 'id' from an earlier call to 'behaviourStarted'.

5.7.6 Runtime error

Signature

- runtimeError(error: Object): **void**

Description

The implementation shall log a runtime error. Runtime errors shall result in the termination of the test execution. The type of the 'error' object is implementation-specific and may include an operation stack trace or other information that facilitates locating the source of the error.

NOTE: It is not the responsibility of the logger to handle the errors in any way except logging them.

5.8 Predefined Functions

The predefined functions component provides implementations for all functions specified in clause 10.5 in [1]. The test executor shall resolve the implementation functions using mappings as explained in clause 4.2.

Annex A (informative): Technical Representation of the Runtime Interfaces

A technical representation of the TDL runtime interfaces is available in the TOP repository [i.1].

History

Document history		
V1.1.1	May 2025	MAP process MV 20250707: 2025-05-08 to 2025-07-07