# Experiment on Using Model-Based Testing for Automatic Tests Generation on a Software Radio Protocol

**Shuai Li, Michel Bourdellès, Alexandre Acebedo**
THALES Communications & Security
160 Boulevard de Valmy
92700 Colombes, France
Telephone: +33 (0)1 46 13 21 65
E-mail: {shuai.li, michel.bourdelles, alexandre.acebedo}@fr.thalesgroup.com

**Julien Botella, Fabien Peureux***
Smartesting R&D Center
18 rue Alain Savary
25000 Besançon, France
Telephone: +33 (0)3 81 25 53 63
E-mail: {julien.botella, fabien.peureux}@smartesting.com

**Abstract:** In this paper we present an industrial experiment report on the use of automatic tests generation integrated to Model-Based Testing solutions. We detail the different necessary steps to integrate the automatic tests generation in a model-driven design flow. The case-study is an embedded software radio protocol. UML has been used to model the system with a component-based approach. The experimented automatic tests generation tool is provided by Smartesting.

**Key words**: Model-Based Testing; UML; Software Radio; Embedded System; Automatic Tests Generation

## 1. INTRODUCTION

In this paper we explore software testing for a Software Radio Protocol (SRP) with models. A SRP is an embedded system in the telecommunication domain. SRP have become more and more complex through time, with more and more components interacting to achieve the desired functionalities. Requirements on the system are growing, as possible scenarios explode, and software testing is an essential step in the development of such systems. When done manually, this task is time consuming. Using Model-Based Testing (MBT) [12] is one solution to automate the tests.

Previous experiments on MBT use have been applied on industrial test cases. One such experiment on THALES Airborne System object-oriented models is exposed in [6]. The authors suggest formalizing existing practices in the industry instead of pushing formal methods into such domains. Test cases are generated from use-case contracts and use-case scenarios, improved by the introduction of contracts. Collaborative projects have also been established in the past in order to develop MBT tools suitable for the industry. AGEDIS [3] uses an UML-based language to model applications to be tested, with an integrated tests generation and execution environment, experimented on case-studies provided by IBM, France Telecom and Intrasoft.

In this paper we present an experiment report on automatically generating test suites for a Software Radio Protocol through MBT. The SRP use-case is provided by THALES Communications & Security, specialized in radio equipments. We show how we model the SRP as a component-based model and how we integrate the Smartesting CertifyIt [11] tests generation tool into the SRP's design.

The rest of the paper is organized as follows. Section 2 details our SRP use-case and its model. In section 3 we show how we integrated the Smartesting CertifyIt tool with our design method. Experimental results on generating tests are exposed in section 4. In section 5 we evaluate our work and we list the metrics. Finally we conclude in section 6 with some future works.

## 2. THE SOFTWARE RADIO PROTOCOL CASE-STUDY

In the following sections the Software Radio Protocol system is first presented. We then show how we model it in UML.

*Also affiliated with FEMTO-ST Institute - UMR CNRS 6174, Besançon, France

### 2-1 Software Radio Protocol Application

Software Radio Protocols (SRP) are communication protocols embedded in radio equipment. A SRP development is related to the software design layers in the OSI [13] model. A SRP is composed of one or several applications called waveforms. A waveform is composed of different software components that manage radio channel access technologies, radio protocol and routing. The waveform design may be separated into several functional layers following coarsely the OSI model:

- PHY: Synchronization, data transmission/reception
- LINK: Protocol management
- NET: User packet Handling

The SRP we have experimented with, is used in an ad-hoc network. The protocol uses Time Division Multiplexing Access (TDMA) [1] as a method to share access to the radio channel. In TDMA, time is divided into several time slots. At each slot, a radio has an action to perform (e.g. transmit, receive or ignore operations). Verifying that the correct operation is performed at each slot is thus necessary, as well as correct data output according to input data.

For our work we decided to focus on the MAC inter-layer between the PHY and the LINK layers. The MAC processes the slots and sends instructions to the other layers. The different internal components in the MAC collaborate to provide functionality such as data transmission/reception, radio resource allocation (i.e. TDMA allocation) and PHY configuration.

### 2-2 Software Radio Protocol Model

With respect to current software development practices at THALES Communications & Security [8], the SRP is modeled in UML as a component-based model. This model is used for wrapper code generation.

The layers have been modeled according to a typical architecture description approach [5]. The elements in the components model are the following:

- Functional Component: A set of functionalities and services that can be connected with other components sharing its interfaces.
- Port: A component's port is the entry-point for other components. Through a port a component requires or provides services. A port is typed with an interface.
- Interface: An interface regroups operations. Operations are the services and functionalities implemented by the component.
- Data type and Enumeration: Operations have parameters that are typed.
- Connector: Connects two components through ports.

In the following section, we show how we generate tests automatically from the SRP model.

## 3. INTEGRATING AUTOMATIC TESTS GENERATION

In order to automatically generate tests for the SRP system, we have defined a tests generation flow. This flow is presented in Table 1.

Table 1. Tests Generation Flow

| Step | Description |
|---|---|
| S1 | Choose data types to abstract from the source model. |
| S2 | Generate the System Under Test model and complete it. |
| S3 | Add constraints in OCL. |
| S4 | Generate the data pool structure and complete it with test inputs. |
| S5 | Generate the tests. |
| S6 | Publish executable tests as C/C++ source. |

In the following sections we detail the steps in this flow.

### 3-1 Data abstraction (S1)

Before generating the System Under Test (SUT) model, the user has to choose the correct data types to abstract. Abstracting a data type limits the possible number of tests generated. For example an integer data type may be

abstracted as an enumeration of possible integer values, a byte sequence may be abstracted as a valid or non-valid sequence.

### 3-2 A Component-based Model to an Object-Oriented Model (S2)

The current component-based model (CBM) cannot be used as an input for the Smartesting CertifyIt tests generation tool, which works with classes and instances to model the SUT. It is necessary to extract information from the original model and represent it in an object-oriented model (OOM). We use model transformation [10] to do this. In our work, the output meta-model was implemented as a UML profile [4].

Figure 1 describes the profile we defined for our output OOM. The elements of this library are the following:

- Primitives imported from UML: Boolean, Integer, Unlimited Natural, and String
- Meta-classes imported from UML: Class, Property, and Parameter
- ConnectedClass: ConnectedClasses are Ports in the original model.
- AbstractedDataType: This stereotype is applied to Properties of DataTypes and Parameters of Operations.

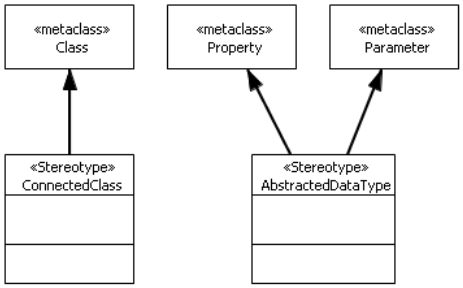


Figure 1. UML Profile for the Object-Oriented SUT Model

Table 2. Component-Based Model to Smartesting Object-Oriented Model Transformation Rules

| Rule | CBM (input model) | OOM (output model) | Additional details |
|---|---|---|---|
| R1 | Datatype, Enumeration | Class, Enumeration | |
| R2 | Component | Class | The output Class contains properties typed by the owned Ports. |
| R3 | Port | ConnectedClass | The ConnectedClass contains a Property typed by the Component Class (in the OOM) to which it belongs. |
| R4 | Component::Port | Association | An association exists between a Component Class and any of the ConnectedClasses it owns as Ports. |
| R5 | Interface::Operation | ConnectedClass::Operation | A ConnectedClass is composed by Operations, the same that are defined by the Interface typing the original Port. |
| R6 | Operation::Parameter | AbstractedDatatype | A ConnectedClass' Operations' Parameters are stereotyped << AbstractedDatatype>>. |
| R7 | Datatype::Property | AbstractedDatatype | Datatypes with Properties are structures in the CBM. A Datatype Class in the OOM has properties stereotyped << AbstractedDatatype>>. |

We use the transformation rules in Table 2 to transform the CBM into an OOM. Instances for Classes and Associations are created automatically by the transformation. Data type instances need to be added manually as they represent values in the model to be used as test inputs.

### 3-3 Adding OCL Constraints (S3)

After the generation and completion of the SUT model, postconditions have to be added on operations that are provided by ports of a component. A postcondition is described using an UML Constraint and is written in the Object Constraint Language (OCL) [9]. Typical postconditions are the values that an operation should return or a Boolean if the operation does not have any return value (the Boolean indicates the operation's success/failure). These postconditions thus describe the abstract behavior of the implemented operations.

### 3-4 Data Pool Generation (S4)

In order to generate test inputs, it is necessary to provide it with test data coming from a data pool.

A model-to-text transformation has been developed in order to generate the data pool structure. This structure takes into account every abstract data type used in the SUT model and creates for each:

- A C++ source file: This file contains a class representing the data type. The class has a "getValue" function that returns the correct input used for the test, according to an identifier parameter passed to it. The returned value can be for example an integer or an enumeration.
- A C++ header file for the source file.
- An identifier: The parameter used to get the test input value is called the identifier. For example "DEFAULT" is an enumeration literal that may identify the kind of sequence that has to be returned.

The only effort needed by the user is to complete the identifiers and write the "getValue" function, i.e. what value should be returned according to the identifier passed as a parameter of the function.

### 3-5 Tests Generation and Execution (S5 & S6)

After generating the SUT model, writing the OCL constraints and writing the data pool, the tests are generated with Smartesting CertifyIt. The tests need to be published and related to an activator framework. Smartesting provides publishers to the most popular activators on the market. In the experiment, the targeted activator was Google C++ Testing Framework [2]. The test scenarios are generated as C++ source files that are compiled into the executable tests on the system. One test is associated with one system requirement.

## 4. EXPERIMENTAL RESULTS

Tests generation has been experimented on three components in the MAC. Figure 2 shows the MAC model and these components' position in the MAC.

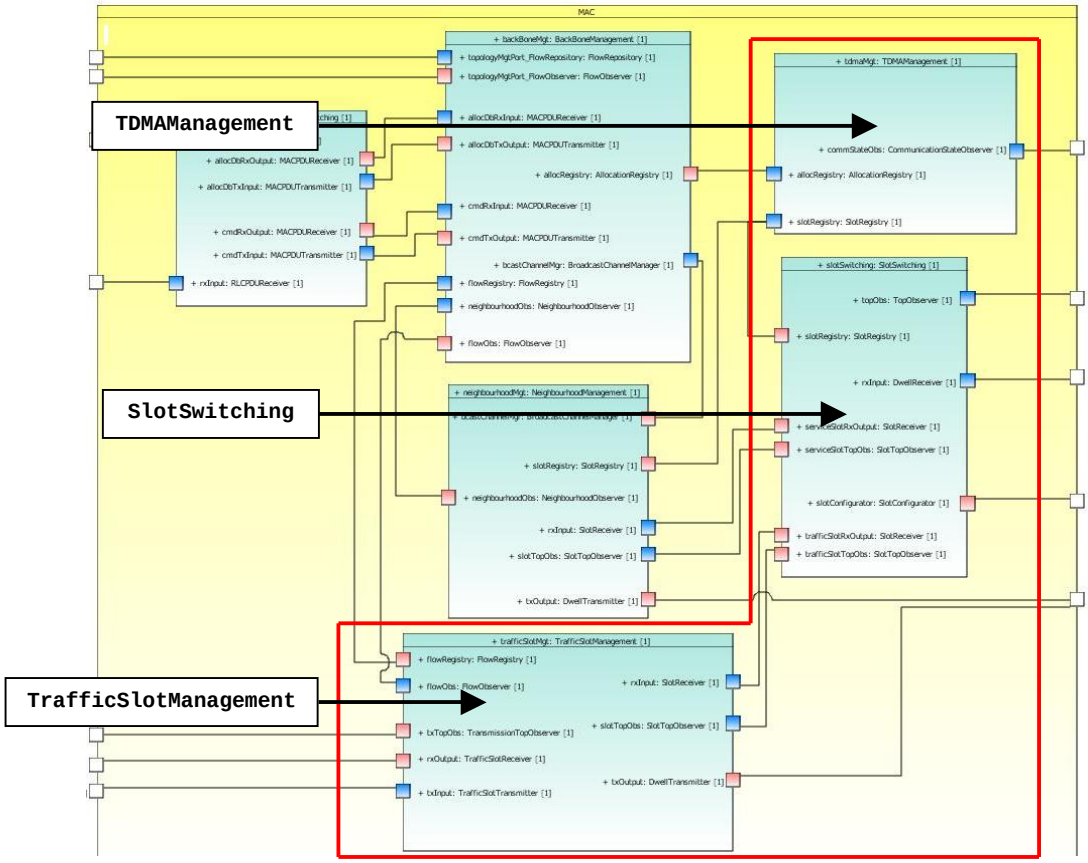


Figure 2. Experimented Components in the MAC

The `SlotSwitching` component plays a switch role. It receives as input a slot time. According to the action to perform at that time, it will delegate operations to other components in the MAC. The kinds of tests that have been automatically generated are port activation tests, i.e. if a certain kind of slot is received (transmission, reception or ignore), it will activate certain ports but never others.

The `TDMAManagement` component is a register component. It stores the TDMA structure that indicates the action to perform at each slot time. Tests on the correct data return and update are generated.

Finally the `TrafficSlotManagement` component is a state machine. This component will execute operations and activate certain ports according to its current state and its previous states.

Figure 3 shows the `SlotSwitching` component's transformation into its representation in the SUT model (S2). Datatype instances (S2) and postconditions in OCL (S3) are added to the SUT model.

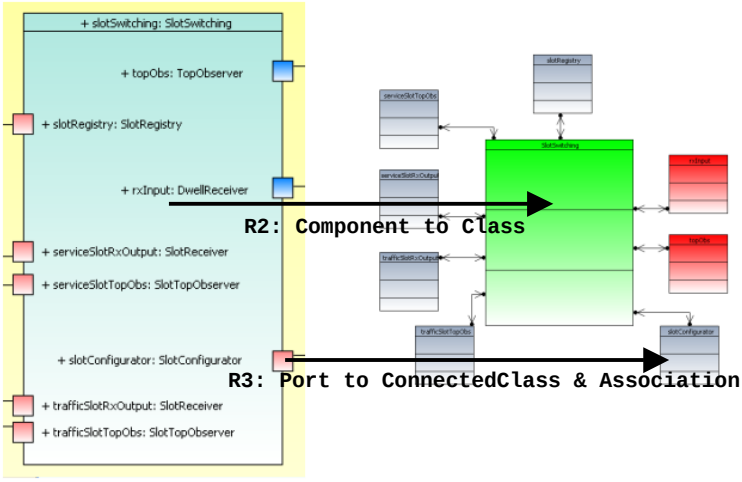


Figure 3. SlotSwitching Component Transformation from Rules in Table 2

Between 32 and 54 tests have been generated for each component. The test scenarios generated from the test model were executed (S6) on a Linux-based platform as a complement of the software integration on host. The generated tests use the wrapper code generated from the CBM to execute.

## 5. EVALUATION OF THE AUTOMATIC TESTS GENERATION FLOW

We have used the following metrics to evaluate this experiment's results:

- (M1) Capacity to model system requirements (state space explosion, data value expressivity capabilities)
- (M2) Consistency of generated tests
- (M3) Effort needed to concretize and execute tests for the developed application

For M1, as requirements are expressed as control vertices access in the underlying SUT, only a subset of the system requirements on software can be verified. For example requirements on complex exchanges of multiple radio equipments, mixing watch dogs and data values assertions, are particularly awkward to express in the SUT model. Due to state space explosion, the right tradeoff to be tuned between abstraction level and expressivity is tedious to define.

For M2, inconsistencies between manually written tests and generated tests were effectively detected. In particular a same requirement was interpreted differently by a manual test approach and the automatic generation approach. This induced in two tests that had different implemented functional behaviors for the same requirement.

For M3, OCL was used to model system behavior. This diverts the language from its initial use on conditions capturing. An adapted behavior modeling language (e.g. ALF [7], or preferably graphical activity diagrams) is better suited for an industrial usage.

In conclusion of our evaluation of MBT applied to a Software Radio Protocol using Smartesting CertifyIt, we state the following: MBT is the appropriate link between formal verification and software integration validation by manual tests. Automatic tests generation is seen as a complement of manual tests related to requirements. Moreover automatic tests generation provides coverage information for building the SUT specifications.

## 6. CONCLUSION

In this paper we have described our experiment on applying an automatic tests generation tool, through MBT, on a Software Radio Protocol design at THALES Communications & Security.

The radio protocol has been modeled as an UML component-based model. A tests generation flow has been established to integrate the Smartesting CertifyIt tool to the current design flow. The developed model transformations for the flow have been described. Finally practical experiments on automatic tests generation with the Smartesting tools have been presented and we have evaluated of this approach.

Using the MBT approach we are currently able to generate functional tests based on the control flow of the application. Further works have to concentrate on the use of test generation pillars, which offer robustness determination on a range of input data for the tests. For example we would like tools to indicate that a range of data will always follow the same path of control points, and thus test several values for this range of data to validate the path.

## 7. ACKNOWLEDGEMENTS

This work is performed in the framework of the ITEA-2 funded project VERDE (http://www.itea-verde.org). The views expressed in this document do not necessarily represent the views of the complete consortium. The community is not liable for any use that may be made of the information contained herein. The authors would like to thank Patrick Kolodziejczyk for his help on this experiment.

## 8. REFERENCES

[1] T. S. Chan: Time-Division Multiple Access; *Handbook of Computer Networks*, John Wiley & Sons, pp. 769-778, 2011
[2] Google: Google C++ Testing Framework; http://code.google.com/p/googletest/
[3] A. Hartman, K. Nagin: The AGEDIS Tools for Model Based Testing; *ACM SIGSOFT Software Engineering Notes*, vol. 29, n°. 4, pp. 129, 2004.
[4] M. Kandé, A. Strohmeier: Towards a UML Profile for Software Architecture Descriptions; *UML 2000 - The Unified Modeling Language*, Springer Berlin Heidelberg, pp. 513-527, 2000
[5] N. Medvidovic and R. N. Taylor: A classification and comparison framework for software architecture description languages; *IEEE Transactions on Software Engineering*, vol. 26, n°. 1, pp. 70-93, 2000
[6] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel: Automatic Test Generation: A Use Case Driven Approach; *IEEE Transactions on Software Engineering*, vol. 32, n°. 3, pp. 140-155, 2006
[7] OMG: ALF Specification; http://www.omg.org/spec/ALF/1.0/Beta2/PDF/, 2010
[8] OMG: CORBA Component Model Specification; http://www.omg.org/spec/CCM/4.0/PDF/, 2006
[9] OMG: Object Constraint Language Specification; http://www.omg.org/spec/OCL/2.3.1/PDF, 2012
[10] S. Sendall and W. Kozaczynski: Model Transformation: The Heart and Soul of Model-Driven Software Development; *IEEE Software*, vol. 20, n°. 5, pp. 42-45, 2003
[11] Smartesting: Smartesting; http://www.smartesting.com/index.php/cms/en/home
[12] M. Utting and B. Legeard: Practical Model-Based Testing: A Tools Approach; Elsevier, 2007
[13] H. Zimmermann: OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection; *IEEE Transactions on Communications*, vol. 28, n°. 4, pp. 425-432, 1980