Testing Hybrid Systems with *TTCN-3*

vorgelegt von
Dipl.-Inform. (FH) Jürgen Großmann
geb. in Göttingen

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr. Ing. –

genehmigte Dissertation

Berichter:	Prof. Dr.-Ing. Ina Schieferdecker
	Prof. Dr.-Ing. habil Prof. e.h. Dr. h.c. Radu Popescu-Zeletin
	Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Vorsitzender:	Prof. Dr. Axel Küpper

Tag der wissenschaftlichen Aussprache: 19.05.2014

Berlin 2014
D 83

Zusammenfassung

Die Entwicklung sicherer, komfortabler und flexibel einsetzbarer technischer Systeme wird – mehr denn je – durch Software und die damit zusammenhängenden Softwareentwicklungsprozesse, -techniken und -methoden bestimmt. Die Qualitätssicherung solcher Systeme hat inzwischen einen Grad an Komplexität erreicht, der nur durch den Einsatz dedizierter Qualitätssicherungstechniken und -methoden zu beherrschen ist. Eingebettete Systeme sind zunehmend vernetzt und setzen Kommunikationsprotokolle ein, die in ihrem Einsatz bisher auf den Bereich der Telekommunikation bzw. das Internet beschränkt waren. Im Gegensatz zu herkömmlichen Softwareanwendungen sind eingebettete Systeme jedoch weiterhin häufig hybride Echtzeitsysteme, die über Sensoren und Aktuatoren mit einer physikalischen Umgebung in Wechselwirkung treten. Sie müssen einerseits kontinuierliche Datenströme verarbeiten, die ihnen ein möglichst exaktes Abbild ihrer Umgebung vermitteln, andererseits nehmen sie Kontrollaufgaben wahr, deren Ergebnisse als diskrete Ereignisse in einem verteilten Systemverbund propagiert werden sollen. Speziell in der Automobilindustrie ist die Anzahl der Steuergeräte in einem Fahrzeug auf 50 bis 80 Stück angestiegen. Eine etablierte Testtechnologie, die einerseits den neuen Anforderungen aus den Bereichen der Telekommunikation gerecht wird und darüber hinaus den bestehenden Anforderungen eingebetteter Systeme nachkommt, gibt es bisher nicht.

In dieser Arbeit wird mit *TTCN-3 embedded* eine Testtechnologie entwickelt, die speziell für das Testen vernetzter, hybrider Systeme geeignet ist und die bestehenden Qualitätssicherungsprozesse in der Automobilindustrie effektiv verbessern kann. Ausgehend von etablierten Formalismen aus der Theorie hybrider Systeme wird eine ausführbare Testspezifikationssprache abgeleitet, die eine intuitive Spezifikation automatisierter Tests für Systeme der Automobilindustrie ermöglicht. Die Spezifikation dedizierte Schnittstellen zur Stimulation, Auswertung und Zeitkontrolle sorgen für eine nahtlose Integration der Sprache in industrielle Werkzeugketten. Die Testspezifikationssprache und ihre Schnittstellendefinitionen sind als Erweiterung des *TTCN-3* Standards konzipiert. Die Integration in den *TTCN-3* erlaubt einerseits eine einfache und fundierte Industrialisierung der Ideen und schafft andererseits die notwendige Integration mit den Konzepten zum Testen diskreter nachrichtenbasierter Systeme, da diese bereits in *TTCN-3* vorhanden sind. Die Grundlagen von *TTCN-3 embedded* wurden im Forschungsprojekt TEMEA entwickelt[1].

[1]Das Projekt TEMEA [112] wurde von der EU kofinanziert. Die Mittel stammen aus dem Europäischen Fonds für Regionale Entwicklung (EFRE).

Abstract

The development of safer, more convenient and flexible technical systems is now, The development of safer, more convenient and flexible technical systems is now, more than ever, determined by software systems and their related development processes, techniques and methods. Technical systems have reached such a high level of complexity that quality can only be assessed and ascertained through the use of testing methods that have been developed for this specific purpose. Embedded systems are now increasingly networked and use communication protocols that were previously limited to the field of telecommunications and the Internet. Embedded systems can be characterized as so called hybrid real-time systems that directly interact with their physical environment through sensors and actuators. These systems are required to perform two tasks simultaneously: they must assess and process continuous streams of data that give them the most accurate representation of their current environment, and they must perform control tasks on basis of the continuous input streams and feed the results of this tasks as discrete events in a distributed network. In the automotive industry, the use of control devices has increased dramatically in recent years: today's average vehicle contains 50 - 80 networked control devices. There is, however, no established test technology which can meet the new and ever-evolving requirements of these modern, networked embedded systems in the way that traditional telecommunication systems nor traditional embedded systems used to do.

In this thesis, a test technology, namely *TTCN-3 embedded*, which supports and facilitates the testing of networked, hybrid systems is developed. *TTCN-3 embedded* will improve quality assurance processes in the automotive industry. Using the theory of hybrid systems and other well-established formalisms, an executable test specification language of *TTCN-3 embedded* is developed. This test specification language, in turn, allows for intuitive and practical specification of automated tests for hybrid real-time systems in the automotive industry. Dedicated interfaces for stimulation, evaluation and time control allow for a seamless integration with industrial tool chains. The test specification language and its interfaces are designed as an extension of the *TTCN-3* standard. The integration of this new language into the *TTCN-3* standard enables a simple, fast and profound industry-wide application and facilitates the integration of hybrid-system testing with the discrete message-based systems testing that already exists. The basis of *TTCN-3 embedded* has been developed in the context of the project TEMEA[2].

[2]The TEMEA project [112] has been co-financed by the European Union via the European Regional Development Fund (ERDF).

Acknowledgements

I would like to thank Prof. Fanny-Michaela Reisin for guiding me into the world of research and scientific work, Prof.Ina Schieferdecker for her friendly advice and constructive feedback to this work and my parents for providing me with all the necessary basics and allowing me to realize my own potential.

Contents

List of figures

List of tables

Chapter 1

Introduction

Technological progress has led to more complicated and viable products. Since the 19th century, industries have used division of labour and specialisation as strategies to bundle expertise and efficiently organise the production of goods. These production strategies continue today and, over time, they have led to a component-based development paradigm, in which highly specialised suppliers provide only parts of the complete product. These parts are then assembled and integrated by one central integrator that is responsible for the functionality and quality of the end product. For instance, in the automotive industry, automotive component suppliers — or suppliers of the suppliers — deliver hardware and software that are specialised subsystems of the whole automobile (e.g. multimedia electronics, transmissions, shock absorbers, etc). These subsystems are then assembled, integrated and tested by the Original Equipment Manufacturer (OEM). To keep the development and testing of these complex, widely-dispersed manufacturing processes efficient and manageable, an integrated and seamless approach is required. Such an approach has yet to be defined and developed. In the case of testing, such an approach would need to address issues of test exchange, autonomy of infrastructure, methods, platforms and the re-use of tests. This approach would need to be constructed using a domain-specific test language. It would need to be based on a technological basis that unifies tests of communicating, software-based systems in all of the automotive sub domains (telematics, power train, body electronics etc.). Such an approach would therefore be able to unify the infrastructure, definition and documentation of tests. This thesis will introduce such an approach: a high-level test-specification infrastructure based on *TTCN-3*.

1.1 Background and motivation

In recent years, the number of software applications and electronic control systems in modern vehicles has increased dramatically. Software and electronic control systems are used in nearly every feature that a modern vehicle provides and they currently constitute a high percentage of the automotive

industry's value chain. A vehicle built around 1996 had about half a dozen distributed control units, and already had more processing power and memory than the spacecraft that landed on the moon in 1969. Today's vehicles have dozens of control units, thousands of functional features and software with tens of thousands of lines of code.

Future generations of vehicles will be integrated into a comprehensive communications infrastructure which will enable the exchange of data between individual vehicles, groups of vehicles and traffic control centres. These communication-enabled vehicles will be able to provide real-time information about their speed, position, general status and routing. In return, intelligent transport software applications, located either in the vehicles themselves or in traffic control centres, will be able to compute and evaluate these reports and provide drivers with pertinent information: warnings about dangerous situations (e.g. obstacles on the road, dangerous weather conditions, traffic jams, etc), as well as helpful updates about traffic flow and location-specific data (e.g. free parking spaces, local traffic routing, etc). This so-called "car-to-X (C2X)" communication infrastructure, in combination with intelligent telematics and driver assistance systems, will make individual and public transport more efficient, comfortable and secure for all concerned. For C2X to function properly, all of the constituent components must be reliable and high quality. Ensuring the quality of these systems presents a new set of distinct challenges, because the combination of systems is unique and unprecedented, and because the testing systems and the specification of test cases have particular requirements.

In general, embedded systems play an increasing role in complex control functions in many industrial domains. In particular, software-based control systems have specific characteristics, which —at least in their combination— are unique. Embedded systems must: interact with their environment using sensors and actuators, supervise discrete control flows, obtain and process simple and complex structured data, communicate over different bus systems, and meet high safety and real-time requirements. For all of these characteristics to function reliably and efficiently within the control system as a whole, a variety of specific conditions must be met. The development and quality assurance of such systems — especially when they are distributed, real-time systems — thus presents a highly complex, challenging structure that is not yet efficiently managed. This structure continues to produce significantly rising development costs.

While different, model-based development processes and methods for embedded systems exist, an industry-wide, recognised test infrastructure for the analysis and evaluation of these systems is missing. Such a test infrastructure would lead more easily and securely to high-quality, safe and reliable

systems. To be applicable to state-of-the-art development processes, such a test infrastructure would have to address the specific characteristics of embedded systems and would have to neatly integrate in the existing processes and development infrastructures of the industry.

TTCN-3 [35] has the potential to serve as such an infrastructure. It provides concepts for local and distributed testing, as well as for platform- and technology-independent testing. A *TTCN-3*-based test solution can be adapted to specific testing environments and modified to test particular systems via an open test execution environment with well-defined interfaces for adaptation. Control systems in the automotive domain, however, can be characterised as hybrid systems because they encompass both discrete and continuous behaviour (in time). Discrete signals are used for communication and coordination between system components, while continuous signals are used to monitor and control the components and the system environment via sensors and actuators. An adequate test technology for control systems needs to be able to control, observe and analyse the timed, functional behaviour of both of these systems. This behaviour is characterised by a set of discrete and continuous input and output signals and their relationships to each other. While the testing of discrete controls is well-understood and available in *TTCN-3*, concepts for specification-based testing of continuous controls and for the links between discrete and continuous system parts are not available. *TTCN-3* especially lacks concepts for specifying tests for continuous and hybrid behaviour.

1.2 Contribution of this thesis

This thesis aims to bridge the gap that exists between the concepts and technologies which are normally used to test message-based discrete systems and those which are used to test hybrid and continuous systems. The combination of hybrid and continuous systems is especially unique and will be relevant in the testing of future systems that are conceptually continuous or hybrid and make extensive use of message-based communication in their interactions with other systems. To ensure applicability and industrial use, the concepts and technologies in this thesis are tailored to the requirements and processes of the automotive industry. The main contribution of this thesis to the field is a language, namely *TTCN-3 embedded*, which allows the testing of both message-based discrete systems and hybrid or continuous systems. *TTCN-3 embedded* is based on the already standardized and industrialized solution *TTCN-3*. *TTCN-3* already contains mature concepts and constructs for testing message-based systems; *TTCN-3 embedded* expands

upon this by including concepts for continuous and hybrid systems and integrating them neatly into the already-existing language. The motivations for *TTCN-3 embedded*, its specifications and a short evaluation are all included in this thesis. Secondary contributions of this thesis to the field include: a set of selected concepts that can be used to specify and execute tests for continuous and hybrid systems, and a compiler and runtime-infrastructure that is able to gather *TTCN-3 embedded* programmes and execute them in the industrial-grade testing environments of the automotive industry. Since its creation, *TTCN-3 embedded* has been standardized by the ETSI.

The work of this thesis began as part of the TEMEA project and is based on a loose list of requirements that roughly capture the needs of the automotive industry with respect to test environments and test-specification languages. Based on these given requirements, dedicated concepts and principles for testing continuous and hybrid systems were examined and selected. This selection was accomplished via a survey regarding existing specification concepts and languages that are currently being used in the field of hybrid systems testing. In this survey, special attention was paid to gathering information about testing languages and infrastructures from the automotive industry. The next step was to describe these concepts in an abstract way and then to integrate them syntactically and semantically into *TTCN-3*. Both kinds of integration are specified formally in the thesis: the syntactical integration by means of a BNF grammar and the semantic integration by means of an ASM specification. The formal specifications have been used to identify syntactical and semantical gaps and ambiguities.

In order to demonstrate the concepts' applicability and to show their effectiveness and suitability to the special requirements for testing continuous and hybrid systems, a prototype runtime and compiler infrastructure was developed. This prototype allowed *TTCN-3 embedded* to be used to test models and ECUs in several smaller case studies. The results of these case studies, as well as further findings from the implementation of the compiler infrastructure and its integration in simulation and testing environments from the automotive industry, provided feedback that was then used to iteratively refine *TTCN-3 embedded*. Main ideas of this thesis have been already published in [44].

1.3 Structure of this thesis

This thesis is divided into an introduction, conclusion and seven content chapters. After the introduction, the first chapter, titled Chapter 2, discusses the state-of-the-art processes and tools that are currently used to test

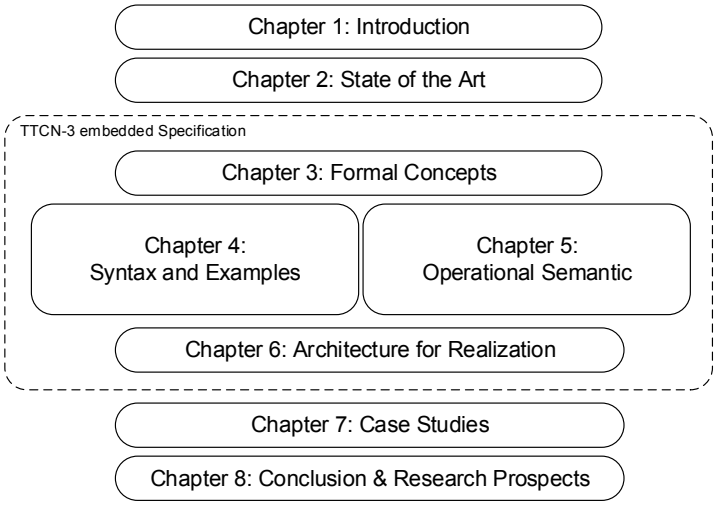

Figure 1.1: The structure of the thesis

software-base automotive control systems. This chapter includes a survey of the concepts, languages and tools that can be used to test continuous and hybrid systems. It ends with a short summary about the state of this technology within the industry and explains the motivations behind this thesis. The next chapter, Chapter 3, describes a set of formal concepts which are necessary for the testing of continuous and hybrid systems and, therefore, have been chosen as the base for the language extension. Chapter 4 follows and includes the specification of *TTCN-3 embedded*. Based on the required concepts outlined in Chapter 3, the individual constructs that enhance the capabilities of *TTCN-3* are defined and then presented. The syntactical structure of each construct is shown with examples and the semantics of each structure are intuitively defined. Chapter 5 uses an ASM specification to create a formal definition of the behavioural semantics of *TTCN-3 embedded*. Chapter 4 and Chapter 5 are strongly related. Both provide specialised perspectives on the language extension and, taken together, they form a comprehensive theory of *TTCN-3 embedded*. 4 introduces the individual constructs, and 5 specifies the operational semantics. Chapter 4 contains references to Chapter 5, which relates the *TTCN-3 embedded* constructs to the ASM rules that define their operational semantics. This enables users to switch between both perspectives.

Chapter 6 presents the runtime infrastructure for *TTCN-3 embedded*. It begins by defining the extensions to the standardized runtime interfaces of *TTCN-3*. It continues by showing how the extended runtime interfaces could be used to integrate *TTCN-3 embedded* into existing simulation and

test execution infrastructures used by the automotive industry. Chapter 7 describes two case studies that used *TTCN-3 embedded* and discusses the outcomes of these studies. Chapter 8 summarizes the thesis and considers further areas of potential study, with particular emphasis on future challenges anticipated within in automotive industry.

Chapter 2

Testing continuous and hybrid automotive systems

The testing of continuous and hybrid systems presents new challenges for established testing methods, languages and concepts. Already, a number of promising approaches and solutions are developing in the industrial application of and the scientific discourse surrounding testing methods, languages and concepts. The following chapter is a summary of some of the basic concepts that (implicitly or explicitly) constitute the basis for testing hybrid systems and automotive control systems. The chapter starts with a short introduction to basic terms and techniques of testing. A review section follows, in which the state of the art in testing electronic control units within the automotive industry is discussed[1]. The chapter continues with a review of the state of the art in concepts, methods, languages and tools for testing continuous and hybrid systems. In this section, the terms continuous system and hybrid system are more closely characterized and the terms *signal* and *stream* are introduced. Finally, the basic specification techniques for continuous and hybrid systems — *algebraic equations*, *difference equations*, *differential equations* and *hybrid automata* — are outlined and an overview which summarizes the formalisms and tools currently used for testing is presented.

2.1 Testing software-based systems

Testing, and software testing in particular, is one of the most important analytic measures of quality assurance. In contrast to other methods, testing allows for the verification of a software system throughout its development and also under conditions (as closely as possible) approximate the intended use of the system. Systematic, thoughtful design of test cases and compliance with relevant test suites are two essential components of a good quality test.

[1]The automotive industry is one of the most relevant fields of practice for testing concepts, particularly those which address the special features and properties of continuous and hybrid systems. The industrial requirements arising from within the automotive industry act as landmarks and as challenges for a proof of concept for this thesis' results.

The definition of a test case determines the category and scope of the test, and the conformity of a test case to the appropriate test suites shapes its completeness and industrial significance.

Software testing, however, cannot prove that a software system is free of errors. It can only determine if certain test cases were executed successfully. As E.W. Dijkstra wrote, "Program testing can be used to show the presence of bugs, but it never shows their absence!" [29]. This is because, with the exception of very simple programmes, exhaustive testing (i.e the testing of all program functions, with all possible input data, in all possible combinations) is nearly impossible. Different systematic test strategies and approaches, therefore, must be employed to effectively test a system. Such testing strategies are never complete, because they do not involve all possible input data and all possible combinations; a well-designed and diverse testing strategy, however, can offer acceptable levels of certainty about a software's function.

2.1.1 Testing techniques and approaches

In [75, 76] a large number of testing techniques and approaches are identified, categorized and described. The authour distinguishes between static and dynamic testing techniques thus: static testing techniques are involve software which is not executed during the test; dynamic test techniques, on the other hand, require that the software be executed during testing. In this study, only dynamic testing techniques are addressed. In this thesis, the term "testing" without further elaboration always refers to dynamic testing.

The main dynamic testing techniques are structure-based testing and functional testing. Structural testing (which is also known as white-box testing) focuses on the structure of a program and its data, while functional testing focuses on the functional requirements and specifications of a system. Due to the real-time requirements of embedded systems, the testing of non-functional properties like timing is of increasing industrial interest. Both functional and non-functional tests are often carried out as black-box tests.

White-box, or structure-based, tests are constructed using knowledge about the internal structure of the system under test. They are often executed in an instrumented setup that provides feedback on the portions of code covered in the test. Typical white-box, or structure-based, testing techniques are control-flow-oriented techniques such as statement coverage tests, condition coverage tests, branch coverage tests and data-flow-oriented testing techniques.

Black-box tests, on the other hand, are developed without prior knowledge

of the internal structure of the system under test. Rather, they are designed on the basis of development documents. In practice, black-box tests are not usually developed by software developers, but by technically-oriented testers, specific departments or testing teams. Black-box testing techniques also include requirements-based testing (the testing of specific requirements) and stochastic testing (in which statistical information forms the test base).

Testing levels

According to Weyuker [128], at least three stages of accuracy testing are absolutely necessary in order to achieve a reliable software-based system:

- module or unit testing, in which individual and self-contained software entities are tested,
- integration testing, in which the subsystems formed by the integration of the individually-tested modules or units are tested as entities, and
- system testing, in which the software system as whole is evaluated in a real-world scenario. This might include functional testing as well as non-functional testing, such as stress testing, performance testing and security testing, if such requirements are required by the system.

The module, or unit-testing, step addresses the testing of individual, self-contained software entities, which are then integrated at a later date (see integration test). The aim of the module testing stage is to find any bugs as early as possible. In addition, the functionality of the modules can be tested more easily while they are still separate, as opposed to when they have already been integrated.

Integration testing occurs during the integration of modules or units into larger entities. Integration testing aims to detect failures in the interoperability or compatibility of components. Integration testing evaluates the correctness of interfaces, messages and protocols, and checks the functionality of the underlying communications infrastructure. This stage of testing assesses some structural aspects (the correctness of interfaces and messages), but is focused mainly on the aspects of module and unit functionality that are directly or transitively based on interaction with other modules and units. Integration testing also considers non-functional aspects like timing and robustness. For the integration of different components, different integration strategies are established [12, 76].

- Vertical integration addresses the composition of entities that are hierarchically structured and follow a specific hierarchical order in their execution (e.g. inheritance structures, decomposition of larger systems via subsystems, well-defined communication relationships with a clear consumer-provider relationship, etc).

- Horizontal integration addresses the integrated modules or units as loose, non-hierarchically-coupled entities. In essence, it treats these modules like objects in an object-oriented environment. The interdependencies between these components are often not specified directly and are, in most cases, only visible during runtime (e.g. objects that are coupled together via method calls).

During system testing, the logical architecture and user requirements of the entire system are tested. All previous integration and testing phases can be performed on a test bench or in a different environment, but system testing is carried out using original hardware and, if possible, in the original environment. Process models such as the V-model [31] distinguish between system testing and acceptance testing. While system testing seeks to verify a system's properties with respect to its specifications, acceptance testing seeks to validate the system realization in light of its user requirements. In acceptance testing, the system is checked to assess its practical applicability from the perspective of a customer or potential end user.

In general, the term 'system' can be interpreted from different angles or vantage points. Suppliers and subcontractors may consider a system to be an individual controller that acts in combination with relevant sensors and actuators in a vehicle; for a car manufacturer, however, 'system' can refer to the entire vehicle. Depending on the definition of the SUT, a test object may be subject of multiple integration processes and may be tested several times.

2.1.2 Test automation

Test automation refers to the standardization of all testing activities: planning, design, implementation, execution and evaluation [124]. In current industrial practice, test automation is mostly understood as the automation of test execution. There are also attempts, however, to standardize other testing activities. In this context, the most promising approach is the automation of test design and test implementation. This approach is now widely designated as 'model-based testing'. Model-based testing is briefly outlined in Section 2.1.3.

Automated test execution has many advantages. It allows for the repeated execution of tests (e.g. every day at specific times), which enables much more frequent test executions than manual testing. Higher testing frequency allows for a more accurate measurement of software quality, because a wider picture of error rates is produced via multiple test runs using the same test suite. Furthermore, automated test execution offers the possibility of shifting lengthy test sessions to more convenient times of day (e.g. at nighttime or in the early morning). This means that testing sessions need not interfere with other software development activities. In general, the benefits of test execution automation are: efficient test execution, reproducible test results with exactly the same test runs, repeatability of test runs with no extra effort, reduction of personnel and material costs, and possible reuse of automated test procedures for multiple test objects. Automated test cases can be executed regardless of the expertise of employees and thus are reliably repeatable. Test runs can be repeated as often as needed and each repetition performs exactly the same test run. Each repetition of an automated test requires minimal effort and an automated test can also be repeated after a alteration or extension of the test object.

On the other hand, the initial effort to create and prepare automated tests is much higher than that which is required for manual testing. In order for tests to be automated, an explicit, detailed and formal description of the test knowledge is necessary. This requires formal definition of the test data and of the necessary test procedures. Currently, there are a number of languages, methods and tools available that support the specification, realization and execution of automated tests. The exact nature of these languages, methods and tools is, however, highly dependent on the application domain.

Unit testing environments for high-level programming languages such as Java, C# [129], and, particularly for the telecommunications industry, the test automation language *TTCN-3* [65], are known and have been developed. An overview of the test automation environments in the automotive industry is presented in Section 2.2.

In summary, the application of automated tests is highly relevant for industry-grade quality assurance processes in complex systems. Although the creation of automated tests is more time-consuming in the short term, it can save a significant amount of time in the long run, especially in the case of multiple test repetitions. Furthermore, repeatability and determinism are essential characteristics of automated tests, and these characteristics are indispensable to modern testing processes. Promising approaches to test automation are being researched and put forward, but, in the automotive industry as in many other domains, there is still no test automation approach which both fulfills all the domain-specific requirements and provides a series

of general concepts that allow for the exchange and reuse of tests between different manufacturers and suppliers.

2.1.3 Model-based testing

Model-Based Testing (MBT) is one of the most promising approaches to the automation of test design and test implementation. The automation of test execution replaces the manual execution of a test via the application of test scripts which allow for an automatic test stimulation and evaluation (see 2.1.2). Model-based testing, on the other hand, replaces manual test design and test implementation with programmes that generate tests using abstract specifications. So, instead of test cases being created manually, they are generated automatically from a model or a set of models. A model, in this case, is usually an abstract and partial representation of either the system under test or its environment. In order to derive relevant test cases, however, these models are usually annotated with testing directives that model individual test objectives and generate the test itself. The test cases derived from this model are mostly functional tests that have the same level of abstraction as the model. These test cases form an abstract test suite that can then be mapped and configured to a specific test platform. For any given SUT, various kind of models might exist.

MBT approaches, like test automation execution approaches, differ in terms of their application domains. There are a large number of scientific and industry-specific papers that deal with MBT approaches and describe the individual requirements of the different industrial domains. According to [42], MBT tools (and thus MBT approaches) can be categorized as Model-based Test Cases, Test Data Editors and Model-based Test Case Generators. Model-based Test Cases and Data Editors model individual abstractions and/or concrete test cases by interpreting the system specification. Model-based Test Case Generators, on the other hand, are tools which support the algorithmic generation of test cases, test models or even entire test suites from one model. Editors often provide a specific notation, which makes it easier for a tester to model the test cases. Said test cases are then refined or interpreted, so that they can be executed against the SUT. Test case generators, as the name implies, generate test cases automatically via the use of a traversal algorithm, which is based on configurable coverage criteria such as structural model coverage and requirements coverage. Once the test cases are derived automatically, test generators have to fulfill tasks similar to those of test editors, i.e. mapping test data and abstract test suites to the technical interfaces of the SUT.

A good overview of some different MBT approaches is provided by Neto et. al. [28]. In this paper, the authors evaluate more than 70 papers and compare the approaches described therein with regard to their modelling paradigms, tool support, level of automation, etc. [122] provides a taxonomy of model-based testing approaches that categorizes these approaches according to the kind of specification used for the modelling, the kind of test generation approach and the kind of test execution. Last but not least, a number of MBT tool vendors and industrial users have developed a standard to unify MBT terminology and have begun to define a common set of concepts required by MBT tools [39].

In summary, MBT has evolved in recent years from an academic field to a mature industrial-grade technology. According to [122], there is empirical evidence that MBT approaches are very effective in detecting failure. MBT also reduces test creation and maintenance costs, and the use of MBT tools enhances the levels of documentation and communication between team members, because of the use of models. The generated test suite and its traceability back through its development provide a clear and unified view of both the System Under Test (SUT) and the test.

Though MBT technologies are now well-developed and MBT approaches have matured, there are still some related fields that are ripe for research. In particular, there are still too few domain-specific solutions that meet the particular requirements of particular domains, such as the automotive industry. For example, testing systems with continuous and mixed signals presents complex requirements that are not yet covered sufficiently. The available solutions are still in research stages or are limited with respect to the grade of automation. In addition, there are still unresolved issues relating to the testing of non-functional requirements. This is especially true for areas like security and usability, but also for more established testing areas such as real-time testing and performance testing. Last but not least, the tool support systems for managing models and model transformations still cannot cope with the complexity of today's industrial-grade development processes.

2.2 Testing automotive control systems

Similarly to other aspects of real-time and safety-critical systems, test processes in the automotive industry are tool-intensive and are affected by technologically heterogeneous test infrastructures. Furthermore, the whole development process is widely distributed and highly fragmented. The Original Equipment Manufacturer (OEM), i.e. the system integrator and solution provider, is responsible for specification and integration at the overall level,

whereas the softwares and hardwares of the individual electronic control units (ECUs[2]) are normally provided by different suppliers.

In recent years, the code development process has become noticeably more effective, automated and abstract via the introduction of model-based specifications in development and the establishment of powerful code generators. Because executable models are now readily available, tests and analytic methods can be applied early and integrated into subsequent development. The positive effects of this — early error detection and early bug fixing — are obvious.

2.2.1 Processes and test processes in the automotive domain

The correlation between system development activities and testing activities is best described by a process model such as the V-model. The V-model was published in the early 1990s by the Federal Ministries for Defense and the Interior. It is a process model for IT projects. In 1997, it was revised to include new development methods and approaches [31]. Since then, it has found widespread application in both industry and academia. The primary use of the V-model is in heavyweight development processes, such as systems development in the automotive and aircraft industry. Figure 2.1 shows the activities of the system and software development according to the V-model.

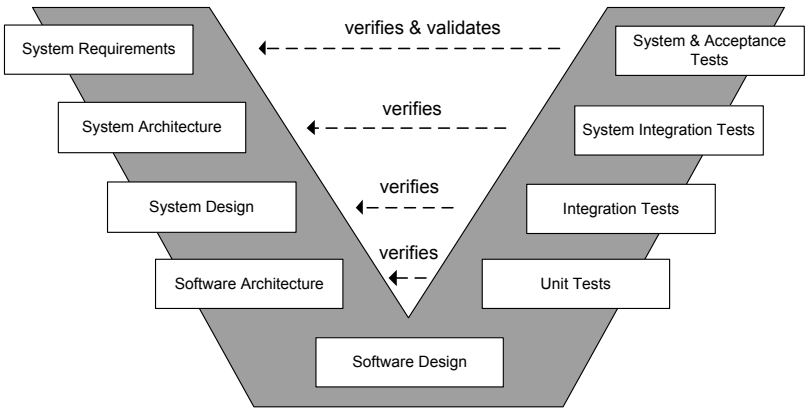

Figure 2.1: The V-Model 97

For any model to remain relevant and up-to-date, it needs to be highly

[2]In this thesis, the term electronic control unit (ECU) is used to refer to systems that are a single piece of hardware. This contrasts with the already-introduced term "control system", which may refer to compound systems, i.e. systems that consist of multiple control units.

adaptable and able to respond to the needs of today's industrial software development projects. To this end, a new edition of the V-model was published in 2005 under the name V-Model XT (XT = eXtreme Tailoring) [57]. The V-Model XT addresses recent practical trends in software development (e.g. agile and incremental approaches) and effectively has replaced the old V-model. It is designed as a guide for the planning and execution of development projects, and it has been developed with consideration of the entire system lifecycle in mind.

Development and testing of embedded systems is often carried out in several stages. Generally, in the first step, the system's required behaviour is modelled and simulated on an ordinary PC. If the system's behaviour can be classified as correct, either code generated from the model itself or separately-coded software is integrated into a so-called prototype. The prototype's hardware is then gradually replaced with the original target hardware, until the prototype has transformed completely into the desired final product.

At each stage of development, the software system undergoes a complete cycle of the V-model, including all three phases (designing, building and testing). In principle, all available functions should be testable at all three development stages. Some properties, however, are only verified in the latter development phases because they are not available or testable as a pure software model or on a prototype board. The motivation for keeping all functions testable throughout the entire development process is one of increased efficiency: making changes to and fixing bugs in the model or prototype is significantly cheaper and faster than doing so with the final product. Early testing offers the possibility of finding big errors before a final product has been generated. According to Breakmann and Notenboom [18], current practice in the development of automotive control units is better represented by a multiple V-model (see Figure 2.2).

The model showing three subsequent V-shaped development cycles, however, is a simplification of current development processes for automotive ECUs. In reality, these processes are much more complex. System development is a multidisciplinary project, and one in which software and hardware development take place parallel to and independently of one another. These complex systems break down, in real terms, into individual components, and each component is often developed by a separate subcontractor or supplier. Furthermore, some components involve suppliers working alongside each other and only integrating subsequently. For each component in a complex system, a V-model can be applied.

The introduction of the AUTOSAR standard is expected to establish stand-alone software packages from independent vendors and thus will introduce new roles for system suppliers and car manufacturers. In the future, not

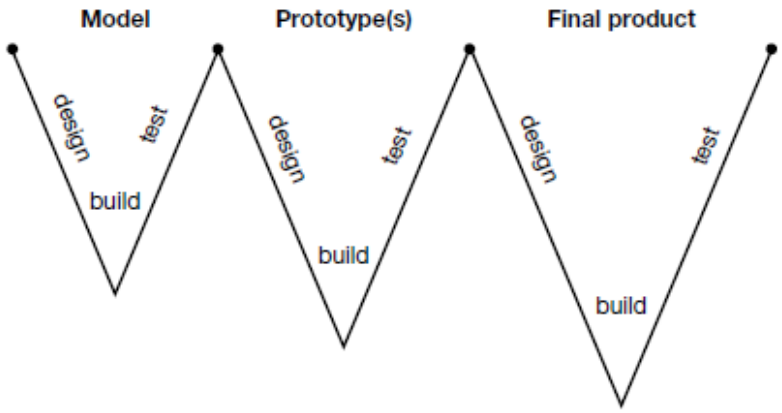

Figure 2.2: Subsequent V-Models for embedded system development according to [18]

only the OEMs but their suppliers, too, will increasingly be forced to deal with the integration of hardware and software, as well as with integration testing.

2.2.2 Test platforms in the automotive domain

Current industrial practice requires that, in order for an embedded control system to be considered adequate for industry application, it must pass several kinds of tests at different levels of integration and maturity. Tests that deal with the integration of the complete vehicle system are mainly the responsibility of the OEM. These tests address the interaction between the control units, the vehicle communication infrastructure and, last but not least, the performance of the complete vehicle system. Tests at the ECU level, on the other hand, are generally the responsibility of the respective suppliers. These tests verify the functionality and electronic characteristics of the ECU. These characteristics are, in most cases, software-driven.

ECUs have real-time requirements and closely interact with their physical environment; correct ECU testing, therefore, must directly consider environmental feedback, as well as feedback from the System Under Test (SUT), in order to generate adequate test input data and to calculate the test verdict. To accomplish this task, so-called closed-loop architectures [87, 78, 70] are often employed. Closed-loop systems incorporate into themselves the part of the feedback control system that is being verified; thus, it can be said that this aspect is 'in the loop' of the system. In ECU testing, feedback from the

environment is also essential and is usually simulated by so-called *environment models*, which are directly linked with the SUT. Aside from scenarios wherein technical differences exist because different systems are under test (an ECU, the ECU's software or a model of it), ECU testing is based on a common architecture, the so-called *closed-loop architecture*.

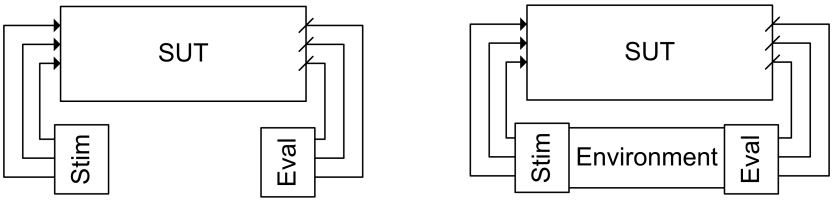

Figure 2.3: Open loop architecture and closed loop architecture.

Unlike systems with open-loop architecture, a dynamic system in a closed-loop architecture is tested in a feedback control loop. This means that the input data for the SUT are calculated directly by the environment model, and the environmental model, in turn, is influenced by the output of the SUT. Equally, the interrelationship between input and output can be seen from the opposite end: the output signals of the SUT are fed back to the inputs of the SUT — either directly or mediated by an environment model. In open-loop architecture, this kind of feedback control loop is interrupted deliberately. Closed-loop architectures also make extensive use of models. Environment models, especially, form a central part of closed-loop architecture. Thus, in closed-loop architectures, execution object and environment model exist together inside of and form one self-contained entity.

When it comes to testing, closed-loop architectures are more difficult to handle than open-loop architectures. Instead of simply defining a set of input data and then assessing the related output data, as is often done with open-loop architecture, tests in a closed-loop scenario have to be integrated with the environment model. Neither environment modelling, nor the integration with the test system, nor the individual tests themselves can be carried out in a generic and repeatable way. Thus, it proves difficult to properly define and describe test cases, to manage them or even to reuse them partially. The following kinds of feedback control loop testing architectures are used in industrial practice:

- *Model-in-the-Loop (MiL):* In model-based system development processes, an executable model of the ECU's software is tested in a simulation environment. The simulation environment ensures the execution of the model and its integration with any present environment models.

The tests assess the correct modelling of the functional requirements and, additionally, produce feedback about the suitability of the test algorithms.

- *Software-in-the-loop (SIL):* The code (either hand-coded or system-generated when model-based development techniques are applied) is tested in a software environment in the development machine. This kind of test aims to verify the correctness of the implementation of the functional requirements and the correctness of the code generated from the previously validated model. Compilation-specific issues, like the scaling of fix point arithmetic results, are considered.

- *Processor-in-the-loop (PiL):* The executable code of the application (or functionality) being tested is placed on an evaluation board, or an appropriate processor simulation. PiL tests aim to find target-specific sources of failure, such as target-specific compilation issues or specifics for a concrete processor architecture. In comparison with tests executed upon original hardware, PiL tests are executed in a controlled and instrumented environment. This allows for additional measurements and observations. The test results can be compared with the results of previously-executed MiL and SiL tests, in order to find any unexpected deviations in the system reaction.

- *Hardware-in-the-loop (HiL):* A HiL environment is the most accurate replica possible of either an ECU's original target environment or of a system composed from multiple ECUs. It contains real hardware and partly simulated sensors, actuators, and mechanical and electrical components. HiL environments enable the testing of electronic characteristics and can also simulate a complete network of interacting ECUs. OEMs generally use HiL Environments during the final stages of development to test and simulate the complete electronic infrastructure of a vehicle. The real-time HiL-computer that controls the test execution offers a realistic test of time-critical requirements and the interaction of sensors, actuators and ECUs over a bus system. It also simulates an environment that will show at least some of the electrical properties of the original system. HiL testing also tests ECUs in their composition and thus possible communication errors can be identified.

- *Road or vehicle tests* are performed with the vehicle-mounted control unit, by a driver, in a real environment (weather, road surface, etc). Vehicle or road tests assess applications and system functionality in the original operation environment, under realistic operation conditions.

Normally, different test and simulation frameworks are applied to carry out tests within different architectures. A test and simulation framework consists of hardware, instrumentation, simulators, environment models, software tools and other supporting artifacts that are needed for test execution. Almost every part of a test and simulation framework has individual requirements for testing methods, testing languages and testing concepts.

2.2.3 Testing languages and approaches in the automotive domain

The testing of softwares used in the automotive industry must be on par with the high-level technical and business requirements of the industry. To wit, the testing methodology must match the complexity of the industry. It must take into account, among other things: the existence of supply chains with a variety of different suppliers; the special role of the OEMs as specifiers and integrators; long development cycles with highly formalized processes; new technologies for distributed embedded systems (such as AUTOSAR); and, finally, the high level of responsibility and accountability inherent in manufacturing for such a safety-critical industry (i.e. the requirement of long-term documentation and archiving of test results and test documentation, as well as the application of formal test procedures and languages).

The established test approaches and tools from National Instruments [88], dSPACE [32], Etas [34], Vector [125], MBtech [84], for example, are highly specialized. They rely on proprietary languages and technologies and they are mostly closed approaches, with respect to their to portability, extension and integration. Efforts to address test exchange, especially within the automotive industry, already exist but these attempts have not yet solved the problem [50, 46, 101]. Attempts to standardize and harmonize the existing languages and test environments are underway; they are still, however, in their early stages.

Modelling and test specification in the automotive industry are often based on proprietary languages that have been developed by OEMs, their suppliers or by variants of finite state charts. Additionally, these models and specifications are often implemented using tools such as Matlab/Simulink, or using UML or SysML state machines. In some cases, other tools are used for specific parts of the test modelling process (e.g. the Classification Tree Editor [81] is used for modelling test data and the TPT tool [17] is used for systematically modelling test cases for continuous systems). In recent years, UML models have become more common in testing. The UML Testing Profile (UTP) [89] is a standardized extension of the UML to facilitate model-based test specification.In addition, AUDI has now developed a test development

environment and a testing methodology called EXAM [85]. EXAM facilitates the representation, implementation and evaluation of test cases via UML sequence diagrams, UML use cases and UML activity diagrams. Test sequences can be modelled graphically, and programming skills in sequence diagrams or activity diagrams are not required. The diagrams can be interpreted by machine and converted into executable test programs. This kind of test case specification is largely independent of the test system itself. EXAM is similar to other approaches, like MODENA [11], an elaborated adapter infrastructure which ensures that the abstract test suites can be applied to the SUT. Testers are freed up, therefore, to create test cases with high level objectives, rather than having to concentrate on the technical details of the test system itself.

In the automotive industry, fully-qualified MBT approaches are mostly still in the research phase. For several reasons, they are not yet widely applied in the industry. Firstly, there are only a small number of mature tools that support MBT. Secondly, change in the automotive industry occurs very slowly: new methods have to be accepted, adapted and integrated into strictly-defined processes. Last but not least, the task of creating and applying fully-qualified MBT approaches is very challenging. Automotive software systems are distributed, concurrent, real-time systems. Modelling such systems is a task in and of itself and, currently, there are only a handful of approaches that have the capacity to generate relevant tests for such models [93, 25, 21]. Nevertheless, MBT is steadily gaining popularity in the automotive industry, due to its functionality and robustness in testing [92]. MBT approaches aim to assess functional models, implementation models, software items (i.e components, modules and software systems) and, finally, integrated ECUs (i.e. the combination of software and hardware components).

Another prominent issue arising within the industry is the testing of variants, or Software Product Lines (SPL). Automotive product variations arise from differences in consumer details and divergent international legal regulations across markets. SPL allows for systematic reuse of tests across clusters of similar products. Testing each product of an SPL individually is generally not feasible. In [90] an approach for model-based SPL testing is introduced. The author suggests an approach that generates a representative set of products for a SPL; this set produces a comprehensive model with coverage of all features. The feature model can then be mapped against a reusable test model, and thereby automatically generate test cases for each product. The development of SPL testing approaches is a vital part of developing high quality products which have a short time to market. Test execution automation within the automotive industry can be considered as established; the use of

formal languages and the specification of tests on a higher level of abstraction is also, more or less, on its way. MBT is gaining more and more relevance in the automotive domain but, as of yet, there has been no breakthrough of MBT into industrial practice.

In general, a testing language should provide suitable abstractions that can define and assess analogue and sampled signals. This is necessary because a testing language must be able to simulate the SUT's physical environment and interact with dedicated environment models that show continuous input and output signals. Additionally, 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 via the exchange of complex messages. They use different communication paradigms, like asynchronous event-based communication or synchronous client-server communication[3]. Typically, communication behaviour is tested using testing languages that provide support for event- or message-based communication and provide a means to assess complex data structures. Because reusability is an important concern, the language should provide sufficient support for modularization and support for the specification of reusable entities, such as functions, operations and parameterization.

While most of the testing approaches, languages and tools that are currently in use are able to deal with discrete data, an approach that unifies the ideas from software development and telecommunication with concepts arising from systems engineering and control theory is missing. A particularly important, and still unavailable, approach would be one that could integrate concepts that can test the continuous aspects of an automotive electronic control unit or a network thereof.

2.3 Introduction to continuous and hybrid systems

On a theoretical level, a significantly large number of today's embedded systems can be characterized as *dynamic hybrid systems*. A dynamic hybrid system is a system that shows both continuous and discrete dynamic behaviour. The terms *discrete system*, *continuous system* and *hybrid system* originate in dynamical system theory [114, 130] and, as such, they describe mathematical models for systems with certain kind of properties.

- A *discrete system* is a dynamic system that has a finite number of states and is therefore fully computational. The state changes are normally

[3]Please refer to AUTOSAR [10]. This is a good example of an innovative industry-grade approach to designing complex control system architectures for distributed environments.

processed at fixed and predictable time steps. Discrete systems belong to the class of transition systems with finite states.

- In contrast, the term *continuous system* refers to a dynamic system that shows analogous input and output characteristics. The inputs and outputs of such a system *flow*. They are capable of changing at any instant, to any real numbered value, and the relation of the inputs and outputs can be modelled as partially or ordinary differential equations. In a continuous system, state changes occur at infinitesimally small time steps and the number of states are infinite.

- The term *hybrid system* characterizes a dynamic system that explicitly allows the existence of *discrete jumps* [54, 5]. This means that the inputs and outputs of a signal may evolve continuously, but, as well, may show discrete behaviour at certain moments in time. This concept is especially necessary when describing the behaviour of software-driven controllers. Software-driven controllers are often used to control continuous processes or quantities, like the cornering ability of a vehicle or the simple velocity of a car. Software-driven controllers, therefore, explicitly deal with continuous quantities. On the other hand, software itself and the ideals of its design envision discrete behaviour. Control software often defines certain phases (or states). These phases are, in fact, often dependent on continuous system inputs and are used to trigger certain (often continuous) behaviour variants. Thus, the detection and the thereby-triggered control behaviour itself are often both continuous; the switches between different phases are, however, discrete by nature. In addition, software-driven controllers are influenced by external discrete events (e.g. activation/deactivation, status messages from other controllers, etc) which have impact on their own behaviour.

In the following chapter, the terms *hybrid system* and *continuous system* will be used as classifications of control systems. As such, the phrase "testing hybrid control systems" will refer to attempts to test a control system that shows the input/output behaviour of a hybrid system. The inputs and outputs of such systems will be referred to as *signals*.

2.3.1 Continuous and discrete signals

A signal is a time-varying quantity that can be measured by a technical system. Such measurable quantities can be represented in different theoretical forms. Typically, quantities with discrete representations (i.e. those represented by an integer value) or with continuous representations (i.e. those

represented by real values) are distinguished from one another. Both discrete and continuous signals can be classified (according to their respective values and time domains) into the following four categories [22, 23, 30].

1. *Analogue signals* are continuous in the domains of both time and value. They are the most 'natural' signal category, because they are characterized by physical units (e.g. current, voltage, velocity) and measured with sensors. Some typical physical quantities used in the area of embedded system development are vehicle velocity and field intensity of a radio station. Analogue signals can be described as a piecewise function over time with $\sigma^a = f(t)$ and $t, \sigma^a \in \mathbb{R}$).

2. *Time-quantified signals* show discrete values in the time domain and continuous values in the value domain. The signal values are defined only at predetermined time points (so-called sampling points) with $\sigma^{tq} = f(t)$ and $t \in \mathbb{N}, \sigma^{tq} \in \mathbb{R}$). One typical example of time-quantified signals could be the time/value pairs of a recorded signal; and a typical representation of a time-quantified signal could be, for example, a series or an array of real numbers. Even if the original signal is a synthetic function, it can only be reconstructed from a time-quantified signal with considerable mathematical effort.

3. *Value-quantified signals* are time-continuous signals with discrete values. A typical example of a value-quantified signal would be data derived from analogue signals, which is dedicated to further processing (e.g. an A/D converted sensor signal that is provides to an electrical control unit).

4. *Digital signals* are discrete with respect to the time and value domain with $\sigma^d = f(t)$ and $t, \sigma^{tq} \in \mathbb{N}$). When the value domain has only two possible signal values (that is to say, when the value domain has exactly two elements), the signal can be characterized as a binary signal. Typical examples of binary signals include switching positions or flags. Digital signals, including the occurrence of events, can be modeled.

2.3.2 Systems of equations

In system theory, the behaviour of dynamic systems is normally described using *Systems of Equations*. These kinds of mathematical models are widely used in natural science to describe physical, real world processes such as the vibration of a pendulum, radioactive decay, the dynamics of fluids, and so

on. Systems of equations modelling is one of the basic techniques in mathematics, natural science and engineering science, and it provides a multitude of methods too numerous to be outlined here. In systems engineering, *algebraic equations*, *difference equations* or *differential equations* are used to describe the dynamics of technical systems, especially mechatronic systems (e.g. electric drives, hydraulic systems, etc). The behaviour of such systems is generally designated as an operator $\mathcal{T}[\cdot]$, which continuously acts on the inputs of the system [77]. The system allocates the outputs, according to a function, over time $\sigma_i = f_i(t)$. Systems of equations make it possible to represent the dynamics of such a complex system as a relation between time, system input and system state. In this thesis, simple classification will be used. While *differential equations* (and thus differential equation systems) relate the values of individual functions to themselves and their derivatives of various orders, a *difference equation* only refers to discrete values. The quantities used for input and output, therefore, are discrete signals or, mathematically speaking, series of values. Many methods of computing numerical solutions of differential equations or of studying the properties of differential equations involve approximating the solution of a differential equation via the solution of a corresponding difference equation. The term *algebraic equation* is used to describe the polynomial equation system of the form $f(x_1, x_2, x_3, ..., x_n) = \sum_{e_1,..,e_n} c_{e_1,e_2,..,e_n} x_1^{e_1} x_2^{e_2} ... x_n^{e_n}$, where the coefficients $c_{e_1,e_2,..,e_n}$ are integers [113].

2.3.3 Hybrid automata

Hybrid systems are often modelled as State-Transition-Networks (STNs) [5, 80, 8, 27]. Similarly to continuous systems, the behaviour for each state is defined via partially or ordinarily differential equations or algebraic equations. The transition between states (which, in fact, models the discrete character of the hybrid system) takes place when certain logical conditions become true.

Hybrid automata are the conceptual extension of *timed automata* [6]. They where first introduced by Alur et al. [5] in 1992, where they were used to analyze properties of hybrid systems. Hybrid Automata consist of State-Transition-Networks (e.g. Finite State Machines) that define so-called phases or modes. Each phase or mode shows different continuous behaviour. While the Alur-Henzinger automata [5] were designed with the intention of supporting the algorithmic analysis of hybrid systems model checking, Hybrid Input/Output Automata [80, 79] are more closely related to hybrid systems design and allow greater compositional modelling, abstrac-

tion and analysis of hybrid systems. Automata with discrete time [3, 55] are more computationally-oriented models. There are modelling environments like HyChart and Charon that are completely dedicated to the specification, analysis and validation of hybrid automata; industrial-grade modelling languages like Matlab/Simulink and SysML have integrated the concepts of hybrid automata into their modelling paradigms, so that they are now able to specify hybrid systems. Different approaches have been developed for the formalization of these hybrid systems [99, 80, 6].

2.3.4 Data streams and stream processing

Within computer science, the term *data stream* is used to describe the flow of (continuous or discrete) sequences of data between Stream Processing Systems (SPS). Streams are similar to pipes, which were introduced by Doug McIlroy for the Unix operating system. Data streams are continuously processed and are particularly suited to the representation of dynamically-evolving quantities over the course of time. The length of a stream cannot be established in advance. There are different kinds of streams with different properties (e.g. varying data rates) and formalisms; the characteristics of a stream depend on the domain of application.

Stream processing has been under research since the early 1960s. In 1974, Kahn introduced the so-called Kahn-Networks: a formalism for parallel computation on the basis of fixed point semantics [69]. In the 1980s and 1990s, stream-processing languages like Lucid [126] and LUSTRE [52] were introduced to model asynchronous and synchronous data flow. SIGNAL [40] and ESTEREL [111] were designed for the specification of signal-processing networks and reactive systems, and STREAM [26] is a language for verifying hardware. For detailed information on stream processing and stream-processing languages, please refer to [109]. In recent years, stream processing has become especially famous in the context of the distribution of multimedia on the Internet [119, 118].

The contribution of M. Broy [19] is of special interest to the specification of embedded systems. Broy has introduced a theory of streams for concurrent components. This theory identifies different kinds of streams (e.g. discrete streams with discrete time, discrete streams with continuous time, dense streams, etc), each of which rely on different kinds of timing models (discrete time, continuous time, dense time, etc). The relations between the different kinds of streams constitute different levels of abstraction with respect to the underlying model of timing. A refinement relation is introduced to map the respective timing models accordingly.

Data streams (or, for short, *streams*) provide an ideal representation of the different types of signals introduced in Section 2.3.1. They have been exhaustively examined in academia and are widely used to describe finite and infinite data flows. In contrast to scalar values, they provide the whole allocation history applied within one a channel and they provide a suitable implementation for the continuous evolution of quantities.

2.3.5 Specification languages for continuous and hybrid systems

In recent years, model-based specification techniques have become available for discrete as well as continuous and hybrid systems. For discrete systems, there already exists a large range of modelling approaches, including state-based approaches like Statechart [53], B [1], Z [105, 106] or UML State Machines. There are also interaction-oriented approaches like Message sequence Charts, UML Message Diagrams; process-oriented approaches like CCS [86], CCP [100], LOTOS [2]; and different variants of Petri Nets [97, 98]. For some of the modelling approaches, dedicated real-time extensions are available. For example, timed automata [6] are widely accepted as a formalism used to check the dynamics of real-time systems. Modelling techniques for hybrid systems include phase transition systems [82], hybrid automata [5] and hybrid I/O automata [80]. The techniques and formalisms have been analyzed and developed carefully in academia for many years. Academic tools like Shift [27], Ptolemy, [73], HyChart [49] and languages like Charon [7] have been available for a long time. There are also already a number of mature industrial-grade tools like Matlab Simulink/Stateflow [115], Esterell Scade [111],and IBM Rhapsody UML, all of which are in daily use and provide industrial-grade modelling, verification and code-generation environments for hybrid systems.

The tools Simulink/Stateflow [117, 115] feature a modelling and simulation framework based on the numerical computing environment MATLAB [116]. Simulink provides engineers with a signalflow-oriented graphical modelling language for embedded systems. Among other things, it supports the specification of systems based on differential equations. Stateflow is an extension for Simulink and supports the intuitive definition of state-based and event-driven systems by means of finite-state machines. Both Simulink and Stateflow can be used, in conjunction with the code-generation facility Target Link, to generate industrial grade C code.

Esterel Scade [111] provides data- and control-flow-oriented modelling approaches which are based on the synchronous and data-flow-oriented language Lustre [52], in combination with finite-state machines. Similarly to

Matlab Simulink/Stateflow, Esterel Scade has evolved towards an integrated modelling environment for embedded systems that, outside of modelling and simulation, also provides a number of mature tools for code generation and verification.

While Esterel Scade and Matlab Simulink/Stateflow provide their own languages, IBM Rhapsody is based on the standardized modelling languages UML2 and SysML.

2.4 Testing techniques and approaches for continuous and hybrid real-time systems

In recent years, some new approaches and methods have been developed to test hybrid systems. The preeminent approaches with an industrial background can be found in the automotive industry. In the academic community, new approaches to model-based conformity testing of hybrid systems have been of special interest. In the following section, the most relevant approaches are outlined.

2.4.1 Test modelling approaches for hybrid systems

The Classification Tree Method for Embedded Systems (CTM/ES) [22] applies the classification tree method to embedded systems and especially addresses the generation of test stimuli for continuous and timed behaviour. The Classification Tree Method [81] is a special approach to partition testing, applied to the input domain of a test object. The input domain is analyzed under different test-relevant aspects and, for each aspect, a number of disjointed classifications can be formed. The classifications are further refined by adding classes and class representatives. The step-by-step partition of the input domain via classifications is represented visually as a classification tree. On the basis of the classification tree, test cases are then derived to subsequently cover the different classes. In addition to the classical CTM, the CTM/ES provides a timeline that associates test steps with distinct time values and provides a number of signal primitives. This allows for continuous definition of transitions between class representatives. The CTM/ES is integrated in the tool MTest, which was originally developed at Daimler and has since been used for testing Simulink/Stateflow Models.

Timed Partition Testing (TPT) [74, 17] is a method for systematic blackbox testing of continuous behaviour. As a tool, TPT includes a modelling language that defines test scenarios on the basis of hybrid automata [5] and stream-processing functions [19]. A TPT model addresses the systematic

definition of test inputs and usually forms a superset of test cases, whereby each individual test case itself is a full-featured state machine. TPT concepts are dedicated to continuous and hybrid systems in the automotive industry, and they allow for the definition of reactive test cases. Based on its theoretical foundations, the TPT tool supports test definition via the TPT language, test execution within a TPT runtime environment and test evaluation via a separate, Python-based library. TPT is mainly used for component testing, integration testing and system testing of embedded systems in Model in the Loop (MiL), Software in the Loop (SiL) and Hardware in the Loop (HiL) scenarios.

On the basis of the TPT concepts, Perez et al. [95, 96] developed a test method for multilevel testing. The authors introduced a modularization approach for test cases, whereby modules are formed in order to separate the invariant and variant parts of a test specification. Invariant parts of a test case form the core, abstract test case (i.e. the part which can be reused); variant parts of a test case, on the other hand, are transferred to a multi-layered test adapter, which has to be redesigned for each testing layer. This so-called multilevel approach allows for the reuse of test cases between different test layers and includes both a test design strategy and sufficient tool support on the basis of the TPT tool.

2.4.2 Assessment of signals via signal properties

The specification of formal properties (in order to denote the requirements of a hybrid system) is a well-known technique from the theory of hybrid automata [5]. Given a set of formal system properties (denoted in a temporal logic calculus), the reachability of the properties can be automatically checked by an appropriate system model, designed specifically for this purpose [54, 67]. The Reactis tool environment [104] provides a similar approach to deriving test cases from models, an approach which can also be applied to the system under test (SUT).

Model-in-the-Loop for Embedded System Test (MiLEST) [131] is a method that addresses the black-box functional testing of hybrid and continuous systems. In [131], a systematic approach for the derivation of so-called validation functions from requirements is described. The approach introduces the notion of signal properties and their respective concatenations. A signal property is an abstract description of certain — sometime very complex — signal characteristics (e.g. value changes, increase or decrease of a signal, signal overshoots, etc). One or several signal properties can be used to closely characterize a signal shape, or an expected evolution of a signal, during a

Name	Characteristic	Description	Locality
Signal Value	value = *exp*	the signal value equals *exp*	local
	value in [*range exp*]	the signal value is in [*range exp*]	local
Value Change	no	a constant signal	frame-local
	increase	an increasing signal	frame-local
	decrease	an decreasing signal	frame-local
Extremal Value	minimum	the signal has a local minimum	frame-local
	maximum	the signal has a local maximum	frame-local
Signal Type	step-wise	a step function	global
	linear	a partially linear signal	global
	flat	a partially flat signal	global

Table 2.1: Signal Properties

black-box test run. The Milest box is based on Simulink/Stateflow and provides predefined test samples that help to form hierarchical test specifications based on signal properties.

In [20, 48, 131], such predicates are used as an explicit part of a test specification in order to facilitate easier assessment of a hybrid system's reaction. In [20], a graphical modelling tool is outlined. This tool is dedicated to facilitating easier specification of signal properties for the off-line evaluation of tests. Both approaches aim to systematically denote signal properties. Table 2.1 shows a selection of*basic properties* adopted from [20].

While the actual signal value is a property that is completely *local* (i.e. it is quantifiable without the history of the signal), the other properties only become allocatable when the predecessor values are known and considered. Properties are referred to as *frame-local* when the history can be limited to a certain frame and referred to as *global* when this is not the case. Local properties are adequate for on-line analysis in any situation, whereas frame-local properties are adequate only in instances of specific frame size. Large frames may constrain the real-time capabilities of the test environment. Global properties are normally not relevant or applicable for on-line analysis, because they depend on the complete signal. This thesis is confined to the discussion of local and frame-local properties.

To address frequencies, monotony and the exact amount of decrease or increase that a signal has, the notion of preprocessing functions are introduced. A preprocessing function obtains a signal as input and then provides the transformed signal in the form of output.

2.4.3 Model-based testing for real-time systems

Model-based testing of real-time systems is mostly based on timed automata. In the UPPAAL tool, it is possible to edit, simulate and check properties of UPPAAL-timed automata in a graphical environment. Hessel et al [56] and Stenh [108] use UPPAAL or UPPAAL model checking capabilities to generate test cases. The idea behind their approach is to formulate the test problem as a reachability problem in the model, and then solve it with existing model-checking solution. UPPAAL has been used to generate test cases for real-time systems.

UPPAAL-TRON [72, 25] is an extension to the UPPAAL verification and simulation environment tool. It is a tool for model-based, black-box testing of real-time systems. In UPPALL-TRON, tests are generated via analysis of timed automata, which describe the system and knowledge of the intended environment. The tests aim to state conformance between a specification of S and an implementation of I. To assess this, the authors define a conformance relation. The conformance relation *rioco* adapts the rioco-theory of Tretmans et al. [120] and extends the theory to comply with timed models. The *rioco* relation guarantees functional and time-intelligent corrections in the application of timed traces.

Peleska et al. [94] describe an MBT approach for test case and data generation which is dedicated to concurrent, real-time systems. The approach derives so-called symbolic test cases from within the abstract syntax of an extended finite state machine. In the model, the test cases are then defined as reachability goals. Concrete test data are calculated by an SMT solver that takes guard conditions as variables and takes the transition relation as input. The SMT solver then calculates possible valuation sequences that, when applied to the system, will lead to situations where the reachability goals are fulfilled within a finite number of transitions. The approach is defined via timed automata with dense time. It has been used in models from the automotive industry [93, 92, 94] as well as in the aviation industry [33].

2.4.4 Model-based testing approaches for hybrid systems

Model-based testing of hybrid systems is usually accomplished on a basis of hybrid automata. The challenge in testing such systems stems from the lack of adequate coverage criteria that can guide the test generation. Structural model coverage criteria used for discrete systems in connection with (E)FSM models cannot readily be transferred, because the discrete state space of a hybrid automaton often provides insufficient explanatory power. Hybrid states, in contrast, are infinite and are not represented in the model structure.

The approaches presented below have two principle objectives. First, the conformity of a hybrid specification and a hybrid realization is formally defined. Second, metrics are presented to measure coverage in the hybrid-state space and thus provide means for a meaningful test selection. The approaches are mainly based on the *ioco*-theory of Jan Tretmans [120] and provide different extensions to cover the properties of hybrid systems.

Conformance testing approaches for hybrid systems

Van Osch [123] has defined a conformance testing approach for hybrid systems. The approach is based on the well-known ioco-theory [120] and extends this theory to the notion of hybrid transition systems (HTS) and a corresponding conformance relation, the so-called hioco-relation. An HTS is based on the ideas of hybrid automata (see Section 2.3.3) and is defined as a tuple with a possibly infinite set of states: an initial state, a set of discrete transitions and a set of hybrid transitions. While discrete transitions are included in the original ioco-theory, hybrid transitions are new. They describe the continuous evolution of values across a set of variables. Hybrid transitions are characterized by so-called trajectories, i.e. partial functions that describe the values of a set of variables over time. The execution of an HTS is seen as a sequence of observable actions and trajectories. A sequence with the initial state as the starting state is known as a trace.

The hioco-relation describes a relation between the observable inputs and outputs of an implementation and a specification. It assumes that both implementation and specification are specified by means of an HTS. Implementation I is considered hybrid input-output conform to specification S (i.e. $IhiocoS$) when, for both states of implementation, the observable input-output actions and input-output trajectories are a subset of the actions and trajectories allowed for the specification. All input is considered to be filtered by the input allowed by the specification.

A conformance test is considered to be an HTS with the additional states of $pass$ and $fail$. A testing HTS is deterministic with respect to actions and trajectories and forms a tree-like structure, with the additional states of $pass$ and $fail$ as final states. The author has defined an inductive algorithm for conformance test generation on the basis of the *hioco*-relation. Theoretically, the algorithm produces tests that are sound and exhaustive; however, the algorithm is not realizable because it produces an infinite number of hybrid states and trajectories and is based on the notion of dense time. For practical execution, a meaningful discretization of time and value would be necessary, and this is currently missing.

Coverage measures and test seclection for hybrid systems

Dang [24] has developed a method and a framework for model-based conformance testing. The approach encompasses the following: a formal framework for conformance testing of hybrid systems; a coverage measure that determines test coverage for hybrid automata; and a coverage-guided test generation framework. The formal conformance testing framework has many similarities to the work of van Osch [123]. It is based on hybrid automata and defines a conformance relation that allows reasoning of the input-output conformance of a specification (i.e. a hybrid automaton) with a SUT.

The coverage measure for hybrid automata is based on the degree to which set of hybrid states are distributed over the hybrid-state space. The author has chosen a geometric approach and uses the star discrepancy, an approach to the reasoning of equal distribution that is well-known in statistics. The challenging problem is to define a suitable distance measure for a hybrid-state space. Dang proposes the use of euclidean distance when the discrete portions of two hybrid states $s = (q, x), s' = (q', x')$ are equal (i.e. $q = q'$). The distance between two hybrid states with different discrete portions (i.e. $q \neq q'$) is calculated using the average length of the trajectories between the two discrete states.

The test generation approach is based on the coverage measure described above and an adapted version of the Rapidly-exploring Random Tree (RRT) algorithm called hRRT. The hRRT algorithm allows a randomized exploration of the reachable hybrid state space of a hybrid automaton. The complete approach has been tested in various case studies.

In [16, 4], a technique called Qualitative Reasoning (QR) is used to model the interaction between an embedded system and its environment on an abstract, formal level. QR is based on differential equations and has been developed as an AI technique. It has been mainly used to model physical systems and their properties. In QR, numerical values of system properties are mapped to so-called intervals and landmarks. An interval represents a series of values and a landmark specifies the transition between two intervals. Time progress is modelled by a sequence of temporal, ordered states. The authors use a QR tool called Garp3 [121] to create QR models. Garp3 allows the intuitive definition of QR models by describing causal dependencies of the system properties and parts, without the need to explicitly define differential equations. QR models generally allow users to reason about systems, even with incomplete knowledge; thus, they allow for user-defined qualitative abstraction. Brand and Auchernig emphasize the mapping between QR models and Labeled Transition Systems (LTS), such that that a conformance testing approach similar to that of Tretmans and van Osch [120, 123]

is also applicable. The conformance testing approach that they introduce is called a Qualitative Reasoning Transition System (QRLTS), and they also introduce a conformance relation that defines the conformance between two QRLTS'. A QRLTS consists of a set of states, an initial state and a transition relation between states. Each state associates all modelled quantities of a QR model with a value and a delta. The delta is formally denoted by $\delta \ : S \ Q \ \{mon, zero, plus\}$ and describes the direction of value change. Furthermore, the transitions are annotated with labels that correspond to constraints upon the system's quantities. The conformance relation is defined as a modified $ioconf$ relation. This modified $ioconf$ relation relates the output quantities of an implementation to a specification, after applying a certain input vector. The input vectors are filtered by the inputs allowed for the specification. The modified conformance relation is called $qrioconf$.

Test generation is defined via formal test purposes that are themselves defined as LTS specifications. The test cases are generated by the synchronous product of the system specification, and the test purpose as it is introduced by [16] and realized by the TGV tool [66]. In addition to normal LTS testing, not only the states themselves but also the change of quantities inside a state are relevant for testing. The authors thus propose modelling the constraints of value quantities by means of regular expressions. These expressions are transformed in the equivalent deterministic automaton, which represents an LTS with labels that correspond to the modelled constraints.

The generated test cases are represented by a complete test graph (CTG). Due to the inherent approximations and abstractions in the QR model, the test cases are abstract with in both the time and value domain. They have, therefore, to be refined by means of an abstraction/refinement relation before they can be applied to a real system. The authors introduce an approach to the refinement of abstraction/refinement relations and deciding whether a concrete implementation is $qrioconf$ to a given QR specification. Since the QR modelling reflects the entire hybrid system in its physical presence, the approach supports conformance testing of the entire system, rather than only of the software. This includes conformance testing of embedded actuators and sensors, and consciousness of the physical environment. The main advantage of the approach is the idea of abstraction, in combination with an existing experimental tool chain. The approach is currently feasible, however, only in small-scale cases; the state space of the QRLTS is already quite big in small examples, such as the Garp3 tool, and quickly reaches its computational limits.

Finally, Julius et al. [68] describe an approach for automatic generation of robust tests on the basis of hybrid automata. The term "robust" is used in this context to identify test parameters that describe qualitative properties with

respect to the system behaviour (e.g. safety or correctness). The parameter space is partitioned accordingly and used to carry out a systematic selection of test cases with significantly different test performances. The authors show that their approach works for small examples and they provide a toolchain based on Matlab/Simulink.

2.4.5 Testing languages for continuous and hybrid systems

Testing languages are a special sort of programming language. Their main focus is the definition of distinct application scenarios, which are then used to test a system. Typically, testing languages provide a set of expressions and statements that support the definition of messages or signals applied to a system. Additionally, they provide a set of evaluation statements to assess the system's reaction. The levels of formalism used to define the expressions and statements of a certain test language are different [22] and are strongly dependent on the domain of application. Executable test languages in the field of real-time testing are specifically defined using formalisms with precise mathematical backgrounds.

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*) [36, 65] is well-established and widely used. *TTCN-3* is a complete redefinition of the Tree and Tabular Combination Notation (TTCN-2) [65]. Both notations have been standardized by the European Telecommunications Standards Institute (ETSI) and the International Telecommunication Union (ITU). Other testing or simulation languages, especially those dedicated to continuous systems, have been developed for the field of hardware testing or control system testing. The Very High Speed Integrated Circuit Hardware Description Language (VHDL) [59] and VHDL for analog and mixed-signal applications (VHDL-AMS) [62] can both be used to simulate discrete and analogue hardware systems. Neither language, however, was directly designed as a testing language. The Boundary-Scan Description Language (BSDL) [91] and The Analog Boundary-Scan Description Language (ABSDL) [110] are testing languages and, as such, they directly support the testing of chips using the boundary scan architecture [63] defined by IEEE. The Classification Tree Method for Embedded Systems (CTM/ES) is defined by the use of set theory and the notion of mathematical functions (cf. [22]). The Time Partition Testing Method (TPT) [17] and the Test Markup Language (TestML) [46] are approaches developed recently within the automotive industry, but they have not yet become standardized. In Time Partition Testing and TestML, mathematical functions are used to de-

fine signals. The notions of streams [19] and hybrid automata [5] are used to describe the overall control flow and the concatenation of simple signals to more complex ones.

The Abbreviated Test Language for All Systems (ATLAS) [60] Language and its supplement, the Signal and Method Modeling Language (SMK) [61], define a language set that was mainly used to test control systems for military purposes. ATLAS was replaced in 2010 by IEEE 1641 [58]. The IEEE 1641 provides a multilayered language framework that consists of several successive specification layers and their respective language definitions. The base layer is formed by a Signal Modelling Language (SML) that provides the mathematical foundations for signal definition and the Basic Signal Components (BSC) layer is used to define formerly-defined and reusable signal-building blocks. The Test Signal Frameworks (TSF) layer describes how BSCs are combined to form more complex signals. TSFs are usually assembled into libraries of related signals. The highest level is the test requirement itself, which can be expressed in several different forms. Like ATLAS, IEEE 1641 also includes its own Test Procedure Language (TPL), which can be used to describe the application and removal of signals.

Furthermore, the IEEE actually finalizes the standardization of a XML-based test exchange format: namely, Automatic Test Mark-up Language (ATML) [101]. ATML is dedicated to exchanging information about test environments, test setups and test results in an accepted and accessible way. Last but not least, there are also a huge number of privately-owned and -developed test control languages that have been designed and made available by commercial test system manufacturers.

Most of the languages mentioned above are not able to deal with complex discrete data (which are extensively used in network interaction), nor are they able to deal with distributed systems. *TTCN-3* does not support discretized or analogue signals to stimulate (or assess) sensors and actuators. ATML, which potentially supports both, is still quite new. ATML is also, so far, only an exchange format and still lacks a user-friendly representation format. ATML can, however, be used to describe test requirements in the IEEE 1641 framework.

2.5 Summary and motivation for this thesis

Current automobiles have dozens of control units, thousands of functional features and software with tens of thousands of line of code. Future generations of vehicles will be integrated into a comprehensive communications infrastructure, which will enable the exchange of data between individual vehicles,

bulks of vehicles and traffic control centres. Such communication-enabled vehicles will provide information about their speed, position, general status and routing information. In return, intelligent transport software applications, located either in the vehicles themselves or in traffic control centres, will be able to compute and evaluate this data and provide drivers with relevant information about dangerous traffic situations (obstacles on the road, dangerous weather events, traffic jams), traffic flow and the local environment (e.g. free parking spaces, local traffic routing, etc). Ensuring systematic and reliable quality in such systems introduce a new set of challenges, which, in their combination, are unique. These challenges present special requirements for test systems and the specification of test cases. In order to adequately respond to these challenges, a testing infrastructure needs

- adequateness of concepts, i.e support for specifying tests for different technologies and testing different kinds of systems (e.g. both communication systems and control systems),
- support for test automation and repeatability of test cases,
- adaptability to different test environments and simulation environments,
- integration with already-existing frameworks for model-based testing,
- long-term availability and understandability (e.g. an adequate level of abstraction and specification languages with self-documenting capabilities), and
- programmes for the qualification of test engineers.

Impending standards and legal norms such as ISO 26262 will demand the long-term documentation and availability of test specification. This will be best covered by standardized approaches and technologies. Standardized approaches and technologies usually evolve in a controlled and slow manner, and often have a longer lifetime than proprietary approaches.

The Testing and Test Control Notation, version 3 (*TTCN-3*) [65, 37, 36] is a test specification and implementation language used to define test procedures for black-box testing. *TTCN-3* was first developed in 1998 by a European Telecommunications Standards Institute (ETSI) team of experts, and is continually being developed. *TTCN-3* is the successor language for TTCN. The development of this language was driven by the industry and its need of a single test notation for the demands of black-box testing. *TTCN-3* expands on TTCN with several additional concepts, including dynamic test configurations, procedure-based communication and module control parts.

It provides concepts for both local and distributed, and for both platform- and technology-independent, testing. A *TTCN-3*-based test solution can be adapted to concrete testing environments and to concrete systems under test via its open test execution environment with well-defined interfaces for adaptation. Since *TTCN-3* is an established technology within the industry, several best practices that support the efficient use of *TTCN-3* exist. Furthermore, there are certified training courses that enable their participants to acquire *TTCN-3* certificates. Although embedded systems (e.g. in mobile phones) are already covered by *TTCN-3*, they are not its main target; hence, dedicated support for real-time, continuous and hybrid behaviour in *TTCN-3* is lacking. In order for *TTCN-3* to reach its full potential as a language for testing hybrid and continuous systems, it needs to be extended.

Control systems in the automotive domain can be characterized as hybrid systems that encompass both discrete and continuous behaviour (in time). Discrete signals are used for communication and coordination between system components. Continuous signals are used for the monitoring and controlling of components, and scanning the system's environment via sensors and actuators. An adequate test technology for control systems needs to be able to control, observe and analyze the timed, functional behaviour of these systems. This behaviour is characterized by a set of discrete and continuous input and output signals and their relationships to each other. While the testing of discrete controls is well-understood and available in *TTCN-3*, concepts for specification-based testing of continuous controls and for the relationships between discrete and continuous system parts are lacking. *TTCN-3* especially lacks concepts for time-controlled processes, sampling, and the definition, generation and assessment of signals.

To overcome its limitations and to empower *TTCN-3* to test embedded systems (particularly continuous and hybrid real-time systems), the standard has to be extended. The first approaches were published in 2005 [102] and 2008 [103]. A revised proposal for such an extension, namely *TTCN-3 embedded*, is the central theme of this thesis. The basis for the extension was developed as part of the TEMEA [112] research project and the extension has now been fully completed in this thesis. *TTCN-3* has been chosen for several compelling reasons. First of all, the *TTCN-3* standard is a formal testing language that has the power and expressiveness of a normal programming language, as well as a formal semantic and a user-friendly textual representation. Furthermore, the *TTCN-3* standard provides strong concepts to stimulate, control and assess message-based and procedure-based communication in distributed environments. It can be anticipated that, in the near future, these kinds of communication will become much more important for distributed control systems. Additionally, some of *TTCN-3*'s communica-

tions concepts can be reused to define signal generators and assessors for continuous systems; thus, they provide a solid basis for the definition of continuous and hybrid test behaviour.

Chapter 3

Selected concepts for testing continuous and hybrid systems

Testing continuous and hybrid systems differ from testing ordinary computer systems in many ways. On a practical level, embedded systems do not have an ordinary user interface. Thus, tests have to be defined for the technical interfaces like the network interface or the sensor and actuator interface. Furthermore, most of the embedded systems are not testable without their environment. Hence, a test system set up has to provide such an environment either in form of a real physical environment during field tests or in form of simulations for laboratory tests.

On a conceptual level continuous and hybrid systems show a completely different input-output behaviour than ordinary computer systems. They have a direct relationship to time, and, as the name suggests, the values at the input-output interfaces are able to change continuously.

A test environment, which is dedicated to systematically test continuous and hybrid systems has necessarily provide means to generate, measure and evaluate continuous and discretized (sampled) signals. Furthermore, a large number of embedded systems are real-time systems. Testing a real-time system requires a test system, which is capable of exactly measuring the timing of signals. A specification environment, which aims to support the test engineer during the process of test specification, should ideally provide the test engineer with specification concepts that ease the specification of time triggered signal generation, time measurement and signal assessment in an intuitive way.

In this chapter a number of formal requirements are introduced. These requirements yield as a foundation of the testing approach, which is developed in this thesis. The requirements relate to formal concepts and well-known formalisms like clocks, sampling, and data streams. Throughout the following chapters, the concepts are used to define the *TTCN-3* extensions for continuous and hybrid systems.

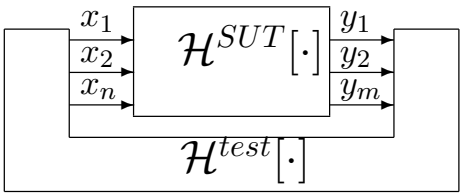

Figure 3.1: A typical black-box test set-up

3.1 TIME

The notion of time progress is essential for the stimulation and assessment of hybrid systems. It is usually measured by so-called clocks or clock variables. A clock $clck \in CLCK$ is a continuous quantity with the property $\dot{c} = 1$. The function

$$t : CLCK \rightarrow \mathbb{R}^+$$

returns the actual clock value. Each test system is considered to have at least a globally accessible clock $clck_0$. At the beginning of each test execution, the actual clock value is stored in t_{start} so that $t(clck_0) - t_{start}$ returns the time that has elapsed since the start of a test case. In a distributed test set-up, multiple clocks may be used that need to be synchronized. These clocks need to be synchronized to form a logical global clock. In the following the short form

$$t \equiv t(clck_0) - t_{start}$$

is used to denote the test case time.

3.2 PORT ALLOCATION AND TEST BEHAVIOUR

In a classical black box testing approach, the System Under Test (SUT) is represented in terms of its interface — the so-called test interface. A test interface is defined by a set of symbols usually referred to as channels or ports. Formally speaking, a test interface is defined by an n-tuple of input ports $\boldsymbol{x} = (x_1, x_2, .., x_n)$ and an m-tuple of output ports $\boldsymbol{y} = (y_1, y_2, .., y_m)$ with $m, n \in \mathbb{N}$ (see Figure 3.1). Each port is further characterized by a data type, a communication characteristic, and a communication direction.

That is, for each port $x_n, y_m \in Port$ there exists a set $X_n, Y_m \subset \mathbb{T}$ defining the domain of the port, a label $dir : Port \rightarrow \{in, out\}$ that denotes the direction of communication and a label $com : Port \rightarrow \{message, procedure, flow\}$ that denotes the communication characteristics. Message-based and

procedure-based ports follow event-driven semantics, i.e. state changes are propagated discretely and instantaneously. While standard *TTCN-3* provides an FIFO queuing mechanism to ease the assessment of message- and procedure-based ports, it currently provides no support for flow ports. The allocation of flow ports are defined by partial functions over time $\sigma_{x_i} : \mathbb{R}^+ \to X_i$ for input channels and $\sigma_{y_i} : \mathbb{R}^+ \to Y_i$ for output channels, respectively. The system behaviour is denoted as an operator $\mathcal{H}^{SUT}[\cdot]$ that continuously operates on the inputs and the internal state of a system [77].

A black box test system, however, is a system that is dedicated to test systems. For test systems, simply stimulation behaviour and evaluation behaviour, which both constitute the test behaviour $\mathcal{H}^{test}[\cdot]$, is distinguished. Test behaviour is considered to be formally defined by a hybrid automaton. Hybrid automata are State-Transition-Networks (STNs) [5, 80, 8] that define behaviour for each discrete state in terms of flows (i.e. by partial or ordinary differential equations or algebraic equations). The transition between states, which in fact models the discrete character of the hybrid system, takes place when a certain logical condition becomes true. The following definition of a hybrid automaton is used as a basis for defining control structures in the next chapters. A hybrid automaton consists of:

- A finite control flow graph (V, E) with vertices in $v \in V$ called modes and edges in $e \in E$ called control switches.
- A set of input and output channels x_n, y_m and a set of internal variables z_k with types $X_n, Y_m, Z_k \subset \mathbb{T}$.
- An edge labeling function *until* that assigns to each control switch $e \in E$ a predicate whose free variables are in (X_n, Y_n, Z_n)
- A finite set of actions A, and an edge labelling function $action : E \to A^*$ that assigns a set of actions to each control switch. The actions are triggered when e has the actual activated mode as source, a is assigned to e, and the respective until predicate $until(e)$ evaluates *true*.
- The vertex labelling functions $flow$, *inv*, *onentry* and *onexit* that assign to each control mode a continuous behaviour operator $flow(v) = \mathcal{T}_v^{test}[\cdot]$, an invariant predicate $inv(v)$, an initial set of actions $onentry(v)$, and a final set of actions $onexit(v)$.

3.3 Sampling and streams

Sampling becomes necessary to obtain an adequate and computable digital representation for continuous signals at flow ports. It is defined as the trans-

formation of a continuous signal $\sigma \in \Sigma$ into a discrete structure. The notion of streams [109, 19] is used to formally operate on such a discrete structure.

3.3.1 Streams

In telecommunications and computing, a data stream is a sequence of messages in the process of being transmitted. Given a set M of messages, the term stream is used to denote the data structure $s \in M^*$, which represents a finite or infinite sequence of messages

$$s \mapsto \langle m_k \rangle \text{ with } m \in M, k \in \mathbb{N}$$

and M^* representing the super set of all messages. Each stream is associated with a data type $T \in \mathbb{T}$ that specifies the domain of the messages m_k. Thus, streams are considered as typed $s^T \in (M^T)^*$. They contain messages $m_k^T \in M^T \subset M$ of the same type. In the following only finite streams are considered, because testing is a process that is naturally finite in time and sampling reduces the amount of signal values to a finite number.

Empty streams $\langle\rangle$ and the empty message $\epsilon \in M^T$ are introduced to complete the definition from above. Both, empty streams and the empty message are compatible with every type, thus $\epsilon \in M^T$ and $\langle\rangle \in (M^T)^*$. The number of messages in a stream is denoted by $|| : (M^T)^* \to \mathbb{N}$. The number of messages in an empty stream is zero by definition, thus $|\langle\rangle| \mapsto 0$. Furthermore, the index operation

$$[] : (M^T)^* \times \mathbb{N} \to M^T$$

yields the i^{th} message of a stream s. The application of the index operation on an empty stream yields the empty message by definition.

$$(\forall\, i \in \mathbb{N}) \qquad \langle\rangle[i] \mapsto \epsilon$$

To operate on streams and their content, simple access and modification operations are necessary. The $\triangleright$ operation adds a message to a stream.

$$\triangleright : M^T \times (M^T)^* \to (M^T)^*, \text{ with } m \triangleright s \mapsto s'$$

$$\text{and } s'[i] \mapsto \begin{cases} m & \text{when } i = |s| + 1 \\ s[i] & \text{otherwise} \end{cases}$$

The concatenation of two streams is defined by the $\bullet$ operation.

$$\begin{aligned}&\bullet : (M^T)^* \times (M^T)^* \to (M^T)^*, \text{ with}\\ &\quad \langle\rangle \bullet s \mapsto s \text{ and } (m \triangleright s) \bullet s' \mapsto m \triangleright (s \bullet s')\end{aligned}$$

To obtain the top element of a stream, the head operation is introduced.

$$\begin{aligned}&hd : (M^T)^* \to M^T, \text{ with}\\ &\quad hd.\langle\rangle \mapsto \epsilon \text{ and } hd.(m \triangleright s) \mapsto m\end{aligned}$$

The timing information relies on a time-stamp function ts that relates a continuous time values to each message. Continuous time values allow for exact timing information without additional transformations.

$$ts : (M^T)^* \to \mathbb{R}^+$$

For the concatenation of timed streams, the following meaning of the time-stamp function is required. Let $s, s' \in (M^T)^*$ be two timed streams of arbitrary length. The time function for the concatenation $(s \bullet s')$ is defined as follows.

$$(\forall i \in [1 : |s| + |s'|])$$
$$ts((s \bullet s')[i]) \mapsto \begin{cases} ts(s[i]) & \text{when } i \leq |s| \\ ts(s[|s|]) + ts(s'[i - |s|]) & \text{otherwise} \end{cases}$$

Finally interpolation operation that returns a message for each point in time $t \in \mathbb{R}^+$ is required.

$$\begin{aligned}&@ : \mathbb{R}^+ \to M^T \text{ with}\\ &\quad s@t \mapsto s[i] \text{ , when } t \in [ts(s[i]), ts(s[i+1])[\end{aligned}$$

3.3.2 Sampling

For sampling, the simple time model $t = k * \Delta$ with a fixed step size Δ and $t,\ \Delta \in \mathbb{R}^+$, $k \in \mathbb{N}$ is used.

Let $sample_\Delta$ be an arbitrary sampling function and $\sigma \in \Sigma$ an arbitrary signal, which is defined for $t \in [0 : k * \Delta[$, then

$$sample_\Delta(\sigma) : \Sigma \rightarrow (M^T)^* \text{ , with } s \ \in (M^T)^*$$
$$\sigma((i-1) * \Delta) \mapsto s[i], \text{ and } i \in [1 : |\frac{t}{\Delta}|]$$

Considering an arbitrarily small step size, any other sampling model, even dynamic sampling, can be derived by the concatenation of discrete streams and down sampling.

3.3.3 Template streams

To be able to characterize and assess incoming streams with a common formal structure, the concept of a template stream is introduced. While a stream is an ordered set of messages a template stream is an ordered set of predicates. Let P be a set of predicates with relational operators in $\{=, <, >, \geq, \leq\}$ a set of bounded variables in M and a free variable in $(M^T)^*$, the structure $b \in (P)^*$, which represents a finite or infinite sequence of predicates $p \mapsto \langle p_k \rangle$ with $p \in P, k \in \mathbb{N}$, is called template stream.

Template streams show similar access and modification methods than ordinary stream, thus the size operation $|| : P^* \rightarrow \mathbb{N}$, the value operation $[] : P \rightarrow M^T$, the add or construction operation $\triangleright : P \times P^* \rightarrow P^*$, and the head operation $hd : P^* \rightarrow P$. Moreover they provide timed related access with $ts : P \rightarrow \mathbb{R}^+$ and $@ : \mathbb{R}^+ \rightarrow P$.

Template streams as well as streams are produced and applied by hybrid automatas similar to the ones that have been defined in Section 3.2. To describe the meaning of applying a template stream to a (value) stream the notion of a generic evaluation function is introduced. The evaluation function $\chi(p, s)$ is defined by

$$\chi : P^* \times (M^T)^* \rightarrow \{\top, \bot\}$$

and evaluates as follows.

$$(\forall p \in P^*, s \in (M^T)^*)$$
$$\chi(p,s) \mapsto \begin{cases} \top & \text{when } \forall (t \in [ts(p(1)) : ts(hd(p))]), (p@t)(s@t) = \top \\ \bot & \text{when not} \end{cases}$$

3.4 Discussion

This chapter provides an overview on base concepts that are considered to be necessary building blocks for the specification of tests for continuous and hybrid systems. None of these concepts are currently addressed by *TTCN-3*. *TTCN-3* is a procedural testing language, thus test behavior is defined by algorithms that typically assign (send) messages to ports and evaluate (receive) messages from ports. For the evaluation of different alternatives of expected (sets of) messages, or timeouts, the port queues and the timeout queues are frozen when the evaluation starts. This so called 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 of *TTCN-3* for the requirements of continuous and hybrid behavior, the following notions are introduced.

- the notions of time and sampling,
- the notions of streams, stream ports, stream variables and template stream, and
- the definition of an automaton-like control flow structure that enables the specification of hybrid behavior.

The following chapters describe the syntactical and semantical integration of the concepts from above with the *TTCN-3* core language. If possible, the names of the constructs in *TTCN-3 embedded* are kept in line with terms and concept names from this chapter, so that the relationship between the *TTCN-3 embedded* constructs with their underlying base concepts remain evident.

Chapter 4

TTCN-3 for hybrid systems

The following sections describe the concepts defined for *TTCN-3 embedded*; these concepts are intended to ease the specification of tests for continuous and hybrid systems. In this section, the semantics of the concepts are described informally. The syntactic structure is defined by BNF snippets that explain the setup of the new constructs. The BNF snippets are based on the already-existing *TTCN-3* grammar in [35]. To highlight the differences between *TTCN-3 embedded* and standard *TTCN-3*, the newly-introduced rules and terminals are marked. The complete grammar for *TTCN-3 embedded*can be found in the appendix.

4.1 Time and sampling

TTCN-3 embedded adopts the concept of a global clock and enhances this concept with notions of sampling and sampled time. As in *TTCN-3*, all time values are denoted as float values, and time is represented in seconds. For sampling, simple equidistant sampling models and dynamic sampling models are used. While equidistant sampling models show the same step size for a longer period, dynamic sampling uses flexible step sizes, e.g. higher sampling rates, to achieve a higher accuracy for certain signal forms and situations.

At a technical level, an equidistant sampling model of the form $t = k * \Delta$ (where t describes the time progress, k specifies the number of executed sampling steps and Δ yields the minimal achievable step size for a given test system) is used as an overall base for modelling equidistant sampling with larger step sizes, or dynamic sampling.

The basic sampling in *TTCN-3 embedded* has a minimal step size of Δ. It is a property of a concrete test system and it is not intended to be specified in the test case specification. Because of this underlying model, however, a test system is only able to execute user-defined samplings if, and only if, all specified sampling rates at test-specification level provide step sizes that are multiples of Δ.

In *TTCN-3 embedded*, each reference to time, whether it is used for definition or evaluation of signals, as well as those gathered by means of

ordinary *TTCN-3* timers, is considered to be completely synchronized with the global clock and the base sampling.

4.1.1 Time

In order to specify time-dependent signal sequences, it must be possible to track the passage of time. Time is accessed via a globally-available clock whose value at any given time can be assessed using the now operation. For the purpose of each test case, time progress starts at the beginning the execution; time values are thus related to the start of the test case. The now operation can be used in any expression that is referred to from within a test case definition or a function definition. Its use is explicitly not allowed for the *TTCN-3* control part.

Intuitive semantics: The now operation yields the time as an amount of seconds that have passed since the start of the test case. In *TTCN-3*, the beginning of the test case is marked by the invocation of the TRI function *triExecuteTestCase* [37]. The time value is returned as a float value. The precision of the time value is, theoretically, unlimited, but, in fact, is bound by the choice of sampling and limited by the capabilities of the executing test framework.

Listing 4.1: The now operation

```
// Use of now to retrieve the lapsed time since the test case
// started                                                    2
var float myTimeValue := now;
```

Syntactical structure (concrete syntax) 1.

OpCall	::=	*ConfigurationOps* | *VerdictOps* | *TimerOps* |
		| *TestcaseInstance* | *FunctionInstance* | *TemplateOps*
		| *ActivateOp* | *NowOperation*
NowOperation	::=	*NowKeyWord*
NowKeyWord	::=	"`now`"

4.1.2 Define the step size for sampling

A *TTCN-3 embedded* test system provides a base sampling rate that is available system-wide. This sampling rate is given by the test system's capabilities and provides the basis for the sampling mechanism. The execution of each computational task is aligned to this base sampling rate.

In addition, separate sampling rates can be set for each stream port. Essentially, the sampling rates at ports multiply the base sampling. The

sampling of ports can be modified as part of the test specification by means of the step size annotation.

Intuitive semantics: The step size annotation defines the step size of ports. It does this by means of charstring values that represent time values in seconds. It is applicable to modules, test cases, groups, component types and stream port types. The step size annotation affects the step size of the port definitions contained in one of these entities, or, in case of stream port types, the annotation affects the respective instances of a stream port.

Listing 4.2: Setting the stepsize for modules

```
module myModule{                                                    1
...
}                                                                   3
with {stepsize "0.0001"};
```

Listing 4.3: Setting the stepsize for testcases

```
// sets the stepsize for a testcase
testcase myTestcase() runs on myComponent{                          2
...
}                                                                   4
with {stepsize "0.0001"};
```

Listing 4.4: Setting the stepsize for ports

```
type port StreamOut stream { out float}                             1
with {stepsize "0.0001"};
```

Syntactical structure (concrete syntax) 2.

WithStatement	::=	*WithKeyword WithAttribList*
WithKeyword	::=	"`with`"
WithAttribList	::=	{ *SingleWithAttrib* [*SemiColon*] }
SingleWithAttrib	::=	*AttribKeyWord* [*OverrideKeyword*]
		[*AttribQualifier*] *AttribSpec*
AttibKeyword	::=	*EncodeKeyword* | *VariantKeyword* | *DisplayKeyword*
		| *StepsizeKeyword* | *HistoryKeyword* | *ExtensionKeyword*
StepsizeKeyword	::=	"`stepsize`"

4.1.3 The wait statement

The wait statement delays the execution of a component until a given time. The given time is specified as a float value and relates to the internal clock of the test system.

Intuitive semantics: The execution of the wait statement delays the execution of the related component until the point in time specified by its argument. If the argument holds a value that precedes the current time value, an error verdict is set. The wait statement has no impact on the overall sampling. All stream ports of the given component are still sampled according to their sampling rate.

Listing 4.5: The wait statement

```
y_1.value = 10.0;
wait(100.0 + now);              // suspends the execution of          2
                                // a component until 100.0
                                // seconds after the start of         4
                                // the testcase
y_1.value = 12.0;                                                     6
```

Syntactical structure (concrete syntax) 3.

WaitStatement ::= *WaitKeyword* "(" *Expression* ")"

4.2 Data streams

In computer science, the term "data stream" is used to describe a continuous or discrete sequence of data. Normally, the length of any given stream cannot be established in advance, and the data rate, i.e. the number of samples per unit of time, may vary. Data streams are continuously processed and are particularly suited to the representation of dynamically-evolving variables over time. Streams are, thus, an ideal representation of the different kinds of discrete and continuous signals mentioned in the beginning of Section 2.3.1. In standard *TTCN-3*, interactions between the test components and the SUT are achieved by the sending and receiving of messages through ports; the interaction between continuous systems, on the other hand, can be represented via so-called streams. In contrast to scalar values, a stream represents the whole allocation history applied to a particular port. In computer science, streams are widely used to describe finite or infinite data flows. So-called timed streams [19] represent the relation to time. Timed streams are streams which provide timing information for each stream value and, thus, make timed behaviour traceable. *TTCN-3 embedded* provides timed streams. In the following chapter, the term "measurement (record)" is used to denote a unit of stream value and the related timing in timed streams. Thus, in the context of continuous data, a measurement record represents an individual measurement which consists of a stream value. The said stream value represents the data side and timing information: the temporal perspective of the measurement. Currently *TTCN-3* offers no direct support for the

specification, management and modification of data streams. In *TTCN-3 embedded*, two different and incompatible representations of data streams are introduced. Consequently, in *TTCN-3 embedded*, a stream is considered to be a sequence of samples, wherein each sample provides information about its timing and value perspective.

- Static perspective: the static perspective provides a direct mapping between a timed stream and the *TTCN-3* data structures *record* and *record-of*. This kind of mapping is referred to below as the static representation of a data stream and allows random access to all elements of the data stream.

- Dynamic perspective: to provide dynamic, online access to data streams and their content, the existing concepts of *TTCN-3* port type and port are extended. A so-called *stream port* references exactly one data stream and provides access to the dynamically-changing values of the referenced data stream.

To denote exemplary streams or stream values, a tabular notation is used. This tabular notation expresses the relation between the value axis of a stream and the respective time axis. Such a stream table has three rows: the first one represents the values of the stream and the second and third represent the temporal perspective. While the second row provides timestamp information synchronized with the overall clock, the third row provides a relative time value, which denotes the distance to the preceding stream element. The following example shows a stream table that defines a stream with the length of 1.4 seconds and float values that change between 1.0 and 1.5.

Stream segment 1:

value	1.2	1.4	1.5	1.7	1.7	1.5	1.2	1.0	1.1	1.4	1.5	1.2
timestamp	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1
delta	0.0	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

4.2.1 Data streams: static perspective

A *TTCN-3 embedded* data stream can be mapped directly onto existing *TTCN-3* data structures. The mapping considers each stream to be denoted with a *TTCN-3* record-of data structure. The individual entries of

this record-of data structure represent the state of the stream at a given point in time. This entity, a so-called sample, represents thus either a measurement of an incoming stream, or a stimulus that is to be applied to an outgoing stream. A sample is represented as a *TTCN-3* record data structure. Such a record is formed by three fields, which are similar to the three columns in a stream table (see Stream Segment 1). The first value field is a generic value field, which allows the instantiation of values for arbitrary (predefined or user-defined) *TTCN-3* data types. The value field (i.e. *value_*) represents what is called the "value of a stream". Its data type has to be aligned to the data type of the represented stream.

The second and third fields describe the temporal perspective of a sample. As its name suggests, the timestamp field provides time stamp information, which is synchronous with the global clock. The delta field, on the other hand, denotes the temporal distance to the preceding sample (the sampling step size delta). The second field and the third field are float values and they represent time values using units of *seconds*. The following example shows the definition of a data structure that specifies individual samples.

Listing 4.6: The definition of a stream data record

```
type record Sample<T>{
  T value_,                                                    2
  float timestamp_,
  float delta_                                                 4
}
```

Because of this underlying structure, a timed-data stream with an arbitrary data type is modelled as a record-of sample (see Listing 4.7).

Listing 4.7: The definition of a record of stream data records

```
type record of Sample<T> MyStreamType<T>;                      1
```

The static representation of data streams can be used for the online and offline evaluation of streams; it can also be used for the partial in-memory definition of streams or stream templates to be applied to stream ports in subsequent test case executions. The static representation of streams can thus be used to assess incoming streams and to define outgoing, or reference, streams and template streams (mostly by means of ordinary *TTCN-3* operations and control structures). The static representation as such can also provide an ideal interface between ordinary *TTCN-3* concepts and *TTCN-3 embedded* concepts. The following example shows a specification of a sampled stream via record of samples.

Stream segment 2:

value	0.0	0.0	0.2	0.1	0.0
timestamp	0.0	0.1	0.3	0.4	0.7
delta	0.0	0.1	0.1	0.1	0.3

Listing 4.8: A record of samples

```
var MyStreamType<float> myStreamVar := {
 {value_:=0.0, timestamp_:=0.0, delta_:=0.0},
 {value_:=0.0, timestamp_:=0.1, delta_:=0.1},
 {value_:=0.2, timestamp_:=0.3, delta_:=0.1},
 {value_:=0.1, timestamp_:=0.4, delta_:=0.1},
 {value_:=0.0, timestamp_:=0.7, delta_:=0.3}
}
```

The creation of larger streams via a manual specification approach is not feasible. The data-processing capabilities of *TTCN-3* are much better suited to the programmatic/algorithmic construction of desired record structures.

The data structures presented in this section are for illustration purposes only. They show how timed-data streams can be mapped onto existing *TTCN-3* data structures and thus processed easily using existing *TTCN-3* language features and operators. *TTCN-3 embedded* does not explicitly offer the type declarations above as part of the language extensions. On the other hand, those operations in *TTCN-3 embedded* that deal with the sampled representation of data streams will provide data structures fully compatible with the scheme explained above.

4.2.2 Data streams: dynamic perspective

In *TTCN-3*, ports are used for the communication with the outside world, i.e. for communication with other test components and with the SUT. To be able to initiate, modify and evaluate stream-based communication between the entities of a test system, *TTCN-3 embedded* extends the concepts of *TTCN-3* port types and ports to include the notion of stream-based communication and stream ports. Stream ports are the endpoints of stream-based communication between test components and the SUT. Stream ports in *TTCN-3 embedded* are thus used to provide access to streams, their values and their respective timing information. A stream port in *TTCN-3 embedded* references exactly one data stream and provides access to that particular stream's values and timing information. The language concepts and extensions that define stream ports and access to stream data are explained in the following sections.

4.2.3 Defining stream port types

Ports facilitate communication between test components, and between test components and the test system interface. *TTCN-3* currently supports message-based and procedure-based ports. *TTCN-3 embedded* allows for the additional definition of stream-based ports. In *TTCN-3*, each port is defined as message-based, procedure-based (or both at the same time as described by Clause 7.4.1 in [65]), or stream-based. The keyword *stream* identifies stream-based ports.

Ports in *TTCN-3* are directional. The directions are specified by the keywords *in* (for the in direction), *out* (for the out direction) and *inout* (for both directions). In contrast to *TTCN-3* and message- and procedure-based ports, each stream port type definition must have one and only one entry, indicating both the allowed type and the allowed communication direction.

Listing 4.9: Stream port definition

```
// Stream-based port which allows stream values of
// type float to be received                                    2
type port StreamIn stream { in float }
                                                                4
// Stream-based port which allows stream values of
// type float to be send                                        6
type port StreamOut stream { out float }
```

Syntactical structure (concrete syntax) 4.

PortDef	::=	*PortKeyword PortDefBody*
PortDefBody	::=	*PortTypeIdentifier PortDefAttribs*
PortKeyword	::=	"`port`"
PortTypeIdentifier	::=	*Identifier*
PortDefAttribs	::=	*StreamAttribs* | *MessageAttribs* | *ProcedureAttribs* | *MixedAttribs*
StreamAttribs	::=	*StreamKeyword* "{" *StreamDirection Type* "}"
StreamList	::=	*Direction AllOrTypeList*
StreamDirection	::=	*InParKeyWord* | *OutParKeyword*
StreamKeyword	::=	"`stream`"

Listing 4.10: Stream port definition with sampling

```
// Stream-based port which allows stream values of type         1
// float and which is sampled every millisecond
type port StreamOut stream { out float }                        3
with { stepsize "0.01" };
```

4.2.4 Defining data stream ports

The definition of stream-based ports is actually quite similar to the definition of message-based and procedure-based ports. The component type defines

which ports are associated with a particular component. These definitions are made in the module definitions part. The port names in a component definition are local to that component, i.e. another component may have ports with the same names. A component type can also have ports with different communication characteristics (e.g. stream-based ports, message-based ports, and procedure-based).

Listing 4.11: The definition of stream ports

```
type port FloatStreamIn stream { in float }
type port FloatStreamOut stream { out float }

type component Tester {
  port FloatStreamIn x_1, x_2, x_3;
  port FloatStreamOut y_1, y_2, y_3;
}
```

Outgoing-stream ports start to emit stream values right after the starting of the component which contains the respective stream port. The same applies for incoming-stream ports. They start receiving data right after the component has been started. Both incoming and outgoing stream ports are updated for each sampling step. If no explicit step size is defined (via step size annotations on module level, test case level, port type level, etc.) the port is initially sampled with the test systems' base sampling, which is the smallest-available step size.

Outgoing-stream ports may already be initialized before their first use, so that their values before and at time 0.0 (start of the test) are defined. The initialization occurs in the context of their declaration.

Listing 4.12: Initialization of stream ports

```
type component Tester {
  port FloatStreamIn x_1, x_2, x_3;
  port FloatStreamOut y_1:=1.0, y_2:=2.0, y_3;
}
```

Outgoing stream ports, which are not explicitly initialized, are automatically initialized with a default value. The default values for the *TTCN-3* basic data types can be found in the following table.

float	integer	boolean	charstring	bitstring	octetstring
0.0	0	False	""	'0'B	'00'O

The initial stream port value for an outgoing-stream port applies to the time point 0.0 and, so long as no other stream value is set, for the following sample steps. The value initialization for incoming streams is the responsibility of the data provider; either the system adapter, or the emitting component (in case of a PTC), is responsible for initializing the streams.

Stream ports can be mixed with message- and procedure-based ports in the same component definition. Thus, the main concepts of standard *TTCN-3* and *TTCN-3 embedded* can be mixed and used together. In this case, the sending of messages and the snap shot semantics are both synchronized with the base sampling rate. Sending of new messages and reception of messages are therefore synchronized with the sampling rate and limited to the beginning of a new sampling step.

Listing 4.13: The definition of stream and message ports

```
type port FloatStreamIn stream { in float }
type port FloatStreamOut stream { out float }                    2
type port FloatMessageIn message { in float }
                                                                 4
type component Tester {
  port FloatStreamIn x_1, x_2, x_3;                              6
  port FloatStreamOut y_1, y_2, y_3;
  port FloatMessageIn m_1, m_2, m_3;                             8
}
```

Syntactical structure (concrete syntax) 5.

PortInstance	::=	*PortKeyword PortType PortElement*
		{"**,**" *PortElement*}
PortElement	::=	*PortIdentifier* [*ArrayDef*]
		[*AssignmentChar PortInitialValue*]
PortIdentifier	::=	*Identifier*
PortInitialValue	::=	*Expression*

4.2.5 Stream-access operations

Similarly to message-based and procedure-based communication, *TTCN-3 embedded* allows the examination of the data of incoming stream-based ports and the control of the data provision at outgoing stream-based ports. In general, access to the current sample of a stream (i.e. the stream value, the respective timing and sampling information) is provided by means of *stream-data operations*. Furthermore, access to the preceding samples is provided via dedicated stream-navigation operations. Last, but not least, it is possible to extract record-structured stream data (as presented in Section 4.2.7) by means of the history operation.

In contrast to message-based and procedure-based communication, stream-data operations and stream-navigation operations are integrated on the expression level. This makes it possible to directly assign values to streams and to read values from streams by means of *TTCN-3* assignments. The stream-data operations and the stream-navigation operations are designated for use inside an automaton environment. They will, therefore, be used particularly

within the hybrid-automata-like control flow structures described in Section 4.4.

Below, the complete syntactical integration of the stream-data operations, the *stream-navigation operations*, and the *stream-evaluation operations* is shown. In the following chapters, these operations and statements will be defined and explained, step-by-step.

Syntactical structure (concrete syntax) 6.

OpCall	::=	*VerdictOps* | *TimerOps* | *TestcaseInstance*
		| *FunctionInstance* | *TemplateOps*
		| *ActivateOp* | *StreamDataOps*
		| *StreamNavigationOps* | *StreamEvalOps*
StreamDataOps	::=	*StreamValueOp* | *StreamTimestampOp*
		| *StreamDeltaOp*
StreamNavigationOps	::=	*StreamPrevOp* | *StreamAtOp*
		["`Dot`" *StreamDataOp*]
StreamEvalOps	::=	*StreamHistoryOp*

The value operation

Each data stream connected to a stream port can be used to access its current value via the value operation. In the case of incoming streams, the value operation yields the current value available at a stream port.

Listing 4.14: Accessing input values of a stream

```
// accessing the current input value of a stream                    1
var float myVar:=x_1.value;
```

Listing 4.15: Comparing input values of a stream with expectations

```
// accessing the current input value of a stream
// and compare it with a given expectation                          2
if (x_1.value>= 100.0) {...};
```

The value (provided by the value operation) is the value measured at the beginning of the current sampling period.

In the case of outgoing streams, the value operation yields a handle that allows the assignment of data to a stream. The assignment has to be type compatible with the data type of the stream.

Listing 4.16: Setting output values of a stream

```
// setting the current output value of a stream                     1
y_1.value:= 100.0;
```

Furthermore, the use of the value operation can be combined such that *TTCN-3 embedded* supports the specification of complex equations and equation systems. This is particularly important because many physical problems

and dynamic phenomena are expressed, in particular, by such equations. The following example shows the *TTCN-3 embedded* realization of a simple model that calculates the velocity of a braking car. Let *y_1, x_1* be stream ports that show the actual velocity of the car (in metres per second) and the brake pedal angle (as a percent value of its maximum angle). The variable *b_max* defines the maximum-achievable negative acceleration (when the brake pedal is fully pressed). The term *v.delta* provides us with the duration of the sampling period.

Listing 4.17: velocity of a braking car

```
// calculating the velocity of a braking car
y_1.value:= y_1.value -  ((x_1.value * b_max / 100)* y_1.delta);
```

The effectiveness of the stream port valuation is generally delayed. A value assigned to a stream port value handle becomes effective inside and outside the component only at the beginning of the next sampling step.

Syntactical structure (concrete syntax) 7.

StreamValueOp	::=	*Port Dot PortValueOp*
PortValueOp	::=	*ValueOpKeyword*
ValueOpKeyword	::=	"`value`"

The timestamp operation

Similarly to the value operation, the timestamp operation provides access to the time-related information of the current sample. The application of the timestamp operation to a stream port thus yields the exact time point at which the current stream port value was measured. The sample time denotes the moment at which a stream value was made available to the test system's input and, therefore, is strongly dependent on the sampling rate.

The time point is represented as a floating-point number (float data type in *TTCN-3*) and is measured with the physical unit of seconds. The timing information is completely synchronized with the test-system clock that is described in Section 4.1.1.

Listing 4.18: Access to sample time

```
// access to the sample time for the current sample
var float measurementTime:=x_1.timestamp;
```

Syntactical structure (concrete syntax) 8.

StreamValueOp	::=	*Port Dot PortTimestampOp*
PortTimestampOp	::=	*TimestampOpKeyword*
TimestampOpKeyword	::=	"`timestamp`"

It should be noted that data streams represent samples in a dynamic measurement process. A sample taken from a data stream is usually historical information, which is to say that it usually represents the state of the system (i.e. the SUT) from a moment in the past.

The delta operation

In *TTCN-3 embedded*, the step size of a data stream can dynamically change during the execution of a test. The change can be initiated either by the test specification, or by the measurement system (i.e. the system adapter). In addition to the timestamp operator, *TTCN-3 embedded* allows access to the step size that has been used to measure a certain value. This information is produced by the delta operation. The delta operation can be used similarly to how the value and the timestamp operations are used: it returns the size of the last sampling step as a floating point number (in seconds).

Intuitive semantics: The delta operation allows read and write access to the current step size of a port. When the delta operation is used for reading, it yields the current step size for a given port. When the delta operation is used for writing, it sets the length of the step size for future writing (and reading) at the given port. The step size is defined as a floating point number and is measured in seconds. A value assigned to a stream port delta handle affects the length of the following sampling period, not the current one. The delta operation cannot, therefore, be used to alter the current sampling step.

Listing 4.19: Accessing the current step size

```
var float actualStepSize;
// reads the current step size from a stream port                    2
currentStepSize: = x_1.delta;
```

Listing 4.20: Setting the current step size

```
// sets the current step size for a stream port                      1
x_1.delta:= 0.01;
y_1.delta:= 0.001;                                                   3
```

Syntactical structure (concrete syntax) 9.

StreamValueOp	::=	*Port Dot PortDeltaOp*
PortDeltaOp	::=	*DeltaOpKeyword*
DeltaOpKeyword	::=	"`timestamp`"

4.2.6 Stream-navigation operations

In addition to enabling access to the current values of a stream, *TTCN-3 embedded* embedded provides additional access to the history of streams. It does this via so-called stream-navigation operations. A navigation operation produces a handle, and a handle allows the application of the value, timestamp or delta operations onto preceding stream states. The preceding state is identified using two different operations. The prev operation backtracks the sample steps, beginning with the current step. It uses an integer index value as an optional guiding parameter that defines the number of sampling steps to be backtracked. The at operation uses time indices as mandatory parameters; these parameters denote the passage of time since the beginning of the test case. The stream-navigation operations are intended for use inside an automaton environment. They will, therefore, be particularly important within the hybrid-automata-like control flow structures described in Section 4.4.1.

The prev operation

The prev operation produces a handle for obtaining stream-related information about the previous states of a given stream.

Intuitive semantics: The prev operation can be parameterized with an index parameter and it returns a handle for retrieving values, timestamps and sampling step sizes from previous stream states. The optional index parameter defines the number of samples counted back in the stream's history. The prev operation should be used in combination with the value operation, the timestamp operation and/or the delta operation in order to provide the most useful results.

Listing 4.21: The prev operation used to access stream values

```
x_1.prev(0).value; // yields the actual stream value                1
x_1.prev.value;    // yields the previous stream value
x_1.prev(1).value; // yields the previous stream value               3
x_1.prev(2).value; // yields the stream value 2 steps ago
```

It should be noted that the expressions *x_1.**prev*** and *x_1.**prev(1)*** yield identical results.

Listing 4.22: The prev operation used to access timestamps and sample step length

```
x_1.prev(0).timestamp;   // yields the timestamp that
                         // denotes the beginning the actual
                         // sampling step
x_1.prev(0).delta;       // yields the length of the last
                         // sampling step
x_1.prev(1).timestamp;   // yields the timestamp that
                         // denotes the beginning the preceding
                         // sampling step
x_1.prev(1).delta;       // yields the length of the
                         // sampling step 2 steps ago
```

Stream segment 3:

value	1.2	1.7	1.7	1.5	1.2	1.0	1.1	1.4	1.5	1.2	1.0	1.1	1.4
timestamp	0.0	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4
delta	0.0	0.3	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Listing 4.23: The prev operation (applied directly after Stream Segment 3 has been recorded, i.e. now yields 1.4)

```
x_1.prev(0).value;              // yields 1.4
x_1.prev.value;                 // yields 1.1
x_1.prev(1).value;              // yields 1.1
x_1.prev(2).value;              // yields 1.0

x_1.prev(0).timestamp;          // yields 1.4
x_1.prev(0).delta;              // yields 0.1
x_1.prev(1).timestamp;          // yields 1.3
x_1.prev(1).delta;              // yields 0.1
```

Syntactical structure (concrete syntax) 10.

StreamValueOp ::= *Port Dot PortPrevOp*
PortPrevOp ::= *PrevOpKeyword* ["(" *IndexValue* ")"]
IndexValue ::= *Expression*
TimestampOpKeyword ::= "`prev`"

The at operation

The at operation returns a handle for obtaining stream-related information for previous states of a stream. These previous states are identified by means of a timestamp value.

Intuitive semantics: The at operation is parameterized with an index parameter i and returns a handle for retrieving values, timestamps and sampling step sizes for preceding states of a stream. The mandatory index parameter represents a timestamp, which identifies a sample at a particular place in time. The timestamp denotes the time that has passed since the start of the test case (see Section 3.1). It references a sample with either the same timestamp, or, if such a sample does not exist, the sample with the next-smaller timestamp. The at operation should be used in combination with the value operation, the timestamp operation and/or the delta operation in order to provide the most relevant and useful results.

Listing 4.24: The at operation used to access stream values

```
x_1.at(now).value;    // yields the current stream value                1
x_1.at(0.0).value;    // yields the initial stream value
                      // (i.e. the stream value at beginning            3
                      // of the test case)
x_1.at(10.0).value;   // yields stream value at the time                5
                      // point 10.0 (i.e. 10. Seconds after
                      // the beginning of the test case)                7
```

Listing 4.25: The at operation used to access timestamps and sample step length

```
x_1.at(now).timestamp;    // yields the beginning of the                1
                          // current sampling step
x_1.at(0.0).timestamp;    // yields the beginning of the                3
                          // initial sampling step (i.e. 0.0)
x_1.at(10.0).timestamp;   // yields the beginning of the                5
                          // sampling step at time point 10.0
```

Listing 4.26: The at operation (applied directly after Stream Segment 3 has been recorded, i.e. now yields 1.4)

```
x_1.at(now).value;              // yields 1.4
x_1.at(0.0).value;              // yields 1.2                           2
x_1.at(1.0).value;              // yields 1.5
x_1.at(1.09).value;             // yields 1.5                           4

x_1.at(now).timstamp;           // yields 1.4                           6
x_1.at(0.0).timstamp;           // yields 0.0
x_1.at(1.09).timestamp;         // yields 1.0                           8
```

Syntactical structure (concrete syntax) 11.

StreamAtOp ::= *Port Dot PortAtOp*
PortAtOp ::= *AtOpKeyword* ["(" *TimeIndexValue* ")"]
TimeIndexValue ::= *Expression*
Topknot ::= "at"

4.2.7 The history operation

The history operation is a means by which the complete, or partial, history of a stream as a *TTCN-3* record-of data structure (see Section 4.2.1) can be obtained. The history operation takes two parameters that characterize the segment using 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 4.27: Setup to apply the history operation

```
type record FloatSample { float value_, float timestamp_, float delta_ }
type port FloatStreamIn { in float }                                        2
type component {
  port x_1 FloatStreamIn                                                    4
}
                                                                            6
var record of FloatSample myStreamRec;
...                                                                         8
myStreamRec:=x_1.history(0.0, now);
```

Intuitive semantics: The history operation provides a sample representation of a stream based on record-of data structures. The operation takes two parameters: one denoting the start time and one denoting the end time of the stream segment designated for export. The parameters are both floating types of data. They represent the time that has passed since the beginning of the test case. Time values are given in units of seconds. The first parameter consists of the time at the moment of the first stream entry (sample) to be considered for the history export. The second parameter states the time of the last record. If the specified start-time value is greater than the end-time value, the history operation results in an empty record-of data structure.

The first example illustrates the use of the history operation in order to obtain the complete history of a stream-based port.

Listing 4.28: Application of the history operation until the current time

```
myStreamRec:= x_1.history(0.0, now);                                        1
```

In the second example, the history operation is applied in Line 2. This yields a record-of structure that represents the first ten seconds at the *ti_Engine_Speed* stream-based port.

Listing 4.29: Application of the history operation for a duration of 10 seconds

```
type record of Measurement<float> Float_Stream_Records;
var Float_Stream_Records speed:=                                            2
    x_1.history(0.0,10.0);
```

The overall size of the record-of data structure (i.e. the number of individual measurement elements) depends on the given sampling rate and on the time interval defined by the parameters of the history operation.

Listing 4.30: The size of a record of samples (data relate to Stream Segment 3)

```
myStreamRec:=x_2.history(0.0, now);
// yields
// {{1.2,0.0,0.0},{1.7,0.1,0.1},{1.7,0.1,0.1},{1.5,0.1,0.1},
//  {1.2,0.1,0.1},{1.0,0.1,0.1},{1.1,0.1,0.1},{1.4,0.1,0.1},
//  {1.5,0.1,0.1},{1.2,0.1,0.1},{1.0,0.1,0.1},{1.1,0.1,0.1},
//  {1.4,0.1,0.1}
// }
```

Syntactical structure (concrete syntax) 12.

StreamHistoryOp	::=	*Port Dot PortHistoryOp*
PortHistoryOp	::=	*HistoryOpKeyword* ["(" *StartValue* ["," *EndValue*] ")"]
StartValue	::=	*Expression*
EndValue	::=	*Expression*
HistoryOpKeyword	::=	"history"

4.2.8 Limiting the length of the stream history

A *TTCN-3 embedded* test system may have problems managing the large amount of data stored in the history of a stream. This is especially the case if the sampling rates are high, or if the duration of a test run is long and restricts the real-time capabilities of the test system. The amount of data to be managed can be reduced by limiting the length of the history of a stream. This can be accomplished by means of the history annotation. The history annotation consists of a history keyword and a charstring value. The charstring value represents either the maximum number of elements, or the maximum temporal length of the stream's port history. When the charstring is a float or integer number directly followed by the character "s", it is assumed to represent a time value in seconds. It is applicable to modules, test cases, groups, component types and stream port types. The charstring value affects the history length for stream ports that are contained in one of these entities, or, in case of stream port types, the value affects the the respective instances of a stream port.

Intuitive semantics: The history annotation defines the overall length of the history of a stream port. The history always provides the most recent data and the history annotation causes, that the oldest stream elements are removed, when the history has reached its limit. Thus, the elements that

can be accesses by the stream-navigation operation and the stream-history operation are reduced to the time-frame that is defined by the history annotation. If the stream-navigation operation and the stream-history operation are used to access data, that are no longer in the history of a stream port, the error verdict is set.

Listing 4.31: Setting the length of a stream for all ports in a module

```
// Setting the length of a stream for all ports in a module
// to 10 elements
module myModule{
...
}
with {history "10"};
```

Listing 4.32: Setting the length of a stream for all ports addressed in a testcases

```
// sets the stream history length for all ports adressed in a testcase
// to a value of 10 seconds
testcase myTestcase() runs on myComponent{
...
}
with {history "10s"};
```

Listing 4.33: Setting the length of streams of a given port type

```
type port StreamOut stream { out float}
with {history "10"};
```

Syntactical structure (concrete syntax) 13.

WithAttribList	::=	{ *SingleWithAttrib* [*SemiColon*] }
SingleWithAttrib	::=	*AttribKeyWord* [*OverrideKeyword*] [*AttribQualifier*] *AttribSpec*
AttibKeyword	::=	*EncodeKeyword* | *VariantKeyword* | *DisplayKeyword* | *StepsizeKeyword* | *HistoryKeyword* | *ExtensionKeyword*
HistoryKeyword	::=	"`history`"

4.3 The assert statement

The assert statement is used as shorthand for the specification of expected system behaviour.

Intuitive semantics: The assert statement specifies a predicate that expresses expectations about the SUT and an optional list of log items that can be communicated via the logging interface. The predicate consists of an arbitrary *TTCN-3* Boolean expression. If the predicate fails, the assert statement automatically sets the verdict to "fail" and transmits the log items to the logging system. The assert statement is allowed at any place in the

TTCN-3 source code where the application of the set-verdict statement is allowed. To assess continuous data, the assert statement is particularly useful within the hybrid-automata-like control flow structures described in Section 4.4.1.

Listing 4.34: The assert statement

```
assert(x_1.value==4.0);
```

Listing 4.35: The assert statement with a log message

```
assert(x_1.value==4.0, "Value is not 4.0");
```

Syntactical structure (concrete syntax) 14.

VerdictStatements ::= *SetLocalVerdict* | *AssertStatement*
AssertStatement ::= *AssertKeyword* ["(" *AssertionList* ")"]
AssertKeyword ::= "assert"
AssertList ::= *Expression* { "," *Expression* }

It should be noted that the semantics of the assert statement can be mapped to existing *TTCN-3* statements in the following way:

Listing 4.36: The semantics of the assert statement

```
assert (pred1, pred2,..., predn);
// is fully equivalent to
if(! pred1) setverdict(fail);
if(! pred2) setverdict(fail);
...
if(! predn) setverdict(fail);
// and
assert (pred1, pred2,..., predn,``reason'');
// is fully equivalent to
if(! pred1) setverdict(fail, ``reason'');
if(! pred2) setverdict(fail, ``reason'');
...
if(! pred3) setverdict(fail, ``reason'');
```

4.4 Control structures for continuous and hybrid behavior

Currently, *TTCN-3* control flow structures do not directly support the parallel and sampled application and assessment of stream values at ports. The concepts defined in Section 4.2 allow the construction, application and assessment of individual streams. For more advanced test behaviour, such as the concurrent application and assessment of multiple streams or the detection of complex events (e.g. zero crossing or flag changes at multiple ports), different concepts are necessary. To this end, the concepts defined in the last sections are combined with state-machine-like specification concepts, the so called *modes*.

A mode expresses a certain runtime mode of a system or a system under test. This kind of runtime mode is characterized by a defined behaviour at ports and a set of predicates that limit the applicability of the behaviour. Unlike the ordinary behaviour of *TTCN-3* statements, a mode applies its behaviour over time (at least for one sampling step).

4.4.1 Modes

The term mode is used to specify the discrete and countable macro states of a dynamic hybrid system. It mainly serves to distinguish the macro states of a hybrid system from the (in theory, infinite) number of micro states. Modes provide a layer of abstraction that helps to distinguish between two kinds of discrete changes in a hybrid system: discrete changes that are relevant from the user's and tester's perspective, and discrete changes introduced by the underlying test environment in order to map continuous behaviour to a computational environment (which is naturally discrete). The interpretation and calculation of micro states depend on the underlying technical environment, i.e. the sampling. A micro state is thus calculated by combining the active macro states and the sampled evaluation of data at the stream ports.

Modes, and the transitions between them, can be represented by a state-machine-like structure defined in the theory of hybrid automata [80, 5]. Figure 1 shows an abstract test specification that consists of three atomic modes, transitions between them, and invariants and assertions that must hold during their execution.

In order to realize such hybrid automata in *TTCN-3*, three new block statements are required: the cont statement, the seq statement, and the par statement. The cont statement is used for the specification of atomic modes, and the par and seq statements are used to aggregate modes into larger constructs via parallel and sequential composition.

Modes, in general, are characterized by their duration and their internal behaviour (i.e. the assignment and assessment of values at stream ports). The duration, or rather the duration of the mode's activity, is defined by a set of predicates, which relate to time or to the valuation of (stream) ports, variables, etc. A mode specification in *TTCN-3 embedded* consists of five syntactical compartments:

- an obligatory *body* for the specification of the mode's internal behaviour;
- an *invariant* block that defines predicates that must hold while the mode is active;

- optional *on-entry* and *on-exit* blocks that define behaviour that is to be executed only once, at the activation or deactivation of a mode;

- and a *transition* block that defines the exit conditions which end the mode's activity and the actions that must be taken when the transition fires

Listing 4.37 shows the definition of an atomic mode. This atomic mode consists of: two assignments to stream ports; an invariant that checks the state of an outgoing stream port; an on-entry block that initializes the variable x; an on-exit block that resets the stream port *to_Set_Point* to the value of 0.0; and transitions that check the value of an incoming stream port.

Listing 4.37: Definition of atomic modes

```
cont { //body
onentry { x:=10.0; }                                    2
inv { //invariant
        y_1.value > 20000.0;                            4
}
   y_1.value:=3.0 * now;                                6
   y_2.value:=0.0 + x;
onexit { y_1.value:=0.0 }                               8
}
until { //transition                                    10
  [x_1.value > 2000.0] { y_2.value:=2.0; }
  [x_1.value > 3000.0] { y_2.value:=1.0; }              12
}
```

In general, the structural setup of composite modes is nearly identical to that of atomic modes. The only differences between composite and atomic modes lie in their behavioural descriptions. Atomic modes contain assignments and assert statements; composite modes, on the other hand, contain other modes. In terms of invariants, on-entry and on-exit blocks, and transitions, the structural setup and behaviour of composite modes and atomic modes are identical.

When a mode is active, each invariant of a composite mode has to hold steady. When a transition fires in a composite mode, the activity of the live mode stopped. Listing 4.38 shows the setup of a parallel mode that contains two sequential modes. Each of the sequential modes contains further atomic modes.

Syntactical structure (concrete syntax) 15.

ModeSpecification	::=	(*BasicMode* | *ComplexMode*) [*UntilBlock*]
InvariantBlock	::=	*InvKeyword* "{" [*InvariantList*] "}"
InvKeyword	::=	"inv"
InvariantList	::=	*BooleanExpression* { "," *BooleanExpression* }
UntilBlock	::=	*UntilKeyword* ("{" *UntilGuardList* | *GuardOp* "}")
		| ("(" *BooleanExpression* ")")
UntilKeyword	::=	"until"
UntilGuardList	::=	{ *UntilGuardStatement* }
UntilGuardStatement	::=	*UntilGuardChar* [*GuardOp*] *StatementBlock*
		[*UntilJump SemiColon*]
UntilGuardChar	::=	"[" [*BooleanExpression* | *ModePredicate*] "]"
UntilJump	::=	*GotoStatement* | *ContinueStatement* | *RepeatStatement*
BasicMode	::=	*ContKeyword* "{" { *Declaration* } [*OnEntryBlock*]
		[*InvariantBlock*] { *BasicModeOp* } [*OnExitBlock*] "}"
ContKeyword	::=	"cont"
BasicModeOp	::=	*Assignment* | *AssertStatement* | *InvariantBlock*
		| *IfStatement*
ComplexMode	::=	(*SeqKeyword* | *ParKeyword*) "{" { *Declaration* }
		[*OnEntryBlock*] [*InvariantBlock*]
		{ *ComplexModeOp* } [*OnExitBlock*] "}"
ParKeyword	::=	"par"
SeqKeyword	::=	"seq"
ComplexModeOp	::=	*ModeSpecification* | *LabelStatement* |
		FunctionOrModeInstance
Declaration	::=	*FunctionLocalInst*

Listing 4.38: Mode composition

```
par {                                                    1
  inv { //invariant
  }                                                      3
  seq { // stimulation sequence
    cont { // stimulation action 1                       5
    }
    cont { // stimulation action 2                       7
    }
    // ...                                               9
  }
  seq { // assessment sequence                           11
    cont { // assessment action 1
    }                                                    13
    cont { // assessment action 2
    }                                                    15
    // ...
  }                                                      17
}
until { // transitions                                   19
}
```

4.4.2 Definition of the until block

The until block allows the specification of two aspects of modes: the exit conditions for modes and the explicit transitions between modes. The entries of the until block are called transitions, and each transition specifies conditions for its activation (i.e. guards and trigger events). In addition, transitions may provide an explicit definition of the mode which is to be activated next (this is known as the "target mode"). An until block can contain several alternative transitions, each of which can specify different exit conditions and target modes.

Definition of transition guards and events

The until block also defines a number of transitions between modes. A transition contains either a guard, or a trigger event specification, or both. The guard and the trigger event specification are both used to determine whether a transition can fire or not. A guard is modelled as a boolean *TTCN-3* expression. A trigger event is modelled using *TTCN-3* receiving-operations (receive statement, trigger statement, getcall statement, etc). The guard and/or the trigger may be followed by an optional statement block, which contains instructions that are to be executed upon the activation of the transition. The syntax of the until block is inspired by the syntax of the *TTCN-3* alt statement: the syntactical structure of transitions is similar to the syntactical structure of a *TTCN-3* alt statement.

Intuitive semantics: A transition is considered to have been activated when: the guard expression has been satisfied (when only a guard expression has been specified); a valid receiving event has occurred at the *TTCN-3* receiving operation (when only the receiving-operation has been specified); or, when both the guard expression has been satisfied and a valid receiving event has occurred at the *TTCN-3* receiving operation (when both the guard expression and the receiving operation have been specified).

At each sampling step, the transitions for the live mode are checked. If a transition becomes active, the optional statement block is executed once. Afterwards, the enclosing mode and all of its child modes are deactivated. The control flow continues with the activation of the follow-up mode. The transitions in an until block are checked in the sequential order given by the specification (*TTCN-3 embedded* source code). If multiple transitions exist, the first transition that fulfills the activation conditions is activated.

Listing 4.39: The until block of a mode

```
cont { //mode
  y_1.value:=3.0;
}
until { // transitions
 [x_1.value > 4.0] m_1.receive(TemplExp) {
   log("statement block 1"); }
 [x_1.value > 4.0 and x_1.value > x_2.value] {
   log("statement block 2"); }
 [] m_2.receive(TemplExp) {
    log("statement block 3"); }
}
```

In addition, the following abbreviations or shorthand notations for the specification of simple transitions are allowed.

Listing 4.40: A shorthand for an until block

```
until (y_1.value > 3.0);
// shorthand for until { [ x_1.value > 3.0]{} }
```

Listing 4.41: A shorthand for an until block with a receive statement

```
until ( m_1.receive(tmpl) );
// shorthand for until { []m_1.receive(tmpl){} }
```

In the optional statement block of a transition, any *TTCN-3* statement is, in principle, allowed, except for the following:

- blocking instructions (i.e. receive statements, alt statements) — such constructs can block the execution of the statement block, causing the next sampling step to be missed.
- any type of control-flow-related statement that can lead to the leaving of the enclosing mode (e.g. goto or return).

Syntactical structure (concrete syntax) 16.

UntilBlock	::=	*UntilKeyword* ("{" *UntilGuardList* | *GuardOp* "}")
		| ("(" *BooleanExpression* ")")
UntilKeyword	::=	"`until`"
UntilGuardList	::=	{ *UntilGuardStatement* }
UntilGuardStatement	::=	*UntilGuardChar* [*GuardOp*] *StatementBlock*
		[*UntilJump SemiColon*]
UntilGuardChar	::=	"[" [*BooleanExpression* | *ModePredicate*] "]"
UntilJump	::=	*GotoStatement* | *ContinueStatement* | *RepeatStatement*

Definition of follow-up modes

TTCN-3 embedded allows the explicit definition of follow-up modes via the goto statement. A mode specification can thus have a preceding label that defines the target for a goto jump. Furthermore, each action block of a transition can have an optional goto statement that refers to an arbitrary label.

Goto statements, however, are only allowed after the statement block. If such a transition is activated, the optional statement block is executed and then the execution is continued at the label position with the activation of the next mode. Listing 4.42 shows the application of labels and goto statements in the context of modes.

Listing 4.42: The goto statements in modes

```
label state1;
cont { //mode                                                              2
  y_1.value:=3.0;
}                                                                           4
until (x_1.value > 2.0)
label state2;                                                               6
cont { //mode
  x_1.value:=4.0;                                                           8
} until { // transitions
 [x_1.value > 4.0] { log("statement block 1"); } goto state1;              10
 [x_1.value > x_2.value] { log("statement block 2"); } goto state2;
 [] m_1.receive(TemplExp) { log("statement block 3"); }                    12
}
```

The use of the goto statement is already restricted in standard *TTCN-3*, and *TTCN-3 embedded* embedded defines additional restrictions for its use in the context of modes. Goto jumps are only allowed in a sequential environment: either inside sequential modes, or on the top level of a composition, i.e. directly on the test case level. Furthermore, goto jumps are not allowed to violate the composition hierarchy. It is, therefore, not possible to jump to a parent mode or a child mode. Jumps are only allowed between modes that exist at the same hierarchy level.

If, however, no follow-up mode is explicitly defined via a goto statement, the sequential ordering of mode specification implicitly defines the follow-up mode. Thus, when two atomic modes follow one another in the specification, the second mode is the follow-up mode for all of the active transitions of the preceding mode.

Listing 4.43: Subsequent modes

```
cont { y_1.value := 3.0; } until (x_1.value > 3.0);                         1
cont { y_1.value := 5.0; } until (x_1.value >= 3.0 * x_2.value)
cont { y_1.value := 7.0; } until (x_1.value >= 3.0)                         3
```

Syntactical structure (concrete syntax) 17.

UntilBlock	::=	*UntilKeyword* "{" *UntilGuardList* "}"
UntilKeyword	::=	"until"
UntilGuardList	::=	{ *UntilGuardStatement* } |"(" *BooleanExpression* ")"
UntilGuardStatement	::=	*UntilGuardChar GuardOp StatementBlock*
UntilGuardChar	::=	"[" [*BooleanExpression* | *ModePredicate*] "]"

The use of the repeat statement

The repeat statement triggers the re-execution of a par statement, a seq statement, or a cont statement. This means that the par statement, the seq statement or the cont statement is activated again and the statement is repeated with the next sampling step. When the repeat statement is executed, the local time of the respective mode (see duration operation in Section 4.4.4) is reset. When the repeat statement is executed in the case of composite modes, the child modes are activated according to the type of mode (i.e. according to whether it is parallel or sequential). In addition, the respective on-entry and on-exit blocks are executed.

Listing 4.44: The repeat statement for modes

```
cont { //mode                                                          1
  y_1.value:=4.0;
} until { // transitions                                               3
 [x_1.value > 4.0] { log("statement block 1"); } goto state1;
 [x_1.value > x_2.value] { log("repeat the execution"); } repeat;      5
 [] m_1.receive(TemplExp) { log("statement block 2"); }
}                                                                      7
```

It should be noted that the use of the repeat statement is functionally equivalent to the use of a goto statement that addresses a label directly above the live mode.

The use of the continue statement The continue statement triggers the further execution of a par statement, a seq statement, or a cont statement. This means that these statements are continued into the next sampling step without a reset of the local time (see duration operation in Section 4.4.4). The on-entry and on-exit blocks are not executed.

Listing 4.45: The continue statement for modes

```
cont { //mode                                                          1
  y_1.value:=4.0;
} until { // transitions                                               3
 [x_1.value > 4.0] { log("statement block 12"); } goto state1;
 [x_1.value > x_2.value] { log("continue the execution")} continue;    5
 [] m_1.receive(TemplExp) { log("statement block 2"); }
}                                                                      7
```

4.4.3 Definition of generic mode body elements

The mode body defines the behaviour of a mode. There are different kinds of mode body elements for atomic modes, parallel modes, and sequential modes (see Sections 4.4.5, 4.4.6, and 4.4.7). A mode body can also, however, contain a number of elements that are syntactically and semantically identical across all kinds of modes. These elements are explained in the following sections.

Definition of invariant blocks

An invariant block contains Boolean predicates (expressions) that characterize the applicability of a mode. An invariant block is thus always related to its containing mode specification. It also specifies the terms and conditions of validity for a mode, which are checked during runtime.

Intuitive semantics: For each mode, all invariants are checked during each sampling step when the mode is active. While a mode is active, none of its invariants can be violated. If an invariant of an active mode is violated, the mode must be able to switch to another mode which can tolerate the violation that has occurred. If this is not possible, the test system must give an error verdict. The invariant block is always checked at the beginning of each sampling step, before the body of each mode is executed.

Listing 4.46 shows the definition of an atomic mode that sets the value of outport A continuously to 3.0. The invariant prescribes particular conditions for the values of incoming ports B, C and D. When one of the invariants is violated by a real value at one of the ports, the mode execution is stopped.

Listing 4.46: The invariant-block for modes

```
cont {                                                                    1
  inv { x_1.value > 3.0, x_2.value >= 3.0 * x_3.value }
  y_1.value := 3.0;                                                       3
}
```

The specification of invariants allows for easy and unambiguous definition of conditions that terminate modes. Sequences of modes that are executed one after another (when the invariant state of the active mode changes) are specified via a simple, sequential control flow paradigm.

Listing 4.47: Subsequent modes with invariants

```
cont {
  inv { x_1.value > 3.0, x_2.value >= 3.0 * x_3.value }                   2
  y_1.value:=3.0;
} until(notinv)                                                           4
cont {
  inv { x_1.value <=3.0, x_2.value >= 3.0 * x_3.value }                   6
```

```
  y_1.value:=5.0;
} until(notinv)                                                    8
```

Syntactical structure (concrete syntax) 18.

InvariantBlock	::=	*InvKeyword* "{" [*InvariantList*] "}"
InvKeyword	::=	"inv"
InvariantList	::=	*BooleanExpression* { "," *BooleanExpression* }

Definition of the on-entry block

The on-entry block contains a statement list that is to be executed once, and only once, during the activation of a mode. In an on-entry block, any *TTCN-3* statement is allowed, except blocking instructions (i.e. receive-statements, alt-statements). Blocking instructions are constructs that may block the execution of the statement block. When the statement block's execution is delayed or terminated, the following sampling step can be missed, as can any type of control-flow-related statement that would lead to the leaving of the mode (e.g. the goto statement or the return statement).

Intuitive semantics: The execution of the on-entry block is part of the activation of a mode. For an on-entry block to successfully start, all invariants must hold. In hierarchically-ordered modes, the on-entry blocks are executed sequentially, beginning with the on-entry block of the outermost mode to that of the innermost mode.

Listing 4.48 shows the definition of an atomic mode. This atomic mode sets the sampling of a port during its activation.

Listing 4.48: The onentry-block

```
cont{
  onentry {y_1.delta := 0.001;}                                    2
  y_1.value:=3.0;
}                                                                  4
```

Syntactical structure (concrete syntax) 19.

OnEntryBlock ::= *OnEntryKeyword*
"{" { *NonBlockingStatement* [*SemiColon*] } "}"
OnEntryKeyword ::= "`onentry`"

Definition of the on-exit block

The on-exit block contains a statement list that is to be executed once, and only once, during the deactivation of a mode. In an on-exit block, any *TTCN-3* statement is, in principle, allowed, except for statements which block instructions (i.e. receive statements or alt statements). Blocking constructs prevent the execution of the statement block, and, consequently, the next sampling step can be missed, as can each type of control-flow-related statement that leads to the leaving of the mode (e.g. the goto statement or the return statement).

Intuitive semantics: The on-exit block is executed as a part of the deactivation procedure of a mode. The execution of the on-exit block is triggered either by an activated transition or by the violation of an invariant. Either of these events can lead to the mode's deactivation. In the case of an active transition, the on-exit block is executed directly after the optional action block has been executed. In hierarchically-ordered modes, the on-exit blocks are executed sequentially, beginning with the on-exit block of the innermost mode and moving towards that of the outermost modes.

Listing 4.49 shows the definition of an atomic mode. This atomic mode sets the sampling of a port during its activation time.

Listing 4.49: The on-exit block

```
cont{
   y_1.value := 3.0;                                    2
   onexit {y_1.value := 0.0;}
} until(x_1.value > 3.0)                                4
```

Syntactical structure (concrete syntax) 20.

OnExitBlock ::= *OnExitKeyword*
"{" { *NonBlockingStatement* [*SemiColon*] } "}"
OnExitKeyword ::= "`onexit`"

The not-inv operation in the context of modes

The not-inv operation can be used as a predicate that indicates the violation of any local mode invariant. If a mode is active and one of its invariants

is violated, the evaluation of the not-inv operation yields true. Otherwise, it yields false. The not-inv operation thus allows an explicit response to invariant violation by means of transitions.

Listing 4.50: The not-inv operation

```
cont { //mode
  inv{x_1.value >= 5.0}
  y_1.value := 3.0;
}
until { // transitions
  [notinv] {
    log("Invariant violated"); }
  [] m_1.receive(TemplExp) {
    log("Invariant not violated"); }
}
```

Syntactical structure (concrete syntax) 21.

UntilGuardChar ::= "[" [*BooleanExpression* | *ModePredicate*] "]"
ModePredicate ::= *NotinvKeyword*
NotinvKeyword ::= "notinv"

4.4.4 Local temporal expressions in the context of modes

Within a mode, the time elapsed since the beginning of the test case can be continuously accessed and read. This can be achieved by using the keyword 'now'.

Listing 4.51: The now operation in modes

```
cont{
  y_1.value := 3.0;
}
until (now > 4.0)
// executes the content of the body block until
// the overall test case time has reached 4.0 seconds
```

Measurement of the time elapsed since the enclosing mode construct's activation can be additionally provided via the duration operation. The duration operation is applicable in expressions in all mode-related substructures, such as the body block, the invariant block, and the until block.

Listing 4.52: The duration operation in modes

```
cont{
  y_1.value := 3.0;
}
until (duration > 4.0)
// executes the content of the body block for 4.0 seconds
```

The valuation of the duration operation depends on its context: it may differ depending on its place of application.

The following example shows the application of the duration operation in two different modes. Both modes are activated at different times: the application of the duration operation in the first and in the second cont mode may, therefore, yield different results than the application of the duration operation in the enclosing seq mode.

Listing 4.53: The duration operation in composite structures

```
par{                                                                    1
  cont{ y_1.value := 2.0;} until (duration > 4.0)
  cont{ y_2.value := 3.0;} until (duration > 4.0)                       3
} until(duration > 6.0)
```

Syntactical structure (concrete syntax) 22.

OpCall	::=	*ConfigurationOps* | *VerdictOps* | *TestCaseInstance*
		| *TimerOps* | *FunctionInstance* [*ExtendedFieldReference*]
		| *TemplateOps* [*ExtendedFieldReference*] | *ActivateOp* |
		| *NowOp* | *DurationOp* | *StreamDataOp* |
		| *StreamNavigationOps* | *StreamEvalOps*
NowOp	::=	*NowKeyword*
NowKeyword	::=	"`now`"
DurationOp	::=	*DurationKeyword*
DurationKeyword	::=	"`duration`"

4.4.5 Atomic modes: the cont statement

The cont statement is used to define atomic modes. Atomic modes directly define test behaviour at stream ports by means of value allocation and value assessment. A cont statement, therefore, may contain assignments and assert statements, but also if statements, loops, and function calls. In fact, in a cont statement, any kind of non-blocking *TTCN-3* statement can be used.

Intuitive semantics: The execution of instructions within a cont statement can be theoretically considered as "continuous". Cont statements are executed periodically for each sample step. This kind of time-triggered and periodic execution allows for continuous valuation, periodic revision and regular evaluation of values at stream ports.

When a cont statement is activated, all of the contained elements are executed repetitively, for each sample step. The execution ends when a transition fires or an invariant is violated.

Listing 4.54: The cont statement

```
// executes the assignments at each sample step
cont { // Mode 1                                                        2
  y_1.value := 10.0;
```

```
  y_2.value := 2.0 * duration;
}
until (duration > 5.0)
```

Listing 4.55: The cont statement without until block

```
cont { // mode 1
  y_1.value := x_1.prev.value * 2.0;
  y_2.value := x_1.prev(5).value;
}
```

Listing 4.56: The cont statement with multiple transitions

```
cont { // mode 1
  y_1.value := x_1.prev.value * 2.0;
  y_2.value := x_1.prev(5).value;
  inv{
    y_2.value > 200.0;
  }
}
until { // Transition
  [y_2.value > 150.0] { y_2.value := 0; }
  [y_2.value > 180.0] {}
}
```

Syntactical structure (concrete syntax) 23.

ModeSpecification ::= (*BasicMode* | *ComplexMode*) [*UntilBlock*]
BasicMode ::= *ContKeyword* "{" { *Declaration* } [*OnEntryBlock*] [*InvariantBlock*] { *BasicModeOp* } [*OnExitBlock*] "}"
ContKeyword ::= "cont"
BasicModeOp ::= *ContinuousStatement*

4.4.6 Parallel mode composition: the par statement

The par statement specifies the parallel composition of modes. A parallel composition may contain sequential modes, parallel modes and atomic modes. When a parallel composition is activated, all of its composite (i.e. directly contained) modes are executed.

The general structure of the par statement is similar to that of the cont statement and that of the seq statement. A par statement consists of a body part that defines the overall behaviour of the mode. A par statement's the body part contains the mode definitions that are to be composed in parallel. The body part is preceded by an optional on-entry block and an optional invariant block, and is followed by an optional on-exit block and an optional transition part (see Sections 4.4.3 and 4.4.2). The invariant and the transition parts define the exit conditions that are checked during execution.

Intuitive semantics: The activation of a parallel mode leads to the activation of all of the child modes. During its execution, the parallel mode is responsible for checking the status of all of its contained modes. While it is active, each invariant of a composite mode must hold, and each transition of a composite mode ends the activity of the mode when it fires. Furthermore, each mode includes access to an individual local clock that contains the time elapsed since the mode's activation. The value of the local clock can be obtained using the duration operation. The execution of a parallel mode ends either when a transition in the transition block fires, or when the execution of all child modes is complete.

Listing 4.57: The par statement

```
par {                                                          1
  cont {
    y_1.value := 1.0;                                          3
    y_2.value := 2.0;
  }                                                            5
  until { // Transition
    [x_1.value > 1.0] { }                                      7
    [] m_1.receive(msg1) { }
  }                                                            9
  cont {
    y_1.value := 2.0;                                          11
    y_2.value := 1.0;
  }                                                            13
  until { // Transition
    [x_1.value > 10.0] { }                                     15
    [] m_1.receive { }
  }                                                            17
}
until { // Transition                                          19
  [x.value > 11.0] { }
  [] m_1.receive(msg2) { }                                     21
}
```

Syntactical structure (concrete syntax) 24.

ModeSpecification	::=	(*BasicMode* | *ComplexMode*) [*UntilBlock*]
ComplexMode	::=	(*SeqKeyword* | *ParKeyword*) "{" { *Declaration* } [*OnEntryBlock*] [*InvariantBlock*] { *ComplexModeOp* } [*OnExitBlock*] "}"
ParKeyword	::=	"par"
SeqKeyword	::=	"seq"
ComplexModeOp	::=	*ModeSpecification* | *LabelStatement* | *FunctionOrModeInstance*
Declaration	::=	*FunctionLocalInst*

Par statements organize the execution order of cont statements (as well as the execution order of included seq and par statements); cont tatements, on the other hand, directly define the valuation and assessment of values at ports. In contrast to par and seq statements, cont statements form the fundamental leaves of a hierarchical mode structure.

4.4.7 Sequential mode composition: the seq statement

The seq statement specifies the sequential composition of modes. A sequential composition can contain sequential modes, parallel modes and atomic modes. When activated, a sequential composition leads to the sequential execution of all of its composite (i.e. directly contained) modes.

The general structure of the seq statement is similar to that of the cont statement and that of the par statement. It contains a body part that defines the overall behaviour of the mode. The body party of the seq statement contains the mode definitions that are to be composed. The body part is preceded by an optional on-entry block and an optional invariant block, and is followed by an optional on-exit block and an optional transition part. The invariant and the transtion part define the exit conditions that must be checked during execution.

Intuitive semantics: The activation of a sequential mode prompts the activation of its first child mode. During its execution, the sequential mode is responsible for scheduling the execution of the contained modes in their sequential order. So, when the first child mode has finished, the second child mode is activated; when the second mode has finished, the next child mode is activated; and so on. When a mode is active, each of its composite invariants must hold. Each transition of a composite mode, when fired, ends that mode's activity. Furthermore, each mode includes access to an individual local clock, which returns the time elapsed since the mode's activation. The local clock's value can be obtained via the duration operation. The execution of a sequential mode ends either when a transition in the transition block fires, or when the execution of the last child mode is complete.

The following example defines the sequential execution of two atomic modes. Both modes are composed sequentially, using a sequential mode.

Listing 4.58: The seq statement

```
seq {
  cont {
    y_1.value:=1.0;
    y_2.value:=2.0;
  }
  until { // Transition
    [x_1.value > 2.0] { }
    [] m_1.receive() { }
  }
  cont {
    y_1.value:=2.0;
    y_2.value:=1.0;
  }
  until { // Transition
    [x_1.value > 1.0] { }
    [] m_1.receive { }
```

```
  }
}
until { // Transition
  [x_1.value > 12.0] { }
  [] m_1.receive(msg2) { }
}
```

Seq statements organize the execution order of cont statements (including the execution order of included seq and par statements); cont statements, on the other hand, directly define the valuation and assessment of values at ports.

Syntactical structure (concrete syntax) 25.

ModeSpecification	::=	(*BasicMode* | *ComplexMode*) [*UntilBlock*]
ComplexMode	::=	(*SeqKeyword* | *ParKeyword*) "{" { *Declaration* } [*OnEntryBlock*] [*InvariantBlock*] { *ComplexModeOp* } [*OnExitBlock*] "}"
ParKeyword	::=	"par"
SeqKeyword	::=	"seq"
ComplexModeOp	::=	*ModeSpecification* | *LabelStatement* | *FunctionOrModeInstance*
Declaration	::=	*FunctionLocalInst*

4.4.8 Reusable modes

It is possible to specify modes that can be referenced and parametrized; such modes provide a higher degree of flexibility, which can prove useful. Values, templates, ports and modes can all be used as mode parameters. The definition of such modes is quite similar to the definition of *TTCN-3* functions. In order to be able to use modes as passable parameters, so-called mode types are required. Mode types define the signatures of modes; thus they can be used to classify mode definitions that can referenced. Unlike functions, parametrizable modes are not called in the sense of a function call, but, rather, are inserted by means of a substitution mechanism at compile time. The recursive application of parametrizable modes is thus not possible.

Mode types

Mode types are the set of identifiers of mode definitions with specific parameter lists and runs-on clauses. They denote modes that have a compatible parameter list and compatible runs-on clauses.

Listing 4.59: Mode type definitions

```
// mode type with no parameters
type mode ModeTypeWithoutParameter();
```

```
// mode type with a runs on clause                                    4
type mode ModeTypeWithoutParameter() runs on Tester;
                                                                      6
// mode type with a value parameter
type mode ModeTypeWithValueParameter(in float valueParam);            8

// mode type with a port parameter                                    10
type mode ModeTypeWithPortParameter(FloatStreamIn portParam);
                                                                      12
// mode type with a mode parameter
type mode ModeTypeWithModeParameter(MyModeType modeParam);            14

// mode type with multiple parameters                                 16
type mode ModeTypeWithValueParameter(in float targetVal,
                                     in StreamPort p,                 18
                                     in MdoeType myMode);
```

Syntactical structure (concrete syntax) 26.

ModeType	::=	*TypeDefKeyword ModeKeyword ModeTypeIdentifier* "(" *ModePar* { "," *ModePar* } ")" [*RunsOnSpecOrSelf*]
ModePar	::=	*FormalValuePar* | *FormalTimerPar* | *FormalTemplatePar* | *FormalPortPar* | *FormalModePar*
ModeTypeIdentifier	::=	*Identifier*
RunsOnSpecOrSelf	::=	*RunsKeyword OnKeyword* [*ComponentType* | *SelfOp*]
ModeKeyword	::=	"mode"

Named mode definitions

Like functions and function calls, it is possible to give modes declared names; with a declared name, a mode can be referenced from any context that allows the explicit declaration of modes. A named mode definition allows the definition of reusable modes and, consequently, the referencing (and the use) of the definitions from inside modes, functions or test cases. A mode that can be referenced may be defined within a module; it can also have parameters of arbitrary data types, port types, timers, templates, and mode types. Listing 4.60 shows the definition of named modes and Listing 4.61 shows their use within a test case specification.

Listing 4.60: Named mode definitions

```
mode myMode_1 runs on Tester cont {                                   1
  assert(x_1.value >= 500.0)
}                                                                     3
mode myMode_2(float valueParam, FloatStreamIn portParam) cont {
  assert(portParam.value >= valueParam)                               5
}
```

Listing 4.61: The application of named modes

```
testcase myTestcase() runs on Tester {
```

```
  par {                                                    2
    cont { y_1.value:= 100.0 }
    myMode_1;                                              4
    myMode_2(500.0, x_1);
  }                                                        6
}
```

When the named modes are used as parameters for test cases, functions, or modes, the specification of a named and parametrizable mode must contain reference to the mode type. The number, kind and sequence of parameters used in the mode definition must conform to the referenced specification of mode type.

Listing 4.62: Named modes with parameters

```
type mode MySimpleModeType() runs on Tester;                                1
type mode MyParamModeType(in float val_1,
                          in float val_2,                                   3
                          in ModeType assertion);
// reusable mode declaration                                               5
mode MySimpleModeType assert_mode() runs on Tester cont {
  assert(x_1.value >= 500.0)                                                7
}
mode MyParamModeType stim_seq(in float val_1,                               9
                              in float val_2,
                              in ModeType assertion)                        11
  runs on Tester par {
    seq{ // perturbation sequence                                           13
      cont { y_1.value := val_1 }
      until (duration >= 2.0)                                               15
      cont { y_1.value := val_1 +  duration / y_1.delta * val_2 }
      until (duration >= 5.0)                                               17
    }
    assertion();                                                            19
 }
                                                                            21
 testcase myTestcase() runs on EngineTester {
   // reusable mode application                                            23
   stim_seq(1000.0, 10.0, assert_mode);
   stim_seq(5000.0, 1.0,  cont {                                            25
     assert(x_1.value >= 0.0) }
   );                                                                       27
 }
```

Listing 4.62 illustrates the application of parametrizable reusable modes. Lines 24 and 25 inside the test case definition show two references to a named and parameterized mode definition. The first one sets the parameter values for *val_1* to 1000.0 and the parameter for *val_2* to 10.0, and a mode called *assert_mode* is passed as a parameter to be applied within the *stim_seq* mode. The next line shows a more or less similar application of *stim_seq*; in this application, an inline mode declaration is passed as the mode parameter.

The behaviour of a named mode can be defined using the statements and operations described in Sections 4.4.2 to 4.4.7. If a mode uses variables, constants, timers and ports that are declared in the component type definition,

the component type is selected by means of the runs-on clause in the mode header. The one exception to this rule is: if all component-wide information is used within, the mode is passed in as a parameter. A mode without a runs-on clause shall never locally invoke functions, or modes with a runs-on clause.

Syntactical structure (concrete syntax) 27.

NamedMode	::=	*ModeKeyword ModeIdentifier* [*ModeTypeIdentifier*] "(" *ModePar* { "," *ModePar* } ")" [*RunsOnSpecOrSelf*] *ModeSpecification*
ModePar	::=	*FormalValuePar* | *FormalTimerPar* | *FormalTemplatePar* | *FormalPortPar* | *FormalModePar*
ModeTypeIdentifier	::=	*Identifier*
ModeIdentifier	::=	*Identifier*
ModeKeyword	::=	"`mode`"
ModeSpecification	::=	(*BasicMode* | *ComplexMode*) [*UntilBlock*]

Chapter 5

Operational integration with *TTCN-3*

The integration of new concepts into an existing language must be carefully planned, specified and documented. Many modern programming languages are specified in an informal way or via reference implementation; *TTCN-3* , in contrast, is documented formally with respect to both its syntactical structure and its operational semantics. The syntactical and semantic integration of the concepts specified in Section 3 with *TTCN-3* will, therefore, partially make use of these specifications.

While the syntactical structure of *TTCN-3* is specified in [35] via the Extended Backus Naur Form (EBNF) [71, 64], so-called flow graphs are used for the operational semantics [36]. EBNF is widely accepted as a formalism that denotes the syntax of textual language; therefore, it will be reused in this thesis for the purpose of describing the syntactical structures of the language extension, as well as for their integration with Standard *TTCN-3*. Flow graphs are not very popular for this purpose, nor do they provide a compact representation. Abstract state machines (ASMs) [51, 14] are, in contrast, widely accepted; they allow algorithms to be described in a clear, purely mathematical form. ASMs were originally introduced in order to meet the requirements of complexity theory. Their applicability for formal specification in multiple fields has been demonstrated in various publications [13].

The following sections describe the operational semantics of the *TTCN-3 embedded* concepts introduced in Section 4. The chapters also highlight how the new concepts integrate with the already-existing *TTCN-3* concepts documented in [35, 37, 36]. The general approach — that is to say, the ASM framework and the definition and redefinition of consecutive subsets of the language — is inspired by [15].

5.1 Abstract state machines

ASM specifications have already been successfully applied to the semantics definition for various programming languages. Examples include Prolog, C++ [127], and Java [107]. Furthermore, the complete dynamic semantics of the ITU standard SDL 2000 were described by means of ASMs [41].

An abstract state machine performs state transitions with algebras. Each algebra is represented by a state. An ASM program is defined by means of guarded function updates, so-called *rules*. These rules are nested conditional clauses of the form

if *Condition* **then** *Updates*
else *Updates*
end if

Each rule specifies a set of function updates, known as a "block". When a set of rules fires, a state transition is performed in an ASM with function updates $f(t_1, ..., t_r) := t_0$, where t_i are terms. Each individual function update of each block is executed simultaneously.

ASMs are multi-sorted and based on the notion of universes. The mathematical universes of Boolean, Integer, Real, etc., as well as the standard algebraic operations, are considered available. Furthermore, a set of high-level constructs eases the specification process by providing often-used, standardized functionalities. A universe can be dynamically extended with individual entities by

extend *Universe* **by** *v*
  Rule
end extend

where v is a variable that is bound by the **extend** constructor to the element that has been added to $Universe$. The **ranges over** constructor simultaneously defines the instantiation of a rule for each element in $Universe$.

var *v* **rangesover** *Universe*
  Rule
end var

The constructor basically spawns n rules so that v iterates overall values in $Universe$ and n is the number of elements in $Universe$.

Sequences of entities are also introduced. A sequence is denoted with $\langle e_2, e_1 \rangle \in Universe^*$. The concatenation of sequences is simply expressed by the row position of two single sequences. Thus $\langle e_3 \rangle \langle e_2, e_1 \rangle \mapsto \langle e_3, e_2, e_1 \rangle$. The length of a sequence is defined by $len(\langle f_n, f_{n-1}, .., f_1 \rangle) \mapsto n$. On this basis, the most common stack (lifo) functions are defined as follows,

$push : Universe \times Universe^* \rightarrow Universe^*,$
with $push(e_1, \langle\rangle) \mapsto \langle e_1 \rangle$
and $push(e_{n+1}, \langle e_n, e_{n-1}, .., e_1 \rangle) \mapsto \langle e_{n+1} \rangle \langle e_n, .., e_1 \rangle \mapsto \langle e_{n+1}, e_n, .., e_1 \rangle$
$top : Universe^* \rightarrow Universe,$
with $top(\langle e_n, e_{n-1}, .., e_1 \rangle) \mapsto e_n$
and $top(\langle\rangle) \mapsto undef$
$pop : Universe^* \rightarrow Universe^*,$
with $pop(\langle e_n, e_{n-1}, .., e_1 \rangle) \mapsto \langle e_{n-1}, .., e_1 \rangle$
and $pop(\langle\rangle) \mapsto \langle\rangle$

and the most common queue (fifo) functions are defined as follows

$enqueue : Universe^* \times Universe \rightarrow Universe^*,$
with $enqueue(e_1, \langle\rangle) \mapsto \langle e_1 \rangle$
with $enqueue(e_{n+1}, \langle e_1, .., e_n \rangle) \mapsto \langle e_{n+1} \rangle \langle e_1, .., e_n \rangle \mapsto \langle e_1, .., e_n, e_{n+1} \rangle.$
$dequeue : Universe^* \rightarrow Universe^*,$
with $dequeue(\langle e_1, e_2, .., e_n \rangle) \mapsto \langle e_2, .., e_n \rangle$
and $dequeue(\langle\rangle) \mapsto \langle\rangle$

5.2 Approach to operational semantics description

TTCN-3 is a procedural, statement-based testing language. The testing behaviour is defined by algorithms, and these algorithms interact with the environment by assigning messages to ports and evaluating incoming messages from ports. Without losing generality, we can consider a *TTCN-3* program to consist of definitions (i.e. $def \in DEF$), statements (i.e. $stmt \in STMT$) and expressions (i.e. $exp \in EXP$). Formally speaking, we define

$$TTCNProg \in \mathcal{P}(DEF \cup STMT \cup EXP)$$

The whole semantics is defined as a set of subsequent extensions; this denotes the operational semantics cleanly and clearly. This kind of step-by-step

approach provides iterative language refinements that begin from an imperative core. The imperative core defines the basic and, so far, unspecialized core of the language. This core has a lot in common with other widely-used imperative languages such as C, C# and Java.

The imperative core $TTCN_{\mathcal{I}}$ of *TTCN-3* describes the basic imperative language features, i.e., the statement execution, expression evaluation and control structures of *TTCN-3*. On the basis of that core, subsequent language refinements are specified. Each of these refinements introduces new language constructs and thus improves the original semantics.

- $TTCN_{\mathcal{C}}$ is based on $TTCN_{\mathcal{I}}$ and specifies *TTCN-3* with features like test case execution, components and function calls.
- $TTCN_{\mathcal{T}}$ is based on $TTCN_{\mathcal{C}}$ and specifies *TTCN-3* with features like sending and receiving messages, alt statements, altsteps and timers.
- $TTCN_{\Delta}$ is based on $TTCN_{\mathcal{T}}$ and specifies *TTCN-3 embedded* with features like global time and sampling.
- $TTCN_{\mathcal{E}}$ is based on $TTCN_{\mathcal{E}}$ and specifies *TTCN-3 embedded* with modes.

$TTCN_{\mathcal{I}}$, $TTCN_{\mathcal{C}}$ and especially $TTCN_{\mathcal{T}}$ cover the existing *TTCN-3* standard; $TTCN_{\Delta}$ and $TTCN_{\mathcal{E}}$, on the other hand, explicitly introduce the concepts of *TTCN-3 embedded*. Because the purpose of this thesis is to present *TTCN-3 embedded*, the semantics are stripped down to fully define $TTCN_{\Delta}$ and $TTCN_{\mathcal{E}}$. To provide an understandable basis for $TTCN_{\Delta}$ and $TTCN_{\mathcal{E}}$, $TTCN_{\mathcal{I}}$, $TTCN_{\mathcal{C}}$ and $TTCN_{\mathcal{T}}$ are presented partially (see Section 5.4).

The interaction between the *TTCN-3* abstract test execution environment (TE) and its environment is specified by means of standardized interfaces. The semantics address these standard interfaces, thereby restricting their interactions with the environment, in particular the Test Runtime Interface (TRI). The interactions with the Test Control Interface (TCI) as well as the Encoding/Decoding of messages are not considered.

Finally, the original *TTCN-3* operational semantics [36] make intensive use of so-called statement replacements. Assuming that a set of *TTCN-3* statements of a certain kind can be completely replaced by a combination of different *TTCN-3* statements, the original operational semantics focus on *TTCN-3* constructs which cannot be replaced. In a similar manner, the ASM specification only provides formalisms for non-replaceable *TTCN-3 embedded* statements.

5.3 The basic ASM framework

A run of a *TTCN-3* program generally consists of statement execution and expression evaluation. Expression evaluation is used to calculate values and statement execution either controls the program flow, initiates the calculation of values or defines the interaction with the environment. Each run of a *TTCN-3* program or *TTCN-3 embedded* program can be seen as a walk through an annotated Abstract Syntax Tree (AST). The grammar below exemplifies the overall set up of the AST and shows the statements and expressions used in $TTCN_{\mathcal{I}}$. The term $finished$ is an auxiliary construct and is used to denote the end of a program or of a statement block.

Abstract syntax 1: Extracts from the *TTCN-3* basic behavioural constructs.

EXP	::=	*LIT* | *VAR* | *CONST* | *UNOP* | *BOP* | *OpCall*
STMT	::=	*Assignment* | *LogStatement* | *LoopConstruct*
		| *ConditionalConstruct* | *GotoStatement*
		| *StatementBlock*
PHRASE	::=	*EXP* | *STMT* | "`finished`"

In the following, the abstract syntax is considered complete with respect to the behavioural entities (i.e. statements and expressions) of *TTCN-3* and *TTCN-3 embedded*. Definitions, such as type definitions, variable definitions, template definitions and function definitions, are considered to be known and to have already been checked for validity and structural correctness. The abstract test execution environment (TE) is aware of these definitions and it provides the means to discover and use the definitions whenever they are needed. Furthermore, the definitions are considered to be organized in optimal execution order. The TE, in theory, provides enough memory and a processor that is able to process a finite number of ordinary statements within a given, arbitrary, small timebound Δ.

We use the same identifiers within the ASM universes, the abstract syntax and the concrete syntax; this allows exchange between all three formalisms. Identifiers starting with a capital letter, those that may be written in camel-case notation (e.g. *Assignment*, *AltConstuct*, etc.), are directly connected to the concrete syntax. Identifiers that are written completely in capital letters (e.g. $PHRASE$, EXP, $STMT$, etc.) are used to sum up constructs that are not distinguished in regards to their behavioural semantics; these identifiers generally represent more abstract nodes in the AST. Identifiers in small letters represent instances of nodes or elements of AST universes — for example, $phrase \in PHRASE$.

Operational progress in *TTCN-3* is achieved by means of a program

counter that points to the individual nodes of an AST. Following the example of [15], we define the operational core of the *TTCN-3* language using such a program counter. The main required entities are representations for components and so-called agents. Components with $c \in Component$ describe the variety of *TTCN-3* components. The following definition of the operational semantics of *TTCN-3 embedded* uses distributed ASMs. These ASMs consist of a set of autonomously operating agents $a \in AGENT$. An agent is the abstract executer of a *TTCN-3* component. The 0-ary dynamic function $Self$ is interpreted by each agent a as a. If the context provides us with a clear interpretation of $Self$, we denotationally suppress $Self$.

Let $AGENT$ be the abstract set of agents a that move over the hierarchical syntax structure of a *TTCN-3* program. An agent is associated with exactly one *TTCN-3* component at any given time, and $current : AGENT \rightarrow Component$ is a dynamic function that indicates the component under execution. Because a component is considered the carrier of a *TTCN-3* program execution, each $c \in Component$ has a program pointer which designates the next $phrase \in PHRASE$ to be executed. A *phrase* defines the occurrence of a certain programming construct, and all constructs are subject to the *TTCN-3* ASM semantics. The abstract function *task* triggers an abstract program counter; the program counter allows the current *phrase* to execute until $finished$ is reached.

$$task : Component \rightarrow PHRASE$$

The program counter *task* will update according the *TTCN-3* control flow. The execution of expressions, subexpressions, statements and sub-statements is defined by the static functions fst and nxt.

$$fst, nxt : PHRASE \rightarrow PHRASE$$

The definition of each of these functions depends upon the respective element $phrase \in PHRASE$; the definition is provided recursively during the refinement of the language elements. The formal function *loc* describes the relation between the variables and their values:

$$loc \ : \ Component \rightarrow Value^T \qquad \text{with } T \in \mathbb{T}$$

The access to intermediate values – values not assigned to a variable but rather used during expression evaluation, for operand handling, for parameter

passing and for return-value passing, as well as for expressions in conditional statements – is formalized by the function *val*.

$$val \ : \ Component \rightarrow Value^{T} \qquad \text{with } T \in \mathbb{T}$$

The universe $Value^{T}$ contains all kinds of *TTCN-3* values of a certain type $T \in \mathbb{T} = \{integer, float, boolean, ...,\}$ so that $Value^{float}$ denotes all *TTCN-3* float values, $Value^{integer}$ denotes all *TTCN-3* integer values, and so on and so forth. For simplicity's sake, the *TTCN-3* Boolean values are associated with the ASM values *true* and $false$.

The macro *is* reflects the current execution status within the ASM semantic. It also controls the execution of ASM rules.

$task$ **is** $phrase \equiv task = phrase \ \wedge \ phrase \in PHRASE$

TTCN-3 and *TTCN-3 embedded* provide statements contained within other statements, in the form of statement blocks (e.g. alt statements or modes). For the purposes of control flow, it is necessary to be able to refer from such "contained" statements to their enclosing block statements during runtime. This system of referral allows the operand to manifest abrupt breaks and other exit behaviour. The function *up* provides the TE with this feature.

$$up : PHRASE \rightarrow PHRASE$$

The function *up* takes an arbitrary phrase as a parameter and always returns the enclosing phrase. This hierarchy, which is used to resolve the containment relationship, is defined in the abstract syntax tree.

5.4 Extracts from $TTCN_{\mathcal{I}}$, $TTCN_{\mathcal{C}}$, and $TTCN_{\mathcal{T}}$

This section provides a summary of the relevant signatures and rules that define the behaviour of $TTCN_{\mathcal{I}}$, $TTCN_{\mathcal{C}}$, and $TTCN_{\mathcal{T}}$.

5.4.1 Component signature

A *TTCN-3* component has ports, a status and a type. It provides value-of-type verdicts that indicate the test verdict. Formally, these properties are indicated using the following functions:

$$
\begin{aligned}
ports &: Component \rightarrow Port \\
status &: Component \rightarrow \{running, inactive, stopped, killed\} \\
type &: Component \rightarrow \{control, mtc, tc\} \\
alive &: Component \rightarrow \{true, false\} \\
verdict &: Component \rightarrow \{none, pass, inconclusive, error, fail\}
\end{aligned}
$$

The function *ports* returns the set of ports associated with an individual component. The function *status* returns the processing state of a component. The label *running* indicates that the component is active and processing a *TTCN-3* program; the label *inactive* indicates a pause of execution; the label *stopped* indicates that the component has finally stopped the execution; and the label *killed* indicates a component that has been killed. The function *type* returns either *control* (for a component that started to execute the control part of a module), *mtc* (for the main test component) or *tc* (for an arbitrary test component). The function *alive* returns *true* if the component is flagged as "alive"; otherwise, it returns $false$. And, finally, the function *verdict* returns the actual verdict value for a component.

5.4.2 Message signature

TTCN-3 supports message-based and procedure-based unicast, multicast and broadcast communication. Technically speaking, a message is a piece of information that has been prepared for transportation between two entities. A message may be encoded and enriched with transport-related information (sender, receiver, timing information, etc.). A *TTCN-3* message $m \in Message$ is a structure that encapsulates information about the transmitted value and its nature, and the sender and receiver of the message. In *TTCN-3*, both sender and receiver of messages are components. The ASM signatures below define the functions that address the individual elements of the message structure.

$$
\begin{aligned}
type &: Message \rightarrow T \\
sender &: Message \rightarrow Component \\
recipient &: Message \rightarrow Component \\
val &: Message \rightarrow Value^t \text{ with } t \ = \ type(m) \ \in T
\end{aligned}
$$

Furthermore, *TTCN-3* provides a mechanism for matching messages against a pattern expression. The pattern expression is called a template and it char-

acterizes messages according to their content and structure. Without losing generality, a template can be considered as an expression $exp \in EXP$. A function *map* allows to check whether or not a message matches a template.

$$match : Message \times EXP \rightarrow Value^{booelan}$$

The function returns "true" if the template matches the message; otherwise, it returns "false".

5.4.3 Port signature

TTCN-3 uses ports to communicate with the external environment (i.e. the SUT) and to communicate between *TTCN-3* components. Message ports are used for non-blocking (asynchronous), message-based communication; procedure ports, on the other hand, are used for classical client-server communication. The signatures for *TTCN-3* ports are defined below.

$$\begin{aligned} kind &: Port \rightarrow \{message, procedure\} \\ direction &: Port \rightarrow \{in, out, inout\} \\ msgs &: Port \rightarrow (Message)^* \\ types &: Port \rightarrow \mathcal{P}(T) \\ owner &: Port \rightarrow Component \\ status &: Port \rightarrow \{started,\ halted,\ stopped\} \end{aligned}$$

The function *kind* is used to refer to the different communication characteristics of a port. Furthermore, each port has a *direction* that specifies the communication direction, namely *in*, *out*, and *inout*. The function *msgs* obtains the port queues (i.e. the chronologically-sorted sequence of received messages) and the function *types* returns the data types that can be communicated. Finally, the function *owner* returns the component that holds a given port and the function *status* returns the execution status of port.

In *TTCN-3*, ports can be mapped or connected to other ports. This forms the base communication architecture of the test system setup and defines the general conditions for the flow of messages between components. *TTCN-3* allows one or more connections and it distinguishes port connections between test components and the system under tests from port connections between test components. The first kind of connection is called port "mapping"; it

can be dynamically defined by means of the map statement. The second kind of connection is called "connection"; it can be dynamically defined via the connect statement. These operational semantics do not explicitly define meaning for the statements that directly handle the mapping and connection of ports; rather, these operational semantics refer to the mapping and connection information. Thus, the function

$$mapsto : Port \rightarrow Port* \cup \{undef\}$$

returns the set of port that contains all ports mapped or connected to a particular port. When a port is not mapped or connected, the function returns $undef$.

5.4.4 Timer signature

A *TTCN-3* timer is a counter incremented according to current time progress. Such a timer can be configured with a timeout value. If a timer reaches its timeout value, the timer is considered to have timed out, i.e., it changes from the status *running* to the status *timedout*. In general, a *TTCN-3* timer has the status of *running*, *inactive*, or *expired*. The status of a timer is modelled using the function *status*. The status *running* corresponds to a running timer — that is to say, to a timer that counts. The status *inactive* corresponds to a stopped timer: to a timer that has not been started or that has been stopped using the function *stop*.

$$\begin{aligned} start &: Timer \times Value^{float} \rightarrow \\ stop &: Timer \rightarrow \\ status &: Timer \rightarrow \{running,\ inactive,\ expired\} \\ val &: Timer \rightarrow Value^{float} \\ timeoutval &: Timer \rightarrow Value^{float} \end{aligned}$$

The function *start* sets the timer's timeout value and starts the timer. The status of the timer is set to *running*. The function *stop* stops a timer. The respective status changes to *inactive*. The function *status* returns the current status of a timer, the function *val* returns the current timer value and the function *timoutval* the value, at which the timer will time out, or has timed out.

5.4.5 Snapshots

The assessment of ports in *TTCN-3* is organized according to so-called snapshot semantics. Snapshot semantics enforce that the relevant test system state is frozen when an assessment starts. This kind of snapshot guarantees a consistent view on the test system input during an individual assessment step. A snapshot covers all relevant stopped test components, all relevant timeout events and the top messages (as well as calls, replies and exceptions, in the case of procedure-based communication) in the relevant incoming port queues. To access the state of ports and timers of a snapshot, primed versions of the functions *value*, *status* and *msgs* are introduced.

$$\begin{aligned} val' &: Timer \rightarrow Value^{float} \\ status' &: Timer \rightarrow \{running,\ inactive,\ expired\} \\ msgs' &: Port \rightarrow Message^{*} \end{aligned}$$

The functions val', $status'$ and $msgs'$ can be used similarly to the non-primed version of the functions. The difference is that, where *val*, *status* and *msgs* return the actual values or messages, val', $status'$ and $msgs'$ return the state of the timers and queues at the last snapshot.

5.4.6 Statements and expression execution

The control flow in *TTCN-3* and *TTCN-3 embedded* is determined by statements. Each statement in *TTCN-3* occurs in a statement block, and a statement block itself is a statement. The behavioural semantics of an empty statement block are defined by $fst(statementBlock) = nxt(statementBlock)$. The dynamics of a statement block with one or more statements is defined as follows:

$$\begin{aligned} &\textbf{let } statementBlock = (stmt_1, stmt_2, ..., stmt_n) \textbf{ in} \\ &\quad fst(statementBlock) = fst(stmt_1) \\ &\quad nxt(stmt_i) = fst(stmt_{i+1}),\ 0 < i < n \\ &\quad nxt(stmt_n) = nxt(statementBlock) \end{aligned}$$

Expressions in *TTCN-3* are either literals, variables or constants, and operators can have one or more operands or function calls. An expression may be composed of sub-expressions. The order of expression evaluation is given by the recursive definition of the static functions fst and nxt. For expressions without any sub-expression, we define $fst(exp) = exp$. Any other expression model can be handled through the generalization of expressions with an arbitrary number of sub-expressions to $f_{exp}(exp_1, exp_2, ..., exp_n)$, where f_{exp} represents the n-ary expression constructor for the expression exp and $exp_1, exp_2, ..., exp_n$ the respective sub-expressions. In this case, the control flow is determined by the following definitions:

$$\begin{aligned}&\textbf{let } exp = (exp_1, exp_2, .., exp_n) \textbf{ in}\\ &\quad fst(exp) = fst(exp_1)\\ &\quad nxt(exp_i) = fst(exp_{i+1}), 0 < i < n\\ &\quad nxt(exp_n) = exp\end{aligned}$$

In the simple case of an unconditioned execution of atomic statements and expressions, *TTCN-3* control flow is defined with the following ASM macro.

$$proceed \equiv task := nxt(task)$$

5.4.7 Further refinements to support $TTCN_{\mathcal{C}}$ and $TTCN_{\mathcal{T}}$

In contrast to $TTCN_{\mathcal{I}}$, $TTCN_{\mathcal{C}}$ and $TTCN_{\mathcal{T}}$ support parallel test components and user-defined functions with recursions. To support these and other features in $TTCN_{\mathcal{C}}$ and $TTCN_{\mathcal{T}}$, the basic macros and functions that have been defined in Section 5.3 need to be redefined.

Parallel test components are supported by multiple ASM agents. As has already been defined, an agent $a \in AGENT$ is associated with exactly one *TTCN-3* component at any given time. The function $Self$ is interpreted by each agent a as a. If the context provides a clear interpretation of $Self$, $Self$ is denotationally suppressed. Each $c \in Component$ has an agent a and a program pointer, which signals the the next $phrase \in PHRASE$ to be executed.

The program pointer and the functions *loc* and *val* must have a stack-alike functionality, so that they can support nested scope units, due to nested and potentially recursive function calls or altstep calls. To this end, the signatures

of these functions are refined such that they provide sequences of dynamic functions as return values.

$$\begin{aligned} task_{\mathcal{C}} &: Component \rightarrow PHRASE^* \\ loc_{\mathcal{C}} &: Component \rightarrow (VAR \rightarrow V)^* \\ val_{\mathcal{C}} &: Component \rightarrow (EXP \rightarrow V)^* \end{aligned}$$

For simplicity's sake, the term $frames$ is used to bundle together the functions needed for the recursive calling. Thus, for every $c \in Component$ let

$$frames(c) \equiv (task_{\mathcal{C}}(c), loc_{\mathcal{C}}(c), val_{\mathcal{C}}(c))$$

In the case of a *TTCN-3* function call, the sequences in $frames(c)$ are enlarged with further dynamic functions, now representing either the program counter, the local environment, or the temporary values of the function's scope. When a *TTCN-3* function is left, the original state — that is to say, the state before the function was called — can be restored by removing the recently-added dynamic functions from the list.

$$\begin{aligned} task &\equiv top(task_{\mathcal{C}}(Self)) \\ loc &\equiv top(loc_{\mathcal{C}}(Self)) \\ val &\equiv top(val_{\mathcal{C}}(Self)) \end{aligned}$$

$TTCN_{\mathcal{I}}$ and the basic framework do not provide support for nested scopes and recursions. As shown above, however, the functions $task$, loc and val defined in the previous sections can be mapped to the functions $task_{\mathcal{C}}(Self)$, $loc_{\mathcal{C}}(Self)$ and $val_{\mathcal{C}}(Self)$ in $TTCN_{\mathcal{C}}$ by associating $Self$ and by taking the topmost elements. This ensures that the rules from $TTCN_{\mathcal{I}}$ can be transferred directly to $TTCN_{\mathcal{I}}$. For reasons of simplicity, $Self$ is notationally suppressed by writing $task_{\mathcal{C}}$, $loc_{\mathcal{C}}$ and $val_{\mathcal{C}}$ instead of $task_{\mathcal{C}}(Self)$, $loc_{\mathcal{C}}(Self)$ and $val_{\mathcal{C}}(Self)$. Finally, the terms $task$, loc and val are used like they are used in $TTCN_{\mathcal{I}}$, except when the stack functionality must explicitly be mentioned.

Introducing parallel components implicitly requires the management of the status of components; therefore, the macro *is* is refined.

$task$ **is** $phrase \equiv$
$task = phrase \wedge phrase \in PHRASE \wedge status \notin \{inactive, stopped, killed\}$

Unless stated otherwise, rules are executed only if the execution status is not *inactive*, *stopped*, or *killed*.

5.5 $TTCN_{\Delta}$: *TTCN-3* WITH STREAMS, GLOBAL TIME AND SAMPLING

$TTCN_{\Delta}$ describes the operational semantics of *TTCN-3* with streams, global time and sampling. Whereas the previous specifications describe the behaviour of standard *TTCN-3*, this section introduces new concepts that have been sketched in 3 and structurally defined in Section 4.1, Section 4.2, and Section 4.3.

5.5.1 Refinement of statements and expression universes

$TTCN_{\Delta}$ includes statements and expressions that can be used to control timing issues and to handle messages in stream ports. These statements and expressions are introduced in Section 4.1 and Section 4.2. To cover these statements within the ASM context, the universes EXP and $STMT$ must be extended in the following way.

Abstract syntax 2: $TTCN_{\Delta}$ basic behavioural constructs.

EXP_{Δ}	::=	$EXP_{\mathcal{C}}$ | *StreamDataOps* | *StreamNavigationOps* | *StreamEvalOps*
$STMT_{\Delta}$	::=	$STMT_{\mathcal{C}}$ | *WaitStatement*
StreamDataOps	::=	*StreamValueOp* | *StreamTimestampOp* | *StreamDeltaOp*
StreamNavigationOps	::=	*StreamPrevOp* | *StreamAtOp* ["`Dot`" *StreamDataOp*]
StreamEvalOps	::=	*StreamHistoryOp*

5.5.2 $TTCN_{\Delta}$ time and sampling signatures

In standard *TTCN-3*, time progress itself has only implicit semantics. It is possible to measure time progress with the timer operations and thus change the control flow of a *TTCN-3* program as a function of time. In $TTCN_{\Delta}$, time and time progress become a more central concept. $TTCN_{\Delta}$ is a synchronous language. That means that the statement execution and especially the interaction through ports are synchronized on the basis of a system-wide valid sampling rate (see sampling in Sections 3.3 and 3.3.2).

Time is referred to by the functions $time$, t_{sample}, t'_{sample}.

$$t :\rightarrow Value^{float}$$
$$t_{sample} :\rightarrow Value^{float}$$
$$t'_{sample} :\rightarrow Value^{float}$$

The function t provides access to the exact time that has passed since the beginning of a test case and t_{sample} represents the time point of the next sample step. The function is initialised at the beginning of a test case with $t_{sample} = t + \Delta$ and updated for every new sample step. Its former value is preserved by t'_{sample}. The function Δ represents the minimum base sampling rate of the test execution environment. It is set at the beginning of the test execution and remains fixed during the complete test execution.

While t is considered to be updated automatically, the other functions represent time with respect to sampling. They are updated by a so-called test execution controller. The test execution controller is precisely specified in one of the next sections.

5.5.3 $TTCN_{\Delta}$ stream port signature

Stream ports differs from ordinary *TTCN-3* message ports and procedure ports in that they provide a sampled access to the actual value and (theoretically, at least) also provide access to the complete history of a continuous signal. The underlying data structure, however, is quite similar to the data structure defined for *TTCN-3* message and procedure ports. Most of the functions that have been defined for message and procedure ports (see Section 5.4.3) are thus adapted for stream ports.

$$type_{\Delta} : Port_{\Delta} \rightarrow \mathcal{P}(T)$$
$$msgs_{\Delta} : Port_{\Delta} \rightarrow Message^{*}$$
$$msgs'_{\Delta} : Port_{\Delta} \rightarrow Message^{*}$$
$$direction_{\Delta} : Port_{\Delta} \rightarrow \{in, out, inout\}$$

with

$$Port_{\Delta} = Port_{\mathcal{T}} \vee Port.$$

There are, however, some slight differences. A stream port has a distinct data type; hence, the function *type* returns a set with one element if it is applied to a stream port. Furthermore, the functions $msgs$ and $msgs'$ return a sequence of messages — for incoming as well as for outgoing stream ports. Last but not least, a newly-introduced function *delta* enables access to the currently-used sampling time of that stream port.

$$delta_{\Delta} : Port^{stream} \rightarrow Value^{float}$$

Finally, the function *kind*, which helps to distinguish between the different kinds of ports, must also be refined in order to address stream ports.

$$kind_{\Delta} : Port_{\Delta} \rightarrow \{message, procedure, stream\}$$

The main differences between stream ports and message or procedure ports are twofold. The first difference regards the sampling: for each sample step, there is a message that is stored in the port's message sequence. The second difference regards the ability to randomly access all messages that have been sent or received at a port. The latter feature is realized by introducing the two functions [] and @ .

$$\begin{aligned} &[] : Message^{*} \times V \rightarrow Message, \text{ with} \\ &\langle m_n, ..., m_2, m_1 \rangle [i] \mapsto m_{n-i} \end{aligned}$$

$$\begin{aligned} &@ : Message^{*} \times V \rightarrow Message, \text{ with} \\ &\langle m_n, .., m_2, m_1 \rangle @t \mapsto m_x, \text{ with} \\ &t \in [0..t_{sample}] \wedge x \in [0..n], \text{ where} \\ &\forall i \in [0..n], \ (ts(m_i) - t) \ < \ (ts(m_x) - t) \rightarrow ts(m_i) > t \end{aligned}$$

For a more detailed explanation of these functions, please refer to Section 3.3.1.

5.5.4 Further $TTCN_\Delta$ signatures

Components in $TTCN_\Delta$ have quite similar signatures to those of the preceding definitions. The only change is that function *ports*, which returns all ports that are available at a component (see Section 5.4.3), is refined to return not only message and procedure ports, but to additionally include stream ports.

$$ports_\Delta : Component_\Delta \to Port_\Delta$$

Furthermore, in $TTCN_\Delta$, there is another status called *wait*. This status is used to suspend the actions of a component, causing it to wait for the next sampling step. It is usually used to suspend a component after the successful completion of all necessary tasks in a particular sampling step.

$$status_\Delta : Component_\Delta \to \{running,\ inactive,\ stopped,\ killed\}$$

Finally, messages have timestamps. These timestamps are stored with the message and they contain both the arrival and sending times of the individual message. The timestamp can be accessed by means of the *ts* function.

$$ts_\Delta : Message_\Delta \to Value^{float}$$

5.5.5 $TTCN_\Delta$ sampling controller

The sampling controller is part of the TE structure. It is responsible for providing the overall sampling and controlling the timing of the test execution. The sampling controller controls time progress, calculates the next sample time, schedules the update of the input and output ports and triggers the resumption of execution for components that have been suspended via the status *wait*.

The functions t, t_{sample}, t'_{sample} and Δ are dynamically updated. t is considered to receive automated updates from $clck_0$; the updates of the functions t_{sample} and t'_{sample}, however, are defined below.

Rule 1: ASM transition rule for the Sampling Controller.

if $t \geq t_{sample}$ **then** ▷ (sampling controller)
  $t'_{sample} = t_{sample}$

$t_{sample} = t_{sample} + \Delta$
for all $c \in Component$ **do**
$updateStreams(c)$
$takeSnapshot(c)$
$status(c) := running$
end for
end if

The function t_{sample} represents the time point of the next sample step. The function is initialised at the beginning of a test case with $t_{sample} = t + \Delta$ and is updated with each new sample step. Its former value is preserved by t'_{sample}. The function Δ represents the minimum base sampling rate of the test execution environment. It is set at the beginning of test execution and remains fixed throughout the complete test execution.

The macro *takeSnaphot* describes the actions undertaken to provide a snapshot for *TTCN-3* timers, as well as those used for message and procedure ports. A snapshot is local to a component; it stores the actual status of each port available at the component, for each timer declared in the component.

$takeSnapshot(c) \equiv$
for all $p \in ports(c)$ **do**
if $status(p) = started$ **then**
$msgs'(p) := msgs(p)$
end if
end for
for all $t \in timer(c)$ **do**
if $status(t) = running$ **then**
$status'(t) := status(t)$
$val'(t) := val(t)$
end if
end for

The macro *updateStreams* iterates over components and ports and is responsible for updating the values at ports. When the individual duration of the port's sampling step has expired, the port's message lists and the external environment are updated with the port's current values. Please note that the term $val(p)$ is the value currently associated to a port by means of an assignment in a *TTCN-3* program or, coming from the SUT, via *triGetGetStreamValue*.

$updateStreams(c) \equiv$
  for all $p \in ports(c)$ **do**
    if $status(p) = started \wedge kind(p) = stream \wedge stepExpired(p)$ **then**
      extend M **by** m
        $value(m) := val(p)$
        $ts(m) := t'_{sample}$
        $push(m, msgs(p))$
        $delta(p) := ts(msgs(port)[0]) - ts(msgs(port)[1])$
      end extend
    end if
  end for

A port's sample step has expired when the actual step time is greater than or equal to the step time of the port.

$stepExpired(p) \equiv t_{sample} \geq ts(p) + delta(p)$

5.5.6 $TTCN_\Delta$ time control

This section defines both the operational semantics for the wait statement and the now expression. Both constructs have been informally specified in Section 4.1.

$TTCN_\Delta$ wait statement

The wait statement suspends the execution of a component until a given time.

Abstract syntax 3: The wait statement.

WaitStatement ::= "`wait`" *EXP*

---

let $wait = (\texttt{wait},\ exp)$ **in**
  $fst(waitStatement) = fst(exp)$
  $nxt(exp) = waitStatement$

The first step of the execution of the wait statement occurs, when the TE evaluates the wait expression and calculates how long the wait statement

should hold. Once its endpoint has been determined, the status of the component is set to *wait*. The TE repeats this rule for each sampling step, as long as the end of the waiting period is reached. Then, the status of the component is changed back to the status *running* and the TE executes the next statement.

Rule 2: ASM transition rule for the wait statement.

```
if task is waitStatement then                          ▷ (wait statement)
    if val(exp) < t'_sample ∧ status(Self) = running then
        verdict(Self) := error
        proceed
    end if
    if val(exp) > t'_sample then
        status(Self) := wait
    else
        status(Self) := running
        proceed
    end if
end if
```

If the end of the waiting period has already been reached before the wait statement is executed, the error verdict is set.

$TTCN_\Delta$ now operation

The now operation returns the start time of the current sampling step.

Abstract syntax 4: The now operation.

nowOperation ::= "`now`"

Rule 3: ASM transition rule for the now operation.

```
if task is nowOperation then                           ▷ (now operation)
    val(nowOperation) := t'_sample
    proceed
end if
```

5.5.7 $TTCN_{\Delta}$ stream-data operations

The main purpose of stream-data operations is to read and modify the values and the properties of a stream port.

Abstract syntax 5: Stream-data operations.

StreamDataOp	::=	*StreamValueOp* | *StreamTimestampOp* | *StreamDeltaOp*
StreamValueOp	::=	"`value`" *VAR*
StreamTimestampOp	::=	"`timestamp`" *VAR*
StreamDeltaOp	::=	"`delta`" *VAR*

let $streamValueOp = (\texttt{value},\ var)$ **in**
$fst(streamValueOp) = streamValueOp$

let $streamDeltaOp = (\texttt{delta},\ var)$ **in**
$fst(streamDeltaOp) = streamDeltaOp$

let $streamValueOp = (\texttt{timstamp},\ var)$ **in**
$fst(streamTimestampOp) = streamTimestampOp$

The execution of the stream-value operation begins with a lookup of the port that is being referenced by the port variable. Afterwards, the *val* function of the stream-value operation is set to the message value of the referenced port (respecting sampling). Thus is the returned message value measured to the beginning of the actual sampling step.

Rule 4: ASM transition rule for the stream-value operation.

if *task* **is** *streamValueOperation* **then** ▷ (stream-value operation)
  $val(streamValueOperation) := value(top(msgs'(port)))$
  where $port = val(var)$
end if

The execution of the stream timestamp and the stream delta operation is quite similar. Both start with the lookup of the port. When the port is retrieved, the *val* function is set according to the kind of operation. In the case of the value operation, it is set to the timestamp of the current sample

step's message,

Rule 5: ASM transition rule for the timestamp operation.

if *task* **is** *streamTimestampOperation* **then** ▷ (timestamp operation)
  $val(streamTimestampOperation) := ts(top(msgs'(port)))$
  where $port = val(var)$
end if

and, in the case of the delta operation, it is set to the size of the actual sampling step.

Rule 6: ASM transition rule for the delta operation.

if *task* **is** *streamDeltaOperation* **then** ▷ (delta operation)
  $val(streamDeltaOperation) := delta(port)$
  where $port = val(var)$
end if

The value operation and delta operation, additionally, are allowed to appear on the left side of an assignment.

Abstract syntax 6: Assignment.

Assignment ::= "`assign`" *LeftSide EXP*
LeftSide ::= *VAR* | *StreamValueOperation* | *StreamDeltaOperation*

In this case, they are treated similarly to the way in which ordinary variables are treated. The expression values of the right side of the assignment are assigned to the properties of the stream port (which are characterized by the stream-data operation). In the case of the value operation, the value for the message that is to be sent at the next sample step is set. In the case of the delta operation, the step size of the sampling can be changed for the port referenced by the stream-data operation.

Rule 7: ASM transition rule for assignments (refines Rule 31).

if *task* **is** *assignment* **then** ▷ (assignment)
  if $leftSide$ **is** *var* **then**
    $loc(var) := val(exp)$
  end if
  if $leftSide$ **is** $streamValueOp$ **then**
    $val(port) := val(exp)$

```
        where  port = val(var)
    end if
    if leftSide is streamDeltaOp then
        delta(port) := val(exp)
        where  port = val(var)
    end if
    proceed
end if
```

5.5.8 $TTCN_{\Delta}$ *stream-navigation operations*

Stream-navigation operations are used to organize the random access to stream port values and properties. They must, therefore, necessarily be combined with a stream-data operation.

Abstract syntax 7: Stream-navigation operations.

StreamNavigationOp	::=	(*StreamPrevOp* | *StreamAtOp*) *StreamDataOp*
StreamPrevOp	::=	"`prev`" *VAR*
StreamAtOp	::=	"`at`" *VAR*

Because the use of the stream-navigation operation has an impact on the meaning of the associated stream-data operation, the semantics of both need to be defined in conjunction.

let $prevOp = (\texttt{prev},\ var,\ exp,\ streamDataOp)$ **in**
$fst(prevOp) = fst(exp)$
$nxt(exp) = prevOp$

The execution of a prev operation starts with the evaluation of the index expression. If the index expression outnumbers the messages available at the port, it is set automatically to the oldest value in the message list.

Rule 8: ASM transition rule for the prev operation.

```
if task is prevOp then                                  ▷ (prev operation)
    if val(exp) ≥ |msgs(p)| then
        val(exp) := |msgs(p)|
```

end if
if *streamDataOp* **is** *valueOp* **then**
$val(prevOp) := (msgs(p))[val(exp)]$
else if *streamDataOp* **is** *timestampOp* **then**
$val(prevOp) := ts((msgs(p))[|(msgs(p))| - val(exp)]))$
else if *streamDataOp* **is** *deltaOp* **then**
$val(prevOp) :=$
$ts((msgs(p))[|(msgs(p))| - val(exp)]))-$
$ts((msgs(p))[|(msgs(p))| - val(exp)] + 1))$
end if
end if
where $p := val(var)$

In case the prev operation is associated with the value operation, it retrieves the message with the index value $val(exp)$ and sets the *val* function for the complete construct to the value of the retrieved message. In the case that the prev operation is associated with the timestamp operation, it does the same with the timestamp of that message. In the case of an association with a delta operation, it calculates the respective distance between the message referenced by the index and its predecessor.

let $atOp = (\texttt{at},\ var,\ exp,\ streamDataOps)$ **in**
$fst(atOp) = fst(exp)$
$nxt(exp) = atOp$
$nxt(atOp) = streamDataOps$

The execution of the at operation also starts with the evaluation of the index expression. The main difference lies in the fact that the index expression addresses the timestamp value of a message in the message list of a port. If the index expression outnumbers the timestamp of the available messages, i.e. the index expression is larger than the time of the actual sample step, or smaller than the timestamp of the first message available, an error verdict occurs. In either case, the message with the timestamp nearest to value of the index value will be used for further processing.

Rule 9: ASM transition rule for the at operation.

---

if *task* **is** *atOp* **then** ▷ (at operation)
if $val(exp) > t'_{sample}$ **then**
$verdict(Self) := error$

$val(exp) := ts(hd(p))$
end if
if $val(exp) < ts((msgs(p))[|(msgs(port))|])$ **then**
$verdict(Self) := error$
$val(exp) := ts((msgs(port))[|(msgs(port))|])$
end if
if *streamDataOp* **is** *valueOp* **then**
$val(atOp) := (msgs(p))@(val(exp))$
else if *streamDataOp* **is** *timestampOp* **then**
$val(atOp) := ts((msgs(p))@(val(exp)))$
else if *streamDataOp* **is** *deltaOp* **then**
$val(atOp) :=$
$ts(msgs(port)@(val(exp)))-$
$ts(pred(msgs(port)@(val(exp))))$
end if
end if
where $p = val(var)$

In any case, the *val* function of the at operation is set to return the values and properties of the stream content, according to the associated stream value operation. The main difference to the prev operation lies in the retrieval of the messages subject to evaluation. The *TTCN-3*$_{\Delta}$ at operation thus makes intensive use of the function @ as defined in section 3.3.1 and refined in section 5.5.3.

5.5.9 $TTCN_{\Delta}$ history operation

The history operation is used to retrieve and evaluate a larger set of messages at a stream.

Abstract syntax 8: History operation.

StreamHistoryOp ::= "`history`" *PortVar Exp Exp*

The history operation returns a list of a stream's messages. The messages on this list are characterized by their upper and lower temporal bound. The

execution of the history operation starts with the calculation of the bounds.

$$
\begin{aligned}
&\textbf{let } streamHistoryOp = (\texttt{history},\ var,\ exp_1,\ exp_2) \textbf{ in}\\
&\quad fst(streamHistoryOp) = fst(exp_1)\\
&\quad nxt(exp_n) = fst(exp_n + 1)\\
&\quad nxt(exp_2) = streamHistoryOp
\end{aligned}
$$

If the lower bound exceeds the timestamp of the first message available at a port, or if the upper bound exceeds the time of the current sample step, the index values are adjusted so that they point to the most suitable of the messages available. In both cases, an error verdict is set. The error verdict happens as well when the lower bound exceeds the upper bound. In this case, the history operation returns an empty list of messages.

Rule 10: ASM transition rule for the history operation.

---

if $task$ **is** $streamHistoryOp$ **then** ▷ (history operation)
  if $val(exp_1) > t'_{sample}$ **then**
    $verdict(Self) := error$
    $val(exp_1) := ts(current(port))$
  end if
  if $val(exp_2) < ts(msgs'(port_2)[len(msgs'(port))])$ **then**
    $verdict(Self) := error$
    $val(exp_2) := ts(msgs'(port)[len(msgs'(port))])$
  end if
  if $val(exp_1) > val(exp_2)$ **then**
    $verdict(Self) := error$
    $val(streamHistoryOp) := \langle\rangle$
  else
    $val(streamHistoryOp) := \langle m_x..m_y\rangle, textwith$
      $m_x = msgs'(port)@val(exp_1)\ \wedge m_y = msgs'(port)@val(exp_2)$
  end if
end if
where $port := lookup(val(var))$

In any other case, the history operation returns the list of messages characterized by the upper and lower bounds. The messages, however, are returned in a special data format; this format is explained in detail in Section 4.2.7.

5.6 $TTCN_{\mathcal{E}}$: *TTCN-3* WITH MODES

$TTCN_{\mathcal{E}}$ contains all of the features of *TTCN-3 embedded*. It provides support for the simultaneous updating and evaluation of stream values at ports. The modes — hybrid automata alike control flow constructs — allow $TTCN_{\mathcal{E}}$ to define multiple execution paths via defining different modes of execution (i.e. different modes of signal generation and assessment). $TTCN_{\mathcal{E}}$ can also define transitions between these modes using predicates on values at ports.

The transition between $TTCN_{\Delta}$ and $TTCN_{\mathcal{E}}$ is accomplished by simply extending $TTCN_{\Delta}$ with modes. This requires only minimal changes in the signatures defined for n $TTCN_{\Delta}$. Firstly, the sets for statements and expressions need to be extended to catch up with the constructs that have been introduced for mode definition and control. Equally, there is also a need to explicitly handle the state of the modes during runtime.

5.6.1 $TTCN_{\mathcal{E}}$ statements and expressions

The extension of the sets EXP and $STMT$ is relatively straightforward. For expressions, the duration expression is introduced (see Section 4.4.4) and, for statements, the definitions for each kind of mode (i.e. atomic mode, parallel mode, sequential mode) are added.

Abstract syntax 9: $TTCN_{\mathcal{E}}$ basic behavioural constructs.

$$
\begin{array}{rcl}
EXP_{\Delta} & ::= & EXP_{\mathcal{E}} \mid duration \\
STMT_{\Delta} & ::= & STMT_{\mathcal{E}} \mid Mode \\
Mode & ::= & ContMode \mid ParMode \mid SeqMode
\end{array}
$$

5.6.2 $TTCN_{\mathcal{E}}$ mode signature

On a conceptual level, modes are categorized as atomic, parallel or sequential (see Section 4.4.1). During runtime, this is reflected by the introduction of a set called $Mode$; this set contains all kind of modes, and also contains individual subsets that contain specific kinds of modes.

$$Mode = AtomicMode \cup ParMode \cup SeqMode$$

Regardless of the type of mode, the TE provides a structure called $runtime \in Runtime$. This structure stores runtime information for each mode currently being executed. The information is maintained for each component

separately. This allows the execution of functions and the testing of different components that potentially refer to the same mode definitions. There is, therefore, a set $Runtime_c \subset Runtime$ for each individual component $c \in Component$.

The *runtime* structure provides a set of properties that allows a fine-grained control of the mode's execution during sampling. The properties are made available via the functions defined below.

$$\begin{aligned} status &: Runtime_c \rightarrow \{init, running, stopped\} \\ subtask &: Runtime_c \rightarrow \{invariants, guards, body, finalize\} \\ mode &: Runtime_c \rightarrow Mode \\ stime &: Runtime_c \rightarrow Value^{float} \\ trans &: Runtime_c \rightarrow PHRASE \end{aligned}$$

The function *status* returns the current runtime status of a mode. It can either yield *init* (when the mode is initialized), *running* (when the mode executes its contents) or *stopped* (when the execution has ended). For each mode there is an additional execution subtask; the subtask can be obtained by using the function $subTask$. The subtask of a mode defines which parts of a mode are to be executed executed. It can vary between the values *invariants*, *guards*, *body*, and *wait*; the subtask, therefore, directly relates to the overall structure of a mode. The function *mode* provides a link to the specific AST node, which, in turn, provides the definition of the mode for which the status is being maintained. When only the AST node is known, the *mode* function can be used to look up the mode state. Last but not least, the runtime structure contains the point in time where the associated mode was first initialized. The time value is provided by the function *time*. Finally, the function *trans* specifies the phrase to be executed when a mode is finished. It normally points to $nxt(mode)$ but this can change if transition statements are being used.

The function *runtime* is used to obtain the runtime information $r \in Runtime$ for a concrete AST node that represents a mode. The function uses the *mode* function in order to decide if the correct runtime structure has been found.

$$\begin{aligned} runtime : Component \times Mode \rightarrow Runtime_c, \\ \text{with } runtime(c, mode) \mapsto ms \\ \text{and } ms.mode \ = \ mode \wedge ms \in Runtime_c \end{aligned}$$

The function is parametrized with a component and a mode and it returns the correspondent runtime structure. If no such runtime structure exists, the function returns $undef$.

The execution of modes differs from the execution of ordinary statements and expressions because it provides an implicit sampling loop. The content of a mode is executed once at each sample step. In order to describe this kind of control flow, the function

$$loop : PHRASE \rightarrow PHRASE$$

is introduced in $TTCN_{\mathcal{E}}$. It complements the already-existing functions $next$ and fst. It also introduces a third control flow option, which is used if a mode is in a sampling loop. This function is used to indicate the *phrase* that must be executed if a mode repeats its content. It is set as an aspect of preparation for the next sample step. If not defined otherwise, the function points to the phrase given as its parameter. The trick, however, is that the function can be changed to reflect special cases.

$$loop(phrase) := phrase$$

Last but not least, a reactivation flag is needed in order to indicate the reactivation to already-stopped modes. This flag enables transitions between modes that have already been executed and must be reactivated. The reactivation flag is located at the component level.

$$reactivateNext : Component \rightarrow \mathbb{B}$$

5.6.3 $TTCN_{\mathcal{E}}$ modes

A mode is a block construct that consists of different substructures. Each of the substructures can be executed separately. The structure starts with the on entry block, which is followed by an invariant block, the mode's body, the on exit block and, finally, the until block (please also refer to Section 4.4.1

for a precise definition of the static structure of a mode).

Abstract syntax 10: Modes.

Mode	::=	("`cont`" | "`seq`" | "`par`") *OnEntry InvariantList Body OnExit Until*
Body	::=	*ContBody* | *SeqBody* | *ParBody*
InvariantList	::=	{ *EXP* }
Until	::=	"`until`" { *UntilGuardStatement* }

Mode substructures show similarities with both compound expressions and ordinary statement blocks. They generally contain expressions or statements and their content is executed subsequently. In the context of sampling, however, modes show a specific behaviour defined in the following sections.

Most of the substructure of a mode is, in fact, optional (see grammar in Section 4.4.1). In order to facilitate easier specification of their operational semantics, the optional blocks appear in the AST as non-optional but potentially empty. If such a block is missing in the original specification, the parser is expected to create an empty block (i.e. a statement or expression list with no content). Parts of a mode's overall execution order can, therefore, be statically defined by the following functions.

$$
\begin{aligned}
&\textbf{let } Mode = (onEntry, invariantList, body, onExit, until) \textbf{ in}\\
&\quad fst(mode) = fst(invariantList)\\
&\quad nxt(invariantList) = mode\\
&\quad nxt(onEntry) = mode\\
&\quad nxt(until) = fst(onExit)\\
&\quad nxt(onExit) = nxt(mode)\\
&\quad nxt(body) = loop(mode)
\end{aligned}
$$

The execution of a mode starts with the calculation of the invariants in the invariant list. After the calculation, the substructure of the mode is executed, depending on the current runtime status of a mode. In this phase, the invariants and guards are normally checked and the contents of the mode's body are subsequently executed. It is important to note that the different bodies of the different kind of modes (atomic, parallel and sequential) are executed in different ways. The precise semantics for the execution of each body are given below by specific ASM rules. When the execution of the mode's body is finished, the next task to be executed is determined by $loop(mode)$. This function points to the next mode or statement that is to be scheduled

within the current sampling step. It is set according to the current execution context and either loops back to the mode itself or points to $nxt(mode)$.

The execution of a mode ends when one of the guards in the until block is triggered. In this situation, the task switches to $nxt(until)$ and the on exit block is executed. Afterwards, the TE proceeds with the phrase that directly follows the mode.

As already mentioned, the static order is by any means incomplete and needs to be extended with updates of the dynamic functions in the rules below. This is especially necessary because the substructure of a mode is executed in different ways, dependent on the runtime status (see function *status* for mode runtime states) and the type of the mode.

The invariant list

Abstract syntax 11: Invariant lists.

InvariantList ::= { *EXP* }

The $invariantList \in InvariantList$ is handled similarly to a compound expression.

$$\begin{aligned}
&\textbf{let } invariantList = (\texttt{inv}, exp_1, exp_2, ..., exp_n) \textbf{ in}\\
&\quad fst(invariantList) = fst(exp_1)\\
&\quad nxt(exp_i) = fst(exp_i + 1), 0 < i < n\\
&\quad nxt(exp_n) = nxt(invariantList)
\end{aligned}$$

$$\begin{aligned}
&\textbf{let } invariantList = (\texttt{inv}) \textbf{ in}\\
&\quad fst(invariantList) = nxt(invariantList)
\end{aligned}$$

All invariant expressions in the invariant list are calculated one after the other. If the invariant list is empty, the list is simply skipped and the TE directly executes the statement that follows the invariant block. Finally, the invariant list is evaluated as a coherent expression, and it evaluates to

$$val(invariantList) \mapsto \begin{cases} exp_1 \wedge exp_2 \wedge ... \wedge exp_n & \text{when } n > 0\\ false & \text{otherwise} \end{cases}$$

.

Initialization of modes

If a mode structure is executed for the first time, there is no preexisting runtime element that can be retrieved (from the current component) for that specific mode.

Rule 11: ASM transition rule for mode execution (entering a mode).

if $task$ **is** $mode$ ▷ (mode execution ,initialization)
  $\wedge\ runtime(Self, mode) = undef$ **then**
  extend $Runtimes_{Self}$ **by** $runtime$
    $status(runtime) := init$
  end extend
end if

During this phase (i.e. during a mode's first appearance), a runtime element is created and added to the component's set of runtime elements. The function $runtime$ thus returns this value (i.e. $runtime(self, mode) \neq undef$) from this point on. After the mode's runtime element is created, the mode is initialised. During initialisation, the invariants are checked. If the invariants are violated, the execution of the mode is skipped and the error verdict is set. If the invariants match, the basic runtime information is set. The execution of the on entry block and the mode's body is then triggered and the on entry block is set as the next task to execute. The mode's runtime status changes to run.

Rule 12: ASM transition rule for mode execution (first loop).

if $task$ **is** $mode$ ▷ (mode execution, first loop)
  $\wedge\ status(runtime(self, mode)) = init$ **then**
  $reactivateNext(Self) := false$
  if $isViolated(invariantList)$ **then**
    $verdict(Self) := error$
    $status(modeState(Self, mode)) = stopped$
    $task := nxt(mode)$
  else
    $subtask(runtime(self, mode)) := body$
    $mode(runtime(self, mode)) := mode$
    $starttime(runtime(self, mode)) := t'_{sample}$
    $trans(runtime(self, mode)) := nxt(mode)$
    $status(runtime(self, mode)) = run$
    $task := onEntry$
  end if

end if

The invariants are evaluated by checking each invariant expression on the list.

$$isViolated(invariantList) \equiv$$
$$\exists\ exp \in invariantList,\ val(exp) = false$$

If one of the invariants is violated, the mode is skipped and the error verdict is set. As a consequence, the runtime status is set to *stopped*. If none of the invariants is violated, the on entry block and the mode's body are executed and the runtime status of the mode changes to *run*.

The on entry block

The on entry block is executed when a mode is successfully entered (i.e. when none of the invariants are violated). It is an ordinary statement block, and, therefore, all of its contained statements are subsequently executed.

$$\textbf{let}\ onEntry = (\texttt{onEntry}, stmt_1, stmt_2, ..., stmt_n)\ \textbf{in}$$
$$fst(onEntry) = fst(stmt_1)$$
$$nxt(stmt_i) = fst(stmt_i + 1), 0 < i < n$$
$$nxt(stmt_n) = nxt(onEntry)$$

$$\textbf{let}\ onEntry = (\texttt{onEntry})\ \textbf{in}$$
$$fst(onEntry) = nxt(onEntry)$$

When it is empty, a block is skipped. According to the specification above, when it ends, the on entry block triggers the execution of the mode's body.

The repeated execution of modes

In runtime status *run*, the execution of a mode is repeated for each sampling step. During this step, the TE triggers different subtasks of a mode; each subtask, in turn, initiates the execution of one of the modes's subblocks. The subtask *invariants* triggers the calculation or recalculation of the invariant

expressions; the sub-task *guards* triggers the execution of the until block and the evaluation of the guards; the sub-task *body* triggers the execution of the mode's body; and the sub-task $finalize$ completes the processing of the mode's current sampling step.

Rule 13: ASM transition rule for mode execution (repeating).

```
if task is mode                                   ▷ (mode execution, repeating)
    ∧ status(runtime(Self, mode)) = run then
  if subtask(runtime(Self, mode)) = invariants then
    task := fst(invariantList)
    subtask(runtime(Self, mode)) = guards
  else if subtask(runtime(Self, mode)) = guards then
    task := fst(until)
    if isViolated(invariantList) then
      verdict(Self) := error
      status(modeState(Self, mode)) = stopped
      task := nxt(mode)
    else
      subtask(mode) = body
    end if
  else if subtask(runtime(Self, mode)) = body then
    subtask(mode) = finalize
    if endOfChilds then
      task := onExit
    else
      task := fst(contBody)
    end if
  else if subtask(runtime(Self, mode)) = finalize then
    if endOfStepExecution then
      resetModes(Self)
      status(Self) := wait
    else
      task := up(up(mode))
    end if
  end if
end if
```

Before the body is executed, it is important to check whether a parallel or sequential mode has come to its natural end. This is important for both kinds of mode, if there is no contained mode that still has the runtime status *running*. The check is achieved by the macro $endOfChilds$. This macro

iterates over all runtime states. It checks whether or not the runtime status is *running*, and checks if the belonging mode is a child of the current mode.

$$endOfChilds(mode, c) \equiv$$
$$\forall\, rs \in Runtime_c,\ (mode = up(up(mode(rs)))) \rightarrow status(rs) \neq running$$

After the mode's body has been executed, the mode retains the runtime status *running* and executes the runtime subtask $finalize$. In this phase, some final checks are carried out before the execution of the mode is suspended; the mode waits until the next sampling step triggers its re-execution. Before the execution is suspended, the runtime subtask $finalize$ checks whether all running modes of a component have been suspended (i.e., whether all have reached the subtask $finalize$). This indicates that the execution has returned to the outermost mode and that all tasks (for this sampling step, on this component) have been accomplished. The check is done by the macro $endOfStepExecution$. If the macro evaluates as true, the mode's runtime states are reset by $resetRuntimes$ and the overall status of the component is set to *wait*. The execution of the component is then completely suspended until the next sampling step begins.

$$endOfStepExecution \equiv$$
$$\exists\, runtime \in runtimes(Self)\ ,\ subtask(runtime) \neq finalize$$

$$resetModes(c) \equiv$$
var *runtimes* **rangesover** $Runtime_c$
  if $status(runtime) = running$ **then**
    $subtask(runtime) = invariants$
  end if
end var

It should be noted that the execution of the mode's body shows different behaviour for each different kind of mode. The exact description can be found in the definition of the static execution order for the different elements, in the corresponding sections below.

If a mode is in runtime status *stopped*, it is checked whether an activation flag has been set. If it is set — and this is normally the case when a transition statement has already been executed once before — the mode is set to the runtime status *init* and is thus reactivated again. If not, it is simply skipped and the execution proceeds with the statement that directly follows the mode.

Rule 14: ASM transition rule for mode execution (stopped mode).

if *task* **is** *mode* ▷ (mode execution, stopped mode)

$\wedge\ status(runtime(Self, mode)) = stopped$ **then**
if reactivateNext(Self) **then**
$reactivateNext(Self) := false$
$status(runtime(Self, mode)) := init$
else
proceed
end if
end if

The until block

The until block contains a set of until guard statements; these statements define transitions to other modes. It should be noted that the concrete syntax alllows certain short forms and omissions. In the abstract syntax, these short forms and omissions are all resolved into the below-depicted form.

Abstract syntax 12: The until block.

UntilBlock	::=	"`until`" { *UntilGuardStatement* }
UntilGuardStatement	::=	*EXP GuardOp StatementBlock* [*UntilJump*]

An until block consists of so-called until guard statements. An until guard statement specifies a guard that, when passed, ends the enclosing mode. If the predicates of such a guard match, the rest of the until guard statement is triggered; the mode is then left, without the other until guard statements being evaluated. The until guard statements in an until block are subsequently evaluated.

let $untilBlock = (\texttt{inv}, stmt_1, stmt_2, ..., stmt_n)$ **in**
$fst(untilBlock) = fst(stmt_1)$
$nxt(stmt_i) = fst(stmt_{i+1}),\ 0 < i < n$
$nxt(stmt_n) = up(untilBlock)$

let $untilBlock = ()$ **in**
$fst(untilBlock) = up(untilBlock)$

If none of the until guard statements are triggered the execution loops back to the beginning of the enclosing mode. If the until block is empty, it is simply skipped.

The execution of an until guard statement begins with the calculation of the guard expression. It should be noted that the concrete syntax allows guard expressions to be omitted. In such a case, the guard expression in the abstract syntax is set to *true*.

let $untilGuardStatement = (exp, guardOp, statementBlock)$ **in**
$fst(untilGuardStatement) = fst(exp)$
$nxt(exp) = untilGuardStatement$
$nxt(statementBlock) = nxt(up(untilGuardStatement))$

let $untilGuardStatement = (exp, guardOp, statementBlock, transition)$ **in**
$fst(untilGuardStatement) = fst(exp)$
$nxt(exp) = untilGuardStatement$
$nxt(statementBlock) = untilJump$
$nxt(untilJump) = nxt(up(untilGuardStatement))$

Afterwards, the guard expression and guard operation are both evaluated (see rule below). The respective function updates respect the evaluation of the guard expression and initiate the execution of the guard operation.

Rule 15: ASM transition rule for the until-guard statement.

```
if task is untilGuardStatement then                ▷ (until-guard statement)
    if val(exp) then
        if guardOp = undef then
            task := statementBlock
        else
            task := guardOp
        end if
    else
        task := nxt(guardStatement)
    end if
end if
```

If the guard operation has not been specified in the concrete syntax, it appears as $guardOp = undef$ in the abstract syntax. Finally, if the statement

block of the until guard statement has been executed, the enclosing until block's successor determines the next task.

Transition statements

Transition statements are transfer the control flow between modes. Normally, if a mode ends, the next mode is executed. In a sequential environment, this is usually the mode following the mode that has just ended.

Abstract syntax 13: Transition statements.

Transition ::= *GotoTransition* *RepeatTransition* *ContinueTransition*

The go to transition allows "jumping" between modes that reside in the same statement block. During the creation of AST, the go to transition is enriched with information about the jump target. $nxt(gotoTransiton)$ thus automatically points to the statement which follows the label in the original specification. In the context of a mode, the go to transition initiates the potential reactivation of a mode, and also directs the control flow to execute the jump target after the on exit block has been executed.

Rule 16: ASM transition rule for the goto statement in modes.

---

if *task* **is** $GotoTransition$ **then** ▷ (goto statement in modes)
  $reactivateNext(Self) := true$
  $trans(runtime(up(up(up(RepeatMode))), Self)) := nxt(gotoTransition)$
end if

The repeat transition allows repetition of the current mode. It initiates the reactivation of the mode by setting the $reactivateNext$ flag and directs the control flow to repeat the current mode after the on exit block has been executed.

Rule 17: ASM transition rule for the repeat statement in modes.

---

if *task* **is** $RepeatTransiton$ **then** ▷ (repeat statement in modes)
  $reactivateNext(Self) := true$
  $trans(runtime(up(up(up(RepeatMode))), Self)) := up(up(up(RepeatMode)))$
end if

The continue transition allows repetition of the current mode without a reinitialization. It signals to the control flow to directly execute the current mode again. The on exit block, in this case, is skipped.

Rule 18: ASM transition rule for the continue statement in modes.

if *task* **is** *ContinueMode* **then** ▷ (continue statement in modes)
  $task := up(up(up(ContinueMode)))$
end if

The on exit block

The on exit block behaves, in fact, like an ordinary statement block: it is executed when a mode is left. Its execution, however, is preceded by the runtime status of a mode being set to *stopped*

$$\begin{aligned}&\textbf{let } onExit = (\texttt{onExit},\ stmt_1,\ stmt_2,\ \cdots,\ stmt_n) \textbf{ in}\\ &\quad fst(onExit) = fst(stmt_1)\\ &\quad nxt(stmt_i) = fst(stmt_i + 1), 0 < i < n\\ &\quad nxt(stmt_n) = onExit\end{aligned}$$

.

Rule 19: ASM transition rule for the on-exit statement.

if *task* **is** *onExit* **then** ▷ (on-exit statement)
  $status(runtime(Self, mode)) := stopped$
  $task := folowUp(runtime(Self, mode))$
end if

Body execution for atomic modes

An atomic mode is structured and executed according the scheme defined in the main Section above.

Abstract syntax 14: Atomic modes.

ContMode ::= "`cont`" *OnEntry Invariant ContBody OnExit Until*

The execution of an atomic mode differs from the execution of other modes in that it shows different behaviour for its body. The cont body is an ordinary statement block. The static order of its elements is defined below. When the cont body is entered, all containing statements are subsequently executed. When the end of the body is reached, the execution proceeds with the next statement scheduled to be executed within the current sampling step.

let $contBody = (\texttt{contBody},\ mode_1,\ mode_2,\ \cdots,\ mode_n)$ **in**
$fst(contBody) = fst(mode_1)$
$nxt(mode_i) = fst(mode_{i+1}),\ 0 < i < n$
$nxt(mode_n) = loop(up(contBody))$

let $contBody = ()$ **in**
$fst(contBody) = loop(up(contBody))$

Body execution for sequential modes

A sequential mode defines a sequential execution order for all directly-contained modes.

Abstract syntax 15: Sequential modes.

SeqMode ::= "`seq`" *OnEntry Invariant SeqBody OnExit Until*
SeqBody ::= { *Mode* }

The body of a sequential mode (the seq body) starts with the execution of the first mode statement and ends either when a mode statement with runtime status *running* executes its body (see definition of the function *loop*), or when the end of the seq body is reached (see definition of the function *next*). When a contained mode ends during the execution, the next mode in the sequence of contained modes activates and executes (see definition of the

function *nxt*).

let $seqBody = (\texttt{seqBody},\ mode_1,\ mode_2,\ \cdots,\ mode_n)$ **in**
$fst(seqBody) = fst(mode_1)$
$loop(mode_i) = up(seqBody),\ 0 < i \leq n$
$nxt(mode_i) = fst(mode_{n+1}),\ 0 < i < n$
$nxt(mode_n) = loop(up(seqBody))$

let $seqBody = ()$ **in**
$fst(seqBody) = loop(up(seqBody))$

Body execution for Parallel modes

A par mode defines a parallel execution order for all directly-contained modes.

Abstract syntax 16: Parallel modes.

ParMode	::=	"`cont`" *OnEntry Invariant ParBody OnExit Until*
ParBody	::=	{ Mode }

The body of a parallel mode (the par body) starts with the execution of the first mode statement and ends when the end of the par body is reached (see the definitions for the function *loop*).

let $parBody = (\texttt{parBody}\ , mode_1\ , mode_2,\ \cdots,\ mode_n)$ **in**
$fst(parBody) = fst(mode_1)$
$loop(mode_i) = fst(mode_{n+1}),\ 0 < i < n$
$loop(mode_n) = nxt(up(parBody))$
$nxt(mode_i) = loop(mode_i),\ 0 < i \leq n$

let $seqBody = ()$ **in**
$fst(parBody) = loop(up(parBody))$

5.6.4 $TTCN_{\mathcal{E}}$ duration expression

The duration expression yields the time period (in seconds) for which a mode has been running.

Rule 20: ASM transition rule for the duration expression in modes.

if *task* **is** *duration* **then** ▷ (duration expression)
  $val(duration) := t'_{sample} - starttime(runtime(parent(duration), Self)$
  proceed
end if

5.7 DISCUSSION

These operational semantics discuss the basic statements of *TTCN-3 embedded*. Constructs for describing reusable modes are not part of this specification; this is because they are resolved to normal modes at compile time. The semantics have been developed to simplify discussions about the different possible interpretations of the runtime behaviour of *TTCN-3 embedded*'s intuitive semantics. The aim of this chapter is to provide a means of describing the open issues regarding the runtime behaviour and to document their solution in an easily-understandable form.

These operational semantics have been developed without concrete tool support. They have not been checked by a compiler, nor by a runtime environment. These semantics have, however, been the basis for both of the *TTCN-3 embedded* runtime implementations developed and used in this thesis as proof of concept. These implementations are the basis for both the Vector CANoe integration and the Matlab/Simulink integration; these integrations are outlined in the next sections. When creating the *TTCN-3 embedded* runtime environment, these semantics have been revised and adjusted several times. This means that the basic rules can be improved upon. Even so, there is currently no explicit proof that these semantics are free of small — especially syntactical — errors.

Chapter 6

Architecture for realization

This chapter describes the base architecture for a runtime environment which allows for the realization and application of the language constructs described in the previous chapters. Since *TTCN-3* already specifies a runtime architecture, this same architecture is used as the basis for *TTCN-3 embedded*; it is extended to incorporate the requirements for the testing of hybrid and continuous real-time systems.

6.1 The overall runtime architecture

A *TTCN-3* test system consists of several entities that perform distinct and different tasks during test execution. The overall architecture of a *TTCN-3* test system is depicted in Figure 6.1. For this thesis, the most relevant entities are the Test Executable (TE), the SUT Adaptor (SA), the Platform Adaptor (PA) and the Component Handling (CH).

- The TE is responsible for the interpretation and/or execution of the *TTCN-3* programmes. It provides services for the interpretation of the language constructs and an environment for the handling of messages. In this thesis, the TE is extended, so that the language constructs of *TTCN-3 embedded* are supported and the sampled dispatch and reception of messages from data streams becomes possible.
- The SA adapts the *TTCN-3* test system for the SUT. The SA is aware of the mapping between the *TTCN-3* communication ports and the test system interface ports. It implements the real test system interface. During the test execution, it broadcasts send requests from the TE to the SUT. Furthermore, it notifies the TE of any received test events; it does this by appending the events to the port queues of the TE. In this thesis, the SA is extended to enable the stream-based communication of *TTCN-3 embedded*.
- The PA binds the TE and test platform with respect to their time-controlled operations and the *TTCN-3* external functions. The interface between the TE and the PA allows for the TE to have transparent

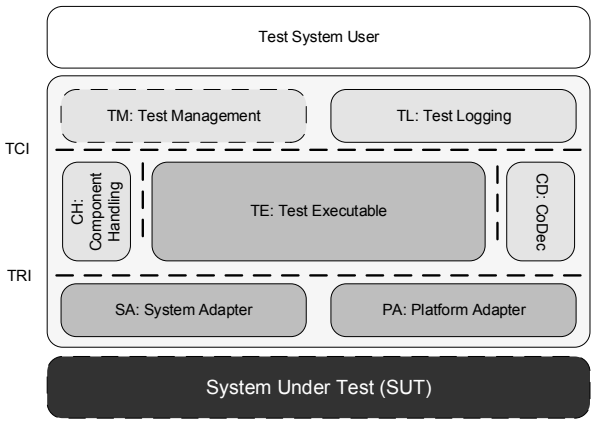

Figure 6.1: The overall *TTCN-3* test architecture

timer control (creating, starting, stopping). Additionally, the PA notifies the TE if a timer has expired. In *TTCN-3 embedded*, the PA is extended to provide the status of the overall clock and to actualize the sampling controller. With respect to sampling, the PA notifies the TE of every new sampling step.

- The CH is responsible for managing distributed, parallel test components. This distribution can occur across one, or many, physical systems. In this thesis, the CH is extended to support time synchronization between physically-distributed test components.

Other entities of the *TTCN-3* test architecture depicted in Figure 6.1 do not change in this thesis. These entities are needed in both *TTCN-3* and *TTCN-3 embedded* test systems. Their main purposes are discussed briefly below.

The Test Management (TM) is responsible for overall test management and functionality. The unit's main responsibility is the initiation of test execution; additionally, it usually provides the interface used by test system users. The Test Logging (TL) maintains the overall test log. It provides an unidirectional interface that receives logging requests from the TE and stores the test logs in a dedicated format. Last but not least, the external encoding and decoding of data associated with message-based, procedure- or stream-based communication is achieved by the Coding (CD). The external codecs have a standardized interface which allows for their reuse between different *TTCN-3* systems and tools.

The *TTCN-3* standard defines the interfaces that enable the exchange of information between the entities described above. Through this standardiza-

tion, the application of the entities is mostly defined and it becomes possible to replace individual entities.

The *TTCN-3* standard distinguishes two interface sets, the *TTCN-3* Control Interface (TCI) and the *TTCN-3* Runtime Interface (TRI). The TCI specifies the interface between Test Management, Test Logging, Component Handling, Encoding/Decoding and the *TTCN-3* Test Executable; the TRI, on the other hand, specifies the interface between the *TTCN-3* Test Executable, the SUT Adaptor and the Platform Adaptor.

In the following text, the functional extensions of the test system entities are described in detail on the basis of their interfaces (TRI and TCI) and complementary sequence diagrams. The basis for the extension is the *TTCN-3* standard. *TTCN-3 embedded*, which is an extension of the *TTCN-3* standard, provides the interfaces defined in the *TTCN-3* standard, as well as the extensions defined in the following sections.

6.2 Extensions of the *TTCN-3* Test Runtime Interface (TRI)

The TRI is responsible for the interaction between the TE and the SA and PA. It controls the interaction with the SUT and allows singular access to time. To ensure the functionality of *TTCN-3 embedded*, the TRI has to be extended; this enables the sending and receiving of messages, which is a necessary tool for stream-based communication. In addition, the TRI defines access to the global clock, and provides functions for the realization of the sampling.

6.2.1 Access to time

All time-related operations in *TTCN-3* are accessed via the platform adapter. This allows a controlled and singular-access time; it also facilitates the support of different time models that can be implemented via different implementations of the interfaces. *TTCN-3*'s original TRI already supports timer functionality, but the access to a global clock must be added for *TTCN-3 embedded*. Global clock access is added via the TRI operations *triStartClock*and *triReadClock*. The operation *triStartClock* starts the test system clock with a given precision. The precision is defined by the parameter *ticksPerSecond*; this parameter specifies the number of time units (ticks) that constitute one second.

The TRI operation *triReadClock* yields the actual clock value. The clock value is made available through out parameter *timepoint*, which represents the number of time units (ticks) that have elapsed since the clock's start.

Signature	**TriStatus** triStartClock(**in long** ticksPerSecond **in integer** initalTimeValue)
In Parameters	*ticksPerSecond*: the precision of the clock given in ticks per second. *initialTimeValue*: the initial test case start time value e.g. in case an the test case has been started remotely. If the value is set to 0 it is ignored and the current time is taken.
Out Parameters	n.a.
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation.
Constraints	n.a.

Table 6.1: TRI operation: triStartClock

Signature	**TriStatus** triReadClock(**out long** timepoint)
In Parameters	n.a.
Out Parameters	*timepoint:* current time
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation.
Constraints	There was a preceding invocation of *triStartClock(int lont ticksPerSecond.*)

Table 6.2: TRI operation: triReadClock

Figure 6.2 shows the interaction between the test executable (TE) and the TRI. After the test executable has called the operation $triExecuteTestcase$ on the SA, the operation $triStartClock$ is called and the global clock starts. Afterwards, the TE can call the operation $triReadClock$ to check the current time value at any point during the test.

6.2.2 TRI wait operations

The TRI realization for the wait-statement requires the operations $triBeginWait$ and $triEndWait$. The operation $triBeginWait$ is called by the TE and provided by the PA, and the operation $triEndWait$ is initiated by the PA and provided by the TE.

The begin wait operation signals that the execution of a component should be suspended until a specified point of time. A call to the begin wait operation returns immediately. If the begin wait operation's parameter *timepoint* represents a point of time in the past, the operation returns a TRI_Error value and has no further effect.

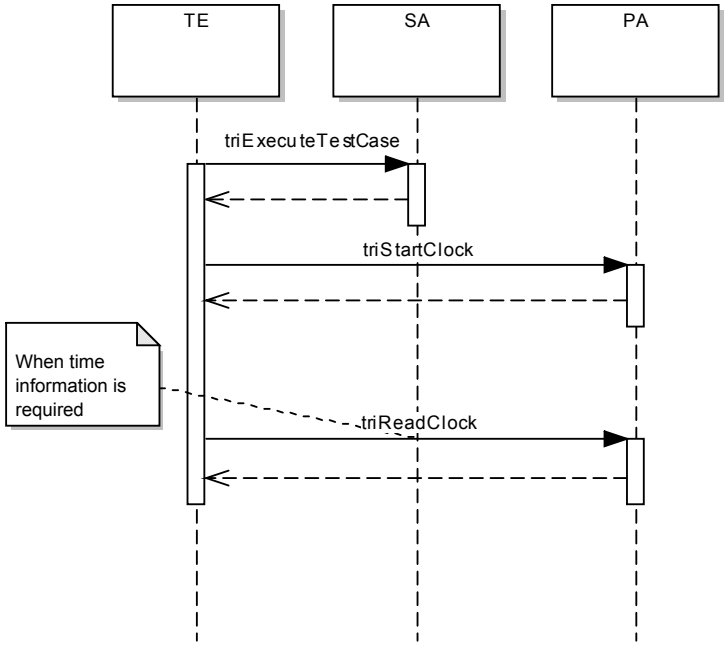

Figure 6.2: The initialisation of the test system clock

Signature	**TriStatus** triBeginWait(**in long** timepoint, **in TriComponentIDType** componentId)
In Parameters	*timepoint*: point in time until which the execution of a component should be suspended. *componentId*: component whose execution should be suspended.
Out Parameters	n.a.
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation.
Constraints	There was a preceding invocation of *triStartClock(int lont ticksPerSecond).*

Table 6.3: TRI operation: triBeginWait

If the test component is suspended, the values at the test component's input ports are not evaluated and the values at the output ports do not change. Incoming stream values and messages are, however, stored in the streams and message queues, so that they will be available when the component is once again reactivated. For reactivation, the PA will issue a call of the operation $triEndWait(component)$.

Figure 6.3 shows the interaction that occurs between the test executable

Signature	**TriStatus** triEndWait(**in TriComponentIDType** componentId)
In Parameters	*componentId:* the component ID referring to the component that shall be reactivated at the operation call.
Out Parameters	n.a.
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation.
Constraints	There was a preceding invocation of *triNextSampling(timepoint, port).*

Table 6.4: TRI operation: triEndWait

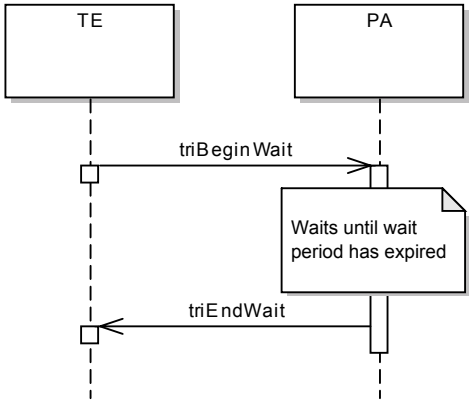

Figure 6.3: The wait operations

(TE) and the TRI when the begin wait operation is called. After the test executable has called the operation *triBeginWait* at the PA, the PA starts the wait process. This holds until the point in time set by the begin wait operation. Once this end time has been reached, the PA immediately triggers the execution of *triEndWait* and the suspended component is reactivated again.

6.2.3 TRI stream value access

The TRI operations *triSetStreamValue* and *triGetStreamValue* facilitate message exchange between the TE and the SA. While the set stream value operation is used to update messages sent to the SUT, the get stream value operation can update the messages coming from the SUT. Both operations are typically called after the TE has been informed that a new sampling step for the respective port has started.

Signature	**TriStatusType** triSetStreamValue(**in TriComponentIdType** componentId, **in TriPortIdType** tsiPortId, **in TriAddressType** SUTaddress, **in TriMessageType** streamValue)
In Parameters	*componentId*: identifier of the sending test component. *tsiPortId*: identifier of the test system interface port, via which the message is sent to the SUT Adaptor. *SUTaddress* (optional): destination address within the SUT. *streamValue*: the encoded stream value (message) to be sent.
Out Parameters	n.a.
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation.
Constraints	The TE calls this operation when it executes a new sampling step (on a sampled-output stream port that has been mapped to a TSI port). Unless one particular system component is specified for a test case (e.g. only a MTC test component is created for a test case), the TE calls the operation for all sampling steps of all outgoing stream ports. The encoding of streamValue has to be done in the TE prior to this TRI operation call.

Table 6.5: TRI operation: triSetStreamValue

The set stream value operation sets a message that is to be sent to the SUT. If the case has been completed successfully, it returns TRI_OK. Otherwise, it returns TRI_Error. It is important to note that the return value TRI_OK does not necessarily imply that the SUT has received the stream value.

The get stream value operation obtains the actual stream value for an input port. The operation returns TRI_OK if it is completed successfully. Otherwise, it returns TRI_Error.

6.2.4 TRI sampling

Sampling is one of the most powerful new features that *TTCN-3 embedded* introduces to the *TTCN-3* language. The sampling of component ports is achieved via the TRI operations $triNextSampling$, $triProcessStep$, and $triEndProcessing$.

The next sampling operation signals that the next sample-step for a given port will start at the specified point in time. The next sampling

Signature	**TriStatusType** triGetStreamValue(**in TriComponentIdType** componentId, **in TriPortIdType** tsiPortId, **in TriAddressType** SUTaddress, **out TriMessageType** streamValue)
In Parameters	*componentId*: identifier of the receiving test component. *tsiPortId*: identifier of the test system interface port via which the message is received from the SUT Adaptor. *SUTaddress* (optional): destination address within the SUT.
Out Parameters	*streamValue*: the encoded stream value (message) that has been received from the SUT.
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation.
Constraints	The TE calls this operation when it executes a new sampling step onto a sampled-input stream port that has been mapped to a TSI port. Unless a particular system component has been specified for the test case (e.g. only a MTC test component is created for the test case), the TE calls the operation for all sampling steps of all incoming stream ports. After this TRI operation is called, the TE must decode the streamValue.

Table 6.6: TRI operation: triGetStreamValue

Signature	**TriStatus** triNextSampling(**in long** timepoint, **in TriPortIDType** port)
In Parameters	*timepoint*: point in time when the execution of the next sample step for a given stream port shall be started. *port*: the stream port the sample step is requested for.
Out Parameters	n.a.
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation.
Constraints	There was a preceding invocation of *triStartClock(int lont ticksPerSecond).*

Table 6.7: TRI operation: triNextSampling

operation's *timepoint* parameter is described as the number of time units (ticks) that have elapsed since the start of the test case (see TRI operation *triStartClock*). A call to this operation returns immediately. The operation simply triggers a corresponding operation (the process step operation). If the parameter *timepoint* represents a point in the past, the operation returns a *TRI_Error* value and has no further effect.

Signature	**void** triProcessStep(**in TriPortIDListType** ports)
In Parameters	*ports*: a list of ports that shall be sampled at the operation call.
Out Parameters	n.a.
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation.
Constraints	There was a preceding invocation of triNextSampling(timepoint, port).

Table 6.8: TRI operation: triProcessStep

If the point in time for a new sampling step is reached, the PA issues a call for the operation *triProcessStep* to inform the TE which ports (and thus components) shall be sampled next. The operation *triEndProcessing* signals that the results for the step have been calculated and that the TE is ready to process the next sampling step.

Signature	**void** triEndProcessing()
In Parameters	n.a.
Out Parameters	n.a.
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation.
Constraints	There was a preceding invocation of *triProcessStep(ports).*

Table 6.9: TRI operation: triProcessStep

Figure 6.4 shows the interaction between the test executable (TE) and the TRI during an example of a sampling loop. It starts with the preamble that initializes the PA and resets the global clock at the beginning of each test case. Afterwards, the step size for each port is transmitted to the PA by the execution of the next sampling operation for each activated port. The PA then takes that, for each port, each new sampling step is communicated to the TE. Each new sampling step is communicated via the the process step operation, and thusly the TE can start any sampling-dependent actions. Typically the TE updates the values for the input streams (see the get stream value operation) and output ports (see the set stream value operation) and calculates the new values (to be updated at the next sampling step).

6.3 Extensions of the *TTCN-3* Test Control Interface (TCI)

The TCI covers operations that govern the overall management of the test system, the external encoding and decoding of *TTCN-3* values, the man-

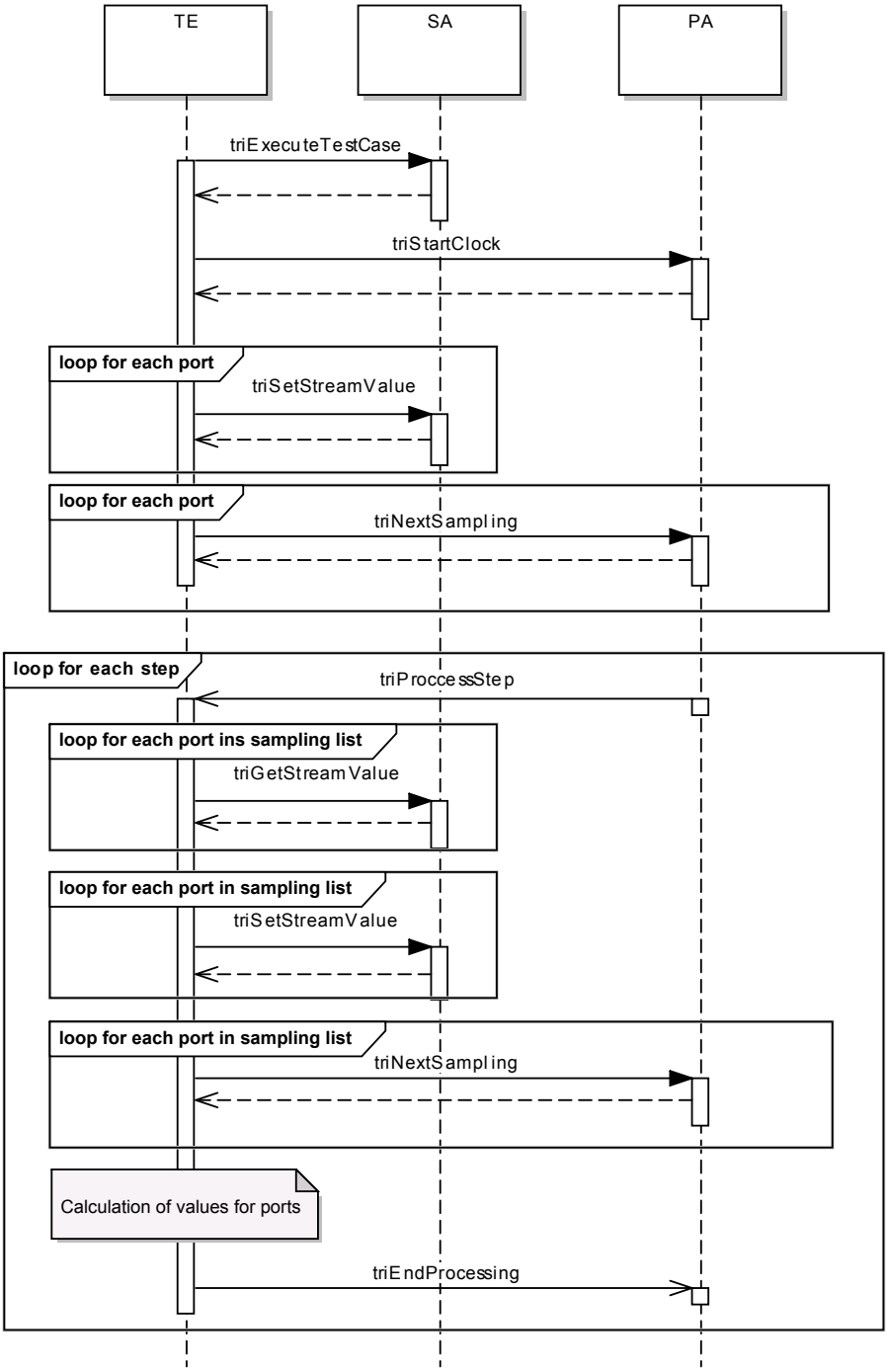

Figure 6.4: The sampling mechanism

agement of potentially distributed components, and the test logging. For *TTCN-3 embedded*, an extension of the interface is necessary to allow for stream-based communication between components and synchronisation of (potentially distributed) components with respect to the global clock.

6.3.1 TCI stream value access

In addition to the TRI operations defined in Section 6.2.3, further TCI operations are required in order to enable the stream-based communication between (potentially distributed) test components. The TCI operations *tciSetStreamValue* and *tciGetStreamValue* are nearly identical to the aforementioned TRI operations: they realize stream-based interactions between distributed test components. The set stream value operation sets a message that is to be sent to another component.

Signature	**void** tciSetStreamValue(**in TriComponentIdType** componentId, **in TriPortIdType** tsiPortId, **in Value** streamValue)
In Parameters	*componentId:* identifier of the sending test component. *tsiPortId:* identifier of the test system interface port via which the message is sent to the SUT Adaptor. *streamValue:* the stream value (message) to be sent.
Out Parameters	n.a.
Return Value	*void*
Constraints	The TE calls this operation when it executes a new sampling step on a sampled-output stream port that has been mapped to another component port. The TE calls the operation for all sampling steps of all outgoing stream ports.

Table 6.10: TCI operation: tciSetStreamValue

The get stream value operation (see Table 6.11) obtains the actual stream value for an input port that is mapped to another component.

6.3.2 Component synchronisation and global clock

If the hardware being tested is distributed, time synchronisation needs to be considered carefully. In standard *TTCN-3*, distribution exists on the component level; time synchronisation, therefore, must to be achieved between distributed test components. Synchronisation becomes a general problem, however, when test components are executed on distributed hardware entities that have their own clocks. The synchronisation of distributed hardware

Signature	**void** tciGetStreamValue(**in TriComponentIdType** componentId, **in TriPortIdType** tsiPortId, **out Value** streamValue)
In Parameters	*componentId:* identifier of the receiving test component. *tsiPortId:* identifier of the test system interface port via which the message is received from the SUT Adaptor. *streamValue:* the stream value (message) that has been received from the SUT.
Out Parameters	n.a.
Return Value	*void*
Constraints	The TE calls this operation when it executes a new sampling step on a sampled-input stream port that has been mapped to another component port. The TE calls the operation for all sampling steps of all incoming stream ports.

Table 6.11: TCI operation: tciGetStreamValue

clocks is not an objective of *TTCN-3 embedded*. Such a synchronisation can be achieved by common methods such GPS time synchronisation or the use of the NTP protocol. Distributed clock synchronisation thereby achieved, the *TTCN-3 embedded* test infrastructure must ensure that the global test clock, i.e. the call to *triGetClock*, returns an identical time value to all distributed test components. Synchronizing the test clock with the test clock of the MTC at the beginning of a test component achieves this desired time synchronisation. In the following considerations, it has been assumed that the clocks of the distributed hardware are synchronised with acceptable accuracy.

In order to get synchronised time for distributed test components, related time values need to be exchanged by means of the TCI interface at both testcase start time and component start time. At testcase start time, the value of t_{start} is distributed by means of the TCI operations $tciExecuteTestCase$ and $tciExecuteTestCaseReq$ respectively. This enables each test entity to calculate the current testcase time. For this purpose, the already-existing operation $tciExecuteTestCase$ and $tciExecuteTestCaseReq$ are extended with an additional parameter: $startTime$, which is an integer-type parameter that holds the start time of the test case in UNIX time. Table 6.12 specifies the corresponding changes in the operation signature of $tciExecuteTestCase$.

At component start, the test-case-related time values t_{sample} and Δ are propagated via the operations $tciStartTestComponent$ and $tciStartTestComponentReq$. This means that each test component is therefore aware of its start time and the time at which it must proceed with the next sampling step. All of the asynchronous *TTCN-3* concepts, such as timers and snapshot semantics, are generally controlled by the sampling loop.

Signature	**void** tciExecuteTestCase(**inTciTestCaseIdType** testCaseId, **in TriPortIdListType** tsiPortList, **in Integer** testCaseStartTime)
In Parameters	*testCaseId*: a test case identifier as defined in the *TTCN-3* standard. *tsiPortId*: a list of port identifiers as defined in the *TTCN-3* standard. *startTime*: the start time of the test case in UNIX time.
Out Parameters	n.a.
Return Value	*void*
Constraints	This operation shall be called by the CH at the appropriate local TE when at a remote TE an execution request (in course of a TTCN-3 execute operation) has been called.

Table 6.12: TCI operation: tciExecuteTestCase

Communication-related delays and synchronization, however, may impose restrictions on the accuracy of time measurement and the highest-achievable sampling rate. Table 6.13 specifies the corresponding changes in the operation signature of *tciStartTestComponen*

Signature	**void** tciStartTestComponent(**inTriComponentIdType** componentId, **in TciBehaviourIdType** behaviour, **in TciParameterListType** parameterList, **in Float** componentStartTime)
In Parameters	*componentId*: identifier of the component to be started as defined in the *TTCN-3* standard *behaviour*: identifier of the behaviour to be started on the component. *parameterList*: a list of Values where each value defines a parameter from the parameter list as defined in the TTCN-3 function declaration of the function being started. *componentStartTime*: the start time of the test case in UNIX time.
Out Parameters	n.a.
Return Value	*void*
Constraints	This operation shall be called by the CH at the local TE when at a remote TE a TTCN-3 start operation has been called.

Table 6.13: TCI operation: tciStartTestComponent

Figure 6.5 describes the synchronization of distributed test components.

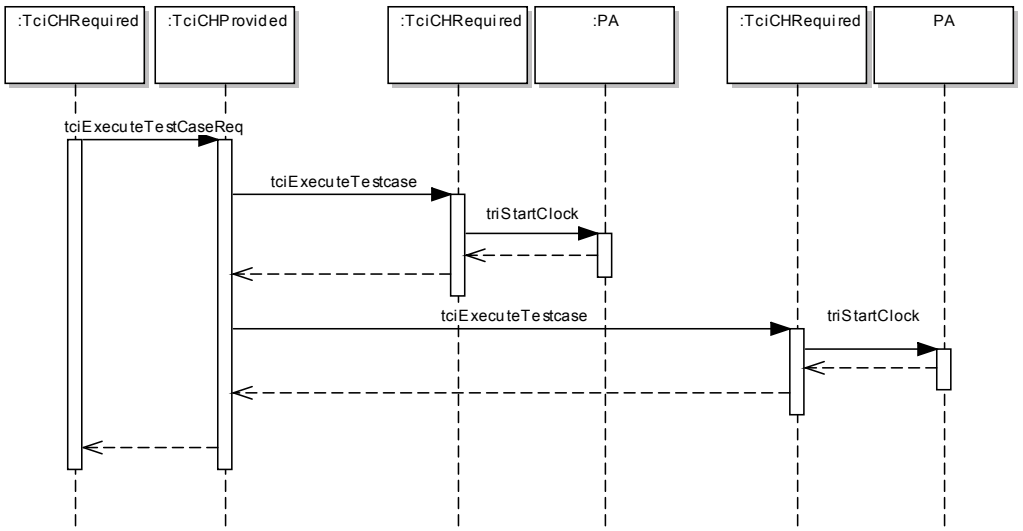

Figure 6.5: The initialisation of the test system clock in a distributed environment

6.4 Integration with Matlab Simulink

Matlab Simulink [117] has become one of the major platforms for modelling, simulating and testing embedded automotive real-time systems. It provides exhaustive means for the modelling of both continuous and hybrid behaviour, and it allows for the execution of models on different hardware platforms. Matlab Simulink's execution and simulation environment is open for extensions; this means that complementary tools, e.g. testing tools, can be adapted to the simulation infrastructure and provide their services during a simulation run.

The integration of *TTCN-3 embedded* into the MATLAB and Simulink tool suites shows the ability of *TTCN-3 embedded* to interact with the formalisms of MATLAB Simulink. When integrated into the MATLAB and Simulink suites, *TTCN-3 embedded* becomes able to perform systematic MiL test cases. The *TTCN-3 embedded* test executable code is integrated as a Simulink S-function that automatically executes during a simulation run.

6.4.1 Simulink S-function overview

A Simulink S-function is a computer language realization of a Simulink block. Such blocks can represent multiple equations and formally consist of a set of inputs u, a set of states x and a set of outputs y. The inputs and outputs are referred to as signals; these signals describe time-varying quantities that

have both a name and a data type. The outputs are calculated formally by the algorithim $y := f_B(t, x, u)$. The execution of an S-function is divided into an initialization phase (wherein the S-function and its states, inputs and outputs are initialized), a simulation phase (wherein the outputs are calculated for each time step) and a finalization phase (wherein the S-function and its resources are freed).

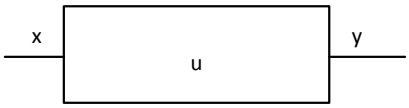

Figure 6.6: Simulink block principles

To serve the tasks of all three phases, a Simulink S-function provides a set of callback methods. These callback methods can be used for the overall initialization of the S-function, for the calculation of the block's outputs and for the updating of the block's states. During a simulation run, Simulink calls the appropriate methods for each S-function block in the model. The main functions are described below.

- The functions *mdlInitializeSizes*, *mdlInitializeSampleTimes*, *mdlStart* and similar functions (see Matlab documentation for details) initialize the S-function prior to execution. These functions are used to initialize the data structures, to set the number and dimensions of input and output ports, to set the block's sample times and to allocate storage areas.

- The function *mdlGetTimeOfNextVarHit* calculates the next sample step; this is only necessary if the block uses a variable sample time.

- The function *mdlOutputs* calculates the outputs for major and minor time steps.

- The function *mdlUpdate* updates the discrete states of the S-function for each major time step; this function is optional.

- The functions *mdlDerivative* and *mdlZeroCrossing* apply to blocks with continuous states. The functions are called for each minor time step and calculate the continuous states of an S-function.

- The function *mdlTerminate* is called at the end of a simulation.

To drive the simulation, Simulink provides discrete solvers and continuous solvers. Discrete solvers either have a fixed step size or a variable step size. A fixed-step solver chooses an optimal step size that is fast enough to execute state changes in the fastest block of a model. A variable-step solver adjusts the step size during the simulation such that unnecessary steps are avoided and, thus, the simulation time is optimized. Both kinds of discrete solver trigger (subsequently and for each major step) the execution of *mdlOutputs* and *mdlUpdate*. Some continuous solvers subdivide the simulation timespan into major and minor time steps. These types of solvers initiate the calculation of results at each major time step and at each minor time step. Minor time steps, in this case, represent a subdivision of the major time step and are used to improve the accuracy of the results at major time steps. Just like discrete solvers, these kinds of continuous solvers trigger the execution of *mdlOutputs* and *mdlUpdate* subsequently. Additionally, *mdlOutputs* and *mdlDerivatives* are called for each minor time step.

6.4.2 The Simulink S-function adapter and codec

The Simulink S-function Adapter adapts *TTCN-3 embedded* to Matlab Simulink S-functions and allows for the integration of the *TTCN-3* TE as a Matlab Simulink S-function block. The adapter implements the callback function from Simulink and then implements the *TTCN-3 embedded* TCI/TRI operations defined in Section 6.2 and Section 6.3. The Matlab Simulink S-function CoDec allows for the mapping of *TTCN-3 embedded* data structures onto Matlab Simulink native C types. The current implementation of the Matlab Simulink S-function CoDec only supports encoding and decoding of basic *TTCN-3* types, such as *integer*, *float*, *boolean* and enumerations. It should, however, be possible to extend the CoDec. A principal approach on how complex data (such as that which occurs in *TTCN-3*) can be mapped is already outlined in [9]. Figure 6.7 shows the high-level architecture of the Simulink S-function adapter. Please note that the adapter provides the functionality of a *TTCN-3* Platform Adapter (PA) and System Adapter (SA).

In the following, the mapping of TCI/TRI operations with corresponding S-function callback functions are explained; the different phases passed through during the execution of a Matlab Simulink simulation are also discussed. The interactions between related components are illustrated with sequence diagrams.

During the *initialization phase* Matlab Simulink calls *mdlInitializeSizes*, *mdlInitializeSampleTimes*, and *mdlStart*. *mdlInitializeSizes* and *mdlInitializeSampleTimes* are used to set internals of the adapter, and

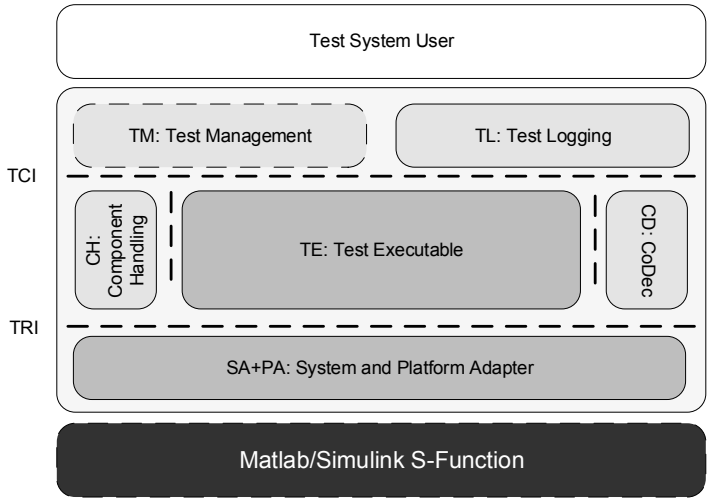

Figure 6.7: Architecture of the Simulink S-function adapter

the function *mdlStart* triggers the TCI operation *tciStartTestCase* at the TE. The TE itself initialises the test case that is being executed and calls *triExecuteTestCase* at the S-function adapter.

The operation *triExecuteTestCase* resets the adapter and prepares it for test execution (i.e., it transfers the amount and names of the input and output ports). Afterwards, the TE triggers the starting of the clock and sets the initial sample times for each of the ports via *triNextSampling*. After these operations, Matlab Simulink begins the execution of the simulation loop.

During the *simulation phase*, Matlab Simulink calls *mdlGetTimeOfNextVarHit*, *mdlOutputs* and *mdlUpdate* for each simulation step. The function *mdlGetTimeOfNextVarHit* is used to broadcast the sample times within the Matlab Simulink environment; the function *mdlOutputs*, on the other hand, triggers the call of *triProcessStep* at the TE, which triggers the calculation of the *TTCN-3* values for the current sampling step. The TE itself updates the input values by calling *triGetStreamValue*, propagates the output values to the Simulink S-function Adapter by calling *triSetStreamValue*, and informs the adapter about the new sample times by calling *triNextSampling*. Afterwards, the TE calls *triEndProcessing*, which calculates the outputs for the next sample step and returns.

During the *finalisation phase*, Matlab Simulink calls *mdlTerminate*. This function internally calls *triEndTestCase* and thus frees all adapter resources from the perspective of both Matlab Simulink and and *TTCN-3*.

The current version of the Simulink S-function adapter does not support

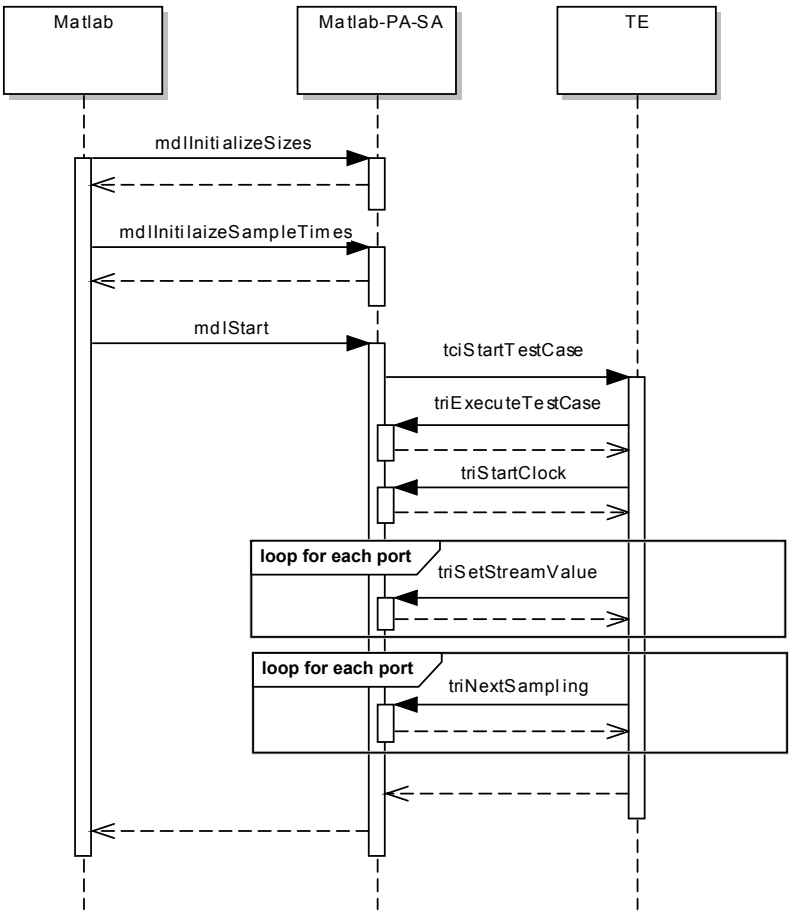

Figure 6.8: Initialization of a simulation run

advanced features like zero-crossing detection or explicit handling of continuous states. It can, however, be used with discrete and continuous solvers and has already been used for MiL testing of several typical automotive control systems (including an Adaptive Cruise Control, an Automatic Transmission Engine, and an Engine Controller) [45]. The examples have shown the applicability of the *TTCN-3 embedded* concepts for the intuitive definition of stimulation and assessment procedures, and particularly for the testing of systems with continuous signals and the execution of tests with simulation time.

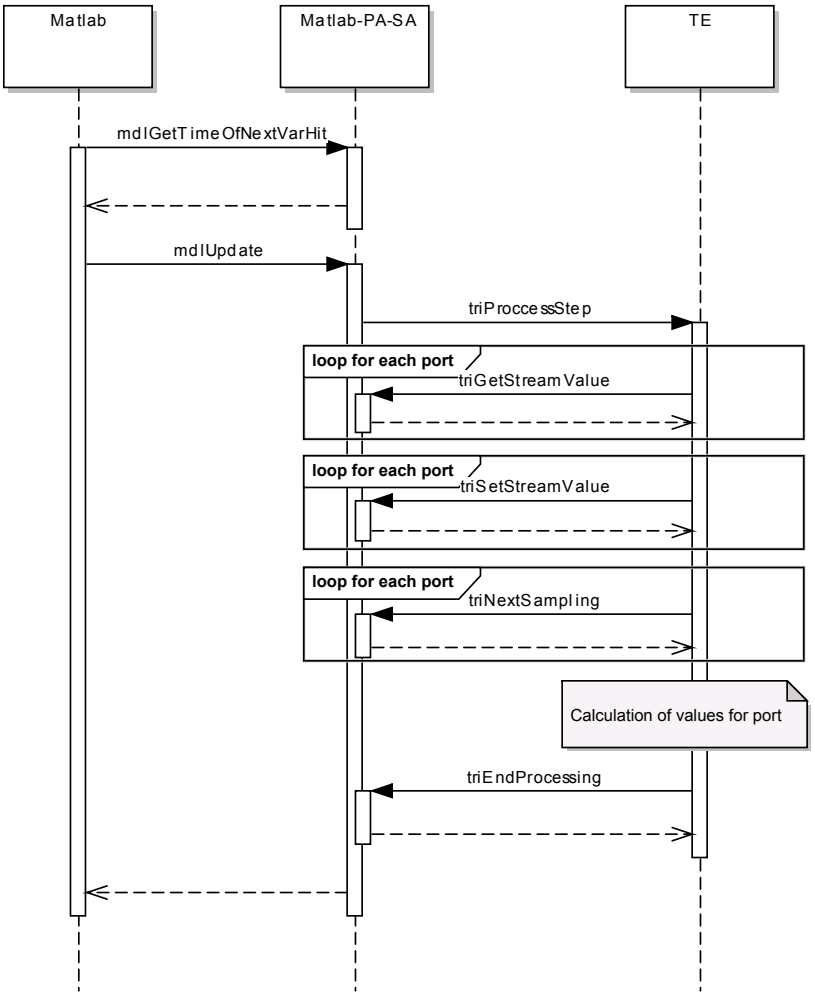

Figure 6.9: The simulation loop

6.5 Integration with Vector CANoe

The integration with Vector CANoe [125] allows for the testing of control systems in a simulation environment, as well as with real hardware in a networked environment. In this integration, the *TTCN-3* test executable is integrated as an independent test component of the CANoe test environment. The *TTCN-3* test executable interacts with the CANoe test environment and the SUT by adapting to the CANoe environment variables and the CANoe network API. The overall architecture and the adaptation scheme is similar to the approach described for Matlab Simulink, except that the CANoe adapter is realized in C# and, additionally, CANoe uses a real-time clock to indicate

time progress. In particular, the case studies realized on the basis of the *TTCN-3 embedded*-CANoe integration address the testing of control systems on both HiL and SiL levels. Case studies are available for an indicator and light control system [83], as well as for a window-lifter control system.

Chapter 7

Case Study Experiences

This section describes the results of two small case studies. These two case studies demonstrate the newly introduced concepts and show their applicability to automotive test tasks: both studies include test tasks typical for the automotive industry. The first case study (the MiL case study) shows the integration of *TTCN-3 embedded* with Matlab/Simulink. This setup demonstrates the applicability of *TTCN-3 embedded* concepts to the intuitive definition of stimulation and assessment procedures, particularly those which involve continuous signals and the execution of tests with simulation time. The second case study (the HiL case study) concerns the integration of *TTCN-3 embedded* with Vector CANoe. This case study is especially relevant because it addresses the testing of control systems on both HiL and SiL levels. Additional case studies discussing *TTCN-3 embedded* and its implementation in an automatic transmission engine, an engine controller, and an indicator and light control system [45], [83] have also been carried out. These case studies are available but are not included in this thesis.

7.1 The MiL case study

The first case study concerns a cruise control system that includes a distance control mode. This cruise control system is known as *Adaptive Cruise Control* (ACC), because it manages the speed of the ego vehicle while, at the same time, ensuring that it maintains a safe distance from the vehicle ahead. An activated ACC system monitors the road ahead and automatically detects vehicles in front of the car. If the ACC detects a slow vehicle ahead (this is known as a 'target' vehicle), it adjusts the running speed (*v_ego*) so that a safe distance from the detected vehicle ahead can be guaranteed. This is an example of ACC's distance control function. When there is no vehicle ahead, an ACC works like any other cruise control: it controls velocity.

Simple systems provide visual and acoustic warnings for the driver. A maximum brake activation of up to 25% of the maximal possible deceleration is allowed. Advanced systems are also equipped for the possibility of emergency braking: they aim to shorten the distance required to fully and

completely brake. Vehicles with adaptive cruise controls have been available on the market for several years, from a number of different manufacturers. The following sections describe how *TTCN-3 embedded* can be used to test an ACC. To ensure the industrial relevance of this case study's results, the overall test approach has been taken from [22].

7.1.1 The ACC system

Like many specified functions of contemporary automobiles, an ACC's functions are model-based. These models are often developed using the MATLAB® [116] and Simulink® tool suites. Figure 7.1 shows the outer view of the Simulink model used for a Model-in-the-Loop (MiL) testing of an ACC system (ACCS). This picture provides an overview of the testable interface of the ACCS.

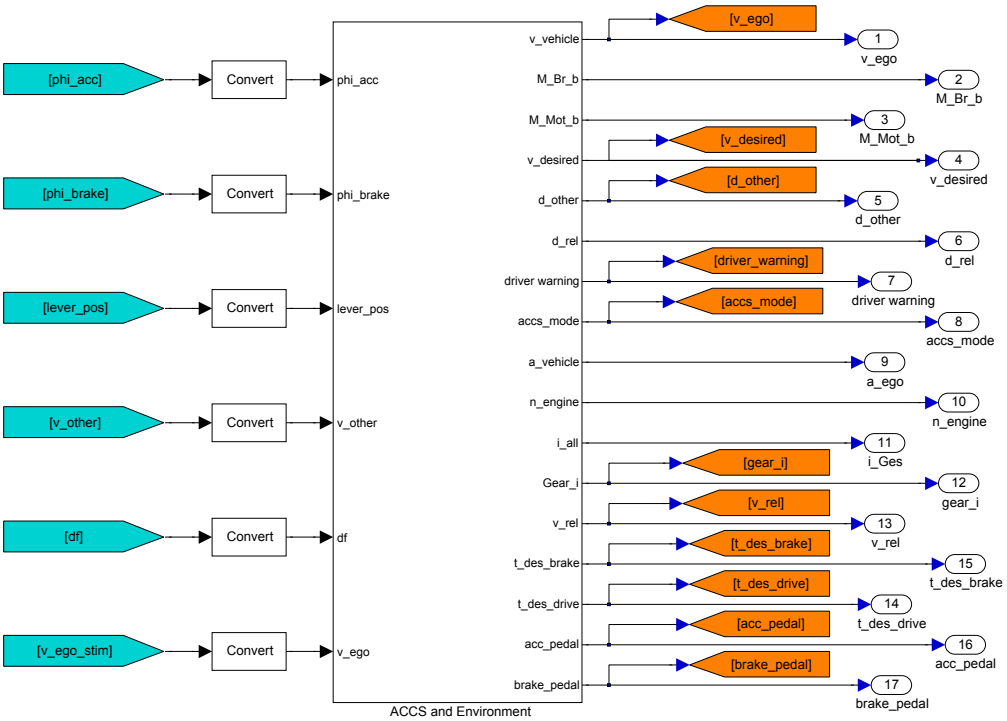

Figure 7.1: Testable interface of the Adaptive Cruise Control

In typical ECU software-testing protocol, feedback from the environment is essential. Usually, it is simulated and, normally, this simulation is defined by so-called environment models that are directly linked with the system during testing.

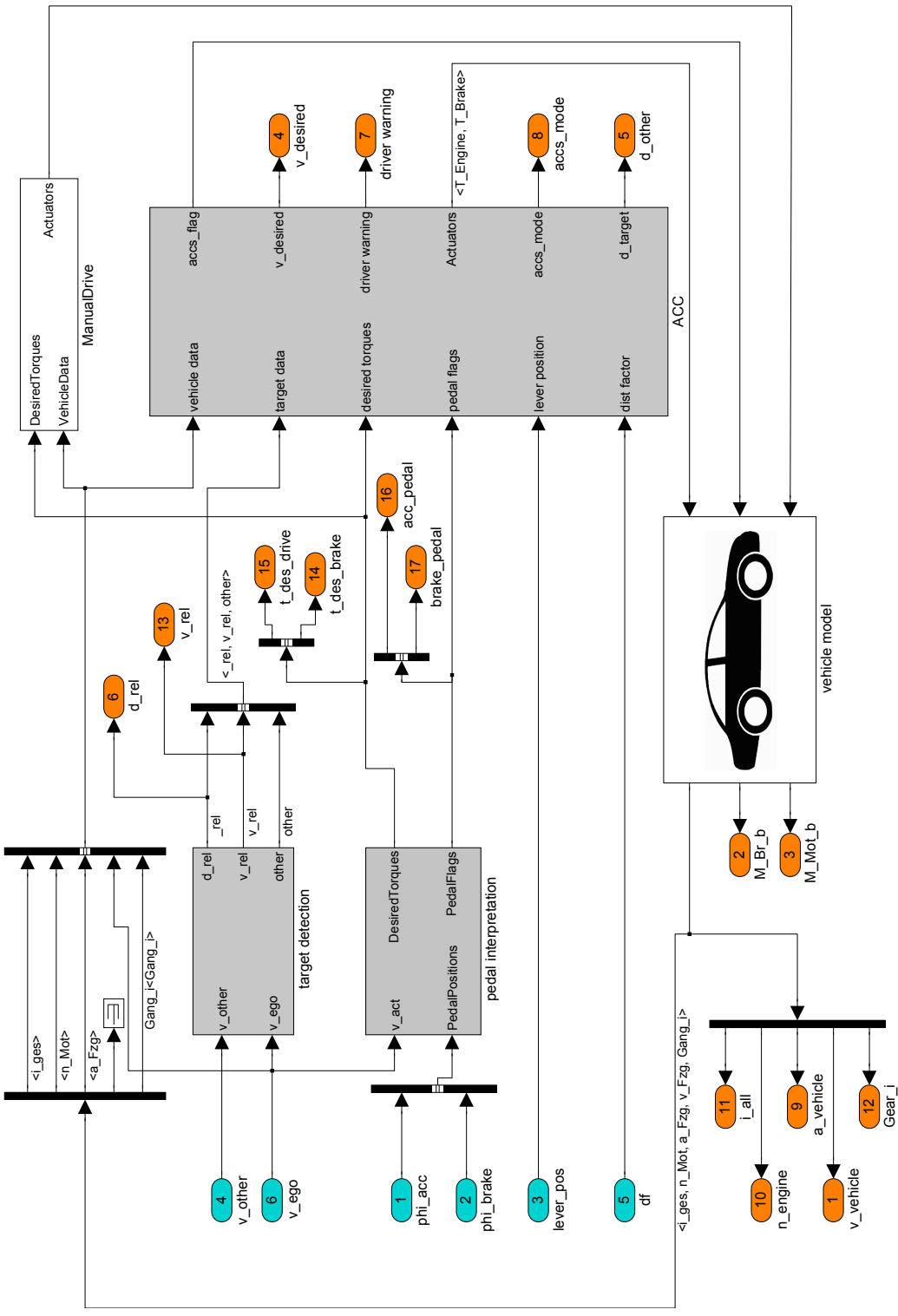

Figure 7.2: Simulink model of the Adaptive Cruise Control and its environment

Figure 7.2 shows the interior design of the ACCS and its environment. The ACCS consists of three subsystems, which, taken together, constitute the overall software functionality. The *ACC-control* unit delivers the main functional behaviour. It is responsible for controlling the running velocity (*v_ego*) of the vehicle, monitoring the distance to the vehicle ahead (*d_other*) and providing a warning signal that informs the driver when the safe distance has been violated (*driver_warning*). The ACC-Control is supplemented by the *target detection* unit. This unit preprocesses sensed information about the velocity of the target vehicle (*v_other*). The third unit, the *pedal interpretation* unit, preprocesses the driver's input (*phi_brake*, *phi_acc*). These three units, taken together, constitute the *vehicle model*, which models the comprehensive behaviour of the vehicle and serves as the environment model for closed-loop simulation and testing.

The following examples show how *TTCN-3 embedded* can be used to describe tests that assess either the complete ACC system, a selected subsystem thereof, or both. The test requirements and high-level specifications in this case study are similar to those currently used in the automotive industry. The *TTCN-3 embedded* specifications are used to unambiguously describe the executable test cases in an efficient and highly reusable fashion.

The case study starts with scenarios that are used to test the pedal interpretation, one of the ACC subsystems. The abstract test specifications that are the basis for that scenarios are taken from Mirko Conrad's thesis [22] and from previous work in the IMMOS project [50]. Thereafter, the case study elaborates tests for the complete ACC system. Both, the subsystem test and the system tests, show clearly how *TTCN-3 embedded* can be used to unambiguously specify MiL tests across various levels of integration.

7.1.2 Testing the pedal interpretation

The subsystem pedal interpretation reads the current positions of the gas pedal and the brake pedal. Taking into account the current speed of the vehicle and the pedal positions, the pedal interpretation subsystem calculates the corresponding input and output torques. In addition, two flags are used to monitor whether one of the pedals is further pressed or not. Figure 7.3 shows the internals of the Matlab/Simulink model. Figure 7.4 and Table 7.1 show the effective test interface as seen in Matlab/Simulink, from the perspective of the tester (i.e. the system inputs are declared as outputs and system outputs as inputs), in *TTCN-3 embedded*.

This first example illustrates tests that check the correct values of the pedal flags. The subsystem pedal interpretation is tested via a set of selected

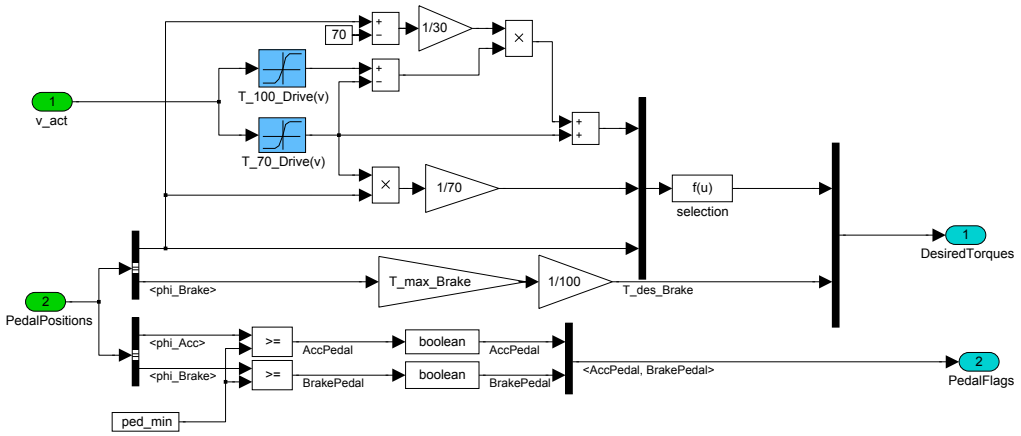

Figure 7.3: The pedal interpretation subsystem

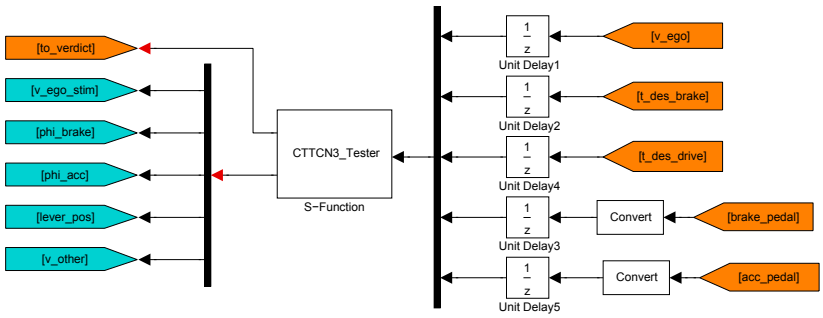

Figure 7.4: The test system perspective of the Adaptive Cruise Control

input vectors which are dedicated to triggering the pedal recognition. The pedal recognition subsystem reads the status of the pedals via the values of the pedal flags. To illustrate the overall test objectives, the Classification Tree Editor (CTE) is used. The tests can be carried out without the environment model because all of the relevant input data are either configured statically, i.e. by parameters in Matlab/Simulink, or are generated by the *TTCN-3 embedded* test program itself.

Figure 7.5 shows a classification tree for expected interface values. It provides a classification over the value domain for the input and output interfaces of the subsystem by providing typical classifications, i.e. for the boundary values and for the mean value of the pedal positions. These classifications are supplemented by an additional partitioning of the input space that allows for the testing of threshold values outside of those which the system already recognizes when a pedal is pressed. The overall partitioning is accomplished

Table 7.1: ACC test interface (pedal interpretation)

symbol	dir	unit	datatype
v_ego_stim	out	m/s	double
phi_brake	out	%	double
phi_acc	out	%	double
brake_pedal	in	-	boolean
acc_pedal	in	m	boolean

via the introduction of the variable *ped_min*. The variable *ped_min* specifies the threshold and divides the overall interval]0.100[for valid input into the areas above and below *ped_min*. Finally, additional classifications have been defined to represent the boundary values 0 and 100 and the exact threshold *ped_min*. The classes for the velocity input *v_ego* are defined including the following considerations: reverse driving (negative velocity), halting, and both slow and fast riding.

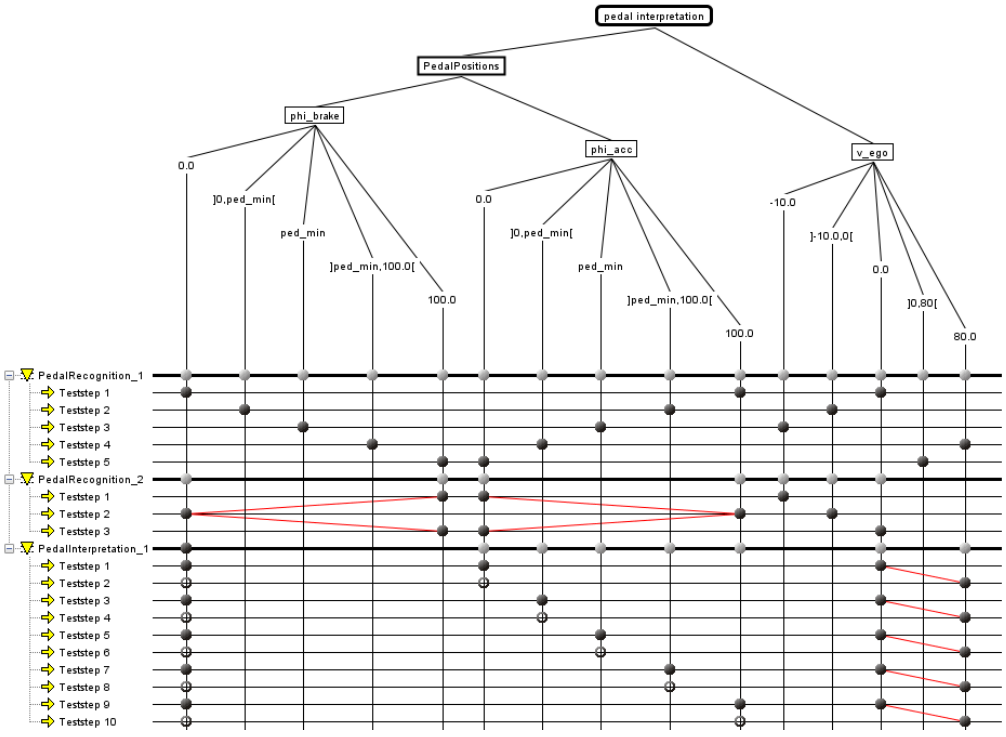

Figure 7.5: The high-level test specification for the pedal interpretation

- The first test scenario (named test sequence *PedalRecognition_1*, see Figure 7.5) assesses the recognition of the pedal input. Characteristic value sets are applied one-by-one to the system for a pre-defined period of time. This period must be significantly larger than the simulation step size and must be kept constant throughout the test. The velocity is varied across different input classes. Each input set is supplemented by the expected outputs for *brake_pedal* and *acc_pedal*.

- A second test scenario (named test sequence *PedalRecognition_2*, see Figure 7.5) analyzes the specific characteristics of the pedal recognition hysteresis. A ramp-shaped signal is introduced across the entire range of possible pedal positions. The velocity *v_ego* is set with selected constant values. The values for *brake_pedal* and *acc_pedal* are expected to change, respectively, from *true* and *false* to *false* and *true*. The actual checking mechanism of the hysteresis is only specified in the *TTCN-3 embedded* test cases.

- A third test scenario (named test sequence *PedalInterpretation_1*, see Figure 7.5) varies in a linear fashion across different accelerator pedal positions from 0.0 to 100.0. The velocity is ramped up in the non-negative speed range from 0.0 to 80.0 ms. The value *phi_acc* is varied across all of the classes, while the other two input values are both stimulated with a constant behaviour (*phi_Brake = 0.0*, *v_act* increases linearly and uniformly). The output value at *brake_pedal* is expected to remain *false* and the value at *acc_pedal* is expected to change according to the input at *phi_acc*.

It should be noted that these three scenarios are only a subset of the possible test cases which could be used to test the subsystem *pedal interpretation*. These specific cases have been chosen, however, because they succinctly display *TTCN-3 embedded*'s range of possibilities. These cases, as the following subsection shows, demonstrate how *TTCN-3 embedded* can be used to develop intuitive concepts for the realization of any given specification and show how such specifications can be converted to directly executable, automated test cases.

The TTCN-3 embedded test (case) specification

The *TTCN-3 embedded* test specification begins with the definition of the data types and the test components. The test interface is not restricted to the input/output vector defined in Figure 7.5; rather, it contains the complete

output interface of the subsystem *pedal interpretation*, such that it can be used to test system properties other than the pedal recognition.

The type definition starts by providing the port types for the data streams. There are Float streams for the Float-valued outputs *phi_brake*, *phi_acc* and for the Float-valued inputs *t_des_brake*, *t_des_drive*. The Boolean-valued *brake_pedal*, *acc_pedal* inputs are modelled as Boolean streams.

Listing 7.1: Definition of the pedal interpretation test component

```
module PedalInterpretation{
  import from signal_generators all;

  type port FloatOut stream {out float};
  type port BoolIn stream {in boolean};

  type component PedIntTester {
    port FloatOut v_ego_stim, phi_acc, phi_brake;
    port BoolIn brake_pedal, acc_pedal;
  }
...
} with {stepsize "0.01"}
```

The component *PedIntTester* specifies the test interface using the port names defined in Figure 7.4. The step-size for the test execution is set to one millisecond and stays constantly thus.

For the resolution of the test data, all variables and ports used in the classification tree are assigned specific values. For all classes, specific constants are created that provide a representation value for each respective class (e.g. the class $]ped\_min, 100.0[$) is represented by the constant *c_phi_4*, which has the value *ped_min + 45.0*).

Listing 7.2: Testing the pedal recognition, constant defintions

```
  // definition of the class values
  const float tol:= 0.1;
  const float ped_min:= 5.0;

  const float c_phi_1:= 0.0;
  const float c_phi_2:= ped_min - 2.5;
  const float c_phi_3:= ped_min;
  const float c_phi_4:= ped_min + 45.0;
  const float c_phi_5:= 100.0;


  const float c_vel_1:= -10.0;
  const float c_vel_2:= - 5.0;
  const float c_vel_3:= 0.0;
  const float c_vel_4:= 50.0;
  const float c_vel_5:= 80.0;
```

The variable *ped_min* is set to 5.0%. The variable *tol* specifies the maximum allowed tolerance (i.e. the hysteresis) between a change of the pedal position and the output of the system.

Test case *PedalRecognition_1*: Each test step from Figure 7.5 is represented in the *TTCN-3 embedded* program by an atomic mode that assigns values to the output ports and also checks the values at the input ports (see Listing 7.6). The atomic modes are executed successively with a duration of two seconds each. The checking of the input ports *brake_pedal* and *acc_pedal* is delayed with respect to the hysteresis.

Listing 7.3: Testing the pedal recognition, test step definitions

```
testcase PedalRecognition_1() runs on PedIntTester{
  setverdict(pass);

  // Teststep 1
  cont{
    phi_brake.value := c_phi_1;
    phi_acc.value:= c_phi_5;
    v_ego_stim.value:= c_vel_3;
    if(duration >= tol ){
      assert(brake_pedal.value == false);
      assert(acc_pedal.value == true);};
  }
  until(duration >= 2.0)

  // Teststep 2
  cont{
    phi_brake.value:= c_phi_2;
    phi_acc.value:= c_phi_4;
    v_ego_stim.value:= c_vel_2;
    if(duration >= tol ){
      assert(brake_pedal.value == false);
      assert(acc_pedal.value == true);};
  }
  until(duration >= 2.0)

  // Teststep 3
  cont{
      phi_brake.value:= c_phi_3;
      phi_acc.value:= c_phi_3;
      v_ego_stim.value:= c_vel_1;
      if(duration >= tol ){
        assert(brake_pedal.value == true);
        assert(acc_pedal.value == true);};
  }
  until(duration >= 2.0)

  // Teststep 4
  cont{
      phi_brake.value:= c_phi_4;
      phi_acc.value:= c_phi_2;
      v_ego_stim.value:= c_vel_1;
      if(duration >= tol ){
        assert(brake_pedal.value == true);
        assert(acc_pedal.value == false);};
  }
  until(duration >= 2.0)

  // Teststep 5
  cont{
      phi_brake.value:= c_phi_5;
      phi_acc.value:= c_phi_1;
```

```
      v_ego_stim.value:= c_vel_1;
      if(duration >= tol ){
        assert(brake_pedal.value == true);
        assert(acc_pedal.value == false);};
  }
  until(duration >= 2.0)
}
```

Refinement option: The test case specification in Listing 7.3 uses redundant code fragments. A more compact test specification can be found in the parameterizable modes which belong to the advanced programming features of *TTCN-3 embedded*. Listing 7.4 shows one such specification, including parameters for all relevant variables.

Listing 7.4: Testing the pedal recognition

```
mode pedRecStep (in float brake,
                 in float acc,
                 in float v,
                 in float tol_,
                 in boolean assert_brake,
                 in boolean assert_acc,
                 in float dur) runs on PedIntTester
  cont{
    phi_brake.value := brake;
    phi_acc.value := acc;
    v_ego_stim.value := v;
    if(duration >= tol_){
      assert(brake_pedal.value == assert_brake);
      assert(acc_pedal.value == assert_acc)
    }
}
until (duration >= dur)
```

The specification of the individual test steps can be reduced when the parameterizable mode specification from Listing 7.4 is used, because it eliminates repeated code. This makes testing more efficient and faster. Listing 7.5 depicts the application of this kind of mode.

Listing 7.5: Testing the pedal recognition, first test case, refinement

```
testcase PedalRecognition_1_refined() runs on PedIntTester{
  setverdict(pass);

  // Teststep 1
  pedRecStep(c_phi_1, c_phi_5, c_vel_3, tol, false, true, 2.0);
  // Teststep 2
  pedRecStep(c_phi_2, c_phi_4, c_vel_2, tol, false, true, 2.0);
  // Teststep 3
  pedRecStep(c_phi_3, c_phi_3, c_vel_1, tol, true, true, 2.0);
  // Teststep 4
  pedRecStep(c_phi_4, c_phi_2, c_vel_1, tol, true, false, 2.0);
  // Teststep 5
  pedRecStep(c_phi_5, c_phi_1, c_vel_1, tol, true, false, 2.0);
}
```

Test case *PedalRecognition_2*: The second test sequence depicted in Figure 7.5 conventionally differs from the first by requiring ramp-shaped input stimuli. The overall setup (i.e. the test component and the constant specification) of the *TTCN-3 embedded* test case is, however, the same as that proposed for the first test sequence. The *TTCN-3*-3 source code is given in Listing 7.6. Again, each test step is represented by an atomic mode. To provide a ramp-shaped output signal, the respective ports are continuously assigned increasing values that are calculated following a simple ramp function (see Listing 7.6, line 5,6 and line 14,15).

Listing 7.6: Testing the pedal recognition, second test case

```
testcase PedalRecognition_2() runs on PedIntTester{
                                                                    2
  setverdict(pass);
  cont {                                                            4
    phi_brake.value:=c_phi_5 - (duration * c_phi_5);
    phi_acc.value:= c_phi_1 + (duration * c_phi_5);                 6
    v_ego_stim.value:= c_vel_1;
    if(phi_brake.value >= ped_min){                                 8
      assert(brake_pedal.value == true)
    }                                                               10
    if(phi_acc.value >= ped_min){
      assert(brake_pedal.value == true)                             12
    };
  }                                                                 14
  until(duration >= 1.0)
                                                                    16
  cont{
    phi_brake.value:= c_phi_1 + (duration * c_phi_5);               18
    phi_acc.value:= c_phi_5 - (duration *c_phi_5);
    v_ego_stim.value:= c_vel_1;                                     20
    if(phi_brake.value >= ped_min){
      assert(brake_pedal.value == true)                             22
    };
    if(phi_acc.value >= ped_min){                                   24
       assert(brake_pedal.value == true)
    }                                                               26
  }
  until(duration >= 1.0)                                            28
}
```

Refinement option: The use of libraries with predefined modes and functions allow for even further simplification of the test specification process. These predefined modes and functions can provide parmeterizeable signal generators for commonly used signal shapes, such as ramps, square waves, sine waves, saw-tooth waves, etc. Libraries in *TTCN-3* are usually provided as separate modules that can be imported into the main testing module. Listing 7.7 shows a possible definition of one such library of modes and functions.

The *ramp* function provides the values for a ramp with a range of *0.0* to *dur*. The ramp is normalized to *1.0* and has a slope of *1/dur*, a figure which also represents the major length of the ramp in seconds.

Listing 7.7: Library with signal generators

```
module signal_generators {
  type port FloatOut message {out float};

  function ramp(in float dur, in float actual_t) return float
  {
    if(actual_t > dur or actual_t < 0.0) {return 0.0;}
    else {return actual_t * (1.0/dur);}
  }

  mode apply_ramp(in FloatOut p,
                  in float dur,
                  in float startval,
                  in float amplitude,
                  in float actual_t,
                  in integer repetitions)
       cont {
         inv{repetitions > 0}
         p.value:= startval + (ramp(dur,actual_t) * amplitude);
       }
       until{
         [duration >= dur] {
           repetitions := repetitions - 1;
           if(repetitions > 0){repeat}
         }
       }
  ...
}
```

This parameterizable mode *apply_ramp* applies a ramp with a given number of seconds (parameter *dur*) and a given amplitude (parameter *amplitude*) to a given port (parameter *p*). The signal starts at a given start value (parameter *startval*), can be repeated randomly and is controlled via the parameter *repetitions*. Listing 7.8 redefines the previous test case *pedalInterpretation_2* (see Listing 7.6) by applying this parameterizable mode.

Listing 7.8: Testing the pedal recognition, second test case, refinement

```
testcase PedalRecognition_2_refined() runs on PedIntTester{

  setverdict(pass);
  // test sequence 1
  par{
    apply_ramp(phi_brake, 1.0, c_phi_5,
               c_phi_1 - c_phi_5, duration, 1);
    apply_ramp(phi_acc, 1.0, c_phi_1,
               c_phi_5 - c_phi_1, duration, 1);
    cont{
      v_ego_stim.value:= c_vel_1;
      if(phi_brake.value >= ped_min){
        assert(brake_pedal.value == true)
      };
      if(phi_acc.value >= ped_min ){
        assert(brake_pedal.value == true)
```

```
      };
    }
  }
  until(duration >= 1.0)
  // test sequence 2
  par{
    apply_ramp(phi_brake, 1.0, c_phi_1,
               c_phi_5 - c_phi_1, duration, 1);
    apply_ramp(phi_acc, 1.0, c_phi_5,
               c_phi_1 - c_phi_5, duration, 1);
    cont{
      v_ego_stim.value:= c_vel_1;
      if(phi_brake.value >= ped_min){
         assert(brake_pedal.value == true)
      };
      if(phi_acc.value >= ped_min){
         assert(brake_pedal.value == true)
      };
    }
  }
  until(duration >= 1.0)}
```

Test case *PedalRecognition_3*: The third test sequence from Figure 7.5 uses parameterizable modes from the outset. Listing 7.9 shows two definitions and demonstrates how parameterizable modes can be combined. The parameterizable mode definition *flag_assertions* describes the assertions for the pedal flags *brake_pedal* and *acc_pedal*. The parameterizable mode *pedRec3Step* is comprised of stimuli and assertions. It contains both a parameter that sets the values for the *phi_brake* port and a mode parameter.

Listing 7.9: Testing the pedal recognition, third test case, parameterizable modes

```
mode flag_assertions(in boolean b_pedal, in boolean a_pedal)
runs on PedIntTester
cont{
  if(duration >= 0.1 ){
    assert(brake_pedal.value == b_pedal);
    assert(acc_pedal.value == a_pedal);
  }
}

mode pedRec3Step(in float ac_acc,
                 in mode assertions)
runs on PedIntTester
par{
  onentry{
    phi_acc.value:= ac_acc;
  }
  assertions;
  apply_ramp(v_ego_stim, 2.0, c_vel_3,
             c_vel_5 - c_vel_3, duration, 1);
} until (duration >= 2.0)
```

The mode parameter is used to flexibly add individual assertions, in order to evaluate each test step in a focused and specific way. This kind of flexibility might also be used to provide different versions of a particular test

case by means of different kind of assertions. Both parameterizable modes use the ***runs on*** clause, which enables them to directly access the ports of *PedIntTester*.

Listing 7.10 shows the application of the predefined modes in the performance of the third test in the Figure 7.5 sequence. The test case starts by setting the verdict and the value for *phi_brake*. The value for *phi_brake* remains constant, so that the port need not be adjusted further. The test steps are then subsequently defined. Each step is parameterized with the actual value for *phi_acc* and the respective assertions for the pedal flags.

Listing 7.10: Testing the pedal recognition, third test case

```
testcase pedalRecognition_3() runs on PedIntTester{
                                                                    2
  setverdict(pass);
  phi_brake.value:= c_phi_1;                                        4

  pedRec3Step(c_phi_1, flag_assertions(false,false));               6
  pedRec3Step(c_phi_2, flag_assertions(false,false));
  pedRec3Step(c_phi_3, flag_assertions(true,false));                8
  pedRec3Step(c_phi_4, flag_assertions(true,false));
  pedRec3Step(c_phi_5, flag_assertions(true,false));               10
}
```

Results of the test run

The *TTCN-3 embedded* test specifications given in the listings above can be compiled and executed in principle. The execution is driven by a Matlab/Simulink simulation run. The executable code has been compiled via a prototypic *TTCN-3 embedded* compiler and integrated into a Simulink S-Function via the Simulink adapter discussed in Section 6.4. The tests were executed successively and were not performed in real time, but rather in a simulation time generated by the Matlab/Simulink simulation solver.

Figure 7.6 shows the testing results recorded during the entire simulation run. The recordings of the first 10 seconds belong to the test case *PedelRecognition_1*, the recordings of the following 2 seconds (from second 10 to second 12) belong to the test case *PedalRecognition_2* and the recordings at the end of the test log (from second 12 to 20) belong to the test case *PedalRecognition_3*.

The signals 4 : *phi_acc*, 3 : *phi_brake* and *v_ego_stim* are generated by the *TTCN-3 embedded* test system. These signals drive the test run and try to provoke an unspecified system reaction. The signals 1 : *acc_pedal* and 0 : *brake_pedal* represent the reaction of the system which is used for the automatic generation of verdicts. These two signals are continuously checked during each test run and across all of the three test cases. If one of the values

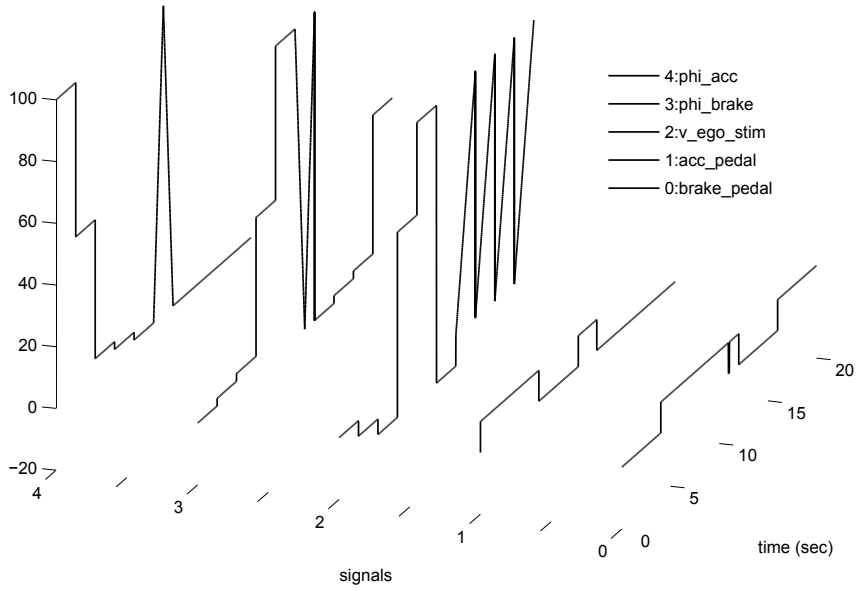

Figure 7.6: Signal logs of the pedal interpretation test

changes to an unexpected value, the system sets the overall verdict of that test case to $fail == 0.0$. Otherwise, the verdict value remains $pass == 5$. The tests are executed with different configurations for the values *tol* and *ped_min*. While a change of *ped_min* within the range of [1.0..10.0] did not lead to an error, a small *tol* value $tol < 0.05$ in the first test case sometimes provoked errors.

7.1.3 Testing the complete ACC system

To test the complete ACC system, a scenario-oriented paradigm is used. The scenarios mimic common driving situations and are applied to the system using a slightly different test interface than that which is used in the previous example. Unlike the pedal interpretation test cases, the following test cases need the input of an environment model that simulates the overall behaviour of the vehicle's engine (see Figure 7.2). The test interface is specified in Figure 7.7 and in Table 7.2. The table provides a test-system-centric perspective of

the test interface: system inputs are declared as outputs and system outputs as inputs.

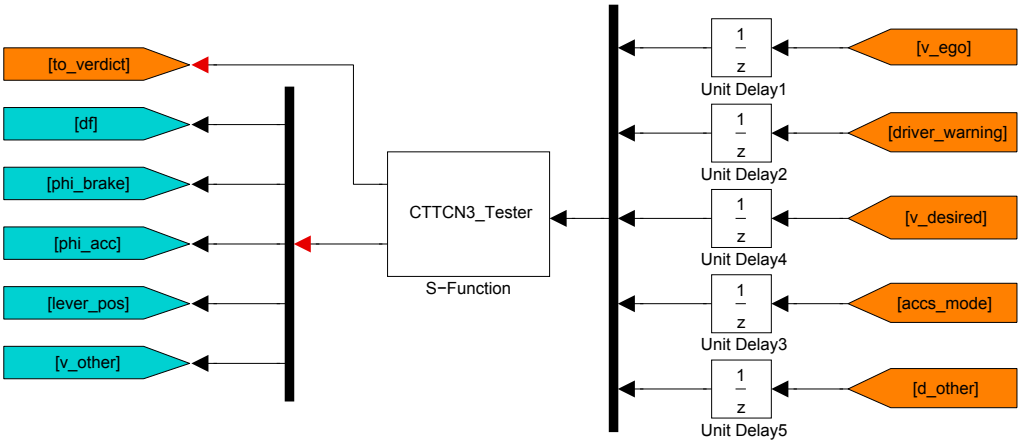

Figure 7.7: The test system perspective of the Adaptive Cruise Control

Table 7.2: ACC test interface (system)

symbol	dir	unit	datatype
df	out	-	double
phi_brake	out	%	double
phi_acc	out	%	double
lever_pos	out	-	enumeration
v_other	out	m/s	double
v_ego	in	m/s	double
driver_warning	in	-	boolean
v_desired	in	m/s	double
accs_mode	in	-	enumeration
d_other	in	m	double

The complete test setup is depicted in Figure 7.2 and Figure 7.7. It consists of the ACC system, an environment model and the test system. The test is conducted as a closed-loop test. This means that the system output is played back into the system from which it originates via the vehicle model and test system. The environment model provides the overall behaviour of the vehicle's engine. The test system is responsible for both for the stimulation of the system and the evaluation of its output.

Exemplary test purposes: Changeover velocity control mode and distance control mode

In this first scenario, the changeover between velocity control mode and distance control mode is tested. Velocity control mode is a simple control mode which does not include distance control. This mode is used when no other vehicle is directly ahead. Distance control mode, on the other hand, provides active distance control. It is activated automatically when there is a slower vehicle directly in front of the car. A correct changeover between velocity control and distance control shows that the ACCS is able to detect slower vehicles correctly and thus one of the major indicators that the system is functioning correctly. The corresponding scenario specification is given below.

Test scenario 1: Switch to distance control mode

1. **Init:** Introduce a vehicle ahead by setting the velocity of the vehicle ahead to *v_other:= 25* m/s. The initial distance *d_init:=90* m is set before the test execution using the parameter interface in Matlab/Simulink. Accelerate the vehicle *phi_acc:=80* until the velocity *v_ego* rises to more than 35 m/s. Then activate the cruise control by setting the *leverpos:=HOLD_ACC*.

2. **Activate Velocity Control Mode:** The ACC shall switch to the velocity control mode and the vehicle shall hold the current velocity with a tolerance of 10 per cent. After a few seconds, the vehicle reaches the safe distance to the vehicle ahead. The safe distance to the vehicle ahead can be calculated with *v_acc/2∗df*. The symbol *df* represents a distance factor that is set to the value of *2.0* here.

3. **Activate Distance Control Mode:** When the safe distance is reached, the ACC switches to distance control mode. The ACC should now adjust the running velocity according to the vehicle ahead with a tolerance of *10%* here. The safe distance to the vehicle ahead must be guaranteed.

In a second scenario, the changeover between distance control mode — when there is a slow vehicle just in front— and back to velocity control mode — when the vehicle directly ahead accelerates— is tested. This changeover is one of the more complex tasks of an ACC and can be evaluated using the following test behaviour. In this second scenario, the original scenario is supplemented with an additional step.

Test scenario 2: Back to velocity control mode

1. **Init:** see similar step in test scenario 1
2. **Activate Velocity Control Mode:** see similar step in test scenario 1
3. **Activate Distance Control Mode:** see similar step in test scenario 1
4. **Accelerate Target:** To test whether the ACC switches back to velocity control mode, we accelerate the target vehicle using a smooth ramp. The ACC should now adjust the velocity of the ego vehicle in response to the behaviour of the the target vehicle (whose velocity must exceed *40* m/s). For this assessment, a velocity tolerance of *10%* can be allowed. Once the target vehicle's velocity is assessed and deemed sufficient, the ACC should switch back to velocity control mode.

The TTCN-3 embedded test (case) specification

The *TTCN-3 embedded*test specification starts with the definition of the data, port and test component types. The data type definition sets the figures which depict both the states of the lever and the different modes of the ACCS. The port type definitions articulate the port types that will be used for the data streams. There are Float streams for the Float-valued outputs (like the dist factor (*df*), the brake pedal (*phi_brake*), the acceleration pedal (*phi_acc*), and the velocity of the vehicle ahead (*v_other*)). There is also another data output stream, known as *lever_pos*, which communicates the position of the lever and is used to appropriately adjust the position of the lever. The Float-valued inputs *v_ego*, *v_desired*, and *d_desired* are modeled as Float streams, the Boolean-valued input (*driver_warning* as a Booelan stream and the current ACCS mode *accs_mode*) as an ACCSMode stream.

Listing 7.11: Definition of the ACCS tester component

```
module ACCSystemTest{
  type enumerated LeverState { MIDDLE, HOLD_ACC, HOLD_DEC, OFF };
  type enumerated ACCSMode { PASSIVE, PRESSED, VCM, DCM};

  type port FloatOut stream {out float};
  type port LeverOut stream {out LeverState};
  type port FloatIn stream {in float};
  type port BoolIn stream {in boolean};
  type port ACCSModeIn stream {in ACCSMode};

  type component ACCSTester {
    port FloatOut df;
    port FloatOut phi_brake;
    port FloatOut phi_acc;
    port LeverOut lever_pos;
    port FloatOut v_other;
```

```
    port FloatIn v_ego;                                          18
    port BoolIn driver_warning;
    port FloatIn v_desired;                                      20
    port ACCSModeIn accs_mode;
    port FloatIn d_other;                                        22
  }
                                                                 24
...
}with {stepsize "0.1"}                                           26
```

The test cases themselves begin with the setting of the initial verdict and the initial values at ports. In the first test case in Listing 7.12, the distance factor *df* is set to 2.0 and the lever is set to the position "OFF". Additionally, the velocity for the vehicle ahead is set to 25 ms. In Lines 10 to 18, the velocity is steadily accelerated until it reaches 35 ms. At this point, the lever is set to the middle position, thereby activating the ACCS.

Listing 7.12: Testing the changeover from velocity to distance control mode

```
testcase Switch_to_distance_control_1() runs on ACCSTester{
                                                                 2
  setverdict(pass);
  df.value:= 2.0;                                                4
  lever_pos.value:= OFF;
  v_other.value:= 25.0;                                          6
  phi_brake.value:= 0.0;
                                                                 8
  // accelerate vehicle until 35 ms and activate ACCS
  cont{                                                          10
    phi_acc.value:=80.0;
  }                                                              12
  until{
    [v_ego.value > 35.0] {                                       14
      phi_acc.value:=0.0;
      lever_pos.value:= MIDDLE;                                  16
    }
  }                                                              18

  // wait for several seconds                                    20
  wait(now + 10.0) ;
                                                                 22
 // check if the velocity has not increased
 // and if the distance control mode has been activated          24
  cont{
    onentry{assert(accs_mode.value == DCM)}                      26
    assert(v_ego.value <= 38.0);
  }                                                              28
  until{
    [duration > 10.0] {}                                         30
  }
}                                                                32
```

After the ACCS has been activated, the test case waits for 10 seconds. For the next 10 seconds, the test case checks to see if the velocity of the primary vehicle has not, in fact, increased (line 27) and if the distance control mode has rather been activated due to the detection of a vehicle ahead (line 28).

Figure 7.8 shows the data recorded during the execution of the test case. Signal 3 : *v_other* shows the velocity of the vehicle ahead (which is set at the beginning of the test case). The acceleration phase (seconds 0 to 25) is defined by the signals 1 : *phi_brake* and 4 : *v_ego*. 1 : *phi_brake* shows that the gas pedal has been pressed and 4 : *v_ego* shows the increase in velocity. Afterwards, the ACCS adjusts the velocity of the ego vehicle in response to the target vehicle.

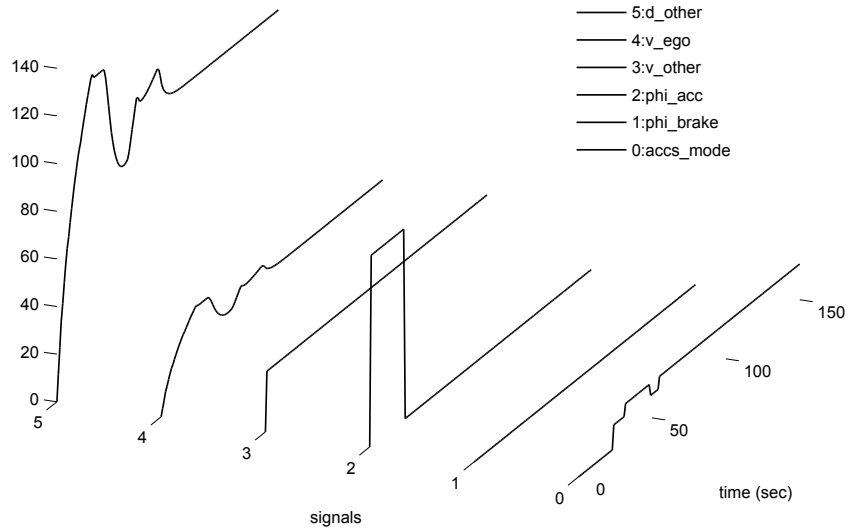

Figure 7.8: Execution of *Switch_to_distance_control_1*

The signal 0 : *accs_mode* shows the settings of the ACCS mode. It should be noted that the original value range is 0 to 2. To make the signal changes visible in the signal log, the values must be scaled up by a factor of 5. The signal, however, shows that the ACCS mode changes from passive mode to velocity control mode when the ACCS is activated. After a while, when the vehicle has reached the (pre-defined) critical distance to the vehicle ahead, the distance control mode is activated.

The second test case is, in fact, a simple extension of the first. The beginning of the scenario is completely unchanged; Listing 7.13 shows its implementation. A vehicle ahead is introduced with a fixed velocity of 25 ms, the ego vehicle is accelerated and thus the ACCS switches into distance con-

trol mode. Afterwards, however, the vehicle ahead is accelerated gradually by ramping up *v_other* in line 33 of Listing.

Listing 7.13: Testing the change back from distance to velocity control mode

```
testcase Back_to_velocity_control_1() runs on ACCSTester{
                                                                    2
  setverdict(pass);
  df.value:= 2.0;                                                   4
  v_other.value:= 25.0;
  phi_brake.value:= 0.0;                                            6
  lever_pos.value:= OFF;
                                                                    8
  // accelerate vehicle until 35 ms and activate ACCS
  cont{                                                             10
    phi_acc.value:=80.0;
  }                                                                 12
  until{
    [v_ego.value > 35.0] {                                          14
      phi_acc.value:= 0.0;
      lever_pos.value:= MIDDLE;                                     16
    }
  }                                                                 18

  // wait for several seconds                                       20
  wait(now + 10.0);
                                                                    22
  // check if the velocity has not increased
  // and if the distance control mode has been activated            24
  cont{
    onentry{assert(accs_mode.value == DCM)}                         26
    assert(v_ego.value <= 38.0);
  }                                                                 28
  until (duration > 50.0)
                                                                    30
  // accelerates the the vehicle ahead
  // and checks whether ACCS switches back to velocity control mode 32
  cont{
    v_other.value:= 25.0 + (duration * 0.5) ;                       34
  }
  until{                                                            36
    [duration > 50.0] {setverdict(fail)}
    [accs_mode.value == VCM] {setverdict(pass)}                     38
  }
}                                                                   40
```

During the acceleration of the target vehicle, the test case checks if the ego vehicle switches properly back from distance control mode into velocity control mode. If this indeed happens during the subsequent 50 seconds, the test case ends with a verdict of success (a pass); otherwise, it ends with a verdict of failure (a fail).

Similarly to Figure 7.8, Figure 7.9 shows the recordings of the signals. Signal 3 : *v_other* shows the gradual acceleration of the vehicle ahead about 70 seconds after its initial appearance. The signal 5 : *d_other* shows that the distance to the target vehicle ahead is increasing as the target accelerates and the signal *v_ego* shows that the ego vehicle starts to accelerate concomitantly.

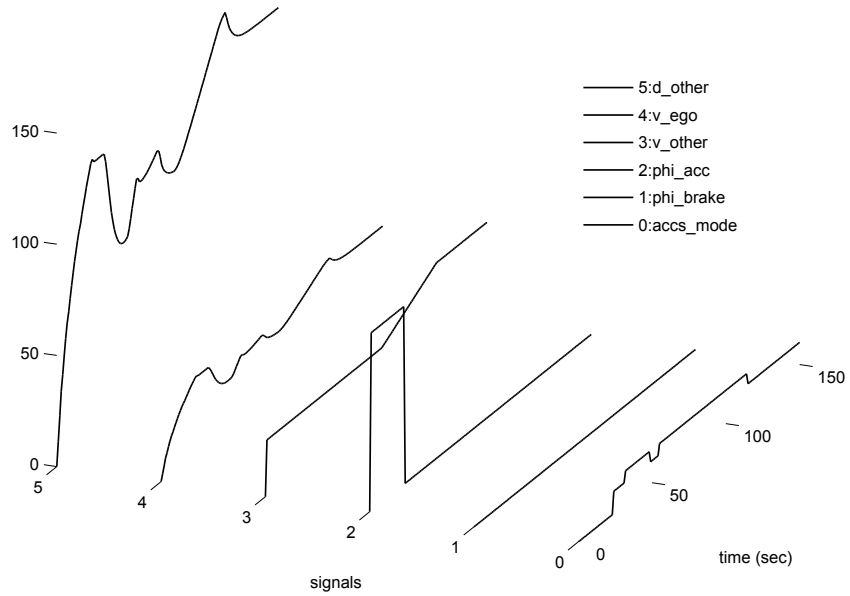

Figure 7.9: Execution of *Back_to_velocity_control_1*

Finally, after about 100 seconds, signal 0 : *accs_mode* shows that the ACCS has returned to velocity control mode and, ergo, the vehicle returns to its former fixed velocity of 35 ms (signal 4 : *v_ego*).

The test results

The ACCS test cases above belong to a test suite for which tests have been executed with multiple different settings: *df*, *v_other*, and *phi_acc*. It could be shown that, if the difference between the velocities is high (i.e. *v_ego* is high and *v_other* is quite low), the ACCS needs time to properly adjust the velocity of the ego vehicle. This leads sometimes to overshots, wherein the ACCS switches between velocity control mode and distance control mode several times. This behaviour is not problematic so long as the change of the velocity in the ego vehicle remains smooth. The information from the signals in Figures 7.8 and 7.9 show this kind of behaviour after about 50 seconds, when the signal 0 : *accs_mode* suddenly changes from distance control mode to velocity control mode, and then back again.

7.2 The HiL case study

The HiL case study is executed using a Vector CANoe test environment that has been extended to integrate *TTCN-3 embedded* test executables. The SUT is a window-lifting ECU and is part of an Automotive Application Evaluation System (AAES). This case study has been designed to show that the concepts already used for testing simulations could be also used in an environment with real-time requirements.

7.2.1 Automotive Application Evaluation System (AAES)

The Automotive Application Evaluation System (AAES) is used to develop and test software for automotive ECUs, under realistic conditions. The AAES is made up mostly of components which are used in the automotive industry. The softwares of the individual vehicle functions are designed and implemented as a distributed application. The components are connected by a typical vehicle CAN network, which can be controlled by CANoe software. The Figure 7.10 shows the AAES.

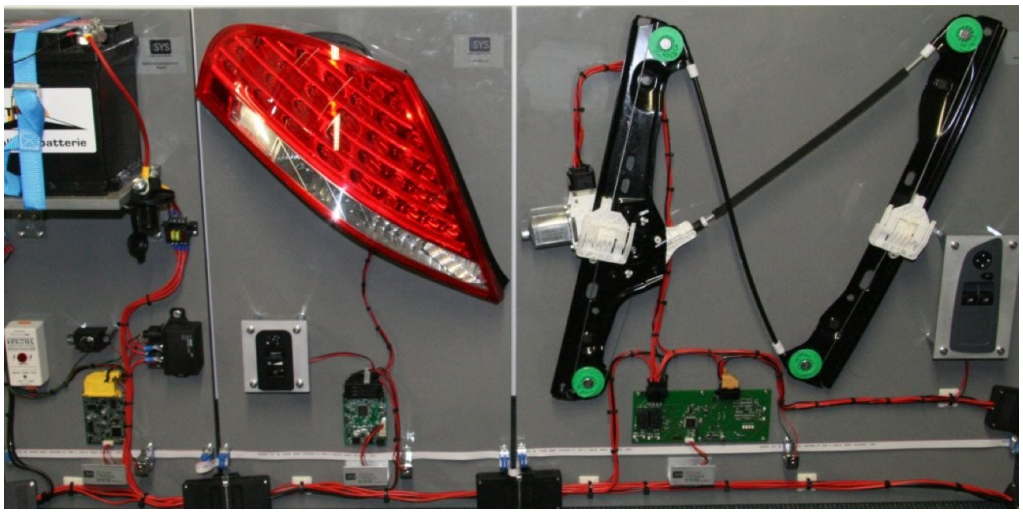

Figure 7.10: The Automotive Application Evaluation System (AAES)

The window-lifter module consists of a control unit with actuators that drive the motor of the window lifter. The keypad can be used to control the lifter and a hall sensor is used to control the power of the window-lifting motor.

7.2.2 Testing the window lifter module

In the following case study, the AAES window-lifter module is tested under realistic conditions. The window lifter can be controlled by driver commands and by passenger commands, both of which are normally applied to the window-lifter module by means of a keypad. The commands are: up, down or neutral. When a driver or a passenger holds the up command, the window position increases until the window is closed. When a driver or a passenger holds the down command, the window position decreases until the window is completely open. The window controller has an obstacle detection mechanism: that is to say, when an obstacle is introduced as the window is closing, the obstacle is detected and the closing process stops immediately.

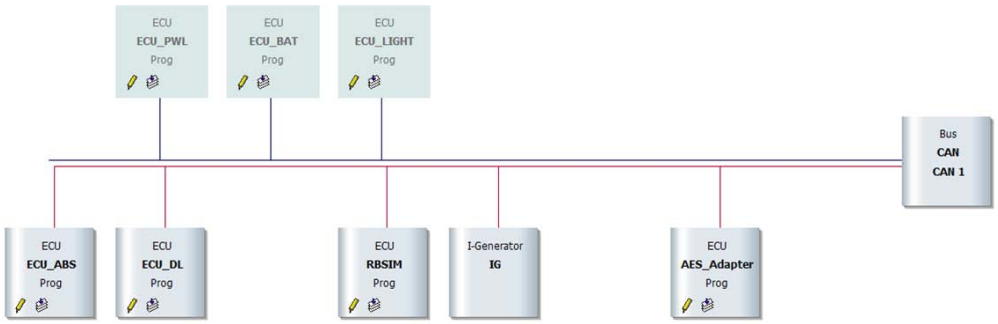

Figure 7.11: Setup of the window lifter module in the Vector CANoe environment

For testing the window-lifter module, the *TTCN-3 embedded* test executable has been integrated into the Vector CANoe environment, as depicted in Section 6.5. The inputs and outputs of the test executable are adapted to the respective environment variables in CANoe which, by themselves, control the CAN bus interaction with the SUT.

Table 7.3: The test interface of the AAES window lifter module

symbol	dir	unit	datatype
PWLMoveState	out	-	enumerated
UV_Crash	out	-	enumerated
Window_Position	in	-	double
Window_Current	in	-	double

The test cases simulate user interaction by setting the values *idle*, *auto_up*, *up*, *down*, *auto_down*, or *unknown* at *PWLMoveState*. When these values change, the CANoe environment is triggered to generate CAN messages, which are then sent on the CAN bus and transmitted to the window lifter ECU.

The test cases performed upon the AAES system can be used as simple regression tests to evaluate the basic behaviour of the window-lifter module. They are designed to correspond to the level of overall user interaction. This is helpful principally because it allows the application of the tests in MiL and HiL environments without the trouble of adding extra test interfaces. The following scenario depicts the level of abstraction used in the test design.

- The driver presses and holds the down button. The window opens (i.e. the position value increases).
- When the window position reaches the down position (value <= -1000) or the test has exceeded its maximum duration, the test ends.
- It is checked if the window has reached the down position after a minimum of 6 seconds.

All *TTCN-3 embedded* scenarios are based on a common test interface specification, which is depicted in 7.14.

Listing 7.14: Window lifter test (test interface)

```
type enumerated MoveState {idle, auto_up, up, down, auto_down, unknown};
type enumerated CrashState {OK, Crash1, Crash2};

type port StreamIn stream {in float};
type port StreamPWLOut stream {out MoveState};
type port StreamCrashOut stream {out CrashState};

type component Tester {
  port StreamPWLOut PWLMoveState;
  port StreamCrashOut UV_Crash;
  port StreamIn Window_Current;
  port StreamIn Window_Position;
}

function setUp() runs on Tester{
  PWLMoveState.value:= idle;
  UV_Crash.value:= OK;

  cont{
    PWLMoveState.value:= up;
  }until{
    [Window_Position.value >= 0.0]{PWLMoveState.value:= idle}
  }
  setverdict(pass);
}

function tearDown() runs on Tester{
    PWLMoveState.value:= idle;
```

```
    UV_Crash.value:= OK;
}                                                                    30
```

Furthermore, the general set-up and tear-down functions are defined to contain actions which reset the AAES window-lifter module. The set-up function ensures that the window is indeed fully closed when the test starts. The tear-down function is simply used to turn off every action that triggers an actuator, so that the hardware stops moving altogether. Listing 7.15 shows the *TTCN-3 embedded* implementation of the scenario defined above.

Listing 7.15: Window lifter test (down mode)

```
// tests the window down function of the window lifter
testcase test_down_mode() runs on Tester {                           2
  setUp();
  cont{                                                              4
    PWLMoveState.value:= down;
    assert(duration <= 6.0);                                         6
  }until{
    [Window_Position.value < - 1000.0]{}                             8
    [duration > 10.0]{setverdict(fail)}
   }                                                                 10
   tearDown();
}//testcase                                                         12
```

Other test cases check not only if the window has reached a certain position, but also check if it continuously change position while the window-lifter is activated. The test case in Listing 7.16 tests if the window reaches its half-open position after 3 seconds and if the window position decreases during the test run; this is determined by comparing the previous values and with the extant values (see line 6).

Listing 7.16: Window lifter test (open half)

```
testcase test_open_half() runs on Tester {
  setUp();                                                           2
  cont{
    PWLMoveState.value:= down;                                       4
    assert(duration <= 3.0);
    assert(Window_Position.prev.value > Window_Position.value);      6
  }until(Window_Position.value < - 500.0)
  tearDown();                                                        8
}//testcase
```

Figure 7.12 shows the log window of the CANoe environment. The *TTCN-3 embedded* logs and the verdict results are transferred to this window and thereby shown to the tester. Figure 7.13 shows the signals for the window position and the window current as they are presented to the tester.

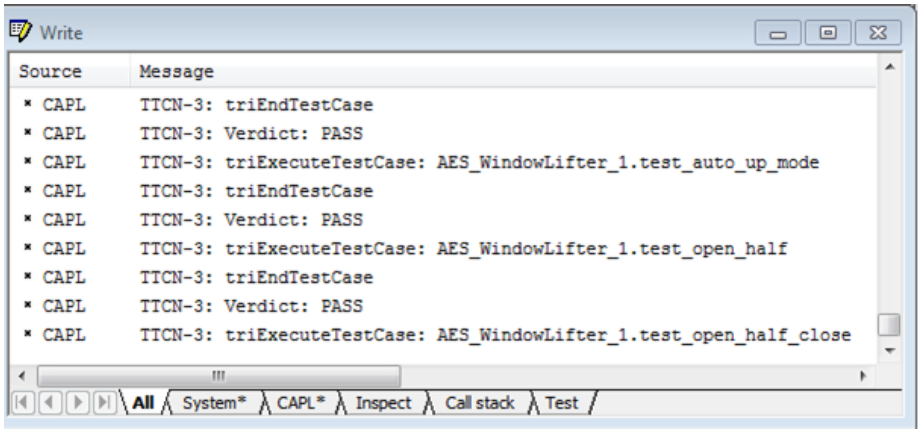

Figure 7.12: The results of the window lifter application

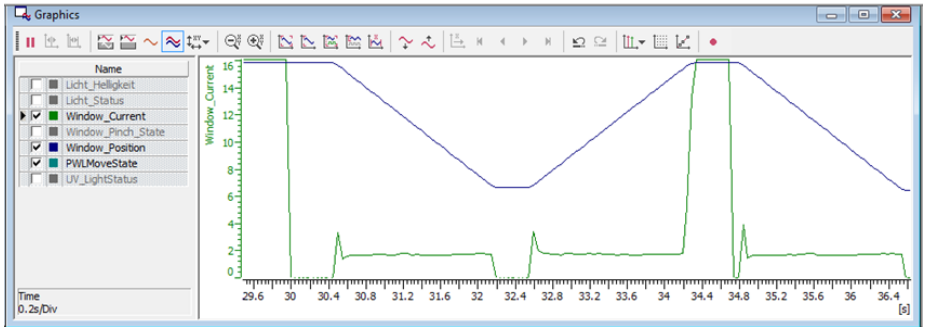

Figure 7.13: The outputs of the window lifter application

7.3 Discussion

The two case studies documented above show that *TTCN-3 embedded* is suited for the testing of continuous systems in a MiL simulation, as well as for the testing of real controller hardware in a HiL simulation. The MiL study, in particular, shows that *TTCN-3 embedded* is specifically appropriate and useful for the stimulation and analysis of continuous systems. Furthermore, it is demonstrated that the language constructs are powerful enough to allow the simplification of test descriptions and the reuse of test fragments in other contexts. The examples from the case studies mainly address the capabilities of the new language constructs and, therefore, show only a subset of the overall capabilities of *TTCN-3*. Thus, in combination with the concept of parameters in *TTCN-3* (e.g. parameterized test cases, parameterized functions and parameterized templates), *TTCN-3 embedded* allows for both combina-

tion of and reuse of test fragments and includes the variability required for such manoeuvres. It has additionally been shown that *TTCN-3 embedded* can be used to test individual components and subsystems, such as the pedal interpretation, and to apply complex driving scenarios that test an application in the context of its integration with the overall vehicle. In the latter case, the *TTCN-3 embedded* tester is used as integral part of the simulation that generates or perpetuates outgoing signals and assesses incoming signals.

The HiL case study shows that *TTCN-3 embedded* language constructs are suitable for the testing of time-dependent processes in a real-time environment. Since the window-lifter application only has soft real-time requirements and the execution in Vector CANoe is also limited to soft real-time requirements, a proof of concepts for harder real-time requirements remains pending.

Apart from the two case studies documented here, a number of additional case studies have been carried out. These studies show other relevant aspects of the language. In the course of the project TEMEA, it was shown that *TTCN-3 embedded* test cases can be reused between MiL, SiL and HiL simulations. The case study was based on tools from dSpace. A thesis written in cooperation with the Technical University of Aachen and the ESG also showed how *TTCN-3 embedded* could be effectively integrated in the testing processes of a large OEM [83].

Chapter 8

Conclusion and research prospects

This thesis has specified and designed a uniform test-specification technology for embedded real-time systems in the automotive industry. The technology addresses the various aspects of automotive components and control systems and allows for the testing of: discrete behaviour for communication; continuous behaviour for the interaction with physical processes; and hybrid behaviour for the technical control of the physical processes and interaction between the environment, other components and the user. This technology facilitates testing and assessment of the functionality, robustness, performance and scalability of automotive components and their configuration. The main results of this thesis, the language *TTCN-3 embedded*, is technically based on the standardized test-specification language *TTCN-3*. *TTCN-3* has been selected as a foundation language because it already provides dedicated testing instruments that can test message-based and procedure-based systems. Furthermore, it is standardized within the industry and is thus already in broad use. *TTCN-3 embedded* is a systematic extension of *TTCN-3* that seamlessly aligns new concepts for testing continuous and hybrid systems with the existing *TTCN-3* approach and tools.

The thesis began by describing the current state of embedded systems testing in the automotive industry and identifying industrial and academic approaches towards the testing of hybrid and continuous systems. Based on this, a set of dedicated concepts for testing continuous and hybrid systems were identified and deduced in Chapter 3. The syntactic and semantic integration of the newly introduced concepts with *TTCN-3* followed in Chapter 4 and Chapter 5. Chapter 6 was concerned with the integration of the new language constructs, their communication-related requirements within the overall *TTCN-3* test architecture, and the integration of *TTCN-3 embedded* into test and simulation environments typically found in the automotive industry. Finally, Chapter 7 showed the applicability of the concepts to two automotive case studies.

In addition to the results of this thesis, the project TEMEA has developed a methodology that systematically integrates *TTCN-3 embedded* into the research and development processes of the automotive industry. This

methodology specifically supports an objective, implementation- and device-independent and technically detailed definition of test requirements and test procedures. The methodology also specifically addresses development processes and technologies that are already established in the automotive industry, thus creating a common platform that is interesting for industrial applications. Especially by addressing *TTCN-3 embedded*'s integration with the tools Mathworks Matlab/Simulink, Vector CANoe, and dSPACE Automation Desk, this thesis and TEMEA have shown that *TTCN-3 embedded* can smoothly integrate into existing automotive tool chains. Furthermore, *TTCN-3 embedded* was standardized by the ETSI in 2012 [38].

8.1 The definition of *TTCN-3 embedded*

TTCN-3 embedded is a test-specification language that has been specifically tailored to meet the requirements of the automotive industry and to integrate with various testing protocols of automotive software and system components that are already in use across the industry. Particular attention has been paid to the combined testing of discrete and continuous behaviour and to the ability of *TTCN-3 embedded* to test real-time systems. *TTCN-3 embedded* supports

- the integrated testing of discrete and continuous behaviour,
- the cross-platform exchange of test definitions (MiL / SiL / HiL)
- the definition of tests for the entire test and integration process.

TTCN-3 is a high-level language that allows a particular abstraction from technical details and from technology to be tested and its implementation assessed. By means of *TTCN-3*, users are able to specify tests at an abstract level and focus on the definition of the test cases rather than testing the adaptation and execution of the system as a whole. *TTCN-3* enables, therefore, a systematic and specification-based test development for various kinds of tests, including tests that assess functionality, scalability, load, interoperability, robustness, regression, total system and integration. *TTCN-3 embedded* has been designed using the same level of abstraction. It has extended this common technological foundation to now include various testing activities like integration testing and system and acceptance testing for a large portion of the software-based systems that exist in the automotive industry. In general, *TTCN-3*, when combined with *TTCN-3 embedded*, has shown the potential to serve as a testing infrastructure that positively influences the degree of automation and reuse in industrial processes.

One of the biggest challenges in the writing of this thesis was to link the concepts for testing of continuous and hybrid systems with the existing concepts of *TTCN-3*. *TTCN-3* is an asynchronous language that has rarely been used for testing real-time systems. The concepts for *TTCN-3 embedded*, however, require that the language is synchronized and can be used in a real-time setup.

The basis for synchronization in *TTCN-3 embedded* is sampling. Consequently, all processing steps of a *TTCN-3 embedded* test program must be incorporated in the sampling mechanism. Inherent processes, such as the *TTCN-3* snapshot semantics, must also be aligned with the sampling, rather than with the explicit processes that create and start distributed components. Section 5.4.5 describes how the integration of the snapshot semantics was accomplished. Snapshots are simply timed to occur at full steps of the underlying base sampling, i.e. at the steps of the fastest possible clock rate of the test system. The synchronized start of distributed components is described in Section 6.3.2. Basically, it is assumed that the distributed hardware clocks are synchronized with a mechanism that is outside of *TTCN-3*. This can be done using established methods such as GPS synchronization and NTP synchronization. In this case, the *TTCN-3 embedded* environment is only responsible for communicating the start time of a component correctly.

Challenges with respect to the real-time capability of the language occur in two notable areas: the language structures are time-consuming during execution simply due to their complexity, and they can block instructions that contradict the idea of sampling. The latter has led to some *TTCN-3* statements not being used in time-critical sections of a *TTCN-3 embedded* program. So, for example, using receive-statements in the body of a mode construct is not allowed; therefore, *TTCN-3* statements are not used in such a construct. These limitations are, however, acceptable and their compliance with the respective rules is ensured by the grammar. The issue of complex constructs (like, for example, stream ports) may, at high sample rates and long test times, produce a considerable execution and management effort. This effort may have a negative impact on overall execution time and thus hinder the real-time capabilities of the test system. To counteract this special issue, the length of the stream history can be fixed, so that only a small amount of messages needs to be managed. In general, the language has been designed in such a way that there is no conceptual level construct that undermines the language's ability to perform as a real-time test system.

On the practical level, real-time performance is a matter of speed of execution, which is determined by the real-time capabilities of the language and the capacities of the operating system that is used to implement the test. For professional use, versions based on higher programming languages (e.g. Java)

are currently used. These languages bring along uncertainties and introduce some non-determinism regarding real-time behaviour.

In order to prove and test concepts, a prototypical *TTCN-3* compiler has been developed. The compiler supports the main *TTCN-3 embedded* concepts and generates executable test code in C# or C which integrates with typical automotive testing environments. The C# code-based test system has been used to integrate *TTCN-3 embedded* with CANoe and the C code-based test system to integrate *TTCN-3 embedded* with Simulink S-Functions.

The integration with Simulink S-Functions allows *TTCN-3 embedded* to perform systematic MiL test cases. The *TTCN-3 embedded* test executable fits neatly in the S-Function interface, which automatically executes during a simulation run. Details of the integration can be found in Section 6.4. Based on this integration, several MiL test cases for several typical automotive control systems (including an Adaptive Cruise Control, an Automatic Transmission Engine and an Engine Controller) [45] have been carried out. The case study concerning the Automotive Cruise Control is documented in Chapter 7. Generally, the case studies showed the applicability of the *TTCN-3* embedded concepts for the intuitive definition of stimulation and assessment procedures, especially for continuous signals and the execution of tests with simulation time. The integration of *TTCN-3 embedded* with Vector CANoe [125] allows for the possibility of testing control systems in both a simulation environment and with real hardware in a networked environment. In this particular integration, the *TTCN-3* test executable was integrated as an independent test component of the CANoe test environment. The test executable interacts with the CANoe test environment and the SUT by adapting to the environment variables and the CANoe network API. The case studies performed using the *TTCN-3* and embedded CANoe integration are especially relevant because they address the testing of control systems on HiL and SiL levels. Other available case studies about *TTCN-3* include an indicator and light control system test [83] and a window lifter control system test [43]. The main concepts of this thesis and their integration with *TTCN-3* have been submitted to the ETSI for standardization. In the meantime, *TTCN-3 embedded* has been accepted and published [38].

8.2 Future work and prospects for industrialization

Today's automobiles have dozens of control units, thousands of functional features and software with tens of thousands of line of code. The dispersed production of automobile component parts has produced inconsistency and inefficiency in the testing of said parts. This work has shown that, with

TTCN-3 embedded, a standardized solution is available. A standardized solution is valuable because it allows for the convergence and harmonization of test technologies that aim to evaluate discrete telecommunications systems and continuous control systems. *TTCN-3 embedded* enables the development of test cases which can be exchanged between different platforms, because it allows users to compose tests with suitable and variable levels of abstraction and particular design specifications. In *TTCN-3 embedded*, users also have the ability to directly execute test specifications. These factors make *TTCN-3 embedded* an ideal language for industry-wide use in the testing of telecommunications systems and continuous control systems.

The reuse of tests and testing artifacts based on *TTCN-3 embedded* in model-based testing approaches has been outlined in [45]. The fact that *TTCN-3 embedded* is standardized increases the chances that the language can be used by suppliers as well as the OEM. In addition, it increases the chance that reliable training programmes will become available for disposal. Future generations of vehicles will be integrated into a comprehensive communications infrastructure which will enable the exchange of data between individual vehicles, bulks of vehicles and traffic control centres. Ensuring the quality of these systems presents a new set of distinct challenges, because the combination of systems is unique and unprecedented, and because the testing systems and the specification of test cases have particular requirements. *TTCN-3 embedded* has shown that it has the potential to meet the requirements of these kinds of systems [47]. However, the approach described in the aforementioned research has yet not rolled out due to the absence of an industrial grade *TTCN-3 embedded* compiler.

During the development of *TTCN-3 embedded*, there were parallel research endeavours and attempts to introduce so-called signal assessment patterns. Assessment patterns are an approach that introduce regular expressions for signal evaluation, like syntax. These expressions follow the idea of *TTCN-3* templates and have been optimized in such a way that they can express the evaluation of larger data stream segments. A first approach addressing this issue has been published in [48]. Unfortunately, these expressions do not match the advantages of expressiveness and clarity that are offered by the automaton model described in this thesis, and thus they have not been further pursued. There is still the need, however, to find suitable abstractions to concisely define signal quality properties and signal assessments in an intuitive way. Moreover, the link to model-based testing approaches needs to be clarified. *TTCN-3 embedded* already provides a higher degree of abstraction and automation with respect to test execution, but a link to methods and techniques that provide automated test generation based on models is still lacking. *TTCN-3 embedded* can provide a common execution

environment for testing these approaches; the concrete role, however, still needs to be defined.

Last but not least, the overall interest of the automotive industry is still, so far, low. The reasons are manifold: they range from a basic skepticism about technologies that have not been developed in the automotive industry itself, to the existence of well-established alternative technologies, and the lack of an industrial-grade implementation of the concepts that have been developed in this thesis. Furthermore, car manufacturers have invested heavily in recent years in developing their own test infrastructure. These include test infrastructures developed at VW, Audi (Exam) and Mercedes (ProveTechTA). However, at present, the Fraunhofer Institute FOKUS is developing an open source *TTCN-3* compiler, which will provide another opportunity to put the concepts described in this thesis onto a solid tool basis and thus make them of greater interest for the industry.

Glossary

Abstract state machine (ASM) An abstract state machine (ASM) is a computational model that operates on states defined by algebras. Abstract state machine perform state transitions via guarded function updates, which are known in the parlance as rules; these rules describe how one algebra can be transformed into another. Abstract state machines were originally introduced in complexity theory. Their applicability to formal specification in multiple fields has been demonstrated in various publications; likewise, their applicability to the definition of semantics in various programming languages has also been shown.

Continuous signal A continuous signal is a time-varying quantity that can be measured by a technical system and whose values show continuous representation (i.e. are represented by real values).

Continuous system A dynamic system whose inputs and outputs are capable of changing at any instant of time; also known as a continuous-time signal system.

Control system A control system is a device, or a set of devices, that manages, commands, directs or regulates the behaviour of other devices or systems. In this thesis, software-based control systems (defined as control systems that rely heavily on software in order to produce the required functions) are given special consideration.

Electronic control unit (ECU) An electronic control unit (ECU) is a software-based control systems that consists of a single piece of hardware. This contrasts with the already-introduced term "control system", which can refer to compound systems (i.e. systems that consist of multiple control units).

Hardware in the Loop (HiL) Is a simulation environment that contains both real hardware and partially-simulated sensors, actuators, and mechanical and electrical components. HiL environments enable the testing of electronic characteristics and they can also simulate a complete network of interacting ECUs. OEMs generally use HiL Environments during the final stages of development to test and simulate the complete electronic infrastructure of a vehicle. The realtime HiL computer that controls the test execution offers a realistic test of time-critical requirements and the interaction of sensors, actuators and ECUs over a bus system. It also simulates an environment that shows at least some of the electrical properties of the original system. HiL testing also tests ECUs in their composition and, thus, possible communication errors can be identified.

Hybrid automaton Hybrid automata are a conceptual extension of Timed Automata. They where first introduced by Alur et al. in 1992, when they were used to analyze the properties of hybrid systems. Hybrid Automata consist of State Transition Networks (e.g. Finite State Machines) that define so-called phases or modes. Each phase or mode shows different continuous behaviour.

Hybrid system A hybrid system is a dynamic system, the behavioural evolution of which depends on variables with continuous and discrete quantities. This term encompasses a larger class of dynamic systems that show similar dynamic phenomena.

Model-in-the-Loop (MiL) A MiL environment is a simulation environment dedicated to model-based system development. The simulation environment ensures the execution of the model and its integration with any other present environment models. The tests assess the correct modelling of the functional requirements and, additionally, produce feedback about the suitability of the test algorithms.

Processor in the Loop (PiL) A PiL environment is a simulation environment in which the executable code of the application (or functionality) being tested is placed on an evaluation board, or on an appropriate processor simulation. PiL tests aim to find target-specific sources of failure, such as target-specific compilation issues or specifics for a concrete processor architecture. In comparison with tests executed upon original hardware, PiL tests are executed in a controlled and instrumented environment. This allows for additional measurement and observation. The test results can be compared with the results of previously-executed MiL and SiL tests, in order to find any unexpected deviations in the system reaction.

Software-in-the-Loop (SiL) A SiL environment is a simulation environment in which the code (which can be either hand-coded or system-generated, using the application of model-based development techniques) is tested in a software environment in the development machine. This sort of test attempts to assess the implementation of the functional requirements and the correctness of the code that has been generated out of the previously-validated model. Compilation-specific issues, like the scaling of fix point arithmetic results, are considered.

Test case A test case is a set preconditions, inputs (including actions, where applicable), and expected results; it is developed in order to determine whether or not the covered part of the SUT has been implemented correctly.

Test result A test result is an indication of whether or not a specific test case has passed or failed, i.e. if the actual result corresponds to the expected result or if deviations were observed. Relevant testing standards refer to test results with the verdict values of none, pass, inconclusive, fail or error.

Test suite A test suite is sequence of test cases in execution order, and any associated actions that may be required to set up the initial preconditions and any wrap up activities post execution.

Testing and Test Control Notation (TTCN-3) The Testing and Test Control Notation (The Testing and Test Control Notation (*TTCN-3*) is a well-established and widely-used test specification and execution environment. *TTCN-3* is a complete redefinition of the Tree and Tabular Combination Notation (TTCN-2). Both notations have been standardized by the ETSI and the ITU.

Bibliography

[1] J.-R. Abrial, *The B-book: assigning programs to meanings*, Cambridge University Press, New York, NY, USA, 1996. [26]

[2] J.-R. Abrial, *LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, Cambridge University Press, New York, NY, USA, 1996. [26]

[3] M. Agrawal and P. S. Thiagarajan, *The discrete time behavior of lazy linear hybrid automata*, in HSCC, M. Morari and L. Thiele, eds., Lecture Notes in Computer Science 3414, Springer, 2005, pp. 55–69. [25]

[4] B. K. Aichernig, H. Brandl, and F. Wotawa, *Conformance testing of hybrid systems with qualitative reasoning models*, Electr. Notes Theor. Comput. Sci. **253**, no. 2 (2009), pp. 53–69. [32]

[5] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, *Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems*, in Hybrid Systems, 1992, pp. 209–229. [22, 24, 26, 27, 28, 35, 41, 67]

[6] R. Alur and D. L. Dill, *A theory of timed automata*, Theoretical Computer Science **126**, no. 2 (1994), pp. 183–235. [24, 25, 26]

[7] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee, *Modular specification of hybrid systems in charon*, in HSCC '00: Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control, London, UK, 2000, Springer-Verlag, pp. 6–19. [26]

[8] R. Alur and T. Henzinger, *Modularity for timed and hybrid systems*, in CONCUR '97: Concurrency Theory, A. Mazurkiewicz and J. Winkowski, eds., Lecture Notes in Computer Science 1243, Springer Berlin / Heidelberg, 1997, pp. 74–88. [24, 41]

[9] AUTOSAR Consortium, *Applying Simulink to AUTOSAR, V1.0.5, R3.0*, 2008. [144]

[10] AUTOSAR Consortium, *Web site of the AUTOSAR consortium.* URL: `http://www.autosar.org`, 2013. [21]

[11] BERNER & MATTNER SYSTEMTECHNIK GMBH GERMANY, *MODENA – the specification and test tool for infotainment components*. URL: `http://www.berner-mattner.com/en/berner-mattner-home/products/modena/index.html`, June 2013. [20]

[12] R. V. BINDER, *Testing object-oriented systems: models, patterns, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. [9]

[13] E. BÖRGER, *Annotated bibliography on evolving algebras*, in Specification and Validation Methods, E. Börger, ed., Oxford University Press, 1994. [87]

[14] E. BÖRGER, *High level system design and analysis using abstract state machines*, in Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods, FM-Trends 98, London, UK, 1999, Springer-Verlag, pp. 1–43. [87]

[15] E. BÖRGER AND W. SCHULTE, *A programmer friendly modular definition of the semantics of java*, in Formal Syntax and Semantics of Java, London, UK, 1999, Springer-Verlag, pp. 353–404. [87, 92]

[16] H. BRANDL, G. FRASER, AND F. WOTAWA, *Qr-model based testing*, in Proceedings of the 3rd international workshop on Automation of software test, AST '08, New York, NY, USA, 2008, ACM, pp. 20–17. [32, 33]

[17] E. BRINGMANN AND A. KRAEMER, *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, 2006, ACM Press, pp. 13–20. [19, 27, 34]

[18] B. BROEKMAN AND E. NOTENBOOM, *Testing Embedded Software*, Addison-Wesley, 2002. [xiii, 15, 16]

[19] M. BROY, *Refinement of Time*, in Transformation-Based Reactive System Development, ARTS'97, M. Bertran and T. Rus, eds., no. LNCS 1231, TCS, 1997, pp. 44 – 63. [25, 27, 35, 42, 50]

[20] H.-W. W. CARSTEN GIPS, *Proposal for an automatic evaluation of ecu output*, in Dagstuhl-Workshop MBEES 2007: Modellbasierte Entwicklung eingebetteter Systeme, Braunschweig, GER, 2007. [29]

[21] CONFORMIQ, *Conformiq tool suite*. http://www.conformiq.com/products/, 2011. [20]

[22] M. CONRAD, *Modell-basierter Test eingebetteter Software im Automobil*, PhD thesis, TU-Berlin, 2004. [23, 27, 34, 150, 152]

[23] M. CONRAD AND E. SAX, *Mixed signals*, in Testing Embedded Software, 2003, pp. 229–249. [23]

[24] T. Dang, *Model-Based Testing for Embedded Systems*, CRC Press, 2011, ch. Model-based Testing of Hybrid Systems. [32]

[25] A. David, K. G. Larsen, S. Li, and B. Nielsen, *Timed testing under partial observability*, International Conference on Software Testing, Verification, and Validation, 2008 **0** (2009), pp. 61–70. [20, 30]

[26] C. Delgado Kloos, *The stream language*, in Semantics of Digital Circuits, Lecture Notes in Computer Science 285, Springer Berlin / Heidelberg, 1987, pp. 5–28. [25]

[27] A. Deshpande, A. Göllü, and P. Varaiya, *Shift: A formalism and a programming language for dynamic networks of hybrid automata*, in Hybrid Systems, P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, eds., Lecture Notes in Computer Science 1273, Springer, 1996, pp. 113–133. [24, 26]

[28] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, *A survey on model-based testing approaches: a systematic review*, in Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007, WEASELTech '07, New York, NY, USA, 2007, ACM, pp. 31–36. [13]

[29] E. W. Dijkstra, *Notes on Structured Programming.* circulated privately, Apr. 1970. [8]

[30] DIN, *DIN40146 Telecommunication – basic terms and definitions - DKE/K 111 Terminologie*, 07 2007. [23]

[31] W. Dröschel and M. Wiemers, *Das V-Modell 97. Der Standard fuer die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz*, Oldenbourg Wissenschaftsverlag, 1999. [10, 14]

[32] dSpace AG, *dSPACE - AutomationDesk.* URL: `https://www.dspace.com/de/gmb/home/products/sw/test_automation_software/automdesk.cfm`, 2013. [19]

[33] C. Efkemann and J. Peleska, *Model-based testing for the second generation of integrated modular avionics*, in Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW '11, Washington, DC, USA, 2011, IEEE Computer Society, pp. 55–62. [30]

[34] Etas Group, *ETAS - Hardware in the loop (HiL) - ECU Testing - Applications & Solutions.* URL: `http://www.etas.com/de/products/applications_ecu_development-testing.php`, 2013. [19]

[35] ETSI: ES 201 873-1 V4.5.1, *Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 1: TTCN-3 Core Language*, March 2013. [3, 47, 87]

[36] ETSI: ES 201 873-4 V4.4.1, *Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 4: TTCN-3 Operational Semantics*, March 2012. [34, 36, 87, 90]

[37] ETSI: ES 201 873-5 V4.5.1, *Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 5: TTCN-3 Runtime Interface (TRI)*, March 2013. [36, 48, 87]

[38] ETSI: ES 202 786 V1.1.1, *Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, TTCN-3 Language Extensions: Support of interfaces with continuous signals*, Febr. 2012. [178, 180]

[39] ETSI: TR 102 840 V1.2.1, *Methods for Testing and Specification (MTS). Model-based testing in standardization*, tech. report, European Telecommunications Standards Institute (ETSI), Sophia Antipolis, France, Febr. 2011. [13]

[40] A. GAMATIE, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*, Springer, New York, 2009. [25]

[41] U. GLÄSSER, R. GOTZHEIN, AND A. PRINZ, *Towards a New Formal SDL Semantics Based on Abstract State Machines*, SDL '99 - The Next Millenium, 9th SDL Forum Proceedings (1999). [87]

[42] H. GÖTZ, M. NICKOLAUS, T. ROSSNER, AND K. SALOMON, *Modellbasiertes Testen: Modellierung und Generierung von Tests - Grundlagen, Kriterien für Werkzeugeinsatz, Werkzeuge in der Übersicht*, iX Studie 01/2009, Heise Verlag, 2009. [12]

[43] J. GROSSMANN, *TEMEA Deliverable D2.4 concepts for the specification of tests for systems with continuous or hybrid behaviour*. 2010. [180]

[44] J. GROSSMANN, *Testing hybrid systems with TTCN-3 embedded*, International Journal on Software Tools for Technology Transfer (2014), pp. 247–267. [4]

[45] J. GROSSMANN, P. MAKEDONSKI, H.-W. WIESBROCK, J. SVACINA, I. SCHIEFERDECKER, AND J. GRABOWSKI, *Model-based X-in-the-loop testing*, CRC Press, Sept. 2011. [146, 149, 180, 181]

[46] J. GROSSMANN AND W. MUELLER, *A formal behavioral semantics for TestML*, in Proc. of IEEE ISoLA 06, Paphos Cyprus, 2006, pp. 453–460. [19, 34]

[47] J. Grossmann, C. Neumann, A. Hinnerichs, H. Rechner, T. Hecker, and I. Radusch, *Test von verteilten C2X-Applikationen*, in GI Jahrestagung (2), K.-P. Fähnrich and B. Franczyk, eds., LNI 176, GI, 2010, pp. 491–496. [181]

[48] J. Grossmann, I. Schieferdecker, and H.-W. Wiesbrock, *Modeling property based stream templates with TTCN-3*, in TestCom/FATES, K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, eds., Lecture Notes in Computer Science 5047, Springer, 2008, pp. 70–85. [29, 181]

[49] R. Grosu and T. Stauner, *Modular and visual specification of hybrid systems – an introduction to HyCharts*, Tech. Report TU-I9801, Technische Univerität München, 1998. [26]

[50] J. Grossmann, M. Conrad, I. Fey, C. Wewetzer, K. Lamberg, and A. Krupp, *TestML a language for exchange of tests.* IMMOS project deliverable, 2006. [19, 152]

[51] Y. Gurevich, *Evolving algebras 1993: Lipari guide*, Oxford University Press, Inc., New York, NY, USA, 1995, pp. 9–36. [87]

[52] N. Halbwachs, F. Lagnier, and C. Ratel, *Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE*, IEEE Transactions on Software Engineering **18** (1992), pp. 785–793. [25, 26]

[53] D. Harel, *Statecharts: A visual formalism for complex systems*, Sci. Comput. Program. **8**, no. 3 (1987), pp. 231–274. [26]

[54] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, *Hytech: A model checker for hybrid systems*, in CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification, London, UK, 1997, Springer-Verlag, pp. 460–463. [22, 28]

[55] T. A. Henzinger and P. W. Kopke, *Discrete-time control for rectangular hybrid automata*, Theor. Comput. Sci. **221**, no. 1-2 (1999), pp. 369–392. [25]

[56] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, *Testing real-time systems using UPPAAL*, in Formal Methods and Testing, R. M. Hierons, J. P. Bowen, and M. Harman, eds., Lecture Notes in Computer Science 4949, Springer, 2008, pp. 77–117. [30]

[57] *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*, Springer, Berlin, 2008. [15]

[58] IEEE: The Institute of Electrical and Electronic Engineers, *IEEE Standard for signal and test definition*, IEEE Std 1641-2010 (Revision of IEEE Std 1641-2004) (2010), pp. 1 –336. [35]

[59] IEEE: THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC., *IEEE Standard VHDL (IEEE Std.1076-1993.)*, 1993. [34]

[60] IEEE: THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC., *IEEE Standard Test Language for all Systems–Common/Abbreviated Test Language for All Systems (C/ATLAS) (IEEE Std.716-1995.)*, 1995. [35]

[61] IEEE: THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC., *User's Manual for the Signal and Method Modeling Language*. URL: `http://grouper.ieee.org/groups/scc20/atlas/SMMLusers_manual.doc`, 1998. [35]

[62] IEEE: THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC., *IEEE Standard VHDL Analog and Mixed-Signal Extensions (IEEE Std.1076.1-1999.)*, 1999. [34]

[63] IEEE: THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC., *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE Std.1149.1 -2001)*, 2001. [34]

[64] ISO/IEC, *Information technology – Syntactic Metalanguage – Extended BNF, ISO/IEC 14977:1996(E), 1st ed.*, December 1996. [87]

[65] ISO/IEC, *Information technology - Open systems interconnection — Conformance testing methodology and framework - Part 3: The Tree and Tabular combined Notation (TTCN), ISO/IEC 9646-3, 2nd ed.*, November 1998. [11, 34, 36, 54]

[66] C. JARD AND T. JERON, *TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems*, Int. J. Softw. Tools Technol. Transf. **7**, no. 4 (2005), pp. 297–315. [33]

[67] A. JOSHI AND M. P. E. HEIMDAHL, *Model-based safety analysis of Simulink models using SCADE Design Verifier*, in SAFECOMP, R. Winther, B. A. Gran, and G. Dahll, eds., Lecture Notes in Computer Science 3688, Springer, 2005, pp. 122–135. [28]

[68] A. A. JULIUS, G. E. FAINEKOS, M. ANAND, I. LEE, AND G. J. PAPPAS, *Robust test generation and coverage for hybrid systems*, in Proceedings of the 10th international conference on Hybrid systems: computation and control, HSCC'07, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 329–342. [33]

[69] G. KAHN, *The semantics of a simple language for parallel programming*, in Information Processing '74: Proceedings of the IFIP Congress, J. L. Rosenfeld, ed., North-Holland, New York, NY, 1974, pp. 471–475. [25]

[70] I. R. Kendall and R. P. Jones, *An investigation into the use of hardware-in-the-loop simulation testing for automotive electronic control systems*, Control Engineering Practice **7**, no. 11 (1999), pp. 1343 – 1356. [16]

[71] D. E. Knuth, *Backus normal form vs. backus naur form*, Commun. ACM **7** (1964), pp. 735–736. [87]

[72] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, *Testing real-time embedded software using UPPAAL-TRON: an industrial case study*, in Proceedings of the 5th ACM international conference on Embedded software, EMSOFT '05, New York, NY, USA, 2005, ACM, pp. 299–306. [30]

[73] E. A. Lee, C. Hylands, J. Janneck, J. Davis II, J. Liu, X. Liu, S. Neuendorffer, S. S. M. Stewart, K. Vissers, and P. Whitaker, *Overview of the Ptolemy project*, Tech. Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001. [26]

[74] E. Lehmann, *Time Partition Testing: Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen*, PhD thesis, TU-Berlin, Berlin, 2004. [27]

[75] P. Liggesmeyer, *Modultest und Modulverifikation : state of the art*, Angewandte Informatik ; 4, BI-Wiss.-Verl., Mannheim [u.a.], 1990. [8]

[76] P. Liggesmeyer, *Software-Qualität: Testen, Analysieren und Verifizieren von Software*, Spektrum Akademischer Verlag, Heidelberg, Berlin, 2002. [8, 9]

[77] *Advanced topics in signal processing*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. [24, 41]

[78] B. Lu, W. McKay, S. Lentijo, X. W. A. Monti, and R. Dougal, *The real time extension of the virtual test bed*, in Huntsville Simulation Conference, Huntsville, AL, 2002. [16]

[79] N. A. Lynch, R. Segala, and F. W. Vaandrager, *Hybrid i/o automata revisited*, in HSCC '01: Proceedings of the 4th International Workshop on Hybrid Systems, London, UK, 2001, Springer-Verlag, pp. 403–417. [24]

[80] N. A. Lynch, R. Segala, F. W. Vaandrager, and H. B. Weinberg, *Hybrid i/o automata.*, in Rajeev Alur [99], pp. 496–510. [24, 25, 26, 41, 67]

[81] K. G. M. Grochtmann, *Classification trees for partition testing*, Software Testing, Verification & Reliability, volume 3, No 2 (1993), pp. 63–82. [19, 27]

[82] O. Maler, Z. Manna, and A. Pnueli, *From timed to hybrid systems*, in In Real-Time: Theory in Practice, REX Workshop, LNCS 600, Springer-Verlag, 1992, pp. 447–484. [26]

[83] M. MAUEL, H. W. NISSEN, AND G. HARTUNG, *An application of TTCN-3 embedded in the automotive sector.* URL: http://www.ttcn-3.org/TTCN3UC2011/Pres/26_T3UC-Mauel_AnApplicationOfTTCN3EmbeddedInTheAutomotiveSector.ppt, 2011. [148, 149, 176, 180]

[84] MBTECH GROUP, *PROVEtech:TA - Überblick.* URL: https://www.mbtech-group.com/eu-de/electronics_solutions/tools_equipment/provetech_toolsuite.html, 2013. [19]

[85] MICRONOVA SOFTWARE UND SYSTEME, *EXAM.* URL: http://www.exam-ta.de/, June 2013. [20]

[86] R. MILNER, *Calculus of Communicating Systems*, Springer Verlag, 1980. [26]

[87] S. MONTENEGRO, S. JAEHNICHEN, AND O. MAIBAUM, *Simulation-based testing of embedded software in space applications*, in Embedded Systems - Modeling, Technology, and Applications, G. Hommel and S. Huanye, eds., Springer Netherlands, 2006, pp. 73–82. 10.1007/1-4020-4933-1_8. [16]

[88] NATIONAL INSTRUMENTS, *NI TestStand - Products and Services - National Instruments.* URL: http://www.ni.com/teststand/, 2013. [19]

[89] OMG, *Uml 2.0 testing profile specification*, Mai 2005. [19]

[90] S. OSTER, *Feature Model-based Software Product Line Testing*, PhD thesis, Technische Universität Darmstadt, December 2011. Dissertation. [20]

[91] K. P. PARKER AND S. ORESJO, *A language for describing boundary scan devices*, J. Electron. Test. **2**, no. 1 (1991), pp. 43–75. [34]

[92] J. PELESKA, H. FIDANOSKA, A. HONISCH, H. LÖDING, H. SCHMID, P. SMUDA, AND D. TILLE, *Model-based testing in the automotive domain - challenges and solution.* Presentation slides, extended english version of presentation given on Iqnite 2011, http://www.informatik.uni-bremen.de/agbs/jp/papers/peleska_et_al_iqnite2011_eng_v5.pdf, May 2011. Duesseldorf. [20, 30]

[93] J. PELESKA, A. HONISCH, F. LAPSCHIES, H. LÖDING, H. SCHMID, P. SMUDA, E. VOROBEV, AND C. ZAHLTEN, *A real-world benchmark model for testing concurrent real-time systems in the automotive domain*, in Proceedings of the 23rd IFIP WG 6.1 international conference on Testing software and systems, ICTSS'11, Berlin, Heidelberg, 2011, Springer-Verlag, pp. 146–161. [20, 30]

[94] J. PELESKA, E. VOROBEV, AND F. LAPSCHIES, *Automated test case generation with smt-solving and abstract interpretation*, in Proceedings of the

Third international conference on NASA Formal methods, NFM'11, Berlin, Heidelberg, 2011, Springer-Verlag, pp. 298–312. [30]

[95] A. M. Pérez and S. Kaiser, *Multi-level test models for embedded systems*, in Software Engineering, G. Engels, M. Luckey, and W. Schäfer, eds., LNI 159, GI, 2010, pp. 213–224. [28]

[96] A. M. Pérez and S. Kaiser, *Top-down reuse for multi-level testing*, in ECBS, R. Sterritt, B. Eames, and J. Sprinkle, eds., IEEE Computer Society, 2010, pp. 150–159. [28]

[97] C. A. Petri, *Kommunikation mit Automaten*, PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962. [26]

[98] H. Plünnecke and W. Reisig, *Bibliography on Petri nets 1990.*, in Applications and Theory of Petri Nets, G. Rozenberg, ed., Lecture Notes in Computer Science 524, Springer, 1990, pp. 317–572. [26]

[99] R. Alur, T. A. Henzinger, E. D. Sontag, *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, Lecture Notes in Computer Science 1066, Springer, 1996. [25, 191]

[100] V. A. Saraswat, *Concurrent constraint programming*, MIT Press, Cambridge, MA, USA, 1993. [26]

[101] SCC20 ATML Group, *IEEE ATML specification drafts and IEEE ATML status reports.* URL: `http://grouper.ieee.org/groups/scc20/tii/`, 2006. [19, 35]

[102] I. Schieferdecker, E. Bringmann, and J. Grossmann, *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, 2006, ACM Press, pp. 29–36. [37]

[103] I. Schieferdecker and J. Grossmann, *Testing hybrid control systems with TTCN-3, an overview on continuous TTCN-3*, TTCN-3 STTT (2008). [37]

[104] S. Sims and D. C. DuVarney, *Experience report: the reactis validation tool.*, in ICFP, R. Hinze and N. Ramsey, eds., ACM, 2007, pp. 137–140. [28]

[105] J. M. Spivey, *Understanding Z: a specification language and its formal semantics*, Cambridge University Press, New York, NY, USA, 1988. [26]

[106] J. M. Spivey, *The Z notation: a reference manual*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. [26]

[107] R. F. STÄRK, J. SCHMID, AND E. BÖRGER, *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag, 2001. [87]

[108] F. STENH, *Extending a real-time model-checker to a test-case generation tool using libcoverage*, master's thesis, Uppsala University, Department of Information Technology, 2008. [30]

[109] R. STEPHENS, *A survey of stream processing, acta informatica, 34(7):491–541*, 1997. [25, 42]

[110] B. SUPARJO, A. LEY, A. CRON, AND H. EHRENBERG, *Analog boundary-scan description language (absdl) for mixed-signal board test*, in International Test Conference, 2006, pp. 152–160. [34]

[111] E. TECHNOLOGIES, *SCADE 6 A Model Based Solution For Safety Critical Software Development.* URL: http://www.esterel-technologies.com/technology/document-request/white-papers/ERTS2008-SCADE-6-A-Model-Based-Solution-For-Safety-Critical-Software.html, 2013. [25, 26]

[112] TEMEA, *The TEMEA research project (Testing Methods for Embedded Systems of the Automotive Industry), founded by the European Community (EFRE).* URL: `http://www.temea.org`, 2013. [iii, v, 37]

[113] D. TERR AND E. W. WEISSTEIN, *Algebraic Equation. from MathWorld – A Wolfram web resource.* URL: `http://mathworld.wolfram.com/AlgebraicEquation.html`, 2013. [24]

[114] G. TESCHL, *Ordinary Differential Equations and Dynamical Systems*, Graduate Studies in Mathematics 140, American Mathematical Society, 2012. [21]

[115] THE MATHWORKS, *Web pages of Stateflow - Design and simulate state machines and control logic.* URL: `http://www.mathworks.com/products/stateflow/`, 2013. [26]

[116] THE MATHWORKS, *Web pages of Matlab - The language of technical computing.* URL: `http://www.mathworks.com/products/matlab/`, 2013. [26, 150]

[117] THE MATHWORKS, *Web pages of Simulink - Simulation and model-based design.* URL: `http://www.mathworks.com/products/simulink/`, 2013. [26, 142]

[118] W. THIES, *Language and Compiler Support for Stream Programs*, ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2009. [25]

[119] W. THIES, M. KARCZMAREK, AND S. AMARASINGHE, *Streamit: A language for streaming applications*, in International Conference on Compiler Construction, Grenoble, France, Apr 2002. [25]

[120] J. Tretmans, *Test generation with inputs, outputs, and quiescence*, in TACAS, T. Margaria and B. Steffen, eds., Lecture Notes in Computer Science 1055, Springer, 1996, pp. 127–146. [30, 31, 32]

[121] University of Amsterdam, HCS Laboratory, *The qualitative reasoning and modelling portal*. URL: `http://hcs.science.uva.nl/QRM/index.html`, June 2013. [32]

[122] M. Utting, A. Pretschner, and B. Legeard, *A taxonomy of model-based testing approaches*, Software Testing, Verification and Reliability (2011), pp. n/a–n/a. [13]

[123] M. van Osch, *Hybrid input-output conformance and test generation*, in FATES/RV, K. Havelund, M. Núñez, G. Rosu, and B. Wolff, eds., Lecture Notes in Computer Science 4262, Springer, 2006, pp. 70–84. [31, 32]

[124] E. van Veenendaal, *Standard glossary of terms used in software testing, version 2.1*, April 2010. Produced by the – Glossary Working Party – International Software Testing Qualifications Board. [10]

[125] Vector Informatik GmbH, *Portfolio-Übersicht Steuergeräte-Test*. URL: `http://www.vector.com/vi_portfolio_en.html`, 2013. [19, 147, 180]

[126] W. W. Wadge and E. A. Ashcroft, *LUCID, the dataflow programming language*, Academic Press Professional, Inc., San Diego, CA, USA, 1985. [25]

[127] C. Wallace, *The Semantics of the C++ Programming Language*, in Specification and Validation Methods, E. Börger, ed., Oxford University Press, 1994. [87]

[128] E. Weyuker, *Testing component-based software – a cautionary tale:*, in IEEE Software, IEEE, 1998, pp. 54–59. [9]

[129] Wikipedia, the free encyclopedia, *List of unit testing frameworks*. URL: `http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`, June 2013. [11]

[130] J. Willems, *Paradigms and puzzles in the theory of dynamical systems*, Automatic Control, IEEE Transactions on **36**, no. 3 (1991), pp. 259–294. [21]

[131] J. Zander-Nowicka, *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain*, ph.d. thesis, Technical University Berlin, 2008. [28, 29]

Appendix A

TTCN-3 embedded grammar

Appendix A defines the syntax of *TTCN-3 embedded* embedded using extended Backus Naur Form (BNF). The syntax definition is based on the *TTCN-3* syntax definition in the current *TTCN-3* standard (ES 201 873-1 V4.4.1), which has been altered with the newly introduced *TTCN-3 embedded* constructs. Listing A.1 show the modifications to the original *TTCN-3* syntax definition. The newly introduced symbols are underlined and marked in red. The rule numbers refer to the original rules as defined in ES 201 873-1 V4.4.1. Listing A.2 show the additional rules that need to be integrated to define the *TTCN-3 embedded* syntax.

Listing A.1: Modified rules in *TTCN-3* grammar

```
07. ModuleDefinition ::= ( ([Visibility]
                             ( TypeDef |
                               ConstDef |
                               TemplateDef |
                               ModulePardef |
                               FunctionDef |
                               SignatureDef |
                               TestcaseDef |
                               AltstepDef |
                               ImportDef |
                               ExtFunctionDef |
                               ExtConstDef |
                               ModeDef |
                               ModeType)
                            ) |
                           (["public"] GroupDef ) |
                           (["private"] FriendModuleDef )
                         ) [WithStatement]

49. PortDefAttribs ::= MessageAttribs | ProcedureAttribs
                     | MixedAttribs | StreamAttribs

78. PortElement ::= PortIdentifier [ArrayDef]
                    [AssignmentChar PortInitialValue]

377. TimerStatements ::= TimeoutStatement | NonBlockingTimerStatements

419. ReferencedValue ::= ValueReference [ExtendedFieldReference] |
                         StreamOperation

455. AttribKeyword ::= EncodeKeyword |
                       VariantKeyword |
                       DisplayKeyword |
```

```
                        ExtensionKeyword |
                        OptionalKeyword |
                        StepsizeKeyword |
                        HistoryKeyword

467. VerdictStatements ::= SetLocalVerdict "(" SingleExpression
                           {"," LogItem} ")" | AssertStatement

497. BasicStatements ::= ModeSpecification | NonBlockingBasicStatements |
                         WaitStatement| StatementBlock

512. Assignment ::= ( VariableRef | AssignableStreamOps ) AssignmentChar
                    (Expression | TemplateBody)

529. OpCall ::= ConfigurationOps |
                VerdictOps |
                TimerOps |
                TestcaseInstance |
                ( FunctionOrModeInstance [ ExtendedFieldReference ] ) |
                ( TemplateOps [ ExtendedFieldReference ] ) |
                ActivateOp |
                NowOp |
                DurationOp |
                StreamDataOps |
                StreamNavigationOps |
                StreamEvalOps
```

Listing A.2: New rules in *TTCN-3 embedded* grammar

```
007a. ModeDef ::= ModeKeyword ModeDefIdentifier
                  "(" [FunctionFormalParList] ")"
                  [RunsOnSpec] ModeSpecification
007b. ModeKeyword ::= "mode"
007c. ModeDefIdentifier ::= Identifier

007d. ModeSpecification ::= BasicMode | ComplexMode [UntilBlock]
007e. BasicMode ::= ContKeyword "{" {Declaration} [OnEntryBlock]
                    [InvariantBlock] {BasicModeOp}
                   [OnExitBlock] "}"
007f. ContKeyword ::= "cont"
007g. BasicModeOp ::= ContinuesStatement
007h. ComplexMode ::= (ParKeyword | SeqKeyword ) "{" {Declaration}
                      [OnEntryBlock] [InvariantBlock] {ComplexModeOp}
                      [OnExitBlock] "}"
007i. ParKeyword ::= "par"
007j. SeqKeyword ::= "seq"
007k. ComplexModeOp ::= ModeSpecification | FunctionOrModeInstance |
                        LabelStatement

007l. UntilBlock ::= UntilKeyword ( "{" { UntilGuardList | GuardOp } "}"
                     | "(" [BooleanExpression] ")" )
007m. UntilKeyword ::= "until"
007n. UntilGuardList ::= {UntilGuardStatement}
007o. UntilGuardStatement ::= UntilGuardChar [GuardOp] StatementBlock
                              [UntilJump SemiColon]
007p. UntilJump ::= GotoStatement | ContinueStatement | RepeatStatement
007q. UntilGuardChar ::= "[" [BooleanExpression | ModePredicate] "]"
007r. ModePredicate ::= NotinvKeyword
007s. NotinvKeyword ::= "notinv"
```

```
007t. ModeType ::= TypeDefKeyword ModeKeyWord ModeTypeIdentifier
                   "(" [ ModePar { "," ModePar } ] ")"
                 [ RunsOnSpecOrSelf ]
007u. ModePar ::= FormalValuePar | FormalTimerPar
                | FormalTemplatePar | FormalPortPar | FormalModePar
007v. RunsOnSpecOrSelf ::= RunsKeyword OnKeyword ( ComponentType | SelfOp )
007w. ModeTypeIdentifier ::= Identifier
007x. ModeKeyWord ::= "mode"

007y. NamedMode ::= ModeKeyword ModeIdentifier [ModeTypeIdentifier]
                  "(" [ ModePar { "," ModePar } ] ")"
                  [RunsOnSpec] ModeSpecification
007z. ModeIdentifier ::= Identifier


007aa. InvariantBlock ::= InvKeyword "{" [InvariantList] "}"
007ab. InvKeyword ::= "inv"
007ac. InvariantList ::= BooleanExpression { SemiColon BooleanExpression }
007ad. OnEntryBlock ::= OnEntryKeyword NonBlockingStatementBlock
007ae. OnEntryKeyword ::= "onentry"
007af. OnExitBlock ::= OnExitKeyword NonBlockingStatementBlock
007ag. OnExitKeyword ::= "onexit"

007ae. Declaration ::= FunctionLocalInst

007af. NonBlockingStatementBlock ::= "{" { NonBlockingStatement
                                     [SemiColon] } "}"

007ag. NonBlockingStatement ::= NonBlockingFunctionStatement |
                              FunctionLocalDef |
                              FunctionLocalInst

007ah. NonBlockingFunctionStatement ::= NonBlockingTimerStatements |
                                      NonBlockingConfigurationStatements |
                                      ContinuesStatement
007ai. ContinuesStatement ::= FunctionOrModeInstance |
                            VerdictStatements |
                            NonBlockingBasicStatements |
                            NonBlockingCommunicationStatements
007aj. FunctionOrModeInstance ::= FunctionOrModeRef
                                "(" [FunctionActualParList] ")"
007ak. FunctionOrModeRef ::= [GlobalModuleId Dot] (FunctionIdentifier |
                            ExtFunctionIdentifier | ModeIdentifier ) |
                           PreDefFunctionIdentifier


007al. ConfigurationStatements ::= DoneStatement |
                                 NonBlockingConfigurationStatements
007am. NonBlockingConfigurationStatements ::= ConnectStatement |
                                 MapStatement |
                                 DisconnectStatement |
                                 UnmapStatement |
                                 KilledStatement |
                                 StartTCStatement |
                                 StopTCStatement |
                                 KillTCStatement


007af. NonBlockingCommunicationStatements ::= SendStatement |
                                          ReplyStatement |
                                          RaiseStatement |
                                          CheckStatement
```

```
049a. StreamAttribs ::= StreamKeyword  "{" { MessageList [SemiColon] }+ "}"
049b. StreamKeyword ::= "stream"

078a. PortInitialValue ::= Expression

377a. NonBlockingTimerStatements ::= StartTimerStatement |
                                     StopTimerStatement

419a. StreamOperation ::= StreamIdentifier Dot MatchKeyword
                          "(" [TemplateBody] ")"
419b. StreamIdentifier ::= Identifier

455a. StepsizeKeyword ::= "stepsize"
455b. HistoryKeyword ::= "history"

467a. AssertStatement ::= AssertKeyword "(" Expression {"," Expression } ")"
467b. AssertKeyword ::= "assert"

497a. NonBlockingBasicStatements ::= Assignment | LogStatement |
                                     LoopConstruct | ConditionalConstruct |
                                     SelectCaseConstruct
497b. WaitStatement ::= WaitKeyword "(" SingleExpression ")"
497c. WaitKeyword ::= "wait"

512a. AssignableStreamOps ::= Port Dot StreamDataAssignableOps
512b. StreamDataAssignableOps ::= ValueKeyword | DeltaKeyword
512c. ValueOpKeyword ::= "value"
512d. DeltaKeyword ::= "delta"

529a. NowOp ::= NowKeyword
529b. NowKeyword ::= "now"
529c. DurationOp ::= DurationKeyword
529d. DurationKeyword ::= "duration"

529e. StreamDataOps ::= StreamValueOp | StreamTimestampOp | StreamDeltaOp
529f. StreamValueOp ::= Port Dot PortValueOp
529g. PortValueOp ::= ValueOpKeyword

529h. StreamTimestampOp ::= Port Dot PortTimestampOp
529i. PortTimestampOp ::= TimestampOpKeyword
529j. TimestampOpKeyword ::= "timestamp"
529k. StreamDeltaOp ::= Port Dot PortDeltaOp
529l. PortDeltaOp ::= DeltaOpKeyword
529m. DeltaOpKeyword ::= "delta"
529n. StreamNavigationOps ::= ( StreamPrevOp | StreamAtOp )
                              [ Dot StreamDataOps ]
529o. StreamPrevOp ::= Port Dot PortPrevOp
529p. PortPrevOp ::= PrevOpKeyword [ "(" IndexValue ")" ]
529q. IndexValue ::= Expression
529r. PrevOpKeyword ::= "prev"
529s. StreamAtOp ::= Port Dot PortAtOp
529t. PortAtOp ::= AtOpKeyword [ "(" TimeIndexValue ")" ]
529u. TimeIndexValue ::= Expression
529v. AtOpKeyword ::= "at"
529w. FormalModePar ::= [InOutParKeyword] ModeTypeIdentifier
                        ModeParIdentifier
529x. ModeParIdentifier ::= Identifier

529y. StreamEvalOps ::= StreamHistoryOp
529z. StreamHistoryOp ::= Port Dot PortHistoryOp
529aa. PortHistoryOp ::= HistoryOpKeyword
```

```
                        [ "(" StartValue [ "," EndValue ] ")" ]
529ab. StartValue ::= Expression
529ac. EndValue ::= Expression
529ad. HistoryOpKeyword ::= "history"
```

Appendix B

The operational semantics of *TTCN-3 embedded*

Appendix B describes the ASM semantics of *TTCN-3*. The whole semantics is defined as a set of subsequent extensions. This kind of stepwise approach provides iterative language refinements beginning from an imperative core $TTCN_{\mathcal{I}}$ of *TTCN-3* that describes the basic imperative language features i.e. statement execution, expression evaluation, control structures of *TTCN-3*. On basis of that core, subsequent language refinements are specified. Each of them introduces new language constructs and thus refines the original semantics subsequently.

- $TTCN_{\mathcal{C}}$ is based $TTCN_{\mathcal{I}}$ and specifies *TTCN-3* with features like test case execution, components and function calls.
- $TTCN_{\mathcal{T}}$ is based $TTCN_{\mathcal{C}}$ and specifies *TTCN-3* with features like sending and receiving messages, alt statements, altsteps and timers.
- $TTCN_{\Delta}$ is based $TTCN_{\mathcal{T}}$ and specifies *TTCN-3 embedded* with features like global time and sampling.
- $TTCN_{\mathcal{E}}$ is based $TTCN_{\mathcal{E}}$ and specifies *TTCN-3 embedded* with modes.

While $TTCN_{\mathcal{I}}$, $TTCN_{\mathcal{C}}$, and especially $TTCN_{\mathcal{T}}$ covers the existing *TTCN-3* standard, $TTCN_{\Delta}$ and $TTCN_{\mathcal{E}}$ explicitly introduce the concepts of *TTCN-3 embedded*. Please note, the operational semantics of *TTCN-3* given below will not cover all aspects of the *TTCN-3* core language. It is restricted to the definition of behavioural elements in *TTCN-3*, which provide the behavioral basis for *TTCN-3 embedded* (i.e. expression evaluation, control structures, send and receive operations, the alt statement etc.).

B.1 Abstract syntax

The following sections denote the abstract syntax that is used as a basis for the abstract state machine specification. To allow back referencing between the abstract syntax used for the behavioral semantics and the concrete syntax of *TTCN-3* the node identifiers from the original *TTCN-3* grammar are also used for the AST definitions. The node identifiers, which are completely aligned with original *TTCN-3* grammar start with a capital letter and are written in camel case notation (e.g. *Assignment*, *ConditionalConstuct* etc.). However, to reduce complexity and to provide a better understanding of the AST, abstract nodes with identifiers completely written in capital letters (e.g. $PHRASE$, EXP, $STMT$, UOP, BOP etc.) are introduced. These identifiers are used to sum up constructs that, regarding the behavioural semantics, need not to be distinguished (e.g. BOP stands for a binary operation of the form $BOP ::= EXP \odot EXP$ with binary operators in $\odot \in \{+, *, -, /, mod, rem, ...\}$).

B.1.1 $TTCN_{\mathcal{I}}$ abstract syntax

Abstract syntax 17: *TTCN-3* basic behavioural constructs.

EXP	::=	*LIT* | *VAR* | *CONST* | *UNOP* | *BOP* | *OpCall*
STMT	::=	*Assignment* | *LogStatement* | *LoopConstruct* | *ConditionalConstruct* | *GotoStatement* | *StatementBlock*
PHRASE	::=	*EXP* | *STMT* | "`finished`"
UOP	::=	$\circledcirc EXP$
BOP	::=	$EXP_1 \odot EXP_2$
OpCall	::=	*FunctionInstance*
FunctionInstance	::=	*Function* { *FunctionActualPar* }
Assignment	::=	"`assign`" *VAREXP*
LogStatement	::=	"`log`" { *EXP* }
LoopConstruct	::=	*ForStatement* | *WhileStatement* | *DoWhileStatement*
ForStatement	::=	"`for`" *Assignment EXP Assignment StatementBlock*
WhileStatement	::=	"`while`" *EXP StatementBlock*
DoWhileStatement	::=	"`do`" *StatementBlock* "`while`" *EXP StatementBlock*
ConditionalConstruct	::=	"`if`" *EXP StatementBlock* [*StatementBlock*]
GotoStatement	::=	"`goto`" *LabelId*

B.1.2 $TTCN_{\mathcal{C}}$ abstract syntax

Abstract syntax 18: $TTCN_{\mathcal{C}}$ basic behavioural constructs.

$EXP_{\mathcal{C}}$	::=	$EXP_{\mathcal{I}}$ | *CreateOP* | *StartTCStatement*
$STMT_{\mathcal{C}}$	::=	$STMT_{\mathcal{I}}$ | *TestcaseInstance* | *StartTCStatement StopTCStatement*
		| *ReturnStatement* |
TTCN3Module	::=	"`module`" *Identifier DefinitionList StatementBlock*
TestcaseInstance	::=	"`execute`" *TestcaseRef* { *TestcaseActualPar* }
ReturnStatement	::=	"`return`" [*EXP*]
CreateOP	::=	"`create`" *Component Exp Alive*
StartTCStatement	::=	"`start`" *VAR FunctionInstance*
StartTCStatement	::=	"`kill`" *VAR*
StopTCStatement	::=	"`stop`" *VAR*

B.1.3 $TTCN_{\mathcal{T}}$ abstract syntax

Abstract syntax 19: $TTCN_{\mathcal{T}}$ basic behavioural constructs.

$EXP_{\mathcal{T}}$	::=	$EXP_{\mathcal{C}}$ | *ReadTimerOP* | *Receive*
$STMT_{\mathcal{T}}$	::=	$STMT_{\mathcal{C}}$ | *SendStatement* | *ReceiveStatement* |
		| *StartTimerStatement* | *StopTimerStatement*
StartTimerStatement	::=	"`start`" *VAR EXP*
		| *AltConstruct* |
StopTimerStatement	::=	"`stop`" *VAR*
ReadTimerOperation	::=	"`read`" *VAR*
SendStatement	::=	"`send`" *PortVar EXP Addresses*
AltConstruct	::=	"`alt`" { *GuardStatement* | *ElseStatement* }
GuardStatement	::=	*EXP AltStepInstance* [*StatementBlock*]
		| *GuardOp StatementBlock*
ElseStatement	::=	"`else`" *StatementBlock*
GuardOp	::=	*TimeoutStatement* | *ReceiveStatement* | *TriggerStatement*
		| *GetCallStatement* | *CatchStatement* | *CheckStatement*
		| *GetReplyStatement* | *DoneStatement* | *KilledStatement*
ReceiveStatement	::=	*Port* "`receive`" *TemplateInstance*
		["`from`" *FromSpec*]
		["`->`" *RedirectSpec*]
TimeoutStatement	::=	*Timer* "`timeout`"
ReceiveStatement	::=	*Port* "`receive`" *TemplateInstance*
		["`from`" *FromSpec*]
		["`->`" *RedirectSpec*]
TimeoutStatement	::=	*Timer* "`timeout`"

B.1.4 $TTCN_{\Delta}$ abstract syntax

Abstract syntax 20: $TTCN_{\Delta}$ basic behavioural constructs.

EXP_{Δ}	::=	$EXP_{\mathcal{T}}$ | *StreamDataOps* | *StreamNavigationOps*
$STMT_{\Delta}$	::=	$STMT_{\mathcal{T}}$ | *StreamEvalOps* | *WaitStatement*
StreamDataOps	::=	*StreamValueOp* | *StreamTimestampOp* | *StreamDeltaOp*
StreamNavigationOps	::=	*StreamPrevOp* | *StreamAtOp* ["`Dot`" *StreamDataOp*]
StreamEvalOps	::=	*StreamHistoryOp*
WaitStatement	::=	"`wait`" *EXP*
nowOperation	::=	"`now`"
StreamDataOp	::=	*StreamValueOp* | *StreamTimestampOp* | *StreamDeltaOp*
StreamValueOp	::=	"`value`" *PortVar*
StreamTimestampOp	::=	"`timestamp`" *PortVar*
StreamDeltaOp	::=	"`delta`" *PortVar*
Assignment	::=	"`assign`" *LeftSide EXP*
LeftSide	::=	*VAR* | *StreamValueOperation* | *StreamDeltaOperation*
StreamNavigationOp	::=	(*StreamPrevOp* | *StreamAtOp*) *StreamDataOp*
StreamPrevOp	::=	"`prev`" *PortVar*
StreamAtOp	::=	"`at`" *PortVar*
StreamEvalOps	::=	*StreamHistoryOp*
StreamHistoryOp	::=	"`history`" *PortVar Exp Exp*

B.1.5 $TTCN_{\mathcal{E}}$ abstract syntax

Abstract syntax 21: $TTCN_{\mathcal{E}}$ basic behavioural constructs.

$EXP_{\mathcal{E}}$	::=	EXP_{Δ} | *Duration*
$STMT_{\mathcal{E}}$	::=	$STMT_{\Delta}$ | *Mode*
Mode	::=	*ContMode* | *ParMode* | *SeqMode*
Mode	::=	("`cont`" | "`seq`" | "`par`") *OnEntry InvariantList Body OnExit Until*
Body	::=	*ContBody* | *SeqBody* | *ParBody*
InvariantList	::=	{ *EXP* }
Until	::=	"`until`" { *UntilGuardStatement* }
UntilBlock	::=	"`until`" { *UntilGuardStatement* }
UntilGuardStatement	::=	*EXP GuardOp StatementBlock* [*UntilJump*]
Transition	::=	*GotoTransition RepeatTransition ContinueTransition*
ContMode	::=	"`cont`" *OnEntry Invariant ContBody OnExit Until*
SeqMode	::=	"`seq`" *OnEntry Invariant SeqBody OnExit Until*
SeqBody	::=	{ *Mode* }
ParMode	::=	"`cont`" *OnEntry Invariant ParBody OnExit Until*
ParBody	::=	{Mode}

B.2 SIGNATURES

The following sections define the signatures that are used as a basis for the abstract state machine definition. In the following the same identifiers and denotations are used for the AST nodes and the ASM universes. Eventually, this kind of double definition facilitates the understanding of the ASM rules because it directly relates the terms used in the ASM rules with the AST definition and therefore with the *TTCN-3* grammar. Thus, the AST definition

$$PHRASE \quad ::= \quad EXP \,|\, STMT \,|\, \texttt{"finished"}$$

is considered to define the set-theoretic structure

$$PHRASE = EXP \cup STMT \cup \{finished\}$$

with elements

$$stmt \in STMT,\ exp \in EXP,\ phrase \in PHRASE.$$

These elements themselves are each representing n-ary tuples, which are referred to either in a compact or a developed form.

B.2.1 $TTCN_{\mathcal{I}}$ signatures

A run of a *TTCN-3* program is considered to be a subsequent run over the executable entities of an annotated AST. The entities are called phrases and are defined as elements of the universe $PHRASE$. A phrase defines the occurrence of a certain programming construct, which is subject of the formal specification in the following ASM semantics. For procedural, statement based languages, the operational progress can be described by means of program counter *task* that points to the individual nodes of an AST. The program counter yields the current *phrase* to execute until $finished$ is reached. The abstract program counter *task* will update according the *TTCN-3* control flow. The execution of expressions, subexpressions, statements and substatements is defined by the static functions fst and nxt. The definitions of these functions depend on the respective element $phrase \in PHRASE$ and will provided recursively during the refinement of the language elements.

$$\begin{aligned} task &: \rightarrow PHRASE \\ fst, nxt &: PHRASE \rightarrow PHRASE \end{aligned}$$

The universe $Value^T$ contains all kinds of *TTCN-3* values of a certain type $T \in \mathbb{T} = \{integer, float, boolean, ...,\}$ so that $Value^{float}$ denotes all *TTCN-3* float values, and $Value^{integer}$ denotes all *TTCN-3* integer values and so forth. For reasons of simplicity, the *TTCN-3* Boolean values are associated with the ASM values *true* and $false$.

To describe the relation between variables and their values the formal function *loc* is used.

$$loc \ : \ Component \rightarrow Value^T \qquad \text{with } T \in \mathbb{T}$$

The access to intermediate values –which are not assigned to a variable but used during expression evaluation, for operand handling, for parameter passing and for return-value passing, as well as for expressions in conditional statements– is formalized by the function *val*.

$$val \ : \ Component \rightarrow Value^T \qquad \text{with } T \in \mathbb{T}$$

B.2.2 $TTCN_{\mathcal{C}}$ signatures

This section defines the operational semantics of $TTCN_{\mathcal{C}}$. In addition to $TTCN_{\mathcal{I}}$, $TTCN_{\mathcal{C}}$provides a component based test architecture. 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 channels, so called ports. *TTCN-3* distinguishes three different types of components. A component type called "control" that is used to execute the control parts of modules, a component type called "Main Test Component (MTC)" that is used to execute test case bodies and a component type called "Test Component (TC)" that is used to realize test set ups with multiple components.

The main entities that are needed in $TTCN_{\mathcal{C}}$ are representations for components and so called agents. Components with $c \in Component$ describe the variety of *TTCN-3* components.

$$
\begin{aligned}
ports &: Component \rightarrow Port \\
status &: Component \rightarrow \{running,\ inactive,\ stopped,\ killed\} \\
type &: Component \rightarrow \{control, mtc, tc\} \\
alive &: Component \rightarrow \{true,\ false\} \\
verdict &: Component \rightarrow Value^{verdict}
\end{aligned}
$$

Moreover, sequences of entities are introduced. A sequence is denoted with $\langle e_2, e_1\rangle \in Universe^*$. The concatenation of sequences is simply expressed by a row position of two single sequences. Thus $\langle e_3\rangle\langle e_2, e_1\rangle \mapsto \langle e_3, e_2, e_1\rangle$. The length of a sequence is defined by $len(\langle f_n, f_{n-1}, .., f_1\rangle) \mapsto n$. On basis of this the most common stack (lifo) functions are defined as follows.

$$
\begin{aligned}
&push : Universe \times Universe^* \rightarrow Universe^*, \\
&\quad \text{with } push(e_1, \langle\rangle) \mapsto \langle e_1\rangle \\
&\quad \text{and } push(e_{n+1}, \langle e_n, e_{n-1}, .., e_1\rangle) \mapsto \langle e_{n+1}\rangle\langle e_n, .., e_1\rangle \mapsto \langle e_{n+1}, e_n, .., e_1\rangle \\
&top : Universe^* \rightarrow Universe, \\
&\quad \text{with } top(\langle e_n, e_{n-1}, .., e_1\rangle) \mapsto e_n \\
&\quad \text{and } top(\langle\rangle) \mapsto undef \\
&pop : Universe^* \rightarrow Universe^*, \\
&\quad \text{with } pop(\langle e_n, e_{n-1}, .., e_1\rangle) \mapsto \langle e_{n-1}, .., e_1\rangle \\
&\quad \text{and } pop(\langle\rangle) \mapsto \langle\rangle
\end{aligned}
$$

For simplicity, the term $frames$ is used in order to bundle the functions needed for the recursive calling. Thus, for every $c \in Component$ let

$$frames(c) \equiv (task_{\mathcal{C}}(c), loc_{\mathcal{C}}(c), val_{\mathcal{C}}(c))$$

$TTCN_{\mathcal{I}}$ does not support nested scopes and recursions. However, as shown above, the functions $task$, loc and val in $TTCN_{\mathcal{I}}$ can be mapped to the functions $task_{\mathcal{C}}(Self)$, $loc_{\mathcal{C}}(Self)$ and $val_{\mathcal{C}}(Self)$ in $TTCN_{\mathcal{C}}$ by associating $Self$ and by taking the topmost elements. This ensures that the rules from $TTCN_{\mathcal{I}}$ can be transfered directly to $TTCN_{\mathcal{I}}$. For reason of simplicity, $Self$ is notational suppressed by writing $task_{\mathcal{C}}$, $loc_{\mathcal{C}}$ and $val_{\mathcal{C}}$ instead of $task_{\mathcal{C}}(Self)$, $loc_{\mathcal{C}}(Self)$ and $val_{\mathcal{C}}(Self)$. Finally the terms $task$, loc and val are used like in $TTCN_{\mathcal{I}}$, except when the stack functionality needs explicitly be mentioned.

$$\begin{aligned} task_{\mathcal{C}} &: Component \rightarrow PHRASE^* \\ loc_{\mathcal{C}} &: Component \rightarrow (VAR \rightarrow Value)^* \\ val_{\mathcal{C}} &: Component \rightarrow (EXP \rightarrow Value)^* \end{aligned}$$

$$current : AGENT \rightarrow Component$$

An agent is the abstract executer of a *TTCN-3* component. Let $AGENT$ be the abstract set of agents a that move over the hierarchical syntax structure of an *TTCN-3* program. An agent is associated with exactly one *TTCN-3* component at any time, thus $current : AGENT \rightarrow Component$ is a dynamic function that denotes the component under execution. The 0-ary dynamic function $Self$ is interpreted by each agent a as a. Because a component is considered the carrier of *TTCN-3* program execution, each $c \in Component$ has a program pointer, which points to the next $phrase \in PHRASE$ to execute. To support nested scope units due to nested and potentially recursive function calls or altstep calls, the program pointer as well as the functions loc and val have to realize a stack alike functionality. To this end, the signatures of these functions are refined to the effect that they provide sequences of dynamic functions as return values.

B.2.3 $TTCN_{\mathcal{T}}$ signatures

TTCN-3 supports message-based and procedure-based unicast, multicast and broadcast communication. Technical speaking a messages is a piece of information, which is prepared for transportation between two entities. Thus it may be encoded and enriched with transport related information (sender, receiver, timing information etc.).

A *TTCN-3* test system supports the sending of messages and the systematic evaluation of incoming messages at ports. A port is associated with a message queue that stores all incoming messages in order of arrival. The evaluation of individual messages always refers to top elements of the port queues. Access control to ports and evaluation of the port queues' content is done by means of so called communication operations. In addition, time-dependent processes are controlled by providing access to so-called timer.

The main entities that are needed to specify the behaviour of port and timer handling in $TTCN_{\mathcal{T}}$ are covered by the respective ASM universes $Port$, $Message$ and $Timer$.

A *TTCN-3* timer is a counter that is incremented according to the current time progress. Such a timer can be configured with a timeout value. If a timer reaches its timeout value the timer is considered to have *timedout*, thus it changes from the status *running* to the status *expired*.

$$
\begin{aligned}
start &: Timer \times Value^{float} \rightarrow \\
stop &: Timer \rightarrow \\
status &: Timer \rightarrow \{\ running,\ inactive,\ expired\} \\
val &: Timer \rightarrow Value^{float} \\
timeoutval &: Timer \rightarrow Value^{float}
\end{aligned}
$$

TTCN-3 uses ports to communicate with the external environment (i.e. the SUT) as well as between *TTCN-3* components. Ports are defined by means of port type definitions. *TTCN-3* distinguishes between ports of kind message and procedure. Message ports are used for non-blocking (asynchronous) message based communication while procedure ports are used to realize classical client-server communication. A port can have a set of *types*. The set of types define the types for the message values that are allowed to be communicated via a certain port.

$$
\begin{aligned}
kind &: Port \rightarrow \{message, procedure\} \\
direction &: Port \rightarrow \{in, out, inout\} \\
types &: Port \rightarrow \mathcal{P}(T) \\
msgs &: Port \rightarrow Message^* \\
owner &: Port \rightarrow Component \\
status &: Port \rightarrow \{\ started,\ halted,\ stopped\}
\end{aligned}
$$

Using the function *msgs* it is possible to obtain the port queues, i.e. the chronological sorted sequence of messages that has been received at a port. The function *owner* returns the component that holds a given port. Please note, in $TTCN_{\mathcal{C}}$ the sequence of incoming messages (message from the external environment or from other *TTCN-3* components) at a port are handled by means of queue operations. Incoming messages are enqueued. They can be retrieved and assessed by a *TTCN-3* program using the queue-based operations that are defined below.

$$
\begin{aligned}
&enqueue : Universe^* \times Universe \rightarrow Universe^*, \\
&\qquad \text{with } enqueue(e_1, \langle\rangle) \mapsto \langle e_1 \rangle \\
&\qquad \text{with } enqueue(e_{n+1}, \langle e_1, .., e_n \rangle) \mapsto \langle e_{n+1} \rangle \langle e_1, .., e_n \rangle \mapsto \langle e_1, .., e_n, e_{n+1} \rangle. \\
&dequeue : Universe^* \rightarrow Universe^*, \\
&\qquad \text{with } dequeue(\langle e_1, e_2, .., e_n \rangle) \mapsto \langle e_2, .., e_n \rangle \\
&\qquad \text{and } dequeue(\langle\rangle) \mapsto \langle\rangle
\end{aligned}
$$

In *TTCN-3* ports can be mapped or connected to other ports. This forms the base communication architecture of test system set up and defines the general condition for the message flow between components. *TTCN-3* allows one to many connections and distinguishes port connections between a test component and the system under tests and port connections between test components. This operational semantics does not explicitly define meaning for statements that are directly handle the mapping and connection of ports but requires to refer to the mapping and connection information. Thus, the function

$$mapsto : Port \rightarrow Port * \cup \{undef\}$$

returns the set of port that contains all ports that are mapped or connected to a given port. The function returns $undef$ in the case that a port is not mapped or connected.

A *TTCN-3* message is a structure that encapsulates information about the transmitted value, the type of the transmitted value and the sender and receiver of the message. In *TTCN-3* sender and receiver of messages are components. Moreover *TTCN-3* provides a mechanism to match messages against a pattern expression. The pattern expression is called template and characterizes messages according to their content and structure. Without any loss generality a template can be considered as an expression $exp \in EXP$. A function map allows to check whether a message matches a template or not.

$$\begin{aligned} type &: Message \rightarrow T \\ sender &: Message \rightarrow Component \\ recipient &: Message \rightarrow Component \\ val &: Message \rightarrow Value^{T} \\ ts &: Message \rightarrow Value^{float} \end{aligned}$$

$$match : Message \times EXP \rightarrow Value^{booelan}$$

The assessement of ports in *TTCN-3* is organized according to the so called snapshot semantics. The snapshot semantics enforces that the relevant test system state is frozen when an assessment starts. This kind of snapshot guarantees a consistent view on the test system input during an individual assessment step. A snaphsot covers all relevant stopped test components, all relevant timeout events and the top messages (as well as calls, replies and exceptions in the case of procedure based communication) in the relevant incoming port queues. Let me introduce the following primed functions to cover snapshots for timer and ports.

$$\begin{aligned} val' &: Timer \rightarrow Value^{float} \\ status' &: Timer \rightarrow \{running,\ inactive,\ expired\} \\ msgs' &: Port \rightarrow Message^{*} \end{aligned}$$

The functions val', $status'$ and $msgs'$ can be used similarly to the non-primed version of the functions. The difference is that, where val, $status$ and $msgs$ return the actual values or messages, val', $status'$ and $msgs'$ return the state of the timers and queues at the last snapshot.

Finally *TTCN-3* and *TTCN-3 embedded* provide statements that contain other statements in form of a statement block (e.g. alt statements or modes). For control flow purposes it becomes necessary to be able to refer from such a contained statement to the enclosing block statement during runtime. This helps realizing abrupt breaks and other exit behaviour. The function up provides the TE with such a feature.

$$up : PHRASE \rightarrow PHRASE$$

The function up gets an arbitrary phrase as a parameter and always returns the enclosing phrase. The hierachy, which is used to resolve the containment relationship, is given by the definition of the abstract syntax tree.

B.2.4 $TTCN_\Delta$ signatures

In all foregoing specifications time progress itself only has had an implicit semantics. It was possible to measure time progress with the timer operations and thus change the control flow of a *TTCN-3* programm as a function of time. In $TTCN_\Delta$ time and time progress becomes a more central concept. $TTCN_\Delta$ is a synchonuous language. That is, statement execution and especially the interaction through ports is synchronized on basis of a system-wide valid sampling rate. Time is referred to by the functions $time$, t_{sample}, t'_{sample}.

$$\begin{aligned} t &:\rightarrow Value^{float} \\ t_{sample} &:\rightarrow Value^{float} \\ t'_{sample} &:\rightarrow Value^{float} \end{aligned}$$

The function t provides access to the exact time that has passed since the beginning of a test case, t_{sample} represents the time of the next sampling step whereas t'_{sample} represents the time of the actual sampling step. The last two functions are directly related to each other by the step size *delta* that defines the length of the minimal overall sampling. It is $t_{sample} = t'_{sample} + \delta$ .All time functions return time in seconds as a Float value.

$TTCN_\Delta$ introduces a new kind of port, a so called stream port. Such a port differs from ordinary *TTCN-3* message ports and *TTCN-3*procedure ports by providing a sampled access to the actual value as well as (theoretically) the complete history of a continuous signal. However, the underlying data structure is quite similar to the data structure defined for *TTCN-3* message and procedure ports.

Stream ports $port \in Port^{Stream}$ in general show similar basic properties than message ports and procedure ports. Thus, most of the functions that have been defined for message and procedure ports (see B.3.3) are adopted for stream ports.

$$\begin{aligned} type_\Delta &: Port_\Delta \rightarrow \mathcal{P}(T) \\ msgs_\Delta &: Port_\Delta \rightarrow Message^* \\ msgs'_\Delta &: Port_\Delta \rightarrow Message^* \\ direction_\Delta &: Port_\Delta \rightarrow \{in, out, inout\} \end{aligned}$$

with

$$Port_{\Delta} = Port_{\mathcal{T}} \vee Port.$$

However,there are slight differences. A stream port has a distinct data type. Hence, the function *type* returns a set with one element only for a stream ports. Moreover, only message and procedure ports with direction *in* or *inout* are normally associated with a queue (i.e. list) of message values. In contrast, stream ports always have a message queue, regardless of their direction of communication. Last bit not least the newly introduced function *delta* returns the currently used sampling rate of a stream port and the function *current* returns the actual value of a stream port.

$$delta_{\Delta} : Port^{stream} \rightarrow Value^{Float}$$
$$current_{\Delta} : Port^{stream} \rightarrow Value^{T}$$

The function *kind* helps distinguishing between the different kind of ports. The function has to be refined to also address stream ports.

$$kind_{\Delta} : Port_{\Delta} \rightarrow \{message, procedure, stream\}$$

Moreover I consider that the function *ports* that returns all ports that are available at a component (see Section B.3.2), is refined to support stream ports as well.

$$ports_{\Delta} : Component_{\Delta} \rightarrow Port_{\Delta}$$

Furthermore, in $TTCN_{\Delta}$ components get another state called *wait*. This status allows a component to wait after completing all necessary actions in a sampling step to the end of the sampling step.

$$status_{\Delta} : Component_{\Delta} \rightarrow \{running,\ inactive,\ stopped,\ killed\}$$

The main differences between stream ports and message or procedure ports is sampling on one hand side and the ability to access all messages sent

or received a port randomly. This feature is realized by extending the queue related function with additional list related functions that allow an indexed based access to the messages that have been communicated through a port.

$$[] : Message^* \times Value \rightarrow Message, \text{ with}$$
$$\langle m_1, m_2, .., m_n \rangle [i] \mapsto m_{n-i}$$

$$@ : Message^* \times Value \rightarrow Message, \text{ with}$$
$$\langle m_1, m_2, .., m_n \rangle @t \mapsto m_x, \text{ with}$$
$$t \in [0..t_{sample}] \wedge x \in [0..n], \text{ where}$$
$$\forall i \in [0..n], \ (ts(m_i) - t) \ < \ (ts(m_x) - t) \rightarrow ts(m_i) > t$$

B.2.5 $TTCN_{\mathcal{E}}$ signatures

$TTCN_{\mathcal{E}}$ contains all features of *TTCN-3 embedded*. In addition to $TTCN_{\Delta}$ it allows to change simultaneous update and evaluation of stream values at ports. With the implementation of hybrid automaton like control flow constructs $TTCN_{\mathcal{E}}$ allows to define multiple execution path using predicate on value at ports to describe alternative stimulation and assessment for sampled (i.e. time discrete) streams.

Modes are distinguished on a conceptual level in atomic modes, parallel modes and sequential mode). During runtime this is

$$Mode = AtomicMode \cup ParMode \cup SeqMode$$

Regardless of the kind of mode, the TE provides a structure called $runtime \in Runtime$ to store run time information for each mode that is under execution. The information is maintained for each component separately. This allows the execution of functions and testcases on different components that potentially refer to the same mode definitions. Thus, there is a set $Runtime_c \subset Runtime$ for each component $c \in Component$.

$$\begin{aligned} status &: Runtime_c \to \{init, running, stopped\} \\ subtask &: Runtime_c \to \{invariants, guards, body, finalize\} \\ mode &: Runtime_c \to Mode \\ stime &: Runtime_c \to Value^{float} \\ trans &: Runtime_c \to PHRASE \end{aligned}$$

$$\begin{aligned} runtime &: Component \times Mode \to Runtime_c, \text{ with} \\ &runtime(component, mode) \mapsto ms \in Runtime_c \\ &\text{where } ms.mode \ = \ mode \end{aligned}$$

The *runtime* structure provides a set of properties that allows a fine grained control of the mode execution in the context of sampling. The properties are available by means of a set of function. The function *status* returns the current runtime status of a mode. It can either be *init* when the mode is initialized, *running* when the mode is executing its contents and *stopped*

when the execution has ended. Moreover, for each mode there exist an execution sub task, which can be obtained by means of the function $subTask$. The sub task of mode defines which part of a mode is currently under execution. It can vary between the values $invariants$, $guards$, $body$, and $wait$. The function $mode$ provides a link to the AST node, for which the status is maintained. It can be used to look up the mode state for a concrete mode during execution. Last but not least, the runtime structure maintains the point in time when the associated mode has been initialized. The time value is provided by the function $stime$.

The $loop$ function complements the already existing functions $next$ and fst and introduces a third control flow option, which is taken if a mode is in a sampling loop. The function is used to denote the $phrase$ that has to be executed in case that a mode repeats its content as the preparation for the next sample step. If not defined otherwise, the function points to the phrase that is given as a parameter, thus normally

$$loop : PHRASE \rightarrow PHRASE$$
$$loop(phrase) := phrase$$

Last but not least, a reactivation flag is needed to indicate the reactivation for already stopped modes. This is necessary to enable transitions between modes. The reactivation flag is located on component level.

$$reactivateNext : Component \rightarrow \mathbb{B}$$

B.3 Rules

B.3.1 $TTCN_{\mathcal{I}}$: The imperative core of TTCN-3

Rule 1: ASM transition rule for literals.

if *task* **is** *lit* **then** ▷ (literal)
  $val(lit) = \widetilde{lit}$
  proceed
end if

Rule 2: ASM transition rule for variables.

if *task* **is** *var* **then** ▷ (variable)
  $val(var) := loc(var)$
  proceed
end if

Rule 3: ASM transition rule for constants.

if *task* **is** *const* **then** ▷ (constant)
  $val(const) := loc(const)$
  proceed
end if

Rule 4: ASM transition rule for unary operators.

if *task* **is** *uop* **then** ▷ (unary operator)
  $val(uop) := \widetilde{\circledcirc}\ val(exp)$
  proceed
end if

Rule 5: ASM transition rule for binary operators.

if *task* **is** *bop* **then** ▷ (binary operator)
  $val(bop) := val(exp_1)\ \widetilde{\odot}\ val(exp)$
  proceed
end if

Rule 6: ASM transition rule for log statements.

if *task* **is** $functionInstance$ ▷ (function instance)
$\wedge\ kind(functionInstance) \in \{ext, \widetilde{pre}\}$ **then**
  $val(functionInstance) := \widetilde{functionId}(exp_1, exp_2, .., exp_n)$
  proceed
end if

Rule 7: ASM transition rule for assignments.

if *task* **is** *assignment* **then** ▷ (assignment)
  $loc(var) := val(exp)$
  proceed
end if

Rule 8: ASM transition rule for log statements.

if *task* **is** *logStatement* **then** ▷ (log statement)
  proceed
end if

Rule 9: ASM transition rule for for statements.

if *task* **is** $forStatement$ **then** ▷ (for statement)
  if $val(exp)$ **then**
    $task := fst(statementBlock)$
  else
    $task := next(forStatment)$
  end if
end if

Rule 10: ASM transition rule for while statements.

if *task* **is** *whileStatement* **then** ▷ (while statement)
  if $val(exp)$ **then**
    $task := fst(statementBlock)$
  else
    $task := next(whileStatment)$
  end if
end if

Rule 11: ASM transition rule for do-while statements.

if *task* **is** *dowhileStatement* **then** ▷ (do-while statement)
  if $val(exp)$ **then**
    $task := fst(statementBlock)$
  else
    $task := next(whileStatment)$
  end if
end if

Rule 12: ASM transition rule for if statements.

if *task* **is** $ifStatement$ **then** ▷ (if statement)
  if $val(exp)$ **then**
    $task := fst(statementBlock_1)$
  else
    $task := fst(statementBlock_2)$

end if
end if

Rule 13: ASM transition rule for goto statements.

if *task* **is** *gotoStatement* **then** ▷ (goto statement)
proceed
end if

B.3.2 $TTCN_{\mathcal{C}}$: TTCN-3 with testcases, functions and components

Rule 14: ASM transition rule for the module control.

if *task* **is** $TTCN3Module$ **then** ▷ (module control)
extend *Component* **by** *control*
$initControl(control)$
$task := stmt_1$
end extend
end if

Rule 15: ASM transition rule for the end of control part.

if *task* **is** $finished$ ▷ (end of control part)
$\wedge length(task_{\mathcal{C}}) = 1 \wedge type = control$ **then**
$status := stopped$
end if

Rule 16: ASM transition rule for the start of testcase instances.

if *task* **is** $testcaseInstance$ **then** ▷ (testcase instance, start mode)
$startTest(\langle val(exp_1), ..., val(exp_n)\rangle, args, fst(body), cRef)$
where $(args, body, cRef) := lookupTest(testcaseInstance)$
$status := inactive$
proceed
end if

Rule 17: ASM transition rule for the return of testcase instances.

if *task* **is** $finished$ ▷ (testcase instance)
$\wedge length(task_{\mathcal{C}}) = 1 \wedge type = MTC$ **then**
$startControl$
end if

Rule 18: ASM transition rule for the start of function instances.

if *task* **is** $functionInstance$ ▷ (function instance, start mode)
$\wedge\ kind(functionInstance) \in \{def\}$ **then**

$startF(\langle val(exp_1), ..., val(exp_n)\rangle, args, fst(body), frames)$
where $(args, body) := lookupF(functionInstance)$
end if

Rule 19: ASM transition rule for the simple return of function instances.

if $task$ **is** $finished \wedge length(task_{\mathcal{C}}) > 1$ **then** ▷ (simple return)
$return(frames)$
end if

Rule 20: ASM transition rule for return statements.

if $task$ **is** $returnStatement$ **then** ▷ (return statement)
if $exp = undef$ **then**
$return(frames)$
else
$result(frames, val(exp))$
end if
end if

Rule 21: ASM transition rule for the creation of test components.

if $task$ **is** $createOP$ **then** ▷ (create operation)
extend $Component$ **by** c
$init(inits, vars, c)$
where $((c, ports, vars, inits)) := lookup(cRef)$
$val(createOP) := c$
if alive = undef **then**
$alive(c) := false$
else
$alive(c) := true$
end if
end extend
$proceed$
end if

Rule 22: ASM transition rule for the start of test components.

if $task$ **is** $startTCStatement$ **then** ▷ (start test component)
$startF(\langle val(exp_1), ..., val(exp_n)\rangle, args, fst(body), frames(val(var)))$
where $(args, body) := lookupF(functionInstance)$
$status(val(var)) := running$
$proceed$
end if

Rule 23: ASM transition rule for the killing test components.

if $task$ **is** $killTCStatement$ **then** ▷ (kill test component)

```
    if type(val(var)) = mtc then
        var c rangesover Component
            killTC(c)
        end var
    else
        killTC(val(var))
    end if
    proceed
end if
```

Rule 24: ASM transition rule for the stop of a test components.

```
if task is stopTCStatement then                    ▷ (stop test component)
    if type(val(var)) = mtc then
        var c rangesover Component
            stopTC(c)
        end var
    else
        stopTC(val(var))
    end if
    proceed
end if
```

B.3.3 $TTCN_{\mathcal{T}}$: TTCN-3 with messages, alt statements, altsteps and timers

Rule 25: ASM transition rule for the start of timers.

```
if task is startTimerStatement then                    ▷ (start timer)
    startTimer(timer, exp)
    where (timer) := lookupTimer(val(var), Self)
    proceed
end if
```

Rule 26: ASM transition rule for the stop of timers.

```
if task is stopTimerStatement then                     ▷ (stop timer)
    stopTimer(timer)
    where (timer) := lookupTimer(val(var), Self)
    proceed
end if
```

Rule 27: ASM transition rule for read timers.

```
if task is readTimerOperation then                     ▷ (read timer)
    if status(Self) = snapshot then
```

$val(readTimerOperation) := val'(timer)$
else
$val(readTimerOperation) := val(timer)$
end if
where $(timer) := lookupTimer(val(var), Self)$
$proceed$
end if

Rule 28: ASM transition rule for send statements.

if $task$ **is** $sendStatement$ **then** ▷ (send statement)
$sendMessage(exp, port, adresses)$
where $((port)) := lookup(val(portVar))$
$proceed$
end if

Rule 29: ASM transition rule for alt statements.

if $task$ **is** $altConstruct$ **then** ▷ (alt statement)
$takeSnapshot(Self)$
$task := fst(stmt_1)$
end if

Rule 30: ASM transition rule for guard operations.

if $task$ **is** $guardStatement$ ▷ (guard statement, guard operation)
$\wedge\ guardStatement = (exp,\ guardOp,\ statementBlock)$ **then**
if $val(exp)$ **then**
$task := guardOp$
else
$task := nxt(guardStatment)$
end if
end if

Rule 31: ASM transition rule for alt steps.

if $task$ **is** $guardStatement$ ▷ (guard statement, alt step)
$\wedge\ guardStatement \in EXP \times AltStepInstance \times StatementBlock$ **then**
if $val(exp)$ **then**
$task := fst(altStepInstance)$
else
$task := nxt(guardStatment)$
end if
end if

Rule 32: ASM transition rule for alt step instances.

if $task$ **is** $altStepInstance$ **then** ▷ (alt step instance)

$startF(\langle val(exp_1), ..., val(exp_n)\rangle, args, fst(body), frames)$
where $(args, body) := lookupF(altSetpInstance)$
end if

Rule 33: ASM transition rule for receive statements.

if *task* **is** *receiveStatement* **then** ▷ (receive statement)
  if $match(port, val(exp)) \wedge sender(top(msgs'(port))) \in fromSpec$ **then**
    $handleRedirect(top(msgs'(port)), redirectSpec)$
    $pop(msgs'(port))$
    $task := nxt(receiveStatement)$
  else
    $task := nxt(up(receiveStatement))$
  end if
  where $port := lookup(val(var))$
end if

Rule 34: ASM transition rule for timeout statements.

if *task* **is** *timeoutStatement* **then** ▷ (timeout statement)
  if $status'(timer) = expired$ **then**
    $task := nxt(timeoutStatement)$
  else
    $task := nxt(up(timeoutStatement))$
  end if
  where $((timer)) := lookup(val(var))$
end if

B.3.4 $TTCN_\Delta$: TTCN-3 with streams, global time and sampling

Rule 35: ASM transition rule for the test execution controller.

if $time \geq t_{sample} + \delta$ **then** ▷ (test execution controller)
  $t'_{sample} = t_{sample}$
  $t_{sample} = t_{sample} + \delta$
  for all $c \in Component$ **do**
    $takeSnapshot(c)$
    $updateStreams(c)$
    $status(c) := running$
  end for
end if

Rule 36: ASM transition rule for wait statements.

if *task* **is** *waitStatement* **then** ▷ (wait statement)

if $val(exp) < t'_{sample} \wedge status(Self) = running$ **then**
  $verdict(Self) := error$
end if
if $val(exp) > t'_{sample}$ **then**
  $status(Self) := wait$
else
  $status(Self) := running$
  $proceed$
end if
end if

Rule 37: ASM transition rule for now operations.

if $task$ **is** $nowOperation$ **then** ▷ (now-expression)
  $val(nowOperation) := t'_{sample}$
  $proceed$
end if

Rule 38: ASM transition rule for value operations.

if $task$ **is** $streamValueOperation$ **then** ▷ (value operation)
  $val(streamValueOperation) := value(msgs'(port)([0])$
  where $port := lookup(val(var))$
end if

Rule 39: ASM transition rule for timestamp operations.

if $task$ **is** $streamTimestampOperation$ **then** ▷ (timestamp operation)
  $val(streamTimestampOperation) := ts(msgs'(port)[0])$
  where $port := lookup(val(var))$
end if

Rule 40: ASM transition rule for delta operations.

if $task$ **is** $streamDeltaOperation$ **then** ▷ (delta operation)
  $val(streamDeltaOperation) := delta(port)$
  where $port := lookup(val(var))$
end if

Rule 41: ASM transition rule for assignments (refines assignment rule).

if $task$ **is** $assignment$ **then** ▷ (assignment)
  if $leftSide$ **is** var **then**
    $loc(var) := val(exp)$
  end if
  if $leftSide$ **is** $streamValueOp$ **then**
    $currentVal(port) := val(exp)$
    where $port := lookup(val(pvar))$

```
    end if
    if leftSide is streamDeltaOp then
        delta(port) := val(exp)
        where  port := lookup(val(pvar))
    end if
    proceed
  end if
```

Rule 42: ASM transition rule for prev operations.

```
  if task is prevOp then                                   ▷ (prev operation)
    if val(exp) ≥ len is valueOp then
        val(exp) := len(msgs′(port))
    end if
    if streamDataOp is valueOp then
        val(prevOp) := value(msgs′(port)[val(exp)])
    else if streamDataOp is timestampOp then
        val(prevOp) := ts(msgs′(port)[val(exp)])
    else if streamDataOp is deltaOp then
        val(prevOp) :=
            ts(msgs′(port)[val(exp)]) − ts(msgs′(port)[val(exp) + 1])
    end if
  end if
  where  port := lookup(val(var))
```

Rule 43: ASM transition rule for at operations.

```
  if task is atOp then                                     ▷ (at operation)
    if val(exp) > t′sample then
        verdict(Self) := error
        val(exp) := ts(current(port))
    end if
    if val(exp) < ts(msgs′(port)[len(msgs′(port))]) then
        verdict(Self) := error
        val(exp) := ts(msgs′(port)[len(msgs′(port))])
    end if
    if streamDataOp is valueOp then
        val(atOp) := value(msgs′(port)@(val(exp)))
    else if streamDataOp is timestampOp then
        val(atOp) := ts(msgs′(port)@(val(exp)))
    else if streamDataOp is deltaOp then
        val(atOp) :=
            ts(msgs′(port)@(val(exp))) − ts(pred(msgs′(port)@(val(exp))))
    end if
  end if
```

where $port := lookup(val(var))$

Rule 21: ASM transition rule for the history operation.

if $task$ **is** $streamHistoryOp$ **then** ▷ (history operation)
  if $val(exp_1) > t'_{sample}$ **then**
    $verdict(Self) := error$
    $val(exp_1) := ts(current(port))$
  end if
  if $val(exp_2) < ts(msgs'(port_2)[len(msgs'(port))])$ **then**
    $verdict(Self) := error$
    $val(exp_2) := ts(msgs'(port)[len(msgs'(port))])$
  end if
  if $val(exp_1) > val(exp_2)$ **then**
    $verdict(Self) := error$
    $val(streamHistoryOp) := \langle\rangle$
  else
    $val(streamHistoryOp) := \langle m_x..m_y \rangle, textwith$
      $m_x = msgs'(port)@val(exp_1) \ \wedge m_y = msgs'(port)@val(exp_2)$
  end if
end if
where $port := lookup(val(var))$

B.3.5 $TTCN_{\mathcal{E}}$: TTCN-3 with modes

Rule 44: ASM transition rule for mode execution (entering a mode).

if $task$ **is** $mode$ ▷ (mode execution ,initialization)
  $\wedge\ runtime(Self, mode) = undef$ **then**
  extend $Runtimes_{Self}$ **by** $runtime$
    $status(runtime) := init$
  end extend
end if

Rule 45: ASM transition rule for mode execution (first loop).

if $task$ **is** $mode$ ▷ (mode execution, first loop)
  $\wedge\ status(runtime(self, mode)) = init$ **then**
  $reactivateNext(Self) := false$
  if $isViolated(invariantList)$ **then**
    $verdict(Self) := error$
    $status(modeState(Self, mode)) = stopped$

```
        task := nxt(mode)
    else
        subtask(runtime(self, mode)) := body
        mode(runtime(self, mode)) := mode
        starttime(runtime(self, mode)) := t'_sample
        trans(runtime(self, mode)) := nxt(mode)
        status(runtime(self, mode)) = run
        task := onEntry
    end if
end if
```

Rule 46: ASM transition rule for mode execution (repeating).

```
if task is mode                                   ▷ (mode execution, repeating)
    ∧ status(runtime(Self, mode)) = run then
    if subtask(runtime(Self, mode)) = invariants then
        task := fst(invariantList)
        subtask(runtime(Self, mode)) = guards
    else if subtask(runtime(Self, mode)) = guards then
        if isViolated(invariantList) then
            verdict(Self) := error
            status(modeState(Self, mode)) = stopped
            task := nxt(mode)
        else
            task := fst(until)
            subtask(mode) = body
        end if
    else if subtask(runtime(Self, mode)) = body then
        subtask(mode) = finalize
        if endOfChilds then
            task := onExit
        else
            task := fst(contBody)
        end if
    else if subtask(runtime(Self, mode)) = finalize then
        if endOfStepExecution then
            resetModes(Self)
            status(Self) := finalize
        else
            task := up(up(mode))
        end if
    end if
end if
```

Rule 47: ASM transition rule for mode execution (stopped mode).

if *task* **is** *mode* ▷ (mode execution, stopped mode)
  $\wedge$ $status(runtime(Self, mode)) = stopped$ **then**
  if reactivateNext(Self) **then**
    $reactivateNext(Self) := false$
    $status(runtime(Self, mode)) := init$
  else
    proceed
  end if
end if

Rule 48: ASM transition rule for until-guard statements.

if *task* **is** *untilGuardStatement* **then** ▷ (until-guard statement)
  if $val(exp)$ **then**
    if $guardOp = undef$ **then**
      $task := statementBlock$
    else
      $task := guardOp$
    end if
  else
    $task := nxt(guardStatement)$
  end if
end if

Rule 49: ASM transition rule for goto statements in modes.

if *task* **is** $GotoTransition$ **then** ▷ (goto statement in modes)
  $reactivateNext(Self) := true$
  $trans(runtime(up(up(up(RepeatMode))), Self)) := nxt(gotoTransition)$
end if

Rule 50: ASM transition rule for repeat statements in modes.

if *task* **is** $RepeatTransiton$ **then** ▷ (repeat statement in modes)
  $reactivateNext(Self) := true$
  $trans(runtime(up(up(up(RepeatMode))), Self)) := up(up(up(RepeatMode)))$
end if

Rule 51: ASM transition rule for continue statements in modes.

if *task* **is** $ContinueMode$ **then** ▷ (continue statement in modes)
  $task := up(up(up(ContinueMode)))$
end if

Rule 52: ASM transition rule for on-exit statements.

if $task$ **is** $onExit$ **then** ▷ (on-exit statement)
  $status(runtime(Self, mode)) := stopped$
  $task := folowUp(runtime(Self, mode))$
end if

Rule 53: ASM transition rule for duration-expressions in modes.

if $task$ **is** $duration$ **then** ▷ (duration-expression)
  $val(duration) := t'_{sample} - starttime(runtime(parent(duration), Self)$
  $proceed$
end if

B.4 MACROS

B.4.1 $TTCN_{\mathcal{I}}$ macros

$proceed \equiv task := nxt(task)$

$task$ **is** $phrase \equiv task = phrase \ \wedge \ phrase \in PHRASE$

B.4.2 $TTCN_{\mathcal{C}}$ macros

$task \equiv top(task_{\mathcal{C}}(Self))$

$loc \equiv top(loc_{\mathcal{C}}(Self))$

$val \equiv top(val_{\mathcal{C}}(Self))$

$frames(c) \equiv (task_{\mathcal{C}}(c), loc_{\mathcal{C}}(c), val_{\mathcal{C}}(c))$

$task$ **is** $phrase \equiv$
$task = phrase \wedge phrase \in PHRASE \wedge status \notin \{inactive, stopped, killed\}$

$startTC(\langle val_1, ..., val_n\rangle, \langle arg_1, ..., arg_n\rangle, tcRef, phrase) \equiv$
extend $Component$ **by** c
$init(inits, vars, c)$
where $((c, ports, vars, inits)) := lookup(cRef)$
var i **rangesover** $(1..n)$
$(loc_{\mathcal{C}}(c))(arg_i) := (val_{\mathcal{C}}(c))(val_i)$
end var
$status(c) := running$
$type(c) := mtc$
end extend

$init(\langle val_1, ..., val_n\rangle, \langle var_1, ..., var_n\rangle, c) \equiv$
$frames(c) := (\langle\emptyset\rangle, \langle\emptyset\rangle, \langle\{(var_1, val_1), ..., (var_n, val_n)\}\rangle)$

$initControl(control) \equiv$

$verdict(control) := none$
$status(control) := running$
$type(control) := control$

$startControl \equiv$
choose c **in** $\{c \mid type(c) = control \wedge c \in Component\}$
$status(c) := running$
end choose

$startF(\langle val_1, ..., val_n\rangle, \langle arg_1, ..., arg_n\rangle, phrase, (tasks, vals, locs)) \equiv$
$frames := (\langle phrase\rangle\ tasks, \langle\emptyset\rangle\ vals, \langle\{(arg_1, val_1), ..., (arg_n, val_n)\}\rangle\ locs)$

$return(\langle ?, inv\rangle\ tasks, \langle ?\rangle\ env, \langle ?\rangle\ vals) \equiv$
$frames := (\langle next(inv)\rangle\ tasks,\ env,\ vals)$

$result((\langle ?, inv\rangle\ tasks, \langle ?\rangle\ env, \langle ?, val\rangle\ vals), res) \equiv$
$frames := (\langle next(inv)\rangle\ tasks,\ env,\ \langle val[inv/res]\rangle\ vals)$

$init(\langle val_1, ..., val_n\rangle, \langle var_1, ..., var_n\rangle, c) \equiv$
$frames := (\langle\emptyset\rangle, \langle\emptyset\rangle, \langle\{(var_1, val_1), ..., (var_n, val_n)\}\rangle)$
$status(c) := inactive$

$killTC(c) \equiv$
$status(c) := killed$

$stopTC(c) \equiv$
if alive(c) **then**
$status(c) := stopped$
else
$killTC(c)$
end if

$startTimer(timer, exp) \equiv$
$timeoutval(timer) := val(exp)$
$val(timer) := 0.0$
$status(timer) := running$

```
stopTimer(timer) ≡
    status(timer) := inactive


sendMesssage(m, port, addresses) ≡
    if mapsto(port) = undef then
        setverdict(error)
    else
        var r rangesover mapsto(port)
            extend Messages by m
                value(m) := val(exp)
                sender(m) := Self
                if owner(mapsto(port)) = SYSTEM then
                    triSendMessage(m, port, adresses)
                else
                    tciSendConnected(m, port, adresses)
                end if
            end extend
        end var
    end if


takeSnapshot(c) ≡
    for all p ∈ ports(c) do
        if status(p) = started then
            msgs'(p) := msgs(p)
        end if
    end for
    for all t ∈ timer(c) do
        if status(t) = running then
            status'(t) := status(t)
            value'(t) := value(t)
        end if
    end for
```

B.4.3 TTCN$_\Delta$ macros

```
updateStreams(c) ≡
    for all p ∈ ports(c) do
        if status(p) = started ∧ kind(p) = stream ∧ stepExpired(p) then
            if direction(p) = in then
                enqueue(msgs(p), triGetMessage(p))
```

end if
if $direction(p) = out$ **then**
extend $Message$ **by** m
$value(m) := value(p)$
$delta(m) := delta(p)$
$ts(m) := t'_{sample}$
$enqueue(msgs(p), m)$
end extend
end if
if $direction(p) = in$ **then**
$value(p) := value(msgs'(p)([0])$
$delta(p) := ts(msgs'(port)[0]) - ts(msgs'(port)[1])$
end if
end if
end for

$stepExpired(p) \equiv t_{sample} \geq ts(p) + delta(p)$

B.4.4 $TTCN_{\mathcal{E}}$ macros

$isViolated(invariantList) \equiv$
$\exists\, exp \in invariantList,\ val(exp) = false$

$endOfChilds(mode, c) \equiv$
$\forall\, rs \in Runtime_c,\ (mode = up(up(mode(rs)))) \rightarrow status(rs) \neq running$

$endOfStepExecution \equiv$
$\exists\, runtime \in runtimes(Self)\ ,\ subtask(runtime) \neq finalize$

$resetModes(c) \equiv$
var $runtimes$ **rangesover** $Runtime_c$
if $status(runtime) = running$ **then**
$subtask(runtime) = invariants$
end if
end var

Appendix C

Publications

The work presented in this thesis is original work undertaken at the Fraunhofer Institute for Open Communication Systems, Competence Center – Modeling and Testing for System and Service Solutions. Portions of this work have been already presented at conferences, in books and in journals and resulted in the following publications:

- J. Grossmann, *Testing hybrid systems with TTCN-3 embedded*, International Journal on Software Tools for Technology Transfer (2014), pp. 247–267.
- J. Grossmann, P. Makedonski, H.-W. Wiesbrock, J. Svacina, I. Schieferdecker, and J. Grabowski, *Model-based X-in-the-loop testing*, CRC Press, Sept. 2011.
- J. Grossmann, D. A. Serbanescu, and I. Schieferdecker, *Testing Embedded Real Time Systems with TTCN-3*, in ICST 2009, IEEE Computer Society, pp. 81–90.
- J. Grossmann, I. Schieferdecker, and H.-W. Wiesbrock, *Modeling property based stream templates with TTCN-3*, in TestCom/FATES, K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, eds., Lecture Notes in Computer Science 5047, Springer, 2008, pp. 70–85.
- J. Grossmann and I. Schieferdecker, *Testing Hybrid Control Systems with TTCN-3*, International Journal on Software Tools for Technology Transfer (2008), pp. 383–400.

Moreover the work of this thesis constituted the project deliverable *TEMEA Deliverable D2.4 concepts for the specification of tests for systems with continuous or hybrid behaviour* that later on was used as a basis for an European Standard at the ETSI.

- ETSI: ES 202 786 V1.1.1, *Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, TTCN-3 Language Extensions: Support of interfaces with continuous signals*, Febr. 2012.