Execution of External Applications using TTCN-3

Theofanis Vassiliou-Gioles[1], George Din[2], Ina Schieferdecker[2]

[1]Testing Technologies IST GmbH, Oranienburger Str. 65, D-10117 Berlin, Germany
vassiliou@testingtech.de
[2]Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany
{din,schieferdecker}@fokus.fraunhofer.de

Abstract. TTCN-3 allows an easy and efficient description of complex distributed test behaviour in terms of sequences, alternatives, loops and parallel stimuli and responses. The test system can use any number of test components to perform test procedures in parallel. Features which are not directly supported in TTCN-3 can be included by the use of external types, data and functions. However, the access to external test behaviour is rather limited as external functions are executed as plain functions only without being performed on test components. The combination of external functions and test components allows the extension of the communication between the TTCN-3 test system and the external behaviours towards complex send/receive patterns and supports the distribution of external behaviours on remote nodes. This paper discusses the concepts of external behaviours, proposes extensions to TTCN-3 to realise them and demonstrates its application by a concrete example.

1 Introduction

TTCN-3, the Testing and Test Control Notation, is the test specification and implementation language defined by the European Telecommunications Standards Institute (ETSI) for the precise definition of test procedures for black-box testing of distributed systems.

TTCN-3 allows an easy and efficient description of complex distributed test behaviour in terms of sequences, alternatives, loops and parallel stimuli and responses. The test system can use any number of test components to perform test procedures in parallel. One essential benefit of TTCN-3 is that it enables the specification of tests in a platform independent manner. TTCN-3 provides the concepts of test components, their creation, communication links between them and to the system under test (SUT), their execution and termination on an abstract level, yet together with TTCN-3 execution interfaces to provide the realisation of concrete executable tests on different target test platforms. Features and capabilities being beyond TTCN-3 can be integrated into TTCN-3 by the use of external types, data and functions.

TTCN-3 is currently been used in different domains, like in the telecommunications domain by network equipment vendors and carriers or in the service testing domain. Quite frequently it has been observed by these different user groups that the integration of external behaviour, i.e. non-TTCN-3 behaviour, is currently not foreseen within the language. As it will be shown in the following, the concept of external

function lacks flexibility in order to integrate behaviour provided in different languages than TTCN-3 seamlessly. Although TTCN-3 offers strong concepts for black-box testing, management of test cases and verdicts, the inability to define the execution of arbitrary test scripts keeps potential users with an already existing stock of different test solutions from using TTCN-3. The possible translation of existing test scripts being defined for example in Perl, Python or Tcl is not feasible as there is no mapping available for every script language. Even if the mappings would have been available, the implementation of translators or migration tools would exceed current budgets allocated for testing. So, the overall goal was and is to provide a generic approach on how to integrate arbitrary scripts in a TTCN-3 execution environment.

Extending the TTCN-3 language with the capability of external behaviours requires extensions in the execution environments as well. Beside the language artefacts required for working with external behaviours, we present in this paper also a general strategy of how existing TTCN-3 tools can be extended to support this feature. Although handling external behaviours imposes some changes in the execution environment, our approach respects fully the specification of the TTCN-3 test system architecture defined in [1][2][3]. Having such a framework available, a broad range of new applications for TTCN-3 is opened. For example, TTCN-3 can be used more efficient as a test management language since e.g. existing scripts for traffic generation can be easily related to TTCN-3 test behaviours and controlled by TTCN-3.

At first, this paper motivates in Section 2 the need to extend TTCN-3 with the concepts of external component and external behaviour. Section 3 outlines implementations strategies based on the existing TTCN-3 execution interface specifications TRI and TCI. Section 4 presents a concrete example combining TTCN-3 efficiently with a test script written in the Perl programming language. Section 5 concludes the paper with a summary and an outlook on future work.

2 A Concept Framework for External TTCN-3 Behaviour

The TTCN-3 language defines several constructs for describing the functionality of a test system. The behaviour of the test system can be specified by using one of the following constructs: control, test case, function or external function.

The *control part*, as the name also suggests, is the part of the test system where the overall behaviour is controlled. From the control part, the test cases are executed and the final verdict can be collected. The control part is defined with the **`control`** construct.

```
control { … }
```

A *test component* is the entity on which a test case or a function can be started. It may include in its type definition timers, variables, constants and ports, which can be accessed by any function or test case running on it. Test components are defined in TTCN-3 with the keyword **`component`**.

```
type component TestComponent {

     port PortType p;
     timer t1;

}
```

A *test case* is defined by the **testcase** keyword. It specifies the interaction with the SUT and/or the creation of test configurations using a number of parallel test components in addition to the main test component. It is able to instantiate types, to interact with the SUT by sending stimuli and receiving data, and to set verdicts. The communication with the SUT is realised via ports, which are defined by the test component on which it runs. In addition to the test component on which the test case runs, a system component can be specified which defines the interface to the SUT. If more than one test component is used by the test case, then the system component must be specified explicitly:

```
testcase T( … ) runs on TestComponent system SystemComponent { … }
```

A *function* defined by the keyword **function** is similar to a test case; it may deal with data, can access the variables, ports or timers of the test component on which it runs and may set verdicts. In the case where the function interacts with the SUT or with other test components, we call it a behaviour as compared to pure computational functions that do not interact with their environment. A behaviour differs from a test case by the fact that no system component is defined for it. A pure computational function has no runs on clause and no port parameters.

```
function F( … ) runs on TestComponent { … }
```

TTCN-3 has also the concept of *external functions*. An external function is introduced by **external function** keywords and defines the interface of a method whose implementation is outside the TTCN-3 test system. The glue between an external function and the TTCN-3 test system is realised in the platform adapter (PA) [3]. External functions can be called from control, test cases or functions. In contrast to a function, an external function does not run on a component and, because of that, cannot use ports to communicate with other test components.

```
external function F( … );
```

For our purpose of using external behaviours flexibly, the language concepts seem not to be enough. Test cases or functions, the only behaviours which are allowed to run on components cannot be used to execute external behaviours.

As presented, the only way to define non-TTCN-3 code in TTCN-3 is the external function concept, but even this is limited to the signature (i.e. the interface) of the external function. TTCN-3 explicitly forbids the communication between external functions and test components. Indeed, an external function can be used to run scripts defined outside TTCN-3; however we consider this not to be enough. At least, the following features should be supported by an external behaviour:

- The possibility to set verdicts from an external behaviour.
- An external behaviour may produce TCI errors, which cause the termination of the test execution.
- External behaviours may be distributed and started over many test hosts.

- An external behaviour shall use ports to communicate with other test components.

An external function, as defined in TTCN-3, can help only for the first two requirements as the return value can be used to return some termination status of the external function or to encode different error codes. The last one, which is probably the most important one, is not supported in TTCN-3 and can also not be substituted by any other construct.

Consequently, we were motivated to introduce the concepts of external behaviour and external component. The principal architecture for a test system using TTCN-3 test components and external test components is depicted in Fig. 1.

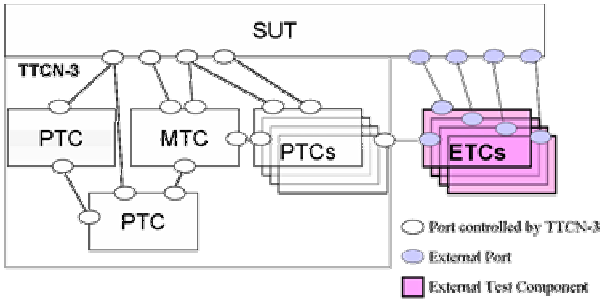

Fig. 1. Architecture for a test system using TTCN-3 test components (MTC – main test component and PTC – parallel test component) and external test components (ETC)

An external component is defined by using the keyword **external** with a component type definition:

```
type external component ExternalComponent {
     port PortType p;
}
```

An *external behaviour* is an external function which runs on a test component of type external:

```
external function F( … ) runs on ExternalComponent;
```

An external function with a **runs on** clause is the construct for a behaviour defined in another language than TTCN-3, able to interact with the SUT by using specific operations and able to communicate with other TTCN-3 components by using TTCN-3 ports. An external behaviour is executed on an external test component having ports only since timers and variables are only meaningful within the TTCN-3 test system. External behaviours may not return values – neither via inout or out parameters nor via a function return. The only way to pass information between the TTCN-3 test system and the external behaviour is via the ports the external behaviour is run-

ning on[1]. External behaviours are started with a start operation called on an external component.

The TTCN-3 language extension for external behaviour is defined below:

```
ExtBehaviourDef::= ExtKeyword FunctionKeyword FunctionIdentifier "("
[FunctionFormalParList] ")" RunsOnSpec ";"
```

We introduced in the TTCN-3 grammar the production of the non-terminal `ExtBehaviourDef`. It is similar to `ExtFunctionDef` except that we introduced the `RunsOnSpec`, omitted the return `ReturnType` and allow only in parameters as parameters to the external behaviour function.

An external behaviour is considered to be running on an entity being outside the TTCN-3 test system having its own types and data. It uses technology specific possibilities to interact with the SUT and for the definition of its behaviour. An external behaviour may contain the notion of timers and local variables and may evaluate even verdicts. However, these concepts do not align with the concepts of timers, variables and verdicts as defined in TTCN-3 – they are specific to the external behaviour only. An external behaviour is considered a test component by itself which we want to integrate into a TTCN-3 test system. By combining them, we understand the possibility that a TTCN-3 test system may create and start external behaviours locally or remotely, may exchange data with it and by doing so may get its result. A TTCN-3 test system should not share the local variables or constants, timers or functions with the external behaviour. We refer to the external behaviour as an application (script or program) written independently of TTCN-3. Thus, for example, any Perl or Tcl script can be integrated with TTCN-3 and no further extension or adaptation for it is needed.

The following items characterise an external behaviour:

- It may be an application which may run independently of TTCN-3.
- Therefore, no access to TTCN-3 data, types, timers, functions should be possible.
- An external behaviour communicates with the TTCN-3 test system by using ports. This is the reason why an external behaviour is started on an external component whose ports are used for communication with the TTCN-3 test system.
- Within the external behaviour no direct reference/use of TTCN-3 ports is possible, but rather the communication between the TTCN-3 ports and external behaviour is realised via an adapter – the platform specific wrapper (PSW) - that relates the communication interfaces of the external behaviour to the TTCN-3 ports they are connected to.
- The wrapper handles the communications with the application; as defined, the external behaviour has no access to the internal TTCN-3 types and

[1] Please note that also in TTCN-3 functions started on a test component cannot return values. Although the function might have defined a return type, a possible return is ignored when the function is being started on a test component.

data. In order to be able to receive and send, data encoding and decoding of TTCN-3 values must be performed by the PSW.

Since an external behaviour is not allowed to access timers or variables, it cannot be executed on regular components. Thus, we define the concept of external component as a restricted TTCN-3 test component.

An *external component* is a TTCN-3 test component on which external behaviours run. An external component may contain only ports which may connect to other TTCN-3 ports, but no map operation is permitted. Communication of the external behaviour with the SUT is completely defined by the external behaviour. No timers, variables or constants are allowed. Below we present the TTCN-3 language extension for external component definition:

```
StructuredTypeDef::=
    RecordDef
    | UnionDef
    | SetDef
    | RecordOfDef
    | SetOfDef
    | EnumDef
    | PortDef
    | ComponentDef
    | ExtComponentDef
```

A `StructuredTypeDef` contains now also the non-terminal `ExtComponentDef`. It is similar to `ComponentDef` except that it is prefixed by the `ExtKeyword`.

```
ExtComponentDef::= ExtKeyword ComponentKeyword
ComponentTypeIdentifier BeginChar [ExtComponentDefList] EndChar
```

Instead of a `ComponentDefList`, the `ExtComponentDef` uses an `ExtComponentDefList` non-terminal.

```
ExtComponentDefList::= { ExtComponentElementDef [SemiColon] }
```

The `ExtComponentDefList` is defined through the `ExtComponentElementDef` non-terminal which contains only the `PortInstance` non-terminal.

```
ExtComponentElementDef::=
    PortInstance
```

From the TTCN-3 view, an external component is functionally a minimised component. Except the few constraints presented before, external components are compatible with normal test components: they run in the TTCN-3 Executable (TE), are handled by the TCI component handling (CH) and may connect via ports to other TTCN-3 test components.

3 Implementation Guidelines

This section explains how to realise technically the concepts presented previously. We experimented with our Java based implementation of the test system architecture as presented in ETSI standard [1] by extending it to support the execution of external behaviours. In the practical setup, we used the Perl language to describe external behaviours. Based on our experience, we present subsequently a general approach, of how to extend existing TTCN execution environments with external behaviours.

The main design requirement for our system was to follow strictly the TTCN-3 execution interfaces TRI [2] and TCI [3]. This way, our approach is general and reusable. A different implementation can follow the same patterns of implementation.

3.1 Test System Architecture

Firstly, we describe the implementation of the ETSI Test System Architecture and its related interfaces. The general structure of a distributed TTCN-3 test system is presented in Fig. 2. A TTCN-3 test system is build up of a set of interacting entities, which manage the test execution, interpretation or execution of TTCN-3 code, realise the communication with the SUT, implement external functions and handle timer operations.

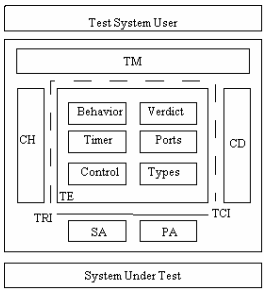

Fig. 2. General structure of a TTCN-3 test system as defined in TCI [3]. The test system contains the TTCN-3 executable (TE), which in turn communicates with the test management system (TM), the component handling (CH) and the coding/decoding (CD) via the TTCN-3 control interfaces. Communication with the SUT is performed using the TTCN-3 runtime interfaces (TRI), which define the interfaces between the TE and the system adapter (SA) and the platform adapter (PA).

The 6th part of the ETSI TTCN-3 standard [3] provides a standardised adaptation of a test system to a particular test platform by means of the management and handling of test components and encoding/decoding and defines the interaction between the three main entities: test management (TM), test component handling (CH) and coding/decoding (CD)[2]:

[2] We refer to the entity that provides encoding and coding functionality also as Codec.

- *TTCN-3 executable (TE)* interprets or executes the compiled TTCN-3 code. This component manages different entities: control, behaviour, component, type, value and queue, entities which are the basic constructors for the executable code.
- *Component handling (CH)* handles the communication between components. The CH API contains operations to: create, start, stop test components, establish the connection between test components (map, connect), handle the communication operations (send, receive, call and reply) and manage the verdicts. The information about the created components and their physical locations is stored in a repository within the Execution Environment.
- *Test management (TM)* manages the test execution. It implements operations to execute tests, to provide and set module parameters and external constants. The test logging is also realised by this component.
- *Coding/decoding (CD)*: encodes and decodes types and values. TTCN-3 values are encoded into bitstrings before being sent to the SUT. Received data being also bitstrings are decoded back into the corresponding TTCN-3 values.

According to this architecture, a test can be distributed over many test hosts and different test behaviours can be executed in parallel and simultaneously. As it is conceived within the standard, on each host an instance of the test system is created. In detail, the TTCN-3 execution (TE) must be installed on each host with its own coding and decoding (CD), system adapter (SA) and platform adapter (PA).

The 5th part of the ETSI TTCN-3 standard [2] defines the TTCN-3 runtime interfaces for the communication between the TE and the test adapter (TA) and platform adapter (PA). This part defines methods how the TE sends data to the SUT and manipulates timers and methods how the SA and PA notify the TE about received test data and timeouts.

Our implementation of that architecture is represented in Fig. 3. A Java and CORBA-based platform is the platform used to implement the *execution environment* (EE). The EE contains the additional entities needed to manage the test system entities. The interaction between EE and the test system components is tool specific and depends on the underlying technology. For example in Java, it is very easy to implement the EE as a static class which can be accessed from everywhere. The TE is a standalone process, which instantiates components, timers, ports and behaviours. TE gets a reference to EE by which the CH, TM, SA and PA are accessible. The timers, ports and behaviours are also seen as parallel processes which are managed by the TE. Basically, the ports, timers and behaviours belong to a component, which is an object global to them and contains some global information such as identifier, local verdict, etc.

In our implementation, the platform consists of the JVM (Java virtual machine) and the CORBA communication middleware included in Sun's Java 1.4 SDK. The EE running on that platform is the implementation of the test system entities. The CH, TM, CD, SA and PA are main entities providing the specified methods for communication, management, user interaction etc. They can be required by any built-in entity

of EE; EE is seen as a global entity. TE instantiates and manages the ports, timers and behaviours which are implemented as Java threads; at creation each entity knows the TE so that it may use the other test system entities CH, TM etc.

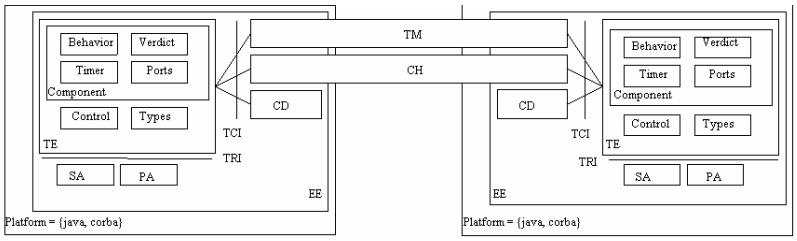

Fig. 3. Implementation view of a distributed TTCN-3 runtime environment. While the TTCN-3 execution environment is available on each test device (see [3]) only once, possibly distributed instances of the test management (TM) and the component handling (CH) entity exist.

As described in the general approach of the execution environment and as realised in the Java based implementation, behaviours are standalone processes running in parallel with other behaviours and/or entities. They may access ports or timers from the test component to which they belong.

An external behaviour is implemented by a different technology than the one used to implement the execution environment. However, the external component technology can be the same as the EE technology. Therefore, we use a *platform specific wrapper* (PSW) able to interconnect the external component in the TE with the external behaviour executing on the EB platform. An external component uses such a wrapper to communicate with the external behaviour but is also able to communicate with the other entities of the platform like a normal test component.

Fig. 4 shows an extension of the test system architecture where the external behaviours are deployed in a PSW (Platform Specific Wrapper).

An external component maintains the characteristics of a normal test component (management of ports and verdicts) and uses additionally the PSW. Depending on the external behaviour technology, the wrapper is more or less difficult to realise. The problem is basically to find a mechanism to communicate between an application written in the EE technology (in our case the PSW) and an application written in the EB technology (in our case the external behaviour). The interaction may be either synchronous or asynchronous. Some possible strategies are:

- Create the external behaviour as a process and use for communication the input, output or error buffers.
- Use sockets to communicate via ports. In this case, one may have to define an own protocol to establish the communication.
- Use native communication interfaces. For example, Java has the JNI (Java Native Interface) to communicate with C applications.
- Use a communication middleware like CORBA.

The external component may support several types of external behaviours: Perl, Python, etc. Therefore, a wrapper for each type of behaviour is needed. A component

decides which PSW to use depending on the type of behaviour that runs on it. Only one behaviour can run on the external component at once, thus, only one PSW is instantiated, namely the one associated to the type of the external behaviour.

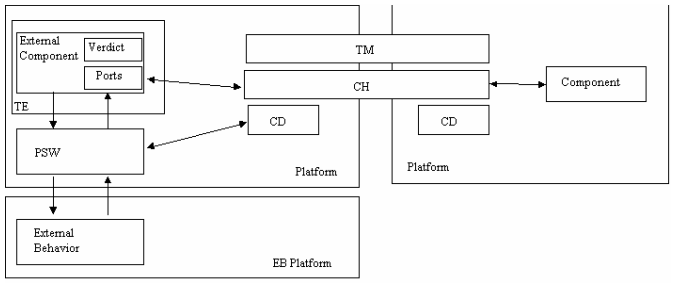

Fig. 4. Handling of external behaviours through PSW. An external behaviour runs on an external behaviour platform and interacts with the remaining test system via the PSW. The PSW is designed for a particular type of external behaviour platforms and not for a particular behaviour. It manages "incoming" CH communication and forwards the communication after translation into the external behaviour platform. Communication originating from the platform is mapped to the appropriate CH operations and thus forwarded into the TTCN-3 test system.

3.2 Operations with External Components

An external component is very similar to normal components. It can be created, can have ports, may execute external behaviours and can be stopped. In order to adhere to the TTCN-3 standard, all these operations should be compatible with the TCI interface specifications.

To be aligned with the TCI standard, an external component should be managed by CH. While the external component has still access to the native TTCN-3 data type representation (via the TCI value interface) the communication with the external behaviour has to be performed using encoded data. The CH should be able to handle an external component in the same way as it handles a normal component. Almost all operations, the CH applies to normal components (creation, start of a behaviour etc), are also applicable to external components. An external behaviour can access the component on which it runs, via the PSW, and therefore can conceptually access the local verdict and ports as well. The ports of the external components are used to realise the communication between an external behaviour and other TTCN-3 test components. Since the data format in the external behaviours is different from the TTCN-3 data, a codec is needed. This is the reason to establish a link between the PSW and the CD.

Fig. 4 shows how the external components are managed by CH and how they can interact with other external components or normal components.

Almost all TCI CH operations can be re-used for the management of external components and for the execution of external behaviour purposes; they only require

minimal extensions. Next, we describe some of the operations of CH and present how they are used to work with external components and behaviours. First, we consider the operations performed from a TE on another TE running an external component

To create an external component, the TE uses the `tciCreateTestComponentReq` method of CH. The CH checks if the component is an external one; if this is the case, the external component and a platform specific wrapper are created. The PSW is associated to the external behaviour type. The created external component functionality is seen as a normal component, since it provides the same interface to CH, TM etc. In a similar way, `tciStopTestComponent` is used when an external behaviour is stopped.

The ports of the external components are accessed via PSW from the external behaviour. This enables the external behaviour to use the ports to connect the external component ports with other components ports. The connection of two ports is realised through the `tciConnectReq` method. Since the ports are TTCN-3 ports (the external components use normal TTCN-3 Ports), the `tciConnectReq` implementation does not require further extensions. The `tciDisconnectReq` method behaves equally when it is used to disconnect a port.

To send data to a port of an external component, the `tciSendConnected` method of CH is used. CH accesses the TE of the external component and asks the PSW to deliver the data to the external behaviour. The same strategy applies for the `tciCallConnected` method.

To start an external behaviour, TE uses the `tciStartTestComponentReq`. The `TriComponentId` is the identifier of the external behaviour. The external behaviour is identified by the `TciBehaviourId` argument of the `tciStartTestComponentReq` method and can be started by parameters specified in the `TciParameterList`. Its implementation is complex, since it has to validate if the `TriComponentId` and the `TciBehaviourId` are external. If these values are not valid, error messages are returned. The external behaviours are started in the form of parallel processes and can be monitored or managed through PSW.

To check whether an external component is running, the `tciCreateTestComponentReq` is used. This method checks if the external process, used to start the external behaviour, is alive or not. The `tciTestComponentTerminatedReq` and `tciTestComponentDoneReq` work similarly.

Now we describe the operations an external behaviour can perform. To communicate with other test components, an external behaviour needs to access the ports of the external component where it runs. The external behaviour accesses the ports via PSW. Next, the PSW asks the external component to execute the requested operations which are: send, call etc. This is performed by using the CH methods: `tciSendConnectedReq`, `tciCallConnectedReq`.

The termination of the external behaviour is notified to the external component where it runs, which calls the `tciTestComponentTerminatedReq` method. This way, the verdict of the `ExternalBehaviour` is communicated to CH.

Some operations of TCI can not be called from the TE running external components since their corresponding operations are not possible:

- `tciCreateTestComponentReq`: an external behaviour cannot create other test components.

- `tciStartTestComponentReq`: an external behaviour cannot start behaviours on other test components.
- `tciExecuteTestCaseReq`: an external behaviour cannot start test cases.
- `tciMapReq`: the ports of an external component cannot be mapped to Test System Interface ports. The system component cannot be accessed by EB at all.
- `tciEncode`: there is no need that the EB uses the TCI codec as it uses a different type and value system.

3.3 Coupling with the SUT

External behaviours connect to the SUT by using proprietary technology specific mechanisms. An external behaviour should not be able to communicate to the SUT via the TTCN-3 TRI test adapter. The test adapter belongs to the TTCN-3 test system; it makes no sense that the external behaviours may access it. This is also underpinned by the fact that the external behaviours typically communicate already directly with the SUT which is simpler than going via TTCN-3. Basically, the EB does not/may not need a TRI adapter to communicate with the SUT. This is why we usually opt for using an external behaviour since it allows reusing existing assets for testing the SUT and often simplifies the communication with the SUT.

The stimulus and the received data are managed by the external behaviour itself. Beside the verdict (which can be returned to the test system as seen in previous chapter) other data can be transferred to the TTCN3 test system. The external behaviour runs on an external component whose ports can be accessed. If the external component has ports connected to other TTCN-3 test components, the external behaviour may use them to deliver data to the TTCN3 test system.

If transferring data from/to EB to/from the test system a codec is needed. This codec can be the default one provided by the platform specific wrapper or another one defined by the user.

4 An Example of Integrating Perl Scripts into a TTCN-3 Environment

In this section, we show how a possible application of the concepts introduced above might look like. For this, we are using a Perl script that performs an ftp-download as external test behaviour.

The following script defines a complete downloading procedure using Perl.

```
#!/usr/bin/perl -w
 use Net::FTP;
 my($ftp) ;
 $ftp = Net::FTP->new("a.server.com");
 die($ftp->message()) if( !($ftp->login("foo","bar")) );
 die($ftp->message()) if( !($ftp->binary()) );
 die($ftp->message()) if( !($ftp->get("AVeryBigFile.bin"")) );
 exit(0);
```

As it can be seen, the test behaviour is defined in a very compact way using only built-in operations available in Perl. This is typically what in practice is being encountered, simple scripts developed for a particular purpose without being designed for a particular test framework. The presented script shows also some very common scripting techniques. On success the scripts is executed without any output and returns with an exit code of zero. In any other case, some diagnostic output is provided and the script returns with a non-zero exit code.

The following TTCN-3 extract shows type definitions for defining the necessary external test component and behaviour in order to handle the above introduced Perl script.

```
// type definitions
type port STDERR message { out charstring ; }
type port STDIN  message { in charstring ; }

type external component ScriptComponent {
    port STDERR stderr ;
}

type component MainTestComponent {
    port STDIN  fromScript ;
}

external function scriptStart(charstring scriptName)
    runs on ScriptComponent ;
```

Basically, the fragment defines the environment of the script as having a single port, in this case the stderr buffer. Thus, the external behaviour, i.e. the Perl script, can communicate with other test components, in this case the main test component (MTC) via this port. The MTC has only a single port, which receives data that has been provided by the external component via its `stderr` port.

For starting the script, the external behaviour `scriptStart` has been declared. `scriptStart` runs on the `ScriptComponent`. Therefore, its ports can be connected to MTC ports.

The following TTCN-3 fragment shows how a functional test case can be defined.

```
// test case definitions
modulepar { charstring SCRIPT_NAME := "download.pl" ; }

testcase functionalTest() runs on MainTestComponent {
    var ScriptComponent p := ScriptComponent.create ;
    connect(mtc:fromScript, p:stderr) ;
    p.start(scriptStart(SCRIPT_NAME));
    var charstring logMessage ;
```

```
    alt {
      [] p.done { stop }
      [] fromScript.receive -> value logMessage {
       log(logMessage) ;
       if (lengthof(logMessage) != 0) setverdict(fail);
       repeat ;
      }
    }
}
```

The `functionalTest()` test case running on the `MainTestComponent` creates a `ScriptComponent` component and connects MTC's `fromScript` port with the `stderr` port of the script component. The external behaviour is started by executing TTCN-3 start command on the external component. The actual Perl script is referenced by the module parameter `SCRIPT_NAME`. After starting the external behaviour, the MTC waits for the termination of the external test component. The test case verdict is being determined by the exit code of the Perl script. For this, the PSW exports the script's stderr buffer to the `stderr` port as defined in the external component definition.

The presented functional test case can now be easily extended to a load test for an ftp server. The following TTCN-3 fragment shows an example for replicating the same external behaviour on numerous external test components.

```
testcase loadTest(integer load) runs on MainTestComponent {
    var integer i := 0 ;
    var ScriptComponent p ;
    for(i := 0 ; i < load ; i := i + 1 ) {
       p := ScriptComponent.create ;
       connect(mtc:fromScript, p:stderr) ;
       p.start(scriptStart(SCRIPT_NAME));
    }
    var charstring logMessage ;
    alt {
      [] all component.done { }
      [] any port.receive -> value logMessage {
       log(logMessage) ;
       repeat ;
      }
    }
}
```

This example shows how TTCN-3 can be used efficiently only for the instantiation of multiple scripts which are executed on many components. The code could also be written completely in Perl, but this would require the implementation of a distributed execution environment to be able to execute distributed test campaigns as described in [4]. For this, reusing existing TTCN-3 infrastructure for distributed test management can significantly reduce the complexity as the TTCN-3 execution environment hides the technical complexity.

```
control {
    if(execute(functionalTest(),2.0) == pass) {
       execute(loadTest(100), 10.0) ;
    }
}
```

This TTCN-3 code fragment displays further possibilities in order to reuse existing TTCN-3 infrastructure for the control of test campaigns.

5 Conclusions

This paper discussed the concepts of external behaviour and external component. External behaviours define non-TTCN-3 behaviours whose execution is controlled from TTCN-3. An external behaviour runs on an external component. Since the external component should not have access to timers or local variables, the concept of external component was introduced.

Extensions to existing TTCN-3 test system implementations to support external behaviours are possible along the standard TTCN-3 test system architecture. A common strategy of how to extend the actual TTCN-3 execution environments based on TCI was presented.

Our approach provides a generic adaptation method for running external behaviours. The adaptation is made in the TTCN-3 test system itself by means of technology-specific PSWs, which are reusable for any external behaviour provided in that technology.

Future work will demonstrate the applicability of our approach to cases where non-TTCN-3 applications different to Perl scripts will be included. A comparative study of the gain (in terms of development efforts and performance) in accessing external behaviours directly rather than to translate them into TTCN-3 will be done.

References

[1] ETSI ES 201 873 – 1, v2.2.1: "The Testing and Test Control Notation TTCN-3: Core Language ", Oct. 2002.

[2] ETSI ES 201 873-5 V1.1.1: "The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)", February 2003.

[3] ETSI ES 201 873-6 V1.0.0: "The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interfaces (TCI)", March 2003

[4] I. Schieferdecker, T.Vassiliou-Gioles: Realizing distributed test systems with TCI, IFIP 15th International Conference on Testing of Communicating Systems (TestCom 2003), Sophia-Antipolis (France), May 2003.

[5] The Perl directory: Online documentation. available at http://www.perl.org/docs.html