The ETSI Test Description Language TDL and its Application

Andreas Ulrich[1], Sylvia Jell[1], Anjelika Votintseva[1] and Andres Kull[2]
[1]*Siemens AG, Corporate Technology, Munich, Germany*
[2]*Elvior, Tallinn, Estonia*
{andreas.ulrich, sylvia.jell, anjelika.votintseva}@siemens.com, andres.kull@elvior.com

Keywords: Model-based Testing, Domain-Specific Languages, Meta-modelling, Rail Application.

Abstract: The wide-scale introduction of model-based testing techniques in an industrial context faces many obstacles. One of the obstacles is the existing methodology gap between informally described test purposes and formally defined test descriptions used as the starting point for test automation. The provision of an explicit test description becomes increasingly essential when integrating complex, distributed systems and providing support for conformance and interoperability tests of such systems. The upcoming ETSI standard on the Test Definition Language (TDL) covers this gap. It allows describing scenarios on a higher abstraction level than programming or scripting languages. Furthermore, TDL can be used as an intermediate representation of tests generated from other sources, e.g. simulators, test case generators, or logs from previous test runs. TDL is based on a meta-modelling approach that expresses its abstract syntax. Deploying this design approach, individual concrete syntaxes of TDL can be designed for different application domains. The paper provides an overview of TDL and discusses its application on a use case from the rail domain.

1 INTRODUCTION

The trend towards a higher degree of system integration such as in case of cyber-physical systems or service-oriented architectures leads to a growing importance of integration testing of such distributed, concurrent, and real-time systems. Integration testing, which is a black-box testing approach, encompasses also conformance testing of a system against a standard and interoperability testing of two or more systems of different vendors.

Test automation is required for many phases in the quality assurance process such as regression tests, smoke tests, or acceptance tests. Automating tests is a software development activity that involves the production of test code/scripts. Moving towards a model-based approach in testing, there are some obstacles to overcome for the wide-scale introduction of model-based testing. One of these obstacles is the existing divergence between manually created testing artefacts (which must be understood and managed by humans) and the need for defining them formally to allow automation. As a consequence, there has been a methodology gap between the simple expression of a test purpose described frequently in prose and the complex coding of executable tests scripts. TDL (ETSI ES 203 119, 2013) covers that gap. Dedicated test descriptions will have a positive impact on the quality of the tests through better design and by making them easier to review by non-testing experts. This will improve the general productivity of test development. Moreover, it is also important to provide a fault-free transfer of specifications between tools participating in the development of tool-chains where manual interaction by a test engineer is often needed.

The language design of TDL centres on the meta-modelling approach for the abstract syntax. A number of concrete syntaxes can be defined that all map to the same meta-model to provide dedicated support for different application domains. Given that the elements of the meta-model are formally defined, TDL specifications can be analysed beforehand for consistency and internal correctness to ensure a high quality of the test descriptions. Being an abstract test specification language, different test implementations can be derived to reflect the particularities of concrete test environments, e.g. a distributed tester could be derived supporting asynchronous message-passing communication between tester and system under test (SUT) or a sequential tester that puts emphasis on validating real-time constraints between tester/SUT interactions.

The publicly funded European ARTEMIS

project MBAT (MBAT, 2013) focuses on delivering a methodology on combining Model-based Analysis and Testing in the development process for embedded software in the transportation domain (avionics, automotive, rail). In this context, the work on standardising TDL at ETSI is one piece of dissemination activities of some project partners to distribute essential results from this project. Though, it shall be noted that the actual standardisation activity on TDL is not performed by MBAT partners solely. A huge interest on TDL from industry could be observed in private communications.

The paper is structured as follows. After reviewing related work (Section 2), the paper provides an overview of TDL and its principal design approach (Section 3). The advantageous usage of this upcoming standard is demonstrated in Section 4 by a use case from the rail domain at Siemens developed within the MBAT project. Section 5 provides an overview about the used technology implementing TDL. Section 6 concludes the paper by providing directions of application and further research.

2 RELATED WORK

Testing complex systems becomes such a complex activity that it needs to follow a development process on its own. ETSI has defined such a test development process for its own purpose (Figure 1). While most phases of this process are covered with efficient methods, notably TPlan (ETSI ES 202 553, 2009) for the specification of test purposes and TTCN-3 (ETSI ES 201 873-1, 2013) for the specification of test cases, a method that provides support for the specification of test descriptions is lacking. This gap shall be closed with TDL.

In the testing domain, a variety of languages and frameworks are used to express tests at different levels of abstraction (from concrete, executable tests to high-level test descriptions), e.g. xUnit test framework (xUnit.net, 2013), CCDL (Razorcat, 2010), TTCN-3, UTP (OMG UTP, 2013). However, tests expressed at a higher abstraction level using existing technologies tend to get syntactically and semantically loose such that implementations of those approaches become heavily tool-dependent. Thus, different language features are supported by different tools and the execution of the same test specifications in different environments leads to different, possibly unexpected results. Approaches based on a concrete executable language, e.g. TTCN-3, deliver precise execution semantics. But their fixed syntax makes it hard to comprehend the tests without explicit knowledge of the language and the tool, in which the tests are represented.

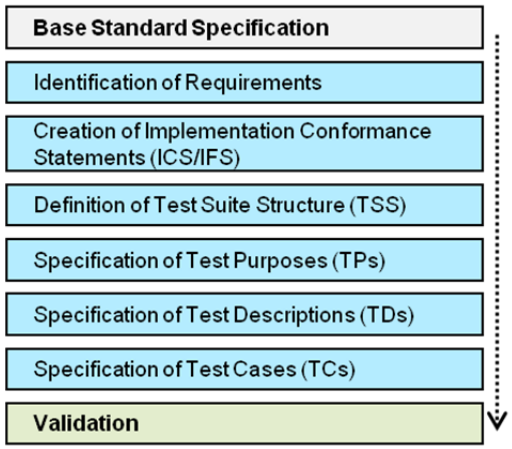

Figure 1: ETSI test development process (ETSI EG 203 130, 2013).

Some specific domains have initiated standardisation efforts to allow for exchangeable test specifications between tools such as the meta-model for test of avionics embedded systems (Guduvan et al, 2013) or the Automotive Test Exchange Format (ASAM ATX, 2012) and TestML (Grossmann, J., Müller, W., 2006) in the automotive domain. These languages typically cover only one abstraction level inheriting the challenges of very high-level specifications (for the Guduvan et al method) with very loose or even no semantics or suffering from the complexity of scripting languages (as ASAM ATX and TestML). Also these languages, being very effective within one domain and specific testing activities, cannot be easily reused in other settings.

The generic standards for different application domains like ISO/IEC 29119 Software Testing (ISO/IEC 29119, 2013) provide guidelines for the testing process and its techniques, which are complemented with notations for testing artefacts, but without strict semantics. The Precise UML (Bouquet et al, 2007) considers subsets of the UML (OMG UML, 2011) and OCL (OMG OCL, 2012) to define a behavioural model of the system under test (SUT). This approach combines graphical models with formal descriptions of the expected system behaviour as OCL expressions. Its drawback is that it covers only one single testing activity, the test case generation out of a SUT model.

New testing techniques that stem from agile development methods such as test-driven development (TDD) rely heavily on the specification of so-called ‘user stories’ represented as scenarios, i.e. interaction flows, between the system and a user of this

system. The principles of testing such systems are well stated by Cem Kaner (Kaner, 2003). Agile methods appear in today's industrial practice somehow contrary to model-based testing approaches that rely heavily on the creation of sufficiently detailed test models and consider the derivation of executable tests only as a simple generation step based on simplistic assumptions on test coverage.

The International Telecommunication Union's (ITU) Message Sequence Chart MSC was one of the first languages to specify scenarios (ITU, 2004). Later its features were included into the OMG's language UML 2 under the name Sequence Diagram. While conceptually well suited for the specification of scenarios, the many different usages of sequence diagrams result in quite different interpretations of their semantics (as discussed, for example, in Micskei, 2011) limiting its use as a universal, tool-independent test description language.

Another OMG's standard, the UML Testing Profile (UTP), covers most of the artefacts from test modelling. Being UML-based, it inherits advantages and drawbacks of the UML. In particular, its loose semantics relies heavily on tool-specific solutions. This makes test models hardly transferrable. Although having a wide scope, UTP still does not cover some aspects important for testing activities. For example, the MARTE UML profile is additionally needed to capture timing aspects. In contrast to UTP, TDL focuses on test descriptions for the real-time interactions between testers and the SUT with a formal semantics. The common meta-model for test in TDL fosters reuse of tests and tools while various concrete syntaxes can be used aligned to the common abstract syntax.

A forerunner to TDL is the approach described in (Ulrich et al, 2010). It discusses the *ScenTest* tool for scenario-based testing that supports the generation of concurrent tests from a special sub-class of UML sequence diagrams.

3 THE TEST DESCRIPTION LANGUAGE

3.1 General Approach

TDL bridges the gap between high-level test purpose specifications and executable test cases. It provides a generic language for the formal specification of test descriptions which can be used as the basis for the implementation of concrete tests on a given test execution platform or simply for the visualisation of test scenarios for different stakeholders. TDL is designed to support the black-box test of distributed, concurrent real-time systems.

TDL supports a scenario-based approach using modelling techniques from model-based testing and UTP. Test scenarios are described at a higher abstraction level than what is possible with scripting languages such as TTCN-3. It is indifferent on the basic communication mechanism used between tester and SUT being message-based, procedural or communication-based on shared variables or other types of interfaces. Furthermore, TDL can be used as an intermediate representation of tests generated from other sources, e.g. simulators, test case generators, or logs from previous test runs.

TDL is designed around a meta-model approach based on the OMG's meta-object facility MOF (OMG MOF, 2013) to describe its abstract syntax. This way, it is able to support different concrete syntaxes, also with a different feature set according to the needs of different application domains.

While the TDL meta-model is based on a well-defined underlying formal semantics, it is possible to provide supportive tools for correctness analysis of (manually) specified test descriptions, the construction of test cases according to a chosen fault model, the visualisation of test run results, or the exchange of test descriptions between different tools. The formal semantics prevents misinterpretation of the artefact specifications between different tools. The approach is driven by industry to foster the benefits of model-based software engineering in the test process.

ETSI has set up Special Task Force (STF) to standardise TDL. By writing the current paper the STF has worked out a first stable draft of a meta-model description of TDL (ETSI ES 203 119, 2013). Publication of the final ETSI standard is expected in early 2014.

3.2 TDL Design Principles

TDL makes a clear distinction between concrete syntax that is adjustable to different application domains and a common abstract syntax, which a concrete syntax can be mapped to. The definition of an abstract syntax for a TDL specification plays the key role in offering interchangeability and unambiguous semantics of test descriptions. It is defined in the TDL standard in terms of a MOF meta-model.

A TDL specification consists of the following major parts:

- A *test configuration* consisting of two or more tester and SUT components and their

interconnections reflecting the test setup;
- A set of *test descriptions*, each of them describing one test scenario based on *interactions* between the components on a given test configuration and abstract tester *actions* plus *behavioural operations* such as sequential, alternative, parallel, iterative behaviour etc.;
- A set of typed *data instances* used in the interactions of the test descriptions.

Using these major ingredients, a TDL specification is abstract in the sense that it does not detail how a test description is implemented. More specifically:
- Interactions between tester and SUT components of a test configuration are considered to be atomic and not detailed further. For example, an interaction can represent a message exchange, a function/procedure call, or a shared variable access.
- All behavioural elements of a test description are totally ordered, unless specified otherwise. That is, there is an implicit synchronization mechanism assumed to exist between the components of a test configuration. A TDL implementation must ensure that the specified execution order of interactions is obeyed.
- The behaviour of a test description represents the expected, foreseen behaviour of a test scenario assuming an implicit test verdict mechanism, if not specified otherwise.
- The data exchanged in interactions of a test description or used in parameters of actions are represented as name tuples without further details of their underlying semantics, which is implementation specific.

A TDL specification represents a closed tester/SUT system. That is, each interaction has a sender and a receiver component that is contained within the given test configuration a test description runs on.

All behavioural elements of a test description, e.g. interactions, actions, timer, time, and verdict operations are totally ordered, unless specified otherwise, even if they occur at different components of a test configuration. In particular, synchronization between components, e.g. by inserting special sync messages, is assumed to be implicit. It is up to the concrete implementation of the test description to ensure the correct ordering during test execution.

Time in TDL is considered to be global and progresses in discrete quantities of arbitrary granularity. Progress in time is expressed as a monotonically increasing function. Time starts with the execution of a test description and is reset at the end of execution.

TDL offers an implicit verdict mechanism. Specified behaviour of a test description is assumed to represent correct behaviour (verdict *pass*). Any deviation from the specified behaviour observed during a test run is considered a failure (verdict *fail*). To overwrite this default rule, one can set verdicts explicitly in a test description if needed. The standard verdict values *pass*, *inconclusive*, *fail* are predefined. In addition further values can be defined by a user. However, there is no assumption about verdict arbitration, which is left to the concrete realisation of a test description.

3.3 TDL Meta-Model Overview

The meta-model is structured in packages, which are briefly described in the following (based on the current stable draft TDL specification).

The *Foundation* package covers the fundamental concepts needed to express the structure and contents of a TDL specification and defines additional elements of a test description such as test objectives, annotations, and comments.

The *Test Architecture* package describes all elements needed to define a test configuration consisting of tester and SUT components, gates, and their interconnections. Connections link two or more gates of different components. A component can act either in the role of a *tester* or a *SUT*.

The *Data* package defines the elements needed to express data sets and data instances used in test descriptions. TDL does not feature a complete data type system. Instead, it relies on parameterizable data instances, which serve as surrogates for concrete data types and values outside of TDL.

The *Test Behaviour* package defines all elements needed to describe the structure and behaviour of a test description. It is the most elaborated package to accommodate various behaviour kinds and provides the definitions for interactions and actions. Moreover, it defines a way to link a test description or any of the behavioural elements it contains to *test objectives* reflecting test purposes or other forms of system requirements to be validated.

The *Time* package defines all elements to express time and operations over time. There are two different concepts in TDL to operate on time:
- A descriptive way to express time in terms of *wait* (for the tester) and *quiescence* (for the SUT) operations;
- An operational way in terms of timers and operations over timers *start*, *stop*, *timeout*.

Additionally, time constraints can be specified between behavioural elements.

4 CASE STUDY

The advantageous usage of TDL as a notation and approach to specify test descriptions is demonstrated in a use case from the rail automation domain developed within the MBAT project. The use case features a track warrant control system for regional railroads and a train-borne protection system. Requirements for automatic train operation are used to derive test scenarios.

Test scenarios are closely linked to an instance of a realistic physics simulation of a typical train system, which consists of different applications performing different tasks:

- Physics engine (for simulation purposes),
- Interlocking system,
- Automatic Train Protection (ATP),
- Driver Machine Interface (DMI).

All participating applications communicate via a shared information broker. The topology of the tracks is stored in a common database accessible by the interlocking system.

A tester component links itself to the information broker. This way, it is able to receive status information from the telegrams sent between the applications (observations mode) and to inject its own telegrams (stimuli mode) to the train system.

A basic requirement is the collision-free operation of the ATP application in integration with the interlocking system featuring:

- Route setting,
- Protection against overlapping routes, and
- Prevention of movement authorities being overridden.

In the following example a basic test configuration of two components is considered, the *train system* (comprising ATP and interlocking) in the role of the SUT and the *operator* in the role of the tester that sets the route on the tracks and operates the train (Figure 2). The endpoints of the connection between SUT and tester are denoted as gates *g*.

A train starts at a given *way point* in a given track layout and initially requests the necessary switch position on the track and its speed. Status messages from the train system (`ATPStatus`) are sent in regular intervals (every 50msec) and collected by the tester to check the correctness of parameters like train position (way point) and speed.

The test scenario checks that the train stops at a signal showing ‘Stop’ and does not proceed for a given period of time until a request to change the signal aspect to ‘Proceed’ has been honoured. The purpose of this test is given as a test objective linked to the test scenario (Figure 3). This test scenario—along with other test scenarios—constitutes the test model.

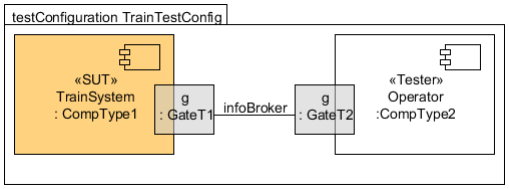

Figure 2: Test configuration.

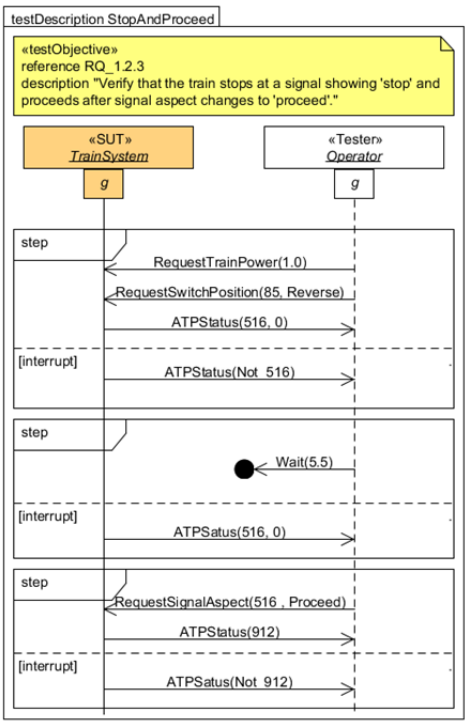

Figure 3: Test description as a scenario (concrete syntax).

A test scenario reflects the expected behaviour of the SUT that shall be observed by the tester during test execution. If it is observed, the test verdict will be set to pass implicitly. Otherwise it will be different, i.e. inconclusive or fail depending on the concrete test implementation. The TDL behavioural operation interrupt is used to discard status messages from the SUT that are correct per se, but do not contribute to the test objective.

The example of a test description considered in this section is composed of a series of test steps: First the system is triggered by a request to set the train power to a specific power fraction `RequestTrainPower(1.0)` and to set the switch position at an upcoming way point `RequestSwitchPosition(85, Reverse)`. This enables the train to proceed until way point 516 that is guarded with a

'Stop' signal. The expected speed parameter at this way point is therefore zero in the status message `ATPStatus(516, 0)`. Status messages with a way point parameter different from 516 are discarded `ATPStatus(Not_516)` until the expected message for waypoint 516 with speed 0 is received.

In the next step, the system is not triggered for a specific time period `Wait(5.5)`. The train is expected to remain at way point 516 without any further movement. That is, only status message `ATPStatus(516, 0)` must be received. Status messages with these parameter values are therefore expected and can be filtered whereas status messages with other way point or speed parameter values would lead to a *fail* verdict.

In the final test step, the system is triggered by a request to set the signal aspect to 'Proceed' `RequestSignalAspect(516, Proceed)`. This signal change enables the train to proceed to way point 912, which is reached when the status message `ATPStatus(912)` is observed. There the test ends. Status messages at other way points are discarded using the *interrupt* operation with message `ATPStatus (Not_912)`.

TDL makes a clear distinction between an adjustable concrete syntax and a common abstract syntax (an instance of the TDL meta-model), which a concrete syntax is mapped to. The test description discussed in this section can be represented in the TDL abstract syntax as shown on Figure 4.

A concrete syntax can have a textual or a graphical form or a combination thereof. For our example the representation of the test description can either be graphically expressed by means of a TDL sequence diagram (Figure 3) or in pure text format (see Listing 2 below).

Each element of the TDL sequence diagram, which can be considered as a dialect of a UML sequence diagram, is mapped to its corresponding meta-model element which carries a clear and unambiguous meaning. The same holds for the textual representation. This way, TDL is flexible to cope with different representation needs of test descriptions that stem from different application domains.

5 TOOL SUPPORT FOR TDL

The meta-model for TDL is provided as a MOF meta-model. This approach allows for the selection of a range of supportive tools that ease the implementation of the TDL approach for a number of tasks such as:

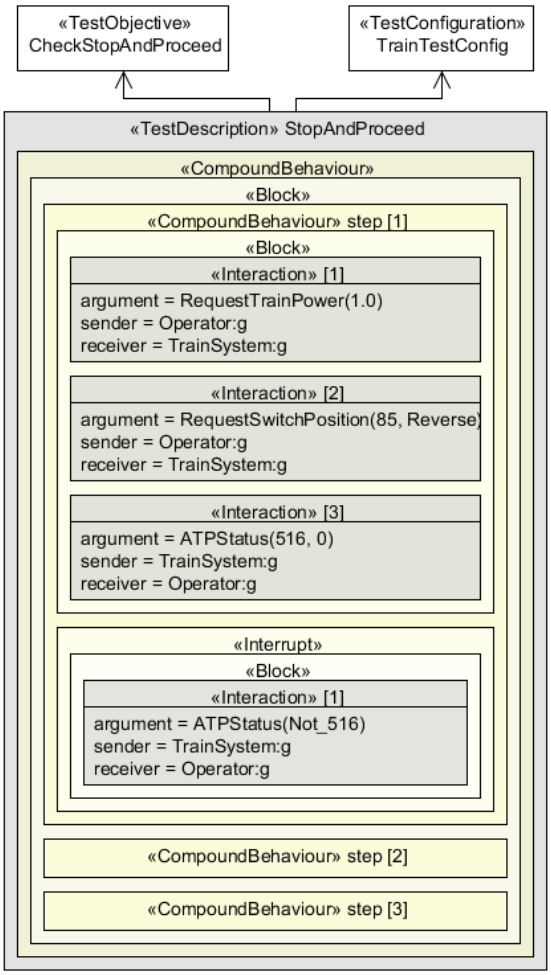

Figure 4: Test description as instantiation of the TDL meta-model (abstract syntax).

- Derivation of a TDL concrete syntax along with a supporting editor,
- Validation against the TDL meta-model,
- Generation of executable test code.

We deploy Eclipse Modelling Framework (EMF, 2013) that provides a code generation facility to generate Java implementation classes from the meta-model. It also provides a model editor for creating, modifying, saving, and loading model elements.

Furthermore, an Eclipse plug-in EMFText tightly integrated with EMF (EMFText, 2013) is used to support the definition of a textual syntax corresponding to the TDL meta-model. EMFText has the following advantages:

- It comes with a simple syntax specification language (the Concrete Syntax Specification Language CS) based on the Extended Backus-Naur Form (EBNF).
- It offers options to derive an initial concrete syntax

that can serve as a starting point for a domain-specific language based on the given meta-model.
- It provides a comprehensive syntax analysis for CS specifications.
- It generates tool support from CS language specifications. For example, a feature-rich, language specific editor integrated in Eclipse can be derived.

Listing 1 shows a part of the concrete syntax definition based on the TDL meta-model. It was automatically derived from the model as a Human-Usable Textual Notation (OMG HUTN, 2004) and then incrementally refined to our needs and language design requests for the considered case study.

```
SYNTAXDEF tdl
FOR <http://www.etsi.org/spec/TDL/20130606>
START TDLSpecification

RULES {
    TDLSpecification ::= "TDLSpecification" "{"
                         ("name" name['"','"']) ?
                         comment* annotation* content* "}";
    TestDescription  ::= "TestDescription" "{"
                         ("name" name['"','"']) ?
                         comment* annotation*
                         ("owningPackage" owningPackage[]) ?
                         ("formalParameter" formalParameter[])*
                         "testConfiguration" testConfiguration[]
                         ("testObjective" testObjective[])*
                         behaviour timeConstraint* "}";

```

Listing 1: Part of concrete syntax for TDL.

Starting with a default syntax that is derived automatically reduces the efforts to specify the necessary syntax rules significantly. In addition, EMFText also provides decent support during refinement, for model evolvement as well as for syntax alignment. A number of customization techniques and options are available to adjust the language specific plug-ins generated by EMFText to user specific needs, like preserving manual changes in generated code or adaptable tokens and resolving of references.

The textual test description as shown in Listing 2 was created using the editor generated by means of EMFText. It provides features similar to the built-in Eclipse Java editor such as code completion, syntax highlighting, code folding, text hovers, and others.
Using EMFText it was possible, to successfully validate the designed TDL meta-model by experimenting with its various features. Moreover this case study serves as a starting point for the design of textual test description languages customised to different application domains as demonstrated here with its application to system integration testing in the rail domain.

The standardisation of TDL currently provides only the specification of the meta-model (as a Papyrus UML project). Follow-up activities in 2014 will consider the design of a standardized graphical syntax likely similar to the one shown in Figure 2.

```
TestDescription(name "TestWithInterrupt"
                testConfiguration MinimalConfig
                testObjective Check_Stop_and_Proceed
    CompoundBehaviour(
        Block(
            CompoundBehaviour(
                Interrupt(
                    Block(
                        Interaction(argument ATPStatus(Not_516)
                                    sender ts_gate1 receiver op_gate1)))
                Block(
                    Interaction(argument RequestSwitchPosition(85, Reverse)
                                sender op_gate1 receiver ts_gate1),
                    Interaction(argument RequestTrainPower(1.0)
                                sender op_gate1 receiver ts_gate1),
                    Interaction(argument ATPStatus(516, 0)
                                sender ts_gate1 receiver op_gate1))),
            CompoundBehaviour(
                Interrupt(
                    Block(
                        Interaction(argument ATPStatus(516, 0)
                                    sender ts_gate1 receiver op_gate1)))
                Block(
                    Wait(gateInstance op_gate1
                         period Time(value "5.5"  dimension sec)))),
            CompoundBehaviour(
                Interrupt(
                    Block(
                        Interaction(argument ATPStatus(Not_912)
                                    sender ts_gate1 receiver op_gate1)))
                Block(
                    Interaction(argument RequestSignalAspect(516 , Proceed)
                                sender op_gate1 receiver ts_gate1),
                    Interaction(argument ATPStatus(912)
                                sender ts_gate1 receiver op_gate1)))
        )
    )
)
```

Listing 2: Test description in concrete textual TDL syntax.

Conceptually it will be based on OMG's Diagram Definition approach (OMG DD, 2012).

6 CONCLUSIONS

TDL is an upcoming ETSI standard for the specification of test descriptions supporting the design of black-box tests for testing a wide range of systems. We assume that the following application areas will benefit from the proposed homogeneous, standardised approach of test design with TDL:

- Model-based design of test descriptions derived from the given test objectives, e.g. test purpose specifications (ETSI process), user stories (TDD) or other sources;
- Representation of test descriptions obtained from other sources, e.g. generated tests (output from test generation tools), system simulators, test execution traces from previous test runs.

Being a new notation, there is naturally little tool support that is ready to use. However basing the TDL design on a meta-modelling approach within the Eclipse development framework unlocks the potentials from many development tools of this well-established platform, which can be quickly turned into assets such as the creation of a TDL editor as demonstrated in this paper. Future tool support will concentrate on the design of static analysers of TDL specifications and test code generators.

While the semantics of TDL meta-model elements is currently specified as free text in the ETSI

standard, there are ongoing efforts to define a semantics based on timed automata.

The current version of TDL is designed specifically for the purpose of representing test scenarios. However it can be extended to serve also as an input language to test generators. The necessary amendments to TDL require the support to the design of higher-order TDL specifications that feature non-deterministic choices over data and behaviour to aid the generation of test descriptions according to chosen coverage criteria. This extension is the focus of future research.

ACKNOWLEDGEMENTS

The authors wish to express their gratitude to all people at ETSI and outside involved in the TDL standardization effort for their valuable input and constructive discussion. Moreover the authors are indebted to their Siemens colleagues for providing the rail case study.

This work received partial funding from the ARTEMIS Joint Undertaking, grant agreement no. 269335 (MBAT, 2013).

REFERENCES

ASAM ATX, 2012. *Release Presentation: ASAM AE ATX V1.0.0, Automotive Test Exchange Format*. http://www.asam.net/nc/home/asam-standards.html.

Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., Utting, M., 2007. A subset of precise UML for model-based testing. In *Proc. of the 3rd Int. workshop on Advances in model-based testing* (A-MOST '07). ACM, New York, NY, USA, 95-104.

EMF, 2013. *Eclipse Modeling Framework Project (EMF)*. http://www.eclipse.org/modeling/emf/

EMFText, 2013. *EMFText, concrete syntax mapper*. http://www.emftext.org/

ETSI EG 203 130 Ver. 1.1.1: *Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Methodology for standardized test specification development*. 2013-04-09.

ETSI ES 201 873-1 Ver. 4.5.1: *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3*; Part 1. 2013-04-30.

ETSI ES 202 553 Ver. 1.2.2: *Methods for Testing and Specification (MTS);TPlan: A notation for expressing Test Purposes.* 2009-06-02.

ETSI ES 203 119 (stable draft): *Methods for Testing and Specification (MTS); The Test Description Language (TDL)*, 2013-09-25.

Grossmann, J., Müller, W., 2006. A Formal Behavioral Semantics for TestML. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (ISOLA '06). IEEE Computer Society, Washington, DC, USA, 441-448.

Guduvan, A., Waeselynck, H., Wiels, V., Durrieu, G., Fusero, Y., Schieber, M., 2013. A Meta-model for Tests of Avionics Embedded Systems. In *MODELSWARD'13, 2nd International Conference on Model-Driven Engineering and Software Development.* SCITEPRESS Digital Library.

International Telecommunication Union, 2004. Recommendation Z.120: Message sequence chart (MSC). http://www.itu.int/rec/T-REC-Z.120.

ISO/IEC/IEEE 29119, 2013. *Software and systems engineering – Software testing* (5 parts). http://softwaretestingstandard.org/

Kaner, C., 2003. On Scenario Testing. In *STQE Magazine*. September/October 2003, 16-22.

MBAT, 2013. Combined Model-based Analysis and Testing, an ARTEMIS project, https://www.mbat-artemis.eu/home/.

Micskei, Z., Waeselynck, H., 2011. The many meanings of UML 2 Sequence Diagrams: a survey. In *Software and Systems Modeling*, Springer, Vol. 10, 489-514.

OMG DD, 2012. *Diagram Definition (DD) V1.0*, formal/12-07-01.

OMG HUTN, 2004. *Human-Usable Textual Notation (HUTN) Specification V1.0*, formal/04-08-01.

OMG MOF, 2013. *OMG Meta Object Facility (MOF) Core Specification V2.4.1*, formal/2013-06-01.

OMG OCL, 2012. *OMG Object Constraint Language (OCL) V2.3.1,* formal/2012-01-01.

OMG UML, 2011. *Unified Modeling Language (UML) V2.4.1, Superstructure specification*, formal/2011-08-06.

OMG UTP, 2013. *UML Testing Profile (UTP) V1.2*, formal/2013-04-03.

Razorcat, 2010. *CCDL Whitepaper*. Razorcat Technical Report, 07 December. www.razorcat.com.

Ulrich, A., Alikacem, E.-H., Hallal, H., Boroday, S., 2010. From scenarios to test implementations via Promela. In: *Testing Software and Systems (ICTSS 2010)*, Springer LNCS 6435, pp. 236–249.

xUnit.net, 2013. *Unit Testing Framework for C# and .NET*. http://xunit.codeplex.com/.