Use of TTCN-3 for Software Module Testing

Andreas Johan Nyberg

Nokia Research Center
P.O. Box 407
FIN-00045 NOKIA GROUP
Andreas.J.Nyberg@nokia.com

Abstract. Efficient testing of software modules remains a challenging task for complex software implementations. TTCN-3 has so far been applied mainly in the telecom domain but not yet in a larger extent to software module testing. This paper describes a multi purpose TTCN-3 test system solution primarily targeted for concurrent software and testing of software modules in isolation. Apart from a test system solution, an approach for type mappings from C to TTCN-3 is discussed, followed by an example of how test cases could be implemented and how the discussed test system be utilized for a simple software module.

Keywords: Software testing, concurrent software, mock objects, TTCN-3

1 Introduction

Software module testing can be a complex task for non-trivial code, especially in concurrent programming. Testing modules of concurrent software in isolation has a number of requirements on the test framework used. The test system and test case implementations must be able to handle concurrent behavior and support the use of emulated components for module isolation. Testing of concurrent software has a number of additional challenges[1],[2] compared to sequential software. These include shared critical areas, synchronization, deadlocks, livelocks[3] and non-deterministic behavior. When testing modules in isolation missing parts of the implementation have to be emulated e.g., components and services.

This paper describes a multi purpose test system targeted for testing of complex software implementations with concurrency aspects and requirements to test modules in isolation. Several testing concepts must be handled by the test system which are treated as requirements: The test system has to provide (1) *test control and test management*, (2) *test suite trace consistency*, (3) handle *non propagating errors* from earlier executed test cases and (4) support free choice of *system under test (SUT) execution environment*. In addition, (5) *language independency* for the implementation under test (IUT) is supported that comes for free with the choice of the test language used in this paper, TTCN-3.

For well known CUnit[5],and JUnit[6] frameworks, software module testing using the native language of the tested software module as test language has its advantages. Among them are easy data type creation and instantiation, conditional tests, and already existing knowledge of the language to write test cases in. Involving another language for test case implementations in the testing process creates an overhead in form of learning another language. Nevertheless using TTCN-3, a standardized test

language for test suite implementations and test systems, has some advantages and might even opens up new possibilities for software testing.

TTCN-3 has evolved from the telecom testing industry and is a test implementation language that looks very much like an ordinary programming language. It is currently the best established standardized testing language on the market. TTCN-3 has been used for testing in telecom related areas, e.g., protocols such as IPv6[7], SIP and WiMax. What these test systems have in common is that they are fully asynchronous and heavily rely on the message-based communication mechanism part of the TTCN-3 language. TTCN-3 supports however in addition also procedure-based communication. The procedure-based paradigm can be applied to cases where there is a clear distinction between a caller and replier of procedures. This makes TTCN-3 suitable for calling remote procedures in a SUT, in the same way as Sun/ONC RPC[RFC 1831] works.

TTCN-3 provides the concept of test components which makes it possible to implement concurrent behavior in a natural synchronous procedure-based manner. TTCN-3 has a built in extension mechanism that enables the import of notations in to a test system. Examples of these extension mechanisms can be found in the standardization work involving the IDL and XML language integrations [13]. As TTCN-3 is independent of the implementation language of the tested code the language has to be mapped and imported to TTCN-3 through the extension mechanism. The mapping has to cover up to the extent of functions, function parameter and return value types. No semantics need to be mapped, only what is publicly visible in interfaces. This paper covers a subset of the C to TTCN-3 mapping which will be a future part of the TTCN-3 standards. The C language has language specific features like pointers allowing for definition of complex types that have not been covered in other TTCN-3 language mappings before.

2 Background

The targeted problems in this paper are to exemplify the possibility to create multi purpose software testing test system based on the TTCN-3 standards. Difficult testing areas such as concurrent software and module isolation with emulated components will be addressed.

Two ways of testing concurrent software is through multiple execution and deterministic execution of test cases[2],[8]. Test case implementation based on methodologies for testing concurrent software such as above and in addition mutation testing[9] and reachability testing[8] will only be addressed in the context of applicability by using TTCN-3. The test system approach in this paper makes it possible to perform multiple and deterministic execution of test cases utilizing testing methodologies that are suitable for concurrent software testing by using the execution semantics of TTCN-3.

Testing software modules in isolation is a difficult task even for non-trivial code. One approach is to use mock objects[4] which are dummy implementations that emulate real implementations to replace functionality in the testing domain of the tested software module. There are many good reasons to use mock objects[10] such as non-deterministic behavior, test object setup and call back functionality. This approach can be utilized efficiently by using TTCN-3 not only for code level mock objects but also for emulating behavior of external services something that will be exemplified in this paper.

TTCN-3 telecom protocol test systems are already in product use, e.g.[7], in addition established testers also for OMG CORBA interfaces are available. Test systems for OMG CORBA implementations based on IDL interfaces require that the interface definitions are mapped to TTCN-3[11]. The test system approach in this

paper requires that the language of the tested software module is mapped to TTCN-3. To be more precise the interface of the module under test has to be represented in the TTCN-3 test suite as is with parameters and type mappings.

3 TTCN-3 Test Systems

Suitable test architectures for testing software modules can be approached from several different angles. Architectures and approaches can differ depending on the stage in the development process or the kind of testing that is looked for. The approach described in this section resembles a unit testing platform which is applicable and targeted for unit testing and higher level testing such as functional testing and system testing. The test language and the test system design allow for efficient test suite implementations by using a mixture of message-based and procedure-based communication. For example, a module can be tested through its available interface at function level and at the same time it can be tested by using message-based communication with external interfaces and mock objects, all from within the same test suite. With multiple test capabilities software faults can be detected at several levels of testing: early on in development using unit testing-like approaches (with high code coverage), integration testing for interface and integration faults as defined in[12], and in system and conformance testing for error detection at a pure black box level.

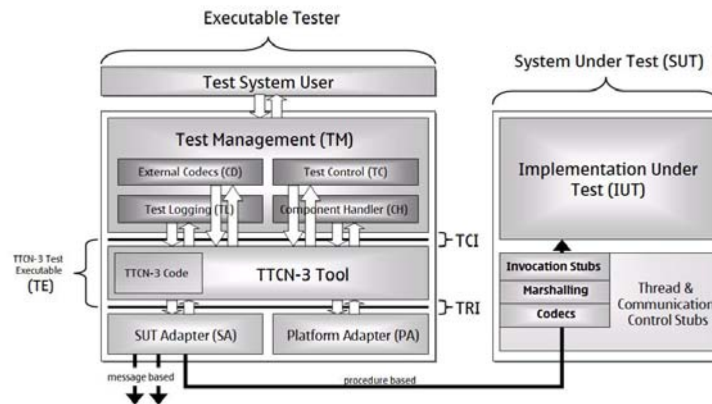

Fig. 1. Overview of a TTCN-3 test system and SUT.

Using TTCN-3 for test suite implementations requires a TTCN-3 test system which contains more than TTCN-3 code[16]. The left part in Figure 1 shows the composition of a TTCN-3 test system as defined in the TTCN-3 standards[13]. There are several different entities involved in the test system which communicate over standardized interfaces named TTCN-3 Control Interface (TCI) and TTCN-3 Runtime Interface (TRI). The most central entity, the TTCN-3 Executable (TE) handles the execution of the TTCN-3 code, either as compiled code or interpreted at runtime. The user interacts through the Test Management (TM) entity which also provides functionality for encoding and decoding of data values. The lower level containing the SUT Adapter (SA) and Platform Adapter (PA) interfaces as the names implies relate to the interaction towards the SUT and towards the test system operating system.

The interaction between the SUT and the SUT Adapter is a design choice for the test system. In the illustrated case a separated Executable Test Suite (ETS) and SUT has been chosen. This is defined as the distributed test method in ITU-T X.290[14].

3.1 IUT and the Test Harness

An IUT cannot be tested as it is. There needs to be a test harness present in the SUT. The IUT together with test harness composes the SUT. As the test system that is discussed in this paper is distributed, the test harness has to be able to handle, communication between the test system and SUT, marshalling, and invocation of the functions in the module under test. Marshalling builds valid C values to use in function invocations. The distributed approach automatically puts requirements on the mapping between C and TTCN-3 in order to make sure that values really can be built from the mapped C value instances in the TTCN-3 test suite. Implementation of the SUT test harness depends on the function interface, programming language of the IUT and available tools for stub generation.

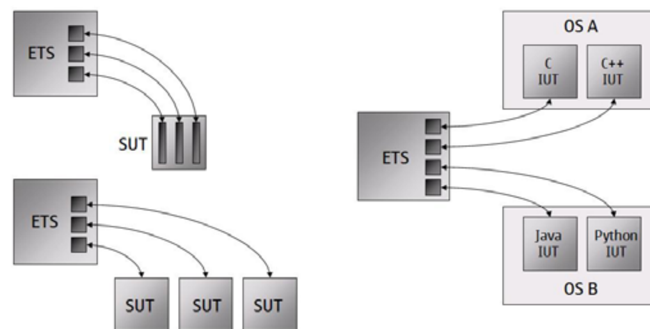

Fig. 2. Examples of distributed test system implementations.

It is up to the SA on the tester side to keep track of the different executing SUTs and its connections to the test executable. This can be managed at TTCN-3 test component level where each executing test component is directly mapped to a single executing SUT or even to a single thread in a SUT. On the left hand side in Fig. 2 software components or SUT threads which run in parallel are mapped to individual TTCN-3 test components. The right hand side in Fig.2 shows an example of the implementation language independency that TTCN-3 can provide.

3.2 TTCN-3 Concepts and Usability for Software Testing

Using TTCN-3 for software testing provides several very useful features. Apart from what has been mentioned already in form of standardized test system interfaces and independency of the language of the IUT there are other very testing specific advantages.

TTCN-3 has very well defined execution semantics that guarantees that a test case execution of a TTCN-3 test system will be the same with any TTCN-3 tool. There is also an internal matching mechanism, that when combined with the TTCN-3 language construct template, allows for the writing of very compact test cases. Using a template construct which represents a value instance, all or selected parts of the value instance can be targeted for matching i.e. when matching two structured types against each other.

Concurrency is something that can be handled very gracefully in TTCN-3 by using test components and a test case verdict that can be set from any test component without any need for component synchronization. The TTCN-3 test components also allow performing of tests on several modules written in different languages, with the same test system - even from within the same test case. Combining the procedural and

message-based aspects with the concept of TTCN-3 test components that can be executed in parallel with individual behavior and complex test case scenarios can be implemented in a clear and concise manner.

3.3 Distributed Test System Main Concepts

Distributing the test system as in Fig. 1 by separating the SUT from the ETS provides capabilities that solve some of the test system requirements stated in the introduction. These capabilities come with a number of benefits that outweigh the drawbacks of a distributed test method, e.g., its communication latency:

- *Test control and test management*

  A potential problem when running large test suites is the possibility to recover and continue a test suite execution in case of a crash, deadlock or livelock in the IUT. With the ETS running in isolation from the IUT the SUT can be restarted by the ETS if needed, e.g. after a crash in the IUT or between each test case.

- *Test suite trace consistency*

  Logs and test suite verdicts and summaries can be kept intact and will not be corrupted. The ETS does not crash from a crash in the IUT and all test execution logs can be finalized gracefully without the risk of loosing valuable test execution information.

- *Non propagating errors*

  Most software testing literature defines that the first step in the unit testing procedure shall be to test the smallest possible unit, e.g. a function, to make sure errors do not propagate into further development. With the separated SUT and ETS the SUT can be restarted for every test case. This usually results in re-initialization of memory through which the propagation of errors could propagate.

- *Language independency*

  Most software unit testing tools are written in the same language as is used for writing the tested software. Using a TTCN-3 based solution the test cases will not be written in the same language as the IUT. In Fig. 2 the implementation language of the IUT is independent of the test suite language, which adds to the goal of a multi purpose test system.

- *SUT execution environment*

  A distributed test system approach allows a free choice in execution environment with regards to operating system and platform of the SUT. The only requirement is that the target system must allow communication between the SUT Adapter in the test system and the SUT test harness.

Test system distribution has advantages, but also has a few drawbacks that have to be considered. Communication overhead is one issue while invocation latency and marshalling are others. The latency has been measured in a comparison between two almost identical test systems. One test system had the IUT directly attached to the SUT adapter and thus execute in the same process as the ETS. The other test system is the one described in this paper. The additional latency of having the SUT separated from the ETS was approximately three to one. Invocation, marshalling and communication latency are not alone the reasons for causing the performance drawback. There is also additional routing functionality that has to be handled in the SUT adaptation due to the possibility to have many separately executing SUTs or

SUT threads. There are a number of possible solutions which can be applied to the distributed test system case for overcoming obstacles such as latency and memory access overhead. One is through the use of inter-process communication (IPC) over UNIX sockets utilizing shared memory between processes running at the same machine which will reduce the time spent on pointer handling in the test cases.

Another drawback is that a model for memory access has to be defined before writing test code for software which is implemented in programming languages that utilize pointers or object references. Such references and pointers only reside in the memory space of the SUT. However, a memory address or object reference has no real meaning inside the test system which is executing in its own separate process. This lack of pointer transparency requires that memory access is explicitly defined in TTCN-3 test cases through the language mappings.

3.4 Testing Multithreaded and Concurrent Software Modules

One of the biggest challenges when testing concurrent software is to make sure that it will be tested thoroughly when the software includes critical areas and shared data. The order of execution of concurrent statements in the different threads or processes must be considered. This non-deterministic behavior can cause execution sequences to be different from execution to execution even with the same input data. This adds synchronization complexity to test case implementations. Synchronization between test components can be solved by connecting test component ports to each other through the TTCN-3 operation *connect* which allows for message communication internal to the TTCN-3 executable. Deterministic execution sequences can then be synchronized utilizing this internal communication. Functional testing of individual functions in isolation does not constitute enough assessment because none of the functions under test will then deal with concurrency. What is needed is a test system that can simultaneously trigger several functions to access critical sections in the IUT at the same time. This problem can be solved by assigning one TTCN-3 test component per thread or software component in the SUT. Here, each test component in the test system creates its own connection to the SUT who in turn spawns a new working thread for every connection and thereby establishes a coupling between the test component and, e.g. a POSIX thread, in a C test harness.

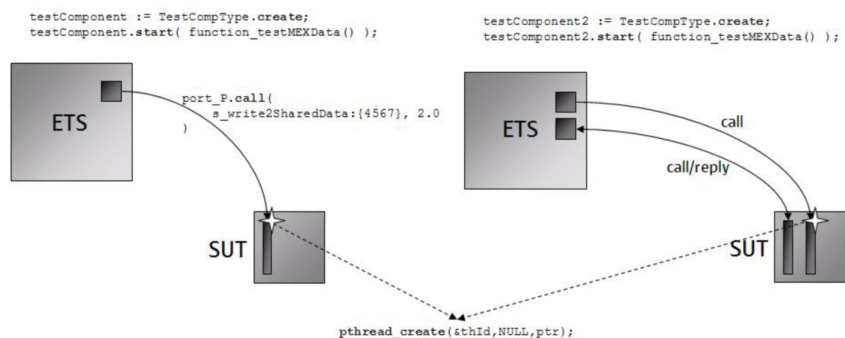

Fig. 3. TTCN-3 test components spawn working threads in the SUT.

All procedure-based calls between a test component and a working thread in the SUT are transmitted via such an established connection. The connection which is not visible from within the test case is a feature of the test system design. The spawned working thread is part of the test harness in the SUT and is utilized for simulating concurrent behavior in the IUT.

3.5 Usage of Mock Objects

When a software module is to be tested as an isolated entity, the test system has to emulate any calls that a particular module makes to other functions or methods, that are external to the tested module. This clearly moves away from pure black box testing, as knowledge and understanding of internal functional details about the module is needed. To properly isolate but still test the module in its intended environment, we need to create test configurations with test components (mock objects) that can act as the external, collaborating components.

By using parallel test components of TTCN-3, we can easily simulate the external collaborating code modules and their behavior by allowing the test components to make or expect calls to and from the tested module. The strength of this approach is that it allows control of the behavior of these external parts in order to simulate correct or incorrect external behavior and then analyze if the tested module is able to handle problematic situations. This means that it will be possible to test error handling code as well.

4 C to TTCN-3 Language Mappings

Specifying test cases in TTCN-3 requires that the tested functions of the software module to test can be called from the TTCN-3 test suite. This is possible provided that a mapping between the language of the software to test and TTCN-3 exists. Utilizing the procedure-based paradigm in TTCN-3 with the concept of signatures a mapping can be defined in a clear and evident way. This applies to all the tested functions, functions in mock objects and to the types used as parameters and return values. This section gives an overview of a subset of the C to TTCN-3 mappings that have been defined.

4.1 A Practical Example

Mapping from the programming language of the IUT is an important part of a working TTCN-3 test system. Selected parts of a C to TTCN-3 mapping is presented and exemplified in this paper. We will use a simple reentrant thread safe C module representing an automated teller machine (ATM) as ongoing example. The module has functions with critical sections.

```
int login   ( int userId, int code );
int balance ( int userId, int* balance );
int withdraw( int userId, int amount, int* balance );
int logout  ();
```

The ATM server example requires the existence of a component that handles user verification. This component in the example is replaced by a mock object to make it possible to test the ATM module in isolation. Below is the sole function of the mock object which is called from the IUT.

```
int verifyUser( int userId, int code );
```

The C module of the ATM can be tested in many ways, primarily through unit and functional testing of functions followed by testing of concurrent scenarios.

The ATM example above is a typical example where a number of problematic scenarios easily can be identified, e.g. read/write locks, starvation and partial failures. Before exemplifying possible test cases for the ATM module the necessary mapping from the language of the IUT (C) to TTCN-3 has to be defined.

4.2 Mapping of Functions

TTCN-3 signatures may specify parameter lists and return values where the direction of the signature parameters is also defined. The interface of the ATM module and mock object function are mapped and exemplified in Table 1.

Table 1. TTCN-3 signatures mapped from a set of the functions in the example.

```
001  /* ATM interface */
002  signature s_login( in CInt p_userId,
003                     in CInt p_code )
004      return CInt;
005  signature s_balance( in CInt p_userId,
006                       in CIntPtr p_balance )
007      return CInt;
008  signature s_withdraw( in CInt p_userId,
009                        in CInt p_amount,
010                        in CIntPtr p_balance )
011      return CInt;
012  signature s_logout  () return CInt;
013
014  /* mock object interface */
015  signature s_verifyUser( in CInt p_userId,
016                          in CInt p_code ) return CInt;
```

All parameters are mapped with the direction set to *in* parameters as if passed by value. This also applies to the case of pointer type parameters since a pointer address cannot change during a function call.

4.3 Mapping of Pointers

Mapping of primitive and built in data types in C is rather straightforward. But pointers need some more attention. A C pointer can been thought of as a four or eight byte aligned value[15] and can therefore be represented as a TTCN-3 integer. The contents pointed at by a pointer on the other hand calls for a more constructive mapping than a simple string representing the memory address of the contents. One possibility is to represent the pointer value by the data type it is pointing at in the test suite. However, in the case of more complex types, e.g. cyclic data structures, this approach fails. Another issue is that a pointer to e.g. an integer may not always reference to a single integer. It can just as well point to the first element of an array of integers of an unknown size. In general, a pointer must always refer to a data array with one or more elements representing the memory space it is pointing at. Therefore the C pointer's *content* has to be mapped to the TTCN-3 list type record of. The pointer itself is still mapped to an integer but when the contents are retrieved they are represented by an ordered list of the pointed at data type. One benefit of this mapping is that pointer arithmetic can be done easily in a test case implementation, e.g. increment and decrement by direct indexing. This mapping is not sensitive regarding the type of the pointer, complex and abstract data types are mapped in the same way as for simple types.

Table 2: Mapping of pointer to integer.

```
C
001  int* balance
```

```
TTCN-3
001  type integer CInt;
002  type integer CIntPtr;
003  type record of CInt CIntArr;
004
005  signature s_MallocCInt( in CInt size ) return CIntPtr;
006  signature s_FreeCInt( in CIntPtr ptr );
007
008  signature s_SetCInt( in CIntPtr ptr, in CIntArr data );
009  signature s_GetCInt( in CIntPtr ptr, in CInt size )
010      return CIntArr;
```

Further functionality in the mapping is needed when working with pointers. A check for null pointers can be enough for a simple test case. What is required are means for allocating memory, assigning data and retrieving data. This functionality is handled through type specific signatures as exemplified in Table 2. The test case writer has to be aware of this mapping due to the nature of the transparent memory access that has to be provided.

For data assignment and retrieval two signatures are sufficient, where the number of elements to retrieve or assign has to be given. At all times a pointer's content is represented by the ordered list type, i.e. the record of type. Dealing with a pointer to a single element is treated as a list containing only one element. For C pointers two TTCN-3 signatures for every type representing the malloc and free calls are enough for complete memory handling.

5 Test Case Implementations

Implementing unit test cases in TTCN-3 is done in a conditional test based manner where return values and pointer parameters are matched in conditional statements individually through implicit templates or through defined templates. If parameters and return values are not enough for evaluation, the test system has to be extended with additional functionality for testing outside a TTCN-3 signature definition, e.g., file streams and connections. Test cases can implement testing functionality resembling the assertion macros and functions of, e.g. the well known testing frameworks CUnit[5] and JUnit[6]. Table 3 shows fragments of a simple unit test case illustrated in Fig. 4. It tests for correct behavior when verifying a user against a mock object that acts as the missing user verification component. The test case in Table 3 tests the case where one user 'known' to the ATM system tries to log in with an incorrect password. It also checks that the login function under test realizes that the password is not correct from the user verification reply. On lines 10 to 11 in Table 3 two test components are created, one for the mock object and one for the function to test. The behavior of the components is implemented in the functions that are passed as arguments when the components are started on lines 13 and 14.

Table 3. The test case body where two components are created and started.

```
TTCN-3
001  testcase TC_UserVerification_Invalid_001()
002      runs on IprPBComp
003      system IpRouterPBTSI {
004
005    // Start two components here. One for the login and one for
006    // the mock object.
007    var IprPBComp testComponent;
008    var IprPBComp mockObject;
```

```
009
010     testComponent := IprPBComp.create;
011     mockObject    := IprPBComp.create;
012
013     mockObject.start( f_verifyUser_mockObject( 0 ) );
014     testComponent.start( f_login_invalid( 1 ) );
015
016     f_waitForComponentsToFinish( c_COMP_TIMEOUT );
017   }
```

The component representing the mock object for the user always returns a failed reply. The behavior is defined in the function in Table 4 where a call from the SUT is accepted (line 10) and the *rejected user* reply is returned (line 11).

Table 4. Code fragments from the mock object function *f_verifyUser_mockObject*.

```
      TTCN-3
001   function f_verifyUser_mockObject( in integer p_compNo )
002     runs on IprPBComp {
003
004     f_initPBTcp( localIp,
005                  localPort+p_compNo,
006                  sutIp,
007                  sutPort );
008
009     // Accept any user and always return invalid.
010     pt_pb.getcall( s_verifyUser:{?,?} );
011     pt_pb.reply( s_verifyUser:{-,-} value c_REJECTED_USER );
012
013     unmap( self:pt_pb, system:pt_pb );
014   }
```

The user verification should fail and the failure shall be detected by the function under test and then later in the test case the test case verdict is set to pass. This is exemplified in Table 5 where the function login in the IUT is called on line 9 and the alternative replies are evaluated on lines 12 to 21.

Table 5. The test purpose functionality that verifies that the function under test can detect a failed login

```
      TTCN-3
001   function f_login_invalid( in integer p_compNo )
002     runs on IprPBComp {
003
004     f_initPBTcp( localIp,
005                  localPort+p_compNo,
006                  sutIp,
007                  sutPort );
008
009     pt_pb.call(
010       s_login:{ c_USERID_A, c_BADSECRETCODE_A }, c_TIMEOUT_SEC ) {
011
012       [] pt_pb.getreply( s_login:? value c_REJECTED_USER ) {
013         setverdict( pass );
014       }
015       [] pt_pb.getreply( s_login:? ) {
016         setverdict( fail );
017       }
018       [] pt_pb.catch( timeout ) {
019         setverdict( fail );
020         stop;
021       }
022     }
023
024     unmap( self:pt_pb, system:pt_pb );
025   }
```

For concurrent software modules the behavior of the threads executing on the IUT has to be controlled by a dedicated TTCN-3 test component. Each test component has to establish a new connection to the SUT which at the point of the accepted connection spawns a working thread in the test harness. This is exemplified in Fig. 3 and in the right part of Fig. 4 which illustrates a test case with mutually exclusive data and multiple accessing threads.

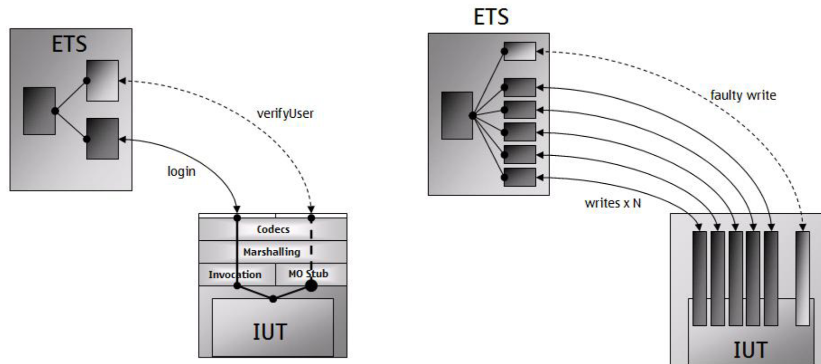

Fig. 4. Test cases visualized.

The purpose of the test case defined in the function in Table 6 is to determine if the ATM module can handle (not crash) and recover from a function call that purposely introduces an invalid parameter (null pointer), Table 7. Recovering means in this case that an invalid parameter shall not halt the ATM module by e.g. keeping mutexes locked. The test case creates a set of components that all but one executes the behavior in Table 6.

Table 6. The test behavior of the test components that correctly calls the withdraw function.

```
    TTCN-3
001 function f_write_valid( in integer p_compNo )
002   runs on IprPBComp {
003
004   f_initPBTcp( localIp,
005                localPort+p_compNo,
006                sutIp,
007                sutPort );
008
009   // Create the integer pointer where to store the account
010   // balance.
011   var CIntPtr v_balance;
012
013   pt_pb.call( s_MallocCInt:{ 1 }, c_TIMEOUT_SEC ) {
014     [] pt_pb.getreply( s_MallocCInt:? value c_NULLPTR ) {
015         setverdict( fail );
016         stop;
017     }
018     [] pt_pb.getreply( s_MallocCInt:? ) -> value v_balance {
019         setverdict( pass );
020     }
021     [] pt_pb.catch( timeout ) {
022         setverdict( fail );
023         stop;
024     }
025   }
026
027   // Continuous withdrawals, no checking.
```

```
028    for ( i:=0; i<c_LOOPS; i:=i+1 ) {
029
030      pt_pb.call(
031          s_withdraw:{ c_USERID_A, 20/*€*/, v_balance },
032          c_TIMEOUT_SEC ) {
033
034          [] pt_pb.getreply( s_withdraw:? value c_WITHDRAW_OK ) {
035              setverdict( pass );
036          }
037          [] pt_pb.getreply( s_withdraw:? value c_WITHDRAW_ERROR) {
038              setverdict( fail );
039              stop;
040          }
041          [] pt_pb.catch( timeout ) {
042              setverdict( fail );
043              stop;
044          }
045      }
046    }
047    unmap( self:pt_pb, system:pt_pb );
048  }
```

In Table 6 the behavior of the correctly executing test components can be seen. A pointer is created (lines 13 to 25) and is used through out the test case execution. A number of withdrawal requests are performed in a loop (lines 28 to 45) to keep the server busy and to determine if it can recover when another test component introduces an error. All components include detection of timeouts which will indicate that the IUT has crashed or got locked by a deadlock or livelock (Table 6 line 14 and Table 7 line 25).

The function in Table 7 is executed by the component that introduces a possible problem by passing a null pointer to the function withdraw (line 15). The correct behavior of the IUT shall at this point be that the null pointer is detected, any locked mutexes are released and an error is returned. Test case verdict according to how the IUT behaves is set on lines 18 to 28.

Table 7. Code fragments of the test component that introduces an error by passing a null.

```
     TTCN-3
001  function f_write_invalid( in integer p_compNo )
002    runs on IprPBComp {
003
004    f_initPBTcp( localIp,
005                 localPort+p_compNo,
006                 sutIp,
007                 sutPort );
008
009    // Introduce the error after a few withdrawals have been done
010    // already.
011    f_sleep( 1.0 );
012
013    // Withdrawal with NULL pointer.
014    pt_pb.call(
015      s_withdraw:{ c_USERID_B, 20/*€*/, c_NULLPTR },
016      c_TIMEOUT_SEC ) {
017
018      [] pt_pb.getreply( s_withdraw:? value c_WITHDRAW_OK ) {
019        setverdict( fail );
020      }
021      [] pt_pb.getreply( s_withdraw:? value c_WITHDRAW_ERROR ) {
022        setverdict( pass );
023        stop;
024      }
025      [] pt_pb.catch( timeout ) {
026        setverdict( fail );
```

```
027        stop;
028      }
029    }
030    unmap( self:pt_pb, system:pt_pb );
031  }
```

The TTCN-3 verdict type is special in a way that its value can never be degraded, meaning that if a verdict is set to fail it cannot later be set to pass. Any test component can set the test case verdict eliminating the need for verdict synchronization after the test components have executed.

6 Conclusions

This paper presented an approach to implement a TTCN-3 test platform for software testing. The capabilities of TTCN-3 enable efficient ways of testing software modules in a standardized way with standardized test system interfaces. One language can be used for all test cases independent of the language of the IUT so there is no need to learn test tool proprietary languages. Utilizing standardized interfaces for modular test platforms make it possible to efficiently set up new test systems and at the same time eliminate possible overlapping test system development work.

Setting up a distributed test system provides a number of advantages that cannot be achieved with a tighter test system, e.g. the problem of a shared memory where the tested IUT can crash the process responsible for executing and evaluating test suites, and target platform independence.

The core TTCN-3 core language allows for creation of multi component test cases and easy means to set up synchronization between the components, making it possible to approach testing problematic domains as concurrency and non-deterministic behavior.

As the procedure-based paradigm is only a part of the TTCN-3 language a combination with the message-based parts can be used for test suites that interact with the SUT through different kinds of interfaces. This can lead to multi purpose test suites with a lot of useful capabilities such as unit testing with mock objects and possibilities to control the SUT with function calls while performing protocol testing. Multi purpose functionality that can be hard to achieve based on test area targeted frameworks e.g. unit testing and protocol testing frameworks.

Further case studies will be done to evaluate the usefulness of TTCN-3 based software module testing in several areas such as ease of use, test suite reusability, concurrent testing efficiency and implementation of synchronized deterministic test cases. Also its applicability to testing of distributed operating systems and language independent SUTs will be further explored. We believe that TTCN-3 has enough advantages to make it as a powerful alternative and language for future software module testing test systems. One test system and one test language offer the ability to perform multi purpose software testing capabilities written in any language.

Currently a lot of mapping work has still to be done manually. However, this can based on the language mappings be generated by tools, including TTCN-3 signatures and types, marshalling, invocation and mock object parts for the SUT based on the definition of the software module to test. The requirement is that a specified mapping exists or can be made from the languages of the IUT to TTCN-3, something that has proven to be a challenge especially for C and C++.

Acknowledgements

A lot of useful ideas for a C mapping has come from the long discussions with Matti Kärkki and Pekka Pulkkinen who are the originators of some the C++ to TTCN-3 mapping work done. Special thanks also to the TTCN-3 people at the Nokia Research Center, Thomas Deiß and Stephan Schulz for feedback and review of this paper, and also to Federico Engler, Sami Heinonen, Martti Söderlund, Stephan Tobies and Colin Willcock.

References

1. Itoh, E. Furukawa, Z. Ushijima, K. A prototype of a concurrent behavior monitoring tool for testing of concurrent programs. IEEE, 1996.
2. Tai, K, C. Testing of Concurrent Software, Computer Software and Applications Conference, 1989. COMPSAC 89, Proceedings of the 13th Annual International 20-22 Sept.
3. Tai, K. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. Proceedings 1994 International Conference Parallel Processing. 1994.
4. Mackinnon, T. Freeman, S. Craig, P. Endo-Testing: Unit Testing with Mock Objects, Proceedings XP2000.
5. CUnit 2005: CUnit (2005) Retrieved November 9, 2005, from CUnit Web site: http://cunit.sourceforge.net/.
6. JUnit 2005: JUnit (2005) Retrieved November 9, 2005, from JUnit Web site: http://junit.org/index.htm.
7. Moseley, S. Randall, S. Wiles, A. Schulz, S. IPv6 Test Specifications from ETSI. Global Ipv6 Summit, Barcelona, June 2005.
8. Hwang, G. Tai, K. Huang, T. Reachability Testing: an approach to testing concurrent software. Software Engineering Conference, 1994. Proceedings. 1994 First Asia-Pacific 7-9 Dec.
9. Carver, R. Mutation-based testing of concurrent programs. Test Conference, 1993. Proceedings. International 17-21 Oct. 1993.
10. Thomas, D. Hunt, A. Mock Objects. Software, IEEE Volume 19, Issue 3, May-June 2002.
11. Ebner, M. (2001). A Mapping of OMG IDL to TTCN-3. University of Lübeck, Germany.
12. Beizer, B. Software Testing Techniques, 2nd ed, p41-54. Van Nostrand Reinhold, 1990.
13. ETSI ES 201 873 “Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3”; V3.0.0, Sophia Antipolis, March 2005.
14. InformationTechnology, OSI conformance testing methodology and framework. ISO/IEC, 1994-1997. International Telecommunication Union recommendation X.290.
15. ISO/IEC 9899:1999: “Programming languages - C”. New York, NY, USA (1999-12)
16. Willcock, C. Deiß, T. Tobies, S. Keil, S. Engler, F. Schulz, S. (2005). TTCN-3 Test Systems in Practice: An Introduction to TTCN-3. England: John Wiley and Sons Ltd.