Using a New UML Profile for Modeling Software Tests

Andrew Diniz da Costa, Carlos José Pereira de Lucena, Ricardo Venieris, Gustavo Carvalho
Laboratory of Software Engineering
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
{acosta, lucena}@inf.puc-rio.br, {rvenieris, guga}@les.inf.puc-rio.br

***Abstract*—The development of complex systems is becoming extremely common; hence, is motivating the work on software testing. When a large number of tests must be executed to validate the release of a system, several data should be used to correctly coordinate the execution of these tests, such as knowing (i) if the current version of a particular test has been updated, (ii) the interdependence between tests, (iii) the order of execution to be followed, (iv) the priority, (v) the risks associated with the tests, etc. Based on this concern for providing and documenting useful data for the coordination of test execution, this paper offers a new modeling language called UML Testing Profile for Coordination (UTP-C). UTP-C was created from testing experiences of several web and desktop applications in the Software Engineering Lab, located at the Pontifical Catholic University of Rio de Janeiro. In order to illustrate the use of UTP-C, the paper presents tests modeled for validating an e-commerce multi-agent system.**

Keywords-UML testing profile; model based test; software testing.

I. INTRODUCTION

Creating and executing software tests is an activity that is extremely important in the development process. Depending on the size and complexity of the system evaluated, System Under Test (SUT), a large number of tests should be created and maintained. The U.S. National Institute of Standards and Technology (NIST) informs that systems without adequate tests generate annual costs of up to US$ 59.5 billion [24]. This is almost 1% of the gross domestic product of the U.S.

In order to control software tests, it is necessary to apply a process of management, which makes it possible to execute these tests to evaluate if each one is behaving as expected. Several concerns are identified in this process, such as high costs to recruit or train people, the defining of documentation standards, etc.

One approach that has gained prominence to document and assist the activities of test creation, execution and maintenance is the application of test modeling languages, which provides a graphic view that facilitates the abstraction of concepts and the communication between stakeholders. In the literature, there are several approaches related to test modeling, such as the UML Testing Profile [1], the AGEDIS Modeling Language [2], and the Unified Testing Modeling Language [3].

Over the past six years, the Software Engineering Lab (LES) at the Pontifical Catholic University of Rio de Janeiro has worked extensively on coordinating and carrying out tests of large-scale software systems developed (for web and desktop) for different domains (e.g., petroleum, e-commerce, etc). Based on this experience and a request from a client, who wanted to have all the tests modeled, we investigated how UML could be used to model relevant test data and hence to help the coordination of test execution. These data, which could be modeled, were identified from different sources: (i) test maturity models (TMM [14] and TMMi [15]); (ii) continuous integration tools [16] (e.g., Hudson, Continuum and Cruise Control); (iii) test management tools (e.g., Rational Quality Manager [19] and Rational Test Manager [20]); (iv) test modeling languages; and (v) IEEE documents (such as, IEEE 829-2008 [21]). Some of the identified data were described in [23].

From this work, a test group of the LES proposed a new test modeling language called UML Testing Profile for Coordination (UTP-C), which is presented in the paper. UTP-C is an extension of the UML Testing Profile, which is an OMG pattern for the UML language. This approach was provided to allow the modeling of useful data that help the coordination of software testing. According to Baker e al. [1], a profile defines new stereotypes, attributes, and methods to provide additional semantics for the UML.

When UTP-C was being created, we identified the possibility of generating a set of useful artifacts from UTP-C models. However, to conduct this generation, an appropriate tool needed to be created and used. The artifacts identified for automatic generation were: (i) javadoc commentaries in test script source code; (ii) reports that provide important data about modeled tests; and (iii) a set of XML files considered as input data for multi-agent systems [22] that use the Java Self-Adaptive Agent Framework for Self-Test (JAAF+T) [5][6].

JAAF+T is a framework that aims to allow the creation of self-adaptive software agents that perform a self-test before executing self-adapted behaviors. We consider self-test as the action of validating some adaptation before using it. These validations are performed by a set of tests described in XML files and that are explained in detail in [5] and [6]. Hence, from the JAAF+T, a self-adaptive agent can coordinate the execution of tests, i.e., choosing and executing which tests will validate some self-adaptation performed by it.

Since different LES projects use the Rational Software Architecture (RSA) tool to model UML diagrams, we decided to create a new plug-in for the tool called “RSA applying Model-Based Test” (RSA-MBT). The main focus of this plug-in is to generate test artifacts from UTP-C models.

 ISBN: 978-1-61208-304-9

Thus, the paper is organized as follows. In Section II, the new UML profile is explained. In Section III, a case study is presented that illustrates examples of UTP-C diagrams at an e-commerce multi-agent system developed for the web. These diagrams are modeled using the Astah tool [13]. In Section IV, the main idea of the RSA-MBT plug-in is presented, and the diagrams modeled from the Astah (in Section III) are modeled in the RSA tool. Thus, it is possible to see the modeling based on UTP-C in two different tools. In Section V, conclusion and future works are presented.

II. UML TESTING PROFILE FOR COORDINATION

In this section, the UML Testing Profile for Coordination (UTP-C), which was created to model useful data to test coordination, is presented. As stated previously, UTP-C is an extension of the UTP, a standard test profile of the OMG for the UML language. UTP-C uses UML class and activity diagrams for modeling a set of test data. These diagrams were chosen because they allow the modeling of structural and dynamic information that helps the coordination of tests.

The meta-class diagram illustrated in Figure 1 presents a set of stereotypes defined by the UTP-C profile, as well as where they can be used in UML elements. Some of these stereotypes are new, while others are provided by the UTP, but had constraints and properties included. In spite of these inclusions in the UTP-C, they do not challenge the compatibility to the ones that use UTP. Due the limited space of the paper, we will not be able to present in detail these constraints and properties that are described in [6]. However, the example presented in Section III illustrates how UTP-C diagrams can be modeled.

Below, the description of each stereotype used by the UTP-C is presented.

- <<TestCase>>: It states a test case of a system under test (SUT). Each test case is composed of a set of data: test type (e.g., white box, functional, non-function, regression, etc), priority of execution, version of the SUT that it is currently updated, type of obligatoriness, i.e., if execution is mandatory or optional, and the related risk of the system when the test case fails (e.g., to stop the system, data inconsistency, etc.). This set of data related to each test case was not considered by the UTP.
- <<TestContext>>: It states that a set of test cases is responsible for testing some artifacts of the SUT. A test context is composed for: 1 to N test cases, it informs the version of the SUT that their test cases should be updated (desired version), test tool used for executing it, test level related (e.g., unit, integration, system or acceptance), and if it is executed automatically or manually. All these data, except the definition that a test context is composed for 1 to N test cases, were not considered by the UTP.
- <<OrderedSuite>>: It is used to represent a test suite, i.e., an entity that executes a set of test contexts and test cases upon a specific order. UTP considers that a test context is a suite. However, to allow a better identification of a suite class that does not have developed test cases, in comparison to a class that has test cases (test context), we decided to offer the <<OrderedSuite>> stereotype.
- <<TestCriterion>>: It defines a criterion of selection to execute tests of the SUT. An example of a criterion is to execute all the regression and unit tests with high priority and mandatory.
- <<ArtifactUnderTest>>: This stereotype is responsible for representing a set of data related to some artifacts under test (AUT) that are provided in a comment entity. Examples of provided data are the following: path where the results of the tests executed to validate the AUT are stored (result's log), name of the AUT, and type of artifact tested (e.g., class, agent of software, web-service, etc.).
- <<TestClassification>>: It represents a test classification. Test classification is any information that allows grouping and relating test contexts and ordered suites. Its focus is to help the visualization of test entities and their conceptual relations.
- <<Development>>: It represents the real package that stores a given created and modeled class. This is different than the stereotype <<TestClassification>>, which represents conceptual views.

In Figure 1, the Element meta-class is a superclass of the *Classifier* meta-class, which is a superclass of the *Class* (used in class diagrams) and *Activity* meta-classes (used in activity diagrams) [4]. Thus, the TestContext, OrderedSuite, and TestCriterion stereotypes can be used in any sub meta-class of Classifier, while the TestCase stereotype is related to the Behavior meta-class to allow the modeling at behavioral entities, such as Activity.

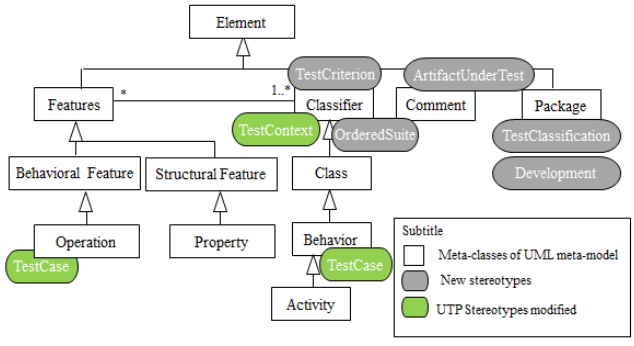

Figure 1. UTP-C meta-model.

Figure 1 also illustrates that the *Classifier* meta-class is related to *StructuralFeature and BehavioralFeature* meta-classes. A structural characteristic is a characteristic of a classifier that specifies the structure of instances of the *StructuralFeature* meta-class, whereas a behavioral characteristic is a characteristic that specifies an aspect of behavior of theirs instances. Thus, the *StructuralFeatures* meta-class is a generalization of *Property* meta-class (attributes of a class are represented as instances of *Property*), and the *BehavioralFeatures* meta-class is a generalization of the *Operation* meta-class, according to the definition of the UML [4]. The original UTP considers that a test case also can be represented as an operation. Hence, the *TestCase* stereotype can be used in the *Operation* meta-class.

The *Comment* meta-class is a subclass of the *Element* meta-class and it can receive the ArtifactUnderTest stereotype. As stated previously, this stereotype informs that data which compose a Comment instance are related to an artifact of the SUT.

TestClassification and Development stereotypes are used in packages (represented by the *Package* meta-class) that allow, respectively, test classifications or development packages to group test contexts and/or suites.

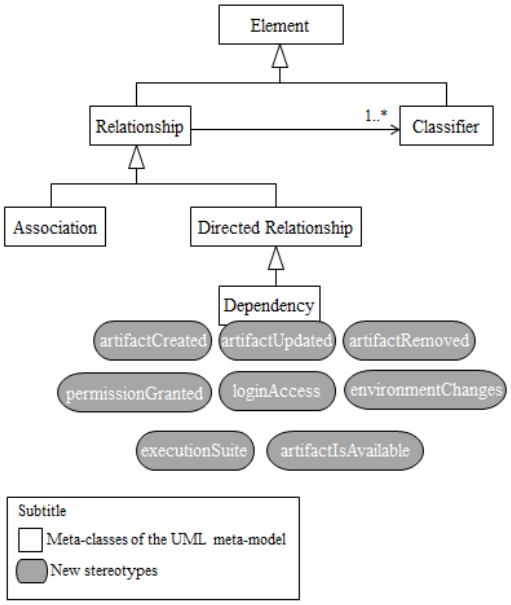

Figure 2. Meta-model of relationships.

Another important data for test coordination is to understand which dependences exist between tests. In order to represent additional semantics on relationships of dependency, a set of stereotypes were proposed by the UTP-C to the UML. These stereotypes were proposed from situations identified in test projects of the Software Engineering Lab. Although this is a limited set, other stereotypes can be included depending on the needs of each project, such as proposals that express more situations of security in SUTs (e.g., <<permissionRevoked>>).

These stereotypes are presented in Figure 2 and described below.

- <<artifactCreated>>: It is used when a test case depends on the creation of some artifact (e.g., file, component, entity, etc.) performed by another test case.
- <<artifactUpdated>>: It states that a test case depends on the updating of an artifact (e.g., changing the name, path, etc.).
- <<artifactRemoved>>: It indicates that the test case depends on the exclusion of another system artifact.
- <<environmentChanges>>: The test case depends on changes in the environment where it is being executed, such as changes to the operating system, environment variables, etc.
- <<permissionGranted>>: It is used when a test case depends on a permission granted from another test case.
- <<loginAccess>>: It states that a test case depends on a login performed in the SUT from another test case.
- <<executionSuite>>: It informs which test contexts an OrderedSuite executes.
- <<artifactIsAvailable>>>: It is used when a test case needs to use an artifact provided by another test case.

III. CASE STUDY: VIRTUAL MARKET PLACE SYSTEM

This section presents the test modeling of the Virtual Marketplace (VMP) application, an e-commerce system where software agents represent users (buyers) and markets (sellers) that sell new and used books. Each buyer agent executes a set of tests to decide which seller will be used to buy his desired books. In order to show how the UTP-C approach can be used a subset of tests created and executed by the buyer agents are modeled. Thus, this section is organized as follows. In Section A, the idea of the VMP system is presented in more detail, and in Section B, UTP-C diagrams are presented and described.

A. Main Idea

Aiming to exemplify the use of the UTP-C, we decided to use the VMP system that provides markets responsible for selling new and used books for users. As stated previously, each user is represented by a buyer software agent, which negotiates with seller agents that represent markets (e.g., Amazon, Ebay, etc.).

Initially, a buyer user should register with the system providing: (i) its preferred market; (ii) the minimum reputation a seller (market agent) must have; and (iii) if he prefers to buy either new or used books. These data are used by the buyer agent to negotiate with seller agents that satisfy the requests made by the user.

 ISBN: 978-1-61208-304-9

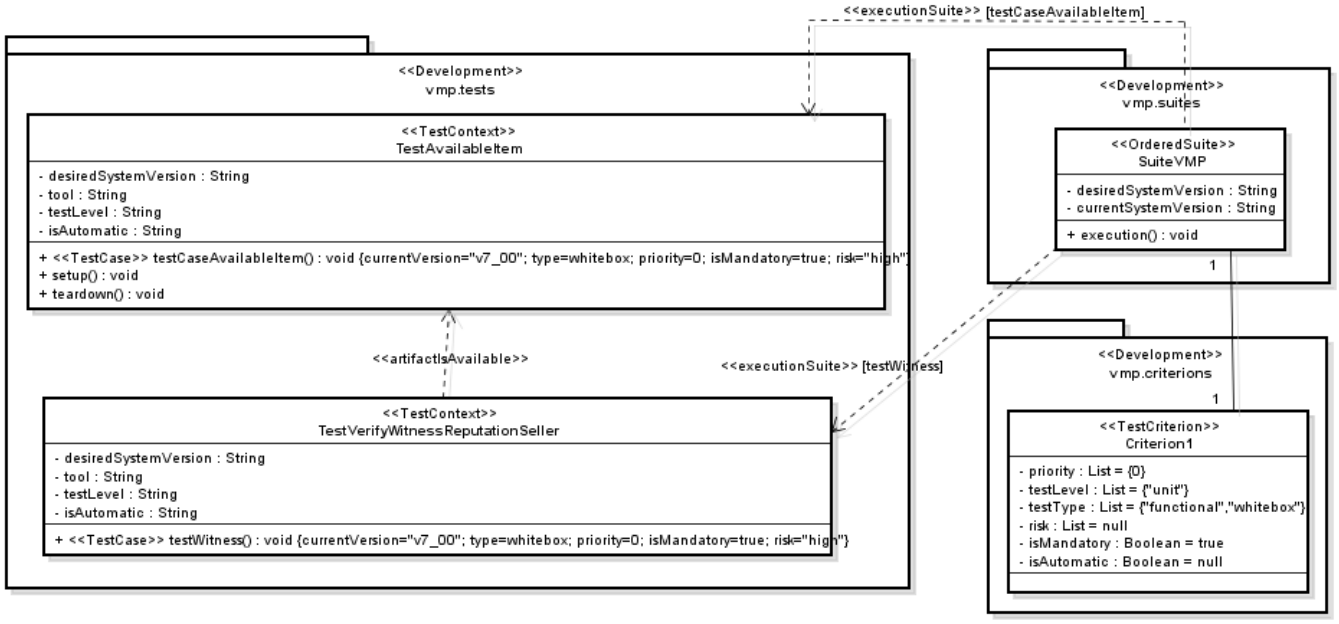

Figure 3. Class Diagram based on UTP-C.

After registering, the user can request the purchase desired. However, a set of data must be provided: (i) title(s) of book(s) desired, (ii) name(s) of author(s), and/or (iii) the maximum price he is willing to pay for book. From these data, the buyer agent (representative of the user) verifies if the seller agent (representative of his preferred market) can meet the request that has been made.

If a seller cannot satisfy the request, the buyer agent tries to meet another seller agent that can sell the desired books. In order to meet another seller, three verifications are performed: (i) if the prices of the desired books provided by the seller are lower than the maximum price informed by the buyer, (ii) if the type of book (used or new) informed by the buyer is respected, and (iii) if the seller agent's reputation is higher than or equal to the minimum reputation of the buyer.

The idea of reputation used on the VMP system is based on the interaction and witness reputations proposed by the Fire model [7]. The interaction reputation is related to the provision of reputations from the negotiation between two agents. In this case, a buyer agent can define a reputation of the seller agent involved in the interaction performed. This reputation is stored in a private buyer agent database. On other hand, the witness reputation allows an agent A to request the reputation (opinion) for an agent B about an agent C. Thus, a buyer agent can request opinions about a seller agent for other buyer agents.

When a seller agent is able to meet the request provided by a buyer, the VMP system presents details of the purchase for the user and it expects confirmation to conclude the negotiation between the agents.

B. Modeling VMP

Figure 3 illustrates a class diagram, created from the Astah tool [13]. This diagram has two test contexts created for the VMP system: TestAvailableItem and TestVerifyWitnessReputationSeller. TestAvailableItem has a test case named testAvailableItem, while TestVerifyWitnessReputationSeller has the test case testWitness. These test contexts execute automatic (use of the attribute isAutomatic) test cases for the version 7.0 of the SUT (represented by the attribute desiredSystemVersion). Furthermore, they use the JAT tool (represented by the attribute tool) [8], which allows the development of unit tests (use of the attribute testLevel) for multi-agent systems.

The main goal of the TestAvailableItem test context is to verify if a seller agent can sell a given book requested by the buyer agent while the TestVerifyWitnessReputationSeller test context verifies if the seller agent has a reputation higher than the reputation informed by the buyer. This conclusion is achieved from the average generated by the reputations provided for other buyer agents of the system about the analyzed seller agent.

Figure 3 shows that each test case of the system contains five more pieces of important associated information: (i) the system version with which the current test case is associated and updated (described by using the attribute currentVersion); (ii) its type of test (e.g., functional, non-functional, regression, etc.); (iii) the priority of the execution (e.g., high, medium, low); (iv) the type of obligatoriness (e.g., mandatory or optional); and (v) the risk related to the test when this test fails. The model allows a description of a risk in detail (e.g., to stop the system) or only its severity related to the SUT (e.g., high severity as illustrated in Figure 3) when a test case to fail. The works presented in [6] and [23] describe in detail the relevance of modeling these test data.

The SuiteVMP class is a test suite responsible for executing the test contexts mentioned previously. If a suite executes a subset of test cases developed through some test context, the modeling can inform which are these test cases

from the following structure: <<executionSuite>> [name_test_case_1, …, name_test_case_N].

The entities modeled in Figure 3 are grouped in packages that have the stereotype <<Development>>. This stereotype represents the package where the classes of a given project are stored. On the other hand, the stereotype <<TestClassification>> can be used to group conceptually test contexts and suites. Packages with this stereotype do not store developed classes, different than packages with the stereotype <<Development>>.

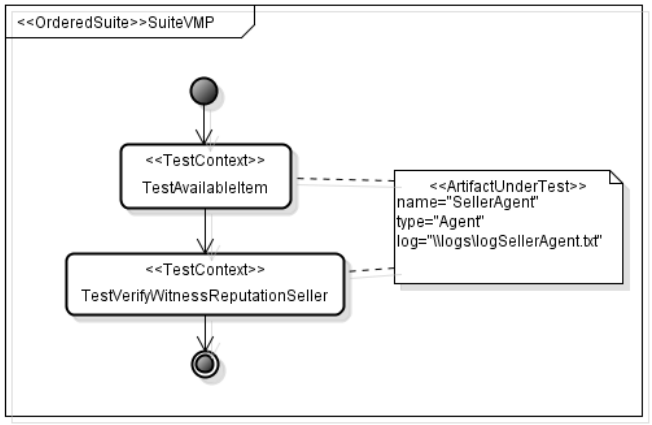

Figure 4. Example of activity diagram.

Finally, but not least important, Figure 4 shows an activity diagram that illustrates the order of execution considered by the SuiteVMP. In this diagram, the first test context to be executed is TestAvailableItem followed by TestVerifyWitnessReputationSeller. The diagram shows that these test contexts are responsible for testing a given seller agent, and the test results are stored at "\\logs\logSellerAgent.txt". These data are provided for a commentary entity with stereotype <<ArtifactUnderTest>> illustrated in Figure 4.

IV. RSA-MBT PLUG-IN

When UTP-C was being created, we identified the possibility of generating a set of useful artifacts from UTP-C models. Thus, the RSA-MBT was proposed. It is an open-source plug-in, developed in Java, for the Rational Software Architecture (RSA) tool, and it is available in [9].

From the RSA-MBT it is possible to generate test artifacts based on UTP-C diagrams. The possible test artifacts, which can be generated from it, are the following: (i) test reports for test teams; (ii) javadoc commentaries; and (iii) a set of XML files used in multi-agent systems that instantiate the JAAF+T framework. Notice that currently the plug-in is not creating test codes. However, we intend to include this generation in the next releases of the plug-in. Thus, the main idea of the RSA-MBT is to generate a set of artifacts that can help the work of test teams, such as understanding characteristics of each test case (e.g., from javadoc commentaries), and knowing which tests are not updated to a specific version of a system under test (e.g., using test reports generated).

The RSA tool allows several transformations, such as from UML diagrams to Java. When this transformation is requested, the RSA-MBT is executed.

Figure 5 illustrates the same classes modeled in Figure 3, but modeled from the RSA tool. Data of the test cases (methods) are presented in the Documentation tab, when a test case method is selected, as illustrated in Figure 6. This approach was considered, because RSA tool does not allow modeling these data of the test cases as the Astah tool. Besides, we informed that the current version of the testWitness is v6_00, which is different from the one in Figure 3. This was performed in order to show better some data generated from the plugin proposed and explained more in the following.

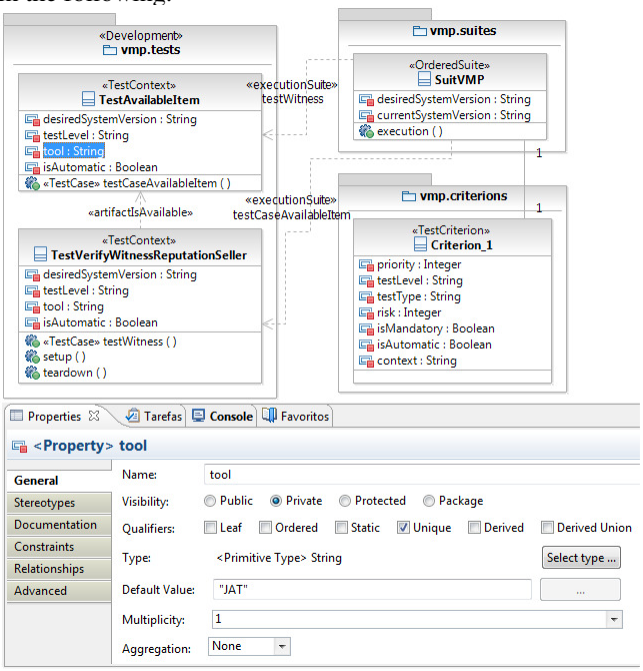

Figure 5. Example of class diagram based on UTP-C.

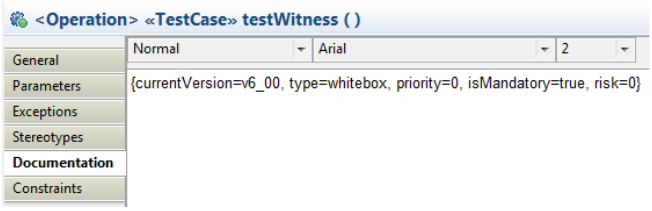

Figure 6. Documentation tab of the RSA tool.

From modeling of diagrams based on the UTP-C approach, the user should request the UML to Java transformation. With this request the main screen of the RSA-MBT is presented (see Figure 7). Such a screen allows choosing which test artifacts will be generated and which language must be considered. Nowadays, the plug-in allows generating artifacts in six different languages: English, Portuguese, Italian, French, Spanish, and German.

 ISBN: 978-1-61208-304-9

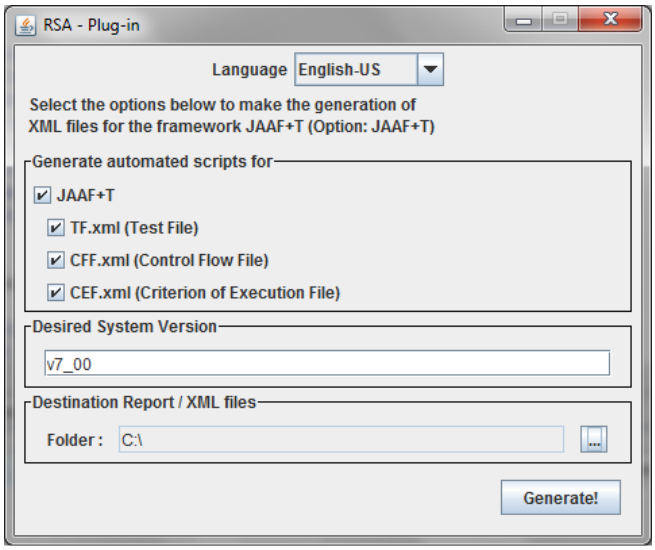

Figure 7. RSA-MBT screen.

Figure 8 illustrates an example of javadoc commentaries generated in English. In this example, commentaries are provided to the class (test context) TestVerifyWitnessReputationSeller and to its test case (testWitness method) modeled in the class diagram presented in Figure 5.

The commentaries generated to the class TestWitness are based on the data provided in the modeled attributes: desiredSystemVersion, testLevel, tool and isAutomatic. Hence, it is informed that such test context uses the JAT tool, is an automatic and unit test context, and should be updated to the version "v7_00" of the SUT. On other hand, the commentaries generated by the "testWitness" method are based on the data provided in the "Documentation" tab presented in Figure 6. Thus, RSA-MBT informs that it is a mandatory and a white-box test case currently updated to version v6_00 of the SUT. Besides, it has priority and risk 0 (zero), i.e., high priority and risk, respectively.

```
/**
 * <!-- begin-UML-doc -->
 * This class has the following stereotype(s): TestContext
 * The desired updated version is "v7_00"
 * This class has the following test case(s):   testWitness();
 * The Test level is "unit"
 * The Test tool is "JAT"
 * It's a Not Automated test
 * <!-- end-UML-doc -->
 * @author Andrew
 */
public class TestVerifyWitnessReputationSeller {
    /**
     * <!-- begin-UML-doc -->
     * {currentVersion=v6_00, type=whitebox, priority=0,
     * isMandatory=true, risk=0}
     * This Test is updated to Version v6_00
     * This test type is whitebox
     * This test priority is 0
     * It's a Mandatory test
     * This test Risk is 0
     * <!-- end-UML-doc -->
     * TestCase
     */
    public void testWitness() {
        // begin-user-code
        // TODO Auto-generated method stub

        // end-user-code
    }
}
```

Figure 8. Example of javadoc commentaries.

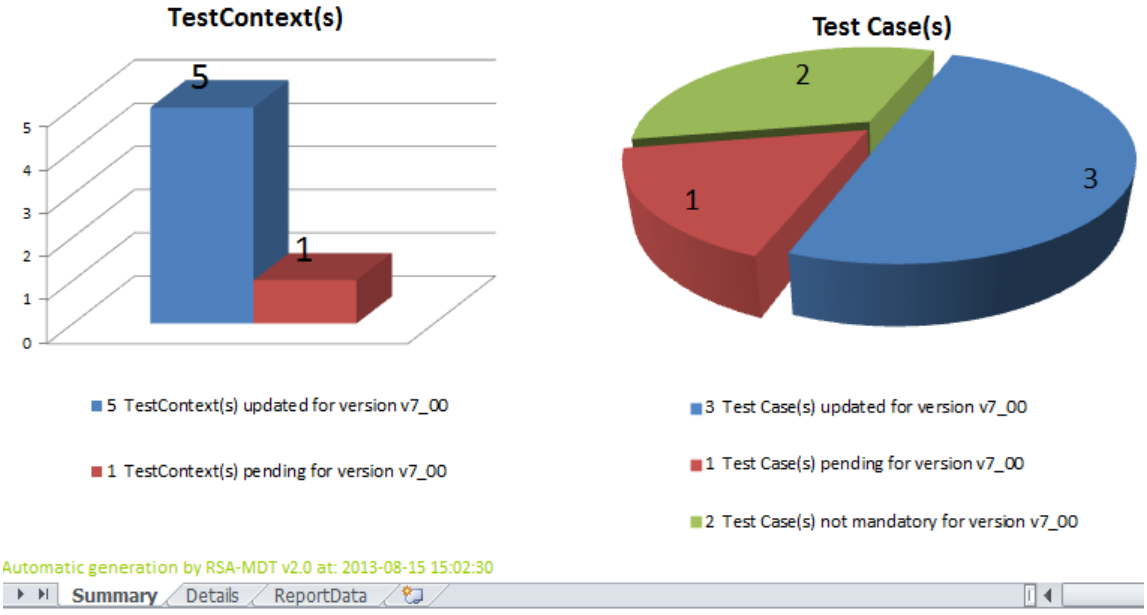

Figure 9. Summary tab – test report generated.

 ISBN: 978-1-61208-304-9

In order to provide an overview of which test contexts and test cases are updated to a specific version informed by the user (by using the text field "Desired System Version" illustrated in Figure 4), a test report (".xls" extension) is created. This report has three tabs, which are explained in detail as follows.

- **Summary tab** (see Figure 9): It presents two graphics that inform the number of test contexts and test cases updated to the version provided by the user (we are considering that the desired version is v7_00).
- **Details tab** (see Figure 10): It lists the test contexts (test classes) updated and not updated to the version desired.
- **ReportData tab** (Figure 11): It presents an overview of the current state of these updates.

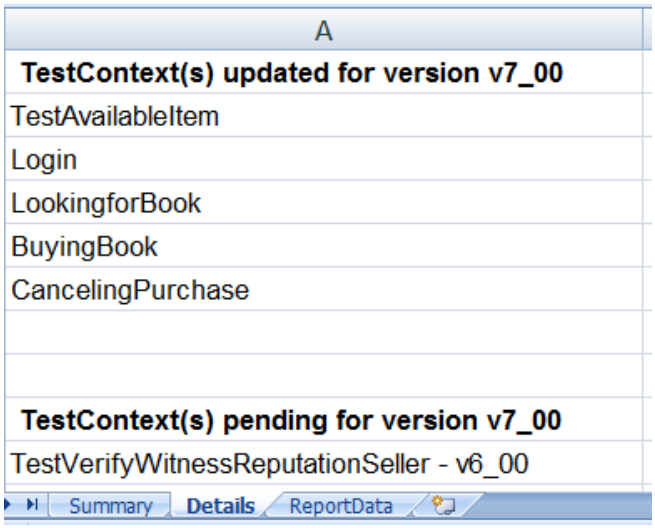

Figure 10. Details tab – test report generated.

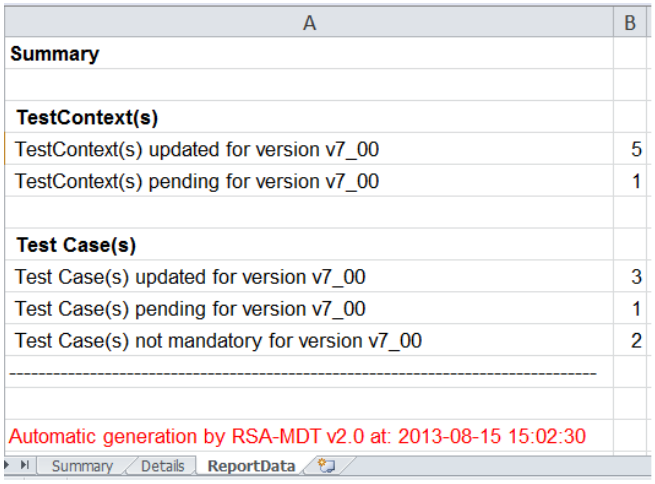

Figure 11. ReportData tab – test report generated.

Also, RSA-MBT generates XML files as input data to the JAAF+T framework. As stated previously, JAAF+T is a framework that allows creating self-adaptive agents that perform self-tests based on a set of XML files.

Three XML files can be generated by the plug-in: TF.xml, CFF.xml and CEF.xml. Test File (TF.xml) is responsible for describing all the tests that can be executed in self-adaptations (see Figure 12). Control Flow File (CFF.xml) presents the order of execution that tests must be executed to validate some artifact of the SUT (see Figure 13). While Criterion of Execution File (CEF.xml) describes the criterions that define which tests, present in the TF.xml file, will be executed (see Figure 14).

```
<tf>
 <test id="testAvailableItem" isAutomatic="true"
        testLevel="unit" testType="whitebox"
        isMandatory="true" context="VMP v1.0">
       <classpath>
       vmp.tests.TestAvailableItem
       </classpath>
       <priority>0</priority>
       <risk>0</risk>
       <tool>JAT</tool>
 </test>
 <test> ... </test>
</tf>
```

Figure 12. Example of a TF.xml file.

```
<cff>
  <artifact id="SuiteVMP" type="Agent"
      logPath="C:\log.txt"
      criterionID="Criterion1">
      <test type="test"
        id="testAvailableItem"/>
      <test type="test"
        id="testVerifyWitnessReputationSeller"/>
  </artifact>
</cff>
```

Figure 13. Example of a CFF.xml file.

The main idea of using UTP-C models was to make creation and maintenance of these XML files easier since, depending on the size of an XML file, the editing work can be difficult. Thus, as all the data considered by the XML files can be modeled in UTP-C diagrams, it is often easier to edit diagrams than to work with XML files. Details of these files are presented in [5].

 ISBN: 978-1-61208-304-9

V. DISCUSSION

One of the most relevant work related to test modeling is the UML Testing Profile [1] that defines a profile for designing, visualizing, and documenting the artifacts of test systems. Such an approach extends UML 2.x [4] with test specific concepts, such as test components, verdicts, defaults, etc. These data are grouped in test architecture, test data, test behavior and time. Being a profile, the UML testing profile seamlessly integrates into UML: it is based on the UML meta-model and reuses UML syntax. Although the approach proposes interesting concepts for modeling test systems, it does not support the modeling of important test data represented by our test modeling language, such as the identification of (i) the system version that each test is able to test, (ii) the mandatory and optional tests, (iii) the test types created, (iv) the types of dependences that exist between the tests (such as data dependence), and (v) the automated and manual tests. On the other hand, the UTP-C approach provides support to represent these test data.

```
<cef>
  <criterion id="Criterion1">
      <priority>0</priority>
      <testLevel>unit</testLevel>
      <testType>functional</testType>
      <testType>whitebox</testType>
      <isMandatory>true</isMandatory>
  </criterion>
</cef>
```

Figure 14. Example of a CEF.xml file.

AGEDIS modeling language (AML) [2], which is another testing language, is based upon the UML (1.4) meta-model and enables the specification of tests for structural (static) and behavioral (dynamic) aspects of computational UML models. AML comes as part of the AGEDIS methodology and has been designed with two main goals in mind: to create a test adequate abstraction of the SUT that will be analyzed by the AGEDIS tools, which allows generating automatically suite tests, and to set meaningful test directives for the testing process. AML presents the same problems mentioned for the UML Testing Profile.

The Testing and Test Control Notation (TTCN-3) [11] is a modular language that has the similar look and feel of a typical programming language. This language is widely accepted as a standard for test system development in the telecommunication and data communication area. The main reason for such acceptance is that it comprises concepts suitable to any type of distributed systems to be tested, such as important features necessary to specify test procedures for functional, conformance, interoperability, load and scalability tests. Besides this, it defines mechanisms to compare the reactions of the system under test with the expected range of values, time handling, distributed test components, ability to specify encoding information, synchronous and asynchronous communication, and monitoring. Similar to the UML Testing Profile, TTCN-3 also does not provide a set of useful concepts that the test modeling language, presented in this paper, proposes. All the concepts not included in the UML Testing Profile and AGEDIS are also not considered in this work.

According to [3], the benefits of Model-Driven Engineering (MDE) for product software development have been demonstrated in numerous instances. Therefore, similar benefits can also be achieved in applying MDE to test software development. This form of Model-Based Testing (MBT) is called Model-Driven Test Engineering (MDTE) or simply Model-Driven Testing (MDT). However, to optimize the efficiency of MDT, good-practices and patterns specific to test development must be taken into account. Based on this idea, Feudjio [12] proposes a Unified Test Modeling Language (UTML) that is a test notation designed for pattern-oriented MDT. It provides the means for designing all aspects of a test system at a high level of abstraction and independent of any specific lower-level test infrastructure. Besides this, at the same time it provides guidance in following test design patterns and avoids usual pitfalls of MDT. Such an approach provides a tool called MDTester that allows modeling the concepts proposed by UTML. However, this tool does not allow to explicitly model the test data provided by the UTP-C, such as, test type, test level, risk, priority, etc.

VI. CONCLUSION AND FUTURE WORK

This paper presented a new test modeling approach named UML Testing Profile for Coordination (UTP-C). This approach extends the UML Testing Profile in order to model useful data that helps test coordination. These data were identified from tests created and executed for different systems (web and desktop) in the Software Engineering Lab. This work has been motivating research related to the test area, especially the Model Based Test, such as the creation of the RSA-MBT plugin, presented in the paper.

Considering that the plug-in was created for the Rational Software Architecture (RSA), when a transformation is requested in the RSA, files generated by the tool are replaced (e.g., Java files created from UML diagrams). Due to this behavior, we are currently developing a treatment that allows applying a merge between Java files. Thus, important contents of Java files already created will not be lost when a UML to Java transformation is requested.

Besides, we are deciding how to automatically generate codes for test scripts for the Rational Functional Tester (RFT) [17] and for the Rational Performance Tester (RPT) [18]. RFT and RTP are tools used in different test projects of the LES that allow creating functional and performance test scripts, respectively.

 ISBN: 978-1-61208-304-9

VII. ACKNOWLEDGMENTS

This work has been sponsored by the INCT on WebScience through grants from CNPq and FAPERJ.

REFERENCES

[1] P. Baker, Z. Ru Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, "Model-Driven Testing: Using the UML Testing Profile", Springer, ed. 2008, December, 2007.

[2] A. Hartman and K. Nagin, "The AGEDIS Tools for Model Based Testing", Book UML Modeling Languages and Applications, vol. 3297, 2005, pp. 277-280, doi: 10.1007/978-3-540-31797-5_33.

[3] UTML - The Unified Test Modeling Language for Pattern-Oriented Test Design, <http://www.fokus.fraunhofer.de/distrib/motion/utml/>, retrieved: August, 2013.

[4] UML 2 Specification, <http://www.omg.org/spec/UML/2.3/>, retrieved: August, 2013.

[5] A. D. Costa, C. Nunes, V. T. Silva, C. J. P. Lucena, and B. Fonseca, "JAAF+T: A Framework to Implement Self-Adaptive Agents that Apply Self-Test", in proceedings of the 25th Symposium On Applied Computing, Sierre, Switzerland, 2010, pp. 928-935.

[6] A. D Costa, "Automation of the Management Process of the Test of Software", Thesis at Portuguese, Pontifical Catholic University of Rio de Janeiro, August, 2012.

[7] T. D. Huynh, N. Jennings, and N. Shadbolt, "FIRE: an Integrated Trust and Reputation Model for Open Multi-agent Systems. In Proceedings of the 16th European Conference on Artificial Intelligence", Valencia, Spain, 200, pp.18-22.

[8] R. Coelho, E. Cirilo, U. Kulesza, A. Staa, A. Rashid, and C. J. P. Lucena, "JAT: A Test Automation Framework for Multi-Agent Systems", in Proceeding of the International Conference on Software Maintenance, France, 2007, pp. 425-434.

[9] RSA-MBT: Web site for downloading, <http://www.les.inf.puc-rio.br/escritorioqualidade/index.php?option=com_content&view=article&id=57&Itemid=58>.

[10] Rational Software Architect, <http://www.ibm.com/developerworks/rational/products/rsa/>, retrieved: August, 2013.

[11] TTCN-3 web site, <http://www.ttcn-3.org/>, retrieved: August, 2013.

[12] A. V. Feudjio, "MDTester User Guide", <http://www.fokus.fraunhofer.de/distrib/motion/utml/>, retrieved: August, 2013.

[13] Astah tool, <http://astah.net/>, retrieved: August, 2013.

[14] I. Burnstein, A. Homyen, R. Grom, and C.R. Carlson, "A Model to Assess Testing Process Maturity", Crosstalk 1998, Software Technology Support Center, Hill Air Force Base, Utah, <http://www.crosstalkonline.org/storage/issue-archives/1998/199811/199811-Burnstein.pdf>, retrieved: August, 2013.

[15] TMMi: The Test Maturity Model Integration, <http://www.tmmifoundation.org/html/tmmiref.html>, retrieved: August, 2013.

[16] P. M. Duvall, S. Matyas, and A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk, Publisher: Addison-Wesley Professional, 2007.

[17] Rational Functional Tester, <http://www-03.ibm.com/software/products/us/en/functional/>, retrieved: August, 2013.

[18] Rational Performance Tester, <http://www-03.ibm.com/software/products/us/en/performance/>, retrieved: August, 2013.

[19] Rational Quality Manager tool, <http://www-03.ibm.com/software/products/us/en/ratiqualmana/>, retrieved: August, 2013.

[20] Rational TestManager tool, <http://www-01.ibm.com/software/awdtools/test/manager/>.

[21] IEEE 829-2008 – IEEE Standard for Software and System Test Documentation, <http://standards.ieee.org/findstds/standard/829-2008.html>, retrieved: August, 2013.

[22] M. Wooldridge and N. R. Jennings, "Pitfalls of agent-oriented development", in Proceedings of the Second International Conference on Autonomous Agents (Agents'98), ACM Press, Minneapolis, USA, 1998, pp. 385-391.

[23] A. D. Costa, V. T. Silva, A. Garcia, and C. J. P. Lucena, "Improving Test Models for Large Scale Industrial Systems: An Inquisitive Study", in Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems, Part I, LNCS Springer 6394, Oslo, Norway, 2010, pp. 301-315.

[24] NIST: National Institute of Standards and Tecnology, Software Errors Cost U.S. Economy $59,5 Billion Annually – NIST Planning Report 02-3, 2002.

 ISBN: 978-1-61208-304-9