The UML 2.0 Testing Profile and Its Relation to TTCN-3

Ina Schieferdecker[1], Zhen Ru Dai[2], Jens Grabowski[2], and Axel Rennoch[1]

[1] Fraunhofer FOKUS, Competence Center for Testing, Interoperability and Performance
Kaiserin-Augusta-Allee 31, D-10589 Berlin
http://www.fokus.fhg.de/tip

[2] University of Lübeck, Institute for Telematics
Ratzeburger Allee 160, D-23538 Lübeck
http://www.itm.mu-luebeck.de

Abstract. UML models focus primarily on the definition of system structure and behaviour, but provide only limited means for describing test objectives and test procedures. However, with the approach towards system engineering with automated code generation, the need for solid conformance testing has increased. In June 2001, an OMG Request For Proposal (RFP) on an UML2.0 Testing Profile (UTP) has been initiated. This RFP solicits proposals for a UML2.0 profile, which enables the specification of tests for structural and behavioural aspects of computational UML models, and which is capable to interoperate with existing test technologies for black box testing. This paper discusses different approaches for testing with UML and discusses the ongoing work of the Testing Profile. Special emphasize is laid on the mapping of UML2.0 testing concepts to the standardized Testing and Test Control Notation (TTCN-3).

1 Introduction

It is well known that the development and implementation of conformance tests is expensive w.r.t. time and money. Several initiatives and efforts have been undertaken to establish an approach to automate - or at least to provide significant support for an automated - test generation. Algorithms have been defined to derive tests from formal system specification given in various notations. Their usage has been demonstrated with sample applications. But today, none of the approaches is widely used in the industrial practise for large applications. One reason may be the difficulty to select the test cases from a (theoretical) unbounded number of tests, which result from test generation algorithms. But often it is simply the lack of a formal system specification of the implementation under test, i.e. the base for the application of the test generation algorithms is missing.

Economic reasons still require computer support for test generation to enhance confidence in software reliability. The traditional distinction between system and test design appears inefficient and is possibly faulty since knowledge transfer between two specifications is needed. Today modern system modelling techniques like UML have an increasing acceptance in the software development community which is much

D. Hogrefe and A. Wiles (Eds.): TestCom 2003, LNCS 2644, pp. 79–94, 2003.

higher than the acceptance of Formal Description Technics in the past (the reason might be that the better tool support nowadays). There is a chance to involve system developers in the test definition process if the modelling language allows to integrate testing related information. Test related information at this early stage means to benefit from the know-how of a system developer and to capture testing related ideas of developers in the test definition process.

In principle, it is possible to start the test derivation with a system model only, i.e. skeletons of the test cases will be generated from the system model. As mentioned before, this process may lead to an unbounded number of tests (e.g. due to an infinite number of test data values). A practical alternative may be the incooperation of test relevant information into the system model, e.g. with annotations provided by the system developer to restrict the number of test cases from the very beginning.

UML technology focuses primarily on the definition of system structure and behaviour and provides limited means for describing test procedures only. With the approach towards system engineering according to model-driven architectures with automated code generation, the need for solid conformance testing, certification and branding has increased. In June 2001, an OMG Request For Proposal (RFP) on an UML Testing Profile has been initiated. It shall provide specification means to define precisely tests for structural (static) and behavioural (dynamic) aspects of systems modelled in UML.

IBM, Ericsson, FOKUS, Motorola, Rational, Softeam and Telelogic formed a consortium to develop that UML Testing Profile (UTP). This profile is based on the concepts of the upcoming version of UML, UML2.0, which is still an ongoing work. The work of UTP is based on recent developments in testing such as TTCN-3 and COTE. It provides mappings to established test environments such as JUnit and TTCN-3.

This paper presents different approaches to UML based testing and discusses the need for test specifications in UML (Section 2). Section 3 presents the recent work on the UML testing profile. In Section 4, the relationship of UTP and TTCN-3 concepts is discussed. In addition, a possible way of mapping UTP specifications to TTCN-3 definitions is presented. The paper concludes with an outlook of the UTP work.

2 Different Approaches on Testing with UML

Several approaches for the integration of testing related information in system developments with UML exist. According to the classical test methodology, two different aspects have to be distinguished: the modelling of system related features (i.e. of the system under test, the SUT), and the definition of the test model features (i.e. of the test system).

In the following, different testing frameworks and practical approaches based on UML systems and tools are discussed. One approach is called *integration testing*. It adopts the UML syntax and uses statechart diagrams for the objects under test. The aim of integration testing is to minimize testing costs, time and effort, i.e. to initially develop customized test drivers, test stubs and test cases and to adapt and rerun them repeatedly for regression testing purposes at each level of integration. Tool support is available e.g. for applications built with the UML modelling tools in Rational Rose.

Other initiatives have developed *UML based test notations*. In , the use of UML to support test development has been investigated to encourage the parallel development of a conformance test suite and a standard system specification. The presented guidelines have been given in the context of TTCN as the target test notation. The suggested test development activities adopt a straight forward approach, i.e. the identification of the independent system components is followed by a definition of the test configuration, test case structure and test cases. UML component or deployment diagrams can be used to represent the test configuration, e.g. in a distributed environment system parts are represented by UML components, points of control and observation (PCOs) by UML interfaces. The test suite structure can be defined in UML with class diagrams whereas any hierarchy of possible test (sub)groups is expressed by a nested class structure. The test behaviour of a test case can be defined using various UML features: UML interaction diagrams, i.e. sequence and collaboration diagrams, and state transition diagrams.

JUnit[19] is a framework for automated unit tests based on Java. Because of its simplicity, JUnit has become popular for *extreme programming* where permanent code integration and code testing are required. JUnit provides an own graphical user interface. A JUnit test defines a testsuite which is composed of several test cases or test suites. A test case contains test methods, a *setup()* method, a *teardown()* method, a *main* method which runs the testcase and an optional *suite()* method which groups test methods in a test suite. At the end of a test run, JUnit reports a *pass, failure* or *error* as its test result. Black box testing can be realized by defining private or protected test methods. JUnit has already been integrated in many case tools, e.g. JBuilder. From a UML model, skeletons for unit tests of individual classes or packages can be recursively generated, including the tested classes and packages. However, the hard part, namely the coding of the dynamic part of the testcases, is not addressed.

The consequent application of UML for the specification of a component-based SUT leads to an interesting approach for the derivation of a *component-based test system*. In , a Test Framework (TFW) is proposed which contains base types of test components. The test purpose independent (i.e. generic) behaviour for these components has been predefined. It covers setup and configuration of a test system, initiation of tests, exchange of coordination messages between test components and collection of test results. The full test system itself is built from the test components. For both test system and SUT the technology independent UML method has been selected e.g. to allow the test system to share the same static information with the SUT. The TFW builds the final test system from a generic test system (GTS) which has one test manager as the main components, one front end, one Main Test Component (MTC) and a set of parallel TCs (PTC). Similar to the Conformance Testing Methodology and Framework (CTMF) , the system behaviour of the GTS comprises test preamble (configuration establishment and test initiation), test body (generic test case) and test postamble (test end and configuration release), whereas the configuration may be hold for several tests in sequence. A user defined test system (UTS) will inherit from the GTS components MTC and TCs and will be refined according to the selected test purposes. During these refinements, both the test case independent and dependent behaviour have to be distinguished. The generic behaviour may be overloaded to locate references to e.g. naming service, timer management, SUT specific initial objects or UTS specific object repositories, or assign a particular order on the PTC

start, etc. The testcase dependent behaviour can be defined with sequence diagrams to provide e.g. sequential request/reply pairs at several interfaces. Multiple instances of these behavioural definitions may be used to describe performance tests, too. It has been proposed to define other (alternative) behaviour with separate sequence diagrams (on default events, timeouts, unexpected behaviour to be ignored). An association of such diagrams to the test case dependent behaviour may be possible with the introduction of activity diagrams.

In most cases, the automation of test generation processes is based on a sequence of different tool applications. The tools perform individual tasks within a sequence of manual or (semi-) automatic transformations (e.g. compilation) of refinements (e.g. extractions) in a step-wise approach. An UML based test tool chain has been proposed by the AGEDIS project. The approach is characterized by the goal to reuse/adopt existing system validation and test code generation tools. It starts with a standard UML system model managed by usual UML tools. Further processing of the UML model results (via XML) in the Intermediate Format (IF) which has been defined and chosen to describe the system model in a state machine manner, and to have a suitable input document to existing model checking and test suite generation tools. The resulting Abstract Test Suite (ATS) is provided by the Test Generation with Verification (TGV) tool in the standardized TTCN format. This allows the application of TTCN tools from the telecommunication industry to produce executable test cases in the desired target API language (e.g. C, C++, Java).

The approaches presented above show that different academic and industrial initiatives have already been undertaken to test on the basis of UML. It appears reasonable that testing becomes an issue within UML itself.

3 The UML2.0 Testing Profile

In this section, the ongoing work for the testing profile is presented. The profile is developed in several steps: the concept space combining various testing areas is defined, a metamodel aligned with the UML 2.0 meta-model is developed, an the applicability of the concepts is analyzed via example and a mapping to existing test infrastructures demonstrates the practical use.

The work on UTP was – besides other sources - based on as the only standardized test notation TTCN-3. The initial intent to base the UML Testing Profile on the Graphical Format of TTCN-3 could not be taken directly since additional requirements from software testing together with the alignment with UML required additions and generalizations. Major generalizations in UTP are:

- the separation of test behaviour and test evaluation by introducing a new test component: the *arbiter*. This enables the easy reuse of test behaviour for other testing kinds without changing the test behaviour but just the arbiter. This concept is comparable to the evaluate function of TSSL.
- the integration of the concepts of test control, test group and test case into just *one concept of a test case*, which can be decomposed into several lower level test cases. This enables the easy reuse of test case definitions in various hierarchies. A test suite is then just a top-level test case. This concept is comparable to the test object concept of TSSL.

- the support of *data partitions* not only for observations, but also for stimuli. This allows to describe test cases logically without having the need to define the stimulus data completely but as a set or range of values. This concept is comparable to the concept of Test Data Definitions (TDD) of ADL.

Furthermore, some additions ease the practical use of the UML Testing Profile:

- an *initial test configuration* is used to describe the setup of the test components and the connectivity to the SUT and between each other
- component and deployment diagrams are used to enable the definition of software components realizing a test suite and to describe the requirements regarding test execution on certain nodes in a network.

The different background of the UTP members has led to an intensive discussion on the basic set of terms as a number of topics allow alternative views. The result is explained by describing the actual terminology. It has been agreed to distinguish three major groups of terms:

- *test architecture*, i.e. the elements and their relationship which are involved in a test,
- *test data*, i.e. the structures and meaning of values to be processed in a test, and
- *test behaviour*, which address the observations and activities during a test.

3.1 Test Architecture

The test architecture sub package covers the concepts for specifying test components, the interfaces of and connections between test components and to the SUT. Test components are active entities within the test system which perform the test behavior defined in a test case (see Test Behavior sub package) by using test data as defined in the Test Data sub package.

The *test architecture* is a set of related classes and/or components from which test case specific configurations may be specified. A *test context* groups test cases with the same initial test configuration. The *test configuration* is a collection of parts representing test components and the SUT and the connections between the test components and to the SUT. The test configuration defines both (1) test components and connections when a test case is started and (2) the maximal number of test components and connections during the test execution. A *test component* is an active object within a test system performing a test scenario. A test component has a set of interfaces via which it may communicate with other test components or with the SUT when the respective interfaces are connected. An *arbiter* is a specific test component to evaluate test results and to assign the overall verdict of a test case. There is a default arbiter for functional, conformance testing, which generates *pass*, *fail*, *inconc*, and *error* as verdict, where these verdicts are ordered as pass < inconc < fail < error. In addition to test components, utility parts can be used to denote helper and miscellaneous parts to realize a test system, e.g. to contain additional data to be used during testing.

An *interface* is a specification of a set of possible operations/messages which a client may request of/send to a test component or to the SUT or which a server may receive from a test component or SUT. An interface is either procedure-based or message-based. A *connection* is a communication path between two interfaces.

The *system under test (SUT)* is characterised by the set of interfaces via which a real SUT can be controlled and observed during testing. An SUT can be on different abstraction levels: a complete systems, a subsystem, a single component, object or even a class.

An example is given for a bank automaton - an ATM (Fig. 1). The bank automaton offers various interfaces, in particular, a port to the bank network and interfaces to the user to insert and withdraw a bankcard as well as to take the money. The test objective is to check that it is possible to debit the account provided that enough funds are available. The package `ATMTest` imports the definition from the `ATM` SUT and defines the test suite `ATMtestsuite` as well as the classes for the test components `HWEmulator` and `BankEmulator`. The test suite is used to define one test case `validWithdrawal()`. `authorizeCard()` is an auxillary operation used within the test case.

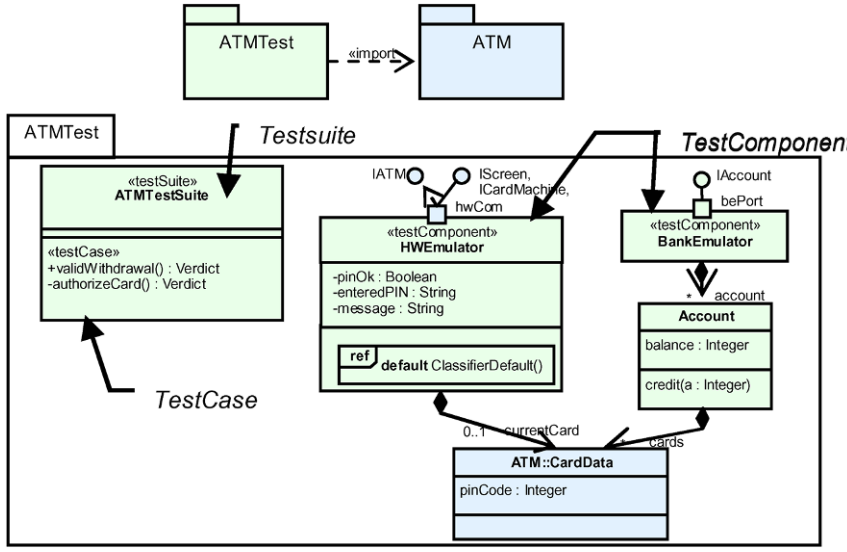

Fig. 1. A UML package containing a test suite for the ATM example

In addition, the test configuration as the internal structure of test suite is given. The test behavior is assigned to the test case and is invoked when the test case is invoked. The test suite uses two test components (Fig. 2Fig. 2): a bank emulator `be` and a hardware emulator `hw`. A utility part `current` represents the bankcard used during the tests. The test components are connected with the SUT via the interfaces `atmPort` and `netCom`.

3.2 Test Data

The test data sub package covers the concepts for data sent to the SUT and received from the SUT. Mechanisms in order to change and compare test data are used to enable precise and succinct test specifications. Data can be concrete (i.e. a specific value) or abstract (i.e. a logically described set of values). Logical partitions are used

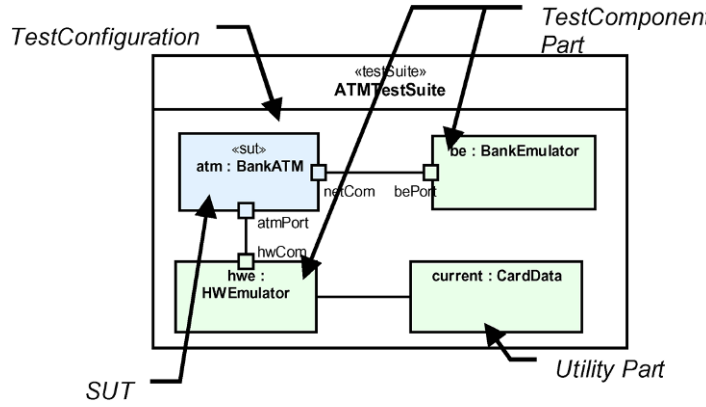

Fig. 2. The test configuration for the ATM example

to define such value sets within test parameters. Coding rules are part of the test specification and denote the encoding and decoding of test data. By means of coding rules, the interfaces of the SUT can be bound to certain encodings such as for CORBA GIOP/IIOP, IDL, ASN.1 PER or XML.

In the ATM example, different messages to and from the SUT are used. They are declared in the class diagram of the ATM such as

```
messageDisplay(in Message:string)
```

In the test behaviour, concrete data is used for example

```
messageDisplay("EnterPIN")
```

Another example is of a data declaration is

```
constant Integer amount {findAccount(current).balance > amount }
```

where `amount` is characterized by the constraint contained in parenthesis, i.e. it has to be less then the balance of the account belonging to the current bankcard.

3.3 Test Behavior

A *test case* is a specification of one case to test the system, including what to test with which input, result, and under which conditions. It uses a concrete technical specification of how the SUT should be tested - the test behaviour. A test case is the implementation of a test objective for a particular test configuration, which is defined by the test behaviour. A test case uses an arbiter to evaluate the outcome of its test behaviour. A test objective is a general description of what should be tested. The test behaviour is the specification of behaviour performed on a given test configuration, i.e. sequences, alternatives, loops and defaults of stimuli to and observations from the SUT. Test behaviours can be defined by any behavioural diagram of UML 2.0, i.e. as interaction diagrams or state machines. There can be a designated main test behaviour for a given test configuration. By invocation, test cases can make use of other test behaviours.

A *verdict* is the outcome of a test case being *pass*, *fail*, *inconc*, or *error* as defined in TTCN-3. Additional verdict information can be used to denote specific test outcomes e.g. for performance tests. Every test component handles a local verdict. Verdict updates are reported to the arbiter for calculation of the overall verdict of the test case. Different schemes to realize an arbiter and the coordination with the test components exist.

A *validation action* is an action to evaluate the status of the execution of a test scenario by assessing the SUT observations and/or additional characteristics/parameters of the SUT. A validation action is performed by a test component and sets the local verdict of that component.

Defaults can be defined on three levels: individually for events in interaction diagrams or for states in state machines, for test components of a specific class or for all test components in a test system, i.e. the *basedefault()*. These defaults are evaluated in sequence – from the event default up to the basedefault.

During the execution of a test case a *test trace* is generated. It contains logs for each action performed during that test case execution and the test result of that test case execution. A log action can be used to store additional information in the test trace.

Fig. 3 (left side) depicts the ATM test case *ValidWithdrawal()*.The objective of the test is to verify that if a user inserts and authorizes a valid card correctly, he is able to withdraw money if he has sufficient funds, i.e. the test case defines a test for a valid withdrawal of money: after authorization of the bankcard (by referencing to the *authorizeCard* operation) the *withdrawal* operation is selected and an *amount* requested, which is smaller than the balance of the account related to the bankcard. This is defined by a logical partition with a constraint on amount (see top on the left side of Fig. 3). The SUT then interacts with the bank emulator *be* to debit the account and delivers the money afterwards. An event specific default *DisplayDefault* is used for the display event in order to handle different display messages specifically. The default behavior is depicted by means of a note notation. The specification of the default is shown in Fig. 3 (right side). Finally, the verdict is set to *pass*.

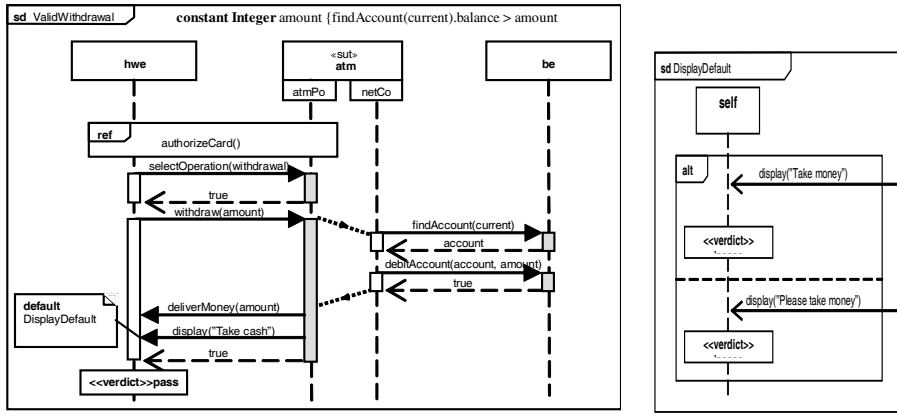

Fig. 3. Test case ValidWithdrawal and Default DisplayDefault

4 The Relation to TTCN-3

Since there was no accepted test notation in UML yet, the UTP request for proposal was an ideal opportunity to bring TTCN-3 in form of GFT to the attention of the UML world. In fact, GFT is the archetype for UTP. UTP uses several concepts being developed in GFT. Still, GFT and UTP differ in several respects: UTP is based on the object oriented paradigm of UML where behaviours are bound to objects only, while GFT is based on the TTCN-3 concept of functions and binding of functions to test components. UTP uses additional diagrams to define e.g. the test architecture, test configuration and test deployment. Test behaviour can be defined as interaction diagrams but also as state machines. While GFT supports dynamic configurations in terms of kind and number of test components and the connectivity to the SUT and between test components[1], UTP uses static configurations where only the number of test components may vary but not the structure of the connections between test components. In addition, UTP has only one FIFO queue per test component, while GFT uses a FIFO queue per test component port.

New concepts in UTP are the arbiter, the validation action, the test trace and the logical partition. According to its definition, an arbiter is a special test component which is responsible for assigning verdicts. Therefore, an arbiter can easily be defined as a test component in TTCN-3 which is created at the beginning of a test-run. In order to overrule the verdict mechanism of TTCN-3, a special verdict type has to be used in addition. Validation action can be realized with external functions. The logical partition of test data for test stimuli is not supported in TTCN-3. Here, a data generation function (as an external function) has to be used in order to select a specific value from that logical partition to be sent to the SUT. A test trace is not specifically part of the TTCN-3 concepts, but can be considered as a test case with just a single sequential execution and, therefore, requires the same specification concepts as test cases.

The verdict handling in GFT is bound to the well-established verdict handling of conformance testing, while UTP uses in addition the ability of user-defined verdicts and the arbitration of verdicts, i.e. the definition of algorithms of when and how verdicts are determined. Additional validation actions can be used to calculate local verdicts of test components by the use of external information from the test execution suite.

Another difference is that of default handling for unexpected or irrelevant behaviour from the SUT: GFT uses function-based defaults which can be dynamically activated and deactivated during test execution, while UTP uses structural defaults, which are bound to the structure of a test system – from test component level down to event/state level – leading to a defaults hierarchy and less dynamic default handling.

UTP supports UML data only, i.e. primitive types (Boolean, String, Integer) and classes, while GFT supports all types available in TTCN-3 such as basic types (integer, char, universal char, float, Boolean, objid, verdicttype), basic string types (bitstring, hexstring, octetstring, charstring, universal charstring), user-defined structured types (record, record of, set, set of, enumerated, union) and anytype. In addition, any imported data like ASN.1 or IDL is supported.

---

[1] In TTCN-3 and hence in GFT, ports can even be connected, reconnected, started, stopped and cleared during test execution, which leads to dynamic test configurations in terms of connectivity between test components and to the SUT.

Table 1. Relation of UTP and TTCN-3 concepts and the principal translation from UTP to TTCN-3

UML Testing Profile	**TTCN-3**
Test Architecture:	
Package	Module
Test Suite	Group covering all test cases of a test suite Having a specific TSI component type (to access the SUT) Having a specific behavioral function to set up the initial test configuration for this test suite
System Under Test (SUT)	The test system accesses the SUT via the abstract test system interfaces (TSI). The SUT interfaces result in port types used by TSI One additional port is needed to communicate with a user-defined arbiter Potentially additional ports are needed to coordinate/synchronize test components
Interfaces	Port types
Test Components	Component types
Test Configurations	Configuration operations create, start, connect, disconnect, map, unmap, running and done for dynamic test configurations. Behavioral function to set up the initial test configuration
Arbiter	The UTP default arbiter is a TTCN-3 built-in User-defined arbiters are realized by the MTC
Test Data:	
Test Parameter, Test Argument	(Inline) templates are used for both test stimulus and test observations
Coding rules	Encode attribute
Test Behaviour:	
Test Case	Testcase
Test Objective	Not part of TTCN-3, just a comment to a test case definition
Test Behaviour	Functions generated via mapping functions per behavior feature of a test suite Test case behavior resulting from creating test components and starting their behavior, MTC just as a „controller“ which also controls the arbiter
Test Trace	Not part of TTCN-3, but could be mapped just as a strict sequential behavioral function
Stimulus	Sending messages and Calling operations Replying to operation invocations (however, raising exceptions is not yet well handled in UML 2.0)
Observation	Rreceiving messages, operation invocations, and operation replies (however, catching exceptions is not yet well handled in UML 2.0)
Default	Altstep and activation/deactivation of the altsteps along the default hierarchy
Coordination	Message exchange between test components.
Verdict	The default arbiter and its verdict handling is an integral part of TTCN-3 For user-defined, a special verdict type and updating the arbiter with set verdicts is needed
Validation Action	External function or data functions resulting in a value of the specific verdict type
Log Action	Log operation

Last but not least, GFT and UTP are on different levels of abstractions: GFT (being part of TTCN-3) is on a detailed test case specification level (i.e. on a level from which executable tests can directly be derived). However, UTP can also be used on more abstract levels by defining just the principal constituents of e.g. a test purpose or of a test case without giving all the details needed to execute the tests. While this is of great advantage in the test design process, additional means have to be taken in order to generate executable tests. For example, the expressiveness of UML 2.0 sequence diagrams allows to describe a whole set of test cases by just one diagram, so that test generation methods have to be applied in order to derive these tests from the diagram.

Overall, UTP is targeted at UML providing selected extensions to the features of GFT/TTCN-3 as well as restricting/omitting other TTCN-3 features. Table 1 compares the UML 2.0 testing profile concepts with existing TTCN-3 testing concepts. All UML Testing Profile concepts have direct correspondence or can be mapped to TTCN-3 testing concepts. A mapping from UTP to TTCN-3 is possible but not the other way around. The principal approach for the mapping to TTCN-3 consists of two major steps: (1) take UTP stereotypes and associations and assign them to TTCN-3 concepts and (2) define procedures how to collect required information for the TTCN-3 modules to be generated.

In the following, an example mapping[2] is provided for the Bank ATM case study described in the previous section (test case in Fig. 3). Two TTCN-3 modules are generated: one for ATM being the SUT (and being defined in a separate UML package) and another module for the ATM test architecture defining the tests for the Bank ATM also in a separate UML package. The module ATM provides all the signatures available at the SUT interfaces, which are used during testing.

```
module ATM {
    //withdraw(amount : Integer): Boolean
    signature withdraw(integer amount) return boolean;
    //isPinCorrect(c : Integer) : Boolean
    signature isPinCorrect(integer c) return boolean;
    //selectOperation(op : OpKind) : Boolean
    signature selectOperation(OpKind op) return boolean;
    … // and so on
}
```

The module for the ATM test architecture `ATMTestArchitecture` imports all the definitions from the ATM module, defines the group for the ATM test suite, provides port and component type definitions within the group, the function to set up the initial test configuration and finally the test cases. In order to make this mapping more compelling, a user-defined arbiter is assumed in addition and the default handling is made explicitly.

```
module ATMTestArchitecture {
    import from ATM all;
    // utility Account type
    type record Account {
                    integer balance,
                    charstring number
    }
    //credit(a : Integer)
    signature credit(integer a);
```

[2] Please note that UTP will provide an example mapping only as there are several ways to map to TTCN-3. It is not the intend to restrict the mappings to a single one, but rather to show the principles and to leave options for the implementers.

```
    //debit(a : Integer)
    signature debit(integer a);
    // utility accnts : Account [0..*]
    external const Account accnts[0..infinity];
    group ATMSuite {
      … // all the definitions constituting the tests for ATM
    } // group ATMSuite
} // module ATMTestArchitecture
```

The required and provided interfaces are reflected in corresponding port definitions `atmPort_PType` and `netCom_PType`, which are then used in the component type definitions `BankEmulator_CType` and `HWEmulator_CType` to constitute the component types for the PTCs:

```
//required interfaces: IScreen, ICardMachine, IMoneyBox
//provided interface: IATM
type port atmPort_PType procedure {
    in display_; //Iscreen
    in ejectCard; //ICardMachine
    in deliverMoney; //IMoneyBox
    in getStatus; // status information
    out withdraw, isPinCorrect,
          selectOperation, storeCardData; //IATM
    out enterPIN; //to give a PIN
}
//required interface: IAccount
//no provided interface
type port netCom_PType procedure {
    in debitAccount, findAccount //IAccount
}
// test component type BankEmulator
type component BankEmulator_CType {
  port netCom_PType bePort;
  port Arbiter_PType arbiter; // user defined arbiter
}
// test component type HWEmulator
type component HWEmulator_CType {
  port atmPort_PType hwCom;
  port Arbiter_PType arbiter; // user defined arbiter
}
```

The following shows the mapping for a user-defined arbiter. A specific type `MyVerdict_Type` together with an arbitration function `Arbitration` is used to calculate the overall verdict during test case execution. The final assessment is given by mapping the user-defined verdicts to the TTCN-3 verdict at the end. This enables e.g. the use of statistical verdicts where e.g. 5% failures lead to fail but less failures to pass. The arbiter is realized by the MTC. It receives verdict update information via a separate port `arbiter`. The arbitrated verdict is stored in a local variable `mv`.

```
//the arbitration
type enumerated MyVerdict_Type {
    pass_, fail_, inconc_, none_
}
type port Arbiter_PType message {
    inout MyVerdict_Type
}
// the MTC is just a controller
type component MTC_CType {
    port Arbiter_PType arbiter; // user defined arbiter
    var MyVerdict_Type mv:= none_;
}
```

```
function Arbitration
(BankEmulator_CType be, HWEmulator_CType hwe)
runs on MTC_CType {
    while (be.running or hwe.running) {
        alt {
          [] arbiter.receive(none_) {…}
            [] …}
        }
    }
    if (mv == pass_) { setverdict(pass) }
    else …
}
```

The defaults in the defaults hierarchy are mapped to several altsteps, which will be invoked later along that hierarchy. In this example, an altstep for every component type is defined, i.e. `HWEmulator_classifierdefault` and `BankEmulator_classifierdefault`.

```
altstep HWEmulator_classifierdefault()
runs on HWEmulator_CType {
    var charstring s;
  [] hwCom.getcall(getStatus:{}) {
      hwCom.reply(getStatus:{} value true);}
  [] hwCom.getcall(ejectCard:{}) {arbiter.send(fail_);}
  [] hwCom.getcall(display_:{?}) -> param (s) {
             if (s == "Connection lost") {
arbiter.send(inconc_) } else {arbiter.send(fail_)} }
}
altstep BankEmulator_classifierdefault()
runs on BankEmulator_CType {
 … }
```

The component type for the test system interface `SUT_CType` is constituted by the ports `netCom` and `atmPort` used during testing in the specific test suite. A configuration function `ATMSuite_Configuration` sets up the initial test configuration and is invoked at first by every test case of that test suite.

```
// SUT
type component SUT_CType {
  port netCom_PType netCom;
  port atmPort_PType atmPort;
}

// setup the configuration
function ATMSuite_Configuration
( in SUT_CType theSUT, in MTC_CType theMTC, inout
    BankEmulator_CType be, inout HWEmulator_CType hwe)
{
 be:=BankEmulator_CType.create;
 map(theSUT:netCom,be:bePort); //map to the SUT
 hwe:=HWEmulator_CType.create;
 map(theSUT:atmPort,hwe:hwCom); //map to the SUT
 connect(theMTC:arbiter,be:arbiter); // arbitration
 connect(theMTC:arbiter,hwe:arbiter); // arbitration
}
```

The `validWithdrawal` test case uses two PTCs `hwe` and `be` each having its own test behaviour, which is defined by behavioural functions `validWithdrawal_hwe` and `validWithdrawal_be` as shown below.

```
function validWithdrawal_hwe()
runs on HWEmulator_CType {
    activate(HWEmulator_default());
    activate(HWEmulator_classifierdefault());
    authorizeCard_hwe();
    hwCom.call(selectOperation:{withdrawal}) {
      [] hwCom.getreply(selectOperation:{?} value true)
           {}
    }
    if (amount <= accnt.balance) {
       hwCom.call(withdraw:{amount},nowait);
       hwCom.getcall(deliverMoney: {amount});
       hwCom.getcall(display_:{"Take cash"});
       hwCom.getreply(withdraw:{?} value true);
    }
    else {
       log("not enough money on the card
              to withdraw amount");
       setverdict(inconc);
    }
    setverdict(pass);
}

function validWithdrawal_be()
  runs on BankEmulator_CType {
    activate(BankEmulator_default());
    bePort.getcall(findAccount: {current});
    bePort.reply(findAccount: {current} value accnt);
    bePort.getcall(debitAccount: {accnt,amount});
    bePort.reply(debitAccount: {accnt,amount}
      value true);
    setverdict(pass);
}
```

Finally, the test case can be provided. According to the initial test configuration, two PTCs `hwe` and `be` are used. The configuration is set up with `ATMSuite_Configuration`. The test behaviour on the PTCs is started with `validWithdrawal_hwe` and `validWithdrawal_be`. The arbiter `Arbitration(be,hwe)` controls the correct termination of the test case. This completes the mapping.

```
//+ validWithdrawal() : Verdict
  testcase validWithdrawal_test()
  runs on MTC_CType system SUT_CType {
      var HWEmulator_CType hwe;
      var BankEmulator_CType be; // initial configuration
      ATMSuite_Configuration(system,mtc,be,hwe);
      hwe.start(validWithdrawal_hwe());
      be.start(validWithdrawal_be());
      Arbitration(be,hwe);
  }
```

5 Outlook

UML has been discovered by both software engineers and test developers to specify system and test models in a platform independent manner. With an integrated approach of developing a system and its tests within one framework, tests can be developed more efficiently and economically. Special attention has been given to the OMG's initiative on defining a UML testing profile. It supports independent test laboratories in their work but also the system engineers to perform the test runs by their own.

The status of the basic test concepts and terminology which have been presented in this paper can be regarded as a consensus of different R&D scientists and engineers working in heterogeneous IT fields like object-oriented systems or telecom protocols specifically on testing aspects. Fundamental elements of the UML testing profile's test architecture, test data and test behaviour have been collected and applied exemplarily using UML related class and sequence diagrams. A comparison with the established concepts of TTCN-3 confirms the suitability of the selected definitions in the UML testing profile. The UML testing profile elements can be mapped to TTCN-3 but not vice versa. This mapping allows to base implementations of the UML testing profile on top of TTCN-3 test environments.

At the time of writing this contribution, the work on the UML profile for testing is still ongoing in OMG and no final decisions have been made on the UML extension mechanisms, i.e. stereotypes, constraints or tagged values, selected for the different testing concepts. Due to the dependencies on the UML2.0 release, which is expected only in 2003, it is expected that the final submission of the UML profile for testing will be available mid to end of 2003. Nevertheless the importance of testing with UML has to be elaborated earlier to assist its acceptance.

Acknowledgements

The authors thank the UTP consortium and supporters for the joint work and discussions. Particular thanks go to Paul Baker, Oystein Haugen, Serge Lucio, Johan Nordin, Eric Samuelson and Clay Williams.

References

1. J. Hartmann et al.: UML-Based Integration Testing. ISSTA´00. Portland, Oregon.
2. ETSI: Methods for Testing and Specifications (MTS); Methodological approach to the use of object-orientation in the standards making process. ETSI EG 201 872 (August 2001). Sophia Antipolis (F).
3. ISO/IEC 9646-3: Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework (CTMF) - Part 3: The Tree and Tabular Combined Notation (TTCN), edition 2, Dec. 1997.
4. M. Born et al.: Test Framework for Component-Based Systems. ICDCS' 2000 & DS-VV'2000, Taipei (Taiwan), April 2000.
5. ISO/IEC 9646: Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework (CTMF).
6. C. Crichton et al.: Using UML for Automatic Test Generation: ASE'2001.
7. A. Cavarra et al.: AGEDIS Language Specification. Project Deliverable 2.2. The AGEDIS project, 2001, http://www.agedis.de.
8. L. Clark et al.: Achieving Cross-Platform Compatibility with Increased Productivity and Quality using the OMG's Model Driven Architecture. Lockheed Martin Corporation, 2001.
9. ETSI ES 201 873-1: The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. V2.1.0 (2001-10), 2001; also an ITU-T standard Z.140.
10. ETSI DES 201 873-3 V2.0.0: The Testing and Test Control Notation version 3; Part3: Graphical Presentation Format for TTCN-3 (GFT). V2.0.0 (2001-11), 2001.

11. J.-C. Fernandez et al.: An experiment in automatic generation of test suites for protocols with verification technology. Science of Computer Programming, 1997. http://citeseer.nj.new.com/2326.html.
12. C. Jard, S. Pickin: COTE – Component Testing using the Unified Modelling Language. - ERCIM News No.48, January 2002.
13. T. Vassiliou-Gioles et al.: Configuration and Execution Support for Distributed Systems.- IWTCS'99, Budapest, Hungary, Sept. 1999.
14. E. Rudolph, J. Grabowski, and P. Graubmann. Towards a Harmonization of UML-Sequence Diagrams and MSC. In R. Dssouli, G. v. Bochmann, and Y. Lahav, editors, SDL'99 - The next Millenium. Elsevier, June 1999.
15. E. Rudolph, I. Schieferdecker, and J. Grabowski. Development of an MSC/UML Test Format. BT'2000 - Formale Beschreibungstechniken für verteilte Systeme. Shaker Verlag, Aachen, June 2000.
16. R. Soley: Model Driven Architecture: An Introduction. http://www.omg.org.
17. The Open Group: ADL 2.0 Translation System, 1998. http://adl.opengroup.org/
18. I. Wilie et al.: UML Action Specification Language (ASL) Reference Guide. Kennedy Carter Ltd., Feb. 2001.
19. R. Hightower, N. Lesiecki: Java Tools for eXtreme Programming, Wiley Computer Publishing, 2002.
20. I.Schieferdecker, J. Grabowski: The Graphical Format of TTCN-3 in the context of MSC and UML. Proceedings of the 3rd Workshop of the SDL Forum Society on SDL and MSC (SAM'2002), Aberystwyth (UK), June, 24 - 26, 2002.
21. UML testing profile home page: http://www.fokus.gmd.de/U2TP/