# Contents

# Chapter 1

# Model-based X-in-the-Loop Testing

Software driven Electronic Control Units (ECUs) are increasingly adopted in the creation of more secure, comfortable, and flexible systems. Unlike conventional software applications, ECUs may be affected directly by the physical environment they operate in. Whereas for simple software applications testing with specified inputs and checking whether the outputs match the expectations is sufficient, such an approach is no longer adequate for the testing of ECUs. Because of the close interrelation with the physical environment, proper testing of ECUs must directly consider the feedback from the environment, as well as the feedback from the System Under Test (SUT) to generate the test input data and calculate the test verdict. Such simulation and testing approaches dedicated to verify feedback control systems are normally realized using so called closed loop architectures, where the part of the feedback control system that is being verified is said to be "in the loop". During the respective stages in the development lifecycle of ECUs, models, software, and hardware are commonly placed in the loop for verification and testing purposes. These different representations of an ECU's functionality generally depend on different proprietary technologies.

Currently, there is no methodology for closed loop testing that allows the technology independent specification and systematic reuse of testing artifacts, such as tests, environment models, etc. In this chapter, we propose such a methodology, namely "X-in-the-Loop testing", which encompasses the testing activities and the involved artifacts during the different development stages. This

work is based on the results from the TEMEA project[1]. Our approach starts with a systematic differentiation of the individual artifacts and architectural elements that are involved in "X-in-the-loop" testing. Apart from the SUT and the tests, the environment models, in particular, must be considered as a subject of systematic design, development, and reuse. Similar to test cases, they shall be designed to be independent from test platform specific functionalities and thus be reusable on different testing levels.

This chapter introduces a generic approach for the specification of reusable "X-in-the-loop" tests on the basis of established modeling and testing technologies. Environment modeling in our context will be based on Simulink® (The MathWorks 2010b). For the specification and realization of the tests, we propose the use of TTCN-3 embedded, an extended version of the standardized test specification language TTCN-3 (ETSI 2009b; ETSI 2009a). The chapter starts with a short motivation in Section 1.1 and provides some generic information about artifact reuse in Section 1.2. In Section 1.3 we describe an overall test architecture for reusable closed loop tests. Section 1.4 introduces TTCN-3 embedded, Section 1.5 provides examples on how vertical and horizontal reuse can be applied to test artifacts, and Section 1.6 presents reuse as a test quality issue. Section 1.7 concludes the chapter.

## 1.1 Motivation

An ECU usually interacts directly with its environment, using sensors and actuators the case of a physical environment, and network systems in the case of an environment that consists of other ECUs. To be able to run and test such systems, the feedback from the environment is essential and usually must be simulated. Normally, such a simulation is defined by so called environment models that are directly linked with either the ECU itself during Hardware-in-the-Loop (HiL) tests, the software of the ECU during Software-in-the-Loop (SiL) tests, or in the case of Model-in-the-Loop (MiL) tests, with an executable model of the ECU's software. Apart from the technical differences that are caused by the different execution objects (an ECU, the ECU's software or a model of it), the three scenarios are based on a common architecture, a so called closed loop architecture.

[1]The project "**Te**sting Specification Technology and **M**ethodology for **E**mbedded Real-Time Systems in **A**utomobiles" (TEMEA) (TEMEA 2010) is co-financed by the European Union. The funds are originated from the European Regional Development Fund (ERDF).

Following this approach, a test architecture can be structurally defined by generic environment models and specific functionality-related test stimuli that are applied to the closed loop. The environment model and the SUT constitute a self-contained functional entity, which is executable without applying any test stimuli. To accommodate such an architecture, test scenarios in this context apply a systematic interference with the intention to disrupt the functionality of the SUT and the environment model. The specification of such test scenarios has to consider certain architectural requirements. Furthermore, we recommend a test specification and execution language which is expressive enough to deal with reactive control systems. At least we need reactive stimulus components for the generation of test input signals that depend on the SUT's outcome, evaluation capabilities for the analysis of the SUT's reaction, and a verdict setting mechanism to propagate the test results.

Because of the application of model-based software development strategies in the automotive domain, the design and development of reusable models is well known and belongs to the state of the art (Conrad and Dörr 2006; Fey, Kleinwechter, Leicher, and Müller 2007; Harrison, Gilbert, Lauzon, Jeffrey, Lalancette, Lestage, and Morin 2009). These approved development strategies and methods can be directly ported to develop highly reusable environment models for testing and simulation and thus provide a basis for a generic test architecture that is dedicated to the reuse of test artifacts. Meanwhile, there are a number of methods and tools available for the specification and realization of environment models, such as Simulink and Modelica (Modelica Association 2010). Simulink in particular is supported by various testing tools and is already well-established in the automotive industry. Both Modelica and Simulink provide a solid technical basis for the realization of environment models, which can be used either as self-contained simulation nodes, or, in combination with other simulation tools, as part of a co-simulation environment.

## 1.2 Reusability Pattern for Testing Artifacts

Software reuse (Karlsson 1995) has been a important topic in software engineering in both research and industry for quite a while now. It is gaining a new momentum with emerging research fields, such as software evolution. Reuse of existing solutions for complex problems re-

duces the effort to duplicate these solutions and the possibility of duplicating errors encountered while duplicating the solutions. The reuse of test specifications, however, has only recently been actively investigated. Notably, the reusability of TTCN-3 tests has been studied in detail as a part of the Tests & Testing Methodologies with Advanced Languages (TT-Medal) (TT-Medal 2010) project, but the issue has been investigated also in (Karinsalo and Abrahamsson 2004; Mäki-Asiala 2004). Reuse has been studied on three levels within TT-Medal—TTCN-3 language level (Mäki-Asiala, Kärki, and Vouffo 2006; Mäki-Asiala, Mäntyniemi, Kärki, and Lehtonen 2005), test process level (Mäntyniemi, Mäki-Asiala, Karinsalo, and Kärki 2005), and test system level (Kärki, Karinsalo, Pulkkinen, Mäki-Asiala, Mäntyniemi, and Vouffo 2005). In the following, focus will be mainly on reusability on the TTCN-3 language level, and how the identified concepts transfer to TTCN-3 embedded.

Means for the development of reusable assets generally include establishing and maintaining a good and consistent structure of the assets, the definition of and adherence to standards, norms, and conventions. It is furthermore necessary to establish well-defined interfaces and to decouple the assets from the environment. In order to make the reusable assets also usable, detailed documentation is necessary, but also the proper management of the reusable assets, which involves collection and classification for easy locating and retrieval. Additionally, the desired granularity of reuse has to be established upfront, so that focus can be put on a particular level of reuse, for example on a component level, or on a function level.

On the other hand, there are the three viewpoints on test reuse as identified in (Kärki, Karinsalo, Pulkkinen, Mäki-Asiala, Mäntyniemi, and Vouffo 2005; Mäki-Asiala 2004; Mäki-Asiala, Mäntyniemi, Kärki, and Lehtonen 2005):

**Vertical** - which is concerned with the reuse between testing levels or types (e.g., component and integration testing, functional and performance testing);

**Horizontal** - which is concerned with the reuse between products in the same domain or family (e.g., standardized test suites, tests for product families, or tests for product lines);

**Historical** - which is concerned with the reuse between product generations (e.g., regression testing).

While the horizontal and historical viewpoints have been long recognized in software reuse, vertical reuse is predominantly applicable only to test assets. "X-in-the-Loop" testing is perhaps closest to the vertical viewpoint on reuse, although it cannot be entirely mapped to any of the viewpoints, since it also covers the horizontal and historical viewpoints as described in the subsequent sections.

Nevertheless, the reuse of real-time test assets can be problematic, since, similar to real-time software, context-specific time and synchronization constraints are often embedded in the reusable entities. Close relations between the base functionality and the real-time constraints often cause interdependencies that reduce the reusability potential. Thus, emphasis shall be placed on context-independent design from the onset of development, identifying possible unwanted dependencies on the desired level of abstraction and trying to avoid them whenever a feasible alternative can be used instead. This approach to reuse, referred to as "revolutionary" (Mäntyniemi, Mäki-Asiala, Karinsalo, and Kärki 2005), or "reuse by design" involves more upfront planning and a sweeping transformation on the organizational level, which requires significant experience in reuse. The main benefit is that multiple implementations are not necessary. In contrast, the "evolutionary" (Mäntyniemi, Mäki-Asiala, Karinsalo, and Kärki 2005) approach to reuse involves a gradual transition towards reusable assets, throughout the development, by means of adaptation and refactoring to suit reusability needs as they emerge. The evolutionary approach requires less upfront investment in and knowledge of reuse, and involves less associated risks, but in turn may also yield less benefits. Knowledge is accumulated during the development process and the reusability potential is identified on site. Such an approach is better suited for vertical reuse in systems where requirements are still changing often. The basic principle of this approach is to develop usable assets first, then turn them into reusable ones. In the context of "X-in-the-Loop" testing, the aim is to establish reusability as a design principle, by providing a framework, an architecture, and support at the language level. Consequently, a revolutionary approach to the development of the test assets is necessary.

## 1.3 A Generic Closed Loop Architecture for Testing

A closed loop architecture describes a feedback control system. In contrast to open loop architectures and simple feedforward controls, models, especially the environment model, form a central part of the architecture. The input data for the execution object is calculated directly by the environment model, which itself is influenced by the output of the execution object. Thus, both execution object and environment model form a self contained entity. In terms of testing, closed loop architectures are more difficult to handle than open loop architectures. Instead of defining a set of input data and assessing the related output data, tests in a closed loop scenario have to be integrated with the environment model. Usually, neither environment modeling nor the integration with the test system and the individual tests are carried out systematically. Thus, it would be rather difficult to properly define and describe test cases, to manage them and even to reuse them partially.

In contrast, we propose a systematic approach on how to design reusable environment models and test cases. We think of an environment model as defining a generic test scenario. The individual test cases are defined as perturbations of the closed loop runs in a controlled manner. A test case in this sense would be defined as a generic environment model (basic test scenario) together with the description of its intended perturbation. Depending on the underlying test strategies and intentions, the relevant perturbations could be designed based on functional requirements (as blackbox tests, for example) or derived by manipulating standard inputs stochastically to test robustness. They could also be determined by analyzing limit values or checking interfaces. In all cases, we obtain a well defined setting for closed loop test specification.

To achieve better maintenance and reusability we rely on two basic principles. First, we introduce a proper architecture, which allows the reuse of parts of an environment model. Second, we propose a standard test specification language, which is used to model the input perturbations and assessments, and allows the reuse of parts of the test specification in other test environments.

### 1.3.1 A Generic Architecture for the Environment Model

For the description of the environment model, we propose a three layer architecture, consisting of a computation layer, a pre- and postprocessing layer, and a mapping layer (Figure 1.3.1). The computation layer contains the components responsible for the assessment and the modeling of the feedback reaction. The pre- and postprocessing layer contains the preprocessing and postprocessing components, which are responsible for signal transformation. The mapping layer contains the so called mapping adapters, that provide the direct connectivity between the SUT and the environment model.

**Please place Figure Fig01-01.pdf here. Caption:** "Closed loop architecture"

Based on this architecture, we will present a method for the testing of controls with feedback. In what follows we present a more detailed description of the individual entities of the environment model. We start with the computation layer and after which we provide a description of the pre- and postprocessing layer and the mapping layer.

#### The Computation Layer

The essential part in closed loop test systems is the calculation of the environmental reaction, that is, for each simulation step, the input values for the execution object are computed by means of the output data and the abstract environment model. Moreover, special components are dedicated to testing. The perturbation component is responsible for the production of test stimuli by means of test perturbations and the assessment component is responsible for the verdict setting.

- Abstract Environment Model

  The abstract environment model is a generic model that simulates the environment where the ECU operates in. It is designed for reuse in multiple test scenarios. The level of technical abstraction is the same as that of the test cases we intend to specify. In a model-based approach, such a model can be defined using Simulink™, Stateflow™ (The MathWorks 2010c), or Modelica. Hardware entities that are connected to the test environment may be used as well. The character of the abstract environment model directly depends on the chosen test level.

For a module test, such a test model can be designed using Simulink. In an integration test, performed with CANoe (Vector Informatics 2010) for example, the model can be replaced by a Communication Application Programming Language (CAPL) Node. And last but not least, in a HiL scenario this virtual node can be replaced by other ECUs that interoperate with the SUT via Controller Area Network (CAN) bus communication.

- Test Perturbation

  As sketched above a test specification is defined as a perturbation of the closed loop run. Initially, we could consider to the following scenario: We break the next input computation from the prepared output data. The perturbation will then be defined by a construction prescription for the next input data, that is, we partly open the closed loop and describe the next input data constructively. For the description of continuous data inputs, we will use the extension to the Testing and Test Control Notation Version 3—TTCN-3 embedded.

- Assessment Component

  In order to conduct an assessment, different concepts are required. The assessment component must observe and assess the outcome and set verdicts, but it should not interfere (apart from possible interrupts). TTCN-3 provides just the proper concepts for the definition of the assessments. Furthermore, by relying on a standard, assessment specifications can be reused in other environments for conformance testing. In order to provide proper assessments, which work on data streams, we will use TTCN-3 embedded.

Using the various access mechanisms provided in TTCN-3 embedded, we can start with the signal flow we would like to perturbate. With the help of the more advanced constructs, we can then manipulate this signal in various ways, such as adding an offset, scaling it with a factor, etc.

### The Pre- and Postprocessing Layer and the Mapping Layer

The pre- and postprocessing layer (consisting of pre- and postprocessing components) and the mapping layer (consisting of mapping adapters) provide the intended level of abstraction between the SUT and the computation layer. Please note that we have chosen a test-centric perspective here—preprocessing refers to the preparation of the data that is emitted by the SUT and fed into

the testing environment and postprocessing refers to the preparation of the data that is emitted by the test perturbation or the abstract environment model and sent to the SUT.

- Preprocessing Component

  The preprocessing component is responsible for the measurement and preparation of the outcome of the SUT for later use in the computation layer. Usually, it is neither intended nor possible to assess the data from the SUT without preprocessing. We need an abstract or condensed version of the output data. For example, the control of a lamp may be performed by pulse-width modulated current. To perform a proper assessment of the signal, it would suffice to know the duty cycle. The preprocessing component serves as such an abstraction layer. This component can be easily designed and developed using modeling tools, such as Simulink, Stateflow, or Modelica.

- Postprocessing Component

  The postprocessing component is responsible for the generation of the concrete input data for the SUT. It adapts the low level interfaces of the SUT to the interfaces of the computation layer, which are usually more abstract. This component is best modeled with the help of Simulink, Stateflow, Modelica, or other tools and programming languages, which are available in the underlying test and simulation infrastructure.

- Mapping Adapter

  The mapping adapter is responsible for the syntactic decoupling of the environment model and the execution object, which in our case is the SUT. Its main purpose is to relate (map) the input ports of the SUT to the output ports of the environment model and vice versa. Thus, changing the names of the interfaces and ports of the SUT would only lead to slight changes in the mapping adapter.

### 1.3.2 Requirements on the Development of the Generic Test Model

The essential part of a closed loop test is the modeling of the environment feedback. Such a model is an abstraction of the environment the SUT operates in. Because this model is only suited for

testing and simulation, generally we need not be concerned with performance and completeness issues. However, closed loop testing always depends on the quality of the test model and we should carefully develop the test model in order to get reliable results. Besides general quality issues that are well known from model-based development, the preparation of reusable environment models must address some additional aspects.

Because of the signal transformations in the pre- and postprocessing components we can design the test model at a high level of abstraction. This eases the reuse in different environments. Moreover, we are not bound to a certain processor architecture. When we use processors with floating point arithmetic in our test system, we do not have to bother with scalings. The possibility to upgrade the performance of our test system, for example, by adding more Random Access Memory (RAM) or a faster processor helps mitigate performance problems. Thus, when we develop a generic test model, we are primarily interested in the correct functionality and less in the performance or the structural correctness. We will therefore focus on functional tests using the aforementioned model and disregard structural or more general white box tests.

To support the reuse in other projects or in later phases of the same project we should carefully document the features of the model, their abstractions, their limits, and the meaning of the model parameters. We will need full version management for test models in order to be able to reproducibly check the correctness of the SUT behavior with the help of closed loop tests.

In order to develop proper environment models, high skills in the application domain and good capabilities in modeling techniques are required. The test engineer, along with the application or system engineer shall therefore consult the environment modeler to achieve sufficient testability and the most appropriate level of abstraction for the model.

### 1.3.3 Running Example

As a guiding example we will consider an engine controller that controls the engine speed by opening and closing the throttle. It is based on a demonstration example (Engine Timing Model with Closed Loop Control) provided by the MATLAB™ (The MathWorks 2010a) and Simulink tool suites.

The engine controller (Figure 1.3.3) has three input values, namely, the valve trigger (Valve_Trig_), the engine speed (Eng_Speed_), and the set point for the engine speed (Set_Point_). Its objective is the control of the air flow to the engine by means of the throttle angle (Throttle_Angle). The controller software is specifically designed and implemented to run on a real ECU with fixed point arithmetic, thus we must be concerned with scaling and rounding.

**Please place Figure Fig01-02.pdf here. Caption:** "ECU model for a simple throttle control"

The environment model (Figure 1.3.3) is a rather abstract model of a four-cylinder spark ignition engine. It processes the engine angular velocity (shortened: Engine Speed (rad/s)) and the crank angular velocity (shortened: Crank Speed (rad/s)) controlled by the throttle angle. The throttle angle is the control variable, the engine speed the observable. The angle of the throttle controls the air flow into the intake manifold. Depending on the engine speed, the air is forced from the intake manifold into the compression subsystem and periodically pumped into the combustion subsystem. By changing the period for pumping the air mass provided for the combustion subsystem is regulated, and the torque of the engine, as well as the engine speed are controlled. Technically, the throttle angle should be controlled at any valve trigger in such a way that the engine speed approximately reaches the set point speed.

**Please place Figure Fig01-03.pdf here. Caption:** "Environment model for a simple throttle control"

To model this kind of environment model we use floating point arithmetic. It is a rather small model, but it is reasonable enough to test basic features of our engine controller (e.g., the proper functionality, robustness, and stability). Moreover, we can potentially extend the scope of the model by providing calibration parameters to adapt the behavior of the model to different situations and controllers later.

To test the example system, we will use the test architecture proposed in Section 1.3.1. The perturbation and assessment component is a TTCN-3 embedded component that compiles to a Simulink S-Function. Figure 1.3.3 shows the complete architecture and identifies the individual components.

**Please place Figure Fig01-04.pdf here. Caption:** "Test architecture in Simulink™"

Table 1.1: Engine Controller Test Interface

| test system symbol | system symbol | direction | unit | data type |
|---|---|---|---|---|
| ti_Crank_Speed | Crank Speed (rad/s) | in | rpm | double |
| ti_Engine_Speed | Engine Speed (rpm) | in | rpm | double |
| ti_Throttle_Angle | Throttle Angle Out | in | rad | double |
| to_Engine_Perturbation | Eng_Speed_ | out | rpm | double |
| to_Set_Point | Set_Point_ | out | rpm | double |

The resulting test interface, that is, the values that can be assessed and controlled by the perturbation and assessment component are depicted in Table 1.1. Note that we use a test system centric perspective for the test interface, that is, system inputs are declared as outputs and system or environment model outputs as inputs. The mapping between the test system specific names and the system and environment model specific names is defined in the first and second column of the table.

On the basis of the test architecture and the test interface, we are now able to identify typical test scenarios:

- The set point speed `to_Set_Point` jumps from low to high. How long will it take till the the engine speed `ti_Engine_Speed` reaches the set point speed?
- The set point speed `to_Set_Point` falls from high to low. How long will it take till the the engine speed `ti_Engine_Speed` reaches the set point speed? Are there any over-regularizations?
- The engine speed sensor retrieves pertubed values `to_Engine_Perturbation`. How does the controller behave?

## 1.4 TTCN-3 Embedded for Closed Loop Tests

For the specification of the tests we rely on a formal testing language, which provides dedicated means to specify the stimulation of the system and the assessment of the system's reaction. To emphasize the reusability, the language should provide at least sufficient support for modularization as well as support for the specification of reusable entities, such as functions, operations, and

parameterization.

In general, a testing language for software driven hybrid control systems should provide suitable abstractions to define and assess analogue and sampled signals. This is necessary in order to be able to simulate the SUT's physical environment and to interact with dedicated environment models that show continuous input and output signals. On the other hand, modern control systems consist of distributed entities (e.g., controllers, sensors, actuators) that are interlinked by network infrastructures (e.g., CAN or FlexRay buses in the automotive domain). These distributed entities communicate with each other by exchanging complex messages using different communication paradigms such as asynchronous event based communication or synchronous client server communication[2]. Typically, this kind of behavior is tested using testing languages, which provide support for event or message based communication and provide means to assess complex data structures.

In recent years, there have been many efforts to define and standardize formal testing languages. In the telecommunications industry the Testing and Test Control Notation (TTCN-3) (ETSI 2009b; ETSI 2009a) is well established and widely proliferated. The language is a complete redefinition of the Tree and Tabular Combination Notation (TTCN-2) (ISO/IEC 1998). Both notations are standardized by the European Telecommunications Standards Institute (ETSI) and the International Telecommunication Union (ITU). Additional testing and simulation languages, especially ones devoted to continuous systems in particular, are available in the field of hardware testing or control system testing. The Very High Speed Integrated Circuit Hardware Description Language (VHDL) (IEEE 1993) and its derivative for analog and mixed-signals (VHDL-AMS) (IEEE 1999) is useful in the simulation of discrete and analogue hardware systems. However, both languages were not specifically designed to be testing languages. The Boundary Scan Description Language (BSDL) (Parker and Oresjo 1991) and its derivative the Analog Boundary Scan Description Language (ABSDL) (Suparjo, Ley, Cron, and Ehrenberg 2006) are testing languages that directly support the testing of chips using the boundary scan architecture (IEEE 2001) defined by the Institute of Electrical and Electronics Engineers (IEEE). The Time Partition Testing Method (TPT) (Bringmann and Kraemer 2006) and the Test Markup Language (TestML) (Grossmann and Mueller 2006) are approaches that have been developed in the automotive industry about ten yeasrs

[2]Please refer to AUTOSAR (AUTOSAR Consortium 2010), which yields a good example of an innovative industry grade approach to designing complex control system architectures for distributed environments.

ago, but are not yet standardized. The Abbreviated Test Language for All Systems (ATLAS) (IEEE 1995) and its supplement, the Signal and Method Modeling Language (SMML) (IEEE 1998), define a language set that was mainly used to test control systems for military purposes. Moreover, the IEEE currently finalizes the standardization of an XML based test exchange format, namely the Automatic Test Mark-up Language (ATML) (SCC20 ATML Group 2006), which is dedicated to exchanging information on test environments, test setups, and test results in a common way and the European Space Agency (ESA) defines requirements on a language used for the development of automated test and operation procedures and standardized a reference language called Test and Operations Procedure Language (ESA-ESTEC 2008). Last, but not least, there exist a huge number of proprietary test control languages that are designed and made available by commercial test system manufacturers.

Most of the languages mentioned above are neither able to deal with complex discrete data that are exhaustively used in network interaction, nor with distributed systems. On the other hand, TTCN-3, which is primarily specializing in testing distributed network systems, lacks support for discretized or analogue signals to stimulate or assess sensors and actuators. ATML, which potentially supports both, is only an exchange format, yet to be established, and still lacking user friendly representation formats.

The TTCN-3 standard provides a formal testing language that has the power and expressiveness of a normal programming language with formal semantics and a user friendly textual representation. It also provides strong concepts for the stimulation, control, and assessment of message based and procedure based communication in distributed environments. Our anticipation is that these kinds of communication will become much more important for distributed control systems in the future. Additionally, some of these concepts can be reused to define signal generators and assessors for continuous systems, and thus provide a solid basis for the definition of analogue and discretized signals. Finally, the overall test system architecture proposed in the TTCN-3 standard (ETSI 2009c) shows abstractions that are similar to the ones we defined in Section 1.3.1. The TTCN-3 system adapter and the flexible codec entities provide abstraction mechanisms that mediate the differences between the technical SUT interface and the specification level interfaces of the test cases. This corresponds to the pre- and postprocessing components from Section 1.3.1. Moreover, the TTCN-3 map statement allows the flexible specification of the mappings between

the ports of the SUT and the ports of the test system at runtime. This directly corresponds to the mapping components from Section 1.3.1. In addition, the TTCN-3 standard defines a set of meta interfaces (i.e., Test Runtime Interface (TRI) (ETSI 2009c), Test Control Interface (TCI) (ETSI 2009d)) that precisely specify the interactions between the test executable, the adapters, and the codecs, and shows a generalizable approach for a common test system architecture. Last, but not least, the TTCN-3 standard is actually one of the major European testing standards with a large number of contributors.

To overcome its limitations and to open TTCN-3 for embedded systems in general and for continuous real-time systems in particular, the standard must be extended. A proposal for such an extension, namely TTCN-3 embedded, was developed within the TEMEA (TEMEA 2010) research project and integrates former attempts to resolve this issue (Schieferdecker and Grossmann 2007; Schieferdecker, Bringmann, and Grossmann 2006). Next, we will outline the basic constructs of TTCN-3 and TTCN-3 embedded and show how the underlying concepts fit to our approach for closed loop testing.

### 1.4.1 Basic Concepts of TTCN-3 Embedded

TTCN-3 is a procedural testing language. Test behavior is defined by algorithms that typically assign messages to ports and evaluate messages from ports. For the evaluation of different alternatives of expected messages, or timeouts, the port queues and the timeout queues are frozen when the evaluation starts. This kind of snapshot semantics guarantees a consistent view on the test system input during an individual evaluation step. Whereas the snapshot semantics provides means for a pseudo parallel evaluation of messages from several ports, there is no notion of simultaneous stimulation and time triggered evaluation. To enhance the core language for the requirements of continuous and hybrid behavior we introduce:

- the notions of time and sampling,
- the notions of streams, stream ports, and stream variables,
- the definition of statements to model a control flow structure similar to that of hybrid automata.

We will not present a complete and exhaustive overview of TTCN-3 embedded[3]. Instead, we will highlight some basic concepts, in part by providing examples, and show the applicability of the language constructs to the closed loop architecture defined in the sections above.

### Time

TTCN-3 embedded provides dedicated support for time measurement and time triggered control of the test system's actions. Time is measured using a global clock, which starts at the beginning of each test case. The actual value of the clock is given as a float value that represents seconds and which is accessible in TTCN-3 embedded using the keyword **now**. The clock is sampled, thus it is periodically updated and has a maximum precision defined by the sampling step size. The step size can be specified by means of annotations to the overall TTCN-3 embedded module. Listing 1.1 shows the definition of a TTCN-3 embedded module that demands a periodic sampling with a step size of one millisecond.

Listing 1.1: Time

```
module myModule { ... } with {stepsize '0.001'}
```

### Streams

TTCN-3 supports a component based test architecture. On a conceptual level, test components are the executable entities of a test program. To realize a test setup at least one test component and an SUT are required. Test components and the SUT are communicating by means of dedicated interfaces called ports. While in standard TTCN-3 interactions between the test components and the SUT are realized by sending and receiving messages through ports, the interaction between continuous systems can be represented by means of so called streams. In contrast to scalar values, a stream represents the entire allocation history applied to a port. In computer science, streams are widely used to describe finite or infinite data flows. To represent the relation to time, so called timed streams (Broy 1997; Lehmann 2004) are used. Timed streams additionally provide timing information for each stream value and thus enable the traceability of timed behavior.

[3]For further information on TTCN-3 embedded please refer to (TEMEA 2010)

TTCN-3 embedded provides timed streams. In the following we will use the term (measurement) record to denote the unity of a stream value and the related timing in timed streams. Thus, concerning the recording of continuous data, a record represents an individual measurement, consisting of a stream value that represents the data side and timing information that represents the temporal perspective of such a measurement.

TTCN-3 embedded sends and receives stream values via ports. The properties of a port are described by means of port types. Listing 1.2 shows the definition of a port type for incoming and outgoing streams of the scalar type float and the definition of a component type that defines instances of these port types (`ti_Crank_Speed`, `ti_Engine_Speed`, `ti_Throttle_Angle`, `to_Set_Point`, `to_Engine_Perturbation`) with the characteristics defined by the related port type specifications (Listing 1.2, Lines 5-6).

Listing 1.2: Ports

```
type port FloatInPortType stream {in float};                                    1
type port FloatOutPortType stream {out float};                                  2
                                                                                3
type component EngineTester{                                                    4
  port FloatInPortType ti_Engine_Speed, ti_Crank_Speed, ti_Throttle_Angle;      5
  port FloatOutPort to_Engine_Perturbation, to_Set_Point;                       6
}                                                                               7
```

With the help of TTCN-3 embedded language constructs, it is possible to modify, access, and assess stream values at ports. Listing 1.3 shows how stream values can be written to an outgoing stream and read from an incoming stream.

Listing 1.3: Stream Value Access

```
to_Set_Point.value := 5000.0;                                                   1
to_Set_Point.value := ti_Engine_Speed.value + 200.0;                            2
```

Apart from the access to an actual stream value, TTCN-3 embedded provides access to the history of stream values by means of index operators. We provide time based indices and sample based indices. The time based index operator **at** interprets time parameters as the time that has

expired since the test has started. Thus, the expression `ti_Engine_Speed.at(10.0).value` yields the value that has been available at the stream 10 seconds after the test case has started. The sample based index operator **prev** interprets the index parameter as the number of sampling steps that have passed between the actual valuation and the one that will be returned by the operator. Thus, `t_engine.prev(12).value` returns the valuation of the stream 12 sampling steps in the past. The expression `t_engine.prev.value` is a short form of `t_engine.prev(1).value`. Listing 1.4 shows some additional expressions based on the index operators.

Listing 1.4: Stream History Access

```
to_Set_Point.value := ti_Engine_Speed.at(10.0).value;
to_Set_Point.value := ti_Engine_Speed.prev.value
                    + ti_Engine_Speed_perturb.prev(2).value;
```

Using the **assert** statement, we can assess the outcome of the SUT. The assert statement specifies the expected behavior on the SUT by means of relational expressions. Hence, we can use simple relational operators that are already available in standard TTCN-3 and apply them to the stream valuation described above to express predicates on the behavior of the system. If any of the predicates specified by an active assert statement is violated, the test verdict is automatically set to `fail` and the test fails. Listing 1.5 shows the specification of an assertion that checks whether the engine speed is in the range between 1000 and 3000 rpm.

Listing 1.5: Assert

```
assert(ti_Engine_Speed.value > 1000.0, ti_Engine_Speed.value < 3000.0);
// the values must be in the range ]1000.0, 3000.0[.
```

### Access to Time Stamps and Sampling Related Information

To complement the value side of a stream, TTCN-3 embedded additionally provides access to sampling related information, such as time stamps and step sizes, so as to provide access to all the necessary information related to measurement records of a timed stream. The time stamp of a measurement is obtained by means of the **timestamp** operator, which can be used instead of the already known value operator. The timestamp operator yields the exact measurement time for

a certain stream value. The exact measurement time denotes the moment when a stream value has been made available to the test system's input and thus strongly depends on the sampling rate. Listing 1.6 shows the retrieval of measurement time stamps for three different measurement records. Line 3 shows the retrieval of the measurement time for the actual measurement record of the engine speed, Line 4 shows the same for the previous record, and Line 5 shows the time stamp of the last measurement record that has been measured before or at 10 seconds after the start of the test case.

Listing 1.6: Time Stamp Access

```
1 var float myTimePoint1, myTimePoint2, myTimePoint3;
2 ...
3 myTimePoint1 := ti_Engine_Speed.timestamp;
4 myTimePoint2 := ti_Engine_Speed.prev().timestamp;
5 myTimePoint3 := ti_Engine_Speed.at(10.0).timestamp;
```

As already noted, the result of the timestamp operator directly relates to the sampling rate. The result of `ti_Engine_Speed.timestamp` need not be equal to **now**, when we consider different sampling rates at ports. The same applies to the expression `ti_Engine_Speed.at(10.0).timestamp`. Dependent on the sampling rate it may yield 10.0, or possibly an earlier time (e.g., when the sampling rate is 3.0 we will have measurement records for the time points 0.0, 3.0, 6.0, and 9.0. and the result of the expression will be 9.0).

In addition to the timestamp operator, TTCN-3 embedded enables one to obtain the step size that has been used to measure a certain value. This information is provided by the **delta** operator, which can be used in a similar way as the **value** and the **timestamp** operators. The delta operator returns the size of the sampling step (in seconds) that precedes the measurement of the respective measurement record. Thus, `ti_Engine_Speed.delta` returns:

```
ti_Engine_Speed.timestamp - ti_Engine_Speed.prev.timestamp
```

Please note, TTCN-3 embedded envisions dynamic sampling rates at ports. The delta and timestamp operators are motivated by the implementation of dynamic sampling strategies and thus can only develop their full potential in such contexts. Because of the space limitations the correspond-

ing concepts are not explained here. For more detail the reader is referred to (TEMEA 2010).

Listing 1.7 shows the retrieval of the step size for different measurement records.

Listing 1.7: Sampling Access

```
var float myStepSize1, myStepSize2, myStepSize3;
...
myStepSize1 := ti_Engine_Speed.delta;
myStepSize2 := ti_Engine_Speed.prev().delta;
myStepSize3 := ti_Engine_Speed.at(10.0).delta;
```

### Integration of Streams with Existing TTCN-3 Data Structures

To enable the processing and assessment of stream values by means of existing TTCN-3 statements, we provide a mapping of streams, stream values and the respective measurement records to standard TTCN-3 data structures, namely records and record-of structures. Thus, each measurement record, which is available at a stream port, can be represented by an ordinary TTCN-3 record with the structure defined in Listing 1.8. Such a record contains fields, which provide access to all value and sampling related information described in the sections above. Thus, it includes the measurement value (`_value`) and its type[4] (`T`), its relation to absolute time by means of the `_timestamp` field as well as the time distance to its predecessor by means of the `_delta` field. Moreover, a complete stream or a stream segment maps to a record-of structure, which arranges subsequent measurement records (see Listing 1.8, Line 4).

Listing 1.8: Mapping to TTCN-3 Data Structures

```
type record Measurement <in type T> {T _value, float _delta, float _timestamp}
type record of Measurement<T> Stream_Records;
```

To obtain stream data in accordance to the structure in Listing 1.8, TTCN-3 embedded provides an operation called **history**. The history operation extracts a segment of a stream from a given stream port and yields a record-of structure (stream-record), which complies to the definitions

[4]The type in this case is passed as a parameter which is possible with the new TTCN-3 advanced parameterization extension(ETSI 2009e).

stated above. Please note, the data type `T` depends on the data type of the stream port and is set automatically for each operation call.

The history operation has two parameters that characterize the segment by means of absolute time values. The first parameter defines the lower temporal limit and the second parameter defines the upper temporal limit of the segment to be returned. Listing 1.9 illustrates the usage of the history operation. We start with the definition of a record-of structure that is intended to hold measurement records with float values. In this context the application of the history-operation in Line 2 yields a stream-record that represents the first ten seconds at `ti_Engine_Speed`. Please note, the overall size of the record of structure that is the number of individual measurement elements depends on the time interval defined by the parameters of the history operation, as well as on the given sampling rate (see Section 1.4.1).

Listing 1.9: The History Operation

```
type record of Measurement<float> Float_Stream_Records;                    1
var Float_Stream_Records speed := ti_Engine_Speed.history(0.0,10.0);       2
```

We can use the record-of representation of streams to assess complete stream segments. This can be achieved by means of an assessment function, which iterates over the individual measurement records of the stream record, or by means of so-called stream templates, which characterize a sequence of measurement records as a whole. While such assessment functions are in fact only small TTCN-3 programs, which conceptually do not differ from similar solutions in any other programming language, the template concepts are worth explaining in more detail here.

A template is a specific data structure that is used to specify the expectations on the SUT by means of distinct values, but also by means of data-type specific patterns. These patterns allow, among other things, the definition of ranges (e.g., `field := (lowerValue .. upperValue)`), lists (e.g., `field := (1.0, 10.0, 20.0)`), and wildcards (e.g., `field := ?` or `field := *`). Moreover, templates can be applied to structured data types and record-of structures. Thus, we are able to define structured templates that have fields with template values. Last but not least, templates are parameterizable so that they can be instantiated with different value sets[5].

[5]Please note that the TTCN-3 template mechanism is a very powerful concept, which cannot be explained in full detail here. To obtain further information on the subject of TTCN-3 templates please refer to (ETSI 2009a).

Templates are checked against existing values by means of certain statements.

- The **match** operation already exists in standard TTCN-3. It tests whether an arbitrary template matches a given value completely. The operation returns true if the template matches and false otherwise. In case of a record-of representation of a stream we can use the match operation to check the individual stream values with templates that conform to the type definitions in Section 1.8.
- The **find** operation has been newly introduced in TTCN-3 embedded. It scans a record-structured stream for the existence of a structure that matches the given template. If such a structure exists, the operation returns the index value of the matching occurrence. Otherwise, it returns -1. In case of a record-of representation of a stream, we can use the find statement to search the stream for the first occurrence of a distinct pattern.
- The **count** operation has been newly introduced in TTCN-3 embedded as well. It scans a record-structured stream and counts the occurrences of structures that match a given template. The operation returns the number of occurrences. Please note, the application of the count-operation is not greedy and checks the templates iteratively starting with each measurement record in a given record-of structure.

Listing 1.10 shows a usage scenario for stream templates. It starts with the manual definition of a record template. The template specifies a signal pattern with the following characteristics: the signal starts with an arbitrary value, after 2 seconds the signal value is between 1900 and 2100, after the next 2 seconds the signal value reaches a value between 2900 and 3100, thereafter (i.e., 2 seconds later) it reaches a value between 2950 and 3050, and finally ends with an arbitrary value. Please note, in this case we are not interested in the absolute time values and thus allow arbitrary values for the `_timestamp` field.

Listing 1.10: Using Templates to Specify Signal Shapes

```
                                                                    1
template Float_Stream_Record toTest := {                            2
  {_value := ?                 ,   _delta := 0.0,   _timestamp:= ?}, 3
  {_value := (1900.0 .. 2100.0) , _delta := 2.0,   _timestamp:= ?}, 4
  {_value := (2900.0 .. 3100.0) , _delta := 2.0,   _timestamp:= ?}, 5
```

```
  {_value := (2950.0 .. 3050.0) , _delta := 2.0,    _timestamp:= ?},
  {_value := ?                  ,    _delta := 2.0,    _timestamp:= ?}
}

// checks, whether a distinct segment conforms to the template toTest
match(ti_Engine_Speed.history(2.0, 10.0), toTest);
// finds the first occurrence of a stream segment that conforms to toTest
find(ti_Engine_Speed.history(0.0, 100.0), toTest);
// counts all occurrences of stream segments that conform to toTest
count(ti_Engine_Speed.history(0.0, 100.0), toTest);
```

One should note, that the application of the match operation requires that the stream segment and the template have the same length. If this is not the case the match operation fails.

The manual definition of stream templates can be cumbersome and time-consuming. To support the specification of more complex patterns we propose the use of generation techniques and automation, which can easily be realized by means of TTCN-3 functions and parameterized templates[6]. Listing 1.11 shows such a function that generates a template record out of a given record. The resulting template allows checking a stream against another stream (represented by means of the `Float_Stream_Record myRecordS`) and additionally allows a parameterizable absolute tolerance for the value side of the stream.

Listing 1.11: Template Generation Function

```

function generateTemplate(in Float_Stream_Record myStreamR, in float tolVal)
return template Float_Stream_Record{
  var integer i;
  var template Float_Stream_Record toGenerate
  template Measurement<float> tolerancePattern(in float delta,
                                               in float value,
                                               in float tol) := {
    _delta := delta,
```

[6]Future work concentrates on the extensions for the TTCN-3 template language to describe repetitive and optional template groups. This will yield a regular expression like calculus, which provides a much more powerful means to describe the assessment for stream-records.

```
    _value := ((value - (tol / 2.0)) .. (value + (tol / 2.0))),
    _timestamp:= ?
  }

  for (i := 0, i < sizeof(myStreamRecord), i := i + 1){
    toGenerate[i] := tolerancePattern(myStreamR[i].delta,
                                      myStreamR[i].value, tolVal);
  }
  return toGenerate;
}
```

These kind of functions are not intended to be specified by the test designer, but rather provided as part of a library. The example function presented here is neither completely elaborated nor does it provide the sufficient flexibility of a state of the art library function. It is only intended to illustrate the expressive power and the potential of TTCN-3 and TTCN-3 embedded.

### Control Flow

So far we have only reflected on the construction, application and assessment of single streams. For more advanced test behavior, such as concurrent applications, assessment of multiple streams, and detection of complex events (e.g., zero crossings or flag changes) richer capabilities are necessary.

For this purpose, we combine the concepts defined in the previous section with state machine-like specification concepts, called modes. Modes are well known from the theory of hybrid automata (Alur, Henzinger, and Sontag 1996; Lynch, Segala, Vaandrager, and Weinberg 1995; Alur, Courcoubetis, Henzinger, and Ho 1992). A mode is characterized by its internal behavior and a set of predicates, which dictate the mode activity. Thus, a simple mode specification in TTCN-3 embedded consists of three syntactical compartments: a mandatory body to specify the mode's internal behavior; an invariant block that defines predicates that must not be violated while the mode is active; and a transition block that defines the exit condition to end the mode's activity.

Listing 1.12: Atomic Mode

```
cont { //body
        // ramp, the value increases at any time step by 3
        to_Set_Point.value := 3.0 * now;
        // constant signal
        to_Engine_Perturbation.value := 0.0;
}
inv {  // invariants
        // stops when the set point exceeds a value of 20000.0
        to_Set_Point.value > 20000.0;
}
until { //transition
        [ti_Engine_Speed.value > 2000.0] {to_Engine_Perturbation.value := 2.0;}
}
```

In the example in Listing 1.12, the set point value `to_Set_Point` increases linearly in time and the engine perturbation `t_engine_perturb` is set constantly to 0.0. This holds as long as the invariant holds and the **until** condition does not fire. If the invariant is violated, that is the set point speed exceeds 20000.0, an error verdict is set and the body action stops. If the **until** condition yields true, that is the value of `ti_Engine_Speed` exceeds 2000.0, the `to_Engine_Perturbation` value is set to 2.0 and the body action stops.

To combine different modes into larger constructs we provide parallel and sequential composition of individual modes and of composite modes. The composition is realized by the **par** operator (for parallel composition) and the **seq** operator (for sequential composition). Listing 1.13 shows two sequences, one for the perturbation actions and one for the assessment actions, which are themselves composed in parallel.

Listing 1.13: Composite Modes

```
par { // overall perturbation and assessment
  seq{// perturbation sequence
    cont{// perturbation action 1}
    cont{// perturbation action 2}
    ...}
  seq{// assessment sequence
```

```
    cont{// assessment action 1}
    cont{// assessment action 1}
    ...}
}
```

In general, composite modes show the same structure and behavior as atomic modes as far as invariants and transitions are concerned. Hence, while being active, each invariant of a composite mode must hold. Additionally, each transition of a composite mode ends the activity of the mode when it fires. Furthermore, each mode provides access to an individual local clock that returns the time that has passed since the mode has been activated. The value of the local clock can be obtained by means of the keyword **duration**.

Listing 1.14: Relative Time

```
seq{// perturbation sequence
    cont{to_Set_Point.value := 20000.0;} until (duration > 3.0)
    cont{to_Set_Point.value := 40000.0 + 100.0 * duration;} until (duration > 2.0)
} until (duration > 4.0)
```

Listing 1.14 shows the definition of three modes, each of which has a restricted execution duration. The value of `to_Set_Point` is increased continuously by means of the duration property. The duration property is defined locally for each mode. Thus, the valuation of the property would yield different results in different modes.

### 1.4.2 Specification of Reusable Entities

This section presents more advanced concepts of TTCN-3 embedded that are especially dedicated to specifying reusable entities. We aim to achieve a higher degree of reusability by modularizing the test specifications and supporting the specification of abstract and modifiable entities. Some of these concepts are well known in computer language design and already available in standard TTCN-3. However, they are only partially available in state of the art test design tools for continuous and hybrid systems and they must be adapted to the concepts we introduced in 1.4.1. The concepts dedicated to support modularization and modification are:

- branches and jumps to specify repetitions and conditional mode execution,
- symbol substitution and referencing mechanisms, and
- parameterization.

### Conditions and Jumps

Apart from the simple sequential and parallel composition of modes, stronger concepts to specify more advanced control flow arrangements, such as conditional execution and repetitions are necessary. TTCN-3 already provides a set of control structures for structured programming. These control structures, such as if statements, while loops, and for loops are applicable to the basic TTCN-3 embedded concepts as well. Hence, the definition of mode repetitions by means of loops, as well as the conditional execution of assertions and assignments inside of modes are allowed. Listing 1.15 shows two different use cases for the application of TTCN-3 control flow structures that directly interact with TTCN-3 embedded constructs. In the first part of the listing (Lines 4 and 6), an if statement is used to specify the conditional execution of assignments inside a mode. In the second part of the listing (Lines 10–14), a while loop is used to repeat the execution of a mode multiple times.

Listing 1.15: Conditional Execution and Loops

```
cont { //body                                                              1
  // ramp until duration >= 4.0                                           2
  if (duration < 4.0) {to_Set_Point.value := 3.0 * now;}                  3
  // afterwards the value remains constant                                4
  else {to_Set_Point.value := to_Set_Point.prev.value;}                   5
}                                                                         6
                                                                          7
// generating a chain tooth signal for 3 minutes with a period of 5.0 seconds  8
while (now < 180.0) {                                                     9
  cont {                                                                  10
    to_Set_Point.value := 3.0 * duration;                                 11
  } until (duration > 5.0)                                                12
}                                                                         13
```

For full compatibility with the concepts of hybrid automata, definition of so-called transitions must be possible as well. A transition specifies the change of activity from one mode to another mode. In TTCN-3 embedded we adopt these concepts and provide a syntax, which seamlessly integrates with already existing TTCN-3 and TTCN-3 embedded concepts. As already introduced in the previous section transitions are specified by means of the `until` block of a mode. In the following, we will show how a mode can refer to multiple consecutive modes by means of multiple transitions and how the control flow is realized.

A transition starts with a conditional expression, which controls the activation of the transition. The control flow of transitions resembles the control flow of the already existing (albeit antiquated) TTCN-3 `label` and `goto` statements. These statements have been determined sufficiently suitable for specifying the exact control flow after a transition has fired. Thus, there is no need to introduce additional constructs here. A transition may optionally contain arbitrary TTCN-3 statements to be executed when the transition fires.

Listing 1.16 illustrates the definition and application of transitions by means of pseudo code elements. The predicate `<activation_predicate>` is an arbitrary predicate expression that may relate to time values or stream values or both. The `<optional_statement_list>` may contain arbitrary TTCN-3 or TTCN-3 embedded statements except blocking or time consuming statements (alt statements and modes). Each goto statement relates to a label definition that specifies the place where the execution is continued.

Listing 1.16: Transitions

```
label labelsymbol_1                                                                    1
cont {} until {                                                                        2
 [<activation_predicate>] {<optional_statement_list>} goto labelsymbol_2;              3
 [<activation_predicate>] {<optional_statement_list>} goto labelsymbol_3;              4
}                                                                                      5
label labelsymbol_2; cont {} goto labelsymbol_4;                                       6
label labelsymbol_3; cont {} goto labelsymbol_1;                                       7
label labelsymbol_4;                                                                   8
```

Listing 1.17 shows a more concrete example that relates to our example from Section 1.3.3. We define a sequence of three modes that specify the continuous valuation of the engine's set point (`to_Set_Point`), depending on the engine's speed (`engine_speed`). When the engine speed exceeds 2000.0 rpm the set point is decreased (**`goto`** `decrease`), otherwise it is increased (**`goto`** `increase`).

Listing 1.17: Condition and Jumps

```
testcase myTestcase() runs on EngineTester {                                    1
                                                                                2
  // reusable mode application                                                  3
  cont {to_Set_Point.value := 3.0 * now;}                                       4
  until {                                                                       5
   [duration > 10.0 and engine_speed.value > 2000.0] goto increase;             6
   [duration > 10.0 and engine_speed.value <= 2000.0] goto decrease             7
  }                                                                             8
  label increase;                                                               9
  cont {to_Set_Point.value := 3 * now;} until  {[duration > 10.0 ] goto end;}   10
  label decrease;                                                               11
  cont {to_Set_Point.value := 3 * now;} until  (duration > 10.0 )               12
  label end;                                                                    13
}                                                                               14
```

### Symbol Substitution and Referencing

Similar to the definition of functions and functions calls, it is possible to declare named modes, which can then be referenced from any context that would allow the explicit declaration of modes. Listing 1.18 shows the declaration of a mode type (Line 1), a named mode[7] (Line 4) and a reference to it within a composite mode definition (Line 11).

Listing 1.18: Symbol Substitution

```
type mode ModeType();                                                           1
                                                                                2
```

[7] Named modes, similar to other TTCN-3 elements that define test behavior, can be declared with a **`runs on`** clause, in order to have access to the ports (or other local fields) of a test component type.

```
// reusable mode declaration
mode ModeType pert_seq() runs on EngineTester seq {
  cont {to_Set_Point := 2000.0} until (duration >= 2.0)
  cont {to_Set_Point := 2000.0 + duration / to_Set_Point.delta * 10.0}
  until (duration >= 5.0)
}

testcase myTestcase() runs on EngineTester {
  par {
    pert_seq(); // reusable mode application
    cont {assert(engine_speed >= 500.0)}
   } until (duration > 10.0)
}
```

### Mode Parameterization

To provide a higher degree of flexibility, it is possible to specify parameterizable modes. Values, templates, ports, and modes can be used as mode parameters. Listing 1.19 shows the definition of a mode type, which allows the application of two float parameters and the application of one mode parameter of the mode type `ModeType`.

Listing 1.19: Mode Parameterization

```
  type mode ModeType2(in float startVal, in float increase, in ModeType assertion);

  // reusable mode declaration
  mode ModeType assert_mode() runs on EngineTester :=
    cont {assert(engine_speed >= 500.0)}

  mode ModeType2 pert_seq_2(in float startVal,
                            in float increase,
                            in ModeType assertion)
  runs on EngineTester par {
    seq{// perturbation sequence
```

```
      cont{to_Set_Point := startVal} until (duration >= 2.0)                    12
      cont{to_Set_Point := startVal + duration / to_Set_Point.delta * increase} 13
      until (duration >= 5.0)                                                   14
    }                                                                           15
    assertion();                                                                16
}                                                                               17
                                                                                18
testcase myTestcase() runs on EngineTester {                                    19
  // reusable mode application                                                  20
  pert_seq_2(1000.0, 10.0, assert_mode);                                        21
  pert_seq_2(5000.0, 1.0, cont{assert(engine_speed >= 0.0)});                   22
}                                                                               23
```

Lines 21 and 22 illustrate the application of parameterizable reusable modes. Line 21 applies `pert_seq_2` and sets the parameter values for the initial set point to 1000.0 and the parameter for the increase to 10.0, and the `assert_mode` is passed as a parameter to be applied within the `pert_seq_2` mode. Line 22 shows a more or less similar application of `pert_seq_2`, where an inline mode declaration is passed as the mode parameter.

## 1.5 Reuse of Closed Loop Test Artifacts

Systematic reuse of test artifacts can increase the test quality and reduce the development and maintenance costs for tests. Thus far, we have discussed reusability on the level of test specification language elements (referencing mechanisms, modifications operators, parameterization),in this section we focus primarily on the reuse of the high-level artifacts of our generic closed loop architecture for testing.

Despite the many enabling factors from a technological perspective, a number of organizational factors inhibiting the adoption of reuse, as well as the risks involved, have been identified in (Lynex and Layzell 1997; Lynex and Layzell 1998; Tripathi and Gupta 2006). Such organizational considerations are concerned primarily with the uncertainties related to the potential for reusability

of production code and its realization. In the current context, however, different aspects must be taken into account. Three-tier development and closed loop testing (MiL, SiL, HiL) methodologies have an increased need for traceability and consistency between the different testing levels, which could ideally be facilitated through reuse. This would potentially reduce redundant development of essentially identical artifacts that also must be maintained consistently throughout the life-cycle of the system. In addition, separate levels of development and testing may be performed by separate organizations, in which case reuse of artifacts can be of significance in reducing the overhead for the involved organizations and may thus be regulated by contractual obligations. As an example for horizontal reuse here, consider a scenario in which one subcontractor delivers test artifacts that are to be used on components from various suppliers. These components in turn may often utilize the same basic infrastructure and parts of a generic environment, which also contributes to increased reuse potential. Furthermore, there may be conformance tests for certain components provided by standardization organizations, which should again be reusable at the same testing level across different implementations, both of the components, but potentially also of the environment they shall operate in. Therefore, despite impeding factors of an organizational nature when production code is concerned, in the scope the current context, there are many organizational factors that not only facilitate the reuse of test artifacts, but also increase the need and potential for reuse.

In the following, a general approach for vertical and horizontal test specification and test model reuse at different closed loop test levels will be presented and accompanied by a concise example that shows the practical reuse of TTCN-3 embedded specifications and the environment model from Section 1.3.3 in a HiL scenario. Since closed loop tests, in general, require a complex test environment, an appropriate test management process and tool support throughout the life cycle of the SUT is required. This subject is addressed in the last subsection.

### 1.5.1 Horizontal Reuse of Closed Loop Test Artifacts

Forcing the strict separation into semantically distinct artifacts, the closed loop architecture for testing allows systematic reuse of the individual artifacts with minimal adaptation. In the usual case, the test model and the perturbation specification are created using different modeling and programming notations (Section 1.3). Such notations usually provide powerful concepts to de-

velop *for* and *with* reuse (Karlsson 1995). Choosing TTCN-3 embedded as the test specification notation for closed loop tests, reusable test suites can be developed using concepts such as symbol substitution, referencing, and parameterization. Environment modeling based on Simulink provides modularization concepts such as subsystems, libraries, and model references, which facilitate the reusability of environment models. Since the generic closed loop architecture for testing clearly separates the heterogeneous artifacts by using well defined interfaces, the notation-specific modularization and reuse concepts can be applied without interfering with each other. By lifting reuse to the architecture level the various artifacts can be reused in several scenarios.

The environment model, which serves as the generic test scenario, and the specification of the perturbation and assessment functionality can be (re-)used with different SUTs. The SUTs may differ in type and in version, but as long as they are built on common interfaces and share common characteristics then this kind of reuse is applicable.

According to Section 1.3, the environment model defines the functional context for the perturbation and assessment specifications. Hence, the environment model (or parts thereof) can be reused with different specifications of perturbations and assessments. Consequently, the expressiveness of the perturbation and assessment specifications are limited by the functionality and context provided by the environment model. This kind of reuse of environment models is commonly used for the testing of different functions of one or more SUTs in the domain defined by the environment model. Adaptation of mapping components is required if the test setup is used for different SUTs, or if the perturbation specification requires a different interpretation of the SUT behavior (pre- and postprocessing components).

Often the SUT must be validated in the context of different environments. Therefore, existing definitions of perturbation and assessment may be reused with different environment models. If the different test models provide the same interfaces, no adaptation of mapping components is required. Otherwise, the mapping components must be modified in order to bridge the abstraction gap between the new environment model and the corresponding SUT.

In general, the reuse of test specifications across different products is nowadays often used for testing products or components, which are based on a common standard (conformance testing). The emerging standardization efforts in embedded systems development (e.g., AUTOSAR (AU-

TOSAR Consortium 2010)) indicate the emerging need for such an approach.

### 1.5.2 Vertical Reuse of Environment Models

Currently, the software for embedded controllers is often developed using model-based techniques. Naturally, there is a necessity to test the artifacts in the early design phases. On the other hand, a certain class of controllers strongly depends on the behavior of the environment, so that a closed loop test is required to assess the quality of the component. Thus, in principle a detailed environment model is needed in the early design phases.

Depending on the type of the SUT, varying testing methods on different test levels may be applied, each suited for a distinct purpose. With the MiL test, the functional aspects of the model are validated. SiL testing is used to detect errors that result from software specific issues, for instance, the usage of fixed-point arithmetic. MiL and SiL tests are used in the early design and development phases, primarily to discover functional errors within the software components. These types of test can be processed on a common PC hardware, for instance through co-simulation, and, therefore, are not suitable for addressing real-time matters. To validate the types of issues that result from the usage of a specific hardware, HiL tests must be used.

Often, much effort must be expended to develop a detailed environment model and the corresponding test specifications. To reduce the effort as well as cost, it is desirable to reuse these artifacts across the different testing levels. In principle, the SUTs exhibit the same logical functionality through all testing levels. However, they are implemented with different technologies and show different integration levels with other components, including the hardware.

In the case of different implementation technologies, which often result in interfaces with the same semantics but different technological constraints and access methods, the reuse of the environment model and the perturbation and assessment specifications is straightforward. The technological adaptation is realized by means of the mapping components that bridge the technological as well as the abstraction gap between the SUT and the environment (Figure 1.5.2).

**Please place Figure Fig01-05.pdf here. Caption:** “Vertical Reuse of Environment Models”

**An Integration Testing Scenario with TTCN-3 embedded**

In Section 1.3.3, we provided an example that shows the application of our ideas within a simple MiL scenario. This section demonstrates the reuse of some of the same artifacts in a HiL scenario and provides a proof of concept illustration of the applicability of our ideas.

As outlined in previous chapters, the main artifacts for reuse are the environment model (i.e., generic test scenarios) and the test specifications. The reuse of the test specifications depends on their level of abstraction (i.e., the semantics of the specification must fit the test levels we focus on) and on some technological issues (e.g., the availability of a TTCN-3 test engine for the respective test platform). Within this example, we will show that we are able to find the most appropriate level of abstraction for the test specifications and the environment model. The technological issues are not discussed here.

Nowadays, in the automotive domain, controllers are usually developed using Simulink or Stateflow. The target environment (environment model) is usually modeled with the same tools (Figure 1.3.3). If we adhere to the architecture we proposed in Section 1.3.1, the basis for a later reuse of artifacts is provided. Tests on MiL level are usually executed in the underlying MATLAB simulation. The environment and the controller have a common time base (the simulation time). Later on, based on the well tested controller model, the object code will be generated, compiled, and flashed on a controller board. The target controller has its own operating system and thus its own time. Often, it will be connected to other controllers, actuators, and sensors via a bus system (e.g., CAN bus, FlexRay Bus, etc.). A well established toolset to test a controller in such environments is CANoe (Vector Informatics 2010).

In principle, we use pre- and postprocessing components to connect the controller board to the bus system and/or analog devices. The technological framework that allows the reuse of the environment model is already provided by the tool vendor. CANoe allows the compilation of a Simulink model and provides means for the insertion of the compiled model as a simulation node in the CANoe simulation environment. In this way the Simulink environment model takes part in test runs via the CAN bus communication.

**Please place Figure Fig01-06.pdf here. Caption:** “Vertical Reuse of Environment Model”

To reuse the TTCN-3 embedded test specification, we only need a one time effort to build a TTCN-3 embedded test adapter for Simulink and CANoe. Both being standard tools, most of the effort can be shared.

We will end this section with a concrete test example specified in TTCN-3 embedded. We refer to a model-based development process and specify tests for the controller that was already introduced in Section 1.3.3. The controller regulates the air flow and the crankshaft of a four-cylinder spark ignition engine with an internal combustion engine (Figure 1.3.3). From the requirements we deduce abstract test cases. They can be defined semi-formally and understandably in natural language. We will use the following test cases as guiding examples.

1. The set point speed jumps from 2000 rpm to 5000 rpm. It should take less than 500 ms to reach the proper actuating variable throttle angle up to an angle of 0.01 degrees.
2. The set point speed falls from 7000 rpm to 5000 rpm. It should take less than 500 ms to reach the proper actuating variable throttle angle up to an angle of 0.01 degrees. No over-regularizations are allowed.
3. The engine speed sensor data is measured up to an uncertainty of 10 rpm. The control must be stable and robust for that range. Given the manipulated variable, the throttle angle, the deviation caused by some disturbance should be less than 0.001 rpm.

In the next step, we have to implement the test cases, i.e., to concretize them so that they can be performed by a test engine. We will use TTCN-3 embedded for that. We will not go into too much details, however, we will sketch out the basic ideas. The ports of our system are defined in Listing 1.20 according to Figure 1.3.3,

Listing 1.20: Guiding Example Ports

```
type port FloatInPortType stream {in float};                                    1
type port FloatOutPortType stream {out float} with {stepsize '0.001'};          2
                                                                                3
type component ThrottleControlTester{                                           4
  port FloatInPortType ti_Throttle_Angle, ti_Engine_Speed, ti_Crank_Speed;       5
  port FloatOutPort to_Engine_Perturbation, to_Set_Point;                       6
```

```
}
```

Please note that the engine speed is given by an environment model. The same applies for the other data. To test the robustness (see abstract test case 3), we must perturb the engine speed. This is realized by means of the output `to_Engine_Perturbation`.

The first and the second abstract test cases analyze the test behavior in two similar situations. We look into the system when the engine speed jumps from one value to another value. In the first testcase, it jumps from 2000 rpm to 5000 rpm and in the second, from 7000 rpm to 5000 rpm. In order to do the specification job only once, we define a parameterizable mode that realizes a step function (Listing 1.21).

Listing 1.21: Speed Jump

```
type mode Set_Value_Jump(in float startVal, in float endVal);

mode Set_Value_Jump pert_seq(in float startVal, in float endVal)
runs on ThrottleControlTester seq{
  cont {to_Set_Point.value := startVal} until (duration >= 2.0)
  // the first 2 seconds the set point is given as startVal rpm
  cont {to_Set_Point.value := endVal } until (duration >= 5.0)
  // the next 3 seconds the set point should be endVal rpm
}

testcase TC_Speed_Jump() runs on ThrottleControlTester {
  // reusable mode application
   pert_seq (2000.0, 5000.0);
 }
```

A refined and more complex version of the mode depicted above, which uses linear interpolation and flexible durations, can easily be developed by using the ideas depicted in Listing 1.17.

In order to assess the tests, we have to check whether the controller reacts in time. For this purpose, we have to check whether the values of a stream are in a certain range. This can be

modeled by means of a reusable mode too (Listing 1.22).

Listing 1.22: Guiding Example Assertion

```
type mode Range_Check(in float startTime, in float endTime, in float setValue,
                      in float Dev, out FloatInPortType measuredStream);

// reusable mode declaration
mode Range_Check range_check (in float startTime,
                              in float endTime, in float setValue,
                              in float Dev, out FloatInPortType measuredStream)
seq {
        // wait until the startTime
        cont {} until (duration >= startTime);
        // check the engine speed until the endTime was reached
        cont {assert(measuredStream.value >= (setValue - Dev) &&
        measuredStream.value <= (setValue + Dev))}
        until (duration >= endTime)
}
```

The executable test specification of the complete abstract test case `TC_Speed_Jump` is given in Listing 1.23.

Listing 1.23: Guiding Example Test Specification

```

 testcase TC_Speed_Jump() runs on ThrottleControlTester{
   par{
     // set the jump
     pert_seq (2000.0, 5000.0);
     // check the control signal, where Desired_Speed is
     // the assumed desired value
     range_check(2.5,5.0, Desired_Speed, 0.01, ti_Engine_Speed );
   }
 }
```

In order to check for a possible overshoot, we will use the maximum value of a stream over a distinct time interval. This can be easily realized by using the constructs introduced in Section 1.4, thus we will not elaborate further on this here.

To test the robustness, a random perturbation of the engine speed is necessary. It is specified by means of the random value function `function rnd(float seed) return float` of TTCN-3. The function returns a random value between 0.0 and 1.0. Listing 1.24 shows the application of the random function to the engine perturbation port.

Listing 1.24: Random Perturbation

```
// rnd(float seed) retrieves random values between [0,1]
to_Engine_Perturbation := rnd(0.2) * 20.0 - 10.0;
```

This random function can be used to define the perturbation resulting from uncertain measurement. To formulate a proper assessment for the third abstract test case, the parameterized mode `range_check` can be reused with the stream `ti_Throttle_Angle`. We will leave this exercise for the reader.

By using proper TTCN-3 embedded test adapter, we can perform the tests in a MATLAB simulation, and as outlined in a CANoe HiL environment as well.

This example shows the reusability of TTCN-3 embedded constructs and together with co-simulation, we establish an integrated test process over vertical testing levels.

### 1.5.3 Test Management for Closed Loop Tests

Closed loop tests require extended test management with respect to reusability, because of the fact that several different kinds of reusable test artifacts are involved. Managing test data systematically relies on a finite data set representation. For open loop tests, this often consists of input data descriptions that are defined by finitely many support points and interpolation prescriptions, such as step functions, ramps, or splines. The expectation is usually described by the expected values or ranges at specific points in time or time spans.

This no longer works for closed loop tests. In such a context, an essential part of a test case specification is defined by a generic abstract environment model, which may contain rather complex algorithms. At a different test level, this software model may be substituted by compiled code or a hardware node. In contrast to open loop testing, there is a need for a distinct development and management process incorporating all additional assets into the management of the test process. In order to achieve reproducibility and reusability, the asset management process must be carefully designed to fit into this context. The following artifacts that uniquely characterize closed loop test specifications must be taken into consideration:

- Abstract environment model (the basic generic test scenario)
- Designed perturbation and assessment specification
- Corresponding pre- and postprocessing components

All artifacts that are relevant for testing must be versioned and managed in a systematic way. Figure 1.5.3 outlines the different underlying supplemental processes. The constructive development process that produces the artifacts to be tested is illustrated on the right hand side. Parallel to it, an analytic process takes place. Whereas in open loop testing the tests are planned, defined, and performed within this analytic process, in closed loop architectures there is a need for an additional complementary development process for the environment models. The skills required to develop such environment models and the points of interest they have to meet distinguish this complementary process from the development processes the tests and the corresponding system.

**Please place Figure Fig01-07.pdf here. Caption:** “Management of Test Specifications”

## 1.6 Quality Assurance and Guidelines for the Specification of Reusable Assets

As outlined above, TTCN-3 embedded, which is similar to standard TTCN-3, has been designed with reusability in mind, providing a multitude of variability mechanisms. This implies that similar aspects shall be taken into consideration for the specification of reusable assets using

TTCN-3 embedded as well. Reusability is inevitably connected to quality, especially when the development of reusable test assets is concerned. Reusability was even identified as one of the main quality characteristics in the proposed quality model for test specifications (Zeiß, Vega, Schieferdecker, Neukirchen, and Grabowski 2007). On the other hand, quality is also critically important for reusability. Reusable assets must be of particularly high quality, since deficiencies in such assets will have a much larger impact on the systems they are reused in. Defects within reusable assets may or may not affect any and every system they are used in. Furthermore, modifications to remedy an issue in one target system, may affect the other target systems, both positively and negatively. Thus, special care should be taken to make sure that the reusable assets are of the necessary quality level. Quality may further affect the reusability in terms of adaptability and maintainability. The assets may have to be adapted in some contexts and they must be maintained to accommodate others, extend or improve functionality, correct issues, or simply be reorganized for even better reusability. If the effort for maintenance or adaptation is too high, it will offset (part of) the benefits of having reusable test assets. Hence, quality is even more important to reuse than reuse is to quality, and thus quality assurance is necessary for the effective development of reusable assets with TTCN-3 embedded.

Furthermore, if an established validation process is employed, the use of validated reusable libraries would increase the percentage of validated test code in the testware, as noted in (Schulz 2008). Establishing such a process on the other hand, will increase the confidence in the reusable assets. A validation process will again involve quality assurance.

In addition to the validation of reusable assets, assessment of the actual reusability of different assets may be necessary to determine the possible candidates for validation and possible candidates for further improvement. This can be achieved by establishing reusability goals and means to determine whether these goals are met (e.g., by defining metrics models or through testability analysis). If they are not met, either the asset is not suitable for reuse, or its implementation does not adhere to the reusability specifications for that asset. In (Mäki-Asiala 2004) two metrics for quantitative evaluation of reusability are illustrated in a small case study. The metrics themselves were taken from (Poulin and Caruso 1993). Further metrics for software reuse are described in (Frakes and Terry 1996). Additional metrics specifically concerning the reuse of test assets may have to be defined. There are ongoing studies that use advanced approaches to assess the reusability of software

components (Sharma, Grover, and Kumar 2009). Such approaches could be adapted to suit the test domain.

While there is a significant body of work on quality assurance for standard TTCN-3 (Bisanz 2006; Neukirchen, Zeiß, and Grabowski 2008; Neukirchen, Zeiß, Grabowski, Baker, and Evans 2008; Zeiß 2006), quality assurance measures for TTCN-3 embedded remain to be studied, as TTCN-3 embedded is still in the draft stage. Similar to standard TTCN-3, metrics, patterns, code smells, guidelines, and refactorings should be defined to assess the quality of test specifications in TTCN-3 embedded, detect issues and correct them efficiently. Based on a survey of existing methodologies, a few examples for quality assurance items for TTCN-3 embedded that are related to reusability will be briefly outlined below.

The main difficulty in the design of TTCN-3 libraries, as identified by (Schulz 2008), is to anticipate the evolution of use of libraries. Therefore, modularization, in the form of separation of concepts and improved selective usage, and a layered structure of library organization are suggested as general guiding principles when developing libraries of reusable assets. Furthermore, in (Schulz 2008) it is also recommended to avoid component variables and timers, as well as local verdicts and stop operations (unless on the highest level, that is, not within a library) when designing reusable behavior entities. The inability to pass functions as parameters, as well as the lack of an effective intermediate verdict mechanism are identified as major drawbacks of the language. TTCN-3 embedded addresses the latter issue in part by allowing modes to be passed as parameters.

Generic style guidelines that may affect the reusability potential of assets are largely transferable to TTCN-3 embedded, for example:

- restricting the nesting levels of modes,
- avoiding duplicated segments in modes,
- restricting the use of magic values (explicit literal or numerical values) or if possible avoiding them altogether,
- avoiding the use of over-specific runs on statements,
- proper grouping of certain closely related constructs, and

- proper ordering of constructs with certain semantics.

In (Mäki-Asiala 2004) ten guidelines for the specification of reusable assets in TTCN-3 were defined. These are concerned with the reusability of testers in concurrent and non-concurrent contexts, the use and reuse of preambles and postambles, the use of high-level functions, parameterization, the use of selection structures, common types, template modifications, wildcards, and modularization based on components and on features. The guidelines are also related to the reusability factors that contributed to their development. The guidelines are rather generic and as such also fully applicable to TTCN-3 embedded.

In (Mäki-Asiala, Kärki, and Vouffo 2006) four additional guidelines specific to the vertical reuse viewpoint are defined. They involve the separation of test configuration from test behavior, the exclusive use of the main test component for coordination and synchronization, redefinition of existing types to address new testing objectives, and the specification of system- and configuration-related data as parameterized templates. Again, these guidelines are valid for TTCN-3 embedded as well. In addition, they can be adapted to the specific features of TTCN-3 embedded. The functional description should be ideally separated from the real-time constraints, and continuous behavior specifications shall be separated from non-continuous behavior.

At this stage, only guidelines based on theoretical assumptions and analogies from similar domains can be proposed. The ultimate test for any guideline is putting it into practice. Apart from validating the effectiveness of guidelines, practice also helps for the improvement and extension of existing guidelines, as well as for the definition of new guidelines.

When discussing guidelines for the development of reusable real-time components, often cited in the literature are the conflicts between performance requirements on one side and reusability and maintainability on the other (Häggander and Lundberg 1998). TTCN-3 embedded, however, abstracts from the specific test platform and thus issues associated with test performance can be largely neglected at the test specification level. Thus, the guidelines shall disregard performance. Ideally, it should be the task of the compilation and adaptation layers to ensure that real-time requirements are met.

The quality issues that may occur in test specifications implemented in TTCN-3 embedded (and

particularly those that affect the reusability) and the means for their detection and removal remain to be studied in more detail. There is ongoing research within the TEMEA project concerning the quality assurance of test specifications implemented in TTCN-3 embedded. As of this writing, there are no published materials on the subject. Once available, approaches to the quality assurance can be ultimately integrated in a test development process and supported by tools to make the development of high quality reusable test specifications a seamless process. Other future prospects include approaches and tool support for determining the reusability potential of assets both during design and during implementation to support both the revolutionary and evolutionary approaches to reuse.

## 1.7 Summary

The ”X-in-the-Loop” testing approach both suggests and presupposes enormous reusability potential. During the development cycle of embedded systems, software models are reused directly (for code-generation), or indirectly (for documentation purposes) for the development of the software. The developed software is then integrated into the hardware (with or without modifications). Thus, it makes sense to reuse tests through all of these development phases. Another hidden benefit is that tests extended in the SiL and HiL levels can be reused back in earlier levels (if new test cases are identified at later levels that may as well be applicable to earlier levels). If on the other hand a strict cycle is followed, where changes are only done at the model level and always propagated onward, this would still reduce the effort significantly, as those changes will have to be made only once. For Original Equipment Manufacturers (OEMs) and suppliers this will also add more transparency and transferability to different suppliers as well (on all levels, meaning reusable tests can be applied to models from one supplier, software from another, hardware from yet another).

The proposed test architecture supports the definition of environment models and test specification on a level of abstraction, which allows the reuse of the artifacts on different test systems and test levels. For the needs of the present domain, we introduced TTCN-3 embedded, an extension of the standardized test specification language TTCN-3, which provides the capabilities to describe test perturbations and assessments for continuous and hybrid systems. Whereas TTCN-3

is a standard already, we propose the introduced extensions for standardization as well. Thus, the language does not only promise to solve the reusability issues on technical level, but also addresses organizational issues, such as long term availability, education, and training.

The ideas presented in this chapter are substantial results of the project "Testing Specification Technology and Methodology for Embedded Real-Time Systems in Automobiles" (TEMEA). The project is co-financed by the European Union. The funds are originated from the European Regional Development Fund (ERDF).

## References

Alur, R., C. Courcoubetis, T. A. Henzinger, and P.-H. Ho (1992). Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pp. 209–229.

Alur, R., T. A. Henzinger, and E. D. Sontag (Eds.) (1996). *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, Volume 1066 of *Lecture Notes in Computer Science*. Springer.

AUTOSAR Consortium (2010). Web site of the AUTOSAR (AUTomotive Open System ARchitecture) consortium. URL: `http://www.autosar.org`.

Bisanz, M. (2006, December). Pattern-based smell detection in TTCN-3 test suites. Master's thesis, ZFI-BM-2006-44, ISSN 1612-6793, Institute of Computer Science, Georg-August-Universität Göttingen.

Bringmann, E. and A. Kraemer (2006). Systematic testing of the continuous behavior of automotive systems. In *SEAS '06: Proceedings of the 2006 international workshop on Software engineering for automotive systems*, New York, NY, USA, pp. 13–20. ACM Press.

Broy, M. (1997). Refinement of Time. In M. Bertran and T. Rus (Eds.), *Transformation-Based Reactive System Development, ARTS'97*, Number 1231 in Lecture Notes on Computer Science (LNCS), pp. 44 – 63. TCS.

Conrad, M. and H. Dörr (2006). Model-based development of in-vehicle software. In G. G. E. Gielen (Ed.), *DATE*, pp. 89–90. European Design and Automation Association, Leuven, Belgium.

ESA-ESTEC (2008). Space engineering: Test and operations procedure language, standard ECSS-E-ST-70-32C.

ETSI (2009a, Febr.). Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 1: TTCN-3 Core Language (ETSI Std. ES 201 873-1 V4.1.1).

ETSI (2009b, Febr.). Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 4: TTCN-3 Operational Semantics (ETSI Std. ES 201 873-4 V4.1.1).

ETSI (2009c, Febr.). Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 5: TTCN-3 Runtime Interfaces (ETSI Std. ES 201 873-5 V4.1.1).

ETSI (2009d, Febr.). Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 6: TTCN-3 Control Interface (ETSI Std. ES 201 873-6 V4.1.1).

ETSI (2009e, July). Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, TTCN-3 Language Extensions: Advanced Parameterization (ETSI Std. : ES 202 784 V1.1.1).

Fey, I., H. Kleinwechter, A. Leicher, and J. Müller (2007). Lessons Learned beim Übergang von Funktionsmodellierung mit Verhaltensmodellen zu Modellbasierter Software-Entwicklung mit Implementierungsmodellen. In R. Koschke, O. Herzog, K.-H. Rödiger, and M. Ronthaler (Eds.), *GI Jahrestagung (2)*, Volume 110 of *LNI*, pp. 557–563. GI.

Frakes, W. and C. Terry (1996). Software reuse: metrics and models. *ACM Comput. Surv. 28*(2), 415–435.

Grossmann, J. and W. Mueller (2006). A formal behavioral semantics for TestML. In *Proc. of IEEE ISoLA 06, Paphos Cyprus*, pp. 453–460.

Häggander, D. and L. Lundberg (1998). Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor. In *ICPP '98: Proceedings of the 1998 International Conference on Parallel Processing*, Washington, DC, USA, pp. 262–269. IEEE Computer Society.

Harrison, N., B. Gilbert, M. Lauzon, A. Jeffrey, C. Lalancette, D. R. Lestage, and A. Morin (2009). A M&S process to achieve reusability and interoperability. URL: `ftp://ftp.rta.nato.int/PubFullText/RTO/MP/RTO-MP-094/MP-094-11.pdf`.

IEEE (1993). *IEEE Standard VHDL (IEEE Std.1076-1993.)*. New York, NY: The Institute of Electrical and Electronics Engineers, Inc.

IEEE (1995). *IEEE Standard Test Language for all Systems–Common/Abbreviated Test Language for All Systems (C/ATLAS) (IEEE Std.716-1995.)*. New York, NY: The Institute of Electrical and Electronics Engineers, Inc.

IEEE (1998). User's manual for the signal and method modeling language. URL: `http://grouper.ieee.org/groups/scc20/atlas/SMMLusers_manual.doc`.

IEEE (1999). *IEEE Standard VHDL Analog and Mixed-Signal Extensions (IEEE Std.1076.1-1999.)*. New York, NY: The Institute of Electrical and Electronics Engineers, Inc.

IEEE (2001). *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE Std.1149.1 -2001)*. New York, NY: The Institute of Electrical and Electronics Engineers, Inc.

ISO/IEC (1998). Information technology - open systems interconnection - conformance testing methodology and framework - part 3: The tree and tabular combined notation (second edition). International Standard 9646-3.

Karinsalo, M. and P. Abrahamsson (2004). Software reuse and the test development process: A combined approach. In *ICSR*, Volume 3107 of *Lecture Notes in Computer Science*, pp. 59–68. Springer.

Kärki, M., M. Karinsalo, P. Pulkkinen, P. Mäki-Asiala, A. Mäntyniemi, and A. Vouffo (2005). Requirements specification of test system supporting reuse (2.0). Technical report, Tests & Testing Methodologies with Advanced Languages (TT-Medal).

Karlsson, E.-A. (Ed.) (1995). *Software reuse: a holistic approach*. New York, NY, USA: John Wiley & Sons, Inc.

Lehmann, E. (2004). *Time Partition Testing Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen*. Ph. D. thesis, TU-Berlin, Berlin.

Lynch, N. A., R. Segala, F. W. Vaandrager, and H. B. Weinberg (1995). Hybrid i/o automata. See Alur, Henzinger, and Sontag (1996), pp. 496–510.

Lynex, A. and P. J. Layzell (1997). Understanding resistance to software reuse. In *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97)*, pp. 339. IEEE Computer Society.

Lynex, A. and P. J. Layzell (1998). Organisational considerations for software reuse. *Ann. Softw. Eng. 5*, 105–124.

Mäki-Asiala, P. (2004). Reuse of TTCN-3 code. Master's thesis, University of Oulu, Department of Electrical and Information Engineering, Finland.

Mäki-Asiala, P., M. Kärki, and A. Vouffo (2006). Guidelines and patterns for reusable TTCN-3 tests (1.0). Technical report, Tests & Testing Methodologies with Advanced Languages (TT-Medal).

Mäki-Asiala, P., A. Mäntyniemi, M. Kärki, and D. Lehtonen (2005). General requirements of reusable TTCN-3 tests (1.0). Technical report, Tests & Testing Methodologies with Advanced Languages (TT-Medal).

Mäntyniemi, A., P. Mäki-Asiala, M. Karinsalo, and M. Kärki (2005). A process model for developing and utilizing reusable test assets (2.0). Technical report, Tests & Testing Methodologies with Advanced Languages (TT-Medal).

Modelica Association (2010). Modelica - a unified object-oriented language for physical systems modeling. URL: `http://www.modelica.org/documents/ModelicaSpec30.pdf`.

Neukirchen, H., B. Zeiß, and J. Grabowski (2008, August). An Approach to Quality Engineering of TTCN-3 Test Specifications. *International Journal on Software Tools for Technology Transfer (STTT), Volume 10, Issue 4. (ISSN 1433-2779)*, 309–326.

Neukirchen, H., B. Zeiß, J. Grabowski, P. Baker, and D. Evans (2008, June). Quality assurance for TTCN-3 test specifications. *Software Testing, Verification and Reliability (STVR), Volume 18, Issue 2. (ISSN 0960-0833)*, 71–97.

Parker, K. P. and S. Oresjo (1991). A language for describing boundary scan devices. *J. Electron. Test. 2*(1), 43–75.

Poulin, J. and J. Caruso (1993). A reuse metrics and return on investment model. In *Proceedings of the Second International Workshop on Software Reusability*, pp. 152–166.

SCC20 ATML Group (2006). IEEE ATML specification drafts and IEEE ATML status reports.

Schieferdecker, I., E. Bringmann, and J. Grossmann (2006). Continuous TTCN-3: testing of embedded control systems. In *SEAS '06: Proceedings of the 2006 international workshop on Software engineering for automotive systems*, New York, NY, USA, pp. 29–36. ACM Press.

Schieferdecker, I. and J. Grossmann (2007). Testing embedded control systems with TTCN-3. In R. Obermaisser, Y. Nah, P. Puschner, and F. Rammig (Eds.), *Software Technologies for Embedded and Ubiquitous Systems*, Volume 4761 of *Lecture Notes in Computer Science*, pp. 125–136. Springer Berlin / Heidelberg.

Schulz, S. (2008). Test suite development with TTCN-3 libraries. *Int. J. Softw. Tools Technol. Transf. 10*(4), 327–336.

Sharma, A., P. S. Grover, and R. Kumar (2009). Reusability assessment for software components. *SIGSOFT Softw. Eng. Notes 34*(2), 1–6.

Suparjo, B., A. Ley, A. Cron, and H. Ehrenberg (2006). Analog boundary-scan description lan-

guage (ABSDL) for mixed-signal board test. In *International Test Conference*, pp. 152–160.

TEMEA (2010). Web site of the TEMEA project (Testing Methods for Embedded Systems of the Automotive Industry), founded by the European Community (EFRE). URL: `http://www.temea.org`.

The MathWorks (2010a). MATLAB™ - the language of technical computing. URL: `http://www.mathworks.com/products/matlab/`.

The MathWorks (2010b). Web site of the Simulink™ tool - simulation and model-based design. URL: `http://www.mathworks.com/products/simulink/`.

The MathWorks (2010c). Web site of the Stateflow™ tool - Design and simulate state machines and control logic. URL: `http://www.mathworks.com/products/stateflow/`.

Tripathi, A. K. and M. Gupta (2006). Risk analysis in reuse-oriented software development. *Int. J. Inf. Technol. Manage. 5*(1), 52–65.

TT-Medal (2010). Web site of the TT-Medal project - Tests & Testing Methodologies with Advanced Languages. URL: `http://www.tt-medal.org/`.

Vector Informatics (2010). Web site of the CANoe tool - the development and test tool for can, lin, most, flexray, ethernet and j1708. URL: `http://www.vector.com/vi_canoe_en.html`.

Zeiß, B. (2006, March). A Refactoring Tool for TTCN-3. Master's thesis, ZFI-BM-2006-05, ISSN 1612-6793, Institute of Computer Science, Georg-August-Universität Göttingen.

Zeiß, B., D. Vega, I. Schieferdecker, H. Neukirchen, and J. Grabowski (2007, March). Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications. In *Software Engineering 2007 (SE 2007). Lecture Notes in Informatics (LNI) 105. Copyright Gesellschaft für Informatik*, pp. 231–242. Köllen Verlag, Bonn.