This regular paper was presented as part of the main technical program at IEEE ETFA'2011

Towards Model-Based Test Automation for Embedded Systems Using UML and UTP

Padma Iyenghar[1,2], Elke Pulvermueller[1] and Clemens Westerkamp[2]
[1]Software Engineering Research Group, University of Osnabueck, Germany
[2]Institute of Computer Engineering, UAS, Osnabrueck, Germany
piyengha@uos.de

Abstract

Model-based methodologies such as the Model-Driven Development (MDD) and Model-Based Testing (MBT) are being explored, in the recent decade, for automation in embedded software engineering projects. In this context, the target of this paper is to demonstrate the adoption and applicability of the Unified Modeling Language (UML) and the UML Testing Profile (UTP) for deploying MBT in Resource-Constrained (RC)-Real-Time Embedded Systems (RTES). Though the UTP standard has been introduced several years ago, concrete approaches or tool support for generating the test artifacts based on the UTP is currently unavailable. This paper aims to close this gap and discusses a concise set of UTP artifacts in the context of MBT for RC-RTES. A detailed discussion on the test artifacts generation algorithm demonstrating the applicability of our approach in a real-life RC-RTES example is presented.

I. Introduction

Embedded systems are ever-growing and find widespread usage in various appliances in our daily life. A majority of these embedded systems comprise of software of varying size and complexity. A big challenge for embedded software engineering is to address this growing variety and complexity of the software in the embedded system and ensure sufficient product quality. In order to achieve this, structured embedded software engineering and automation are inevitable. In this context, model-based methodologies (e.g. MDD and MBT), are being explored for automation in embedded software engineering projects. Adapting such automation methodologies are expected to improve the time-to-market constraints and reduce the cost of embedded software development and testing.

UML and UTP introduced by the Object Management Group (OMG) [1] are among the mature methodologies for MDD and MBT respectively. For instance, MDD (e.g. using UML) and MBT (e.g. using UTP) are being currently used as individual automation methodologies in the existing model-based approaches for embedded systems [2]. However, it is intuitive to perceive a clear advantage in using an integrated model-based approach combining MDD (e.g. using UML) and MBT (e.g. using UTP) for embedded software engineering projects. This is because, UTP being an extension of the UML for testing purposes, there is a possibility of a two fold advantage. For instance, more automation in these projects can be achieved and at the same time the training and other costs involved in adapting an integrated model-based approach for the project can be minimized.

Some existing tools [3] (which use UML and UTP for development and test automation respectively) support integrated model-based approaches for embedded software engineering projects. However, such tools and approaches are not necessarily applicable for RC-RTES, for e.g. 8 bit or 16 bit embedded systems with less than 64 Kbyte of memory. This is mainly because the existing tools (though they combine UML and UTP), for example [3], require downloading the (automatically) generated test harness in the RC-RTES for executing the test cases. Such tools, which involve dynamic source code instrumentation, generate a significant amount of test harness which increases with an increasing RTES application scenario and complexity [4]. Clearly, this is a disadvantage for applying such integrated model-based approaches for RC-RTES scenarios.

To address this gap, we proposed an integrated model-based test automation approach in [4]. In this approach, a generic test framework generation algorithm generates the necessary test artifacts (i.e. the test framework) for deploying MBT in RC-RTES. UML and UTP are made use of in the automatically generated test framework in this approach. However in [4] a detailed description of the procedure for the generation of the *UTP artifact* component in the test framework is not provided. Moreover, demonstrating the applicability of the UTP and its adoption in a real-life industrial RC-RTES project is missing.

The aim of this paper is to present a detailed description of the *UTP artifact* generation procedure, based

978-1-4577-0018-7/11/$26.00

on the chosen System Under Test (SUT) and the design model. A precise set of the the elements from the UTP concept group is defined and implemented for the *UTP artifact* generation. We demonstrate the applicability of the *UTP artifact* in aiding the test case execution process while deploying MBT in RC-RTES. Illustrative examples based on a real-life RC-RTES project are presented.

This paper is outlined as follows. Section II presents related work. Section III provides a background on the UTP and introduces a RC-RTES example. Section IV introduces our integrated model-based approach for deploying MBT in RC-RTES. Section V discusses the test framework generation algorithm. Section VI presents a discussion and conclusion.

II. Related Work

Related research work and tool support pertaining to automation using MDD (section II.A), MBT (section II.B) and UTP (section II.C) in the context of RTES is discussed below.

A. MDD using UML

Applicability and usage of model-based automation approaches such as the MDD and MBT in embedded software engineering has been gaining significant attention in the recent past. Towards this direction, research on using UML for MDD [5] [6] and its applicability for RTES has been widely reported [7] [8] [9] [10].

Some of the tools currently available for supporting the MDD phase, in general, are [11], [12], [13], [14], [15], [16], [17] and [18]. Features supported in these tools include modeling using UML, code generation with/without platform specific inputs, support for implementation of a specific UML-profile, etc. Commercially available tools such as [11], [12] are some of the tools supporting MDD for RTES.

B. MBT

The MBT process itself is divided into the following steps in the literature [19]: (a) Modeling the System Under Test (SUT) and/or its environment, (b) Generating abstract tests from the model, (c) Executing the tests on the SUT and (d) Assigning verdicts and analyzing the test results.

Based on these steps, various MBT methodologies have been reported for different application domains [20] [21]. A number of MBT approaches discussed in the literature concentrate on generating test cases using models extracted from software artifacts [22] [2], i.e. step (b), while a few approaches concentrate on steps (a) and (d) [21].

Similarly, there are several MBT tools [23] which support steps (a), (b) and (d). However, concrete research work or tool support (e.g. using UML/UTP) for deploying MBT (i.e. step (c)) is missing not only in the literature but also in commercially developed MBT tools. This is especially true for RC-RTES.

C. UTP and MBT

In order to support UML with testing related activities (i.e. in the MBT phase), the UTP standard has been introduced by the OMG (in the year 2004-2005). Fundamentally, UTP provides a basis for systematic testing and integration particularly in UML-based development environments. However, unlike the extensive research for using UML in the MDD phase, especially for RTES, applicability of UTP and its usage for RTES is still an emerging technology. A significant reasoning for this could be based on the role of UTP. For instance, the UTP is used only to specify explicitly a typical architecture that can be associated with MBT, while the MBT framework automates the testing process. This implies that the UTP as such does not specify how to carry out the testing process [24]. Hence, there are several MBT approaches for automating the testing process and only a very few of them concentrating on the applicability and specification of the test framework using the UTP.

Nevertheless, some studies based on the applicability of UTP for MBT is available in [24] and [25]. However, these studies do not deal with the generation of a integrated UML and UTP based test framework for deploying MBT in RC-RTES. Similarly, the tool support for UTP is also very limited. For instance, to our knowledge, an add-on [3] for MDD tool [12] is the only available MBT tool which supports development of UTP-based test infrastructure, especially for RTES.

Surveys on MBT approaches conducted in [2] [22], identify the advantages and pitfalls in the existing MBT techniques. Based on the conclusions in [22], it is clear that the MBT approaches are usually not integrated with the software development process, for instance the MDD phase. It is also clear that there is a need for integration tools which combine both the MDD and MBT phases for software engineering projects. For example, though tools such as [12] (e.g. using UML for MDD) and [3] (e.g. using UTP for MBT) together could provide integrated model-based approach for RTES scenarios. However, currently they do not provide support for deploying MBT in RC-RTES.

Thus it is evident that not only UTP-based test framework development for MBT is missing but also, integrated model-based approach and test automation (combining UML and UTP) for deploying MBT in RC-RTES is also missing in the existing approaches. In this paper we aim to address this gap, especially in the context of deploying MBT in RC-RTES.

III. Background

In section III.A, a brief introduction to the UTP and its terminology is provided. In section III.B the concise set of UTP elements used in our approach is discussed. In section III.C, a running example based on a real-life industrial example of a spark extinguishing system is introduced.

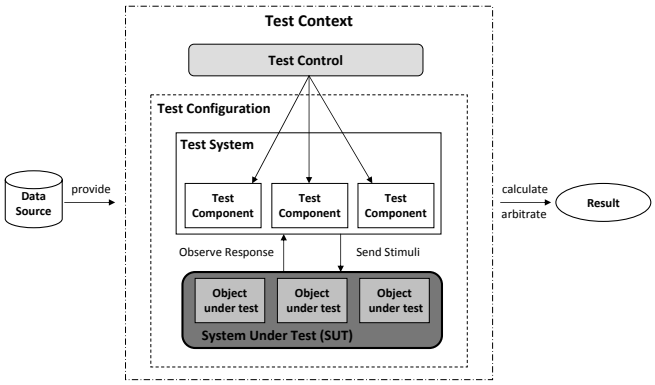

Fig. 1. General schema of black box testing and UTP terminology

A. UML Testing Profile (UTP)

The UTP provides concepts that target the pragmatic development of test specifications and test models for black-box testing [26] [27], where the internal events of the SUT are kept hidden. The UTP can be used stand alone for handling the test artifacts or in an integrated manner with UML for handling the system and test artifacts together. The profile is based on UML 2.0 and extends its meta-model by the stereotype extensibility mechanism.

The profile introduces concept groups namely, *test architecture*, *test behavior*, *test data* and *test time* [1] [27]. The entire set of UTP stereotypes for these concept groups is shown in Table I. The *test architecture* group provides a set of concepts to specify the structural aspects of a test context covering the test components, the SUT, their configuration, etc. A mapping of the UTP elements to the general schema of black-box testing [27] and UTP terminology is shown in Fig. 1. The *test behavior* group provides concepts to specify test behavior, their objectives and the evaluation of the SUT. The *test data* concept group is used to specify data used as stimuli (to the SUT). The time constraints and observations can be specified using concepts in the *test time* group in UTP. Together, these concepts define a modeling language for visualizing, specifying, analyzing, constructing and documenting the artifacts of a test system [26] [1].

B. A concise subset of the UTP elements

Our approach deals with automatically generating the required "test framework infrastructure" and the architectural support at the host side for executing MBT in RC-RTES. Hence, we choose a concise subset of the UTP components (among the extensive set-seen in Table I) from the *test architecture* concept group in UTP for our approach. This is because, the *test architecture* concept group provides concepts and stereotypes for specifying the "structural" aspects of a test context.

The elements *test context, test configuration, test component* and *SUT* from the *test architecture* concept group (refer to elements marked with '$*$' in Table I) are chosen to specify the *UTP artifact* in the test framework infrastructure in our approach. The test cases are specified in the test context based on UTP. In UTP terminology, the *test cases* elements are defined in the *test behavior* concept group (marked with '‡' symbol in Table I). These five UTP elements (marked with '$*$' and '‡' in Table I) constitute our concise subset among the wide variety of the available components in UTP as seen in Table I. The applicabiltiy of the three other elements in the test architecture concept group (marked with '$\circ$' symbol in Table I) will be explored in our future work.

1) Definition and derivation: As a base for the detailed description of our approach and the test framework generation algorithm discussed in section IV, the definitions of the concise set of UTP elements (marked with '$*$' and '‡' in Table I) is provided in this section. Their applicability and derivation[1] based on the system design model (in our proposed approach) is also explained below.

- The *SUT* element is a part and is the system, subsystem or the component being tested. The SUT is a black-box and it is exercised via its public interface operations. In our approach, any UML class element in the design model of a given project can be chosen as the SUT and provided as input for the test framework generation algorithm (discussed in section IV). Based on the chosen SUT a test framework is created in our approach. For the chosen UML class element in the system design model, the corresponding *SUT* (UML) element in the test framework is implemented by creating a new corresponding class element and stereotyping with $\ll SUT \gg$.
- The *test component* object is a class of a test system with a set of interfaces with which it may communicate with other test components or with the SUT. In our approach, we define an element *test driver* of type *test component*, which is used to initiate the test case execution process. This is implemented by creating a new class and assigning it to *test component* class by stereotyping with $\ll TestComponent \gg$.
- The collection of test component objects and the connections between the test component objects and the SUT is termed as *test configuration*. In our approach, a new *test configuration* element *Proxy_Config* is created based on the relevant interfaces between the test components and the SUT. It is stereotyped with $\ll TestConfiguration \gg$.
- A *test case* is a complete technical specification of how the SUT should be tested for a given test

[1]Note that a common methodology for making use of a UML-based profile is to implement the profile as a set of stereotypes (e.g. in MDD/MBT tool), which can be included in a given project. Then, the required/specific elements of the profile (e.g. UTP) can be made use of for a given UML element. This can be achieved by specifying the UML element with a particular stereotype (from the profile). We adhere to this method in our approach.

TABLE I. UTP concept groups and stereotypes

Test Architecture	Test Behavior	Test Data	Test Time
SUT ∗	Test objective	Data pool	Timer
Test component ∗	Test case ‡	Data partition	Time zone
Test context ∗	Defaults	Data selector	
Test configuration ∗	Validation action	Wild cards	
Test control ○	Verdicts	Coding rules	
Arbiter ○			
Scheduler ○			

LEGEND:
∗ and ‡ : Concise set of UTP elements in our approach
∗ : Generated by our approach. ‡ : Specified manually in our approach
○ : Under consideration for future work

Fig. 2. *OperatingState_ShutDown* **class is the chosen SUT**

objective. In our approach, the test cases are derived based on the system behavior defined in the design model and stereotyping with $\ll TestCase \gg$.

- The collection of test cases together with a test configuration on the basis of which the test cases are executed is termed as the *test context*. In our approach this is implemented as a UML class element and stereotyping with $\ll TestContext \gg$.

In our approach, the elements *test component*, *test configuration*, *SUT* and *test driver* in the *UTP artifact* component of the test framework are generated automatically by the test framework generation algorithm (discussed in section IV). However, the *test cases* are specified by the user manually in the generated *UTP artifact* component of the test framework.

C. Real-life RC-RTES example

In this section we introduce a brief example based on a real-life industrial RTES project of a spark extinguishing system [28]. This example will be used to demonstrate the applicability of our approach and the test framework generation algorithm in the next sections. A spark extinguishing system consists of many spark detectors [28] centrally connected to fire extinguishing machine(s). The control console consists of the RTES and comprises of a resource-constrained 16-bit microcontroller [29]. It performs the analysis and processing of the signals received from various spark detectors. Among other functionalities, the control console handles events such as alarm, shutdown and faults.

The functionality for the shutdown state is collectively implemented as a UML class element named *OperatingState_ShutDown* (Fig. 2). The association

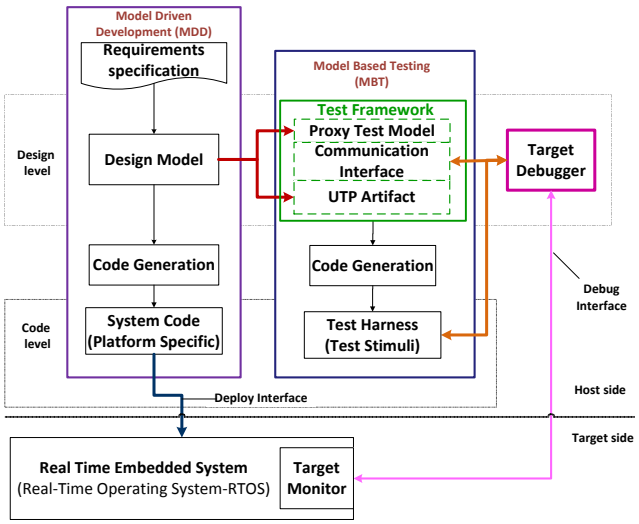

Fig. 3. Model-Based Test Framework

ends of this class are *itsControlDisplay*, *itsControlOf_OperatingStates* and *itsEventShutDown* as seen in Fig. 2. These associations denote the relationship between the *OperatingState_ShutDown* class and the *ControlDisplay*, *ControlOf_OperatingStates* and *EventShutDown* classes respectively. The detailed functionality of the *OperatingState_ShutDown* class is implemented as a UML state chart element. This state chart is reached whenever a shutdown event is triggered (e.g. by external analog input).

IV. Integrated model-based approach

The integrated model-based architecture and test framework for online MBT (introduced in [4]) in RC-RTES is shown in Fig.3. As seen in Fig. 3, the two phases for RTES development and testing in our approach are the MDD and MBT phases respectively. As a base for the test framework generation algorithm discussed in the next section, the major elements of this approach are briefly summarized below.

A. MDD phase

In the first step of the MDD phase, the requirements are specified as design model using UML. In order to derive a test framework automatically (MBT phase),

a pre-defined system design model is necessary. The system design model provides the obligatory information for the derivation of the test framework skeleton. Based on the design model, the platform specific system code is obtained using an automatic code generation process. A MDD tool [12] is used for specifying the design model using UML as well as for the automatic code generation process in our prototype.

During the compilation of the system code, an XML file is created with detailed data about the target (e.g. endianness, etc) and the developed embedded software [30]. The system code is executed on the RTES which runs a Real-Time Operating System (RTOS) bundled with a target monitor (implemented in the programming language "C"). The real-time execution framework (RXF) from [31] is used as the underlying RTOS framework in our prototype. The purpose of the target monitor is to send notifications about the target behavior to the host side using a debug communication interface [30]. A target debugger (implemented in the User Interface framework Qt [32]) decodes the received trace data from the target monitor (with the aid of the XML file). The target debugger aids in communicating the test data between the test framework and the embedded target system.

B. MBT Phase

During the MBT phase, given the design model and the chosen SUT, a test framework is generated for deploying MBT in the RTES. The test framework comprises of three components, namely: (a) *proxy test model* (b) *communication interface* and (c) the *UTP artifact*.

The *proxy test model* component comprises of a corresponding test module (e.g. class) in reference with each module (e.g. class) in the system design model. The *proxy test model* consists of only an abstract design model with no functionality inherited from the system design model. The *proxy test model* (on the host side at the design level) is used to mirror the test case execution process at the RTES. It also consists of the necessary infrastructure to convey the test data to the target via the target debugger and the target monitor as shown in Fig. 3. The test data is conveyed using a pre-defined format namely *<(event) (source) (destination) (parameters)>*. The test data is conveyed to the target debugger with the help of functions in the *communication interface* component of the test framework.

The *communication interface* component aids in communicating (bi-directional) the test data, test results, etc between the test framework and the target debugger. This is implemented as a TCP/IP communication interface in our prototype.

The *UTP artifact* component is based on the concise set of UTP elements defined in section II.A. It consists of the *SUT* under consideration, *test driver* to drive the SUT, *test cases*, *test context* and *test configuration*. In addition to these five elements based on UTP, two more components based on UML elements such as *links* and

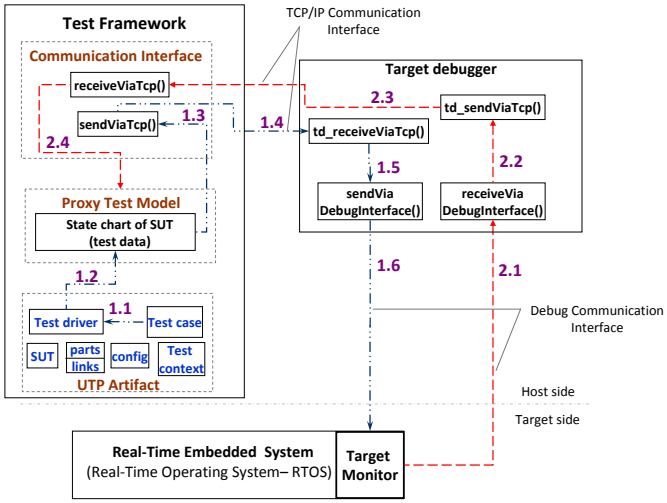

Fig. 4. Steps involved in deploying MBT

parts (instance of type class, i.e instances of associated classes of SUT) are created in the *test context* element in the *UTP artifact*. The *links* and *parts*, in addition to the UTP-based elements created in the *UTP artifact*, are used to aid the test case execution process.

C. Deploying MBT in RC-RTES

Once the test framework is generated (by the algorithm discussed in section V), deploying MBT in RC-RTES using our approach and these UTP elements consists of a series of steps as illustrated in Fig. 4. These steps are outlined below.

- The *test cases* (e.g. sequence diagram test case) are specified by the user in the MBT tool [3] (in which the test framework is generated) (step 1.1).
- The *test driver* initiates the test case execution process, thereby injecting an event to trigger the *SUT* (step 1.2).
- On receiving the event injected by the *test driver*, the state chart of the SUT (in the *proxy test model*) invokes the respective functions to communicate with the target debugger via the TCP/IP communication interface (step 1.3 in Fig. 4.).
- The test data is conveyed to the RTES by the target monitor via the target debugger (step 1.4, 1.5 and 1.6). Thus the test stimuli is executed on the RTES.
- The target monitor sends the test case execution results to the target debugger (via the debug interface) (step 2.1).
- The target debugger conveys (step 2.2 and 2.3) the test data to the corresponding decoding function in the test framework. The decoding function interprets the test results.
- With the aid of the *proxy test model* in the test framework, the test results are visualized (using UML diagrams) in the MBT tool dynamically, in real-time (step 2.4).

Thus, the test cases can be executed in the RTES (with the aid of the UTP artifact and test framework) without downloading significant amount of test harness

on the RTES. The test results are visualized using UML diagrams in the *proxy test model* of the test framework.

V. Algorithm for Test Framework Generation

The test framework generation algorithm (shown in algorithm 1) is implemented and integrated in an MBT tool [3] with the aid of the APIs [33] in the MBT tool. From Fig. 3. it is seen that the proposed test framework comprises of three components, namely, the *proxy test model*, *communication interface* and the *UTP artifact*. The test framework generated for the running example (introduced in section II.C and Fig. 2.) is shown in Fig. 5. The blocks (a), (b) and (c) in Fig. 5. denote the *proxy test model*, *communication interface* and the *UTP artifact* components, respectively, generated by this algorithm for the running example (shown in Fig. 2).

Algorithm 1 takes as input the given system design model and the chosen SUT (selected by the user). The output of the algorithm (algorithm 1) is the generated test framework (comprising of the *proxy test model*, *communication interface* and the *UTP artifact* components) for the chosen SUT.

Algorithm 1 along with the test framework generated for the running example (shown in Fig. 5, where the chosen SUT is *OperatingState_ShutDown*) is described below. Section V.A briefly describes the generation of the *proxy test model* and the *communication interface* components. Section V.B describes the generation of the *UTP artifact* component in detail.

A. Generation of 'proxy test model' and 'communication interface' components (lines:1-15 in algorithm 1)

The first step of the algorithm determines if the given UML package element *Design_Model_Pkg* representing the design model is empty (line:1). If it is non-empty, then the algorithm creates a test framework package *Test_Framework_Pkg* for the chosen SUT (lines:1-2). If it is empty, then the algorithm skips further processing (lines:2-29) and returns (lines:30-32). For our example, since the given design model package is non-empty (Fig. 2), the algorithm creates a test package *TestPkg_OperatingState_ShutDown_proxy* (lines:1-2) as shown in Fig. 5. and proceeds with further processing.

- Lines:3-14 of the algorithm 1 specify the procedure to create the *proxy test model* component of the test framework. The input parameters for this procedure are the SUT under consideration and the newly created test framework package *Test_Framework_Pkg*. This procedure (lines:3-14) creates a proxy class for every class in the association ends of a chosen SUT (lines:4-7). The proxy test model also comprises of the infrastructure to handle the test data of the SUT (in its state chart). This is created by invoking a procedure *CreateSUTStatechart* for the chosen SUT as seen in lines:8-12 of algorithm 1.
- For our example, the created *proxy test model* (with proxy classes) for the SUT *OperatingState_ShutDown* is shown in block (a) in Fig. 5. The proxy classes for the given SUT is created based on the participating classes in each association end of the chosen SUT (Fig. 2). Hence, as seen in Fig. 5, the *proxy test model* for our chosen SUT consists

Algorithm 1 : Test Framework Generation

Input: System design model and SUT

Output: Test framework={*proxy test model*, *communication interface*, *UTP artifact*}

```
1: if Design_Model_Pkg is not empty then
2:   Create Test_Framework_Pkg
3:   for Every association in the given SUT do
4:     Determine the class(es) in the association end
5:     for Every class in association end do
6:       Create class_proxy in Test_Framework_Pkg
7:       Copy Attributes, Associations, Dependencies
         and Operations from class to the corresponding class_proxy
8:       if (class==SUT) then
9:         call procedure CreateSUTStatechart()
10:      else
11:        Create empty state chart in class_proxy
12:      end if
13:    end for
14:  end for
15:  call procedure Create_CommunicationInterface()
16:  if Test_Framework_Pkg is not empty then
17:    Create a test context element, TestContext=TCon_<SUT name>_proxy
18:    Create instanceofSUT inside TestContext
19:    Create a test configuration element named Proxy_Config in TestContext.
20:    Create an element TestDriver in TestContext. ;instance of type TestComponent
21:    Add an association end from TestDriver to instanceofSUT in TestContext.
22:    for Each association end in the SUT do
23:      Create an UML link element between the associated proxy classes
24:      Create instances of proxy classes ;referred as parts
25:    end for
26:  else
27:    exit ;UTP artifact component was not created
28:  end if
29:  return Test Framework
30: else
31:  exit ;Test framework was not created
32: end if
```

of the proxy classes *ControlDisplay_proxy*, *ControlOf_OperatingStates_proxy*, *EventShutDown_proxy* and *OperatingState_ShutDown_proxy*. These classes are the corresponding proxy elements of the participating classes *ControlDisplay*, *ControlOf_OperatingStates*, *EventShutDown* and *OperatingState_ShutDown* in the association(s) of the chosen SUT in the design model (Fig. 2).
Note that a detailed description of the *proxy test model* component creation (lines:5-13) and the *CreateSUTStatechart* (lines:8-12) procedure is beyond the scope of the paper. This is not only because of space constraints, but also because the main scope/aim of the paper is to concentrate on the applicability of the UTP for test automation. Hence, only the *UTP artifact* generation procedure is discussed in detail (in section V.B).

- The *communication interface* component for the test framework is created by invoking the procedure *Create_CommunicationInterface* (line:15). The newly created test framework package *Test_Framework_Pkg* and the chosen SUT are the parameters for this procedure. In our prototype, this is created as a TCP/IP communication interface component in the test framework package. This is shown in block (b) in Fig. 5.

B. 'UTP artifact' component generation (lines:16-28 in algorithm 1, e.g. block(c) in Fig. 5).

The *UTP artifact* component which aids in the test case execution process is created in the next steps (lines:16-28) of the algorithm 1. This procedure takes as input the newly created test framework package and the chosen SUT.

The *UTP artifact* generated by the procedure in lines:16-28, comprises of the four UTP components (described in section II.B and marked with '$*$' symbol in Table I) namely, the *TestContext*, *instanceofSUT*, *Proxy_Config* and *TestDriver*. The fifth element namely the *test cases* for the *UTP artifact* is not generated by this algorithm. This is specified by the user manually in the created test framework. Two more components based on UML elements such as *links* and instances of type class (*parts*) are created in the *TestContext*. The steps involved in the creation of the *UTP artifact* component of the test framework (lines:16-28 of algorithm 1) is described below.

- The *UTP artifact* generation procedure (lines:16-28) proceeds with further processing after verifying if the contents of the newly created test framework package (in line:2 of algorithm 1) is non-empty. This is because the *UTP artifact* generation procedure is invoked after the creation of the *proxy test model* component and the *communication interface* component. Hence, the *Test_Framework_Pkg* is expected to be non-empty. If *Test_Framework_Pkg* is non-

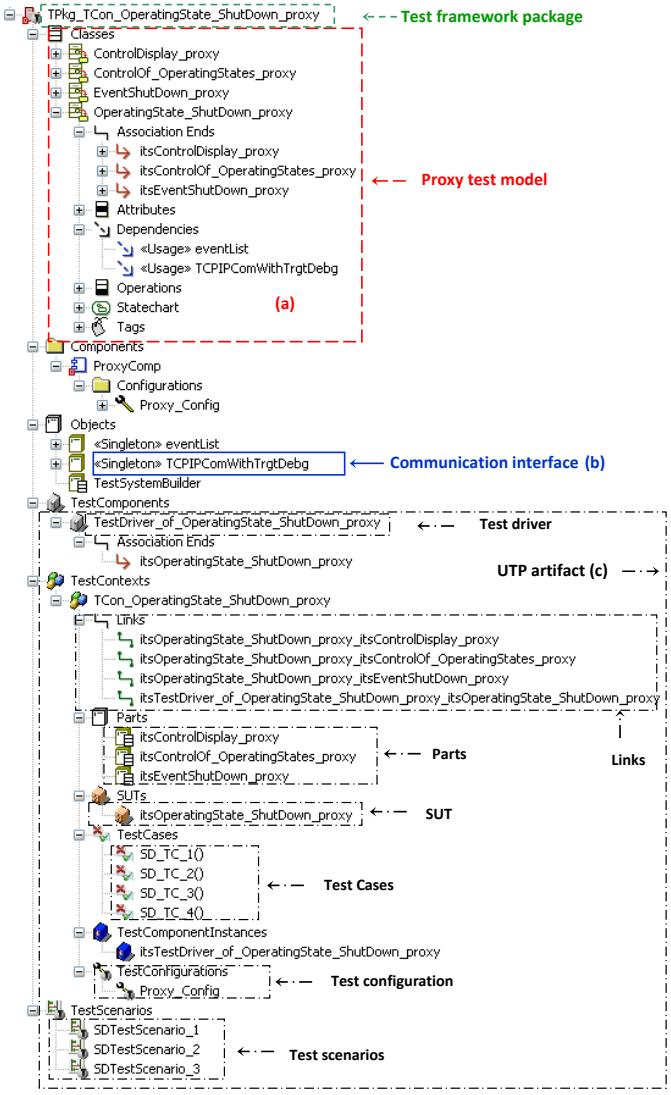

Fig. 5. Test framework generated for the SUT *OperatingState_ShutDown*

empty (line:16), then the *UTP artifact* is created based on the procedure in lines:17-25. If it is empty, then the algorithm skips further processing (lines:17-24) and returns (lines:26-28).
In our running example, for the chosen SUT *OperatingState_ShutDown* the test framework package *TPkg_TCon_OperatingState_ShutDown_proxy* is non-empty, i.e. it consists of the *proxy test model* and the *TCP/IP communication* interface already created by the test framework generation algorithm in lines:1-15 (e.g. blocks (a), (b) in Fig. 5.). Hence, the procedure for creating the *UTP artifact* component in lines:15-25 is processed.

- In line:17, a test context element *TestContext=TCon_<SUT name>_proxy* is created. For the running example, a test context *TCon_OperatingState_ShutDown_proxy* is created with reference to the chosen SUT *OperatingState_ShutDown*.

- Inside the newly created *TestContext*, the following elements are created for the *UTP artifact* (lines:18-21).
  - An instance of the SUT (instance of type class) *instanceofSUT* is created (line:18). This will be used while configuring and executing the test cases inside the test context element. In our example, for the chosen SUT *OperatingState_ShutDown*, an instance of the SUT namely *itsOperatingState_ShutDown_proxy* is created as seen in Fig. 5.
  - A UTP *test configuration* element *Proxy_Config* is created in the next step (line:19). This element is used to define the connections between the test component objects and the SUT. An initial test configuration created in *Proxy_Config* can be altered by the user if required during later stages (before the test execution process). For our example, the *Proxy_Config* element created for the chosen SUT and *TestContext* as seen in Fig. 5.
  - To initiate the test case execution process, a *test driver* is created (as an instance of UTP element type *Test Component*-refer to section II.B) in line:20. For our example, the test driver *itsTestDriver_of_OperatingState_ShutDown_proxy* is created as seen in Fig. 5. An association is created between the newly created *test driver* and the SUT element in line:21 of the algorithm. This enables the test driver to initiate the test case execution process for the chosen SUT. For our example, this is shown as an association end in the test driver element of the *UTP artifact* (refer to block (c) in Fig. 5.).

  From the above description is it seen that the elements *TestContext*, *instanceofSUT*, *Proxy_Config*, *Test Component* are created based on the following elements from our concise set of UTP, namely, *test context*, *SUT*, *test configuration* and *test component* respectively. These elements are derived based on our concise set of UTP elements from the *test architecture* concept group as described in section III.B (also refer to Table I). The purpose and applicability of these elements and their derivation based on the system design model is also provided in the above discussion.
- In line:22, the association end(s) of the SUT are determined. For each association end in the SUT a UML *link* element is created between the associated classes in the *UTP artifact* (line:23). Also, a link element is created between the *TestDriver* of the SUT and the SUT under consideration.
  Based on the associated classes for the given SUT, instance of these classes or *parts* are created (line:23) inside the *TestContext* element. These *links* and *parts* (based on UML), in addition to the UTP-based elements created in the *UTP artifact* are used to aid the test case execution process.
  For the chosen SUT *OperatingState_ShutDown*, the UML link elements created (line:23) based on the association ends of the SUT are shown in block (c) in Fig. 5.

Once the algorithm completes the generation of these components for the *UTP artifact*, the algorithm terminates and returns (line:29). For our example and the chosen SUT *OperatingState_ShutDown*, the test framework comprising of the *proxy test model* (Fig. 5-(a)), *communication interface* (Fig. 5-(b)) and the *UTP artifact* (Fig. 5-(c)) generated by this algorithm upon its termination in the MBT tool [3] is shown in Fig. 5.

VI. Discussion and Conclusion

A. Discussion

Based on the steps described in section V, it is clear that in our approach a test framework architecture is derived based on the system design model and implemented primarily using UML and a concise set of elements (section III.B) from the UTP concept groups. Once the test framework is generated by the algorithm in the MBT tool [3], the *test cases* are specified by the user in the test framework. The *test driver* element of the UTP artifact (generated by the algorithm) initiates the test case execution process as seen in Fig.4.

The steps involved in deploying MBT in RC-RTES with the aid of the generated test framework is discussed in section III.C. The steps illustrated in Fig.4. elaborate the test case execution process using our proposed approach. An important point to note is that this test framework is not used for automatically generating the test cases (e.g. [21]) during the MBT phase.

Based on the description of the UTP (in section II and III), it is seen that the UTP standard [1] specifies a wide variety of test modeling concepts (refer to Table I) by extending UML with stereotype extensibility mechanism. In this paper we make use of a concise set of the UTP concept group elements and arrive at a *UTP artifact* component in the automatically generated test framework. This component aids in mirroring the test case execution process (and results) on the host side/design level based on the RC-RTES behavior (illustrated in Fig.4). Thus, we demonstrate the applicability of UTP (based on our chosen concise set of UTP-elements) for test automation in RC-RTES.

In our prototype, this generic test framework generation algorithm (shown in algorithm 1) is implemented and integrated in the MBT tool [3] with the aid of the APIs [33] in the MBT tool. By this, our approach and our prototype thus also demonstrates step (c) in the MBT process and closes this gap especially for RC-RTES scenarios.

B. Conclusion

Even though the UTP was introduced by the OMG several years ago, tool support for generating the test artifacts based on the UTP is currently very limited. Similarly, an integrated model-based approach combining both MDD and MBT, especially in the context of RC-RTES is still missing. In this paper, we discuss an integrated model-based approach and test framework for deploying MBT in RC-RTES. We describe a test framework generation algorithm aiming towards model-based test automation using UML and a concise set of UTP elements for RC-RTES. We demonstrate the applicability of the UTP in the context of deploying MBT for RC-RTES scenarios. We describe the derivation of the UTP artifacts based on the given system design model for our approach. Illustrative examples from a real-life industrial RC-RTES scenario are presented.

Implementation/support for the discussed approach in open source tools (e.g. [18]) could be carried out in the future. Further, to support domains other than the RTES using our approach, an extensible and generic wrapper class/framework can be developed.

References

[1] Object Management Group. http://www.omg.org, Jul. 2011.

[2] M. Mussa, S. Ouchani, W. Al Sammane, and A. Hamou-Lhadj, "A survey of model-driven testing techniques," in *9th International Conference on Quality Software*, 2009, pp. 167 –172.

[3] IBM Rational Test Conductor Add-on. http://www.btc-es.de/, Jul. 2011.

[4] P. Iyenghar, "Test framework generation for model-based testing in embedded systems," in Proceedings of 37th Euromicro conference on Software Engineering and Advanced Applications, SEAA 2011, to Appear.

[5] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145 – 164, Jan. 2003.

[6] R. B. France, S. Ghosh, T.D-Trong, and A. Solberg, "Model-Driven Development Using UML 2.0: Promises and Pitfalls," pp. 59–66, 2006.

[7] C. Shih, C.-T. Wu, C.-Y. Lin, P.-A. Hsiung, N.-L. Hsueh, C.-H. Chang, C.-S. Koong, and W. Chu, "A model-driven multicore software development environment for embedded system," ser. 33rd Annual IEEE International Conference on Computer Software and Applications, 2009. COMPSAC '09., vol. 2, july 2009, pp. 261 –268.

[8] H. Kashif, M. Mostafa, H. Shokry, and S. Hammad, "Model-based embedded software development flow," in *4th International Design and Test Workshop (IDT)*, November 2009, pp. 1 –4.

[9] C. Bunse, H.-G. Gross, and C. Peper, "Applying a model-based approach for embedded system development," in *33rd EUROMICRO Conference onSoftware Engineering and Advanced Applications.*, 2007.

[10] P. Kukkala, J. Riihimaki, M. Hannikainen, T. D. Hamalainen, and K. Kronlof, "UML 2.0 Profile for Embedded System Design," in *Design, Automation and Test in Europe*, 2005, pp. 710–715.

[11] Enterprise Architect Tool. http://www.sparxsystems.com/, Jul. 2011.

[12] IBM Rational Rhapsody Developer, version 7.5.1. http://www.ibm.com/software/awdtools/rhapsody/, Jul. 2011.

[13] Altova UML Tool. http://www.altova.com/, Jul. 2011.

[14] Visual Paradigm Tool. http://www.visual-paradigm.com/, Jul. 2011.

[15] Magic Draw UML Tool. http://www.magicdraw.com/, Jul. 2011.

[16] Star UML Tool. http://staruml.sourceforge.net/en/, Jul. 2011.

[17] BridgePoint UML Tool. http://www.mentor.com/, Jul. 2011.

[18] Papyrus UML Tool. http://www.papyrusuml.org/, Jul. 2011.

[19] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[20] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 285–294.

[21] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 85–103.

[22] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, ser. WEASELTech '07. New York, NY, USA: ACM, 2007, pp. 31–36.

[23] List of commercially available MBT Tools. http://www.cs.waikato.ac.nz/research/mbt/Tools.pdf, Jul. 2011.

[24] P. Krishnan and P. Pari-Salas, "Model-Based Testing and the UML Testing Profile," in *Semantics and Algebraic Specification*, ser. LNCS, J. Palsberg, Ed. Springer, 2009, vol. 5700, pp. 315–328.

[25] M. Mlynarski, B. Gueldali, M. Spaeth, and G. Engels, "From design models to test models by means of test ideas," in *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, ser. MoDeVVa '09. NY, USA: ACM, 2009, pp. 7:1–7:10.

[26] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[27] Z. R. Dai, "An Approach to Model-Driven Testing - Functional and Real-Time Testing with UML 2.0, U2TP and TTCN-3." Ph.D. dissertation, 2006.

[28] Development of a central spark extinghusing system. http://www-05.ibm.com/de/events/innovate/pdf/Entwicklung-einer-GreCon-Funkenloeschanlage-fuer-die-Holzindustrie-mit-Rhapsody-von-der-Heide-Boerries.pdf, Jul. 2011.

[29] Embedded development tools. http://www.keil.com/, Jul. 2011.

[30] P. Iyenghar, C. Westerkamp, J. Wuebbelmann, and E. Pulvermueller, "A model based approach for debugging embedded systems in real-time," in *Proceedings of the tenth ACM international conference on Embedded software*, ser. EMSOFT '10. NY, USA: ACM, 2010, pp. 69–78.

[31] Willert Software Tools GmbH. http://www.willert.de/, Jul. 2011.

[32] User Interface Framework-Qt. http://qt.nokia.com/, Jul. 2011.

[33] Rational Rhapsody API Reference Manual. http://www.ibm.com, Jul. 2011.