Realizing Distributed TTCN-3 Test Systems with TCI

Ina Schieferdecker[1] and Theofanis Vassiliou-Gioles[2]

[1] Fraunhofer FOKUS, Competence Center for Testing, Interoperability and Performance
Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany
schieferdecker@fokus.fhg.de
[2] Testing Technologies IST GmbH, Oranienburger Str. 65, D-10117 Berlin, Germany
vassiliou@testingtech.de

Abstract. Distributed test setups for efficient load, performance, scalability, interworking, and end-to-end tests are gaining importance for the assessment of distributed communicating systems. The Testing and Test Control Notation TTCN-3 provides concepts for component-based distributed test systems in dynamic test configurations, where test components may reside on various network nodes to be near the interfaces of the tested system. The realization of executable TTCN-3 tests on concrete test platforms involves TTCN-3 compilation/interpretation and adaptations to the test platform. The TTCN-3 Control Interfaces TCI define entities, interfaces, types and operations needed to flexibly manage and distribute TTCN-3 based test systems. It complements and completes the TTCN-3 Runtime Interface TRI. This paper discusses the underlying concepts of TCI and demonstrates its use for the realization of a distributed test for the Session Initiation Protocol SIP.

1 Introduction

The Testing and Test Control Notation TTCN-3 is a test specification and implementation language to define test procedures for black-box testing of distributed systems. TTCN-3 allows an easy and efficient description of complex distributed test behavior in terms of sequences, alternatives, loops and parallel stimuli and responses. Stimuli and responses are exchanged at the interfaces of the system under test, which are defined as a collection of ports being either message-based for asynchronous communication or signature-based for synchronous communication. The test system can use any number of test components to perform test procedures in parallel. Likewise to the interfaces of the system under test, the interfaces of the test components are described as ports.

The development of TTCN-3 was forced by key players of the telecommunication industries and science to get a single test notation for nearly all black-box testing needs. Especially the newly introduced support of dynamic distributed test setups, i.e. dynamic creation and termination of test components including dynamic connections between test components and to the system under test (the SUT), enables new applications of TTCN while keeping the mature and stable test concepts. TTCN-3 test specifications are not only a basis for functional and conformance testing, but also for performance, load and scalability tests. Such tests require varying load conditions for the SUT, which can be realized by an ensemble of parallel test components. Since the

D. Hogrefe and A. Wiles (Eds.): TestCom 2003, LNCS 2644, pp. 95–109, 2003.
© IFIP 2003

test system has to be as performant as the system under test, any realistic load for the SUT can be realized in a distributed environment only: the parallel test component have to be distributed and located on remote nodes in a network constituting a distributed test system.

One essential benefit of TTCN-3 is that it enables the specification of tests in a platform independent manner. Hence, TTCN-3 provides the concepts of test components, their creation, their communication links to each other and to the SUT, their execution and termination as such on abstract level only. Means to control the distributed execution of test components and coordination between them are outside TTCN-3.

However, the application of executable tests to a SUT within test campaigns requires the realization and implementation of such distributed test systems in a network of test nodes – at best in a well-defined manner to enable a standardized adaptation for the management, component handling, communication and logging between distributed test nodes. Another aspect of this adaptation is the ability to reuse external encoders/decoders, which are also outside TTCN-3 and just referenced within a test specification.

Well-defined interfaces as a set of operations independent of the target, i.e. independent of the SUT, processing platform, implementation language, etc will enable that code from any TTCN-3 compiler or interpreter supporting and using these interfaces can be executed on any test platform/test device, which supports and uses these interfaces. A first step towards this code independence was done with the TTCN-3 Runtime Interface TRI 24: TRI provides an interface to adapt a TTCN-3 test system to the SUT by providing means to adapt the communication with the SUT as well as to adapt the timer handling. As such, TRI defines a local adaptation to the SUT only. The aspects of test management, component handling (both on local and remote nodes) as well as the type and value handling have not been considered by TRI. These aspects can be summarized as being the adaptation to the test system being either a single test device or a test platform consisting of several test nodes[1]. TRI has to be supplemented by interfaces to enable a well-defined adaptation to the test platform. These interfaces are called the TTCN-3 Control Interfaces TCI.

TCI together with TRI provide a complete solution for a well-defined adaptation to the test system and to the SUT providing maximal flexibility in realizing TTCN-3 test systems. Only recently – at the ETSI MTS meeting, October 2002 – the importance of TRI and TCI for TTCN-3 has been reflected: TRI 4 and TCI 5 were made integral parts of the multi-part standard for TTCN-3.

TCI is currently developed at ETSI and is expected to become together with TRI the future standard interface set for all TTCN-3 test system implementations. It considers previous work in GCI 8 on type and value interfaces and in TSP1+ 7 and its implementation 6 on distributed test systems. However, the approach of TCI on generic distributed test setups is new in several respects:

- it enables the implementation of the new TTCN-3 concepts for dynamic test configurations – different network and platform technologies can be used to realize distribution and communication within the test platform,

[1] In the following we will use the term test platform to refer both to a single node as well as a multi node test system.

- it enables the flexible reuse of coders/decoders without predefining any internal type and value representation for the test system implementation and
- it enables the integration of TTCN-3 test systems into existing test management environments and applications by concentrating on TTCN-3 related test management aspects only: this enables more flexibility for test management specific application domain and goes beyond TSP1+ 7

We provide first insights into how various kinds of test systems can be build based on the concepts of TCI: in Section 2, the general entities, interfaces and types of TCI together with a selected uses case on a more complex test components handling are presented and discussed. Afterwards in Section 3, the approach is illustrated with an application example for testing the Session Initiation Protocol SIP 1. Section 4 concludes the paper with a summary and an outlook on the future work on TCI.

2 Overview of the TTCN-3 Control Interfaces

A TTCN-3 test system can be conceptually thought of as a set of interacting entities where each entity corresponds to a particular aspect of functionality in a test system implementation. These entities manage test execution, interpret or execute compiled TTCN-3 code, realize proper communication with the SUT, handle types, values and test components, implement external functions, and handle timer operations.

The part of the test system that deals with interpretation and execution of TTCN-3 modules, i.e., the Executable Test Suite (ETS), is shown as the *TTCN-3 Executable (TE)*. Within the TE individual structural elements can be identified, like Control, Behaviour, Components, Types and Values and Queues. The structural elements within the TE represent functionality that is defined within a TTCN-3 module or by a TTCN-3 specification itself. For example, the structural element “Control” represents the control part within a TTCN-3 module, while the structural element “Queues” represent the requirement on a TTCN-3 Executable that each port of a test component maintains its own port queue. While the first is specified within a TTCN-3 module the later is defined by the TTCN-3 specification. The TE corresponds typically to the executable code produced by a TTCN-3 compiler or a TTCN-3 interpreter. Prior to a test system implementation, the *Abstract Test Suite (ATS)*, i.e., the TTCN-3 modules being the test specification, has been compiled into an executable format - the *Executable Test Suite (ETS)*. The TE can be executed centralized, i.e. on a single test device or distributed, i.e. on different physical test devices. Although the structural entities of the TE implement a complete TTCN-3 module, single structural entities might be distributed over several test devices.

The TE implements a TTCN-3 module on an abstract level. The other entities of a TTCN-3 test make these abstract concepts concrete. For example the abstract concept of sending an event or receiving a timeout cannot be implemented within the TE. The platform adaptors of the test system realize e.g. the encoding of the message and it’s sending over concrete physical means or measuring the time and determine when a timer has expired, respectively. The *System Adaptor (SA)* for the communication with the SUT and the *Platform Adaptor (PA)* for the realization of timers and their interaction with the TE are defined in [TRI].

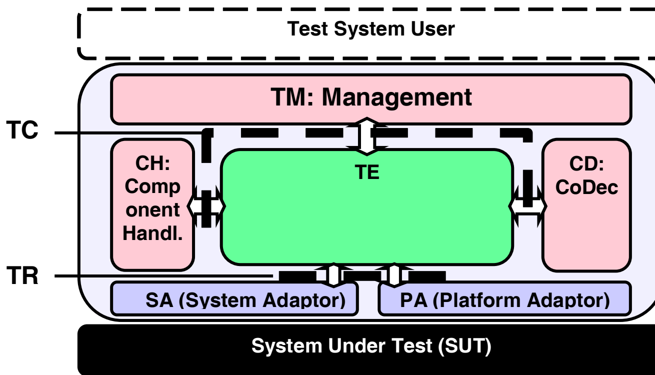

Fig. 1. General Structure of a TTCN-3 Test System. A TTCN-3 test system implementation consists of a part that deals with the interpretation and execution of TTCN-3 module (TE) and parts that either adapt the test system to the System Under Test (SA and PA) or to the test system platform (TM, CD and CH).

The TCI specification defines the interaction between the TE and the *Test Management and Control (TMC)* entities. In TMC, we can distinguish between functionality related to test execution management (TM), component handling (CH), and coding and decoding handling (CD).

The *test management (TM)* entity is responsible for the overall management of a test system. After the test system has been initialised, test execution starts within the TM entity. The entity is responsible for the proper invocation of TTCN-3 modules, i.e., propagating module parameters and/or IXIT information, i.e. implementation extra information for testing, to the TE if necessary. Typically, this entity would also implement a test system user interface. In addition, the TM entity performs test event logging and presentation to the test system user.

As the TE can be distributed among several test devices the *component handling (CH)* is responsible to implement the distribution and communication between the distributed entities. The CH provides the means to synchronize the different entities of the test system being potentially located on several nodes. The general structure of a test system distributed via several nodes is depicted in Figure 2.

On each node, a test execution TE together with system adaptor SA, platform adaptor PA and coder/decoder CD is performed. The entities CH and TM mediate the test management and test component handling between the TEs on each node. There is a special TE that is identified to be the TE that started a test case[2] and that is responsible for calculating the final verdict of that test case. Besides this, all TEs are handled the same.

Communication is in one respect the message or procedure based communication between TTCN-3 components. Therefore, the CH adapts message and procedure based communication of TTCN-3 components to the particular execution platform of the test system. It is aware of connections between TTCN-3 test component communication ports. It is responsible to propagate send request operations from a single

---

[2] Please note that in course of executing a TTCN-3 module there can be at most one test case being executed.

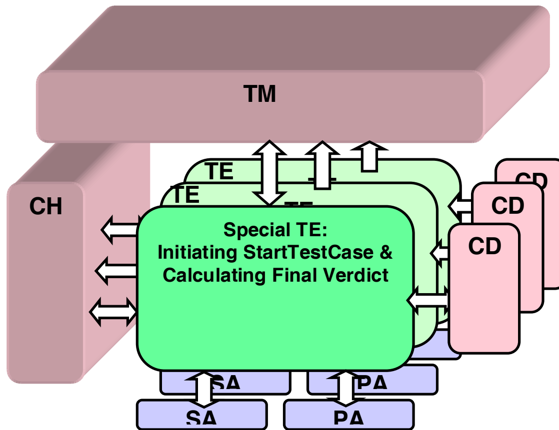

Fig. 2. General Structure of a distributed TTCN-3 Test System. A distributed TTCN-3 test system consists of one CH and TM entity. Each TE is located on a separate node, together with its own SA, PA and CD entities.

TTCN-3 component that resides within a certain TE to the targeted component residing potentially in a different instance of the same TE on a different test device. It then notifies the TE about received test events by enqueueing them in the port queues of the TE.

Procedure based communication operations between TTCN-3 components are also visible at the CH. The CH is responsible to distinguish between the different messages within procedure-based communication (i.e., call, reply, and exception) and to propagate them in the appropriate manner to the targeted component TE. TTCN-3 procedure based communication semantics, i.e., the effect of such operation on TTCN-3 test component execution, are to be handled in the TE.

Furthermore, there is additional test component management communication necessary in order to implement the distribution of test components between several test devices. Component management communication includes the indication of the creation of test components, the starting of execution of a test component, verdict distribution as well as component termination indication. For this the CH does not implement the behaviour of TTCN-3 component but the communication between several components that are implemented within the TE.

The *coding/decoding (CD)* entity is responsible for the encoding and decoding of TTCN-3 values into bitstrings suitable to be sent to the SUT. The TE determines which codecs shall be used and passes the TTCN-3 data to the appropriate codec in order to obtain the encoded data. Received data is decoded in the CD entity by using the appropriate decoder, which translates the received data into TTCN-3 values.

The TCI operations of TM, CH and CD are atomic operations in the calling entity. The called entity, which implements a TCI operation, returns control to the calling entity as soon as its intended effect has been accomplished or if the operation cannot be completed successfully. The called entity is not blocked, so that performant test implementations are enabled.

TCI is defined by a set of abstract types to realize the TTCN-3 type and value system, a set of operations at required and provided subinterfaces of TM, CH and CD, and a set of scenarios to show the use of abstract types and operations. The types and

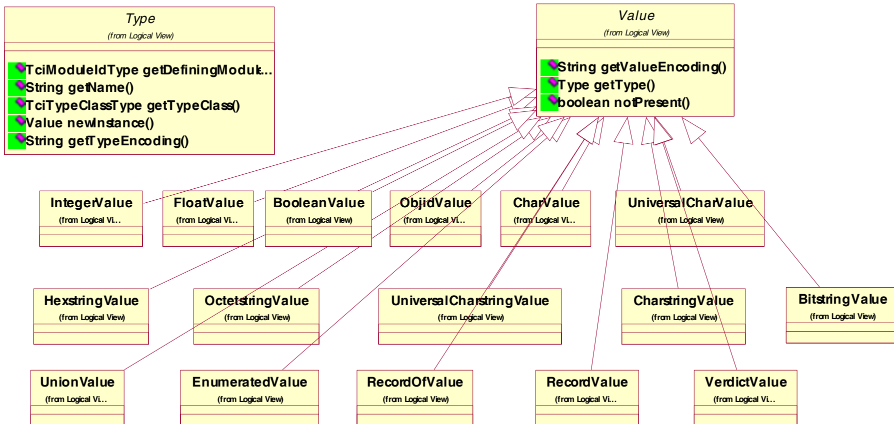

Fig. 3. Hierarchy of abstract values. Each abstract value provides at least the operations of the abstract data type `Value` and form a hierarchy of abstract values. Operations on abstract values are not shown in this figure.

operations are provided in the OMG IDL 9. Language mappings to Java and C will show the realization of TCI on specific test platforms.

2.1 The Abstract Data Type Model

Abstract data types are use to describe on a high-level which kind of data shall be passed from a calling to a called entity. In addition, the abstract data types are used to define how TTCN-3 data is passed from the TE to a coder that encode a TTCN-3 value representation to a bitstring and from a decoder to the TE to decode a bitstring into a TTCN-3 value representation. For these abstract data type a set of operations are defined in order to process the data by the coder/decoder.´ The concrete representation of these abstract data types as well as the definition of basic data types like `String, boolean` are defined in a respective language mappings.

A set of abstract data types builds up the TTCN-3 type and value representation. For every abstract data type a set of operations has been defined in order to define the functionality of the abstract data type. Operations on or with this abstract data type return either a value of this abstract type or a basic type like `boolean`. The abstract TTCN-3 type and value representation consists of two parts:

- An abstract data type `Type` that represents all TTCN-3 types in a TTCN-3 module
- Different abstract data types that represent TTCN-3 values, i.e. TTCN-3 values of a given TTCN-3 type. This can be either values of TTCN-3 predefined types or of TTCN-3 user-defined types.

For the abstract data type `Type` operations to reference predefined and user-defined TTCN-3 data types, and to create and maintain TTCN-3 values are defined. The following figure presents the hierarchy between the abstract data types for TTCN-3 values (short: `abstract values`):

All TTCN-3 abstract values share a common base abstract data type, the abstract data type `Value`. For the abstract values that share the common base abstract data

type, all operations that are defined on the base data type are implicitly defined for the abstract values, too.

2.2 The Test Management Interface

The TCI Test Management Interface (TM) describes the operations a TE is required to implement and the operations a test management implementation shall provide to the TE. A test management implementation provides overall test management to the test system user. It requires from the TE the presence of operations to start and stop test execution of a TTCN-3 module or of certain test cases in a TTCN-3 module. In turn it provides operations to the TE for resolving module parameter at runtime, logging and the indication of execution termination.

2.3 The Component Handling Interface

A component handling implementation distributes TTCN-3 configuration operations like create, connect and start and inter-component communication like send on a connected port among one or more TTCN-3 Executables participating in a test session.

The basic principle is that CD is not *implementing* any kind TTCN-3 functionality. Instead it will be informed by the TE that for example a test component shall be created. Based on component handling (CH) internal knowledge the request for creation of a test component will be transmitted to another (remote) participating TE. This second (remote) participating TE will create the TTCN-3 component and will provide a handle back to the requesting (local) TE. The requesting (local) TE can now operate on the created test component via this component handle.

Within the operation definitions, the terms local TE and remote TE is used to highlight the fact that a test system implementation might be distributed over several test devices, each of them hosting a complete TE. The terms "local" and "remote" always refer to the interface currently described. For convenience reasons, the term "local" refers always to the TE being either the callee of an operation (for *required* operations) or the caller of an operation (for *provided* operations). While the TE is conceptually considered as being distributed the CH is considered to be non-distributed. This can either be achieved using a centralized architecture or by using a middleware-platform that abstracts from distribution aspects. Although the TE might be distributed over different physical device, there might be configurations where only one, non-distributed TE will participate in a test session. In this case the term "local" and "remote" refer to the same TE instance.

Although all TTCN-3 Executable participating in a test session are equal there is a distinct TE*. This TE* is the TE where a test case has been started explicitly, i.e. the explicit `tciStartTestCase()` has been processed, or the test control by means of `tciStartControl()`. The reason for this distinction is, that this TE* is responsible for global verdict calculation and is therefore informed about any test component termination yielding the final local verdict of the terminated test component. Finally, the TE* will notify the test management upon termination of a test case execution with the overall final test case verdict.

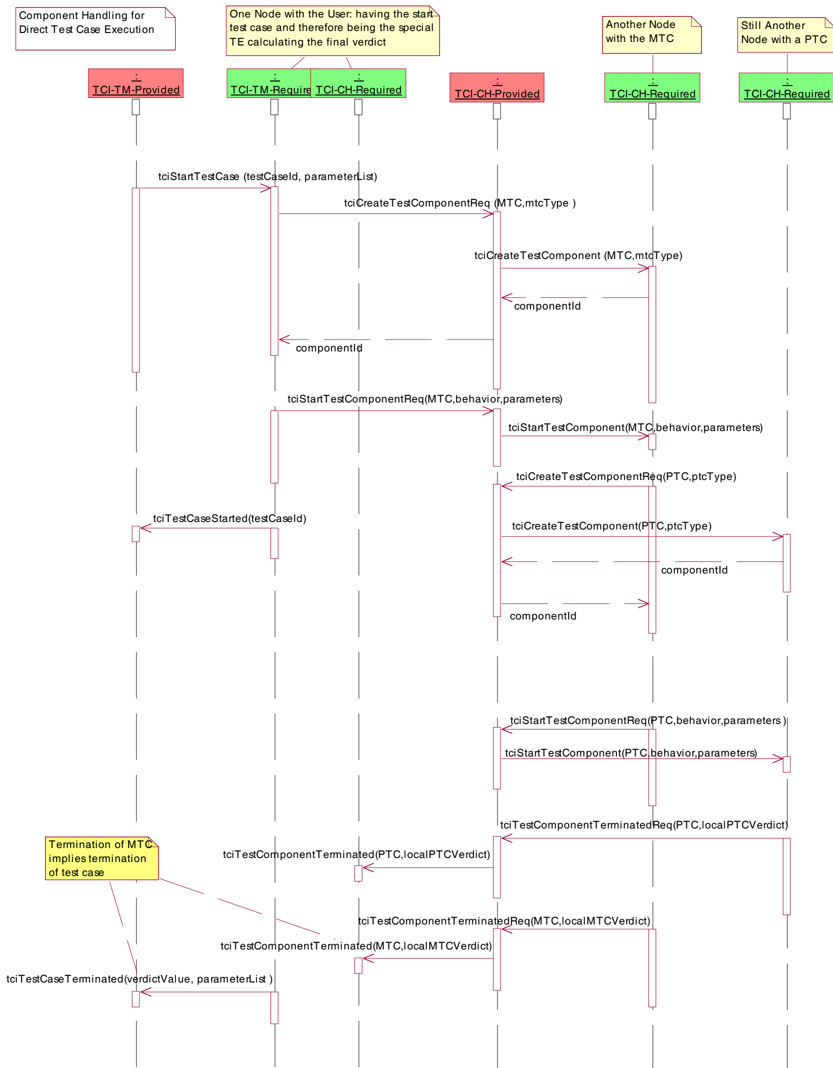

Fig. 4. Test case execution and termination. Test execution will be started at the TCI-TM interface. TM will be informed after termination of test execution together with the final and the actual parameter list.

2.4 The Codec Interface

A codec implementation encodes TTCN-3 values according to the encoding attribute into a bitstring and decodes a bitstring according to decoding hypothesis. To be able to decode a bitstring into a TTCN-3 value, the CD requires certain functionality from the TE. The basic operation required by CD implementation from the TE is the provisioning of value instances, either for basic or structured types. Together with the pos-

sibility to query values on their type information the CD provides encoding and decoding functionality to the TE. An example of how this can be achieved will be presented later.

2.5 A Selected Use Case

This section shows a use case on starting a test case directly from the test management user interface. The module containing the test case is selected first. When the test case is started, the main test component is created. Then, the test case behaviour is started on this main test component. Whenever a parallel test component is used within a test case, it is handled the same: the parallel test component is created first: giving a test component create request to the TCI-CH entity, which propagates the test component create to the TE in which the parallel test component shall be created.

The identifier for the created parallel test component is returned. The identifier is then used to start the PTC behaviour of the start operation. When the PTC terminates its execution, a test component terminate request together with the local test verdict is issued in order to inform CH about this termination. The same is done when the main test component terminates. In addition, the termination of the main test component leads to the overall termination of the test case. The test management interface receives in that case the final verdict.

3 A Distributed TTCN-3 Test System Example

To illustrate the construction of a distributed test system, a real-world IP application scenario is presented. The scenario demonstrates the transition from a pure functional test to a scalability test for the Session Initiation Protocol (SIP) [1]. In the given example, the ability of a proxy server to handle simultaneous initiation of several multimedia sessions is analyzed. SIP is an IETF signaling protocol for the establishment, maintenance and tear down of multimedia connections in the Internet using UDP or TCP as underlying transport mechanism.

Figure 5. illustrates a general test configuration for testing a SIP UA Server. Figure 5.a displays a test configuration where a single Parallel Test Component, executing a functional test behavior, interacts with the System Under Test, while Figure 5.b shows the simultaneous execution of the same functional behavior on multiple parallel test components. In both cases, the Master Test Component coordinates the test execution.

3.1 A TTCN-3 ATS Example

A TTCN-3 abstract test suite (ATS) specifying a simple functional test case using the single PTC configuration can be specified as follows:

```
module sipTest {
  testcase functionalTest()
      runs on MyMtcType system MyTSI
  {
    var MyPtcType sipUAc = MyPtcType.create;
    map(sipUAc:S, system:R);
    connect(mtc:C, sipUAc:C);
    sipUAc.start(uaCBehavior(USER));
    C.send(startPrimitive);
    all component.done;
}
```

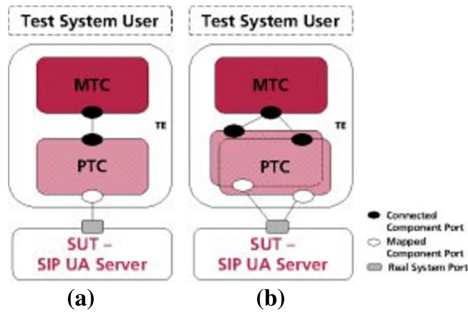

Fig. 5. A general SIP User Agent (UA) Server testing configuration. Parallel test components (PTC) interact with the System Under Test (SUT) at the real the real system port[3]. The parallel test components communicate with the Master Test Component (MTC) using the connected ports. The Master and the Parallel Test Components are executed within the TTCN-3 Executable (TE) while the Test System User manages test execution.

First, the Master Test Component of type `MyMtcType` creates a Parallel Test Component of type `MyPtcType`. After mapping the PTC port `S` to a Test System Interface port `R` (as defined in `MyTSI`) and connecting the port `C` of the MTC and the PTC the behavior `uaCBehavior()` will be executed. The MTC instructs the PTC to proceed with test case execution by sending the `startPrimitive` template. The test case will terminate after the PTC has terminated test behavior execution. This example assumes that `uaCBehavior()` specifies all the necessary behavior in order to asses the SUT.

One aim in a multiple test components scenario is to reuse definitions of a functional test scenario for the definition of e.g. a scalability test. The PTC will execute the same test behavior while the MTC is responsible to set up the test configuration.

In our particular example, `MAXNUMBER` PTCs shall execute their behavior simultaneously. For this the MTC instructs the PTCs to start the test via the `startPrimitive` after all participating test components have been created and the functional test behavior `uaCBehavior()` has been started[4]. `MAXNUMBER` has been defined as an integer module parameter. The value of `MAXNUMBER` will be resolved at runtime.

The following TTCN-3 fragment illustrates this scalability test scenario:

```
modulepar { integer MAXNUMBER := 10 ; // 10 is default }

testcase scalabilityTest()
    runs on MyMtcType system MyTSI
```

[3] The figures display only the mapped component ports at the PTC for the communication with the SUT. The necessary test system interface port has been omitted for the sake of readability.

[4] We distinguish between the starting of test component, which is achieved using the start operation on a component and the execution of a the test, i.e. performing communication with the SUT to asses its validity. From a TTCN-3 point of view the test behavior start after the call of the start operation. From the test logic point of view we assume that the test will start after the reception of the `startPrimitive`.

```
{
  var integer i;

  for(i:=0; i < MAXNUMBER; i := i + 1) {
    sipUAc[i] = MyPtcType.create;
    map(sipUAc[i]:S, system:R);
    connect(mtc:C, sipUAc[i]:C);
    sipUAc.start(uaCBehavior(USER[i]));
  }

  for(i:=0; i < MAXNUMBER; i := i + 1) {
    C.send(startPrimitive) to sipUAc[i] ;
  }

  all component.done ;
}
```

Typically, a TTCN-3 ATS contains not only the type, data and behavior definitions but also a control part, that relates the execution of test cases. This example assumes that the `scalabilityTest()` will be executed only if the `functionalTest()` was successful, i.e. terminated with the verdict `PASS`.

3.2 Application of the TCI Operations

Although the definition of language mapping for TCI is still in progress, a Java language mapping, derived from the IDL definitions in TCI is presented here to illustrate the application of the TCI operations. Although the used operations and signature are by no means complete they present a good selection for an insight into possible implementation of TCI operations.

In order to start test execution, the test system user has to instruct the TTCN-3 Executable (TE) to start either the control part of the module or a particular test case. TCI defines for this two different operations, the `tciStartTestCase()` and the `tciStartControl()` operation. Prior to test execution the module has to be selected.

```
public class MyManagement() implements TciTMProvided {
  static TciTMRequired TE = getTE();

  public void main(String[] args) {
    TE.tciSetModule("sipTest");
    if(startTestCase) {
      TE.tciStartTestCase("functionalTest", null);
      waitForTestCaseTermination() ;
      TE.tciStartTestCase("scalabilityTest", null);
      waitForTestCaseTermination() ;
    } else {
      TE.tciStartControl();
      waitForTestControlTermination(); }
  }
  // -- TCI-TM Provided Implementations --
  ...
  public Value tciGetModulePar(String param) {
    if(param.equals("MAXNUMBER")) return determineMAXNUMBER();
    return null; }

  public void tciTestCaseTerminated(VerdictValue
      verdict, TciParameterList list) { ... }


  public void tciTestControlTerminated() { ... }
}
```

Like all following code fragments, this fragment is not complete and shall give only an impression on how the operations exposed by the TE in the *TM Required* interface might be used, and how the TM and the other components provide the functionality needed for correct execution of the TTCN-3 within the TE.

The main method first determines whether the test cases should be executed from within the control part (`startTestCase == false`) or the test cases shall be started directly. After test execution has been initiated within the TE, the TM (an instance of class `MyManagement`) has to wait until the TE indicates termination of test execution using the methods provided by the TM.

The TE can be accessed by the TM using a global variable `TE`, which implements the interface `TciTMRequired`. The handling of this global knowledge is out of scope of TCI and will not be further discussed.

After test execution has started, the TE will execute the test behavior as defined in the TTCN-3 specification. The TCI Component Handling Interface addresses all issues related to component management and inter-component communication. A possible message exchange for a test component creation is shown in Figure 5.

The following Java fragment shows a possible implementation of the CH:

```
public class MyCH() implements TciCHProvided {


  static TciCHRequired ONE_NODE     = getONE_NODE();
  static TciCHRequired ANOTHER_NODE = getANOTHER_NODE();
  static int noOfComponentsCreated  = 0 ;

  public TriComponentId tciCreateTestComponentReq (
      TciTestComponentKind kind, Type componentType) {
    if(distributed && ((noOfComponentsCreated % 2) == 0) {
       TriComponentId tc =
          ANOTHER_NODE.tciCreateTestComponent(kind,
          componentType); register(kind, tc, ANOTHER_NODE);
       noOfComponentsCreated++;
       return tc ; }
    else {
       TriComponentId tc =
          ONE_NODE.tciCreateTestComponent(kind,
          componentType); register(kind, tc, ONE_NODE);
       noOfComponentsCreated++;
       return tc ; }


  }
}
```

This implementation assumes that there will be only one instance of `MyCH` (the CH) and that this instance is known to the TE. However, the fragment shows also that CH can have access to multiple instances of TE. The location of this, possible different, instances is not restricted to the same test device. It is an implementation decision on how to retrieve references to the distributed objects.

Based on the variable `distributed`, every second component will be created on a remote (`ANOTHER_NODE`) TE implementation. If distributed is `false` every component will be created on the local TE implementation (`ONE_NODE`)[5]. As `tciComponentCreationReq` will be called also when a test component creates another test component (e.g. the MTC a PTC, or a PTC another PTC), this implementation would distribute test components equally on both participating nodes, `ONE_NODE` and `ANOTHER_NODE`. CH performs book-keeping by registering the kind, the id and the place a component resides for later usage.

---

[5] It is assumed that the `tciCreateComponentReq` was called by `ONE_NODE`.

Starting of test behavior as well as setting up connections between components and performing communication is performed in a comparable way. The following Java fragment can give an impression on how the CH could be extended to provide this functionality.

```
public void tciConnectReq (
    TriPortId fromPort, TriPortId toPort) {
  TriComponentId fromC=fromPort.getComponentId() ;
  TriComponentId toC=toPort.getComponentId() ;

  // resolve() returns the TciCHRequired instance the
  // remote component resides on
  resolve(fromC).tciConnected(fromPort, toPort) ;
  resolve(toC).tciConnected(toPort, fromPort) ;
  registerConnection(fromPort, toPort) ;
}

public void tciSendConnected (TriPortId sender,
    TriComponentId receiver, Value sendMessage) {

    // retrieveConnection returns the TciCHRequired
    //instance the receiver resides on
    retrieveConnection(sender, receiver).
        tciEnqueueMsgConnected(sender,
             receiver, sendMessage) ;
}
```

As can be seen, the main task of the CH is to route the requested operations to the destination and monitor the setup and tear down of connections. By exposing this TCI-CH interface to the user, the user can implement distribution strategies as required. Whenever data has to be passed from the TE to the remaining test system, i.e. TM, CH or SA and PA (as defined in [4]), data that has been defined abstract within TTCN-3 has to be translated into a concrete representation. In case of communication with the SUT the abstract data has to be encoded according to the encoding rules. Besides the abstract TTCN-3 data types and values as described in previous sections, TCI defines the Codec Interface (TCI-CD) to enable the TE to pass the abstract TTCN-3 data to the appropriate codecs prior to sending data or performing matching operations on received data. For each encoding rule CD provides two operations to the TE, `encode()` and `decode()`. An implementation in Java might look as follows:

```
public class MyEncodingRule() implements TciCDProvided {
   public TriMessage encode(Value value) {

   // TRI
   TriMessage encodedMsg = new TriMessageImpl();
   return encodedMsg.setEncodedMessage(furtherEncode(value));
   }

   public Value decode(TriMessage message,
      Type decodingHypothesis) {
      byte[] encodedMsg = message.getEncodedMessage() ; //TRI
      return furtherDecode(encodedMsg, decodingHypothesis) ;
   }
}
```

For each encoding rule this functionality has to be provided. In order to be able to encode the TTCN-3 data from a from a `Value` into an encoded message, the TE offers additional functionality to the TM, CH and the CD for the handling of types and values.

The abstract data type `Type` offers some functionality to obtain instances of any TTCN-3 type as defined in a TTCN-3 module. This applies both for predefined and

user-defined types. Differentiation between basic types and structured types can be achieved using a type class concept, i.e. different types classes exists for each predefined or subtype TTCN-3 type, (e.g. the type class is `BITSTRING` if the type represent a TTCN-3 `bitstring` type) and for structured types (e.g. the class is `RECORD` if the type represents a TTCN-3 record type). Additionally the encoding of a type, according to the TTCN-3 specification can be retrieved. Instantiations of a given type, the `Values`, can be created by using the `newInstance()` operation that is defined for the abstract data type `Type`.

Values can be read and manipulated depending on their type. For example for a TTCN-3 `integer` the abstract data type `IntegerValue` has been defined, with operation like

```
- integer getInt()
- void setInt(in integer integerValue )
```

TTCN-3 `record` values have operations like

```
- Value getField(in String fieldName)
- void  setField(in String fieldName, Value fieldValue)
- String[] getFieldNames()
```

Using this operation as offered by the TE, CD is able to encode and decode values according to the encoding, CH is able to send messages without the need of encoding them, and the TM to provide value for module parameters.

4 Conclusions

This paper discusses the realization of distributed test systems being defined in TTCN-3. Although TTCN-3 does not define distribution schemes or distribution patterns for test components, i.e. distribution is not in the scope of TTCN-3, the pure fact that component-based test systems with dynamic test configurations can be used to define e.g. load and scalability tests on an abstract level requires the ability for distribution: only if a test system is performant enough it can make valid assessments about a test systems. The possibility to distribute test components and to handle their setup, coordination and communication is key here.

The TTCN-3 Control Interfaces TCI are an approach to close this gap and provide together with TRI a complete set of interfaces, entities, types and operations for the adaptation of TTCN-3 tests to the test platform and the tested system. TCI provides means for test management, test component handling and coding/decoding. This paper presents the basic concepts of TCI and discussed its implementation concepts. The potential of TCI are illustrated for a load test of a SIP application.

Future work beyond TCI will consider automated means for test deployment onto test platforms: while TCI provides means for automated test execution, the preparation of the test platform with e.g. all the code needed to perform the tests or the configuration of the SUT and the test devices, is not yet considered. First approaches exist for the deployment of distributed systems in general. These concepts need to be investigated for application in a test context in general and specifically for TTCN-3. An automated and flexible mixture of SUT and test components to check certain test purposes will be in particular interesting for software testing.

References

1. J. Rosenberg, H. Schulzrinne, et al: “SIP: Session Initiation Protocol”, Draft IETF SIP RFC 3621, June 2002.
2. S. Schulz, T. Vassiliou-Gioles: “Implementation of TTCN-3 Test Systems using the TRI”, IFIP 14th Intern. Conf. on Testing Communicating Systems -TestCom 2002-, Berlin, Germany, March 2002.
3. ETSI ES 201 873 – 1, v2.2.1: "The Testing and Test Control Notation TTCN-3: Core Language ", Oct. 2002.
4. ETSI ES 201 873 – 5, v1.0: "The TTCN-3 Runtime Interface (TRI); Concepts and Definition of the TRI", Oct. 2002, Draft.
5. ETSI DES 201 873 – 6, v1.0: "The TTCN-3 Control Interfaces (TCI); Concepts and Definition of the TCI", Oct. 2002, Draft.
6. T. Vassiliou-Gioles, M. Li, I. Schieferdecker, M. Born, M. Winkler: Configuration and Execution Support for Distributed Tests. - IFIP 12th International Workshop on Testing of Communicating Systems (IWTCS'99), Budapest (Hungary), Sept. 1999.
7. ETSI ES 201 770 V4.2.4: "The Test Synchronization Protocol TSP1+", Sept. 2000.
8. F. Brady and R.M. Baker, "INTOOL/GCI; Generic Compiler/Interpreter interface; GCI Interface Specification", INTOOL CGI/NPL038 (V2.2), Infrastructural Tools for Information Technology and Telecommunications Conformance Testing, Dec. 1996.
9. OMG CORBA v2.2: “The Common Object Request Broker: Architecture and Specification”, Section 3, Feb. 1998.