A Meta-model for TTCN-3

Ina Schieferdecker[1,2] and George Din[1]

[1] Fraunhofer FOKUS
{schieferdecker,din}@fokus.fraunhofer.de
http://www.fokus.fraunhofer.de/tip
[2] Technical University Berlin, Faculty IV
Chair on Design and Testing of Telecommunication Systems

Abstract. The Testing and Test Control Notation TTCN-3 is a test specification and implementation language that has been defined classically in form of a formal syntax and a semiformal semantics. UML, MOF and MDA has shown that on contrary meta-model based language definitions impose new ways of defining language semantics as a separation of syntax and semantic concept space, of integrating languages via a common meta-model base and of generating and deriving models from other models via model transformers. This paper defines a meta-model for TTCN-3, such that TTCN-3 can take advantage of meta-model based approaches - in particular for integrating test but also system development techniques. It also discusses the realization of the TTCN-3 meta-model and its use for meta-model based tools.

1 Introduction

Model-based development is known in testing since years and used for test generation from models, test validation against models and test coverage analysis. Recently, it has gained a lot in the context of UML. However, with the ability to define tests graphically via the graphical format of TTCN-3 [9] or the UML 2.0 testing profile [4], the need for further integration of test techniques and tools with system development techniques and tools as well as the need for (graphical) model exchange become apparent. Even more gains from other advances in model-based development can be expected. Along the Model Driven Architecture initiative of OMG [15], where system artefacts are developed on different levels of abstraction such as platform independent models (PIM) and platform specific models (PSM), the same concepts and ideas apply to test artefacts. There are platform independent tests (PIT) taking into account test aspects for the general business logic and platform specific tests (PST) taking into account test aspects for platform and technology dependent characteristics of the system under test. Furthermore, PIT and PST (skeletons) can be developed with the help of model transformers from PIMs and PSMs, such that they can reuse information provided in the system models.

This paper considers the first steps towards such an MDA-based test approach: the development of a meta-model for the Testing and Test Control Notation TTCN-3. The main language architecture of TTCN-3 is based on a common

M. Núñez et al. (Eds.): FORTE 2004 Workshops, LNCS 3236, pp. 366–379, 2004.

concept space being specific for the testing domain (in particular for black-and grey-box testing of reactive systems), on the basis of which test specifications can be denoted, developed, visualized and documented by different presentation formats. TTCN-3 supports three different presentation formats: the textual core language [7], the tabular presentation format [8] and the graphical one [9]. Further presentation formats can be defined.

However, TTCN-3 did not take the approach of separating the testing concept space from the presentation formats: it combined the definition of the concept space with the definition of the textual core language. This complicates not only the maintenance of TTCN-3 but also the construction of tools and the exchange of models. For example, the exchange of graphical and tabular TTCN-3 specifications is left open, or, the TTCN-3 semantics definition is mixed with the syntactic definition of the core language.

A solution to these problems is provided by OMG's Meta-Object Facility (MOF): a MOF model can be used as a (structural) definition of a modelling language. It supports mechanisms that determine how a model defined in that modelling language can be serialized into an XML document and represented, inspected, and managed via a CORBA, a Java API, or alike APIs. For a complete language definition, a MOF model has to be enhanced with language syntax (i.e. the presentation formats of TTCN-3 [7, 8, 9]) and a semantics definition (e.g. the operational semantics of TTCN-3 [10]).

Related work to this paper is rather limited as it is the first time that a meta-model is proposed for a test specification language. The work has been inspired by the work on the UML 2.0 Testing Profile [4], which is based on the UML meta-model. A proposal on defining meta-models for ITU languages (and hence including also TTCN) has been presented at SAM 2004 [3], where the emphasis is on deriving meta-models for specification languages from their BNF grammar definitions. In this paper, the meta-model has been developed differently: by considering the semantic concepts of TTCN-3 and identifying the relations between them, the constituting meta-classes and their associations have been identified.

This paper will present a MOF-based TTCN-3 meta-model in Section 2 and discuss its role in the TTCN-3 language architecture for the exchange of models and the construction of tools. A realization and usage of this meta-model in Eclipse will be discussed in Section 3. Section 4 will provide an outlook how to integrate test development with system development.

2 The Development of the TTCN-3 Meta-model

The test meta-model represents the concept space of the Testing and Test Control Notation TTCN-3, which has been developed at ETSI [7, 9] and which has also been standardized at ITU-T. In addition, TTCN-3 served as a basis for the development of the UML 2.0 Testing Profile [4], which has been finalized recently.

The main objectives for the development of a TTCN-3 meta-model were

- The separation of concerns by separating the TTCN-3 concept space and semantics (represented in the TTCN-3 meta-model) from TTCN-3 syntactic aspects (defined in the core language and the presentation formats).
- The ability to define the semantics on concept space level without being affected by syntactic considerations e.g. in the case of syntax changes.
- To ease the exchange of TTCN-3 specifications of any presentation format and not of textual TTCN-3 specifications only.
- To ease the definition of external language mappings to TTCN-3 as such definitions can reuse parts of the conceptual mapping from other languages.
- To integrate TTCN-3 tools into MDA based processes and infrastructures.

2.1 Overview on TTCN-3

Before looking into the details of the TTCN-3 meta-model let us recall some aspects of TTCN-3. TTCN-3 is a language to define test procedures to be used for black-box testing of distributed systems. Stimuli are given to the system under test (SUT); its reactions are observed and compared with the expected ones. On the basis of this comparison, the subsequent test behaviour is determined or the test verdict is assigned. If expected and observed responses differ, then a fault has been discovered which is indicated by a test verdict fail. A successful test is indicated by a test verdict pass.

The core language of TTCN-3 is a textual test specification and implementation language and has been developed to address the needs of testing modern technologies in the telecom and datacom domain. It looks similar to a traditional programming language, but provides test specific concepts. These test specific concepts include e.g. test verdicts, matching mechanisms to compare the reactions of the SUT with an expected range of values, timer handling, distributed test components, ability to specify encoding information, synchronous and asynchronous communication, and monitoring. It has been shown already that test specifiers and engineers find this general-purpose test language, more flexible, user-friendly and easier to use than its predecessors and adopt it as a testing basis for different testing contexts.

TTCN-3 is applicable to various application areas in telecommunication and IT such protocol and service testing (e.g. in fixed and mobile networks and the Internet), system-level testing (e.g. for end-to-end and integration testing), component-level testing (e.g. for Java and C++ objects), platform testing (for CORBA, CORBA components and Web services platforms). It supports different kinds of tests such as functional, interoperability, interworking, robustness, performance, load, scalability, etc. tests.

With TTCN-3 the existing concepts for test specifications have been consolidated. Retaining and improving basic concepts of predecessors of TTCN-and adding new concepts increase the expressive power and applicability of TTCN-3. New concepts are, e.g., a test execution control to describe relations between test cases such as sequences, repetitions and dependencies on test outcomes, dynamic concurrent test configurations and test behaviour in asynchronous and

synchronous communication environments. Improved concepts are, e.g., the integration of ASN.1, the module and grouping concepts to improve the test suite structure, and the test component concepts to describe concurrent and dynamic test setups.

A TTCN-3 test specification (also referred to as test suite) is represent by a TTCN-3 module (or a set of TTCN-3 modules). A TTCN-3 module consists of four main parts:

- Type definitions for test data structures,
- Templates definitions for concrete test data,
- Function and test case definitions for test behaviours, and
- Control definitions for the execution of test cases.

2.2 The Meta Object Facility

The meta-model for TTCN-3 has been defined using MOF - the Meta Object Facility by OMG. MOF resulted from requirements along the extensive application of models and modelling techniques within the scope of the development and use of CORBA-based systems. This lead to the definition of a unique and standardized framework for the management, manipulation and exchange of models and their meta-models. The architecture of MOF is based on the traditional four-layer approach to meta-modelling (see Fig. 1):

- Layer L0 - the instances: information (data) that describes a concrete system at a fixed point in time. This layer consists of instances of elements of the L1-layer.
- Layer L1- the model: definition of the structure and behaviour of a system using a well defined set of general concepts. An L1-model consists of L2-layer instances.
- Layer L2 - the meta-model: The definition of the elements and the structure of a concept space (i.e. the modelling language). An L2-layer model consists of instances of the L3-layer.
- Layer L3 - the meta-meta-model: The definition of the elements and the structure for the description of a meta-model.

MOF supports the following concepts for the definition of meta-models:

- Classes: Classes are first-class modelling constructs. Instances of classes (at M1-layer) have identity, state and behaviour. The structural features of classes are attributes, operations and references. Classes can be organized in a specialization/generalization hierarchy.
- Associations: Associations reflect binary relationships between classes. Instances of associations at the M1-layer are links between class instances and do neither have state nor identity. Properties of association ends may be used to specify the name, the multiplicity or the type of the association end. MOF distinguishes between aggregate (composite) and non-aggregate associations.

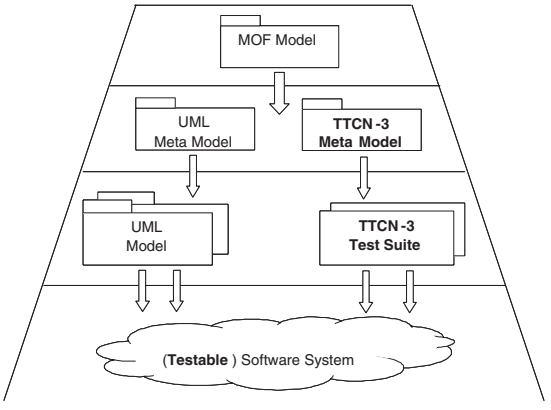

Fig. 1. Model levels of MOF

- Data types: Data types are used to specify types whose values have no identity. Currently MOF comprises of data comparable to CORBA data types and IDL interface types.
- Packages: The purpose of packages is to organize (modularize, partition and package) meta-models. Packages can be organized by use of generalization, nesting, import and clustering.

The MOF specification defines these concepts as well as supporting concepts in detail. Because there is no explicit notation for MOF, the UML notation has been deliberately used to visualize selected concepts.

2.3 Details of the TTCN-3 Meta-model

The TTCN-3 test meta-model defines the TTCN-3 concept space with additional support for the different presentation formats. It does not directly reflect the structure of a TTCN-3 modules but rather the semantical structure of the TTCN-3 language definition. It is defined as a single package with concept structures for

- Types and expressions,
- Modules and scopes,
- Declarations, and
- Statements and operations.

Because of lack of space only excerpts from the meta-model can be presented. Please refer to [2] for further reading. The principal building blocks of test specifications in TTCN-3 are modules (see Fig. 2). A module defines a scope and contains a number of declarations, which can be imports, types, global data covering module parameters, constants, external constants and templates, and functions covering external functions, data functions, behavioural functions, altsteps, test cases and control.

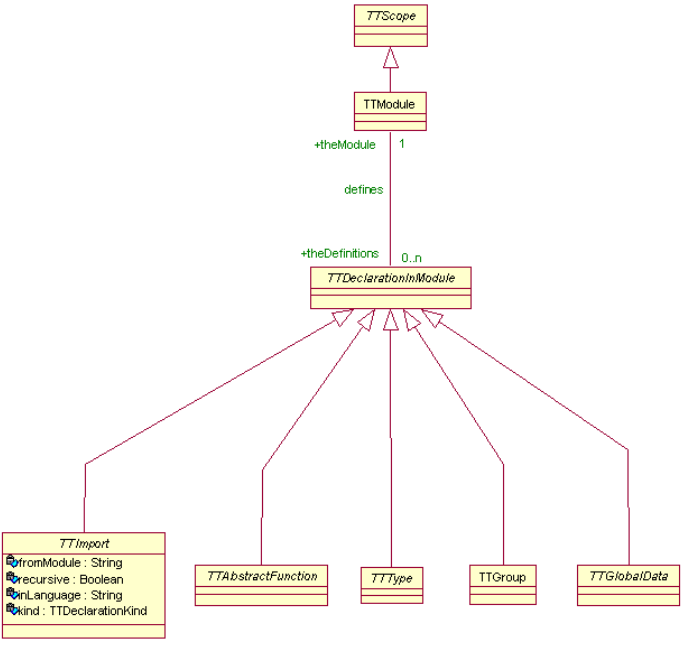

Fig. 2. TTCN-3 Modules

All declarations in TTCN-3 assign an identifier to the declared object and may have a number of with attributes for further characterization of that object (see Fig. 3). As such an object may contain itself a number of other objects, the with attributes may optionally not be applied to some of those.

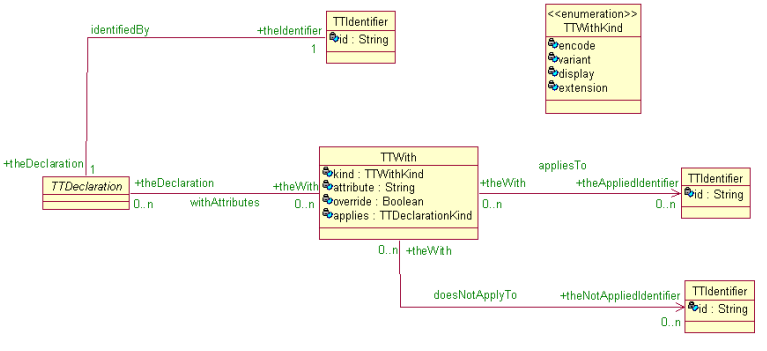

Fig. 3. TTCN-3 Declarations 1

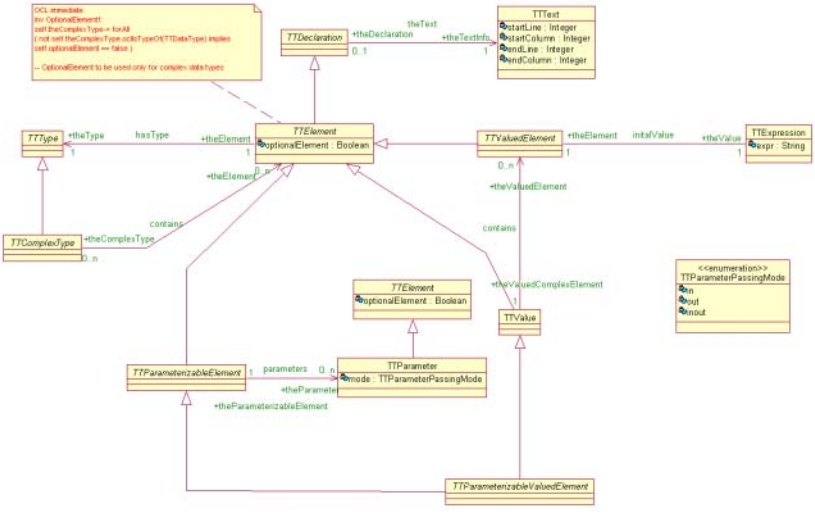

Fig. 4. TTCN-3 Declarations 2

Many declarations in TTCN-3 are typed - we call those declarations elements. Elements can themselves contain elements - in that case the element has a complex type, for example is of a structured type or of a component type. In addition, an element can have an assigned value - such elements are called valued element. Or, an element can be parameterized - then it is called parameterizable element. An element which is both parameterizable and has an assigned value is called parameterizable valued element.

TTCN-3 functions are parameterizable elements with an own scope (Fig. 5). They can be external functions, pure functions, behavioural functions, controls, altsteps, and test cases.

An example for statements is given in the following. Input statements in TTCN-3 cover statements to receive a message, receive a timeout, get a call, get a reply, catch an exception, trigger for a message, check for the top element in the queue, and check for the status of a test component (see Fig. 6). An input statement is always related to a port. The received input as well as the parameters of e.g. an incoming call can be bound to variables. The sender address can be stored as well.

The use of the TTCN-3 meta-model changes the general language architecture of TTCN-3 from a core language centred view to a meta-model centred view while keeping the general front-end to the user but adding new support and tool capabilities to TTCN-3 (see Fig. 7).

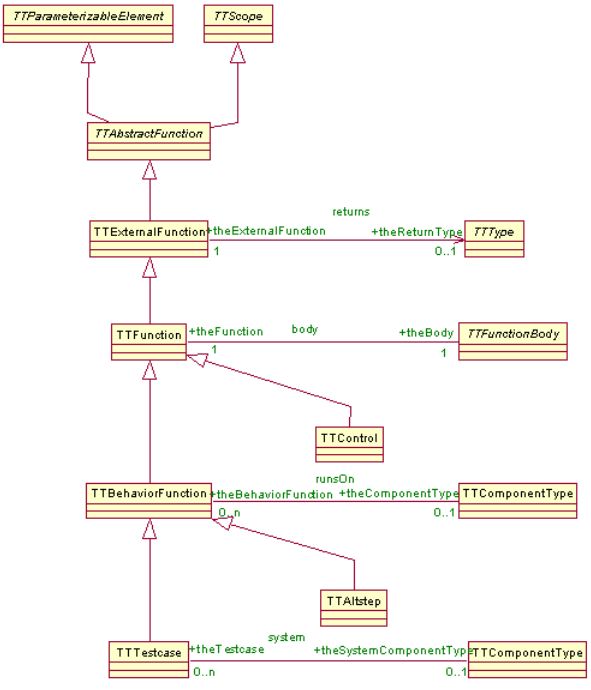

Fig. 5. TTCN-3 Functions

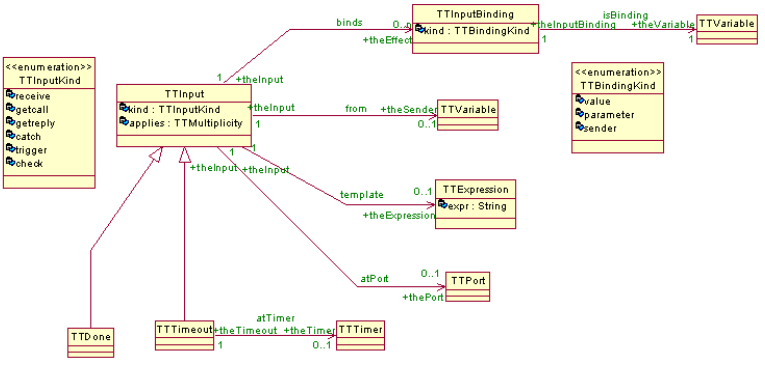

Fig. 6. TTCN-3 Inputs

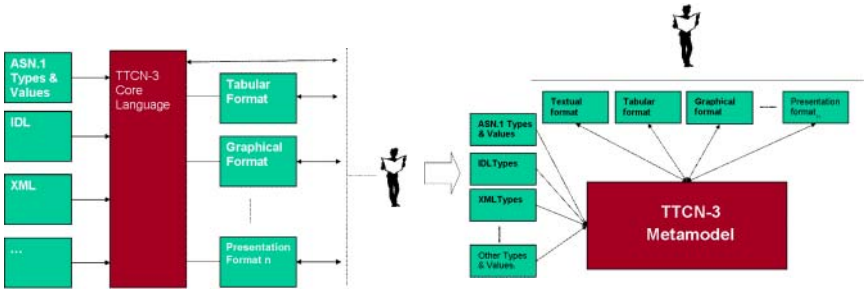

Fig. 7. The change to a meta-model centric TTCN-3 language architecture

3 The Integration of TTCN-3 Tools via the Meta-model

The meta-model for TTCN-3 language was technically realized by using the Eclipse Modelling Framework provided by Eclipse [14]. This allowed us to integrate not only test development and execution tools but also to integrate the development of system artefacts with the development of test artefacts (see Section 4).

3.1 The Realization in Eclipse

This subsection addresses the main capabilities of EMF framework (being part of Eclipe [14]) and discusses the way we used it to implement the TTCN-3 meta-model. EMF is a Java framework and code generation facility for building tools and other applications based on meta-models defined in EMF; it helps turning models rapidly into efficient, correct, and easily customizable Java code. EMF is itself based on a model called Ecore. At the beginning this was intended to be an implementation of the MOF [11], but it evolved from there, now being considered as an efficient Java implementation of a core subset of the MOF API. The EMF Ecore is the meta-model for all the models handled by the EMF.

We used the Rational Rose tool to define the TTCN-3 meta-model in UML. The EMF generator can create a corresponding set of Java implementation classes from such a Rose model. Additional methods and instance variables can be manually added and still regenerated from the model as needed: they will be preserved during the regeneration.

EMF consists of two fundamental frameworks: the *core framework* and *EMF.Edit*. The core framework provides basic code generation through Java EmitterTemplates and Java Merge tools and runtime support to create Java implementation classes for a model. EMF.Edit extends and builds on the core framework, adding support for the generation of adapter classes that enable viewing and command-based editing of a model and even a basic working model editor.

For every class in the meta-model, a Java interface and corresponding implementation class will be generated. Each generated interface contains getter and

setter methods for each attribute and reference of the corresponding model class. The code generation is governed by a couple of mapping rules which coordinates the complex relationships between the entities defined in the meta-model being either one-way, two-way, multiple, and containment references.

One-way references. In case of one-way references objects, the role, i.e. the reference name, will be used to access the target object. Because an object reference is to be handled, the generated get method needs to take care of the possibility that the referenced object might persist in a different resource (document) from the source object. As the EMF persistence framework uses a lazy loading scheme, an object pointer may at some point in time be a proxy for the object instead of the actual referenced object.

Instead of simply returning the object instance variable, the framework method eIsProxy() must be called first, to check if the reference is a proxy, and if it is, then EcoreUtil.resolve() should be called. The resolve() method will attempt to load the target object's document, and consequently the object, using the proxy's URI. If successful, it will return the resolved object. If, however, the document fails to load, it will just return the proxy again.

Two-way references. The two roles of the reference should be used to target two linked objects. The proxy pattern applies also as in the one-way reference case. When setting a two-way reference, one must be aware that the other end of the reference needs to be set as well. This should be done by calling the framework method eInverseAdd().

Multiple references. Multiple references - i.e. any reference where the upper bound is greater than 1 - are manipulated by EMF framework using a collection API, therefore only a get method is generated in the interface.

Containment references. A containment reference indicates that a container-object aggregates, by value, none or more contained-objects. By-value aggregation (containment) associations are particularly important because they identify the parent or owner of a target instance, which implies the physical location of the object. Two very important remarks must be made regarding the way the containment reference affects the generated code. First of all, because a contained object is guaranteed to be in the same resource as its container, proxy resolution will not be needed. Secondly, as an object can only have one container, adding an object to a containment association also means removing the object from any container it's currently in, regardless of the actual association.

Inheritance. Single-inheritance is handled by the EMF generator through the extension of the super interface by the generated interface. Similarly to Java, EMF supports multiple interface inheritance, but each class can only extend one implementation base class. Therefore, in case of a model with multiple inheritance, one needs to identify which of the multiple bases should be used as the implementation base class. The others will then be simply treated as mix-in interfaces, with their implementations generated into the derived implementation class.

Operations. In addition to attributes and references, operations can be added to the model classes. The EMF generator will generate their signature into the

interface and an empty method skeleton into the implementation class. EMF does not support any method of specifying the operation behaviours in the model and therefore the methods must be implemented by hand in the generated Java class.

Factories and Packages. In addition to the model interfaces and implementation classes, EMF generates two more interfaces (and implementation classes): a Factory and a Package. The Factory, as its name implies, is used for creating instances of the model classes, while the Package provides some static constants (e.g. the feature constants used by the generated methods) and convenience methods for accessing the meta-data of the model.

3.2 Use of Generated Java Classes

The generated Java classes can be used to instantiate models and model elements, to manipulate them and to store and load models.

Instantiation. Using the generated classes, a client program can create and access instances with simple Java statements.

Reflective API. Every generated model class can also be manipulated using the reflective API defined in interface EObject. The reflective API is slightly less efficient than calling the generated get and set methods, but opens the model for completely generic access. For example, the reflective methods are used by the EMF.Edit framework to implement a full set of generic commands (AddCommand, RemoveCommand, SetCommand) that can be used with any model.

Serialization and deserialization. Serialization is the process of writing the instance data into a standardized, persistent form, a file on the file system. Loading (sometimes referred to as "deserialization") is the process of reading the persistent form of the data to recreate instances of EObject in memory. In EMF, loading can be accomplished either through an explicit call to a load API or it can happen automatically whenever a reference to an EObject that has not yet been loaded is encountered.

The Eclipse based realization of the TTCN-3 meta-model enabled the integration of the TTCN-3 tools into Eclipse. For the user, the TTCN-3 language support is given via Eclipse plugins, which offer editing, compiling and execution functionality from the Eclipse IDE.

4 Outlook: Integration of System and Test Development

An integrated meta-model based system and test development environment can provide information about system and test artefacts on different levels:

These artefacts are all developed by use of modelling tools and are all stored by use of model repositories (see Fig. 8). A MOF based model bus is used to get access to these models and to transport information between models. Model transformers [1, 2, 13] are used to transform models into models, for example PSM models to code models.

	Platform independent	Platform specific
System artefacts System artefacts	PSM: Platform independent system model	PSM: Platform specific system model
Test artefacts Test artefacts	PIT: Platform independent test model	PST: Platform specific test model

Test generation methods can then be realized in form of model transformers. Having multiple separate models for an entire (test) system, the transformation of these models takes place by the definition of how elements of a source model are transformed into elements of a target model. Using MOF as semantic domain, where the model semantics is defined by MOF meta-models, the transformation definition is also specified at the meta-layer. The specific details of the transformation are given in form of Object Constraint Language (OCL [12]) constraints. One example for such a transformation is the transformation of a CORBA based PSM to a PST. For further details please refer to [2]. An IDL module is transformed to a TTCN-3 module (see Fig. 9). Special control structures within an IDL module (such as include and define) are handled beforehand by a pre-processor. The name of the TTCN-3 module is the same as the name of the IDL module. It contains an import statement that imports the original IDL specification. The TTCN-3 module covers all the TTCN-3 declarations that belong to the transformation for that IDL module.

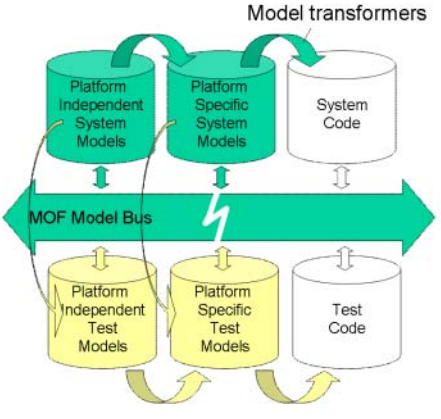

Fig. 8. An integrated system and test development environment

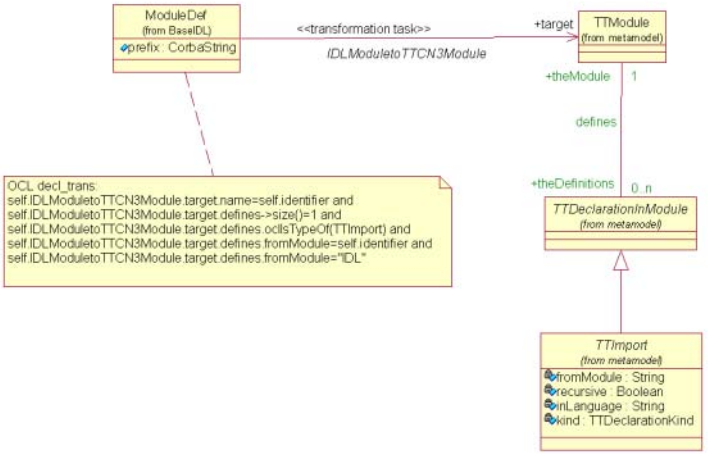

Fig. 9. IDL Modules to TTCN-3 Modules Transformation

5 Summary and Future Work

This paper has discussed the need and aims for a meta-model for TTCN-3. It has described the structure of the meta-model and has shown selected examples of the meta-model definition. Subsequently, the realization of a repository and tool environment based on this TTCN-3 meta-model in Eclipse has been described. Finally, the integration of system development and test development has been discussed in the context of MDA. On the basis of the meta-model, model transformers can be used to derive, generate and transform between PIM, PSM, PIT and PST models.

Our future work will continue in investigating model transformers in particular for platform independent system models to platform independent tests (PIM to PIT) and platform specific system models to platform specific tests (PSM to PST). We do not see much advantage in platform independent tests to platform specific tests (PIT to PST) as the addition of platform specifics is already covered in the transition from PIM to PSM. For the PSM to PST transformer we will consider CORBA based system and base the development on the IDL to TTCN-3 mapping which will give use a TTCN-3 type system and some TTCN-3 structures. In addition, we will add test skeletons for IDL interfaces such as tests on whether the interface exists, whether it provides all defined operations and whether the operations accept correct parameters. Finally, we try to prepare a "ground" for a TTCN-3 meta-model at ETSI as we believe that a TTCN-3 meta-model can ease the language architecture and add to the integration capabilities both between testing methods and tools but also between system development and testing methods and tools.

References

[1] FOKUS, etc: RIVAL T Project: Rapid Engineering, Integration and Validation of Distributed Systems for Telecommunication Applications, Milestone 1: Meta-Models, Dec. 2003. 376

[2] FOKUS, etc: RIVAL T Project: Rapid Engineering, Integration and Validation of Distributed Systems for Telecommunication Applications, Milestone 2: Infrastructure, Transformations and Tools, Febr. 2004. 370, 376, 377

[3] Joachim Fischer, Michael Piefel, and Markus Scheidgen: A Meta-Model for SDL-2000 in the Context of Meta-Modelling ULF, SAM 2004, SDL and MSC Workshop, Ottawa, Canada, June 2004. 367

[4] Object Management Group: UML 2.0 Testing Profile, Final Specification, ptc-2004-04-02, April 2004. `www.fokus.fraunhofer.de u2tp`. 366, 367

[5] Marius Predescu: Meta-Model Based Test Development in Eclipse, MSc Thesis, Politehnica University of Bucharest, Department of Computer Science, September 2003.

[6] Alexandru Cristian Tudose: Meta-model based test suites development - Graphical test specifications under the Eclipse platform, MSc Thesis, Politehnica University of Bucharest, Department of Computer Science, September 2003.

[7] ETSI ES 201 873 - 1, v2.2.1: The TTCN-3 Core Language, Oct. 2003. 367

[8] ETSI ES 201 873 - 2, v2.2.1: The Tabular Presentation Format of TTCN-3 (TFT), Oct. 2003. 367

[9] ETSI ES 201 873 - 3, v2.2.1: The Graphical Presentation Format of TTCN-3 (GFT), Oct. 2003. 366, 367

[10] ETSI ES 201 873 - 4, v2.2.1: The Graphical Presentation Format of TTCN-3 (GFT), Oct. 2003. 367

[11] Object Management Group: Meta-Object Facility (MOF), version 1.4, `http://www.omg.org/technology/documents/formal/mof.htm`. 374

[12] Object Management Group: UML 2.0 OCL 2nd revised submission, `http://www.omg.org/cgi-bin/doc?ad/2003-01-07`. 377

[13] K.Czarnecki, S.Helsen: Classification of Model Transformation Approaches. University of Waterloo, Canada, Technical Report, 2003. 376

[14] Eclipse: Open Source Integrated Development Environment, `www.eclipse.org`. 374

[15] Grady Booch: MDA: A Motivated Manifesto? Software Development Magazine, August 2004 366