Masterarbeit

UBTL
UML Testing Profile Based Testing Language

Johannes Iber, BSc.

Institut für Technische Informatik
Technische Universität Graz
Vorstand: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

Begutachter: Dipl.-Ing. Dr. techn. Christian Kreiner
Betreuer: Dipl.-Ing. Nermin Kajtazović

Graz, im August 2014

Kurzfassung

Testen von Software ist ein komplexer und zeitaufwändiger Prozess. Insbesondere im Kontext von sicherheitskritischen Systemen spielt dies eine äußerst wichtige Rolle. Normen zur funktionalen Sicherheit, die den Entwicklungsprozess und die Qualitätssicherung dieser Systeme regeln, erfordern die Durchführung verschiedener Arten von Tests, die von Tests für kleine Module (Units) bis zu Funktionstests für die vollständige Systemvalidierung reichen. In der Praxis werden diese Testfälle entweder mit existierenden typ–sicheren Unit Testing Frameworks, beispielsweise für C/C++, oder mit neuen angepassten Test–Frameworks realisiert. Letzteres ist wegen vieler Faktoren, wie beispielsweise spezielle Test–Frameworks, der Art der verwendeten Systeme, der Softwarearchitektur und der Vielfalt der zur Verfügung stehenden Prozessoren oft der Fall. Derzeit verwenden die meisten Domänen ein komponentenbasiertes (engl. component–based) Paradigma, um die Produktivität der Entwicklung von Systemen durch die Wiederverwendung von bestimmten Teilen der Software zu verbessern. Beim Testen können mehrere Probleme auftreten. Erstens gibt es nicht viele Test–Frameworks für komponentenbasierte Systeme, und die existierenden sind für gewöhnlich zugeschnitten auf bestimmte Domänen, wie zum Beispiel Fahrzeugtechnik, Medizintechnik oder Stationsleitsysteme. Zweitens legen Sicherheitsnormen im Allgemeinen fest, dass Testfälle auf realer (und meist eingebetteter) Hardware ausgeführt werden müssen, auch Berichte aus diesen Tests müssen reale Umweltbedingungen reflektieren. Andernfalls könnten bereits bestehende Unit Testing Frameworks wiederverwendet werden. Drittens werden Testfälle nicht immer mit Programmiersprachen spezifiziert und können aus mehreren Beschreibungs– und Konfigurationsdateien bestehen.

In dieser Diplomarbeit wird eine textuelle domänenspezifische Sprache (engl. domain–specific language) präsentiert, welche die Aufgabe der Spezifikation von konkreten Testfällen für beliebige Test-Frameworks und Systeme erleichtert. Weiters wird eine leistungsfähige Eclipse–Entwicklungsumgebung zur Verfügung gestellt, die einen Benutzer während des Testprozesses unterstützt. Alle Teile wurden nach der Methodik der modellgetriebenen Softwareentwicklung entwickelt. Die Innovation dieser domänenspezifischen Sprache ist, dass sie automatisch zu UML Modellen kompiliert wird. Zusätzlich werden diese UML Modelle nach der offiziellen Semantik der Spezifikationen von UML Version 2.4.1 und dem UML Testing Profile Version 1.2 aufgebaut, welche von der Object Management Group standardisiert sind. Die daraus resultierenden UML Modelle können zu jeder Notation transformiert oder mit Modellen von Software synthetisiert werden. Es wird das Eclipse UML2 Projekt genutzt, um die UML Modelle zu konstruieren. Das Eclipse UML2 Projekt ist mit mehreren Tools und sogar mit kommerzieller Software kompatibel.

Ein Vorteil dieses Ansatzes ist, dass die Testfälle für verschiedene Testplattformen gleichzeitig genutzt werden können. Unserer Meinung nach eignet sich das besonders für die komponentenbasierte Entwicklung, bei der Komponenten für gewöhnlich im Vorhinein

auf Entwicklercomputern getestet werden und in weiterer Folge wieder auf der Zielhardware. Unser Ansatz ermöglicht es, dass Testplattformen und Hardware sich weiterentwickeln können, ohne dass Testfälle, welche in der vorgestellten domänenspezifische Sprache geschrieben wurden, angepasst werden müssen. Die Sprache selbst ist aufgeteilt zwischen Deklarationen und Definitionen. Die Deklarationen können im Voraus durch Testplattformdesigner oder Maintainer spezifiziert werden, während die Definitionen von Softwaretestern verwendet werden. Zum Beispiel kann ein Testplattformdesigner mögliche Komponententypen deklarieren, während ein Softwaretester Laufzeitobjekte von solchen Typen definiert. Dieser Ansatz macht es leicht, Generatoren zu entwickeln, die Codes erzeugen oder Testfälle mit Modellen von Software synthetisieren.

Bezüglich der Ergebnisse dieser Diplomarbeit wird der beschriebene Ansatz erfolgreich in einem industriellen Projekt angewandt, um Unit Testfälle von Softwarekomponenten nach der IEC 61131 Norm (eine Norm welche auf den industriellen Sektor im Allgemeinen abzielt) zu entwickeln. Am Ende der Diplomarbeit wird ein Auszug aus dieser Anwendung gezeigt, das heißt, es wird erläutert wie eine beispielhafte einfache Softwarekomponente aus der oben genannten Norm definiert und unter Verwendung des vorgeschlagenen Ansatzes getestet wird. Die Definition und das Testen der übrigen Softwarekomponenten wird in einer ähnlichen Weise durchgeführt. Nicht zuletzt wird die Leistung des Ansatzes in Bezug auf Zeit und Speicherverbrauch evaluiert.

Abstract

Testing software is a complex and time consuming process. Especially, in the context of safety–critical systems, this is a crucial part. Standards for functional safety, which regulate the development process and quality assurance of these systems, require performing various types of tests, ranging from tests for small-sized modules (units) to operational tests for complete system validation. In practice, these test cases are either realized with existing type–safe unit testing frameworks, for instance for C/C++, or with new custom test frameworks. The latter is often the case because of factors such as specific test frameworks, the nature of used systems and software architecture, and variety of available processors. Currently, most domains use a component–based paradigm, to improve systems productivity by reusing certain parts of software. Concerning testing, there can be several problems. Firstly, there are not many test frameworks for component–based systems and the existing ones are usually tailored to specific domains, like automotive, medical, or substation automation systems for example. Secondly, safety standards, in general, specify that test cases have to be executed on real (and usually embedded) hardware, and reports from those tests have to reflect real environmental conditions. Otherwise, already existing unit testing frameworks could be reused. Thirdly, test cases are not always only specified with programming languages, but can consist of descriptions and configuration files.

In this thesis we present a textual domain–specific language which eases the task of specifying concrete test cases for arbitrary test frameworks and systems. Further we provide a powerful Eclipse IDE that supports a user during the test process. All parts are implemented following the model–driven software engineering methodology. The innovation of this domain–specific language is, that it is automatically compiled to UML models. Additionally, these UML models are constructed according to the official semantics of UML version 2.4.1 and the UML Testing Profile version 1.2 specifications, which are standardized by the Object Management Group. The resulting UML models can be transformed to any notation or synthesized with models of software. We leverage the Eclipse UML2 project to construct the UML models. The Eclipse UML2 project is compatible with several tools and also with commercial software.

An advantage of our approach is that the test cases can be leveraged for different test platforms simultaneously. In our opinion, this is suited for component–based engineering, where components are usually tested beforehand on developer computers and tested again on target hardware. Our approach allows evolving the test platforms and hardware, without adjusting the test cases written with the presented domain–specific language. The language itself is divided into declarations and definitions. The declarations can be specified in advance by test platform designers or maintainers, while the definitions are used by test engineers. For instance, a test platform designer may declare possible component types, while a test engineer defines runtime objects of such types. This approach makes

it easy to develop generators in order to produce code or to synthesize test cases with models of software.

Concerning the findings of this thesis, we successfully apply our approach in an industrial project to enable unit testing of software components developed according to IEC 61131 standard (a standard that targets the industrial sector in general). In the end of the thesis we show an excerpt of this application, i.e., we show how an exemplary basic software component from the aforementioned standard can be defined and tested using the proposed approach. Definition and testing of remaining software components are conducted in a similar way. Last but not least, we evaluate the performance of the approach in terms of time and memory usage.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Acknowledgments

This master thesis was carried out at the Institute for Technical Informatics, Graz University of Technology.

I especially thank my advisor Dipl.–Ing. Nermin Kajtazović for his incredible mentoring, advice, and commitment. I thank Dipl.-Ing. Dr. techn. Christian Kreiner for offering such an interesting topic.

A special thanks goes to my parents, Roswitha and Josef Iber. Without them nothing would have been possible and they always believe in me. I thank my sister Mag. iur. Andrea Iber for her detailed corrections of this thesis.

Next I thank my girlfriend Mag. pharm. Theresa Holzer for her support and encouraging me in working on this thesis.

I thank my best friends Florian Gollowitsch, B.Ed., Christoph Kronawetter, B.Ed, and Armin Liesinger. I am always having a good time with them besides my studies. I would also like to thank my friend Dipl.–Ing. Lukas Pongratz. Sadly his life ended too early.

Last but not least, I thank all my former colleagues at the university. I really had a great time with them and will always look back with joy.

Graz, in August 2014 Johannes Iber, BSc.

Contents

List of Figures

List of Listings

List of Abbreviations

AST	Abstract Syntax Tree
CIM	Computation Independent Model
CIV	Computation Independent Viewpoint
DSL	Domain–Specific Language
DSML	Domain–Specific Modeling Language
EBNF	Extended Backus–Naur Form
EGL	Epsilon Generation Language
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
EMP	Eclipse Modeling Project
GPL	General–Purpose Language
GPML	General–Purpose Modeling Language
IDE	Integrated Development Environment
MBE	Model–Based Engineering
MBT	Model–Based Testing
MDA	Model–Driven Architecture
MDD	Model–Driven Development
MDE	Model–Driven Engineering
MDSE	Model–Driven Software Engineering
MDT	Model Development Tools
MDT	Model–Driven Testing
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PIT	Platform Independent Test Design Model
PIV	Platform Independent Viewpoint
PSM	Platform Specific Model
PST	Platform Specific Test Design Model
PSV	Platform Specific Viewpoint
SUT	System Under Test
TDD	Test Driven Development
TTCN–3	Testing and Test Control Notation version 3
Ubtl	UML Testing Profile Based Testing Language
UML	Unified Modeling Language
UTP	UML Testing Profile

XMI . . . . . XML Metadata Interchange

Chapter 1

Introduction

In this thesis we present a textual domain–specific language, called "UML Testing Profile Based Testing Language" (Ubtl), for specifying concrete test cases. The main advantage of Ubtl is that Ubtl code is automatically compiled to UML[1] models which use the UML Testing Profile. These models are created according to the semantics of the corresponding Object Management Group specifications. Such UML models can be further transformed to test cases for arbitrary target platforms. Alternatively, the resulting UML models may be synthesized with models of software.

1.1 Motivation

Concerning safety–critical systems, it is mandatory to perform various types of tests on software, according to the desired quality level that has to be achieved (i.e., a Safety Integrity Level (SIL) – a notation used in safety standards), [SS10]. These tests have to be executed on the target hardware, in our case an industrial embedded system. Such systems generally use an architecture based on the component–based paradigm, [Crn02]. A problem is that test frameworks have to be tailored to the specific architecture or system. There is often no appropriate framework and the existing ones are restricted to a specific domain, like automotive, medical, power plants, or substation automation systems. Moreover, many standards and guidelines that define systems and software architecture in aforementioned domains are closed specifications (the IEC 61131/499 industrial component model and the IEC61850 specification for substation automation systems are examples here). Consequently, the community targeting those domains is small, and there are very few practical approaches that address testing on a detailed implementation level (i.e., compared to automotive for example, where the AUTOSAR component model is used, which is an open standard).

The architectures of component–based systems used in aforementioned domains overlap in many aspects. Hence, their difference is reflected in very few details, such as the execution semantics of components, the encapsulation of functions, functional distribution, and coordination services. On the other side, the general concept of a component as a unit of composition with well–defined interfaces remains the same.

[1] `http://www.uml.org`

To support the definition and realization of unit tests for arbitrary platforms and specific domains, it is necessary to provide a generic definition of components and the corresponding test cases. The further mapping of these definitions onto specific component technology, specific test platform, specific test cases, and specific embedded system has to be conducted within the extensible synthesis process.

The architecture of our industrial embedded system, used for demonstration purposes, leverages the component–based paradigm. Components are tested with unit test cases directly on the hardware. Additionally, developers and test engineers test these components on an operating system virtualized with QEMU[2]. The problem is that the concrete test cases differ in their format and programming language. Unit test cases running on the embedded system are defined with XML, while the test cases for the virtualized components are specified with the C++ programming language. Both kind of test case notations usually contain the same test logic.

The motivation for this thesis was to implement or leverage a domain–specific language, preferred a textual one, for specifying these unit test cases abstractly and to translate them into the two different formats. After a comprehensive research, we decided to build a textual domain–specific language based on the UML Testing Profile ([Obj13c]). In our opinion, pure UML is too complex and hard to learn compared to a textual language.

1.2 Goal

As mentioned above, our initial motivation was to implement a testing language for aforementioned two specific different test platforms. This was our first goal. During development, we expanded this goal to create a language which should be flexible and easy to adapt for completely different platforms. So our final goal is to provide a textual language which is abstract enough to only define the concrete test logic and can be transformed to actual test cases running on arbitrary systems. The target test cases can be written with any notation. To be precise, we defined the following subgoals:

Declarations/Definitions: Provide a textual notation, which allows specifying types, classes, and interfaces beforehand. These declarations should be used by definitions, like runtime objects and concrete test cases.

Flexibility: The concepts must be flexible enough to adapt them for different target notations.

Adjustability: It must be possible to disable language features which are not supported on a target platform.

Understandability: The textual domain–specific language must be easy to understand and should use concepts of well–known programming languages. This is crucial for the acceptance of a domain–specific language.

Levels of Testing: The domain–specific language must be usable for unit testing. It may be applied on other levels of testing.

[2] `http://www.qemu.org`

Usability: Provide an integrated development environment for Eclipse, which helps users to create declarations and definitions in Ubtl.

Accuracy: The generated UML model must follow accurately the specified semantics of the official UML and UML Testing Profile specifications by the Object Management Group.

Consistency: The declarations and definitions, specified in Ubtl, must always be transformed to UML in the same way. This is important for generators.

1.3 Thesis Structure

Chapter 2 covers topics related to this thesis. In **Section 2.1** we give a short introduction to software testing. We discuss two fundamental testing strategies, different levels of testing, and test specification languages. In **Section 2.2** we explain Model–Driven Software Engineering, including models, terminology and principles. We also give a short overview of the Model–Driven Architecture and discuss Model–Driven Testing in relation to Model–Based Testing. In **Section 2.3** we give a brief overview of UML. In **Section 2.4** we present the UML Testing Profile and explain details. **Section 2.5** is about the Eclipse Modeling Project. First we discuss the Eclipse Modeling Framework. Then we present the UML2 project. After that we explain the Xtext framework, which we use for constructing the Ubtl compiler and IDE. In the last Subsection we present three model–to–text tools of the Eclipse open–source ecosystem.

In **Chapter 3** we present our textual testing language Ubtl which compiles to UML models. In **Section 3.1** we identify four applications and show how Ubtl can be used. In **Section 3.2** we give a simplified overview of the software architecture. In **Section 3.3** we first give an overview of the textual language elements and then explain them in detail. In **Section 3.4** we discuss how the single language elements are mapped to UML and the UML Testing Profile. In **Section 3.5** we present the customizations that we did of the generated Ubtl IDE. Finally, in **Section 3.6** we explain how we test the Ubtl compiler and the corresponding IDE.

In **Chapter 4** we show our use case of Ubtl. In **Section 4.1** we explain an example component and how we test it. In **Section 4.2** we illustrate how an Ubtl test case concerning this component looks like. **Section 4.3** is about the transformation from the Ubtl test case to a test case written in XML. In **Section 4.4** we present the final XML test case. In **Section 4.5** we evaluate the use case with respect to time and file sizes.

Chapter 5 concludes the work.

In **Chapter 6** we suggest additional useful features for Ubtl.

Chapter 2

Related Work

2.1 Software Testing

Testing is the process of executing a program with the intent of finding errors, [MSB11]. It is a crucial part of every software development process or activity. One problem is the time and money spent for testing software. According to Myers et al. [MSB11], it is still true as a rule of thumb that testing takes in typical programming projects approximately 50% of the elapsed time and more than 50% of the total cost. With this thesis we try to tackle this problem, in the context of specifying the same test case for different platforms.

In the following subsections we give a very short introduction to software testing. First we discuss the two basic software testing strategies. Next we give an overview of the different levels of testing based on the well–known V–Model. After that we discuss test specification languages and present TTCN–3.

2.1.1 Software Testing Strategies

Fundamentally there exist two software testing strategies to identify test cases:

Black–Box Testing Like the name implies this strategy views a system under test (SUT) as black–box. Often this strategy is called functional testing, [Rep09]. The term SUT refers to a system which is being tested. The complexity of a SUT can range from a method to an entire application. Black–box testing relies on the specifications of a system to develop test cases. Also *test data are solely derived from the specifications*, [MSB11]. The internal behavior and structure of a SUT is not taken into account, [MSB11]. A test case simply evaluates the behavior of a SUT by giving pre–defined inputs and by examining if the outputs correspond to known correct outputs, [Rep09]. The problem with this approach is that a system has to be exhaustive tested with all possible inputs to find all errors (exhaustive input testing). This is not feasible, even for small programs, [MSB11]. Therefore several systematic test development methodologies for black–box tests have been developed, for instance equivalence class partitioning and boundary value analysis, [Rep09].

White–Box Testing In contrast to black–box testing, white–box testing uses the internal behavior and structure of a SUT. *This strategy derives test data from an examination of the programs logic (and often, unfortunately, at the neglect of the*

specification), [MSB11]. White–box testing is also known as structural testing. The goal of this strategy is to test every statement of a program. The presumption is that if all possible paths of control flow through the program are executed via test cases, then the program is possibly completely tested (exhaustive path testing), [MSB11]. This strategy has two problems, [MSB11]. First the number of unique logical paths could be extremely large. Second there could still be errors in a program: A program could not match the specification. Also paths could be missing in the program. An *exhaustive path test might not uncover data-sensitivity errors*, [MSB11].
Despite that it is unrealistic to cover all possible paths in a system, several helpful techniques have been developed.

The combination of black–box testing and white–box testing is called grey–box testing, where a test engineer has access to the systems source code, but test cases are executed like black–box tests, [Rep09].

This thesis is only about black–box testing.

2.1.2 Levels of Testing

There exist a wide variety of software engineering process models, ranging from the Waterfall model to agile methodologies like Extreme Programming, [Lin01]. We chose the V–Model to explain the different levels of abstraction in the process of testing software. *Although this model has its drawbacks, it is useful for testing as a means of identifying distinct levels of testing and for clarifying the objectives that pertain to each level*, [Jor13].

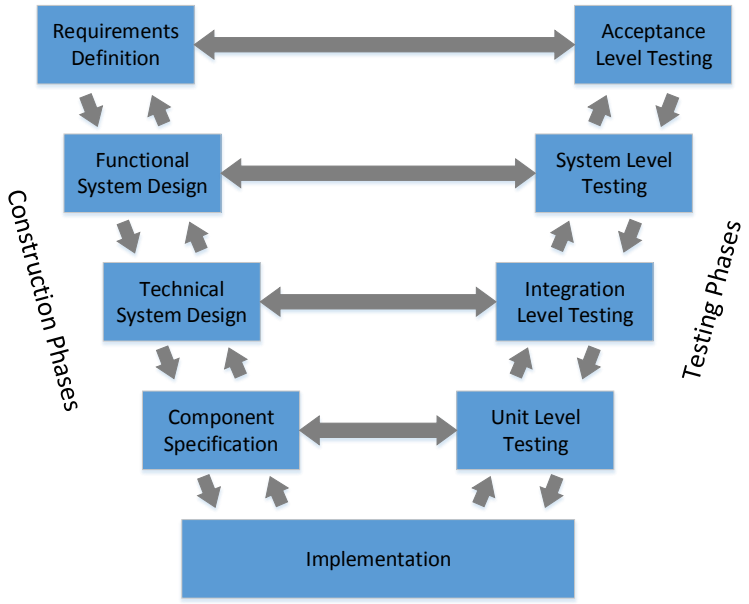


Figure 2.1: V–Model (based on [BDG+08])

Figure 2.1 illustrates the principle structure of the V–Model, [BDG+08]. On the left–hand side we find the *Construction Phases*, while on the right–hand side reside the *Testing Phases*.

The software development process starts with the *Requirement Definition* phase. In this phase requirements are gathered from the customers and possible users. These requirements are the basis for the next phase called *Functional System Design* to develop a functional model of the system. *The functional model should be independent from the future implementation of the system to avoid early design decisions*, [BDG+08]. The actual architecture of the software is modeled in the *Technical System Design* phase. Such a system is split into individual components and the interfaces between them are designed. The *Component Specification* phase deals with the definition of the behavior of these components. In the next step components are implemented and the *Construction Phases* end.

The *Testing Phases* are based on the implementation, and leverage of the specifications and requirements in order to verify and validate the products. *Verification is concerned with the correctness of individual products with respect to their specifications, and validation refers to the correctness of individual products with respect to the intended use of these products*, [Lin01]. *Unit Level Testing* tests the individual behavior of the different components. The *Integration Level Testing* phase tests if the components interact correctly. It ends when all components are integrated. *System Level Testing* targets the whole system and the corresponding specified functionalities are tested. The last phase, called *Acceptance Level Testing*, is similar to *System Level Testing*, but based on the perspective of the customers and users, [BDG+08].

Even though the V-model suggests a procedure where the testing phases are performed after the construction phases, it is well known that the preparation of each testing phase should start as early as possible, that is, in parallel to the corresponding construction phase. This allows for early feedback regarding the testing phase, [BDG+08].

2.1.3 Test Specification Languages

In this thesis we refer to "test specification languages" in the sense of languages where the test case logic is defined manually. We do not deal with concepts where test cases are derived automatically from system models or formal models.

In general, one can specify test cases with the same programming language in which a system is implemented. This is usually supported by xUnit frameworks. Examples are JUnit for the Java programming language or the original SUnit for Smalltalk. The drawback of this approach is that when a system is reimplemented in another programming language, the test cases also have to be reimplemented. Test cases are tied up with the programming language and have to be manually converted. It is also necessary to implement them again when a test platform changes or a system is tested with a different one, like in our case.

A higher level of abstraction is achieved when test cases are implement with a scripting language or high–level general–purpose language, like Python or Tcl. A drawback is that it may be necessary to write adapters in order to test a system. If the test specification language is interpreted, the runtime must be available on the platform on which a system is tested.

With this thesis we aim at providing a textual test specification language that is on an even higher level of abstraction, where the test case logic is independent of any target test platform or programming language. To achieve this, we leverage the UML and UML

Testing Profile, which is also a way of defining test cases abstractly, but usually by using the graphical syntax of UML. We discuss the UML Testing Profile in detail in Section 2.4. In the following we briefly give an overview of TTCN–3. TTCN–3 allows specifying test cases abstractly, but the test cases are meant to be used inside a standardized test system architecture. We present TTCN–3 to make the differences to our test specification language obvious.

TTCN-3

The Testing and Test Control Notation version 3 (TTCN–3) is a standardized test specification language and test system architecture for communication–based systems. It is standardized by the European Telecommunication Standards Institute (ETSI) and the International Telecommunication Union (ITU–T) since the year 2000, [Sch10]. *TTCN–3 can be used for the specification of all types of reactive system tests over a variety of communication ports. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing, testing of CORBA based platforms, APIs, etc. TTCN–3 is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing*, [ETS14a].

It offers a textual language and two graphical formats, which can be used interchangeably. One graphical format is Message Sequence Chart based (similar to UML Sequence Diagrams), the other one uses a tabular format, [SGVGD08]. The core language of TTCN–3 is the textual one which can be interchanged between different tool vendors. The graphical formats are mapped to the textual one according to the TTCN-3 specifications. TTCN–3 allows importing data types from various sources, that are mapped to predefined data types. The TTCN–3 standards specify imports from the languages ASN.1, IDL and XML, [ETS14a]. TTCN–3 embeds test behavior in test cases. The definable test components and SUTs, used by test cases, communicate over ports. The communication can be either message–based or procedure–based. *Message–based communication is based on an asynchronous message exchange. The principle of procedure–based communication is to call procedures in remote entities. This allows a test component to emulate the client or server side during a test. Furthermore, unicast, multicast and broadcast communication are also supported by TTCN–3*, [SGVGD08].

Figure 2.2 illustrates the standardized test system architecture. In principle, the architecture and the contained entities can be realized in any programming language. TTCN–3 code is either compiled and executed, or interpreted by a TTCN—3 Executable (TE). It can also contain a TTCN–3 runtime system, [ETS14b]. Note that the TTCN–3 standards only specify the grammar of the TTCN–3 code, but do not specify an intermediate representation or metamodel (see Section 2.2). Therefore it depends on the tool vendors how this part is realized. The Test Management entity is responsible for the overall management which includes test execution. The Test Logging (TL) entity maintains the test log. The Component Handling (CH) entity distributes parallel components. *The CH entity allows the test management to create and control distributed test systems in a manner which is transparent and independent from the TE*, [ETS14b]. The Coding and Decoding (CD) entity is responsible for external data used by message or procedure based communication inside the TE. External codecs have a standardized interface, and can be used by different

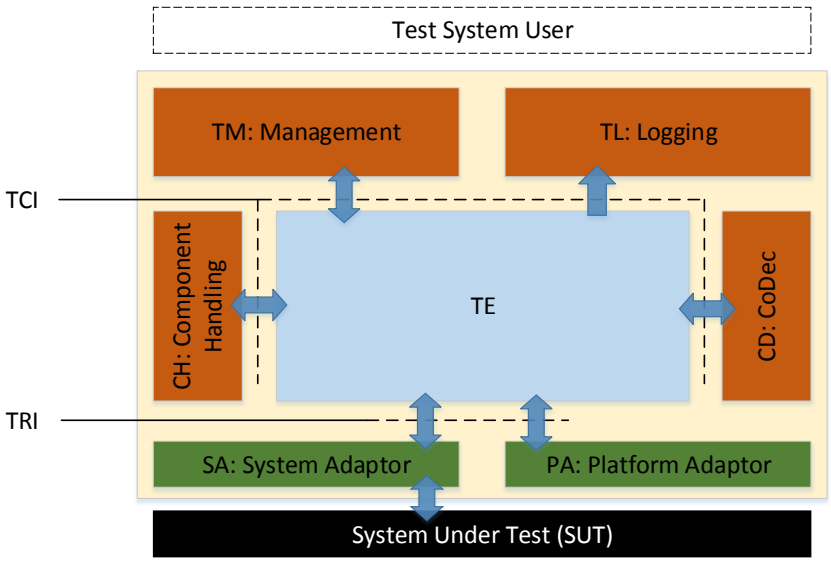


Figure 2.2: TTCN–3 test system architecture (based on [ETS14b])

TTCN-3 systems or tools, [ETS14b]. The SUT Adaptor (SA) realizes the communication with the SUT. This adaptor has to be implemented for different SUTs, and offers a standardized interface to the TE. The Platform Adaptor (PA) implements external TTCN–3 functions and timers. TE communicates with the CH, TM, TL, and CD, entities via the TTCN–3 Control Interface (TCI), which is standardized. The TTCN–3 Runtime Interface (TRI) is also standardized and used by the TE for the communication with the SA and PA. TCI and TRI are specified in CORBA IDL, [Obj12a].

A drawback of TTCN–3 is, in our opinion, that there is no free or open–source compiler and test system. Also TTCN–3 code cannot be compiled to various test frameworks (for instance JUnit) and is restricted to the standardized test system architecture.

2.2 Model–Driven Software Engineering

Model–Driven Software Engineering (MDSE) is a methodology where models are used for all activities of software development, [BCW12].

MDSE (or sometimes called Model–Driven Engineering, MDE) promises improvements in productivity, portability, interoperability, maintenance and documentation of software or development processes, [KWB03]. The work in [HWRK11] mentions *it is difficult to provide absolute measures of the benefits of MDE*. For instance, studies found in the literature, have reported *productivity gains ranging from -27% to +1000%*, [WH12].

In the following subsections we present the theory of MDSE and discuss several aspects of MDSE, starting by explaining what models are and how they can be used. After that we explain acronyms concerning model–based and model–driven used in the literature. Then we focus on basic principles of MDSE which are relevant for this thesis. We also present the Model–Driven Architecture (MDA), a software development approach and a

framework for standards provided by the Object Management Group (OMG). Last but not least, we explain Model–Based Testing (MBT) and the term Model–Driven Testing (MDT), including its relationship to MBT.

2.2.1 Models

Kühne [KÖ6] uses, in the context of MDE, the following definition for models, *A model is an abstraction of a (real or language–based) system allowing predictions or inferences to be made.* Von Bertalanffy [Von68] defines in his work a system as *a set of elements in interaction.*

Abstraction is clearly a powerful capability, as it can hide unnecessary information. Therefore models can represent the pure logic, for instance, of a complex software system. Implementation details can be added in more precise models or by (semi)automatic transformation processes.

Baerisch [Bae10] further differentiates the types of models by two characteristics:

- **Formal or informal:** A *formal* model uses a defined structure and semantics. It can be automatically processed or transformed for its intended purpose. Obviously *informal* models do not offer such possibilities.
- **Prescriptive or descriptive:** According to Bézivin [BÓ5], a *prescriptive* model is a representation of a system to build, while a *descriptive* model can represent an existing system. Ludewig [Lud03] explains that when a descriptive model is modified and then used prescriptively, it is called a *transient* model.

Similar to these characteristics, Brambilla et al. [BCW12] classify the use people make of models as follows:

- Models can be used as sketches and therefore are only used for communication purposes. Such models provide partial views of a system.
- Used as blueprints, models are complete and detailed specifications of systems. E.g. a programmer could implement a system based on a blueprint.
- Last, but not least, models, instead of code, are used to develop systems.

In this thesis we use formal models, which are prescriptive, instead of code to develop test cases.

Selic [Sel03] identifies in the context of model–driven development, five characteristics of useful and effective models:

Abstraction As stated before, a model must remove or hide irrelevant details. Only then it is suitable to cope with complex systems.

Understandability A good model must reduce the intellectual effort required for understanding. It should be intuitive or little information should be necessary for understanding it.

Accuracy Selic [Sel03] states that *a model must provide a true–to–life representation of the modeled system's features of interest.*

Predictiveness It should be possible to predict non–obvious properties of a modeled system, through experimentation or formal analysis.

Inexpensiveness It must be significantly cheaper to construct or analyze a model than a modeled system.

2.2.2 Terminology

In the literature there exist a lot of different acronyms dealing with model–driven and model–based. Brambilla et al. [BCW12] made an overview of the relations between the acronyms and the approaches. The following hierarchy is based on this overview and each acronym is a subset of the preceding acronym.

MBE Model–Based Engineering is a process where models play an important role in the development of software, but are not the source artifacts. They do not *drive* the development. Brambilla et al. [BCW12] mention as an example that a designer could specify a model of a system and then hands it over to programmers, who manually implement it.

MDE In Model–Driven Engineering models are the key artifacts of all development related activities and tasks. The core principle is "Everything is a model", [B04́].

MDD Brambilla et al. [BCW12] state that *Model–Driven Development is a development paradigm that uses models as the primary artifact of the development process.* Such models are usually (semi)automatically transformed to implementations (e.g. code).

MDA Model–Driven Architecture is a vision of MDD by the OMG. OMG provides standards concerning various aspects of modeling. MDA uses these standards to define a development approach.

According to this hierarchy, all model–driven principles and approaches are model–based, but not all model–based techniques are model–driven. To be precise, MDSE is a subset of MDE and only focuses on the development of software. However, in the literature, MDSE is sometimes named MDE.

Concerning the practical part of this thesis, we heavily make use of OMG standards. Therefore we give a short overview of MDA in Subsection 2.2.4.

2.2.3 Basic Principles

According to Brambilla et al. [BCW12], the main ingredients of MDSE are models and transformations. We start with metamodeling, go on to modeling languages, and finally explain transformations. These are the relevant principles for this thesis.

Metamodeling

MDSE relies on formal models. Formal models have to be defined with a concrete modeling language, otherwise they would be arbitrary and informal. Such a modeling language can also be defined by another modeling language. Basically, the process of modeling a modeling language is called *metamodeling*, [BCW12].

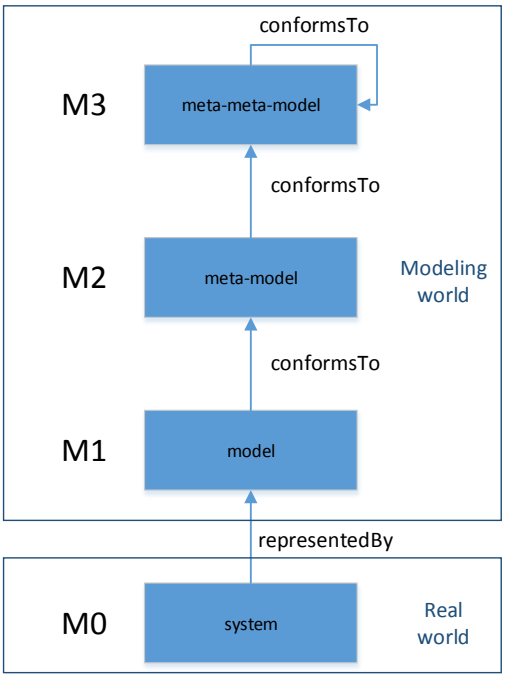


Figure 2.3: Four–layer metamodeling architecture (based on [Bó4])

Figure 2.3 illustrates a typical four–layer metamodeling architecture, that is used in MDSE. Note that a modeling language is in fact a model.

M3 This layer is the basis of the metamodeling architecture. Its purpose is to provide modeling languages for defining modeling languages. Usually a meta–meta–model is defined reflexively, that means it can define itself. In practice, it does not make any sense to define further meta layers, [BCW12]. In Figure 2.3 this behavior is described by a *conformsTo* relationship. Meta–meta–models are similar to grammars like the Extended Backus–Naur Form (EBNF). Such a grammar is used for defining different programming languages and can also describe itself, [Bó5].

M2 The purpose of this layer is to describe modeling languages which are used on the next layer for specifying the actual model. It has to conform to the meta–meta–model at layer M3, like a programming language has to conform to its grammar.

M1 Models at this layer represent modeled systems. They have to conform to the corresponding meta–model.

M0 That layer is not part of the modeling world and part of the real world. It consists of real systems, which can be represented by models.

For demonstration purposes, we present the example hierarchy used in the OMG Unified Modeling Language (UML) Infrastructure specification [Obj11a]. Figure 2.4 illustrates the example hierarchy. Note that OMG uses the notion *instanceOf* instead of *conformsTo*.

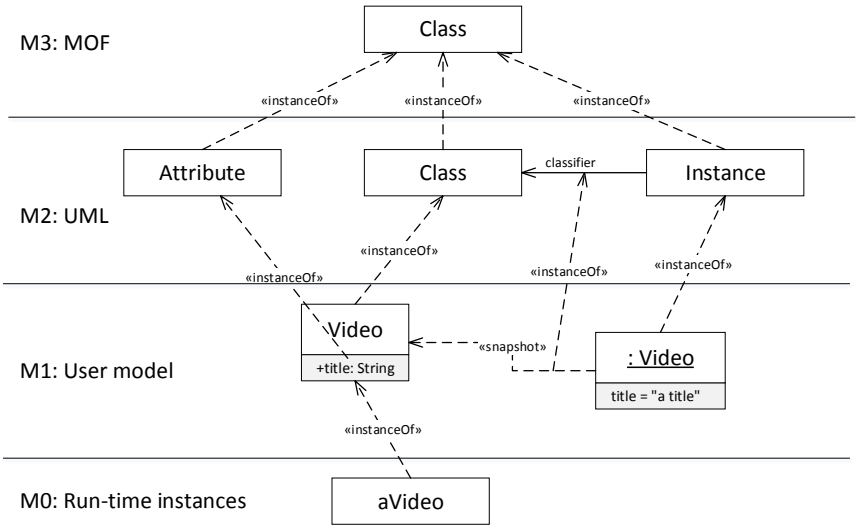


Figure 2.4: UML example four-layer meta-model hierarchy (based on [Obj11a])

Meta Object Facility (MOF) is an OMG standard [Obj13a] for describing meta–models, like UML. As we can see in Figure 2.4, UML concepts, like *Attribute*, *Class* or *Instance*, are defined by the element *Class* of MOF. The user model on layer M1 uses instances of the UML layer, for defining the class *Video* and an instance specification (snapshot) of *Video*. Note that class specifications and instances (objects of classes) are defined on the same layer. Finally such a *Video* class represents the real world concept of a video.

According to Brambilla et al. [BCW12], metamodels can be defined for:

- new programming or modeling languages
- new modeling languages for exchanging and storing information
- new properties or features to be associated with existing information (metadata)

Modeling languages

According to Brambilla et al. [BCW12], a modeling language is defined through three parts:

Abstract Syntax: The abstract syntax of a modeling language is the metamodel. It describes the structure of a language and how different elements are connected and can be combined. It is independent of any particular representation or encoding, [BCW12]. Usually a metamodel contains classes, attributes and associations for describing the language. Such an abstract syntax can be further improved by constraints defined with constraint languages. For instance, a simple constraint would be if the name of an element always has to start with an upper–case letter.

Concrete Syntax: Metamodels only define the abstract syntax, but not the concrete notation of a modeling language. Concrete syntaxes are specific visual representations

of a metamodel. They can be either textual or graphical. Of course, it is possible to define both for one metamodel. Designers work with concrete syntaxes, when they manipulate a metamodel. For instance, if the concrete syntax is graphical, they use one or more diagrams.

Semantics: The correct usage and meaning of elements or the meaning of the different ways they can be combined is described by semantics. Brambilla et al. [BCW12] point out that *the semantics of a language can be defined in various ways: by defining all concepts, properties, relationships and constraints through a formal language; through practical implementations of code generators which implicitly define the semantics of the language by generating code; or by defining in–place transformations for simulating the model's behavior.*

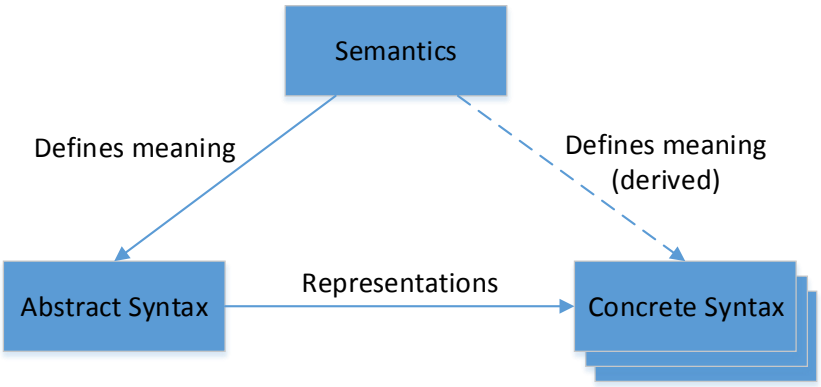


Figure 2.5: The three parts of a modeling language and their relation to each other (based on [BCW12])

Figure 2.5 highlights these parts and shows the relationships between them. Several concrete syntaxes can represent the abstract syntax. The semantics defines the meaning of the abstract syntax and indirectly the meaning of concrete syntaxes. Brambilla et al. [BCW12] state that all three parts are mandatory for a modeling language to be well defined. For instance, a partial or wrong specification of the semantics enables the wrong usage of a language and leads to misinterpretations of the meaning of language elements and purpose. Different people may understand the concepts and models differently.

Modeling languages typically allow to define different models targeting *static* (or *structural*) or *dynamic* aspects of a problem or solution. The first aspect describes modeled entities and their relations, while the dynamic aspect describes their behavior, e.g. actions and interactions, [BCW12].

Generally, languages can be classified into two categories:

Domain–specific modeling languages (DSML) are languages designed for the purpose to target a specific domain. They should *ease the task of people that need to describe things in that domain*, [BCW12].

General–purpose modeling languages (GPML or GML) are languages that can be applied to any domain. Therefore they are more complex and more complicated to understand than DSMLs.

Brambilla et al. [BCW12] mention that *this distinction is not so deterministic and well–defined*. As an example, UML can be used to model any kind of vertical domain (GPML), but can be seen as a DSML tailored to specify software systems. Also it is possible to extend and customize UML for domain specific needs. We explain that in Section 2.3.

DSMLs and domain–specific languages (DSL), used in the non-modeling world, are very similar. Therefore we use the terms DSL and GPL (general–purpose language), in the context of this thesis, instead of DSML and GPML. Brambilla et al. [BCW12] note that DSL is, in the modeling community, instead of DSML by far the most adopted acronym. The same is true for the acronym GPL.

Using a DSL, instead of a GPL, offers several advantages. DSLs can improve the development productivity, [Fow10]. Kelly and Tolvanen [KT08] state that *domain–specific approaches are reported to be on average 300–1000% more productive than general–purpose modeling languages or manual coding practices*. They limit the possibilities how to express something, which leads to fewer mistakes and duplicates. Also they make it easier to modify a system. Through abstraction they make it easier to focus on a problem and less time is spent for dealing with details. DSLs can improve the communication between domain experts by providing a clear and precise language to deal with a domain, [Fow10]. Learning a DSL can be easier than learning a GPL like UML, because domain concepts are known or need to be known, no matter how a software is developed, [KT08]. Further a DSL is usually less complex than a GPL. On the downside, it takes some time to build an appropriate DSL and corresponding tools. After a DSL is built, it is often necessary to constantly maintain and update the language and tools. Fowler [Fow10] mentions that, fundamentally, the only reason for not using a DSL is, if the benefits are not worth the cost of building a DSL.

Brambilla et al. [BCW12] highlight a few principles, which are necessary for a DSL to be useful:

- A language must provide good abstractions for designers. It must be intuitive and make life easier, not harder.
- A language must not depend on the expertise of one person for its adoption and usage. Its definition must be shared among people and agreed upon after some evaluation.
- A language must be kept updated based on the context and user needs.
- A language must come together with supporting tools and methods. Domain experts want to maximize their productivity and are not willing to spend a lot of time for defining methods or tools.
- A good DSL should be open for extension and closed for modifications, according to the *open–close principle*, which states that *software entities (classes, modules, functions) should be open for extension, but closed for modification.*

Transformations

Model transformations play a crucial role in MDSE. Without transformations it would be pointless to specify formal models. Czarnecki et al. [CH06] list several applications of model transformations:

- Generating lower-level models from higher-level models.
- Generating code from models.
- Mapping and synchronizing among models at the same level or different level of abstraction.
- Query–based views of systems.
- Model refactoring.
- Reverse engineering of higher-level models from lower–level models or code.

There exist three variants of transformations, [WNO12]:

- **Model–to–Model** is the transformation from a source model to a target model, e.g. a platform–independent model is transformed to a platform–specific model.
- **Model–to–System** describes the translation of a model to system code, configuration or text, e.g. a model is transformed to Java code.
- **System–to–Model** is the process of generating a model from an existing system, e.g. a model is generated from Java code (reverse engineering).

In the literature, or depending on the context of a transformation, *system* is often replaced by *text* or *code*.

Following the principle "Everything is a model", transformations themselves can be designed as models, [BCW12]. Figure 2.6 illustrates the simplest concept of a Model–to–Model transformation. A transformation model has its own metamodel, while the actual model maps elements of the source metamodel with elements of the target metamodel. In the subsequent steps a transformation model is executed by a transformation engine, which can take one or multiple source models to generate one or multiple target models. The source and target metamodels may be the same. That can be useful for instance for optimization or refactoring of models. According to Mens et al. [MG06], a transformation which takes place between models, expressed by the same metamodel, is called an *endogenous* transformation, while a transformation between different metamodels is called *exogenous*. Further, if only one model is involved in an endogenous transformation, it is called *in–place*, otherwise it is named *out–place*. Exogenous transformations are alway out–place.

Note that Model–to–System and System–To–Model solutions can also involve transformation models. The model–to–text tool we use for the practical part is, in fact, depending on transformation models for transforming models to texts.

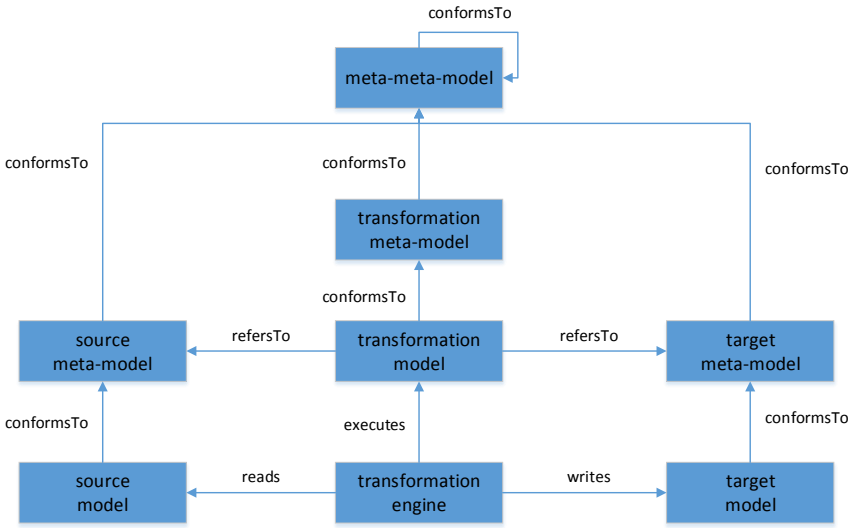


Figure 2.6: Model transformation concept (based on [WNO12])

2.2.4 Model–Driven Architecture

Model–Driven Architecture (MDA) is a software development approach proposed by the OMG. The OMG describes that *the three primary goals of MDA are portability, interoperability, and reusability through architectural separation of concerns*, [Obj03].

OMG does not provide any tools implementing MDA. They purely concentrate on developing standards together with the industry.

The MDA Guide [Obj03], OMGs MDA explanation, references several OMG standards, like UML, MOF or XMI, which are also relevant for this thesis and widely used by the industry and tool developers. We provide an in–depth discussion of UML in Section 2.3. The Meta Object Facility [Obj13a] is a standardized meta–meta–model. Ecore, which is discussed in Section 2.5, is an implementation of a simplified MOF version, called Essential Meta Object Facility (EMOF, [Obj13a]). MOF (including EMOF) serves as metamodel for modeling languages like UML (layer M3, see Section 2.2.3). XMI stands for XML Metadata Interchange [Obj13b] and is a specification for XML documents. These are used for serializing and exchanging models between different tools.

Concept

Figure 2.7 illustrates an overview of the MDA approach. OMG describes Viewpoints on a system as *a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system*, [Obj03]. The Computation Independent Viewpoint (CIV) is about the environment and the requirements of a system. The Platform Independent Viewpoint (PIV) deals with the operation of a system, while the details of a specific platform are hidden. The Platform Specific Viewpoint (PSV) adds platform details to PIV. MDA uses three kinds of abstraction models. Each of these models is a view from a corresponding viewpoint. Several views

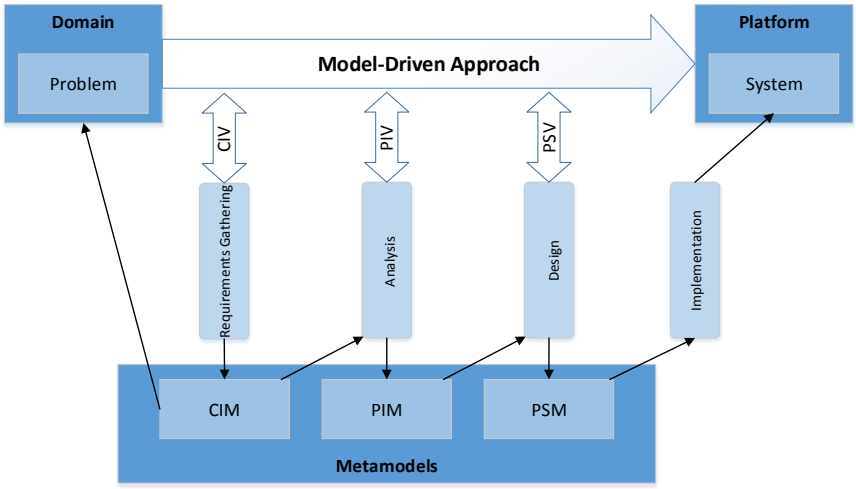


Figure 2.7: MDA Overview (based on [SS09])

can be used for one viewpoint. That is because sometimes one model should not capture all relevant information, or all information included into one model would be too fuzzy. Using several models for one viewpoint can keep different concerns separated, but together offer a comprehensive description of a system, [BCW12].

- Computation Independent Model (CIM) consists of requirements and what a system is expected to do, also in relation to the environment. OMG mentions that it is sometimes called domain model. It does not show any details of the architecture of a system. Further, it can serve as vocabulary for domain practitioners and modeling experts, [Obj03].

- Platform Independent Model (PIM) describes a system, but does not show any details of a target platform. A PIM can be build independently from a CIM.

- Platform Specific Model (PSM) combines the specification of a system (PIM) with information about how a system uses a platform. PSMs are used for generating the system artifacts (e.g. code).

Each abstraction model can be transformed to one or more subsequent models. OMG suggests to use marks for guiding the transformation process of PIM to PSM, [Obj03]. Marks are platform specific. Therefore an architect would take a PIM and mark several elements (e.g. with stereotypes) to indicate their role. The transformation would then be carried out, based on marks and a mapping (specification of a transformation), to generate a PSM. The transformation of CIM to PIM is not covered by the MDA Guide.

A goal of MDA is to provide a framework, which integrates the existing OMG standards, [Ken02]. Therefore the intended modeling language is UML. UML can be further specialized and extended for domain specific purposes. We explain that case in Section 2.3.

2.2.5 Model–Driven Testing

Model–based testing (MBT) is a form of black–box testing (see Section 2.1). Utting et al. [UL07] distinguish between four main approaches known as MBT:

- Generation of test input data from a domain model. The model contains information about input values concerning a specific domain. The generation process consists of selecting and combining of these values to generate useful input data for tests.
- Generation of test cases from an environment model. In that case, the model describes the expected environment of a SUT. An algorithm tries to generate sequences of calls to the SUT. Utting et al. [UL07] mention that this approach cannot *predict the output values because the environment model does not model the behavior of the SUT*. Therefore only a crash/no-crash verdict may be possible to make.
- Generation of test cases with oracles from a behavior model. The model must describe the expected behavior of a SUT, for instance the relationship between input and output values. Based on such a model, executable test cases which include oracle information can be generated. Oracle information can be represented in the form of expected output values or automatic checks on output values, to test if they are correct. Utting et al. [UL07] state that this approach is *a more challenging task than just generating test input data or test sequences that call the SUT but do not check the results*. The obvious advantage of such an approach is that the whole process of specifying tests can be automatized.
- Generation of test scripts from abstract tests. Abstract tests are models of concrete test cases or test suites. This approach focuses on transforming test models into low–level executable test scripts. In our opinion, the advantage is that a test engineer does not have to consider implementation details and can concentrate on the relevant behavior of a test case. Further, an abstract test can be transformed into various target platforms, without reimplementing a test case.

An extensive taxonomy of MBT can be found in the book [ZSM11].

Zander et al. [ZSM11] mention that *most published case studies illustrate that utilizing MBT reduces the overall cost of system and software development*. A typical benefit of using MBT techniques is about 20%–30% cost reduction. Such a benefit may even increase to 90%.

Model–driven testing (MDT) is MBT, but in the context of MDA and OMG standards, [ZSM11]. Dai [Dai04] explains *the philosophy of MDA can be applied both on system modelling and test modelling*. Figure 2.8 highlights this approach. Like in MDA, the modeling language of choice is UML. For the specification of test cases and test suites, an UML profile called UML Testing Profile (UTP, see Section 2.4) is used. A platform independent test design model (PIT), describing test cases on an abstract level, can be derived from a PIM or built independently. The PIT may be transformed to a platform specific test design model (PST) or directly to test code, if a PST is not necessary. An additional transformation from a PSM to a PST can be used for enhancing a PST. After each transformation step, the resulting test design model can be further refined, e.g. test specific properties can be added or test behavior can be completed. Note that MBT techniques can be used to generate test design models.

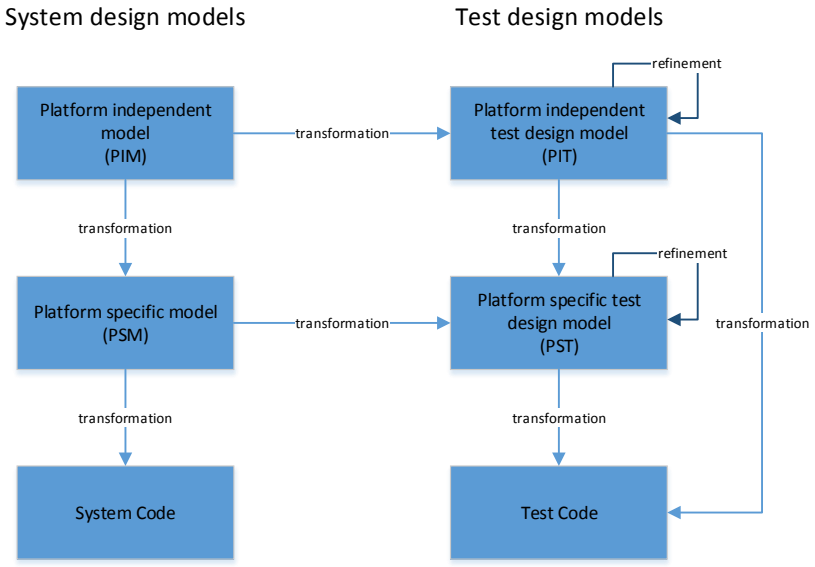


Figure 2.8: MDT architecture (based on [Dai04])

In this thesis we use the MDT architecture, except that system design models do not exist in our solution. In our case a test engineer has to manually create a PIT, which could be transformed to a PST or directly to code. We demonstrate the latter case in the practical part.

2.3 Unified Modeling Language

The Unified Modeling Language (UML) is a general–purpose modeling language, standardized by the OMG.

The development of UML was started at the year 1994, by Booch, Rumbaugh, and shortly after supported by Jacobson, [PP05]. At the time a lot of different modeling approaches and languages existed, which were mostly incompatible. Miles et al. [MH06] mention, *this time of confusion and chaos in the software modeling world is now rather dramatically referred to as the "method wars"*. The OMG accepted UML in 1997 and released it as version 1.1. Since then, many revisions have been released, most notably version 2 of UML in the year 2005, which was a huge improvement and is not compatible with version 1. The current release of UML, at the time of writing, is 2.4.1, which was standardized 2011. We use this version of UML and all following explanations of UML are referring to version 2. UML is still getting improved and updated. The next version will be 2.5, which is currently in a beta state.

In the following subsections, we first explain the specification of UML. Then we give a brief overview of the different diagram types. The last subsection is about UML Profiles.

2.3.1 Specification

UML is actually defined by four complementary specifications, [Obj11a]. All specifications can be found on the OMG website and are freely available at `http://www.omg.org`.

UML Infrastructure This specification [Obj11a] describes the structure and contents of the Infrastructure Library package, which is used for the UML metamodel and also for related metamodels, like the Meta Object Facility (MOF) or the Common Warehouse Metamodel (CWM), [Obj11a]. MOF is the intended metamodel for UML and resides on metamodeling layer M3 (see Section 2.2.3). The primary target audience of that specification are tool vendors.

UML Superstructure The Superstructure specification [Obj11b] is the formal definition of UML elements, their relations and diagrams. It is the authority of UML and weighs over 700 pages.

Object Constraint Language The Object Constraint Language (OCL) [Obj12c] is a simple language for defining constraints and expressions on model elements. For instance, it is used for restricting possible values of properties or parameters. It is also used by the Superstructure to define constraints on elements, which are both human–readable and machine–readable.

Diagram Definition That specification [Obj12b] is about the graphical syntax. It defines precisely what graphical elements are available, but not how their visual appearances look like. Further it enables the interchange of graphical information.

As mentioned in Section 2.2.4, UML leverages XMI for storing and exchanging models.

The work in [PP05] explains, *it is important to realize that while the specification is the definitive source of the formal definition of UML, it is by no means the be–all and end–all of UML. UML is designed to be extended and interpreted depending on the domain, user, and specific application.* There are often several ways to represent a concept in UML, where one has to choose which way suits the best. Pilone and Pitman [PP05] mention it is intended that *there is enough wiggle room in the specification to fit a data center through it.*

2.3.2 Diagrams Overview

An UML model does not have to be build by using diagrams. Diagrams are just the concrete syntax of the abstract UML syntax. It is possible to set the abstract syntax directly. Technically these two syntaxes are often kept separate, resulting in a XMI file containing a model and a corresponding XMI file for the mapping of a model to graphical elements and diagrams. In the practical part of this thesis, we generate UML models, but do not create graphical mappings. However, to explain UML concepts and elements, we present briefly the possible diagram types. The relevant diagram types for the practical part are Class, Object, Package and Sequence Diagram.

UML diagrams can be classified into two categories: structure (static) and behavior diagrams (Figure 2.9). Note that the boundaries between the different kinds of diagram types are not strictly enforced, [Obj11b]. Often, one *can legally use elements from one*

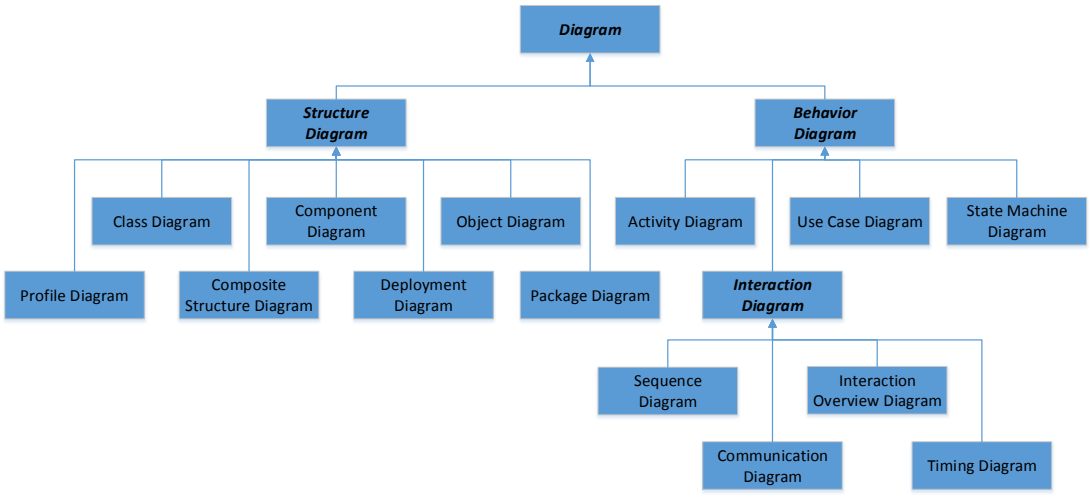


Figure 2.9: Taxonomy of UML diagrams (based on [Obj11b])

diagram type on another type, [Fow04]. Also elements in a structure diagram can reference behavior diagrams or the other way around.

Structure diagrams show the static arrangement of the elements of a system. These diagrams do not show how elements interact in respect of time. Brambilla et al. [BCW12] explain that *they are used extensively in documenting the software systems at two main levels*:

- One level of diagrams emphasize the *conceptual items* of interest of a system. These diagrams provide *a description of the domain and of the system in terms of concepts and their associations*, [BCW12]. The following diagrams describe this part:
  - **Class diagrams** describe the structure of classes and interfaces of a system, including their attributes, operations and relationships. Figure 2.10 illustrates an example class diagram. UML is designed to be independent of any specific object–oriented programming language. For instance, UML interfaces support properties, while Java interfaces do not support properties. A class, like A, realizing an UML interface, has to implement operations the same way they are specified by the interface. An interface property could also be realized by several operations, because they represent a state of a realizing class. The class Y is an abstract class and cannot be instantiated. Class B generalizes from Y and inherits the non abstract operation *doSomething*. Parameters of operations can have four directions: direction **in** indicates that parameter values are passed into a behavioral element (like an operation); **inout** indicates that values are passed into a behavior element and then back out to the caller; **out** indicates that values are passed from a behavioral element to the caller; **return** indicates that values are passed as return values to the caller, [Obj11b]. Return parameters are visualized at the end of an operation.
  - **Composite Structure diagrams** describe the internal structure of classes and the collaboration with their environment.

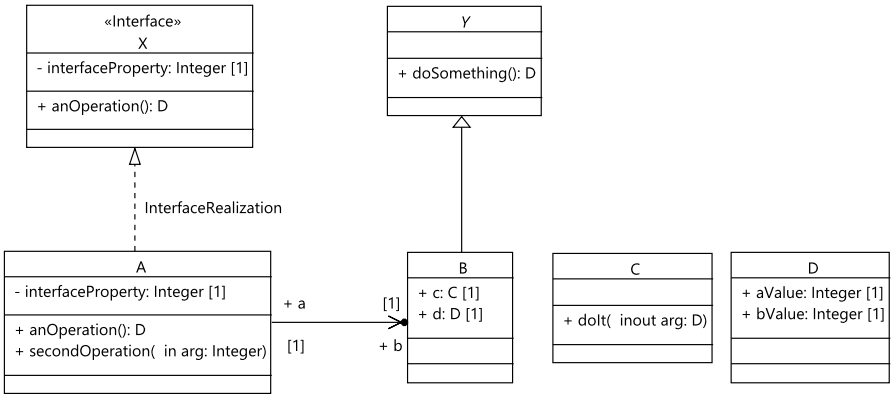


Figure 2.10: UML Class Diagram example

– **Object diagrams** consist of instance specifications and their associations. An instance specification represents an object of one or more classifiers, e.g. a class, at a point in time (snapshot). Some tools allow the modeling of instance specifications with Class diagrams and do not provide dedicated Object diagrams. Figure 2.11 demonstrates instance specifications of the classes specified in Figure 2.10. With instance specifications it is possible to set concrete values through slots, but it is not mandatory to set all properties of classifiers. Associations of classifiers are specified as links on instance specifications. The concrete name of an instance specification is optional and may be empty.

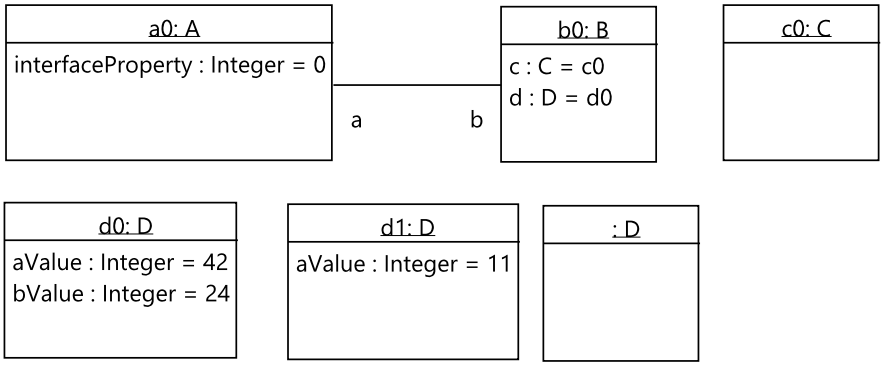


Figure 2.11: UML Object Diagram example

- The other level of diagrams deals with the *architectural representation* of a system. They provide *a description of the architectural organization and structure of the system*, [BCW12]. These diagrams aggregate or base on the conceptual modeling, that is done on the level explained above. The following diagrams describe this level:

  – **Component diagrams** illustrate *how a software system is split up into components and show their dependencies*, [BCW12]. A component is a modular, reusable and replaceable part of a system that encapsulates the content.

  - **Package diagrams** show packages and their dependencies. Packages group elements and provide namespaces for them.
  - **Deployment diagrams** describe how software artifacts are assigned to nodes. Artifacts represent physical elements, which are the result of a development process, while nodes can represent either hardware devices or software execution environments, [Obj11b].

Behavior diagrams show the dynamic aspects of objects of a system, including their methods, collaborations, activities and state histories, [Obj11b]. Such dynamic interactions can be described as a series of changes over time, [Obj11b]. Usually a behavior diagram does not model all behaviors at once. *Every model describes one or few features of the system and the dynamic interactions they involve*, [BCW12]. The following Behavior diagrams exist:

- **Use Case diagrams** describe how external actors use a system. *Typically, they are used to capture the requirements of a system, that is, what a system is supposed to do*, [Obj11b].
- **Activity diagrams** show *the overall flow of data and control for performing the task, through an oriented graph where nodes represent the activities*, [BCW12]. A task must have a specific goal.
- **State Machine diagrams** are used to describe *the states and state transitions of the system, of a subsystem, or of one specific object*, [BCW12]. This diagram type can also express usage protocols, [Obj11b].
- *Interaction diagrams* emphasize the flow of control and data among the elements of a system, [BCW12]. The following diagrams are Interaction diagrams:
  - **Sequence diagrams** illustrate the communication between participants, called lifelines, through a sequence of messages. Figure 2.12 shows the behavior of the operation *anOperation()*, which is part of class A, specified in Figure 2.10. The horizontal axis of the example diagram shows the messages between lifelines, while the vertical axis visualizes the sequential ordering of messages over time. The name **self** of the leftmost lifeline plays a special role. OMG defines that a lifeline named with the keyword **self** represents the object of the classifier that encloses the interaction that owns the lifeline, [Obj11b]. In our case class A owns the operation *anOperation()* and therefore encloses the example interaction. So the lifeline **self** represents class A. The other two lifelines represent properties of the classes A and B. The arguments of method calls have to be concrete value specifications, like instance specifications, expressions or literals. In our case we could take an instance specification of class D as variable. Combined fragments are used to describe a number of traces in a compact and concise manner, [Obj11b]. In the example, we use a combined fragment of type loop (an interaction operator) to specify that the operation *doIt()* is called 20 times. Sequence diagrams allow the specification of ingoing and outgoing message through gates. In our example we use a return message to specify the return of an object of type D. The target of this message is a gate.

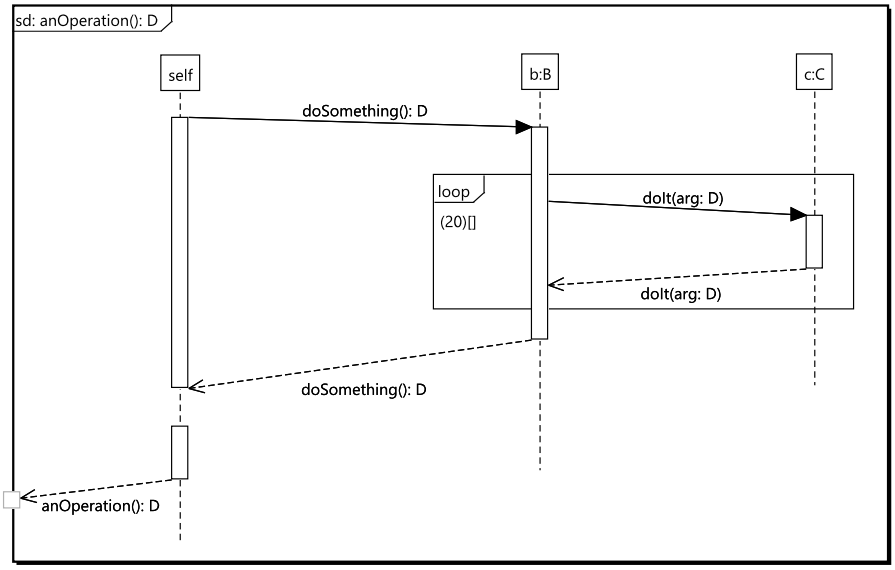


Figure 2.12: UML Sequence Diagram example

- **Communication diagrams** are like simple sequence diagrams, except that they illustrate lifelines and messages similar to Class or Object diagrams. They cannot show combined fragments and the messages have to be numbered, otherwise a designer cannot distinguish which message comes first.
- **Interaction Overview diagrams** describe the control flow between different Interaction diagrams, which are visualized as nodes.
- **Timing diagrams** are a special form of Interaction diagrams that focus on timing constraints and state changes. *Timing diagrams are most often used with real–time or embedded systems*, [PP05].

2.3.3 UML Profiles

Profiles are a *mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform or method*, [Obj11a]. This mechanism can be used to describe a DSL based on UML. Note that profiles cannot change the metamodel and existing constraints. This could be done with MOF (meta layer M4), where one could add or remove metaclasses, relationships and constraints. The drawback of manipulating an existing UML metamodel is that the compatibility with UML tools breaks, because other tools would work with a different metamodel. OMG mentions several reasons for customizing the metamodel with profiles, [Obj11a]:

- Create a terminology that is adapted to a particular platform or domain.
- Give a syntax for constructs that do not have a notation.
- Specify a different graphical notation for already existing symbols. For instance to use a picture of a computer instead of the ordinary node symbol.

- Add semantics that is left unspecified in the metamodel.
- Add semantics that does not exist in the metamodel.
- Add constraints that restrict the way the metamodel and its constructs are used.
- Add information that can be used for model transformations.

Technically profiles are located at meta layer M1, but behave like they would be part of layer M2. A profile itself is a package for stereotypes and constraints, which could be further grouped in packages. Also profiles can import other profiles and packages. To use a profile, it first has to be applied on an UML package and, after that step, one can apply stereotypes on model elements and validate a model against constraints contained in a profile. Note that stereotypes and constraints can be seen as a form of metadata, therefore, if a model is exchanged with a different UML tool without the corresponding profile, it is still a valid model. In that case, the missing stereotypes would be ignored.

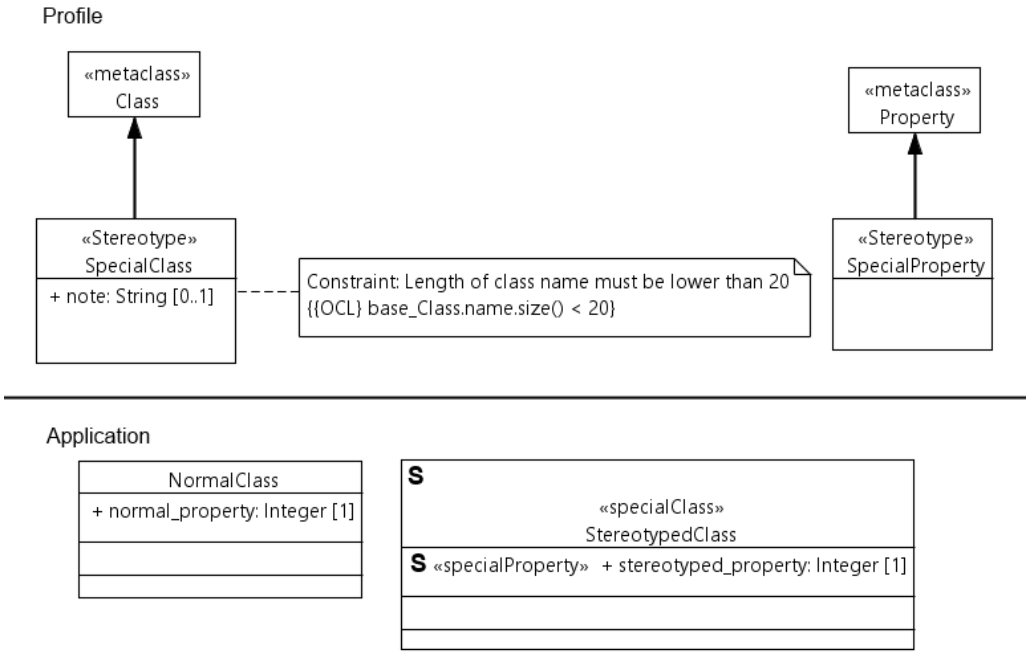


Figure 2.13: Stereotype examples and their application

OMG specifies that *a stereotype defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass*, [Obj11a]. The upper half of Figure 2.13 illustrates how stereotypes and constraints are defined, while the lower half visualizes how these stereotypes are applied. The example stereotypes extend the metaclasses *class* and *property*. Extensions can be marked as required, then the corresponding elements are automatically stereotyped. Metaclasses are usually part of an UML metamodel modeled with UML. Using the metamodel implemented with MOF is often not feasible, as it would cross meta boundaries. However, OMG does not give any restrictions how metaclasses are

actually implemented, [Obj11a]. The stereotype *SpecialClass* holds an attribute, in the literature often referred to as tagged value, which can be used to specify metadata. Such metadata can be relevant for instance in model transformations. The OCL constraint in the example checks if the name of a stereotyped class has less than 20 characters. Stereotypes can also provide custom icons and shapes, like the icons which are visualized on *StereotypedClass*. We added *NormalClass* in the example to highlight the differences to stereotyped elements.

2.4 UML Testing Profile

The UML Testing Profile (UTP) is a standardized profile (DSL) based on UML. According to the specification [Obj13c] by the OMG, it can be used *for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts commonly used in and required for various testing approaches, in particular model-based testing (MBT) approaches.*

The development of UTP was started by a consortium consisting of the organizations Ericsson, Fraunhofer FOKUS, IBM/Rational, Motorola, Telelogic and University of Lübeck, [BDG+08]. OMG standardized UTP in the year 2005. At the time of writing, the current version of UTP is 1.2, released in the year 2013, and we only refer to this version.

UTP is strongly influenced by TTCN–3 (see Subsection 2.1.3) and other testing techniques, [SDGR03], but not limited to them. The UTP specification actually defines mappings to TTCN–3 and the Java unit testing framework JUnit, [Obj13c]. UTP can be used on all levels of the V–Model (see Subsection 2.1.2), [BDG+08].

OMG enumerates several cases how people may use UTP along with UML, [Obj13c]:

- Specify the design and the configuration of a test system.
- Build the model-based test specification on top of already existing system models.
- Model test cases.
- Model test environments.
- Model deployment specifications of test–specific artifacts.
- Model test data.
- Provide necessary information pertinent to test scheduling optimization.
- Document test case execution results.
- Document traceability to requirements and other UML model artifacts.

In the practical part of this thesis, we use UTP for modeling the test architecture and behavior. For modeling the test data, we use instance specifications, without using any dedicated UTP concepts. The test management concepts of UTP are out of the scope of the practical part.

The following subsections have the same structure and title like the sections of the UTP 1.2 specification [Obj13c]. Each subsection summarizes the corresponding stereotypes of the specification.

2.4.1 Predefined Type Library

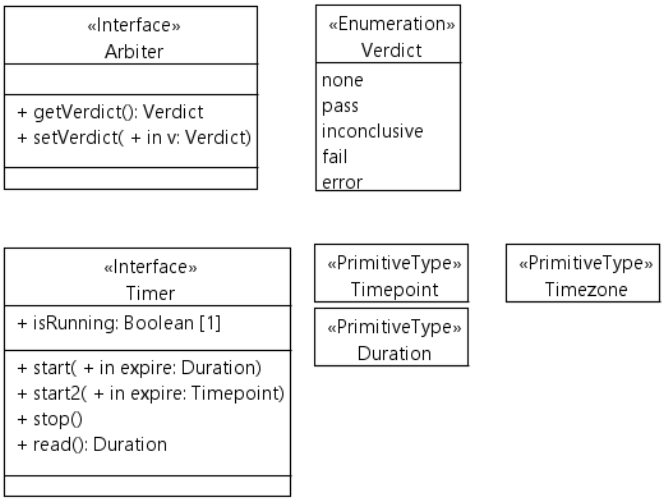


Figure 2.14: UTP type library

The type library is an UML package, which provides interfaces and types for the UTP profile and modeled test cases. Figure 2.14 illustrates this library. Arbiters are used for assigning the final verdict to a test case. The *setVerdict* operation can be used in a test behavior to update the current test case status. The arbitration algorithm for calculating the final verdict can be user–defined. UTP suggests as default the following precedence of verdicts, [Obj13c]: Pass < Inconclusive < Fail < Error.
Timers are used to observe and control test behavior. *Also, a timer can be used to prevent from deadlocks, starvation, and instable system behavior during test execution*, [Obj13c]. When a timer expires, a timeout is automatically sent to the owning active class. A constraint of the timer interface is, that properties realizing the timer interface can only be owned by test components and test contexts.
The primitive type *Timezone* may be used to group test components together, which are considered to be synchronized. By default it is illegal to compare time–critical events from different timezones.

2.4.2 Test Architecture

Test architecture concepts *are essential to specify structural aspects of a test environment and a corresponding test configuration in order to embed and execute test cases against a system under test*, [Obj13c]. A test environment consists of elements which are necessary to conduct test cases. A test configuration describes how these elements are connected with a SUT. Figure 2.15 illustrates test environment and test configuration. UTP does not provide any concepts for designing a SUT, this could be done with UML. However, UTP is used for black–box testing, therefore only the public interfaces of a SUT are necessary. *The system under test may represent a single component (component testing), a cluster of components (integration testing) or a complete system (system testing)*, [Obj13c]. The complexity of a test configuration is always on the same level, regardless if a SUT consists

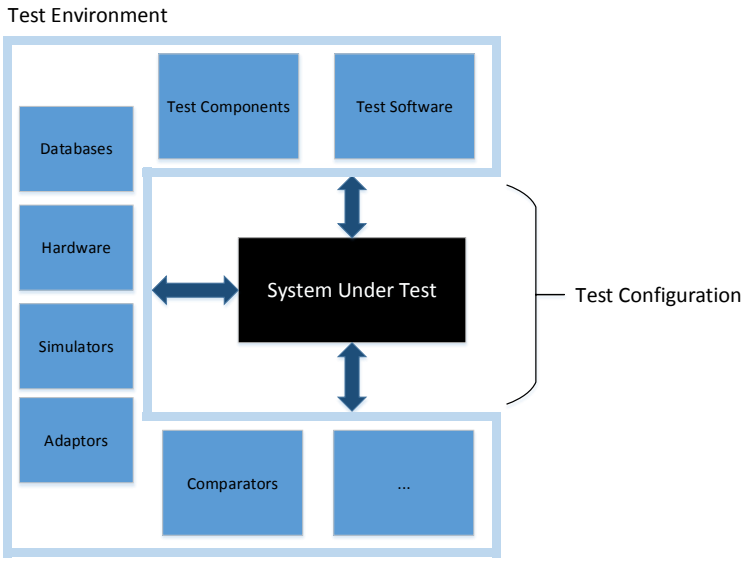


Figure 2.15: Concepts of test environment and test configuration (based on [Obj13c])

of a single component or several components, [Obj13c].

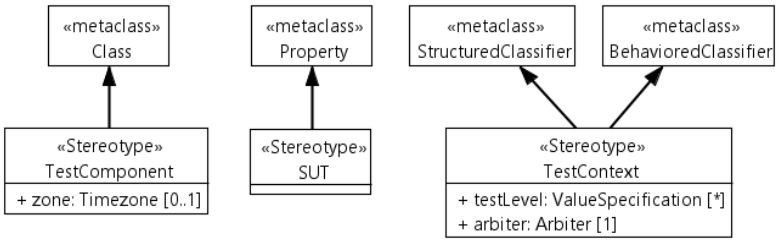


Figure 2.16: Test Architecture stereotypes

OMG defines three stereotypes concerning the test architecture (Figure 2.16). The stereotype *TestComponent* is used to tag classes, components or nodes that are part of the test environment and communicate with other test components or the SUT. The main purpose of test components is to stimulate the SUT with test data and to evaluate whether the responses comply to the expected ones. *In addition, test components can be used to provide auxiliary, user-defined functionality during the execution of a test case*, [Obj13c]. The zone attribute can be used to group different test components in the same timezone. The stereotype *SUT* is applied on properties of classifiers that represent systems under tests. A *SUT* property is part of a corresponding test context. *No internals of an* [sic] *SUT are known or accessible during test case execution, due to its black-box nature*, [Obj13c]. A *TestContext* stereotype can be applied on UML elements which are both a structured and a behaviored classifier, e.g. a class. It is used to group a set of test cases. The composite structure of a test context may be used for specifying the test configuration, which can illustrate the connections between test components before a test case is started and the maximal number of connections and components during the test

execution. In the practical part of this thesis we omit the composite structure. We apply the stereotype *TestContext* on classes, which hold the relevant components as properties and group test cases as operations. The *testLevel* tag can be used to specify the phase of testing process where the text context resides. For instance, a custom value could be "Component Testing". A test context can be part of different test levels. The *arbiter* tag has to be used for specifying the concrete arbiter implementation.

2.4.3 Test Behavior

Test behavior concepts are used to stimulate and observe a SUT in conjunction with a test configuration specified by a test context. All behavior diagrams may be used to specify the test behavior. Concerning the practical part, we decided to only use interactions. These interactions could be visualized as Sequence Diagrams.

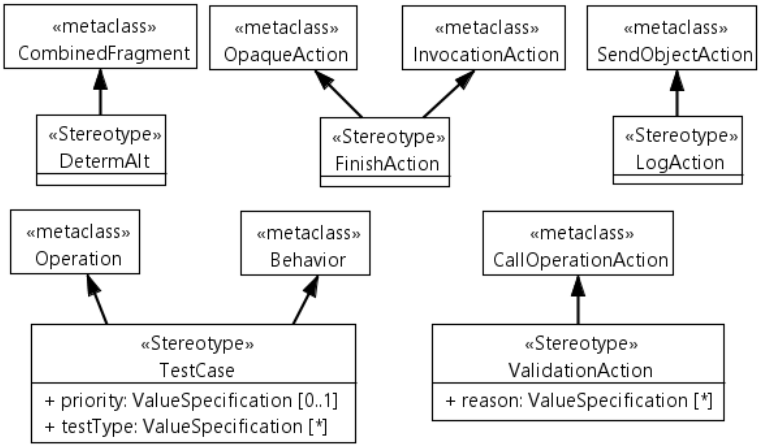


Figure 2.17: Test Behavior foundation

Figure 2.17 illustrates the foundation of the test behavior stereotypes. The stereotype *TestCase* is used to indicate test cases. A test case is a concrete specification of a test on a system, *including the required test behavior with its test inputs, test conditions, and test result*, [Obj13c]. It has to be owned by a test context and therefore has full access on the test configuration, global variables and helper methods. If *TestCase* is applied on an operation, it cannot be applied on the corresponding behavior. A behavior also has to be owned by the same test context. The return parameter of a test case has to be a verdict. This verdict can be calculated by an arbiter or it is provided by the behavior. The *priority* tag may be used to plan the execution order of several test cases. *A test type indicates what concrete quality criteria are going to be verified by the related test case*, [Obj13c]. A test case may realize a test objective which we discuss in Subsection 2.4.5. The actions visualized in Figure 2.17 are used in the behavior of a test case to trigger corresponding classifier. *ValidationAction* sets the verdict of a test case by calling the *setVerdict* operation of an arbiter. An optional reason may be added. *LogAction* can be used to specify what should be logged. The logging facilities can be implicitly implemented by the run–time system or explicitly provided by a model. *FinishAction* immediately completes the test case for a corresponding test component. Other components are not affected by

a finish action, this has to be modeled explicitly. *DetermAlt*, short for deterministic alternative, *is a CombinedFragment where the operands are evaluated in exact the same order as they appear in the model (respectively diagram) regardless of the fact that the guards of more than one InteractionOperand evaluate to true*, [Obj13c]. The evaluation of the guards starts after the involved components receive an event, like a message or time out of a timer. We do not use this stereotype therefore we refer to the UTP specification for a detailed explanation.

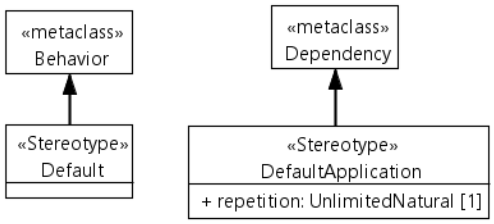


Figure 2.18: Test Behavior defaults

UTP further defines the concept of default behaviors (Figure 2.18). A test case specifies the expected behavior of a SUT, but if an unexpected behavior, which is not specified by the test case, occurs, then a default behavior should handle such a situation, [Obj13c]. The stereotype *Default* can be applied on all kinds of behavior, for instance State Machines or Interactions. *DefaultApplication* can be used to associate a default behavior with a static structural behavior, like an interaction fragment, action or state. We omit this concept in the practical part, therefore we do not discuss these stereotypes in more detail.

For manipulating and accessing timers and timezone tags several action stereotypes are defined by the UTP specification (Figure 2.19). UML already offers time constraints, but the semantics are quite different to the time concepts of UTP. If an UML time constraint does not hold, the entire behavior fails, while the UTP time concepts allow reacting on time constraint violations. Therefore when a UTP timer fails it does not mean that the whole test case fails (of course in most cases it does mean that), [Obj13c]. The *TimeOut* event is meant to be used in State Machines, while the *TimeOutMessage* could be used in Interactions if a timer fails.

2.4.4 Test Data

The intended way to specify test data is to use instance specifications. Data values could be specified in–line within a behavior (e.g. by using literals), but *this approach can be error prone, decrease readability, as well as leading toward duplication*, [BDG+08].

According to the UTP specification test data can serve different purposes, [Obj13c]:

- Data can be supplied with a stimulus, i.e. sent by a test case to a SUT, or retrieved as a response, i.e. values returned by a SUT.

- Data can be used to define the initial state of a SUT needed to start a test case, i.e. the precondition of a test case, and the expected state of a SUT after a test case terminates, i.e. the postcondition of a test case.

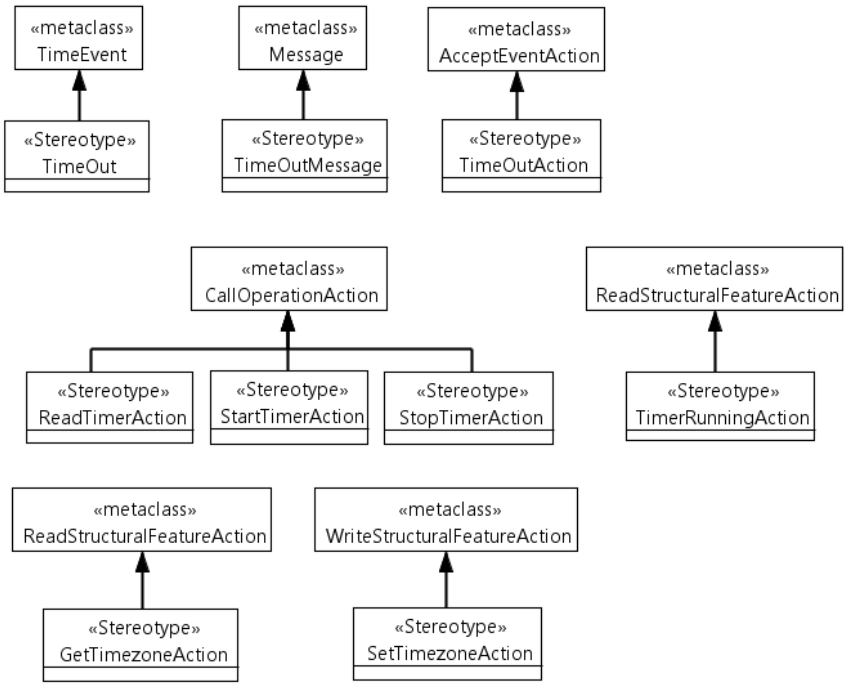


Figure 2.19: Test Behavior timer and timezone stereotypes

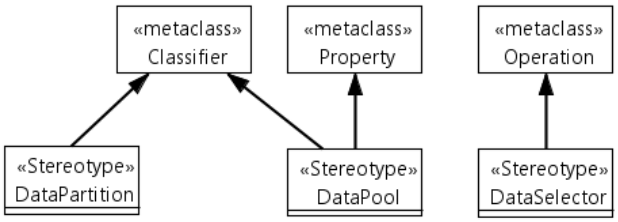


Figure 2.20: Test Data structural specification stereotypes

UTP provides three stereotypes for managing test data. These stereotypes aim at the structural aspects of test data (Figure 2.20). *A data pool is a classifier containing either data partitions (equivalence classes), or explicit values; and can only be associated with either a test context or test components*, [Obj13c]. The structure of a *DataPool* may be a simplified view of the actual container or database scheme. This concept is mainly used when a test case is invoked repeatedly and where a data pool would provide different data values for stimulating a SUT. A data partition is a container for a set of similar data values. Different data partitions are meant to be used as equivalence classes in test cases. A data partition can only be associated with a data pool. *A data selector allows the implementation of different data selection strategies*, [Obj13c]. It can be only be applied on operations hold by a data pool or data partition.

Further, UTP defines four stereotypes which deal with test data values (Figure 2.21). *A coding rule specifies how values are encoded and/or decoded*, [Obj13c]. The tag *coding* refers to the used coding scheme. The stereotypes *LiteralAny* and *LiteralAnyOrNull* are

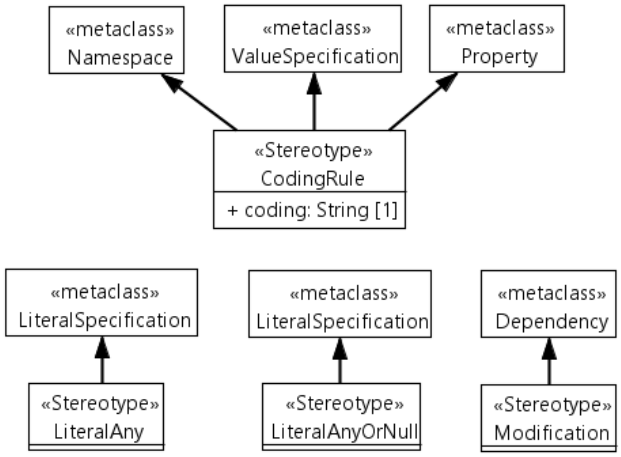


Figure 2.21: Test Data values stereotypes

wildcards. Wildcards allow any possible concrete data or explicitly the absence of data values, [Obj13c]. The dependency stereotype *Modification* is a relationship between two instance specification. For instance, it used if a new instance specification slightly modifies or completes an already existing instance specification. The new instance specification can overwrite, add or reuse slot values. This stereotype allows to create large sets of data by avoiding redundancy, [Obj13c]. The instance specifications must have compatible types and cyclic modifications are not allowed.

2.4.5 Test Management

The UTP test management concepts aim *at the narrow scope of managing individual test activities and tests (i.e., not the full project/test lifecycle). Test management is needed since it is impossible to test a software system exhaustively, thus testing becomes a sampling activity, which must be managed within cost, schedule, qualities, resources (human and facilities), and risk aspects*, [Obj13c]. In general there exist three test management activities, [Obj13c]:

- Test Planning and Scheduling
- Test Monitoring and Control (including test execution)
- Test Results Analysis

Concerning the test planning and scheduling activity, UTP defines the stereotype *TestObjective* (Figure 2.22). Test objectives textually specify the reasons, purposes and targets of test cases. *Test objectives must be measurable, thus, they must include sufficiently precise information how to assess, respectively evaluate the behavior or reaction of the system under test*, [Obj13c]. However, they do not prescribe how a test case realizes a test objective technically. Test objectives can be explicitly linked to test cases. They should be specified before the actual test cases are constructed and executed. Later they can ensure the traceability between requirements and test artifacts. The *priority* tag may be used to specify the execution order of several test cases (scheduling).

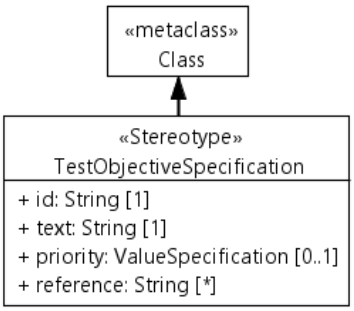


Figure 2.22: Test Management TestObjectiveSpecification stereotype

The test monitoring and control activity ensures that the testing process is performed like it is defined by the test plans and models, [Obj13c]. UTP does not define any dedicated concepts which support this test management activity explicitly, but the existing UTP concepts can be leveraged.

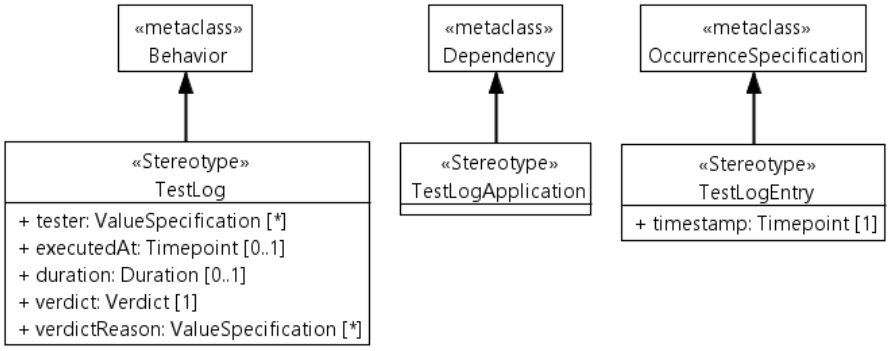


Figure 2.23: Test Management test log stereotypes

UTP supports the test results analysis activity with three stereotypes, which are meant to capture test results from test case executions (Figure 2.23). *A test log is a fixed (immutable) behavioral description resulting from the execution of a test case*, [Obj13c]. It contains several test steps, which can be any events of interest that happened during test execution. For instance, a test log might be an interaction representing the executed operation calls on the SUT. The test log may be automatically generated by a test execution system. Test steps can be stereotyped with *TestLogEntry*, to enable a more precise analysis of the corresponding test log. A constraint of *TestLogEntry* is that it is only allowed to be applied on occurrence specifications which are visible to the test environment. Thus it cannot be applied on occurrence specifications inside a SUT. Test logs can be associated with test cases or test contexts by using a *TestLogApplication*.

2.5 Eclipse Modeling Project

The Eclipse Modeling Project (EMP, [EMP]) is a top–level project at Eclipse, [Gro09]. It *focuses on the evolution and promotion of model–based development technologies within*

the Eclipse community by providing a unified set of modeling frameworks, tooling, and standards implementations, [EMP]. EMP offers a wide variety of modeling technologies grouped in the following sub–projects, [EMP]: Abstract Syntax Development, Concrete Syntax Development, Model Development Tools, Model Transformation, Technology and Research, and Amalgam. All technologies are licensed under the Eclipse Public License and therefore open–source. Like most Eclipse projects, they are based on the Java platform. The open–source nature of the EMP and the numerous tools build around or compatible with the Eclipse Modeling Framework make it unique in the modeling world.

In the following subsections we discuss the Eclipse Modeling Framework, UML2, Xtext and different model–to–text tools.

2.5.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF, [EMFb]) is the heart of the EMP ecosystem and belongs to the Abstract Syntax Development project. EMF is essential for all other modeling techniques which are part of the EMP. It basically provides a meta–modeling language called Ecore. Further, EMF includes code generation facilities to automatically generate a Java implementation (API) of an Ecore model, a simple tree editor, and adapters for editing and displaying generated model classes. The Java implementation is used to create, serialize/deserialize and to manipulate models based on Ecore models. Per default, models are serialized to XMI, which can be customized and replaced with other storage formats.

Ecore models can be generated from various sources. EMF provides an extensible import framework for creating Ecore models from different model formats, [SBPM08]. Importers exist for instance for annotated Java code, Eclipse UML2 models, XML schemas and Rational Rose. Rational Rose is supported, because it was used to bootstrap the implementation of EMF. Of course, Ecore models can also be directly created. The EcoreTools project ([Eco]) provides a graphical modeling editor, which makes it possible to define a model like an UML class diagram.

Historically EMF was started by IBM to develop an implementation of MOF. But the MOF model was very large and too complex, therefore they developed a drastically simplified version of MOF named Ecore. Based on their work, OMG split MOF into EMOF and Complete Meta Object Facility. Ecore and EMOF are quite similar, therefore Ecore can read and write standard EMOF serializations, [EMFa]. EMF became an Eclipse project in the year 2002, [SBPM08].

As stated above, Ecore is a meta–modeling language and resides on layer M3 (see Subsection 2.2.3). Ecore is defined reflexively, so it can describe itself. Figure 2.24 shows an overview of the main Ecore concepts. Note that this Figure does not represent the actual Ecore implementation.

According to Brambilla et al. [BCW12], the main concepts are EClassifier including EClass, EDataType and EEnum, as well EStructuralFeature including EReference and EAttribute. Ecore has been developed with Java in mind, so all concepts seamlessly integrate with it. EClass is similar to a Java class, it can have EAttributes, EReferences to other EClasses and EOperations (not shown in Figure 2.24) for defining methods. Additionally EClasses can inherit from multiple other EClasses (eSuperTypes). An EClass can be abstract and an interface. Note that an EClass has to be abstract if it is an interface.

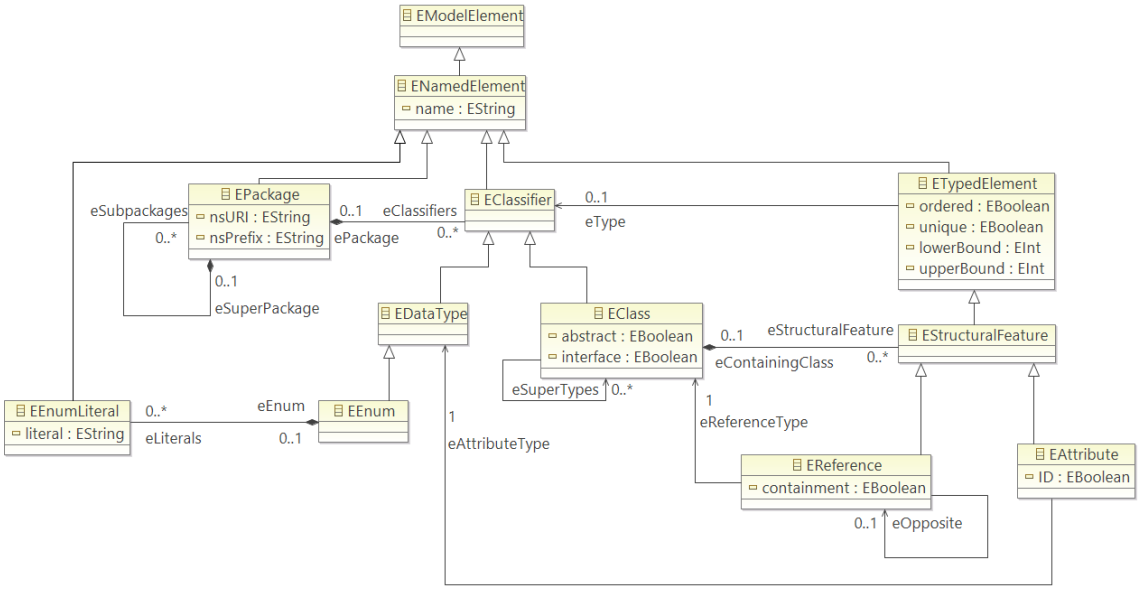


Figure 2.24: Ecore overview (based on [BCW12])

An EReference to an other EClass can be a containment, that means an EClass A *owns* a referenced EClass B, otherwise EClass A just *knows* referenced EClass B. EReference can also know an opposite EReference (eOpposite). For instance if EClass A *owns an* EClass B, EClass B could know EClass A through an additional EReference. EMF would automatically take care of such a relationship if marked as opposite. All EStructuralFeatures have the multiplicity settings of ETypedElement in common. An EAttribute has to have an eAttributeType of EDataType. EDataTypes are primitives like Integer or String. They cannot have any attributes and represent data. Ecore already offers primitive Java types and collections, like list or map, as EDataTypes. Custom EDataTypes have to map to an existing Java class. In general, it is better to use EClasses for complex custom data types, as EMF provides several benefits, e.g. serialization and notification. EEnum is a special EDataType, where the possible values are restricted by the containing EEnumLiterals. It behaves like a Java enum. An EClassifier can be part of an EPackage. EPackages group such elements.

A more comprehensive explanation and overview of EMF can be found in [SBPM08].

2.5.2 UML2

The UML2 project ([MDTb]) is part of the Model Development Tools (MDT) project and implements the OMG UML 2.x metamodel based on EMF. This project *serves as the de facto "reference implementation" of the specification and was developed in collaboration with the specification itself*, [Gro09]. The objectives of the UML2 project are to provide, [MDTb]:

- a useable implementation of the UML metamodel to support the development of modeling tools
- a common XMI schema to facilitate interchange of semantic models

- test cases as a means of validating the specification
- validation rules as a means of defining and enforcing levels of compliance

UML2 only provides the metamodel, but does not provide any UML modeling tools. MDT provides an UML modeling tool called Papyrus ([Pap]) for UML2. We use Papyrus for the UML illustrations in this thesis.

Several commercial or open–source UML modeling tools can import/export UML2 compatible models. A list of these tools can be found at [MDTa]. Prominent commercial tools which support UML2 are for instance Enterprise Architect, MagicDraw UML, and IBM RSM/RSA. Note that the graphical representations of UML models can most of the time not be interchanged between modeling tools. For instance, a diagram (concrete syntax) drawn with Papyrus cannot be opened by Enterprise Architect, but the underlying UML2 model (abstract syntax) is supported. In that case a user would have to create a new diagram based on the model with Enterprise Architect. This is an issue which the OMG tries to solve with the UML 2.5 specification.

2.5.3 Xtext

The Xtext project ([Xted]) is part of the Concrete Syntax Development project of the EMP since 2008. It is a so called language workbench for designing textual languages, ranging from DSLs to GPLs. Xtext is mainly developed by the Itemis AG, who offers services and consulting around Xtext. Itemis AG is a Strategic Member of the Eclipse Foundation and employs several Eclipse commiters of which some (in 2010 four commiters) are working full time on Xtext, [ER10]. *Companies like Google, IBM, BMW and many others have built external and internal products based on Xtext*, [EvdSV+13]. Concerning the practical part we use an Xtext version based on version 2.

The first step to create a textual language with Xtext is to define the syntax with Xtext's grammar definition language. The grammar language is *similar to EBNF, but with additional features to achieve a similar expressivity as metamodeling languages such as Ecore*, [BCW12]. We discuss details of the grammar language when we present Ubtl in the practical part. In general, a new Ecore model (layer M2) is derived from a specified grammar, which describes the structure of its abstract syntax tree (AST), [Xtec]. Alternatively, it is possible to use an existing Ecore model to define a textual syntax for it. This can be useful to provide an optional textual representation for an Ecore modeling language which otherwise can only be represented graphically. In that case no additional Ecore model is created and the framework operates on the existing one. The grammar language also supports the reuse of already existing grammars. Further, an ANTLR v3 grammar is inferred from an Xtext grammar. Xtext leverages ANTLR v3 (**AN**other **T**ool for **L**anguage **R**ecognition) to generate a text–to–model parser written in Java. ANTLR v3 relies on an LL(*) algorithm for generating the parser, [Bet13]. Additionally, Xtext generates a model–to–text serializer. The Xtext framework seamlessly integrates these components into the EMF environment. An *Xtext model just looks like any other Ecore–based model from the outside, making it amenable for the use by other EMF based tools*, [Xtec].

As mentioned above Xtext uses an Ecore model to represent a textual language in–memory and this Ecore model represents the AST. Xtext generates a whole customizable

compiler infrastructure to realize concepts like code generation/compilation, validation, scoping or code formatting, which use the Ecore model to fulfill their task. All components can be customized programmatically.

Additional to the compiler infrastructure Xtext generates, based on the grammar, a customizable Eclipse IDE (Integrated Development Environment). The IDE supports features like content assist, quick fixes, an outline view, a compare view, hyperlinking and syntax coloring. For a user a custom Xtext IDE "feels" like the Eclipse Java IDE .

Xtext encourages language designers to test their languages. It provides several facilities which make it easy to test an Xtext language with JUnit. It is even possible to test the IDE with JUnit test cases.

Xtext keeps these three parts, compiler, IDE and JUnit test cases, logically separate in three Eclipse plugins. The advantage of this is that a language compiler can be integrated as a Java library within a normal Java program without any dependencies on the language IDE or a running Eclipse instance.

The programming language of choice for customizing the generated infrastructure is Xtend ([Xteb]). Xtend is a statically typed GPL. Syntactically and semantically it has its roots in the Java programming language, but offers advanced features like lambda expressions (available in Java since version 8), operator overloading or template expressions, [Xtea]. *One of the goals of Xtend is to have a less "noisy" version of Java*, [Bet13]. Xtend is compiled to Java 5 compatible source code. Therefore it is 100% interoperable with Java code and existing Java libraries, [Xtea]. *Xtend itself is implemented in Xtext and it is a proof of concept of how involved a language implemented in Xtext can be*, [Bet13]. Alternatively the generated language infrastructure can still be customized by using the Java programming language.

The Xtext framework and the generated classes are highly customizable because Xtext relies on Google Guice. Google Guice ([Goo]) is a dependency injection framework for Java 5 and above. Dependency injection is a software design pattern and makes it easy to configure the dependencies between objects at runtime. In practice, this means that a language designer can replace every part of Xtext with an own implementation. For more information about Google Guice we kindly refer to the project homepage.

2.5.4 Model–to–Text Tools

In principle it is possible to develop a model–to–text generator in plain Java by using the generated Java implementation of an Ecore model and the EMF libraries. According to Brambilla et al. [BCW12], this approach has several drawbacks. Firstly, there is no separation between static and dynamic code. Static code is generated for every element the same way, e.g. package definitions or imports, while dynamic code is based on current model information, e.g. class name or variable name. Secondly, the structure of a produced code is embedded into the producing code, therefore it is complicated to grasp the final output. Thirdly, a declarative query language is missing, which leads to a lot of type casts, iterators, conditions or loops. Last but not least, the code for reading the input and writing the produced output, has to be written over and over again.

These are reasons why languages dedicated to transform models to text (code) have been developed. We present three such projects of the Eclipse open–source ecosystem.

Xtend

As stated above in Subsection 2.5.3 Xtend is a GPL, which is compiled to the Java programming language. Concerning model–to–text transformations, it offers a feature called "template expressions". Template expressions are multi–line strings, which can be parameterized. Additionally, conditions and loops are supported. A drawback is that reading models and writing the output has to be manually programmed. Also the generated Java implementation of a modeling language and EMF libraries have to be used. An interesting feature is that white spaces are specially marked by the editor, therefore it is easy to grasp the output structure. Xtend (starting with version 2) is the successor of Xpand, a language purely dedicated to model–to–text transformations, [BCW12].

Epsilon Generation Language

The Epsilon Generation Language (EGL, [Eps]) is a template–based model–to–text language with a textual syntax, developed by the University of York. It supports protected areas (preserving hand–written code), formatting algorithms, traceability mechanisms and cached operations. If a cached operation is called with the same arguments, the operation is not processed a second time, instead a saved state is returned. It also has the ability to create and call methods of Java objects. Because it is based on the Epsilon platform, it is not limited to the EMF and can use other meta–modeling languages ([RPKP08]).

Acceleo

Acceleo ([Acc]) is a pragmatic implementation of the OMG standard *MOF Model to Text Transformation Language* (MOFM2T, [Obj08]) for EMF. It is developed and mainly maintained by the company Obeo. It serves as reference implementation of the OMG standard and offers some optional enhancements. Acceleo is part of the Model Transformation project of the EMP. It is template–based and uses OCL for querying models. Like EGL it supports protected areas, cached queries and traceability. If OCL is not enough for calculating a result, it offers the ability to invoke Java code. Though a designer develops Acceleo templates by using a standardized textual syntax, they are actually models based on the metamodel described in the OMG standard. These models are used by an execution engine to carry out transformations. Concerning integration into the Eclipse IDE, Acceleo offers a customizable UI launcher for Acceleo projects.

Chapter 3

UML Testing Profile Based Testing Language

In principle, one can specify test cases with UML and UTP. In our opinion this has some drawbacks. One is that a test engineer has to be trained how to use UML in order to specify test cases. The reason is that UML is quite complex. Another one is that there are several ways to specify the same thing in UML. That is a problem for code generators, because a test engineer could specify constructs or classes which a generator simply does not expect. A solution is that a test engineer has to follow written guidelines in order to specify useful test cases, which makes it, in our opinion, impractical.

Due to that aforementioned drawbacks, we develop the UML Testing Profile Based Testing Language (Ubtl), a textual DSL. With Ubtl the test platform designers can declare types and components, which a code generator knows and transforms correctly. The test engineer on the other side can use those declarations to define variables, runtime components and test cases. Test cases specified with Ubtl are automatically transformed to UML models, in conjunction with UTP, which can then be further processed. Ubtl offers a concise textual syntax which is easier to learn than UML. It reduces the complexity of UML. It is faster to specify test cases with Ubtl than with UML. Additionally, the Ubtl code is automatically validated whether it contains errors or missing properties. Further, the UML models are always generated in the same way from Ubtl code. That makes it easy to process the UML models and to avoid ambiguousness. We also provide a powerful IDE based on Eclipse for Ubtl.

In the following, we first discuss the applications of Ubtl, we identified. After that we give a simplified overview of the Ubtl software architecture. Then we explain the textual language elements of Ubtl. Next we show how the Ubtl compiler maps the resulting Ubtl model to UML. We also present the changes to the generated IDE we made to make Ubtl easier to use. Finally, we explain how we tested Ubtl.

3.1 Applications of Ubtl

Ubtl code is always compiled to UML models. We identify four applications of this approach:

Application one: Figure 3.1 illustrates the first application. A test engineer could spec-

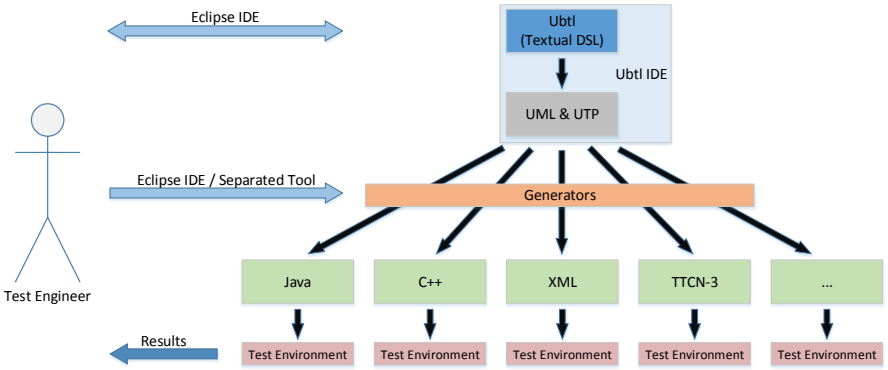


Figure 3.1: Ubtl application number 1 shows how a test engineer could use the Ubtl IDE and different code generators

ify test cases with the Ubtl IDE inside Eclipse. After the Ubtl compiler generates an UML model, a test engineer can manipulate this model with a compatible UML tool when necessary. Further, a test engineer could trigger a code generator by using the UML model. This can happen inside Eclipse or by leveraging an external tool which is compatible to the UML2 project. In Subsection 2.5.4, we present three projects of the Eclipse ecosystem, which can be used to build a code generator. The generated test cases can then be used by the target test environment. These final test cases can be written in any programming/testing language or format like XML. In the last step the test engineer obtains the test results of the final test platform. The benefit of this approach is, that the test engineer does not have to know how the test cases have to look like on the test platform. It is easy to support another test environment, because just a different generator has to be developed. Another benefit is that the test cases do not have to be written for every platform over and over again. Even when there is only one target platform, Ubtl might be useful. For instance, when the platform expects an XML file as a test input, it may be easier to specify it with Ubtl than with XML.

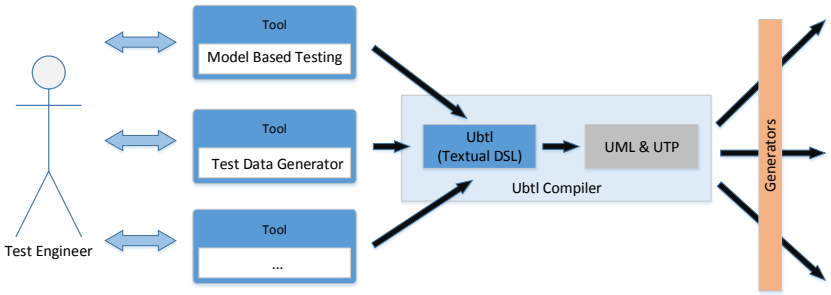


Figure 3.2: Ubtl application number 2 illustrates how Ubtl could be leveraged by other tools

Application two: Ubtl could be used by other tools (see Figure 3.2). For instance, a

MBT tool could specify resulting test cases or test data in Ubtl. The advantage of this is that a tool does not have to be aware of any dedicated platform except Ubtl. It would be easy to add other test platforms, without changing the front tools, because the corresponding generators work with the UML model. The Ubtl compiler can be leveraged as Java library in such an automatic process.

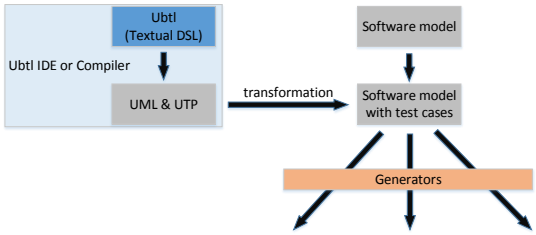


Figure 3.3: Ubtl application number 3 shows how Ubtl could be used in conjunction with existing models

Application three: Ubtl can be used in conjunction with models of software (see Figure 3.3). The test cases would be specified with Ubtl, while the resulting UML models could be transformed to test cases part of the software model or specified in the same modeling language like the model. The advantage can be that it is easier to specify test cases with Ubtl. Also the generators for the model of the software could be reused for the test cases. This variant could be especially useful for component–based system engineering, where the interfaces of components are often modeled, for instance with the EAST–ADL UML2 profile. It would be easy to merge test cases and components, and to synthesize concrete code and test cases. Additionally components could be configured for testing purposes.

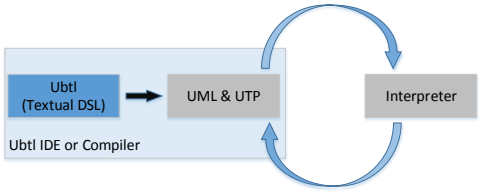


Figure 3.4: Ubtl application number 4 illustrates how an interpreter could use the resulting UML test cases

Application four: The resulting UML models do not have to be used by code generators (see Figure 3.4). An interpreter could use an UML model as input to stimulate test components or SUTs.

3.2 Software Architecture

We chose to develop Ubtl based on Java and Eclipse projects because of two reasons: The UML2 project and the Xtext project.

The UML2 project offers an accurate open–source implementation of the UML specifications. Further, UML2 models are supported by many tools, including commercial ones (see Subsection 2.5.2). This makes it a viable target for Ubtl. The Xtext framework makes it easy, compared to other approaches, to create a DSL compiler and a full–blown Eclipse IDE. It is actively maintained and still new features are introduced which can make a DSL or the corresponding IDE more powerful. For a detailed explanation of the Xtext framework see Subsection 2.5.3.

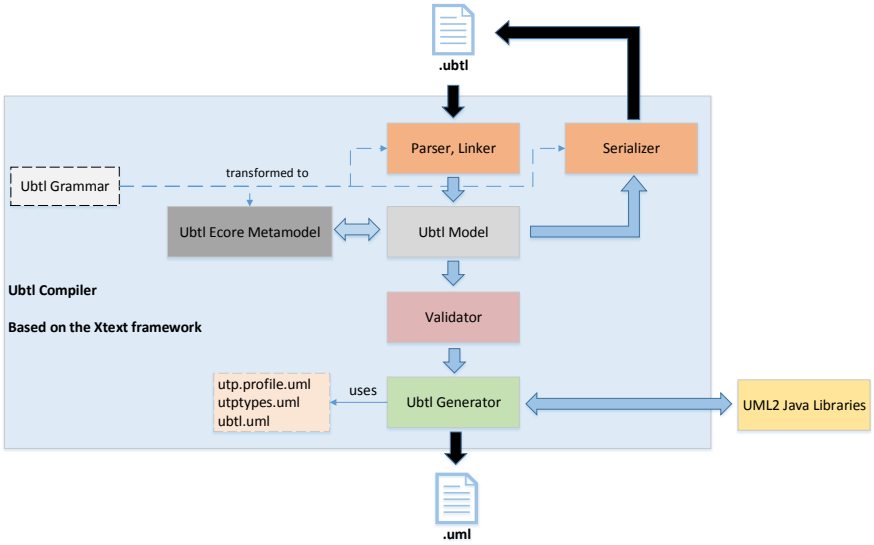


Figure 3.5: Ubtl compiler architecture

Figure 3.5 shows a simplified view of the relevant components of the Ubtl compiler architecture. The Ubtl grammar, specified with the Xtext grammar, is automatically transformed to the Ubtl metamodel, parser, linker, and serializer. The parser and linker are responsible for reading an Ubtl file and generating an Ubtl model. Internally only the Ubtl model is used. The serializer is responsible for transforming an Ubtl model back to the textual representation. The validator contains our restrictions of Ubtl. The Ubtl generator is used to transform an Ubtl model to an UML model. We specified this transformation, like all customizations, with the Xtend programming language. It leverages the UML2 project for this task. Additionally it uses the predefined UML models utp.profile.uml, utptypes.uml, and ubtl.uml. All these compiler components seamlessly integrate into the EMF environment and can be used separately. Concerning front software, an Ubtl model can be constructed directly without a textual representation. The Ubtl IDE, which is automatically generated by the Xtext framework and customized by us (see Section 3.5), also leverages the compiler components. A difference is that the IDE uses a different parser for the content assist, which is in that case faster.

3.3 Textual Language Elements of Ubtl

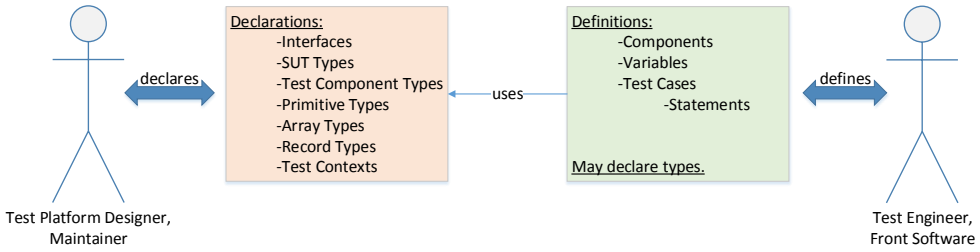


Figure 3.6: Logical separation of the Ubtl language elements

Figure 3.6 gives an overview of the language elements and how we logically separate them. On the left hand side we see the different available declarations which can be specified by test platform designers or maintainers. These declarations are known beforehand by a code generator in order to transform the definitions which we can see on the right hand side. A test engineer or front software may specify those definitions and test cases. Such a definition uses the corresponding declaration as type. Definitions are the elements which interact at runtime. If code generators allow it, a test engineer may also declare types. Declarations, definitions, and test cases have to be specified inside packages and can exist side–by–side. Concerning the test logic embedded in test cases, following statements can be used: full variable definition, assignment, method call, set verdict, assertion, loop, foreach loop, if statement, and log statement. These statements interact with the component and variable definitions.

In the following, we explain the language elements of Ubtl in detail. We also explain details about the Xtext grammar when concepts occur for the first time in the grammar fragments. [1]

General Restriction

A general restriction of the following language elements is that they can only reference elements specified in the code above them. One reason is that the Ubtl–to–UML generator generates the UML models top down. A behavior which could be of course changed. The more important one is that we cannot assume that every possible target platform supports a situation where an element a references an element b, while element b references element a. An exception of this restriction are packages at the top level.

Ubtl Model

Grammar

```
UbtlModel:
  packages+=Package*
;
```

[1] See Listing A.1 in the Appendix for the complete Ubtl grammar. We provide the predefined Xtext terminals which we import for the Ubtl grammar in Listing A.2. Further we provide a syntax graph of the Ubtl grammar in Figure A.1.

In general, parser rules are mapped to EClasses. In the grammar above, an EClass *UbtlModel* with the feature *packages* is created. The **+=** operator is used to indicate that *packages* is a list.

Description

As the name implies *UbtlModel* is the root element. An Ubtl file represents an instance of *UbtlModel* with zero to infinite packages.

Semicolon

Grammar

```
Semicolon:
  (";")?
;
```

Semicolon is just a string. Therefore it does not exist in the Ecore model.

Description

Ubtl allows, like many programming languages, to end an element or statement with a semicolon. However it is just optional and can increase readability.

Package

Grammar

```
Package:
  "package" name=ID "{"
    imports+=PackageImport*
    content+=PackageElement*
  "}" Semicolon
;

PackageImport:
  "import" package=[Package|QualifiedPackageName] Semicolon
;

QualifiedPackageName:
  ID("."ID)*
;

PackageElement:
  Package | Declaration | ComponentDefinition | FullVariableDefinition | TestCase
;
```

Xtext allows specifying cross references inside the grammar. For instance, *PackageImport* references packages. The text between the square brackets refers to an EClass as type and not to the parser rule. This information is actually used by the linker. *QualifiedPackageName* is a mask for accessing packages. By default the *name* feature is used by cross references to find an instance of an EClass.
PackageElement is an EClass and the super type of the comprised elements.

Description

Every Ubtl element, except *Package* and of course *UbtlModel*, has to be contained inside a package. It acts like a namespace and groups language elements. A package automatically inherits the elements of a parent package, which are specified before that package inside the parent package. It is possible to import other packages to access contained

elements, which are then automatically added to the scope of the parent. Also, packages of other Ubtl files can be imported.
PackageElement represents elements which can be directly specified inside a package.
A useful feature of the Xtext framework is the support of Eclipse plugins. Ubtl files can be encapsulated in plugins. An Ubtl project can import such an Eclipse plugin and access the contained packages seamlessly.

Restrictions

A package has to provide a unique name, depending on the package level it is located. For instance, two packages named *demo* are not allowed to exist side by side, but a package *demo* can contain a package *demo* (another level). This restriction is still true when packages with the same name are specified in different files and are held by the same Ubtl project or by imported plugins.
Import cycles are not allowed. For instance package a cannot import package b, while package b imports package a.
It is not allowed to specify *PackageElements* with the same name inside a package. Also they cannot have the same name like an imported or inherited element of another package.
It is not possible to import packages which are specified after an importing package, except it is a top level package. For instance in the example package $b.c.d$ cannot import package $b.c.e$, but could import package g. This is due to the general restriction which is explained in the first subsection.

Example

```
package a {
  package a {}
}

package b {
  package c {
    package d {
      import a.a
    }
    package e {}
    package f {
      import g
    }
  }
}

package g {
  import b.c.d
  import a
  import b.c.e
}
```

Declaration

Grammar

```
Declaration:
  "declare" (ObjectDeclaration|TestContextDeclaration) Semicolon
;

ObjectDeclaration:
  InterfaceDeclaration|ComponentDeclaration|VariableDeclaration
```

```
;

ComponentDeclaration:
  SutDeclaration|TestComponentDeclaration
;

VariableDeclaration:
  PrimitiveDeclaration|RecordDeclaration|ArrayDeclaration
;
```

Description

The grammar above illustrates how we classify different types of declarations logically. Every declaration is initiated by the keyword *declare*. Declarations are used to specify types.

All declarations have in common that they expect a mandatory attribute named *umlName*. This is due to the fact that the name of a declaration has to be an ID (imported from the Xtext terminals) which does not support whitespaces or special characters, but the names of UML elements (and in particulars classes) can be arbitrary. The *umlName* has to be unique inside the corresponding package hierarchy, but can be the same if a package is imported.

A declaration can reference itself.

Restrictions

An *umlName* cannot be empty.

Signature Declaration

Grammar

```
SignatureDeclaration:
  "signature" name=ID "(" (parameters += ParameterDeclaration ("," parameters +=
      ParameterDeclaration)*)? ")" (returnParameter=ReturnParameterDeclaration)?
;

ParameterDeclaration:
  (parameterDirection=ParameterDirection)? name=ID ":" type=[ObjectDeclaration]
;

enum ParameterDirection:
  IN="in" | INOUT="inout" | OUT="out"
;

ReturnParameterDeclaration returns ParameterDeclaration:
  {ReturnParameterDeclaration}":" type=[ObjectDeclaration]
;
```

ParameterDirection is transformed to an EEnum in the resulting Ecore model.

The notation {*ReturnParameterDeclaration*} indicates that a *ReturnParameterDeclaration* instance must be created when the rule is accessed, while the *returns* keyword specifies that it is assigned as *ParameterDeclaration*. Therefore *ParameterDeclaration* is a super type of *ReturnParameterDeclaration*.

Description

Signatures are method definitions. Ubtl currently does not support the specification of method bodies (see Chapter 6). The syntax of parameters is borrowed from UML.

We do support method overriding (or in that case signature overriding). For instance, a test component could specify a signature *doSomething(x : float32)* and a signature

doSomething(x : int32). In that case the types of the parameters have to be different. The correct signature is chosen depending on the argument types.
It is also possible to specify an optional *ParameterDirection*. See the explanation of the UML Class Diagram in Subsection 2.3.2 for more information about parameter directions. Per default the parameter direction is *in*.
We support the use of component declarations as the type of a parameter. It is also permitted to specify an interface declaration as type. In that case components realizing an interface can be used as arguments.
Also the declaration specifying the signature can be used as parameter type.

Restrictions

Signatures can only be specified inside a sut, test component, and interface declaration. We only allow one return parameter, because that is the case in many programming languages. A return parameter cannot have a name.
It is not allowed to override a signature, when the difference is the type of the return parameter.
The parameter name has to be unique inside a signature.

Attribute Declaration

Grammar

```
AttributeDeclaration:
  (required?="required")? "attribute" name=ID ":" type=[ObjectDeclaration]
;
```

The **?=** operator specifies that the corresponding feature is of the type EBoolean. In the grammar above *required* is set to true, when the keyword is used.

Description

Like the name attribute declaration implies, it is possible to specify attributes. The syntax is similar to the one used in UML. A feature is to mark an attribute as *required*. Then component and variable definitions have to specify a corresponding property. Like the types of parameters, all kind of object declarations are allowed. Again, it is permitted to use the encapsulating declaration as type.

Restrictions

Attributes can only be specified inside a sut, test component, interface, and record declaration.
The name of an attribute has to be unique inside the scope of an object declaration. Component declarations realizing an interface cannot override attributes and a different name has to be chosen.

Interface Declaration

Grammar

```
InterfaceDeclaration:
  "interface" name=ID "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &(features+=(SignatureDeclaration|AttributeDeclaration) Semicolon)*
    )
  "}"
;
```

The **&** operator is used to specify unordered groups. Each element may occur in any order, but it must appear once except it is optional like *features*.

Description

Ubtl supports interfaces. These can be useful for test case generators, to indicate that a component is of an expected type with predefined signatures and attributes. Another use case is to specify commands for generators by using attributes. Component declarations can realize zero to infinite interfaces. The relationship *realizes* is equivalent to the *implements* statement in the Java programming language.

Restrictions

Interfaces cannot realize other interfaces. This may change in future Ubtl versions.
If a component declaration realizes several interfaces, the attributes have to be unique. For instance it is not allowed to realize two interfaces at the same time if they specify attributes with the same name. Signatures are not a problem, as we support signature overriding.

Example

```
package interface_declaration_demo {
  //We import primitives for attributes and signatures
  import primitive_declaration_demo

  declare interface Interface_A {
    umlName = "Interface A"

    attribute a: int32
    attribute b: string
    signature doSomething(inout a: Interface_A): int32
    signature doSomething()
  }

  declare interface Generator_Directive {
    umlName = "Code Generator Directive"

    required attribute generateCodeStub: bool
  }

  declare interface Simple_Common_SUT_Type {
    umlName = "Simple Common SUT Type"

    signature doIt()
  }
}
```

SUT Declaration

Grammar

```
SutDeclaration:
  "sut" name=ID ("realizes" interfaceRealizations+=[InterfaceDeclaration]
      (","interfaceRealizations+=[InterfaceDeclaration])*)?  "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &(features+=(SignatureDeclaration|AttributeDeclaration) Semicolon)*
    )
  "}"
;
```

Description

SUT declarations describe the attributes and signatures of SUTs. They represent the

target components of a testcase. Note that a SUT not necessarily has to be a component on the target test platform. A test case generator, which knows the SUT type, could use any representation.

Example

```
package sut_declaration_demo {
  /*We import primitives and interfaces.
    Primitives are imported by the interface_declaration_demo package.*/
  import interface_declaration_demo

  declare sut SutA realizes
  Generator_Directive, Interface_A, Simple_Common_SUT_Type {
    umlName = "Sut A"

    attribute c: float32
    signature run(): bool
  }

  declare sut SutB {
    umlName = "Sut B"

    signature connectTo(i: SutA): bool
    signature run(): bool
    signature manipulate(in i: SutA): SutA
  }

  declare sut SutC realizes Simple_Common_SUT_Type {
    umlName = "Sut C"

    signature doSomething(in arg: int32)
  }
}
```

Test Component Declaration

Grammar

```
TestComponentDeclaration:
  "testcomponent" name=ID ("realizes" interfaceRealizations+=[InterfaceDeclaration]
      (","interfaceRealizations+=[InterfaceDeclaration])*)?  "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &(features+=(SignatureDeclaration|AttributeDeclaration) Semicolon)*
    )
  "}"
;
```

Description

A test component declaration is similar to a SUT declaration, except they differ in the resulting UML model. Test components can be used for helper methods. For instance, a test component could add math methods. See Chapter 6 for our vision how test components could allow the implementation of method bodies in future versions of Ubtl.

Example

```
package testcomponent_declaration_demo {
  import sut_declaration_demo

  declare testcomponent SUT_Resetter realizes Generator_Directive {
    umlName = "SUT Resetter"

    signature reset(i: SutB): bool
```

```
    signature reset(i: Simple_Common_SUT_Type): bool

    //Not possible, because SutA realizes Simple_Common_SUT_Type
    //signature reset(i: SutA): bool
  }

  declare testcomponent TestResultsManager realizes Generator_Directive {
    umlName = "Test Results Manager"

    signature sendTestResults()
  }
}
```

Primitive Declaration

Grammar

```
PrimitiveDeclaration:
  "primitive" name=ID "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &("referenceableOnlyOnce""="referenceableOnlyOnce=BooleanValue Semicolon)?
     &("requireName""="requireName=BooleanValue Semicolon)?
     &("acceptDataType""=" acceptDataType=(
      "IntDataType"|"IntHexDataType"|"UIntDataType"|"UIntHexDataType"|
      "HexDataType"|"StringDataType"|"FloatDataType"|"BooleanDataType") Semicolon)
    )
  "}"
;
```

Description

Primitive declarations are used to specify the primitive types int, unsigned int, hex, string, float, and boolean.

The attribute *referenceableOnlyOnce* means that a variable of this type can only be used by one other variable once. This is useful for test platforms where properties of components or variables are directly set and cannot reference other variables. Otherwise if two variables reference a variable a and variable a is used in a test case it is not identifiable to which container variable it belongs. When variable a can only be referenced by a single variable, a test case generator can determine if it belongs to a container variable.

If *requireName* is set to true, then the name for a corresponding primitive variable always has to be specified, even when it is defined inline for instance inside a record or array.

The default value of the two boolean attributes is false.

Restrictions

The length of the data of a primitive variable can be infinite. This is not adjustable in the current version of Ubtl (see Chapter 6).

Example

```
package primitive_declaration_demo {
  declare primitive int32 {
    umlName = "Int32"
    acceptDataType = IntHexDataType
    referenceableOnlyOnce = true
  }

  declare primitive uint8 {
    umlName = "UInt8"
    acceptDataType = UIntHexDataType
    requireName = true
```

```
  }

  declare primitive float32 {
    requireName = true
    acceptDataType = FloatDataType
    referenceableOnlyOnce = true
    umlName = "Float32"
  }

  declare primitive string {
    umlName = "String"
    acceptDataType = StringDataType
  }

  declare primitive bool {
    umlName = "Bool"
    acceptDataType = BooleanDataType
  }
}
```

Array Declaration

Grammar

```
ArrayDeclaration:
  "array" name=ID "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &("referenceableOnlyOnce""="referenceableOnlyOnce=BooleanValue Semicolon)?
     &(types=AcceptTypes Semicolon)
     &("oneReferenceMultipleTimes""="oneReferenceMultipleTimes=BooleanValue Semicolon)?
     &("requireNameOfPrimitiveVariables""="requireNameOfPrimitiveVariables=BooleanValue
          Semicolon)?
    )
  "}"
;

AcceptTypes:
  "acceptTypes""=" (acceptTypes=ObjectTypeSet|acceptArray?="array"|
  acceptRecord?="record"|acceptPrimitive?="primitive"|acceptVariable?="variable")
;

ObjectTypeSet:
  objectTypeSet+=[ObjectDeclaration] ("," objectTypeSet+=[ObjectDeclaration])*
;
```

Description

Arrays behave like arrays known from different programming languages. An array offers the method *get(...)* to access contained elements.

The attribute *referenceableOnlyOnce* behaves like the same attribute specified by primitive declarations.

The attribute *oneReferenceMultipleTimes* manages if a variable or component can be referenced inside an array multiple times.

If *requireNameOfPrimitiveVariables* is set to true, an array expects names for inline specified primitives.

An array can be set to accept several objects of specific types. It is also possible to specify that an array accepts whole declaration categories, like *array*, *record*, *primitive* or *variable*. If *variable* is chosen, then an array allows to reference all kind of variable definitions.

Restrictions

A restriction is that an array can only accept one component or interface declaration. However, as it is possible to use an interface declaration, different kinds of derived components can be used. We do not support several component or interface declarations in an array, because in a foreach loop the control variable would have an undefined type. Ubtl does not support undefined types.

Variable declarations cannot be mixed with component or interface declarations.

Example

```
package array_declaration_demo {
  import testcomponent_declaration_demo

  declare array ArrayA {
    umlName= "Array A"
    acceptTypes = Simple_Common_SUT_Type
    oneReferenceMultipleTimes = true
  }

  declare array ArrayB {
    umlName= "Array B"
    acceptTypes = primitive
    requireNameOfPrimitiveVariables = true
    referenceableOnlyOnce = false // Optional, by default it is already false
  }
}
```

Record Declaration

Grammar

```
RecordDeclaration:
  "record" name=ID "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &("referenceableOnlyOnce""="referenceableOnlyOnce=BooleanValue Semicolon)?
     &(attributes+=AttributeDeclaration Semicolon)*
    )
  "}"
;
```

Description

Record declarations are used to describe a data structure which holds several member values.

Like the other variable types, it offers the feature *referenceableOnlyOnce*.

Restrictions

If a record declaration does not define any attributes, a warning is shown to the user. It is still a valid declaration.

Example

```
package record_declaration_demo {
  import array_declaration_demo

  declare record RecordA {
    umlName = "Record A"
    attribute arbitrarySut: Simple_Common_SUT_Type
    required attribute a: int32
    attribute b: ArrayA
    attribute c: SutB
  }
```

```
  declare record RecordB {
    umlName = "Record B"
    attribute a: RecordA
    referenceableOnlyOnce = true
  }
}
```

Test Context Declaration

Grammar

```
TestContextDeclaration:
  "testcontext" name=ID "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &("disableVariableDefinition""="disableVariableDefinition=BooleanValue Semicolon)?
     &("disableLoop""="disableLoop=BooleanValue Semicolon)?
     &("disableSetVerdict""="disableSetVerdict=BooleanValue Semicolon)?
     &("disableAssignment""="disableAssignment=BooleanValue Semicolon)?
     &("disableAssertion""="disableAssertion=BooleanValue Semicolon)?
     &("disableForEachLoop""="disableForEachLoop=BooleanValue Semicolon)?
     &("disableTimer""="disableTimer=BooleanValue Semicolon)?
     &("disableIf""="disableIf=BooleanValue Semicolon)?
     &("disableIfComplexCondition""="disableIfComplexCondition=BooleanValue Semicolon)?
     &("disableIfOr""="disableIfOr=BooleanValue Semicolon)?
     &("disableIfAnd""="disableIfAnd=BooleanValue Semicolon)?
     &("disableIfEquality""="disableIfEquality=BooleanValue Semicolon)?
     &("disableIfComparison""="disableIfComparison=BooleanValue Semicolon)?
     &("disableIfNot""="disableIfNot=BooleanValue Semicolon)?
     &("disableLog""="disableLog=BooleanValue Semicolon)?
    )
  "}"
;
```

Description

Test contexts are mandatory for test cases. They allow to disable specific Ubtl language statements inside a test case. This can be useful when a target test platform does not support an Ubtl feature. By default, everything is allowed inside a test case.
The name of a test context is separate to the names of other declarations or definitions. For instance, a sut declaration named *demo* can exist side by side with a test context named *demo*.

Example

```
package test_context_declaration_demo {
  declare testcontext context {
    umlName = "Context"
  }

  declare testcontext context0 {
    umlName = "Context 0"

    disableIf = true
    disableLoop = true
    disableAssignment = true
  }
}
```

Definition

Grammar

```
ObjectDefinition:
  ComponentDefinition|FullVariableDefinition|DataDefinitionElement|ForEachVariableDefinition
;
```

Description

Definitions are instances of declarations. These are the objects which interact and are used at runtime.

Property Definition

Grammar

```
PropertyDefinition:
  name=ID "=" dataDefinition=DataDefinition
;

PrimitivePropertyDefinition returns PropertyDefinition:
  {PrimitivePropertyDefinition}name="value" "=" dataDefinition=DataDefinition
;

ArrayPropertyDefinition returns PropertyDefinition:
  {ArrayPropertyDefinition}name="content" "=" dataDefinition=DataDefinition
;

DataDefinition:
  (data=Data)|(elements+=DataDefinitionElement ("," elements+=DataDefinitionElement)*)
;

DataDefinitionElement:
  variableOrValueType=[ObjectOrPrimitiveDeclaration] ((name=ID)? data=Data)?
;

Data:
  (sign?="-")? (int=IntString| hex=HexString| string=STRING| float=FloatString|
       boolean=BooleanString)
;
```

Description

A property realizes an attribute. If the type of an attribute is a primitive or array declaration, then the values can be defined inline without using an extra primitive or array. We also provide special properties for variables which realize primitives or arrays (*PrimitivePropertyDefinition*, *ArrayPropertyDefinition*).

Restrictions

Only one property can realize a corresponding attribute.

Component Definition

Grammar

```
ComponentDefinition:
  "comp" ((type=[ComponentDeclaration])|(timer?="timer")) name=ID ("{"
    (properties+=PropertyDefinition Semicolon)*
  "}")? Semicolon
;
```

Description

Component definitions specify instances of component declarations or the built in type *timer* . The keyword to start a component definition is *comp*.

A timer offers two methods, *start(expire: duration)* and *stop()*, which correspond to the timer interface specified by UTP. It does not offer any attributes to specify.

Restrictions

Components cannot be defined inside a test case.

Example

```
package component_definition_demo {
  import sut_declaration_demo
  import testcomponent_declaration_demo

  comp SutA sut0 {
    generateCodeStub = false // required due interface Generator_Directive
    a = 0
    b = "test"
    c = 2.0
  }

  comp SutA sut1 {
    generateCodeStub = true
    a = int32 22
    b = string optionalVariableName "demo"
  }

  comp SutB sut2 // no properties

  comp SutC sut3

  comp SUT_Resetter resetter {
    generateCodeStub = true
  }

  comp TestResultsManager resultsManager {
    generateCodeStub = false
  }

  comp timer t0
  comp timer t1
}
```

Full Variable Definition

Grammar

```
FullVariableDefinition:
  "var" ((type=[VariableDeclaration])|(duration?="duration")) name=ID
  ( (bigDefinition?="{"
      (properties+=(PropertyDefinition| PrimitivePropertyDefinition| ArrayPropertyDefinition)
           Semicolon)*
     "}")
    |(smallDefinition?="=" dataDefinition=DataDefinition)
  ) Semicolon
;
```

Description

Full variable definitions can specify all different variable declaration types.

We provide two different concepts how the data of a variable can be specified. One concept is to define the data by using a *big definition*. A *big definition* specifies prop-

erties of variables. This concept has to be used when the type of a variable is a record declaration. We also provide two properties for primitive and array declarations. The other concept, called *small definition*, can only be used in conjunction with primitive or array declarations. The small definition eases the task of defining such declarations.
We also provide a built in type called *duration*. Duration variables have to be used for the *start()* method of timers.
Variables can be defined inside a test case. In that case the variable is not accessible outside the test case.

Example

```
package full_variable_definition_demo {
  import primitive_declaration_demo
  import array_declaration_demo
  import record_declaration_demo
  import component_definition_demo

  var int32 int0 = -20 // small definition

  var float32 float0 { //big definition
    value = 4.2
  }

  var bool bool0 = true

  var string string0 = "Demo String!"

  var ArrayA array0 = sut0, sut1, sut3, sut0 // different component declarations, but same
       interface

  var ArrayB array1 = int0, int32 a 2, int32 b -2, int32 c 10000

  var ArrayB array2 {
    content = float0, bool0, float32 a 2.2, string b "fff", int32 c -20000
    /* If the following would be added to the array it would cause an error
     *  when a variable referencing int0 would be used inside a test case.
     *  The reason is that int32 can only be referenced once and
     *  array1 already references int0.
     *  It would work if array1 does not use int0 or if array1 would be
     *  specified inside a package which is not imported by this one.
     */
    // , int0
  }

  var RecordA record0 {
    a = 0xff // required
    b = array0
    c = sut2
  }

  var RecordA record1 {
    a = int32 a 0
    b = sut0, sut1, sut3 //inline array
  }

  var RecordB record2 {
    a = record0
  }

  var duration d0 = 20
  var duration d1 = 40
}
```

Testcase

Grammar

```
TestCase:
  "testcase" testContext=[TestContextDeclaration] name=ID
    block=Block
;
```

Description

Test cases need a test context as argument. See Section 3.4 for an explanation why this is necessary.

The test case logic is specified inside the block.

Example

```
package testcase_demo {
  import test_context_declaration_demo

  testcase context case0 {
  }

  testcase context case1 {
  }
}
```

Abstract Block

Grammar

```
AbstractBlock:
  Block|OneStatementBlock
;

Block:
  {Block}"{" (statements += Statement )* "}" Semicolon
;

OneStatementBlock:
  statements += Statement
;
```

Description

Blocks serve as body of test cases, if statements, loops, and foreach loops.

Statement

Grammar

```
Statement:
  FullVariableDefinition|
  Loop|
  SetVerdict|
  AssignmentOrCall|
  Assertion|
  ForEachLoop|
  IfStatement|
  LogStatement
;
```

Description

Statements are used inside a block. They specify the test case logic.

Object Reference

```
ObjectReference:
  object=[ReferenceableObject] (features+=ReferenceFeature)* (methodCall?="("
       (arguments+=Argument("," arguments+=Argument)*)? ")")?
;

ReferenceFeature:
  "." ((feature=ID)| get?="get""("(getElement=[DataDefinitionElement]|getPosition=INT)")")
;

Argument:
  {Argument} ((reference=ObjectReference)|(data=Data))
;
```

Description

Object references enable to access components and variables.

A difference to most programming languages is that object reference operates on the data actually defined. For instance, in the Java programming language it is possible to access the attribute of an object even when it is null and the data is not defined. In Ubtl one has to define a property of the corresponding attribute, specify the data and then it can be accessed. Therefore, in Ubtl it is not possible to access values inside the structure of the return parameter of a signature, because it is not defined. This limitation comes from the nature of the UML model.

We also defined a special feature called *get* to access elements inside an array. An element may be accessed by the position or by the name, if it is a primitive with a name specified inline.

Arguments are used by method calls, assignments, assertions and if statements. We also allow the definition of primitive variables inline as arguments.

Restrictions

Only objects defined above an object reference can be referred to.

The *methodCall* part of an object reference can only be used in the following rule.

Assignment and Method Call

Grammar

```
AssignmentOrCall:
  reference=ObjectReference (assignment?="=" assignmentArgument=Argument)? Semicolon
;
```

Description

Assignments and method calls have the same parse rule, because both start with an object reference. Otherwise the parser could not decide with which rule it is dealing with.

An object reference, which is not a method call and not used in an assignment is ignored by the Ubtl–to–UML generator.

The return parameter of a method call can be assigned to a variable, component or property. In that case the related object references still operate on the properties of the existing variable or component. We assume that target platforms implement a real assignment.

Restrictions

An argument cannot be assigned to the definition of a variable. In that case a variable has to be defined first and then values can be assigned to it.

Different declarations cannot be used in assignments. Both parts have to be of the same declaration. An exception to this rule is of course when a method call returns an interface and the target component implements that interface.

In an assignment, the left reference has to be a primitive variable, with the exception that the return of a method is stored in it. We implemented this restriction to prevent mistakes like arrays of different size or undefined properties, which should not be accessed in the following Ubtl code.

Duration variables cannot be used in assignments and only in method calls where the target is the *start()* method of a timer.

Timer definitions cannot be assigned.

Example

```
package assignment_and_method_call_demo {
  import test_context_declaration_demo
  import full_variable_definition_demo
  import component_definition_demo

  testcase context case0 {
    t0.start(25) // inline definition of a duration
    var float32 localVariable = 2.2
    localVariable = record2.a.b.get(0).c // equals sut0.c
    var bool ret = false
    ret = resetter.reset(sut1)
    /*  We always assume the target test platform handles the case
    *   when the statements between start and stop take longer. */
    t0.stop()

  }

  testcase context case1 {
    sut0 = sut2.manipulate(sut0)
    var int32 localVariable = 42
    sut3.doSomething(localVariable)
    sut3.doSomething(0xffa)
  }
}
```

Set Verdict

Grammar

```
SetVerdict:
  "setVerdict" "(" verdict=Verdict ")" Semicolon
;

enum Verdict:
  NONE="none"|PASS="pass"|INCONCLUSIVE="inconclusive"|FAIL="fail"|ERROR="error"
;
```

Description

As the name implies, a set verdict statement is used to manipulate the verdict of a test case. We support the same verdicts like UTP.

Example

```
package set_verdict_demo {
  import test_context_declaration_demo

  testcase context case0 {
    setVerdict(none)
    setVerdict(pass)
    setVerdict(inconclusive)
    setVerdict(fail)
    setVerdict(error)
  }
}
```

Assertion

Grammar

```
Assertion:
  "assert""("leftArgument=Argument assertionType=AssertionType rightArgument=Argument")"
       Semicolon
;

enum AssertionType:
  EQUAL="==" | GREATERTHAN=">" | LOWERTHAN="<" | GREATERTHANOREQUALTO=">="|
  LOWERTHANOREQUALTO="<=" | NOTEQUAL="!="
;
```

Description

Ubtl comes with a built–in assertion statement. Assertions only work for primitive variables.

Restrictions

Only primitives which use the same declaration can be compared.

At least one primitive variable has to be used, in order to determine the type. The other argument can be data, if the primitive does not require a name.

Concerning records, arrays, and components, all values have to be compared individually.

Example

```
package assertion_demo{
  import test_context_declaration_demo
  import full_variable_definition_demo
  import component_definition_demo

  testcase context case0 {
    assert(float0 == array0.get(0).c)
    assert(1 < int0)
    assert(record1.b.get(0).b != "Test")
    var float32 float1 = 0.0
    assert(sut0.c > float1) // Float32 requires a name
  }
}
```

Loop

Grammar

```
Loop:
  "loop" "(" iterations=INT ")" block=AbstractBlock
;
```

Description

Loop is used to repeat a sequence of statements for a defined limit of iterations.

Example

```
package loop_demo {
  import test_context_declaration_demo
  import component_definition_demo

  testcase context case0 {
    loop(200)
      sut0.run()

    loop(50) {
      sut1.doSomething(sut0)
      assert(sut1.a < 100)
    }
  }
}
```

Foreach Loop

Grammar

```
ForEachLoop:
  "foreach""("definitions+=ForEachVariableDefinition(","
      definitions+=ForEachVariableDefinition)*")"
    block=AbstractBlock
;

ForEachVariableDefinition:
  name=ID ":" reference=ObjectReference
;
```

Description

Foreach loops are used to iterate through one or several arrays. For instance, this can be useful to call a method with different input data. Another use case is to check whether results stored in an array are inside an expected range.

Restrictions

Arrays have to be of the same size.

The elements inside an array have to be of the same declaration type. In the case when the type of an array is an interface, the components must implement the same interface.

Properties and methods of control variables are not accessible. Concerning properties this is because the control variable does not hold any data in UML. Methods are not callable because the control variable only exists as instance specification in UML and is not a property of the enclosing test context.

For the same reason, it is not allowed to use a foreach control variable as a reference in another foreach loop.

The name of a control variable must be unique inside a test case.

Example

```
package foreach_demo {
  import test_context_declaration_demo
  import full_variable_definition_demo
  import component_definition_demo

  testcase context case0 {
    foreach(x: array0, y: array1) { // Arrays have same size
       resetter.reset(x)
       assert(y > 0)
    }
  }
}
```

If Statement

Grammar

```
IfStatement:
  "if" "("condition=Condition")" thenBlock=AbstractBlock
  (=> "else" elseBlock=AbstractBlock)?
;

Condition:
  Or
;

Or returns Condition:
  And ({Or.left=current} "||" right=And)*
;

And returns Condition:
  Equality ({And.left=current} "&&" right=Equality)*
;

Equality returns Condition:
  Comparison (
    {Equality.left=current} op=("=="|"!=")
    right=Comparison
  )*
;

Comparison returns Condition:
  Primary (
    {Comparison.left=current} op=(">="|"<="|">"|"<")
    right=Primary
  )*
;

Primary returns Condition:
  "(" Condition ")" |
  {Not} "!" "("condition=Condition")" |
  Atomic
;

Atomic returns Condition:
  {Atomic} argument=Argument
;
```

The **=>** operator indicates, in the grammar above, that the first *else* the parser encounters belongs to the nearest *if*. Otherwise a parser could not decide to which *if* an *else* belongs when several *ifs* are nested. This is also known as dangling else problem.

The notation {*Or.left=current*} above is a tree rewrite operation. In that case a new instance of the EClass *Or* is created and the current element to–be–returned is assigned to the feature *left*.

Description

The grammar above illustrates how the if condition is split up into a parse tree. *Or* has the least precedence while *Primary* the highest. Atomics just hold arguments. We were inspired to implement the if condition in that way by the work of Bettini [Bet13].

Like in an assertion, it is possible to directly specify data, when the other part is a primitive variable.

Restrictions

We only support the use of primitive variables for if conditions.

Though the grammar would allow it, we do not allow that a primitive variable can be used without a comparison or equality rule. This was a design decision. Many target platforms would probably allow it, but to be on the safe side, we restricted that for instance a single boolean represents a condition.

Or and *And* can only be used to associate sub conditions and not atomics

Example

```
package if_demo {
  import test_context_declaration_demo
  import full_variable_definition_demo
  import component_definition_demo

  testcase context case0 {
    if(int0 <= 2 && !(record0.b.get(0).b == "test" || sut0.c != float0))
      setVerdict(pass)
    else
      setVerdict(fail)

    //if(bool0)  // not allowed
    //{}
    if(bool0 == true) // allowed
    {
      if(sut0.b != string0) {}
    }
  }
}
```

Log Statement

Grammar

```
LogStatement:
  "log" "(" information=STRING ")" Semicolon
;
```

Description

The log statement may be used to send string messages to the test platform.

Example

```
package log_demo {
  import test_context_declaration_demo

  testcase context case0 {
    log("Start!")
    // ...
```

```
    log("Finish!")
  }
}
```

Data Rules

Grammar

```
IntString returns ecore::EString:
  INT
;

HexString returns ecore::EString:
  HEX
;

FloatString returns ecore::EString:
  IntString"."IntString
;

BooleanString returns ecore::EString:
  "false"|"true"
;

BooleanValue returns ecore::EBoolean:
  "false"|"true"
;

terminal HEX:
  ("0x"|"0X") ("A"..."F"|"a"..."f"|"0"..."9")+
;
```

All rules above return an EDataType and are not mapped to EClasses.
Terminal rules, like *HEX*, are used by the lexer to generate tokens.

Description

We use the rules above to specify data. We always store data values as strings.
BooleanValue is only used by declarations to specify the settings.
Note that the rules for *ID*, *INT*, and *STRING*, can be found in Listing A.2

3.4 Mapping to UML

In general, UML models are only generated when an Ubtl file (in fact a *UbtlModel*) contains at least one package with a test case. In that case all other packages which are related to the test cases package, through imports or hierarchy, are also generated. We have to generate all packages for each file in a separated way, because UML2 uses UUIDs to access elements. These UUIDs are always set randomly when UML models are generated. In our case, that means the packages generated for one file are not compatible with the same generated packages used by another file. This is not a problem for code generators, because they usually operate on the names of stereotypes or classes inside a UML model. Generated packages are stored in different directories. The directory name is the same of the corresponding file, without the *.ubtl* file ending.

We do not generate graphical diagrams (see Subsection 2.5.2). The following figures are made by hand for explanation purposes.

Package

All Ubtl packages are mapped to UML packages and the defined imports remain. We store each top level package in a separate UML file as UML model. Figure 3.7 illustrates a top level package with the default imports. An *access* dependency represents a private import. The UTP Types package is in detail explained in Subsection 2.4.1. Concerning the UBTL package, we discuss details of it in the following explanations. The Primitive Types package is predefined by UML and provides primitives like String and Integer. We use that package for the definition of data. Additional to these imported packages, UTP is applied on the top level package in order to use the stereotypes.

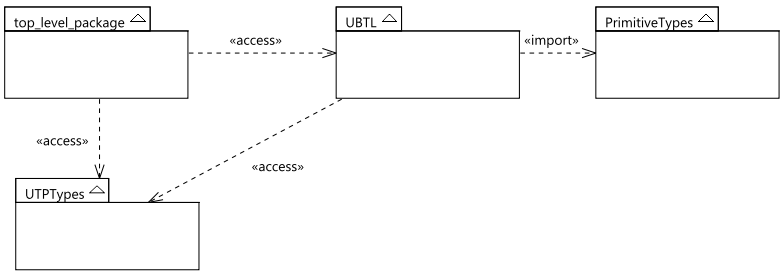


Figure 3.7: Top level package with default imports

Component and Variable Declarations

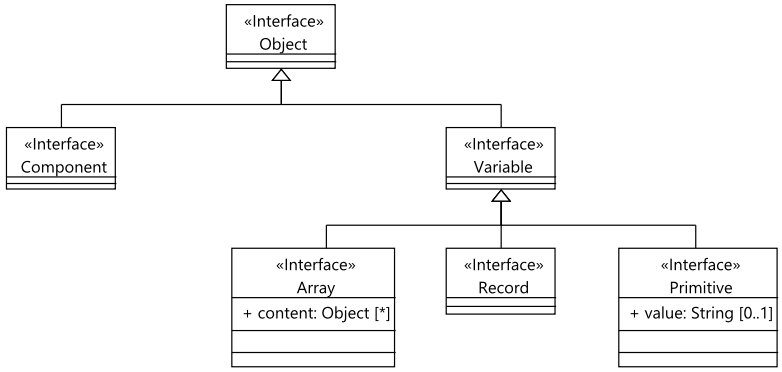


Figure 3.8: Ubtl types interfaces

Component and variable declarations are transformed to UML classes which are contained inside the corresponding UML package. Figure 3.8 illustrates the type interfaces, part of the UBTL package, that are realized by component and variable classes.

Component declarations realize the *Component* interface, except a declaration realizes an interface. In that case the interface generalizes from *Component*. The stereotype *TestComponent* is applied on test component classes. Signatures are transformed to UML operations with the same parameters and return type. The return parameter is named *return*.

Variable declarations realize the corresponding interfaces. The *Array* and *Primitive* interface already provide the attributes which are implemented by their realizations.

When an Ubtl attribute is marked as required, the corresponding UML attribute has the multiplicity one. Otherwise an attribute is optional (multiplicity zero to one).

We annotate the declaration classes with metadata of Ubtl. Metadata is added as comment. It includes the name of a declaration in Ubtl and the different settings. In theory it should be possible to transform declaration classes back to Ubtl declarations.

Definitions

Component and variable definitions are always transformed to instance specifications. We use instance specifications to specify data. Data itself is stored as string. Instances are used as arguments for operations or represent the initial state of a component. They are contained in a child package named *#InstanceSpecifications*. A package in Ubtl cannot have an ID starting with a number sign, therefore no name collisions are possible.

Component Definition

When a signature of a component definition is called in a test case, then the component becomes part of the corresponding test context class as property. The default value of the property refers to the instance specification. If the type of a component definition is a SUT declaration, then the property is marked with the stereotype *SUT*.

Variable Definition

Variable definitions only exist as instance specifications. The elements contained in an array are stored in a separate child package to avoid name collisions with other existing variables.

When the name of a variable definition is unspecified, for instance when it is defined inline, an UUID is generated as name. We mark such instance specifications with a special comment. An exception to this rule are primitives or arrays specified inline as property. In that case the name of the instance specification is a combination of the parent instance name and the property name, separated by a number sign.

Duration variables are also mapped to instance specifications. The difference to other variables is that the instance name is a combination of the variable name and the value, separated with an equal sign. The reason for this is that the UTP type *Duration* is a UML primitive type and does not specify any properties for embedding values.

In the case a variable is defined inside a test case, a call operation action is added to the test case interaction, to indicate to a test platform where a variable is defined first. We suggest that variables which are specified outside of a test case should be initialized at the beginning of a test case. The call operation action is encapsulated in an action execution specification, that is located on the *self* lifeline. The call operation action refers to the *initialize* operation of the *UBTLVariableInitializer* class, illustrated in Figure 3.9. This class is part of the *UBTL* package. The name of the action is *initializeVariable*. Such an instance specification is stored in a child package of the instance specifications package. The name of this package is a combination of the number sign and the test case name.

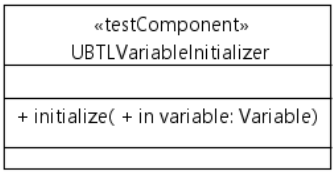


Figure 3.9: UBTLVariableInitializer

Testcase

Test cases are added as UML operations to the corresponding test contexts which are specified as parameter of Ubtl test cases. The test context class is always created on the same package level like the test cases. Therefore, test cases with the same test context, but located in different packages, do not share the same test context class. The name of the test context class is the *umlName* of the Ubtl test context. It is possible to specify several test cases with different test contexts inside a package. That case would result in several test context classes inside an UML package. We save the settings and the Ubtl name of a test context as comment in the package it is declared. Figure 3.10 illustrates an example test context class. Components where a method is called in a test case are added as properties to the test context. Components of the Ubtl type SUT are marked with the UTP stereotype SUT. Test case operations are marked with the corresponding stereotype. We also specify that an operation returns a verdict. This is specified by UTP to be mandatory. We suppose that the target test platform implements the logic how a verdict is assigned. Test context classes are the starting point for code generators.

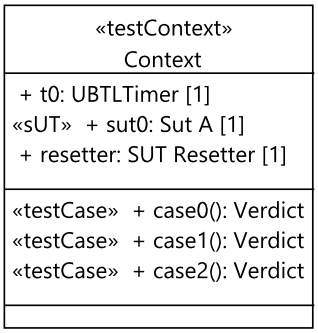


Figure 3.10: Test context class example

Further, a test case is transformed to an interaction which could be drawn as Sequence Diagram. The interaction is part of the test context class. The *method* feature of the test case operation references the interaction. Ubtl statements of a test case are transformed to messages or call operation actions. We use call operation actions to call operations on predefined classes, like *UbtlVariableInitializer* in the subsection above. The arguments of call operations actions are set as value pins and usually reference instance specifications. Messages and operation calls are referenced by occurrence and execution specifications which define the sequence an interaction is processed. By default an interaction owns a lifeline called *self* which represents the test context class. All call operation actions are

covered, encapsulated in an action execution specification, by the *self* lifeline. Messages are referenced by message occurrence specifications and always go from the *self* lifeline to component lifelines.

Object Reference

The information about the object reference itself is lost when Ubtl code is transformed to UML models. The UML generator uses directly the target variable or component. For instance when an element of an array is used as method argument, then the target instance specification, representing the element, is leveraged as argument and not the array. For some test platforms which do not manage data object–oriented it may be necessary to activate the feature *referenceableOnlyOnce* of variable declarations. Then a user of Ubtl is restricted to not reference it more than once and a code generator can decide whether an instance specification is used by another one. EMF offers methods to ease this task.

Assignment

Assignments are transformed to call operation actions which target the *setValue* operation of the class *UBTLValueSetter*. The name of the action is *setValue*. Figure 3.11 illustrates this class, that is part of the UBTL package.

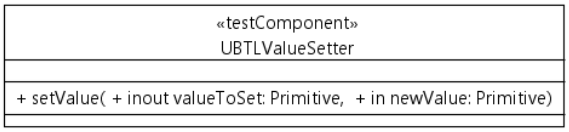


Figure 3.11: UBTLValueSetter

Method Call

The target component of a method call is added to the test context class. Inside the interaction a lifeline is created which represents the component. The method call itself is specified as message, referenced by message occurrence specifications. In the case the return parameter of an operation is assigned to a variable or component, then the target instance specification, representing the variable or component, is used by the return message as argument. Figure 3.12 illustrates a small example.

In the case the method of a timer is called, it is mapped to a call operation action like it is specified by UTP. Figure 3.13 shows the target class which realizes the *Timer* interface and is part of the UBTL package. The name of the call operation actions are *startTimer* and *stopTimer*.

Set Verdict

The set verdict statement is transformed according to UTP. It results in a call operation action, tagged with the stereotype *ValidationAction* and the name *setVerdict*. The target operation *setVerdict* is part of the class *UBTLArbiter* (Figure 3.14). This class is part of the UBTL package. We assume the target test platform processes the verdicts.

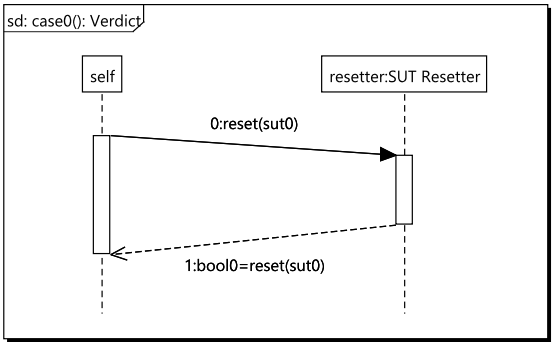


Figure 3.12: Method call example

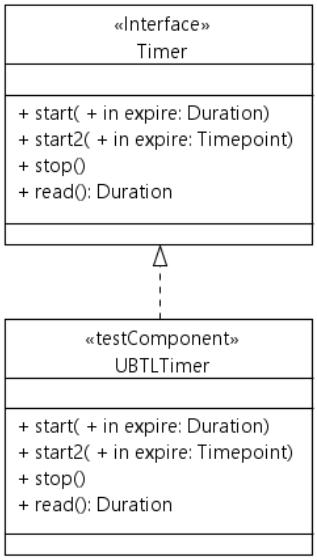


Figure 3.13: UBTLTimer

Assertion

Assertions are transformed to call operation actions which target corresponding operations of the class *UBTLAssert* (Figure 3.15). The name of the action is *assert*.

Loop

Loops are transformed to combined fragments with the name *loop*. The interaction operator is set to *Loop*. Concerning the guard, the minimum iteration is set to the feature *iterations* of the Ubtl loop statement, while the specification is set to false. These are the semantics of a loop according to the UML Superstructure specification [Obj11b].

Foreach Loop

UML does not provide a dedicated concept or semantics to represent foreach loops. We leverage the combined fragment in conjunction with the interaction operator *Loop*. The

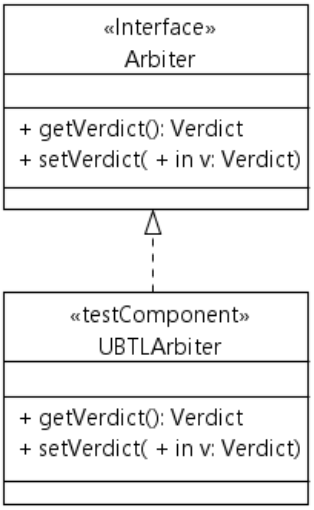


Figure 3.14: UBTLArbiter

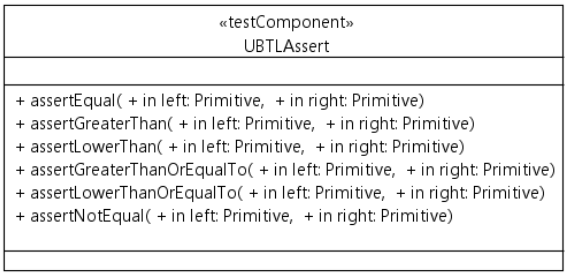


Figure 3.15: UBTLAssert

combined fragment is named *foreach*. The minimum number of iterations is set with the size of the used arrays. The specification of the guard owns references to the instance specifications of the foreach variables and arrays as operands. The instance specifications are ordered in the way that the first operand refers to the foreach variable, while the second one refers to the array. Figure 3.16 illustrates an example foreach loop.

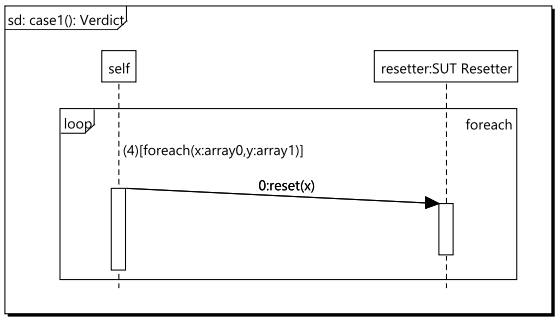


Figure 3.16: Foreach example

If Statement

An if statement is mapped to a combined fragment with the interaction operator *alt*. Alternative fragments are the UML version of ifs. Figure 3.17 illustrates an example, like it is generated by the Ubtl compiler. If conditions are mapped to expressions with the same structure like the Ubtl parse tree. We use the symbol feature of expression to indicate what operator is used or whether it is an atomic. Atomic expressions reference the corresponding instance specifications. The parent expression is referenced by the *specification* feature of the combined fragment guard. The name of the combined fragment is *if*.

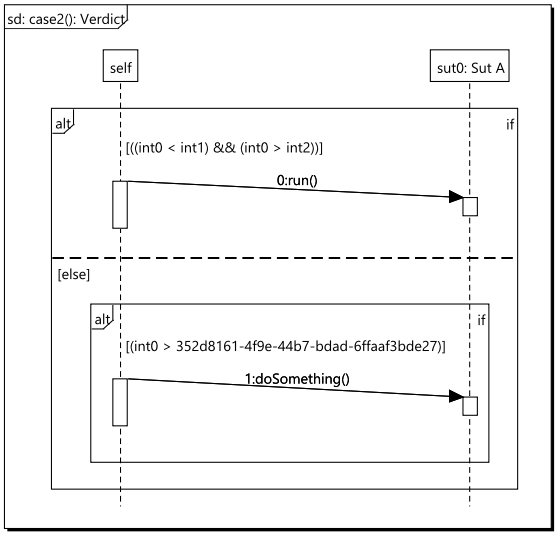


Figure 3.17: If example

Log Statement

Log statements are transformed like it is specified by the UTP specification. It is represented as send object action, tagged with the stereotype *LogAction*. The name of the action is *log*. The request pin of this action holds the *information* feature which is represented just as literal string.

3.5 Customizations of the generated IDE

The Xtext framework automatically generates a useful IDE for Eclipse. In the following list we enumerate changes and additions to the Ubtl IDE:

- We customized the content assist. Though the default one is already very good, we had to do this especially for *Data* proposals. The customized content assist proposes the correct data type depending on the context. For instance if a variable expects a float value, the content assist proposes *0.0* (Figure 3.18a). We adjusted the feature proposal of *ObjectReference*. Only possible calls of features like methods, properties, or the array method *get()* are displayed (Figure 3.18b). Further we customized the proposal of properties. Depending whether a property is already defined in a component or variable, the remaining available properties are shown (Figure 3.18c). For arrays and primitive variables the properties *content* or *value* are proposed. We also did a few minor customizations like the styling of the displayed strings, the proposal of arguments, and the proposal of possible components and variables for an assignment or method call.

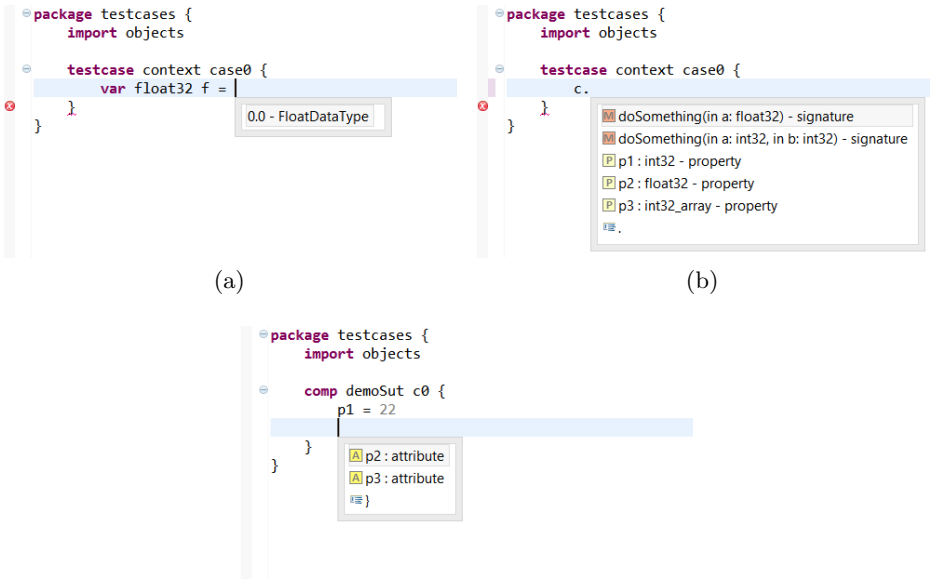
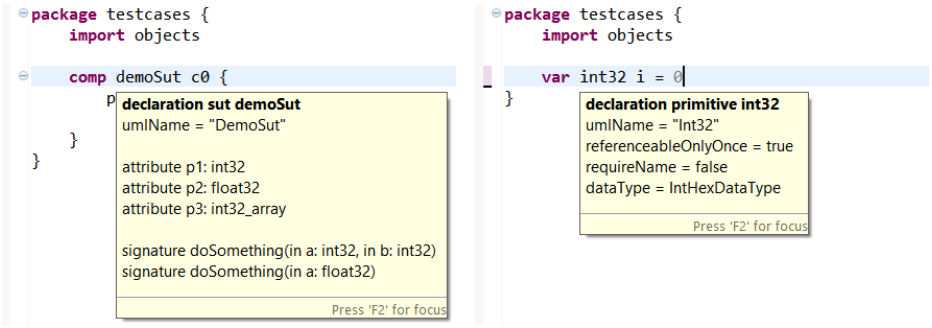


Figure 3.18: Ubtl content assist examples

- We added hover support. This is an optional feature of Xtext. If a user moves the mouse over a reference of a declaration or definition, a pop–up is displayed which illustrates attributes, settings, signatures and the *umlName*. Figure 3.19 illustrates two examples.

- We adjusted the outline view to only display *PackageElements*. Figure 3.20 shows the outline view with example content. If a user clicks on one of the displayed elements, then the editor jumps directly to it inside the code.



(a) (b)

Figure 3.19: Ubtl hover examples

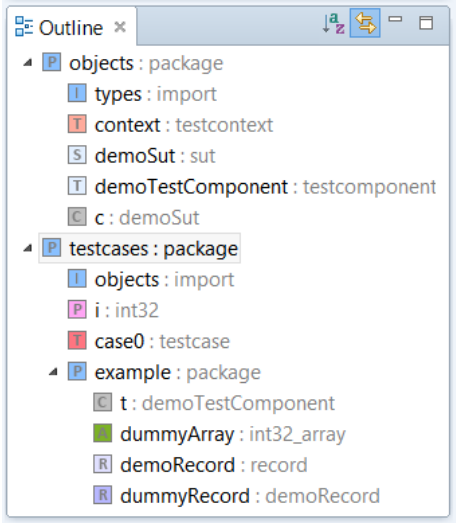


Figure 3.20: Ubtl outline view

- We added custom images and labels for all language elements. The images and labels are used by the outline view, while the content assist only uses the images.
- We provide a custom formatter. This feature is actually part of the compiler. Thus Ubtl code can be also formatted without the IDE and Eclipse.
- We provide wizards for the creation of Ubtl projects and Ubtl files. The project wizard is an optional feature of the Xtext framework and we customized it for Ubtl.
- We designed a custom icon for Ubtl files.
- By default an Ubtl file is compiled to UML when a user saves it. To ease the task of compiling Ubtl files, we provide a generate command for the package and project explorer. The command also works with multiple files. Figure 3.21 displays where a user can find the command.

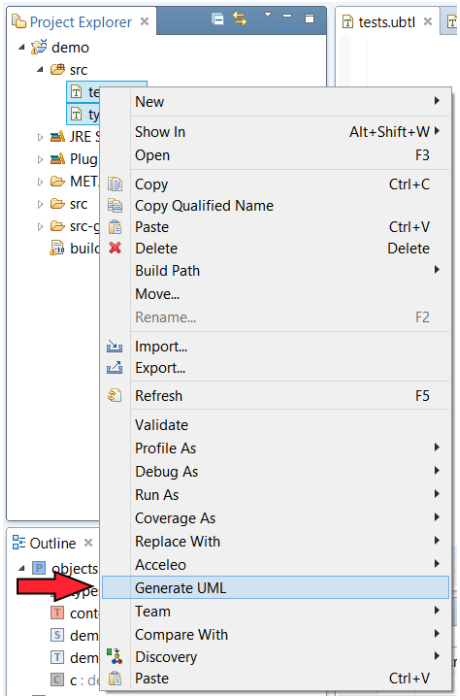


Figure 3.21: Ubtl generate UML command

3.6 Testing Ubtl

We developed Ubtl by using the Test Driven Development (TDD) methodology. As test platform we use the JUnit framework in conjunction with Xtext classes which ease the task of testing a DSL. The test cases are implemented with Xtend.

TDD basically consists of the following steps, [Bec03]:

- Quickly add a test.
- Run all tests and see the new one fail.
- Make a little change.
- Run all tests and see them all succeed.
- Refactor to remove duplication.

- Repeat.

A great side effect of this approach is that modular code is written in order to test it. Also it is easy to change a behavior and to check automatically whether something in the code is broken. We appreciated that while we developed Ubtl.

Concerning the Ubtl compiler, we specified 235 JUnit test cases. These test cases check each functionality in a separate way, including grammar, scope, validator, formatter, util classes and UML generator. Note that most of these test cases test a functionality with several inputs. Figure 3.22 illustrates successful JUnit runs and the test case classes. According to the code coverage tool EclEmma, available under the Eclipse Public License, we reach a coverage of 95.6% of the code we implemented. Concerning the IDE, we tested the content assist with 18 test cases.

Though we tested Ubtl carefully, we cannot guarantee that it is free of any bugs.

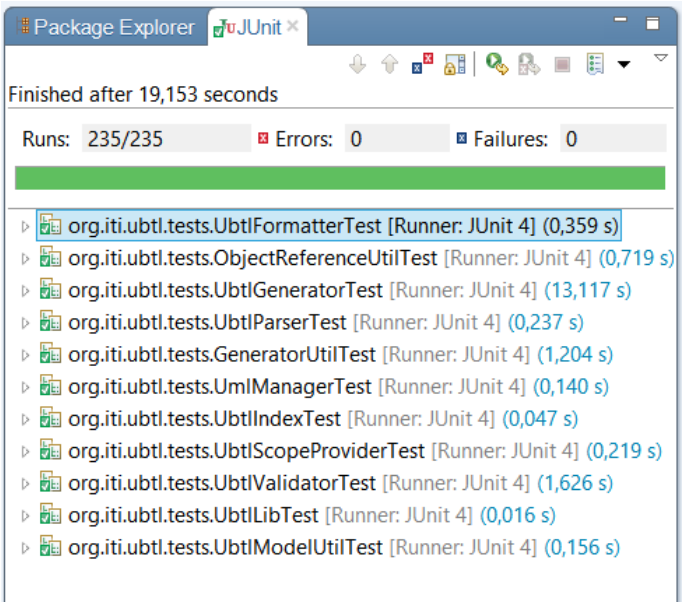


Figure 3.22: Successful JUnit runs

Chapter 4

Use Case

We use Ubtl to specify test cases for components running on embedded systems. Each component does a small task. These tasks range from simply adding two integers to complex controllers or filters. Those components can be connected to compositions which behave like applications. However, in the following we focus on unit testing of single components. We test those components directly on an embedded system with test cases specified in XML. These test cases are executed by the embedded system. We also test them on a virtualized operating system with QEMU. In that case, the components are tested in conjunction with a C++ framework and the test cases are written in the C++ programming language. Ubtl enables us to efficiently write a test case once and to test a component on completely different test platforms with the same test case logic. Additionally, we use Ubtl as back end for a software that generates test data and test cases. Figure 4.1 illustrates this test setup. On the upper half of this figure we see the front software. It interacts with Matlab and test results. Further it is used to specify the system and test data. From these specifications Ubtl test cases are derived. These Ubtl test cases are then transformed to test cases for the embedded system and for the QEMU framework. We leverage Acceleo for the different transformations. A benefit is that a test engineer still has the opportunity to manipulate or adapt Ubtl code. Further, the software does not have to be aware of different test platforms. The test platforms can evolve during development, without changing the front software.

4.1 Test Workflow

As an example we present a unit test case for a component that implements the cosinus function. Such a component has to be called with different input data to ensure that it works correctly. We specify input data and expected output data, also called oracle data. The oracle data represents the correct curve of the cosinus function. We test the component by setting the input data, running it, and finally evaluating the component output against the reference oracle data within specified thresholds. The thresholds represent the maximal deviation between the calculated curve and the oracle data.

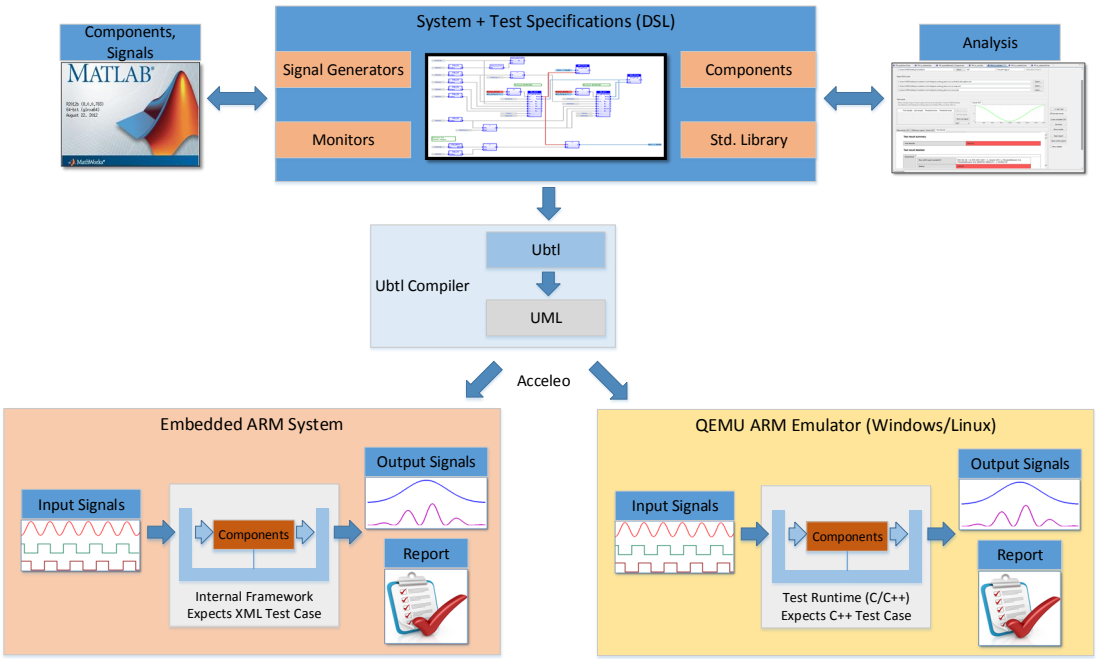


Figure 4.1: Test Setup with front software

4.2 Specifying a Test Case for a Component in Ubtl – COS

Listing 4.1 illustrates our predefined packages for testing components. These two packages are used by a test engineer or front software to specify test cases. The package *component_types* declares the primitive and array variable types. We restrict these types to be referenceable only once, so that a code generator can assume that a variable used by a component only belongs to this one. The package *component* declares the test context, an array named *handle*, the interface *component*, and the test components *component_assertions* and *component_monitor*. Currently we do not support timer, setVerdict, if, and log statements, therefore we disabled them inside the test context. The interface *component* specifies several attributes, for instance the fixed inputs of a component. Each attribute uses the array *handle* which accepts all primitive types. Additionally, the interface offers the signature *doWork()* which is for all components the same and executes a component. The test component *component_assertions* offers specialized assertions. The signature *assertThresholdBounded* asserts that the first operand equals the second operand within a specifiable threshold. The *component_monitor* is used to observe how a variable changes while a test case is executed. All these declarations are known by a code generator beforehand in order to transform test cases. It operates on the *umlNames* of the declarations.

Listing 4.1: Predefined packages for testing components

```
package component_types {
  declare primitive uint32 {
    umlName = "UInt32"
    acceptDataType = UIntHexDataType
    referenceableOnlyOnce = true
  }
  declare primitive float32 {
    umlName = "Float32"
    acceptDataType = FloatDataType
    referenceableOnlyOnce = true
  }
  // ...
  declare array float32_array {
    umlName = "Float32_Array"
    acceptTypes = float32
    oneReferenceMultipleTimes = false
    referenceableOnlyOnce = true
  }
   // ...
}
package component {
  import component_types

  declare testcontext component_context {
    umlName = "ComponentTestContext"
    disableTimer = true
    disableSetVerdict = true
    disableIf = true
    disableLog = true
  }

  declare array handle {
    umlName = "handle"
    acceptTypes = primitive
    requireNameOfPrimitiveVariables = true
    referenceableOnlyOnce = true
  }

  declare interface component {
    umlName = "Component"
    attribute fixedInputs: handle
    attribute expandableInputs: handle
    attribute outputs: handle
    attribute parameters: handle
    attribute systemVariables: handle
    signature doWork()
  }

  declare testcomponent component_assertions {
    umlName = "ComponentAssertions"
    signature assertThresholdBounded(in operand1: float32, in operand2:
    float32, in operand2_min: float32, in operand2_max: float32)
    // ...
  }

  declare testcomponent component_monitor {
    umlName = "ComponentMonitor"
    signature set(in arg: float32)
    signature set(in arg: uint32)
    // ...
  }
  // ...
}
```

Listing 4.2 illustrates the runtime objects and an example test case. The sut declaration declares the cosinus component. It realizes the interface *component*. The component definition *cos* represents the corresponding runtime object, with default values. Note that we could define several components of the cosinus component declaration. The components *assertions* and *monitor* represent the corresponding test component declarations. The two arrays *inputs* and *expected_outputs* hold the test data. We only specify a few values to keep the resulting XML file small. In fact, we would have to test the component with thousands of different test data. At the beginning of the test case we set the monitor to observe the output variable *OUT* of the component *cos_sut*. After that we iterate through the input data and the expected outputs. In each iteration we set the input variable of the component, run the component, and assert that the output is within an expected range.

Listing 4.2: Cosinus component test case

```
package testcases_component {
  import component

  declare sut sut_cos realizes component {
    umlName = "COS"
  }

  comp sut_cos cos {
    fixedInputs = float32 IN 0.0
    outputs = float32 OUT 0.0
    systemVariables = uint32 tA 1000
  }

  comp component_assertions assertions

  comp component_monitor monitor

  var float32_array inputs = float32 0.0, float32 4.514468643

  var float32_array expected_outputs = float32 1.0, float32 -0.196630695

  testcase component_context test {
    monitor.set(cos.outputs.get(OUT))
    foreach(i : inputs, j : expected_outputs) {
      cos.fixedInputs.get(IN) = i
      cos.doWork()
      assertions.assertThresholdBounded(cos.outputs.get(OUT), j, -0.00001, 0.00001)
    }
  }
}
```

4.3 Transformation to XML

We leverage Acceleo to define the transformations from the UML model to the different target platforms. See Subsection 2.5.4 for a short overview of different model–to–text tools of the Eclipse ecosystem. Acceleo is, in our opinion, easy to use and offers a textual notation which is standardized by the OMG. A generator can be used standalone or as Java library without a running Eclipse instance. Listing 4.3 illustrates the logic of our XML generator in pseudocode. Note that it does not matter where in the Ubtl code a monitor is set. We always generate the corresponding call at the beginning of a test case. *VariableManager* and *QualifiedNameManager* are Java classes. *VariableManager* holds

the current value (in fact an instance specification) of a variable and remembers if a value has recently changed through an UML assignment. *QualifiedNameManager* is responsible for the XML name of variables. If a variable is hold by a component it has a special syntax, while other variables have the name *$Const{value}*. Used variables of components, where the component is not called, are transformed to constant values.

Listing 4.3: XML code generator pseudocode

```
Foreach Test Context "ComponentTestContext"
  Foreach Test Case
    Generate XML Header

    // SETUP part of XML
    Foreach Called "Component"
      Set Instance Specifications In VariableManager And QualifiedNameManager
      Generate XML Component Properties
    Foreach "ComponentMonitor" "set" Call
      Generate XML Monitor

    // TEST part of XML
    Foreach Interaction Fragment
      If MessageOccurrenceSpecification And MessageSort::synchCall
        If "Component" "doWork" Call
          Check VariableManager
            Generate XML Changed Component Properties
          Generate XML
        Else If "ComponentAssertions" Call
          Generate XML Assertion
        Else If "ComponentMonitor" "set" Call
          Do Nothing
        Else
          Warning
      Else If "Loop"
        Iterate minint From loopGuard
          Generate Contained Interaction Fragments
      Else If "Foreach Loop"
        Iterate minint From foreachGuard
          Get Instance Specifications From foreachGuard
          Set Instance Specifications In VariableManager
          Generate Contained Interaction Fragments
      Else If CallOperationAction
        If "UBTLAssert" Call
          Generate XML Assertion
        Else If "UBTLValueSetter" Call
          Set Instance Specification In VariableManager
        Else If "UBTLVariableInitializer" Call
          Reset Instance Specification In VariableManager

    Generate XML Footer
```

4.4 Resulting XML code

Listing 4.4 illustrates the generated XML code by the Acceleo generator. We also generate C++ code with a different Acceleo generator, which is a bit complexer, but roughly the same. Therefore we omit to present this code. The XML code is used by the embedded system in order to perform a test case on the system. The results of the monitor and the assertions are sent back to a test engineer as report. Figure 4.2 shows the result which is computed from our XML test case.

Listing 4.4: Generated XML code

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<RDL:ResourceDescription xsi:schemaLocation="urn:AH:COMPONENT:RDL:1.0 testschema.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:RDL="urn:AH:COMPONENT:RDL:1.0"
name="Component Test Specification">
  <Content xsi:type="RDL:TestSpecification" name="test 1">
    <Header id="test" type="resources.test.componentunittest" version="0.1.0" />
    <Sequence>
      <Run xsi:type="RDL:SetValueRun" type="SETUP">
        <Specification name="Component{COS}.Inp{IN}" datatype="Float32" value="0.0" />
      </Run>
      <Run xsi:type="RDL:SetValueRun" type="SETUP">
        <Specification name="Component{COS}.Out{OUT}" datatype="Float32" value="0.0" />
      </Run>
      <Run xsi:type="RDL:SetValueRun" type="SETUP">
        <Specification name="Component{COS}.Sys{tA}" datatype="UInt32" value="1000" />
      </Run>
      <Run xsi:type="RDL:SetMonitorRun" type="SETUP" >
        <Specification name="Component{COS}.Out{OUT}"/>
      </Run>
      <Run xsi:type="RDL:SetValueRun" type="TEST">
        <Specification name="Component{COS}.Inp{IN}" datatype="Float32" value="0.0" />
      </Run>
      <Run xsi:type="RDL:CallMethodRun" type="TEST">
        <Specification name="Component{COS}.Op{doWork}" />
      </Run>
      <Run xsi:type="RDL:AssertRun" type="TEST">
        <Specification operand1="Component{COS}.Out{OUT}" operand2="$Const{1.0}"
        operand2_min="$Const{-0.00001}" operand2_max="$Const{0.00001}"
        operator="THRESHOLD_BOUNDED"/>
      </Run>
      <Run xsi:type="RDL:SetValueRun" type="TEST">
        <Specification name="Component{COS}.Inp{IN}" datatype="Float32" value="4.514468643" />
      </Run>
      <Run xsi:type="RDL:CallMethodRun" type="TEST">
        <Specification name="Component{COS}.Op{doWork}" />
      </Run>
      <Run xsi:type="RDL:AssertRun" type="TEST">
        <Specification operand1="Component{COS}.Out{OUT}" operand2="$Const{-0.196630695}"
        operand2_min="$Const{-0.00001}" operand2_max="$Const{0.00001}"
        operator="THRESHOLD_BOUNDED"/>
      </Run>
    </Sequence>
  </Content>
</RDL:ResourceDescription>
```

4.5 Evaluation of the Use Case

We evaluated Ubtl by using the cosinus test case with a different amount of values hold by the arrays *inputs* and *expected_outputs*. Figure 4.3 illustrates the results of our measurement. The test case explained and specified above is the first one in the measurement. The first bar named *Ubtl* refers to the case when a user generates UML code from Ubtl code in Eclipse. It consists of the steps parsing an Ubtl file, validating the Ubtl model, generating an UML model from the Ubtl model, and writing UML files. The second bar *Acceleo* illustrates the seconds spent when a user generates an XML file of an UML model in Eclipse. The bar *Ubtl & Acceleo* is a combination of these two generators, like they are used in our software which generates test data and test cases. Involved steps are generating an UML model from an Ubtl model, initializing the Acceleo generator with the

Tool status

Return status of the test framework	Testrun finished

Test result summary

Test Result	FAILED

Test result detailed

Assertions		
	Run with input sample#1	COM IN: IN = 0, COM OUT: OUT = 1, Oracle OUT: 1, Threshold(min): -0.0001, Threshold(max): 0.0001, ABS(COM-ORACLE) = 1.19209e-07
	Status	PASSED
	Run with input sample#2	COM IN: IN = 4.51447, COM OUT: OUT = -0.195541, Oracle OUT: -0.196630695, Threshold(min): -0.0001, Threshold(max): 0.0001, ABS(COM-ORACLE) = 0.00108938
	Status	FAILED

Figure 4.2: Returned test case report

generated UML model, generating and writing the XML file. It does not contain the steps parsing an Ubtl file, validating an Ubtl model, and writing an UML file. Therefore it is slightly faster than using these two generators separately in Eclipse when more values are used. *TP* stands for test platform and illustrates the time spent for parsing an XML file on the embedded system, initializing a test case, executing a test case, and generating the results. We executed each case ten times and took the arithmetic mean. We executed our measurements on a computer with an Intel Core i5–4200M CPU (2.5 GHz). The hard disk has an average sequential read speed of 124,838 MB/s and a write speed of 100,455 MB/s. The RAM has an average speed of 10644,65 MB/s. We obtained those values from Winsat, by executing it ten times and calculating the arithmetic mean. On a slower system, Ubtl and Acceleo may take longer. As we can see in the measurement, Ubtl and Acceleo generations take significantly longer when the amount of data is increased. Note that we did not optimize the code of the generators with respect to speed, therefore it may be possible to decrease their execution time. However, we think they are useable in their current state.

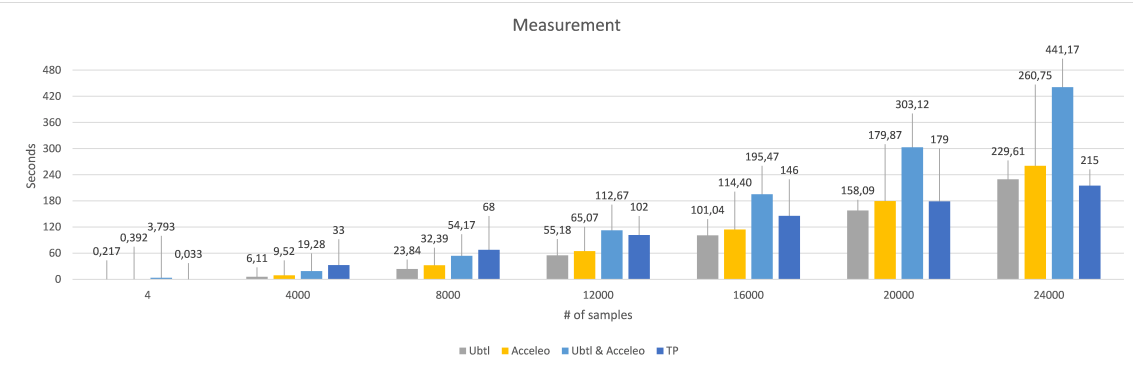


Figure 4.3: Measurement with different numbers of input and output values

Figure 4.4 shows the different file sizes of Ubtl, UML, and XML code in bytes. The UML file sizes include all generated UML models. The Ubtl file sizes only consist of the *testcases_component* package. Obviously a test case written with Ubtl is more compact than with UML and XML.

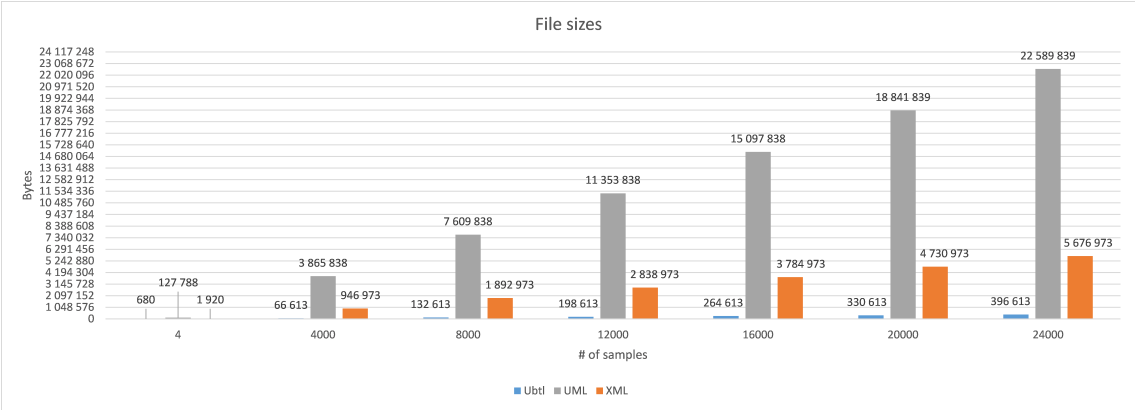


Figure 4.4: File sizes of Ubtl, UML, and XML code

Chapter 5

Conclusion

All together, we can say that we reached the goal to construct a textual test specification language which is abstract, flexible, and easy to use. The resulting UML models enable us to transform test case logic to arbitrary platforms. We successfully apply Ubtl for our use case in an industrial project, where we use Ubtl for testing single components. The separation between declarations and definitions makes it simple to realize generators for any notation. The Eclipse Ubtl IDE offers several features to help users in the task of specifying types, objects, and test cases.

Currently Ubtl is suited for unit testing. For the other levels of testing it lacks certain features like mock objects, test component operations, or test configurations. However, despite this missing features, Ubtl may be successful leveraged on other levels.

We could have implemented Ubtl in the way, that the generated models are not build with UML or do not depend on UTP. In our opinion, this would be the same like reinventing the wheel. UML simply offers a compatibility with other tools, we would have never reached. UTP is actively maintained by several companies and people. It is standardized and offers well–thought–out concepts.

Compared with TTCN–3, Ubtl test cases can be used for arbitrary systems, notations, or test frameworks, while the TTCN–3 test cases are restricted to be used by the standardized TTCN–3 test architecture.

In our opinion, Ubtl is especially useful if different platforms with incompatible test case notations are present. It may also be used when there only exists one test platform, and the final test notation is too complicated. Further Ubtl can be used when the test platform is still under development.

Chapter 6

Future Work

Ubtl is in its current state far away from being perfect. In the following we discuss features which we identified to be useful, but had not the time to implement and were out of scope for this thesis.

Test Component Operations

At the moment it is only possible to specify signatures of operations. A useful feature would be to allow the specification of an operation body similar on how test cases are specified. The scope of such an operation body would be limited to parameters and the properties of a test component. In UML it would be simply realized as interaction part of a test component. This feature would allow the specification of mock objects which simulate the behavior of components. Another use case would be helper methods programmed with Ubtl. Such methods could be reused to set up or tear down components and entire test cases. Listing 6.1 illustrates how the syntax could look like in action. The second operation realizes a signature defined by the interface *demoInterface*.

Listing 6.1: Test Component operations

```
declare testcomponent demo realizes demoInterface {
  umlName = "Demo"
  required attribute math : Float32Math

  operation arithmeticMean(in arg : float32_array): float32 {
    var float32 sum = 0.0
    var float32 n = 0.0
    foreach(x : arg) {
      sum = math.add(sum, x)
      n = math.add(n, 1.0)
    }
    var float32 mean = 0.0
    mean = math.divide(sum, n)
    return mean
  }

  operation realizes doSomething(sut0 : demoSut) {
    var boolean x = false
    x = sut0.doIt()
    return x
  }
}
```

Enhanced Primitive Variables

Currently only the data type of primitive variables can be specified. However, it is not possible to specify properties like maximal data size. For instance, it is not possible to specify that an integer is limited to 32 bits or that a string is limited to 100 characters. Therefore, a useful feature would be a matching mechanism for primitive variables like it is provided by TTCN–3. In TTCN–3 for instance one can restrict the range of numeric values or define string patterns.

Data Modification

Another useful enhancement would be to implement the data modification dependency of UTP. This concept is actually borrowed from TTCN–3. Listing 6.2 illustrates how this concept could look like in Ubtl. Variable *b* automatically inherits *aProperty* and *arrayOne*, while it overrides *arrayTwo*. It also adds the property *anOptionalProperty* which is not specified by the variable *a*. In UML the relationship between the instance specifications would be illustrated with a modification dependency. Ubtl would have to create a dummy value for each inherited value, otherwise it would not be possible to reference a value in a test case and to decide whether it belongs to variable *a* or *b*. The names of the dummy values could start with a special symbol to indicate that the values are a result of a modification. It depends on the executing test system if values are actually copied or if references on the values of for instance variable *a* are used. The advantage of this concept is that it makes it easy to create different variables which are actually only slightly different. It could also be applied for components concerning data properties.

Listing 6.2: Ubtl data modification

```
var aType a {
  aProperty = 34
  arrayOne = int32 0, int32 1
  arrayTwo = int32 2, int32 3
}

var aType b modifies a {
  arrayTwo = int32 2, int32 1
  anOptionalProperty = 1
}
```

Type Conversion

Ubtl does not allow changing the type of a variable into an other. In some cases this feature could be useful, for instance to assign an integer to a float variable or to compare an integer with an other integer type. Therefore we suggest the possibility to make an explicit cast, where the syntax is similar to the Java programming language. Implicit casts should not be possible as they could lead to mistakes and unexpected behavior. Additionally it should be possible to disable this feature inside a test context and a primitive declaration. It could be possible to define compatible types inside a primitive declaration. Note that type conversion can be simulated in the current version of Ubtl by providing a test component

which offers corresponding signatures (Listing 6.3). However a standard way of doing that would be better.

Listing 6.3: Test Component which provides type conversion

```
declare testcomponent TypeConversion {
  umlName = "TypeConversion"

  signature cast(in i : int32): float32
  signature cast(in f : float32): int32
}
```

Arithmetic Operators

Like type conversion, at the moment arithmetic operators have to be simulated by using a test component. Again a standard way could make the Ubtl code easier to read and would make code generators less specialized. Also it should be possible to disable this feature within test contexts and primitive declarations.

Constant Variables and Components

In the current version there are no restrictions to alter the data of variables and components. It is just possible to disable all assignments in test cases by a test context. An enhancement for test engineers would be to allow the specification of constant objects. The keyword could be *const* instead of *var*.

Diagrams

Ubtl currently only generates the abstract syntax of UML. If it is in the future possible to exchange diagrams between different tools then Ubtl could generate Class Diagrams and Sequence Diagrams for test cases. This feature would make it easier to manipulate a generated UML model with UML tools. Also the diagrams could be used for documenting test cases.

Test Configuration

UTP offers the possibility to define a test configuration which specifies the possible connections between component objects. A test configuration uses the composite structure for structured classifier in UML and is part of a test context, [Obj13c]. This feature could be useful in Ubtl for integration testing. Listing 6.4 illustrates how test configurations could be realized in Ubtl. We could introduce the concept *configuration* which allows to connect runtime component objects. Such a configuration would be used by a test case. The optional feature *restrictTestContext* may be used to restrict the available components inside a test case. If this feature would be set to false all available components can be called. If two test cases use the same test context, but different test configurations, they would be part of two different test contexts.

Listing 6.4: Ubtl test configuration

```
package testcases {
  comp sutA sut0
  comp sutB sut1
  comp tcompA tc0
  comp tcompA tc1

  configuration aConfiguration {
    restrictTestContext = false
    connect tc0 and sut0
    connect tc0 and sut1
    connect sut0 and tc1
    connect sut1 and tc1
  }

  testcase aContext case uses aConfiguration {
    //...
  }
}
```

Appendix A

Ubtl Grammar

Listing A.1: Ubtl grammar

```
grammar org.iti.ubtl.Ubtl with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate ubtl "http://www.iti.tugraz.at/ubtl"

/*
 * Parser Rules
 */

UbtlModel:
  packages+=Package*
;

Package:
  "package" name=ID "{"
    imports+=PackageImport*
    content+=PackageElement*
  "}" Semicolon
;

PackageImport:
  "import" package=[Package|QualifiedPackageName] Semicolon
;

QualifiedPackageName:
  ID("."ID)*
;

PackageElement:
  Package | Declaration | ComponentDefinition | FullVariableDefinition | TestCase
;

Declaration:
  "declare" (ObjectDeclaration|TestContextDeclaration) Semicolon
;

ObjectDeclaration:
  InterfaceDeclaration|ComponentDeclaration|VariableDeclaration
;

InterfaceDeclaration:
  "interface" name=ID "{"
    ( ("umlName""=" umlName=STRING Semicolon)
```

```
    &(features+=(SignatureDeclaration|AttributeDeclaration) Semicolon)*
   )
  "}"
;

ComponentDeclaration:
  SutDeclaration|TestComponentDeclaration
;

SutDeclaration:
  "sut" name=ID ("realizes" interfaceRealizations+=[InterfaceDeclaration]
      ("," interfaceRealizations+=[InterfaceDeclaration])*)?  "{"
   ( ("umlName""=" umlName=STRING Semicolon)
    &(features+=(SignatureDeclaration|AttributeDeclaration) Semicolon)*
   )
  "}"
;

TestComponentDeclaration:
  "testcomponent" name=ID ("realizes" interfaceRealizations+=[InterfaceDeclaration]
      ("," interfaceRealizations+=[InterfaceDeclaration])*)?  "{"
   ( ("umlName""=" umlName=STRING Semicolon)
    &(features+=(SignatureDeclaration|AttributeDeclaration) Semicolon)*
   )
  "}"
;

Feature:
  SignatureDeclaration|AttributeDeclaration|PropertyDefinition
;

SignatureDeclaration:
  "signature" name=ID "(" (parameters += ParameterDeclaration ("," parameters +=
      ParameterDeclaration)*)? ")" (returnParameter=ReturnParameterDeclaration)?
;

ParameterDeclaration:
  (parameterDirection=ParameterDirection)? name=ID ":" type=[ObjectDeclaration]
;

enum ParameterDirection:
  IN="in" | INOUT="inout" | OUT="out"
;

ReturnParameterDeclaration returns ParameterDeclaration:
  {ReturnParameterDeclaration}":" type=[ObjectDeclaration]
;

VariableDeclaration:
  PrimitiveDeclaration|RecordDeclaration|ArrayDeclaration
;

PrimitiveDeclaration:
  "primitive" name=ID "{"
   ( ("umlName""=" umlName=STRING Semicolon)
    &("referenceableOnlyOnce""="referenceableOnlyOnce=BooleanValue Semicolon)?
    &("requireName""="requireName=BooleanValue Semicolon)?
    &("acceptDataType""=" acceptDataType=(
     "IntDataType"|"IntHexDataType"|"UIntDataType"|"UIntHexDataType"|
     "HexDataType"|"StringDataType"|"FloatDataType"|"BooleanDataType") Semicolon)
   )
  "}"
;
```

```
RecordDeclaration:
  "record" name=ID "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &("referenceableOnlyOnce""="referenceableOnlyOnce=BooleanValue Semicolon)?
     &(attributes+=AttributeDeclaration Semicolon)*
    )
  "}"
;

AttributeDeclaration:
  (required?="required")? "attribute" name=ID ":" type=[ObjectDeclaration]
;

ArrayDeclaration:
  "array" name=ID "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &("referenceableOnlyOnce""="referenceableOnlyOnce=BooleanValue Semicolon)?
     &(types=AcceptTypes Semicolon)
     &("oneReferenceMultipleTimes""="oneReferenceMultipleTimes=BooleanValue Semicolon)?
     &("requireNameOfPrimitiveVariables""="requireNameOfPrimitiveVariables=BooleanValue
          Semicolon)?
    )
  "}"
;

AcceptTypes:
  "acceptTypes""=" (acceptTypes=ObjectTypeSet|acceptArray?="array"|
  acceptRecord?="record"|acceptPrimitive?="primitive"|acceptVariable?="variable")
;

ObjectTypeSet:
  objectTypeSet+=[ObjectDeclaration] ("," objectTypeSet+=[ObjectDeclaration])*
;

TestContextDeclaration:
  "testcontext" name=ID "{"
    ( ("umlName""=" umlName=STRING Semicolon)
     &("disableVariableDefinition""="disableVariableDefinition=BooleanValue Semicolon)?
     &("disableLoop""="disableLoop=BooleanValue Semicolon)?
     &("disableSetVerdict""="disableSetVerdict=BooleanValue Semicolon)?
     &("disableAssignment""="disableAssignment=BooleanValue Semicolon)?
     &("disableAssertion""="disableAssertion=BooleanValue Semicolon)?
     &("disableForEachLoop""="disableForEachLoop=BooleanValue Semicolon)?
     &("disableTimer""="disableTimer=BooleanValue Semicolon)?
     &("disableIf""="disableIf=BooleanValue Semicolon)?
     &("disableIfComplexCondition""="disableIfComplexCondition=BooleanValue Semicolon)?
     &("disableIfOr""="disableIfOr=BooleanValue Semicolon)?
     &("disableIfAnd""="disableIfAnd=BooleanValue Semicolon)?
     &("disableIfEquality""="disableIfEquality=BooleanValue Semicolon)?
     &("disableIfComparison""="disableIfComparison=BooleanValue Semicolon)?
     &("disableIfNot""="disableIfNot=BooleanValue Semicolon)?
     &("disableLog""="disableLog=BooleanValue Semicolon)?
    )
  "}"
;

ObjectDefinition:
  ComponentDefinition| FullVariableDefinition| DataDefinitionElement| ForEachVariableDefinition
;

ComponentDefinition:
  "comp" ((type=[ComponentDeclaration])|(timer?="timer")) name=ID ("{"
    (properties+=PropertyDefinition Semicolon)*
  "}")? Semicolon
;
```

```
VariableDefinition:
  FullVariableDefinition|DataDefinitionElement|ForEachVariableDefinition
;

FullVariableDefinition:
  "var" ((type=[VariableDeclaration])|(duration?="duration")) name=ID
  ( (bigDefinition?="{"
      (properties+=(PropertyDefinition| PrimitivePropertyDefinition| ArrayPropertyDefinition)
           Semicolon)*
     "}")
    |(smallDefinition?="=" dataDefinition=DataDefinition)
  ) Semicolon
;

PropertyDefinition:
  name=ID "=" dataDefinition=DataDefinition
;

PrimitivePropertyDefinition returns PropertyDefinition:
  {PrimitivePropertyDefinition}name="value" "=" dataDefinition=DataDefinition
;

ArrayPropertyDefinition returns PropertyDefinition:
  {ArrayPropertyDefinition}name="content" "=" dataDefinition=DataDefinition
;

DataDefinition:
  (data=Data)|(elements+=DataDefinitionElement ("," elements+=DataDefinitionElement)*)
;

ObjectOrPrimitiveDeclaration:
  ComponentDefinition|FullVariableDefinition|PrimitiveDeclaration
;

DataDefinitionElement:
  variableOrValueType=[ObjectOrPrimitiveDeclaration] ((name=ID)? data=Data)?
;

Data:
  (sign?="-")? (int=IntString| hex=HexString| string=STRING| float=FloatString|
       boolean=BooleanString)
;

TestCase:
  "testcase" testContext=[TestContextDeclaration] name=ID
    block=Block
;

AbstractBlock:
  Block|OneStatementBlock
;

Block:
  {Block}"{" (statements += Statement )* "}" Semicolon
;

OneStatementBlock:
  statements += Statement
;

Statement:
  FullVariableDefinition|
  Loop|
  SetVerdict|
```

```
  AssignmentOrCall|
  Assertion|
  ForEachLoop|
  IfStatement|
  LogStatement
;

ReferenceableObject:
  ComponentDefinition|FullVariableDefinition|ForEachVariableDefinition
;

ObjectReference:
  object=[ReferenceableObject] (features+=ReferenceFeature)* (methodCall?="("
       (arguments+=Argument("," arguments+=Argument)*)? ")")?
;

ReferenceFeature:
  "." ((feature=ID)| get?="get""("(getElement=[DataDefinitionElement]|getPosition=INT)")")
;

Argument:
  {Argument} ((reference=ObjectReference)|(data=Data))
;

AssignmentOrCall:
  reference=ObjectReference (assignment?="=" assignmentArgument=Argument)? Semicolon
;

Loop:
  "loop" "(" iterations=INT ")" block=AbstractBlock
;

SetVerdict:
  "setVerdict" "(" verdict=Verdict ")" Semicolon
;

enum Verdict:
  NONE="none"|PASS="pass"|INCONCLUSIVE="inconclusive"|FAIL="fail"|ERROR="error"
;

Assertion:
  "assert""("leftArgument=Argument assertionType=AssertionType rightArgument=Argument")" Semicolon
;

enum AssertionType:
  EQUAL="==" | GREATERTHAN=">" | LOWERTHAN="<" | GREATERTHANOREQUALTO=">="|
  LOWERTHANOREQUALTO="<=" | NOTEQUAL="!="
;

ForEachLoop:
  "foreach""("definitions+=ForEachVariableDefinition(","
       definitions+=ForEachVariableDefinition)*")"
    block=AbstractBlock
;

ForEachVariableDefinition:
  name=ID ":" reference=ObjectReference
;

IfStatement:
  "if" "("condition=Condition")" thenBlock=AbstractBlock
  (=> "else" elseBlock=AbstractBlock)?
;
```

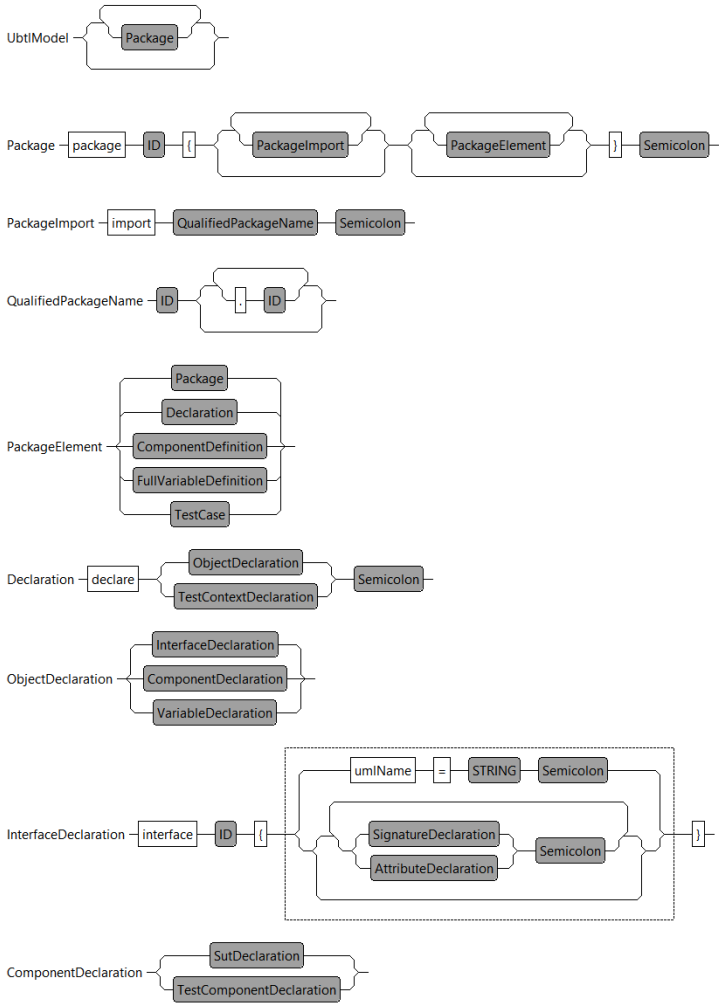
```
Condition:
  Or
;

Or returns Condition:
  And ({Or.left=current} "||" right=And)*
;

And returns Condition:
  Equality ({And.left=current} "&&" right=Equality)*
;

Equality returns Condition:
  Comparison (
    {Equality.left=current} op=("=="|"!=")
    right=Comparison
  )*
;

Comparison returns Condition:
  Primary (
    {Comparison.left=current} op=(">="|"<="|">"|"<")
    right=Primary
  )*
;

Primary returns Condition:
  "(" Condition ")" |
  {Not} "!" "("condition=Condition")" |
  Atomic
;

Atomic returns Condition:
  {Atomic} argument=Argument
;

LogStatement:
  "log" "(" information=STRING ")" Semicolon
;

ContentSuperType:
  PackageElement|DataDefinitionElement|ForEachVariableDefinition
;

IntString returns ecore::EString:
  INT
;

HexString returns ecore::EString:
  HEX
;

FloatString returns ecore::EString:
  IntString"."IntString
;

BooleanString returns ecore::EString:
  "false"|"true"
;

BooleanValue returns ecore::EBoolean:
  "false"|"true"
;
```

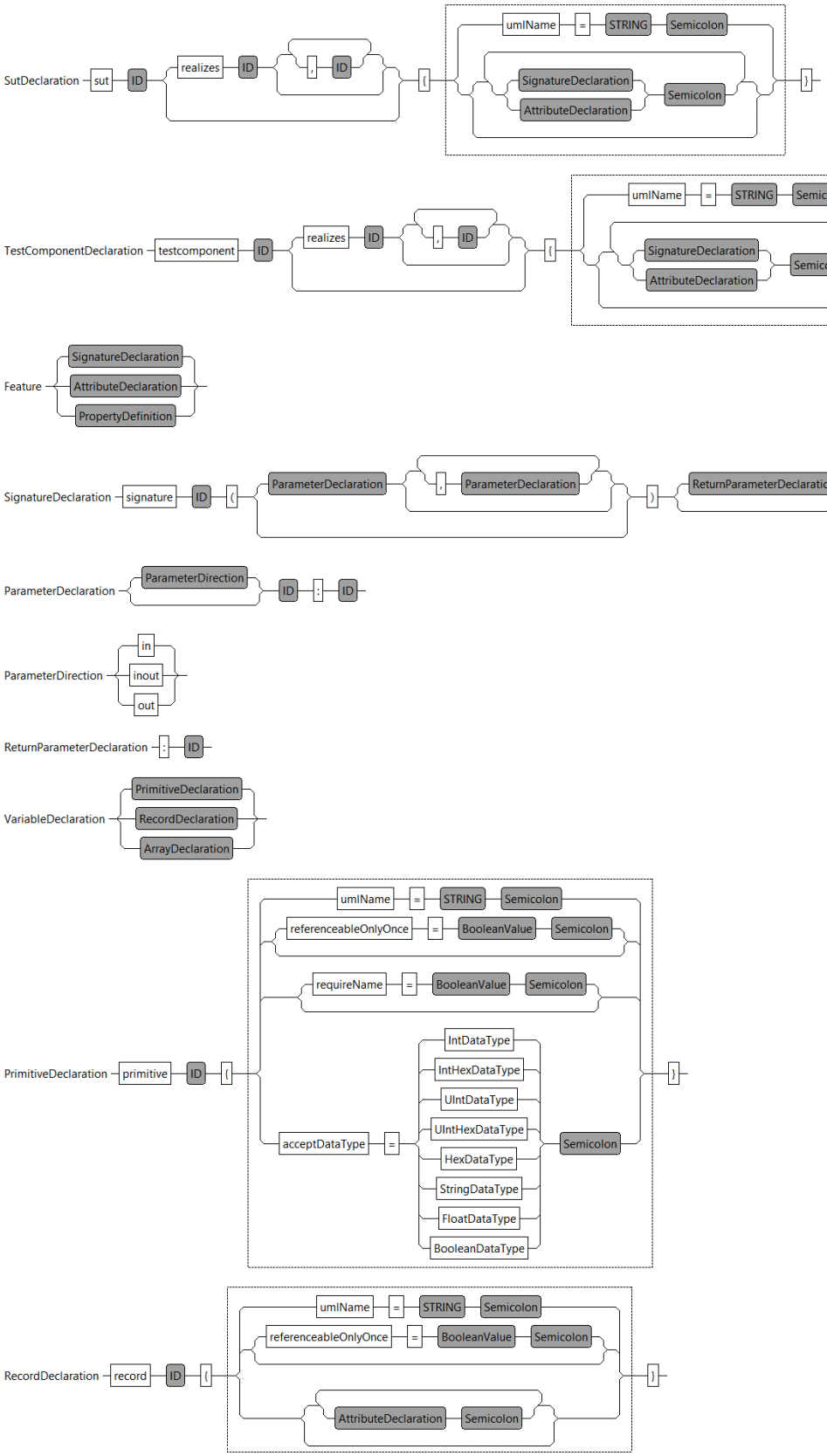
```
Semicolon:
  (";")?
;

/*
 * Terminal Rules
 */

terminal HEX:
  ("0x"|"0X") ("A"..."F"|"a"..."f"|"0"..."9")+
;
```

Listing A.2: Xtext Terminals

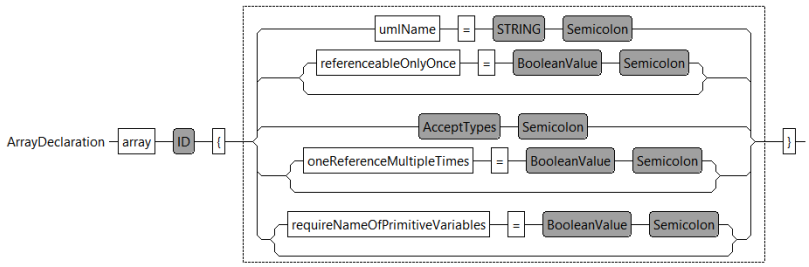
```
/*******************************************************************************
 * Copyright (c) 2008 itemis AG and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *******************************************************************************/
grammar org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

terminal ID     : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING :
      '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )* '"' |
      "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )* "'"
    ;
terminal ML_COMMENT : '/*' -> '*/';
terminal SL_COMMENT   : '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS     : (' '|'\t'|'\r'|'\n')+;

terminal ANY_OTHER: .;
```

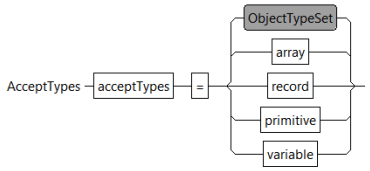
Figure A.1: Ubtl syntax graph

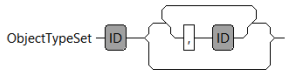


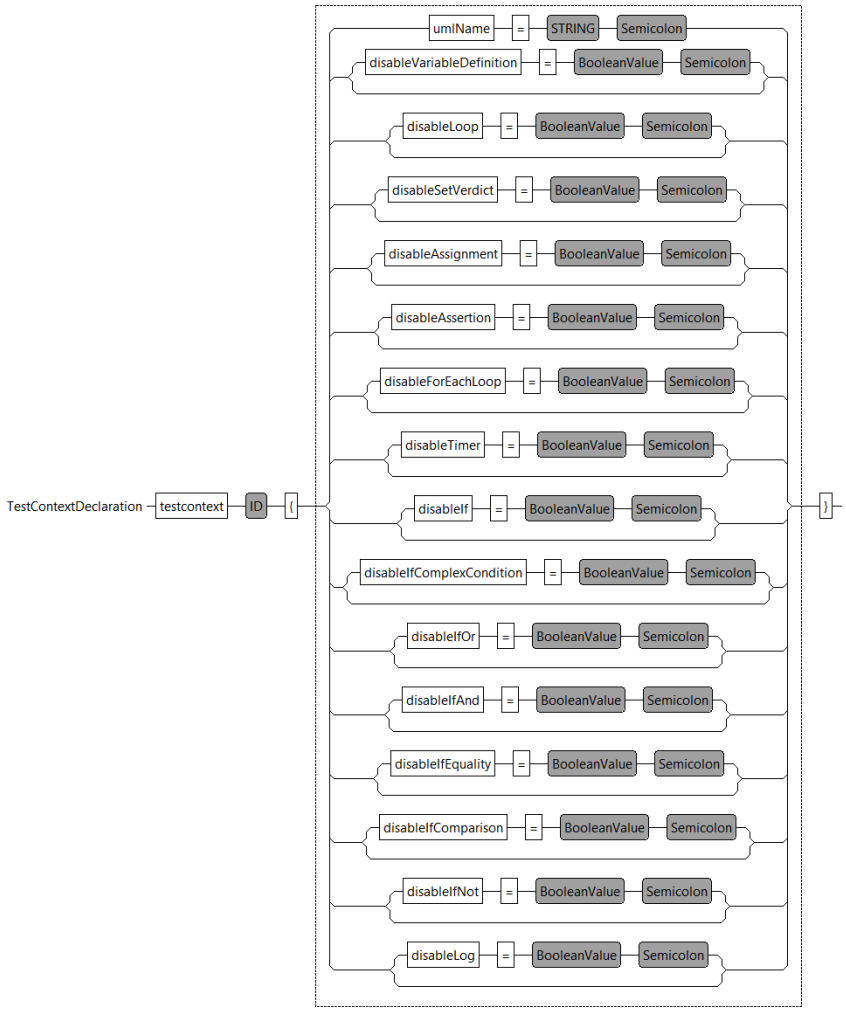

SutDeclaration
sut
ID
realizes
ID
,
ID
{
umlName
=
STRING
Semicolon
SignatureDeclaration
AttributeDeclaration
Semicolon
}
TestComponentDeclaration
testcomponent
ID
realizes
ID
,
ID
{
umlName
=
STRING
Semicolon
SignatureDeclaration
AttributeDeclaration
Semicolon
}
Feature
SignatureDeclaration
AttributeDeclaration
PropertyDefinition
SignatureDeclaration
signature
ID
(
ParameterDeclaration
,
ParameterDeclaration
)
ReturnParameterDeclaration
ParameterDeclaration
ParameterDirection
ID
:
ID
ParameterDirection
in
inout
out
ReturnParameterDeclaration
:
ID
VariableDeclaration
PrimitiveDeclaration
RecordDeclaration
ArrayDeclaration
PrimitiveDeclaration
primitive
ID
{
umlName
=
STRING
Semicolon
referenceableOnlyOnce
=
BooleanValue
Semicolon
requireName
=
BooleanValue
Semicolon
acceptDataType
=
IntDataType
IntHexDataType
UIntDataType
UIntHexDataType
HexDataType
StringDataType
FloatDataType
BooleanDataType
Semicolon
}
RecordDeclaration
record
ID
{
umlName
=
STRING
Semicolon
referenceableOnlyOnce
=
BooleanValue
Semicolon
AttributeDeclaration
Semicolon
}

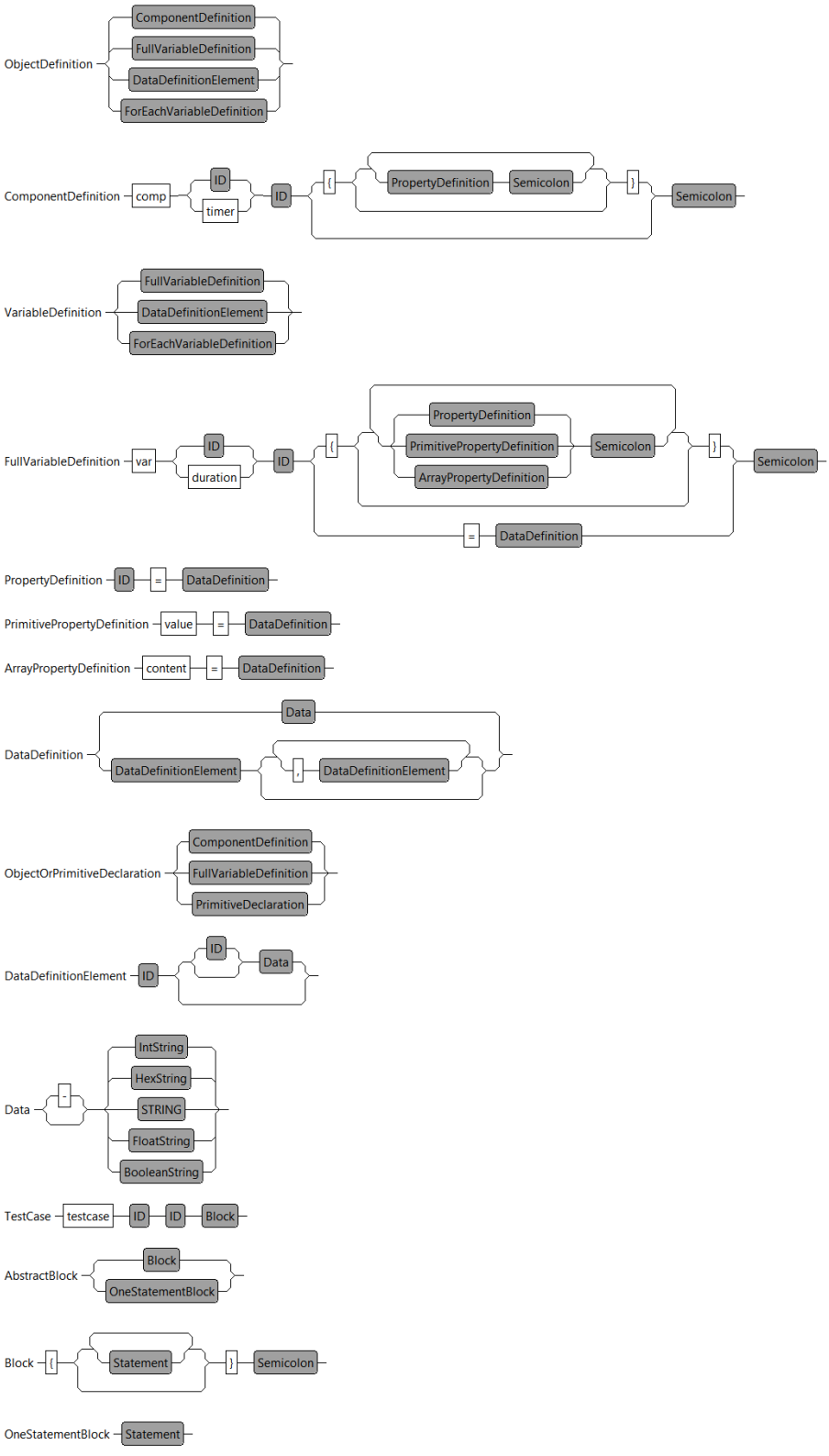

AttributeDeclaration
required
attribute
ID
:
ID

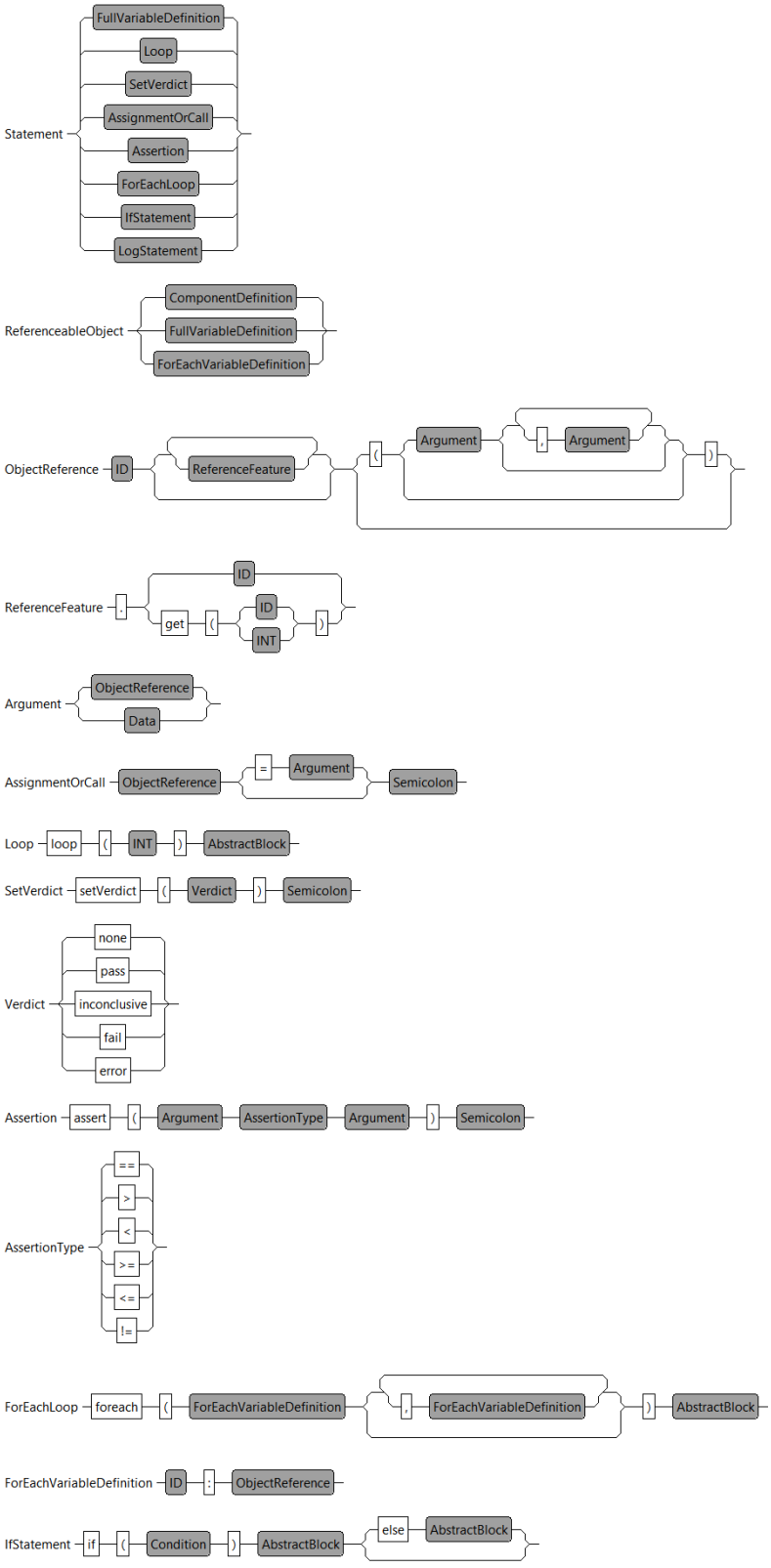

ArrayDeclaration
array
ID
{
umlName
=
STRING
Semicolon
referenceableOnlyOnce
=
BooleanValue
Semicolon
AcceptTypes
Semicolon
oneReferenceMultipleTimes
=
BooleanValue
Semicolon
requireNameOfPrimitiveVariables
=
BooleanValue
Semicolon
}

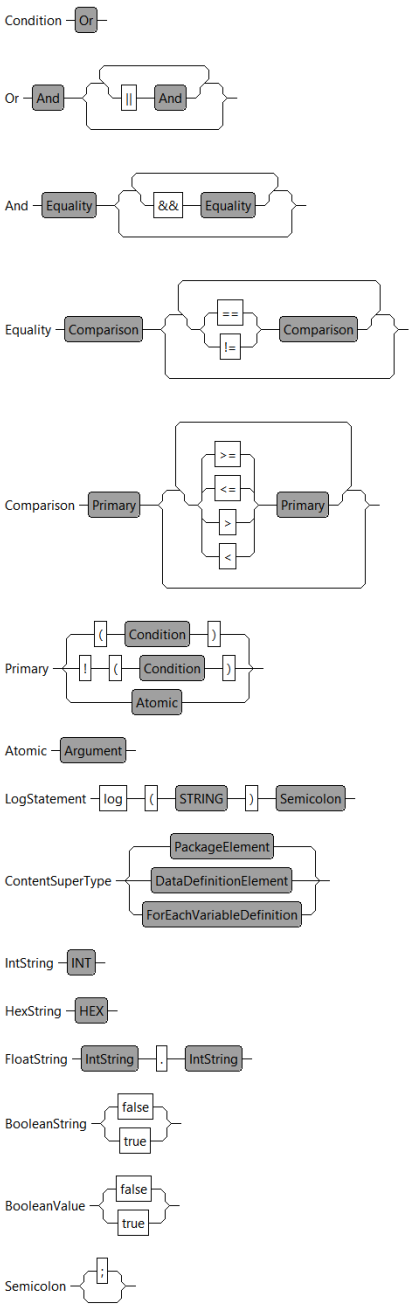

AcceptTypes
acceptTypes
=
ObjectTypeSet
array
record
primitive
variable


ObjectTypeSet
ID
,
ID


TestContextDeclaration
testcontext
ID
{
umlName
=
STRING
Semicolon
disableVariableDefinition
=
BooleanValue
Semicolon
disableLoop
=
BooleanValue
Semicolon
disableSetVerdict
=
BooleanValue
Semicolon
disableAssignment
=
BooleanValue
Semicolon
disableAssertion
=
BooleanValue
Semicolon
disableForEachLoop
=
BooleanValue
Semicolon
disableTimer
=
BooleanValue
Semicolon
disableIf
=
BooleanValue
Semicolon
disableIfComplexCondition
=
BooleanValue
Semicolon
disableIfOr
=
BooleanValue
Semicolon
disableIfAnd
=
BooleanValue
Semicolon
disableIfEquality
=
BooleanValue
Semicolon
disableIfComparison
=
BooleanValue
Semicolon
disableIfNot
=
BooleanValue
Semicolon
disableLog
=
BooleanValue
Semicolon
}


ObjectDefinition
ComponentDefinition
FullVariableDefinition
DataDefinitionElement
ForEachVariableDefinition
ComponentDefinition
comp
ID
timer
ID
{
PropertyDefinition
Semicolon
}
Semicolon
VariableDefinition
FullVariableDefinition
DataDefinitionElement
ForEachVariableDefinition
FullVariableDefinition
var
ID
duration
ID
{
PropertyDefinition
PrimitivePropertyDefinition
ArrayPropertyDefinition
Semicolon
}
Semicolon
=
DataDefinition
PropertyDefinition
ID
=
DataDefinition
PrimitivePropertyDefinition
value
=
DataDefinition
ArrayPropertyDefinition
content
=
DataDefinition
DataDefinition
Data
DataDefinitionElement
,
DataDefinitionElement
ObjectOrPrimitiveDeclaration
ComponentDefinition
FullVariableDefinition
PrimitiveDeclaration
DataDefinitionElement
ID
ID
Data
Data
-
IntString
HexString
STRING
FloatString
BooleanString
TestCase
testcase
ID
ID
Block
AbstractBlock
Block
OneStatementBlock
Block
{
Statement
}
Semicolon
OneStatementBlock
Statement


Statement
FullVariableDefinition
Loop
SetVerdict
AssignmentOrCall
Assertion
ForEachLoop
IfStatement
LogStatement
ReferenceableObject
ComponentDefinition
FullVariableDefinition
ForEachVariableDefinition
ObjectReference
ID
ReferenceFeature
(
Argument
,
Argument
)
ReferenceFeature
.
ID
get
(
ID
INT
)
Argument
ObjectReference
Data
AssignmentOrCall
ObjectReference
=
Argument
Semicolon
Loop
loop
(
INT
)
AbstractBlock
SetVerdict
setVerdict
(
Verdict
)
Semicolon
Verdict
none
pass
inconclusive
fail
error
Assertion
assert
(
Argument
AssertionType
Argument
)
Semicolon
AssertionType
==
>
<
>=
<=
!=
ForEachLoop
foreach
(
ForEachVariableDefinition
,
ForEachVariableDefinition
)
AbstractBlock
ForEachVariableDefinition
ID
:
ObjectReference
IfStatement
if
(
Condition
)
AbstractBlock
else
AbstractBlock


Condition
Or
Or
And
||
And
And
Equality
&&
Equality
Equality
Comparison
==
!=
Comparison
Comparison
Primary
>=
<=
>
<
Primary
Primary
(
Condition
)
!
(
Condition
)
Atomic
Atomic
Argument
LogStatement
log
(
STRING
)
Semicolon
ContentSuperType
PackageElement
DataDefinitionElement
ForEachVariableDefinition
IntString
INT
HexString
HEX
FloatString
IntString
.
IntString
BooleanString
false
true
BooleanValue
false
true
Semicolon
;

Bibliography

[Acc] Acceleo Homepage. `http://www.eclipse.org/acceleo/`. Visited in June 2014.

[BÓ4] Jean Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004.

[BÓ5] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[Bae10] Stefan Baerisch. *Domain-Specific Model-Driven Testing*. Vieweg+Teubner, 2010.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*, volume 1. Morgan & Claypool Publishers, September 2012.

[BDG+08] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[Bec03] Kent Beck. *Test-driven Development: By Example*. Kent Beck Signature Book. Addison-Wesley Professional, 2003.

[Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Crn02] Ivica Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., 2002.

[Dai04] Zhen Ru Dai. Model-driven testing with UML 2.0. *Computer Science at Kent*, page 179, 2004.

[Eco] EcoreTools Homepage. `http://www.eclipse.org/ecoretools/`. Visited in June 2014.

[EMFa] Eclipse Modeling Framework - Interview with Ed Merks. `http://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-10027.html`. Visited in June 2014.

[EMFb] Eclipse Modeling Framework Homepage. `http://www.eclipse.org/modeling/emf/`. Visited in June 2014.

[EMP] Eclipse Modeling Project Homepage. `http://www.eclipse.org/modeling/`. Visited in May 2014.

[Eps] Epsilon Generation Language Homepage. `http://www.eclipse.org/epsilon/doc/egl/`. Visited in June 2014.

[ER10] Moritz Eysholdt and Johannes Rupprecht. Migrating a large modeling environment from XML/UML to Xtext/GMF. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*, SPLASH '10, page 97, New York, New York, USA, 2010. ACM Press.

[ETS14a] ETSI. TTCN-3: Core Language Version 4.6.1, 2014.

[ETS14b] ETSI. TTCN-3: TTCN-3 Runtime Interface Version 4.6.1, 2014.

[EvdSV+13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In Martin Erwig, Richard F. Paige, and Eric Wyk, editors, *Software Language Engineering SE - 11*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013.

[Fow04] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2004.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[Goo] Google Guice Homepage. `https://code.google.com/p/google-guice/`. Visited in June 2014.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1st edition, 2009.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, ICSE '11, page 471, New York, New York, USA, 2011. ACM Press.

[Jor13] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. Taylor & Francis, 4th edition, 2013.

[KÖ6] Thomas Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4):369–385, July 2006.

[Ken02] Stuart Kent. Model Driven Engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods SE - 16*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer Berlin Heidelberg, 2002.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, Inc., Hoboken, NJ, USA, February 2008.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lin01] Jürgen Lind. Basic Concepts in Software Engineering. In *Iterative Software Engineering for Multiagent Systems SE - 3*, volume 1994 of *Lecture Notes in Computer Science*, pages 35–95. Springer Berlin Heidelberg, 2001.

[Lud03] Jochen Ludewig. Models in software engineering - an introduction. *Software and Systems Modeling*, 2(1):5–14, 2003.

[MDTa] MDT-UML2-Tool-Compatibility Homepage. `http://wiki.eclipse.org/MDT-UML2-Tool-Compatibility`. Visited in June 2014.

[MDTb] Model Development Tools Homepage. `http://www.eclipse.org/modeling/mdt/`. Visited in June 2014.

[MG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152(0):125–142, 2006.

[MH06] Russ Miles and Kim Hamilton. *Learning UML 2.0*, volume 286. O'Reilly Media, Inc., 2006.

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[Obj03] Object Management Group (OMG). MDA Guide Version 1.0.1, 2003.

[Obj08] Object Management Group (OMG). MOF Model to Text Transformation Language Version 1.0, 2008.

[Obj11a] Object Management Group (OMG). OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.4.1, 2011.

[Obj11b] Object Management Group (OMG). OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4.1, 2011.

[Obj12a] Object Management Group (OMG). Common Object Request Broker Architecture (CORBA) Specification Version 3.3, 2012.

[Obj12b] Object Management Group (OMG). Diagram Definition Version 1.0, 2012.

[Obj12c] Object Management Group (OMG). Object Constraint Language Version 2.3.1, 2012.

[Obj13a] Object Management Group (OMG). OMG Meta Object Facility (MOF) Core Specification Version 2.4.1, 2013.

[Obj13b] Object Management Group (OMG). OMG MOF 2 XMI Mapping Specification Version 2.4.1, 2013.

[Obj13c] Object Management Group (OMG). UML Testing Profile (UTP) Version 1.2, 2013.

[Pap] Papyrus Homepage. `http://www.eclipse.org/papyrus/`. Visited in June 2014.

[PP05] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell.* O'Reilly Media, Inc., 2005.

[Rep09] Tibor Repasi. Software testing - State of the art and current research challanges, 2009.

[RPKP08] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. The Epsilon Generation Language. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture Foundations and Applications SE - 1*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008.

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, Second Edition.* Addison-Wesley Professional, 2008.

[Sch10] Ina Schieferdecker. Test Automation with TTCN-3 - State of the Art and a Future Perspective. In Alexandre Petrenko, Adenilso Simão, and José Carlos Maldonado, editors, *Testing Software and Systems SE - 1*, volume 6435 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2010.

[SDGR03] Ina Schieferdecker, Zhen Ru Dai, Jens Grabowski, and Axel Rennoch. The UML 2.0 Testing Profile and Its Relation to TTCN-3. In Dieter Hogrefe and Anthony Wiles, editors, *Testing of Communicating Systems SE - 7*, volume 2644 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin Heidelberg, 2003.

[Sel03] Bran Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, 2003.

[SGVGD08] Ina Schieferdecker, Jens Grabowski, Theofanis Vassiliou-Gioles, and George Din. The Test Technology TTCN-3. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing SE - 10*, volume 4949 of *Lecture Notes in Computer Science*, pages 292–319. Springer Berlin Heidelberg, 2008.

[SS09] Yashwant Singh and Manu Sood. Model Driven Architecture: A Perspective. In *2009 IEEE International Advance Computing Conference*, pages 1644–1652. IEEE, March 2009.

[SS10] David J. Smith and Kenneth G. L. Simpson. *Safety Critical Systems Handbook: A Straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849*. Elsevier Science, 2010.

[UL07] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.

[Von68] Ludwig Von Bertalanffy. *General system theory: Foundations, development, applications*. Braziller. New York, 1968.

[WH12] Jon Whittle and John Hutchinson. Mismatches between Industry Practice and Teaching of Model-Driven Software Development. In Jörg Kienzle, editor, *Models in Software Engineering SE - 6*, volume 7167 of *Lecture Notes in Computer Science*, pages 40–47. Springer Berlin Heidelberg, 2012.

[WNO12] Flavio Wagner, Francisco A. M. Nascimento, and Marcio F. S. Oliveira. Model-Driven Engineering of Complex Embedded Systems: Concepts and Tools. In *Critical Embedded Systems School (CES-School) at Brazilian Conference on Critical Embedded Systems (CBSEC)*, 2012.

[Xtea] Xtend Documentation. `http://www.eclipse.org/xtend/documentation.html`. Visited in June 2014.

[Xteb] Xtend Homepage. `http://www.eclipse.org/xtend/`. Visited in June 2014.

[Xtec] Xtext Documentation. `http://www.eclipse.org/Xtext/documentation.html`. Visited in June 2014.

[Xted] Xtext Homepage. `http://www.eclipse.org/Xtext/`. Visited in June 2014.

[ZSM11] Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman. *Model-Based Testing for Embedded Systems*. CRC Press, 2011.