TTCN-3 – A Test Technology for the Automotive Domain

Ina Schieferdecker, Axel Rennoch, Edzard Höfig

Abstract

TTCN (Testing and Test Control Notation) is in the telecommunication domain a widely established and used test technology. In its new version, TTCN-3 has a wider scope and applicability. It cannot be used only for testing the conformance and interoperability of communication protocols but also the interaction of e.g. sensors, actuators and control units connected via bus systems. Therefore, TTCN-3 is now being used in other domains such as automotive, railways, avionics and security systems. This paper gives an overview on TTCN-3, describes basic concepts and presents results on the application of TTCN-3 for testing MOST based components.

1. Introduction

Car manufacturers face the problem of third-party components that need to adhere to OEM requirements and interface specifications and that need to behave functionally correct and with a sufficient performance and scalability. Component testing provides confidence and quality assurance to the OEM. However currently, it is hard to precisely define acceptance criteria for component deliveries as the artefacts used along this process are not integrated, give no a direct way to derive component acceptance tests from component requirements specifications, and do not allow to specify the test procedures being applied in the approval phase. The situation deteriorates with the increasing complexity and flexibility of today's automobiles. New approaches for a systematic handling of automotive components address this problem by

- defining automotive system/software architectures and platforms such as AutoSar [10]
- extending process models towards a precise interface between third-party component vendor and OEM such as the W-model [11] or QDSD [12]
- applying model-based techniques for the specification and testing of systems and components along MDA [13][16], UML [14] or U2TP [15].

This paper considers in particular the systematic testing of components of MOST [7] based applications. We use the only international standardized approach – the Testing and Test Control Notation TTCN-3 [4] – due to its wide acceptance, applicability and available tool support. One of the advantages of TTCN-3 is the availability of both standardized open runtime and control interfaces (TRI and TCI) that enable an exchange of tool components between different test tool providers. We use the TTthree tool chain [6] because it is comprehensive w.r.t. management and logging facilities and has a good portability.
The paper is structured as follows: TTCN-3 is introduced in Section 2. Since MOST messages and interfaces are defined in XML, Section 3 presents basic concepts of the combined use of XML and TTCN-3. Section 4 discusses the application of TTCN-3 to the testing of MOST components. An outlook completes the paper.

2. The TTCN-3 Test Technology

The TTCN-3 language was created due to the imperative necessity to have a universally understood (specification and implementation) language syntax able to describe test behaviours and test procedures. Its development was imposed by industry and science to obtain a single test notation for all black-box and grey-box testing needs. In contrast to earlier test technologies, TTCN-3 encourages the use of a common methodology and style which leads to a simpler maintenance of test suites and products. With the help of TTCN-3, the tester specifies the test suites at an abstract level and focuses on the test logic to check a test purpose itself rather than on the test system adaptation and execution details. A standardized language provides a lot of advantages to both test suite providers and users. Moreover, the use of a standard language reduces the costs for education and training, as a great amount of documentation, examples, and predefined test suites are available. It is obviously preferred to use always the same language for testing than learning different technologies for distinct testing kinds. Constant use and collaboration between TTCN-3 vendors and users ensure a uniform maintenance and development of the language.

TTCN-3 enables systematic, specification-based testing for various kinds of tests including e.g. functional, scalability, load, interoperability, robustness, regression, system and integration testing. TTCN-3 is a language to define test procedures to be used for black-box and grey-box testing of distributed systems. It allows an easy and efficient description of complex distributed test behaviours in terms of sequences, alternatives, and loops of stimuli and responses. The test system can use a number of test components to perform test procedures in parallel. TTCN-3 language is characterized by a well-defined syntax and operational semantics, which allow a precise execution algorithm. The task of describing dynamic and concurrent configurations is easy to perform. The communication can be realized either synchronously or asynchronously. To validate the data transmitted between the entities composing the test system, TTCN-3 supports definition of templates with a powerful matching mechanism representing test data. To validate the described behaviours, a verdict handling mechanism is provided. The types and values can be either described directly in TTCN-3 or can be imported from other languages (e.g. ASN.1, XML schema, or IDL). Moreover in TTCN-3, the parameterization of types and values is allowed. The selection of the test cases to be executed can be either controlled by the user or can be described within the execution control construct. The external configuration of a test suite through module parameters is possible.

Figure 1 shows an overview of the TTCN-3 language. The TTCN-3 meta-model defines the concept space of TTCN-3 [4]. TTCN-3 has a core language for the textual definition and two other presentation formats, which allow the graphical and tabular definition of test artefacts. TTCN-3 provides interfaces to reference data defined in other description languages. As the figure shows, one can import types and values specified in ASN.1, but other formats are also supported (IDL, XML schema etc).

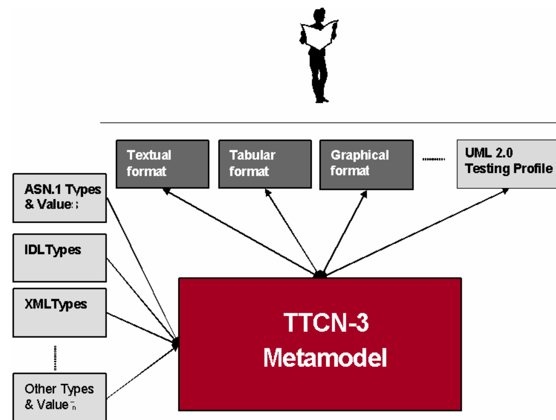

Figure 1: The TTCN-3 Language Architecture

The ETSI standard for TTCN-3 comprises currently seven parts (described below) which are grouped together in the “Methods for Testing and Specification; The Testing and Test Control Notation version 3” document [3]:

1. TTCN-3 Core Language. This document specifies the syntax of TTCN-3 language.
2. Tabular Presentation Format. TTCN-3 offers optional presentation formats. The tabular format is similar in appearance and functionality to earlier versions of TTCN. It was designed for users that prefer the TTCN-2 style of writing test suites. A TTCN-3 module is presented in the tabular format as a collection of tables.
3. Graphical Presentation Format. It is the second presentation format of TTCN-3 and is based on the MSC format (Message Sequence Charts). The graphical format is used to represent graphically the TTCN-3 behaviour definitions as a sequence of diagrams.
4. Operational semantics. This document describes the meaning of TTCN-3 behaviour constructs and provides a state oriented view of the execution of a TTCN-3 module.
5. The TTCN-3 Runtime Interface (TRI). A complete test system implementation requires also a platform specific adaptation layer. The TRI document contains the specification of a common API interface to adapt TTCN-3 test systems to SUT.
6. The TTCN-3 Control Interfaces (TCI). This part provides an implementation guideline for the execution environments of TTCN-3. It contains the specification of the API the TTCN-3 execution environments should implement in order to ensure the communication, management, component handling, external data control and logging.
7. Use of ASN.1 in TTCN-3: This part provides guidelines and mappings rules for the combined use of ASN.1 (Abstract Syntax Notation One) and TTCN-specifications

Additional parts of the standards for the language mappings are currently under development.

2.1 The Concepts of TTCN-3

The TTCN-3 core language is a modular language and has a similar look and feel to a typical programming language. In addition to the typical programming constructs, it contains all the important features necessary to specify test procedures and campaigns for functional, conformance, interoperability, load and scalability tests like test verdicts, matching mechanisms to compare the reactions of the SUT with the expected range of values, timer handling, distributed test components, ability to specify encoding information, synchronous and asynchronous communication, and monitoring. A TTCN-3 test specification consists of imports from other modules; types, test data and templates definition, function, altstep and test case definitions for test behaviour; and control definitions for the execution of test cases (see Fig. 2).
Modules

Figure 2: TTCN-3 Module Structure

The top-level building-block of TTCN-3 is a module. A module contains all other TTCN-3 constructs, but cannot contain sub-modules. It can also import completely or partially the definitions of other modules. The modules are defined with the keyword module. The modules can be parameterized; parameters are sets of values that are supplied by the test environment at runtime. A parameter can be initialized with a default value.
A TTCN-3 module has two parts: the module definition part and the module control part. The definition part contains the data defined by that module (functions, test cases, components, types, templates), which can be used everywhere in the module and can be imported from other modules. The control part is the main program of the module, which describes the execution sequence of the test cases or functions. It can access the verdicts delivered by test cases and, according to them, can decide the next steps of execution. The test behaviours in TTCN-3 are defined within functions, altsteps and testcases. The control part of a module may call any testcase or function defined in the module to which it belongs.

2.2 Test System

A test case is executed by a test system. TTCN-3 allows the specification of dynamic and concurrent test systems. A test system consists of a set of interconnected test components with well-defined communication ports and an explicit test system interface, which defines the boundaries of the test system.
Within every test system, there is one Main Test Component (MTC). All other test components are called Parallel Test Components (PTCs). The MTC is created and started automatically at the beginning of each test case execution. A test case terminates when the MTC terminates, which implies also the termination of all other

PTCs. The behaviour of the MTC is specified in the body of the test case definition. During the execution of a test case, PTCs can be created, started and stopped dynamically. A test component may stop itself or can be stopped by another test component.

For communication purposes, each test component owns a set of local ports. Each port has an in- and an out-direction. The in-direction is modelled as an infinite FIFO queue, which stores the incoming information until it is processed by the test component owning the port. The out-direction is directly linked to the communication partner (another test component or the system under test (SUT)), i.e., outgoing information is not buffered.

During test execution, TTCN-3 distinguishes between connected and mapped ports. Connected ports are used for the communication with other test components. If two ports are connected, the in-direction of one port is linked to the out-direction of the other, and vice versa. A mapped port is used for the communication with the SUT. The mapping of a port owned by a test component to a port in the abstract test system interface can be seen as pure name translation defining how communication streams should be referenced. TTCN-3 distinguishes between the abstract and the real test system interface. The abstract test system interface is modelled as a collection of ports that defines the abstract interface to the SUT. The real test system interface is the application specific part of a TTCN-3-based test environment. It implements the real interface of the SUT and is defined in the TTCN-3 runtime interface (TRI, part 5 of TTCN-3).

In TTCN-3, connections and mappings are created and destroyed dynamically at runtime. There are no restrictions on the number of connections and mappings a component may have. A component (and even a port) may be connected to itself. One-to-many connections are allowed, but TTCN-3 only supports one-to-one communication, i.e., during test execution the communication partner has to be specified uniquely. For the communication among test components and between test components and the SUT, TTCN-3 supports message-based and procedure-based communication. Message-based communication is based on an asynchronous message exchange and the principle of procedure-based communication is to call procedures in remote entities.

2.3 Test Cases and Test Verdicts

Test cases define test behaviours which have to be executed to check whether the SUT passes the test or not. Like a module, a test case is considered to be a self-contained and complete specification of a test procedure that checks a given test purpose. The result of a test case execution is a test verdict.

TTCN-3 provides a special test verdict mechanism for the interpretation of test runs. This mechanism is implemented by a set of predefined verdicts, local and global test verdicts and operations for reading and setting local test verdicts. The predefined verdicts are pass, inconc, fail, error and none. They can be used for the judgment of complete and partial test runs. A pass verdict denotes that the SUT behaves according to the test purpose, a fail indicates that the SUT violates its specification. An inconc (inconclusive) describes a situation where neither a pass nor a fail can be assigned. The verdict error indicates an error in the test devices. The verdict none is the initial value for local and global test verdicts, i.e., no other verdict has been assigned yet. During test execution, each test component maintains its own local test

verdict. A local test verdict is an object that is instantiated automatically for each test component at the time of component creation. A test component can retrieve and set its local verdict. The verdict error is not allowed to be set by a test component. It is set automatically by the TTCN-3 run-time environment, if an error in the test equipment occurs. When changing the value of a local test verdict, special overwriting rules are applied. The overwriting rules only allow that a test verdict becomes worse, e.g., a pass may change to inconc or fail, but a fail cannot change to a pass or inconc.

In addition to the local test verdicts, the TTCN-3 run-time environment maintains a global test verdict for each test case. The global test verdict is not accessible for the test components. It is updated according to the overwriting rules when a test component terminates. The final global test verdict is returned to the module control part when the test case terminates.

2.4 Alternatives and Snapshots

A special feature of the TTCN-3 semantics is the snapshot. Snapshots are related to the behaviour of components. They are needed for the branching of behaviour due to the occurrence of timeouts, the termination of test components and the reception of messages, procedure calls, procedure replies or exceptions. In TTCN-3, this branching is defined by means of alt statements.

An alt statement describes an ordered set of alternatives, i.e., an ordered set of alternative branches of behaviour. Each alternative has a guard. A guard consists of several preconditions, which may refer to the values of variables, the status of timers, the contents of port queues and the identifiers of components, ports and timers. The same precondition can be used in different guards. An alternative becomes executable, if the corresponding guard is fulfilled. If several alternatives are executable, the first executable alternative in the list of alternatives will be executed. If no alternative becomes executable, the alt statement will be executed again.

The evaluation of several guards needs some time. During that time, preconditions may change dynamically. This will lead to inconsistent guard evaluations, if a precondition is verified several times in different guards. TTCN-3 avoids this problem by using snapshots. Snapshots are partial module states, which include all information necessary for the evaluation of alt statements. A snapshot is taken, i.e., recorded, when entering an alternative. For the verification of preconditions, only the information in the current snapshot is used. Thus, dynamic changes of preconditions do not influence the evaluation of guards.

2.5 Default Handling

In TTCN-3, defaults are used to handle communication events which may occur, but which do not contribute to the test objective. Default behaviour can be specified by altsteps and then activated as defaults. For each test component, the defaults, i.e., activated altsteps, are stored as a list. The defaults are listed in the order of their activation. The TTCN-3 operations activate and deactivate operate on the list of defaults. An activate operation appends a new default to the end of the list and a deactivate operation removes a default from that list.

The default mechanism is invoked at the end of each alt statement, if the default list is not empty and if due to the current snapshot none of the alternatives is executable. The default mechanism invokes the first altstep in the list of defaults and waits for the result of its termination. The termination can be successful or unsuccessful. Unsuccessful means that none of the top alternatives of the altstep defining the default behaviour is executable, successful means that one of the top alternatives has been executed.
In case of an unsuccessful termination, the default mechanism invokes the next default in the list. If the last default in the list has terminated unsuccessfully, the default mechanism will return to the alt statement, and indicates an unsuccessful default execution. Unsuccessful default execution causes the alt statement to be executed again.
In case of a successful termination, the default may either stop the test component by means of a stop statement, or the main control flow of the test component will continue immediately after the alt statement from which the default mechanism was called or the test component will execute the alt statement again. The latter has to be specified explicitly by means of a repeat statement. If the selected top alternative of the default ends without a repeat statement the control flow of the test component will continue immediately after the alt statement.
Altsteps are function-like descriptions for structuring component behaviour. TTCN-3 uses altsteps to specify default behaviour or to structure the alternatives of an alt statement. The precise semantics of altsteps is closely related to alternatives and snapshots. Like an alt statement, an altstep defines an ordered set of alternatives, the so-called top alternatives. The difference is that no snapshot is taken when entering an altstep. The evaluation of the top alternatives is based on an existing snapshot. An altstep is always called within an alt statement, which provides the required snapshot. Conceptually, the top alternatives of the altstep are inserted into the alternatives of the alt statement. Within the core language, the user can specify where the top alternatives shall be placed into the list of alternatives. It is also possible to call an altstep like a function. In this case, the altstep is interpreted like an alt statement which only invokes the altstep, i.e., the top alternatives are the only alternatives of the alt statement.

2.6 Communication Operations

Communication operations are important for the specification of test behaviours. TTCN-3 supports message-based and procedure-based communication. The communication operations can be grouped in two parts: stimuli, which send information to SUT and responses, which are used to describe the reaction of the SUT.
Procedure-based communication is synchronous communication. Procedure-based operations defined in TTCN-3 are:

- call: to invoke a remote procedure;
- getcall: to specify that a test component accepts a call from the SUT;
- reply: to reply value when an own procedure is called;
- getreply: specifies that a method is invoked;
- raise: to report an exception when an own procedure is called an something is wrong in the procedure call;
- catch: to collect an exception reported at remote procedure invocation.

Message-based communication is asynchronous communication. The sending operations are non-blocking; after sending the data, the system does not wait for response. The receive operations block the execution until a matching value is received. A receiving operation specifies a port, at which the operation takes place, defines a matching part for selection of valid receiving values and optionally specifies an address to identify the connection if the port is connected to many ports.
Message-based communication operations defined in TTCN-3 are:

- send: send a message to SUT;
- receive: receive a message from SUT;
- trigger: specifies a message that shall receive in order to go to the next statements.

2.7 Test Data Specification

A test system needs to exchange data with the SUT. The communication with the SUT can be either asynchronous, by sending/receiving messages to/from SUT or synchronous, by calling procedures of the SUT or accepting procedure calls from the SUT. In these cases, the test data must be described within the test system, according to the SUT specification. TTCN-3 offers different constructs to describe the test data: types, templates, variables, procedure signatures etc. They can be used to express any type of protocol message, service primitive, procedure invocation or exception handling. Besides this, TTCN-3 offers also the possibility to import data described in other languages (e.g. ASN.1, IDL, or XML schema).
In order to describe basic data types, TTCN-3 provides a number of predefined types. Most of these types are similar to basic types of well-known programming languages (Java, C). Some of them are only testing domain specific:

- Port types define the characteristics (message or procedure based, allowed data in the in and out direction) of ports used in the communication between test components and to the SUT.
- Component types define the properties of test components such as their ports and local variables and timers.
- The verdicttype is an enumeration which defines the possible verdicts that can be associated to a test case: pass, fail, inconc, error, none.
- The anytype is a union of all known TTCN-3 types; the instances of anytype are used as a generic object which is evaluated when the value is known.
- The default type is used for default handling and represents a reference to a default operation.

TTCN-3 also supports ordered structured types such as record, record of, set, set of, enumerated and union. Furthermore, for procedure-based communication, TTCN-3 offers the possibility to define procedure signatures. Signatures are characterized by name, optional list of parameters, optional return value and optional list of exceptions.
Templates represent test data and are data structures used to define message patterns for the data sent or received over ports. They are used either to describe distinct values that are to be transmitted over ports or to evaluate if a received value matches a template specification. Templates can be specified for any type or procedure signature. They can be parameterized, extended or reused in other template definitions. The declaration of a template contains a set of possible values. When

comparing a received message with a template, the message data shall match one of the possible values defined by the template.
As not all details of TTCN-3 can be given here, please refer to the standard, further papers and tools to get a broader overview on the technology.

3. The XML to TTCN-3 Mapping

As the need for communication has grown a lot in the last decades, also the need of structuring information has grown. Information needs to be structured in a standard manner for an easier way of creating, sending, reading, and reusing it. In 1996, XML (eXtensible Markup Language) was created by a group of SGML experts, sponsored by the World Wide Web Consortium (W3C9. In 1998, Version 1.0 of XML was approved by W3C and became a standard that nowadays is the base of a lot of service interfaces, data repositories and communication protocols. XML is a flexible tool for structuring data. Using XML, information from any vocabulary and in any structures can be restructured in an easy way. The base element in XML meta-language is the "tag". It is a kind of a label for the information that it encapsulates. XML language has a vocabulary, and also a grammar, given by the hierarchy of the tags.
In the context of automated testing, it is necessary to reuse available data type definitions of the system under test (SUT) as much as possible in order to avoid any superfluous re-definition of such data structures during test development.
As illustrated in Figure 1, TTCN-3 allows the import of types and components from other languages, e.g. ASN.1, IDL [3] and now XML schema [2]. This import mechanism makes TTCN-3 more flexible and extensible in the sense that test developers do not have to specify data types already available in other languages, but can reference and use them.

3.1 The Mapping Rules

In [1], a generic approach for XML to TTCN-3 has been developed. The main issues of this approach is the mapping of the XML built-in types to TTCN-3 types (e.g. decimal to TTCN-3 float), the consideration of XML facets (e.g. maxInclusive leads to TTCN-3 value restrictions), and the generation of structured TTCN-3 types from XML ComplexType specifications (e.g. choice is translated to TTCN-3 union).

An auxiliary TTCN-3 module XSDAUX has been introduced for the mapping of all built-in types without facets to support short references within TTCN-3. This module represents also a base for the mapping of complex data types. An extract of XSDAUX.ttcn3 is given in the following:

```
module XSDAUX  {
  /* 1.string */
  type charstring string;

  /* 2.decimal */
  type float decimal;

  /* 3.boolean */
  type integer bitXSD(1,0);
    type union booleanXSD
```

```
        {
            bitXSD bit,
            boolean bool
        }
    .....
  }
```

The structure of the mapping follows the structure of the XML schema. At first, we handle built-in data types and afterwards the additional features that XML schema offers.

Built-in data types are structured into primitive ones, and derived ones. The latter are derived from the primitive ones by means of restrictions, like: length, size, list, range etc. These restrictions are called facets. For every simpleType, primitive or derived, facets can be applied, and a new type will be available. So, for every built-in type there is a list of possible facets that can be applied to it, and depending on the facet, the mapping to TTCN-3 will be different. Every built-in type, without any facets, is mapped with its built-in name, in a module called XSDAUX. Whenever a new simpleType is defined having as base type a built-in type, it is mapped using the dot notation.

Example:

```
<xs:simpleType name="I">
<restriction base="xs:integer"/>
</xs:simpleType>
```

becomes:

```
type XSDAUX.integerXSD I;
```

After mapping the basic layer of XML schema, i.e. the built-in types, the mapping of the wrapping structures follows. For that, every structure that can appear, globally or not, will have a corresponding mapping to TTCN-3.

SimpleType components are used to define new simple types by three means: restricting a built-in type by applying a facet to it as discussed before, building lists or unions of other simple types. SimpleTypes can be defined globally, which means the parent is "schema", and the "name" attribute is mandatory, and they will be mapped to a new TTCN-3 type using that name. Hence, they can be reused in other definitions. Or, they are defined locally, i.e. the <name> attribute does not appear. Then, they are mapped by use of an automatically generated name, but they will not be reused in other definitions. Here, we give a small example:

```
<simpleType name='eRestriction'>
<restriction base='string'>
<pattern value='\d{3}-[A-Z]{2}'/>
</restriction>
</simpleType>
```

becomes:

```
type XSDAUX.string eRestriction (pattern "[0-9]#(3)-[A-Z]#(2)");
```

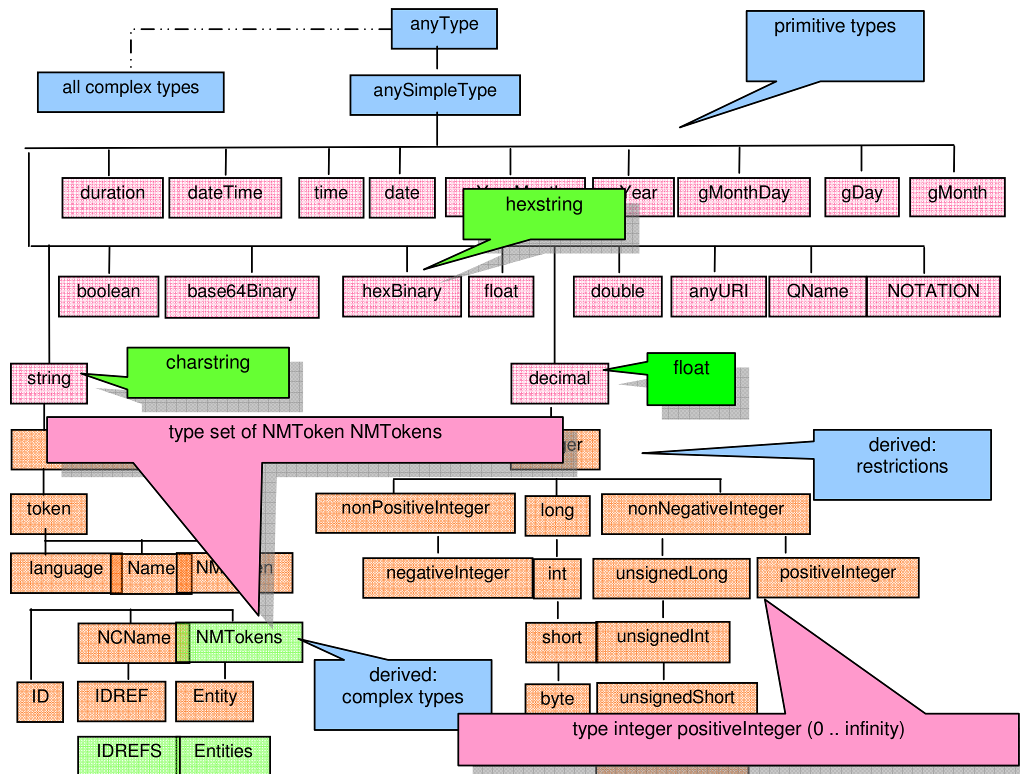

Figure 3: XML simple data type mapping

Figure 3 gives the catalogue of XML build-in types that need a TTCN-3 representation. According to the target TTCN-3 representation, three groups have been distinguished: primitive, restricted and complex TTCN-3 types.

The complexType is used for creating new types that contain other elements and attributes. This is in contrast with the simpleTypes that cannot contain attributes or elements. Just like simpleTypes, complexTypes can be defined globally, which means the possible parents are: <schema> and <redefine>. When they are defined globally, the "name" attribute is mandatory, so the new types will be mapped under the given value of that attribute. And they can be defined locally, in which case the name attribute cannot appear, they will be mapped using an automatic generated name, but they will not be used in other complexType definitions. In other words, they cannot be referenced from other definitions, as there purpose was locally.

For the mapping, every child node is mapped separately to the corresponding TTCN-3 code. The children of complexType are:

- annotation?,
- simpleContent | complexContent
- ((group | all | choice | sequence)?,((attribute | attributeGroup)*,anyAttribute?))

(where *"?"* means zero or one time, *"*"* means zero or many times and *","* means "followed by").

Figure 4: XSD complex data type mapping

Figure 4 provides an overview on XSD simpleType and complexType structures together with samples TTCN-3 representations (e.g. sequence becomes record). As indicated, XSD facets become subtype definitions with length or list restrictions in TTCN-3 (e.g. maxExclusive).

3.2 Implementation

A prototype implementation is available that has been integrated into the commercial TTCN-3 compiler TTthree [6]. The translation from XML schema to TTCN-3 could have been a stand-alone tool, but since it is needed for import handling in TTCN-3 it has been directly included into TTthree. It supports both an implicit mapping of XML structures into an internal tree representation of the XML definitions and also the explicit translation into TTCN-3 core notation syntax by use of TTthree printing features in a second step.
The mapping is divided into two parts: a first mapping translates XML schema to TDOM (Typed Document Object Module) structures and a second mapping translates TDOM structures to TTCN-3 structures. The first mapping is realized with a XML schema, the second one uses a Syntax2Template generator. The TTthree tool is a Java application that comes with its own Java Runtime Environment (JRE 1.3.1) and Java compiler (IBM's Java compiler Jikes v. 1.16).

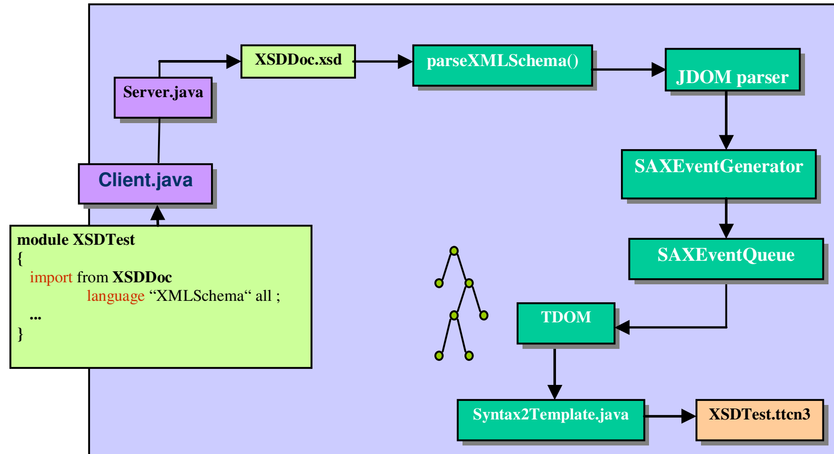

Figure 5: Integration into TTCN-3 compiler

Figure 5 illustrates the sequence of tool applications used for translating an XSD module to a TTCN-3 specification.

4. The Use of TTCN-3 for Testing MOST Applications

This section describes an application of TTCN-3 and its combined use with XML to the systematic testing of MOST applications.

4.1 Description of MOST

MOST stands for "Media Oriented Systems Transport" and defines an accepted set of standards [7] for automobile networking. The standards itself are developed by a cooperation of European car makers.
Communication in a MOST-based system is done between MOST devices, e.g. amplifier, phone, or tuner, following a message-based approach. MOST devices exchange messages by emission of light on the network, which is physically constructed by connecting MOST devices in a ring topology using fibre optic cable.
Transport modes include asynchronous transmission of control commands or general packet based data, there is also a mechanism for bandwidth reservation, enabling devices to synchronously transfer larger amounts of real time data (e.g. video or audio data).
All devices on the MOST bus are provided with unique addresses (*Device-Address*) during start-up of the network. This information is made available in a central registry, enabling arbitrary communication between devices on the bus. A device can implement combined functionality, for example a car's head unit or a combined CD player, tuner and amplifier. Therefore functions are logically grouped in so called function blocks. To distinguish between the function blocks of a device every function block has a distinctive identifier (*FBlockID*). The same is valid for functions (*FktID*).

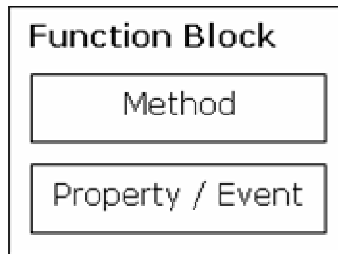

Figure 6: MOST Function Block

Functions are classified either as methods or properties:

- Functions that can be started and which lead to a result after a definable period of time are called “methods”.
- Functions for determining or changing the status of a device are called “properties”.

Additionally, if properties are requested to report changes, they will emit events using the notification service capabilities of the ring network. When accessing functions, different operations can be used. The type of an operation is specified by an identifier (*OPType*) defining how the method or property should be accessed, e.g. *SET* or *GET*. There are operations for storing and retrieving properties, for incrementing and decrementing properties, querying the status of a property, starting and aborting methods or for retrieval of interface information about a certain function. In addition, an operation may transmit sender information by using a different *OPType*, e.g. all OPTypes names that are post fixed with “Ack”. If abstracting from transport level mechanisms, a message on the application layer can be described as a 4-tuple (*FBlockID*, *FktID*, *OpType*, *Parameter*), where *Parameter* contains all arguments used when accessing the function, e.g. a new volume setting when sending a message to an amplifier device. Obviously only a subset of combinations of the tuple’s elements is valid. These combinations have been explicitly specified by the MOST cooperation in a set of specifications known as the “Function Catalogue”.

4.2 The MOST Function Catalogue

MOST devices are independent components and manufactured by different component suppliers, therefore communication between the various components has to be explicitly specified and verified. The catalogue currently defines several basic Function Blocks, e.g. *GeneralFBlock*, *NetworkMaster*, *AudioAmplifier* or *Telephone,* and will be augmented by further definitions as new products and technologies will become available. Apart from the human-readable specification in PDF format, the catalogue also exists in a machine-readable XML format, including a DTD (Document Type Definition) describing its composition. An automatic translation from DTD to XML schema is feasible, as XML schema offers a more powerful and flexible approach to describe XML content than DTD does. When converting the Function Catalogue DTD to XML schema a document structure, like the one shown in figure 6, emerges.

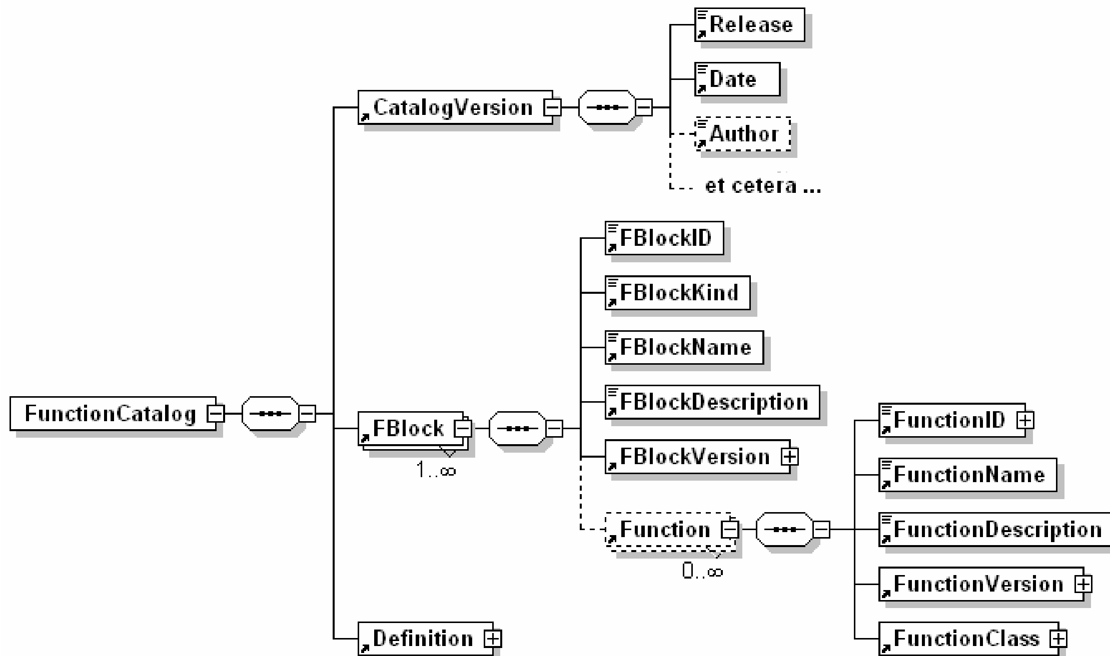

Figure 6: Extract of the upper four nesting levels of the Function Catalogue structure

The diagram can be read as follows: A *FunctionCatalog* element contains a sequence of: Exactly one *CatalogVersion* element followed by an optional amount of *FBlock* elements and concluded by exactly one Definition element. A dashed line signals optional components and a + at the rear end of a box shows that further content follows the element in question. References are labelled as arrows in the bottom-left corner of an element.
The resulting structure is deeply nested: To navigate from a certain *FBlock* to the *OPType* of one of its function's (which are hidden within the *FunctionClass* structure) takes following more than ten references and processing of sequences, respectively. Of course, when importing the catalogue's XML schema this is reflected in the resulting TTCN-3 code, leading – from a programmer's perspective – to cumbersome and hard to use constructs.

4.3 Using the catalogue as a basis for testing

To make use of the benefits of integrating TTCN-3 with the Function Catalogue structures, the XML schema had to be adapted to the requirements of testing. This can be done by pre-processing The XML schema using five steps to simplify the catalogue without breaking the specification.
In a first step, the XML schema data is stripped from futile ballast. For example, when observing figure 6 it's noticeable that the *CatalogVersion* element contains useless information[1], so it can be removed. Other elements to strip from the XML schema data include descriptions and version information in general. The *Definition* element cannot be stripped, as it contains information used by other elements, but it

---

[1] In another context this information might be of great importance

can be integrated directly with the using elements in step five. As *FBlock* is the only element left in *FunctionCatalog*, it's parent element can be removed, as well.
The second step unifies elements. Within the catalogue many elements only differ by a prefix to their names, but not by type or structure. This might a good decision for extensibility of the catalogue, but is not useful with testing. For example take the elements in figure 7: Both *PECommand* and *PACommand* can easily be unified to a type *PCommand*.

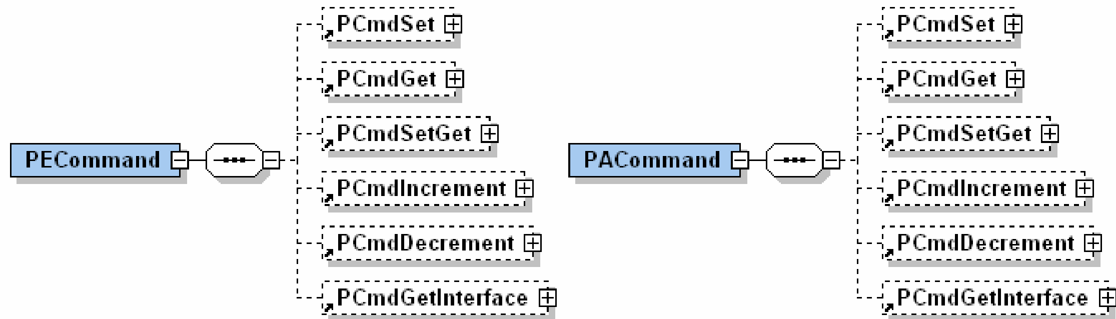

Figure 7: Two elements that can be unified

The third step decides for each element if it should be global, and therefore referencable by others, or if it should be nested within another element. We use three criteria to judge this:

1. The number of attributes of an element
2. The number of children of an element
3. The number of references pointing to this element

These general criteria can be adapted to the input XML schema's structure. With the MOST Function Catalogue, we declare an element local if it has no more than one attribute, no child elements and is only referenced once. This integrates only very simple types with the more complex ones.
The fourth step simplifies structures by mapping them to basic XML schema types, for example the type *TSLong* can be replaced with its counterpart *xs:long*. This step is necessary as DTDs do not have a proper type system and therefore model everything with names, structures and the two CDATA and PCDATA[2] types.
In the last step structural transformations are executed on the XML schema document. For example replacing elements with attributes (see *FBlockID* in figure 6: This element can be translated to an attribute of element *FBlock*) or combining multiple elements in a single one.
After pre-processing, the XML schema is more lightweight, less nested and integrates better with TTCN-3 code. Consider the following fragment from an *AmFmTuner* function block:

```
type XSDAUX.unsignedByte MOST_UBYTE;
… more type definitions …
```

[2] PCDATA is short for "parsed character data" and denominates the text between a start and end tag of an element.

```
type record AmFmTuner_Notification_Set_Signal__Element {
 MOST_UBYTE FBlock,
 MOST_FUNCTION_ID FunctionId,
 MOST_OPTYPE OpType,
 AmFmTuner_Control_Type Control,
 MOST_UWORD DeviceID,
 AmFmTuner_FktIDList_Type FktIDList
}
```

This defines a message to set the notification property of a MOST radio device. The first three entries define the *FBlockID*, *FktID* and *OpType*, where the last three entries define the *Parameter*.

5. Outlook

The test technologies and the tools described in this paper have been successfully used to test the integration of cockpit components.
The case study showed that extensions of the XML to TTCN-3 mapping are useful. As of now, only types can be brought from XML into the TTCN-3 world: our current research in this area focuses on how to generate templates from the instance data contained in the (much bigger) part of MOST Function Catalogue. We are trying to find a method that automatically extracts the data and fills in the generated message structures. If this is finished, a convenient and powerful way exists to test MOST systems using standard TTCN-3 testing environments.
In addition based on this case study, we anticipate that neither generic nor specific extensions to the core language will be required to adequately describe the tests for MOST applications except of their real-time aspects.
We will continue to work on the application of TTCN-3 in the automotive domain to test e.g. also CAN-based components.

References

[1] D.-M. Jeaca: XML schema to TTCN-3 Mapping. Diploma thesis. Politehnica University Bucharest, September 2004.

[2] G. Din: The XML to TTCN-3 mapping. TTCN-3 user conference. Sophia Antipolis, May 2004. http://www.ttcn-3.org/program.htm.

[3] M. Ebener: The IDL to TTCN-3 mapping. TTCN-3 user conference. May 2004.

[4] ETSI ES 201873, Testing and Test Control Notation (TTCN-3), Sophia Antipolis, February 2003. http://www.etsi.org/ptcc/ptccttcn3.htm.

[5] Syntext dtd2xs v2.0 user's guide, Syntext Inc. 2003, http://www.syntext.com/products/dtd2xs/doc/index.html.

[6] TestingTechnologies IST GmbH : TTCN-3 compiler TTthree. http://www.testingtech.de/products/ttthree.php.

[7] MOSTcooperation:http://www.mostnet.de/downloads/Specifications/MOST.

[8] World Wide Web Consortium: XML schema, W3C Recommendations, May 2001. http://www.w3.org/XML/schema.

[9] World Wide Web Consortium: SOAP documentation. W3C Recommendation. http://www.w3.org/TR/soap.

[10] AutoSar: Automotive Open System Architecture, http://www.autosar.org/.

[11] A. Spillner: The W-Model: Strengthening the Bond Between Development and Test, STAREAST 2002, International Conference on Software Testing Analysis and Review. Orlando, Florida, USA, 13. - 17.05.2002.

[12] G. Roper: Quality Driven System Development (QDSD) – a UML-based approach, Conquest 2004, ASQF Press, September 2004, Nuremberg, Germany.

[13] Object Management Group: Model Driven Architecture, http://www.omg.org/mda/.

[14] Object Management Group: Unified Modelling Language 2.0, http://www.uml.org/.

[15] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, S. Lucio, E. Samuelsson, I. Schieferdecker, and C. Williams: The UML 2.0 Testing Profile, Conquest 2004, ASQF Press, September 2004, Nuremberg, Germany.

[16] M. Born, I. Schieferdecker, O. Kath and C. Hirai: Combining System Development and System Test in a Model-centric Approach, RISE 2004 International Workshop on Rapid Integration of Software Engineering techniques, November 26, 2004, Luxembourg, Luxembourg.