Automatic Generation of Test Oracles from Component Based Software Architectures

Maxime Samson and Thomas Vergnaud(✉)

Thales, 1, Avenue Augustin Fresnel, Palaiseau, France
{maxime.samson,thomas.vergnaud}@thalesgroup.com

Abstract. In a software development process, the integration and verification of the different parts of the application under development often require a lot of effort. Component Based Software Engineering (CBSE) approaches help cut software integration costs by enabling the automatic generation of data types, method signatures and middleware configuration from a model of the application structure. Model Based Testing (MBT) techniques help cut software verification costs by enabling the automatic generation of test oracles from a model of the expected application behaviour. Models for CBSE and MBT are usually separate. This may result in discrepancies between them, especially when the application architecture is updated, which always happens.

In this paper, we describe how to rely on a single CBSE model to produce both code generation and oracles for some tests, thus ensuring consistency between them. Our work is based on existing OMG standards, mainly UCM and UML.

Keywords: Component Based Software Engineering · Model based testing · UCM

1 Introduction

The development cycle of a software system involves a verification phase to ensure the system meets its requirements. A typical way of verifying a system is to test it. Test are often very expensive in terms of efforts. Model Based Testing (MBT) [11] is usually considered to be an efficient approach to cut test costs by modelling the expected system properties and automatically producing the tests themselves.

Yet, the creation and maintenance of the test specification models is still a source of difficulty. In particular, system requirements are likely to change during the development cycle, especially in agile processes [2]. The test specification models must be updated to follow the requirement changes.

In this paper, we present our approach to overcome consistency issues by deducing some test specification models from architecture specifications. We use a Component Based Software Engineering model as a unique reference, from which code and tests are generated. Thus we ensure consistency between software and tests, while reducing development cost. We focus on testing that the implementation code of a given component conforms with the a sequence of port calls specified for this component.

The paper explains the process we are currently implementing. We illustrate it with a very simple example. First, we provide a quick overview of UCM; we explain its scope and explain how we combine it with UML to gather all the necessary information to specify expected behaviours. Then we describe Diversity, a MBT tool we use to compare execution traces with the expected behaviours. Then we provide an overview of our process. We conclude the paper by discussing our solution. Our study is done in the scope of project DisTA[1].

2 Software Component Design with UCM

Component based software engineering (CBSE) addresses middleware dependency of software applications [8,9]. It consists in isolating the business code from the middleware configuration code. The business code is encapsulated inside components. Components are connected through ports and connectors. Glue code is generated from the component declarations. This glue code provides an API to the business code; this API depends only on the component declaration, and is independent from the underlying middleware. The implementation of the glue code API is specific to a given middleware implementation; it manages the middleware configuration and control.

UCM [5] is a component model published by the Object Management Group. It is a successor of the CORBA Component Model [3]. While CCM was initially bound to CORBA, UCM is independent from any middleware technology. Also, UCM is more focused on real-time embedded systems than CCM was.

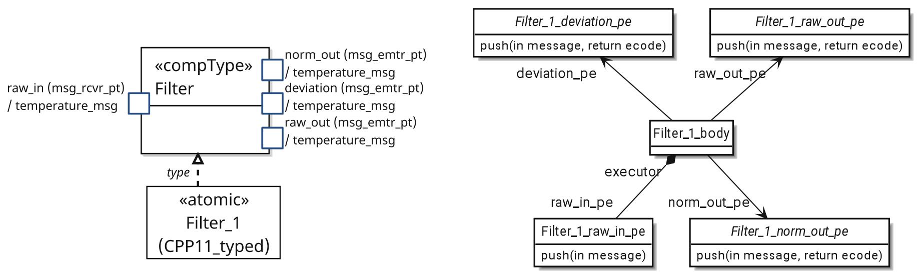
UCM defines three main entities: connectors, technical policies and components. Components are the application itself; they encapsulate the business code. Connectors and technical policies specify the execution platform that supports the execution of the application; tools generate technical code from them.

2.1 Declaration of UCM Components

Components are made of two parts: a component type and a component implementation. A component type specifies the possible interactions between the component and other components. Such interactions consist of ports. A component implementation carries the technical information related with the component realization. The UCM standard specifies how to interpret component declarations to produce the glue code between technical and business code. This part of the UCM standard is called the container model.

[1] https://www.linkedin.com/pulse/automated-distributed-test-platform-iot-testing-fabrice-trollet.

Figure 1a represents an example of UCM component declaration. We design a UCM component that should be connected to a thermometer and filter out erroneous temperature data. Component type `Filter` carries four ports. One is named `raw_in` and receives messages. The others are named `norm_out`, `raw_out` and `deviation`; they all emit messages. All messages carry temperature data—the details of the data binding are declared in binding `temperature_msg`, which is not represented here. The implementation `Filter_1` is implemented in C++11.



(a) Declaration (b) Structure

Fig. 1. Declaration and structure of UCM component `Filter_1`

Components are translated into one class that stores the component states, and classes that hold the business code of each provided interface. The container model class diagram for component `Filter_1` is represented on Fig. 1b. According to the definition of message ports in the UCM standard library, message reception and message emission are realized by methods named "push".

2.2 Adding Behaviour Specifications to UCM Compponents

UCM in itself does not specify component behaviours: the standard defines atomic components as black boxes that contain business code. The component developer needs additional information in order to correctly write the C++ code for `Filter_1`. In the scope of our work, we extend UCM by adding such information.

As the container model actually corresponds to basic UML class diagrams (see Fig. 1b), it is possible to combine UCM declarations with UML sequence diagrams [4] to specify expected behaviours inside components. Such sequence diagrams specify the behaviours of the method implementation code; They consist of sequences of calls to external methods, loops and alternatives.

Let us consider the following behaviour specification for component `Filter_1`. Upon the reception of a raw temperature data on port `raw_in`, `Filter_1` shall have different behaviours, depending on the value of the input temperature. If the input temperature is similar to the previous temperature data, send it through `norm_out`. If one sample of temperature input is obviously erroneous (i.e. very different from the previous value), send the old input value through `norm_out`. If the input temperature is repetitively erroneous, send it through `raw_out` and send the deviation between the current temperature and the last correct temperature through `deviation`. This specification is illustrated in Fig. 2.

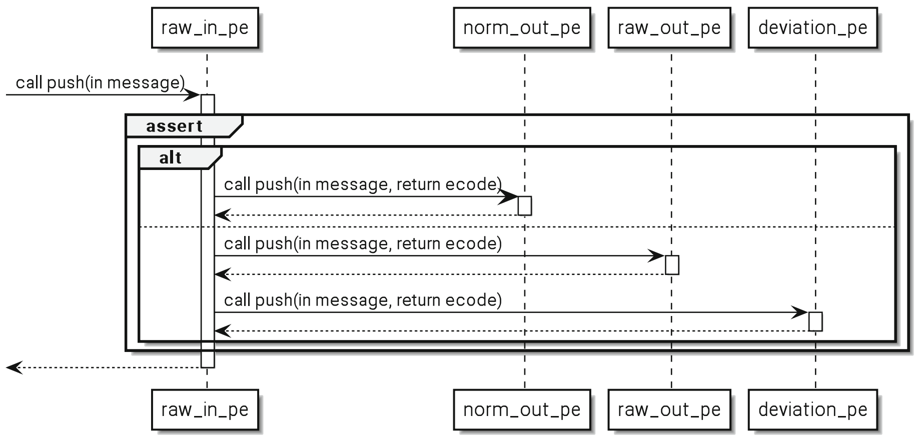


Fig. 2. Behaviour specification associated with UCM component `Filter_1`

The sequence diagram specifies requirements regarding relationships between calls of component port methods. In order to check if the component implementation code is correct with respect to the behaviour specification, we can trace all method invocations, execute the application and analyze the traces: the traces must *always* conform with the component behaviour specification.

In order to automate trace analysis, our solution is to create formal oracles from the sequence diagrams. From the component declaration (Fig. 1a) and the behaviour (Fig. 2) associated with the container model (Fig. 1b), we can build an oracle to check that each invocation of method `push` of port `raw_in` is either followed by a call to method `push` of port `norm_out` *or else* by a call to method `push` of port `raw_out` then a call to method `push` of port `deviation`. The remaining of the paper briefly explains how we do.

3 Coordinating Specification and Verification

Our work focuses on cutting the cost of verification in a software development process. Typical development processes, like the weel-known "V" cycle, involve three steps: specification, implementation and verification. Iterative process usually involves short development cycles that combine these three steps. In the scope of this paper, "verification" consists of checking the system properties against its requirements.

The verification steps consists of ensuring the implementation conforms to some oracles that reflect the specification. Hence, the oracles are supposed to be correct while the implementation might contain errors—that is why we perform verification. In Model Based Testing techniques, verification consists of tests. These tests are built from a model that describes the expected system behaviour. Yet, nothing ensures the test model is correct with respect to the initial specifications. This may lead to inconsistencies between specifications and tests.

Our work consists of automating as much as possible the production of both implementation and verification oracles from the specifications. This way, we reduce the risk of inconsistencies. Figure 3 illustrates the process we follow.

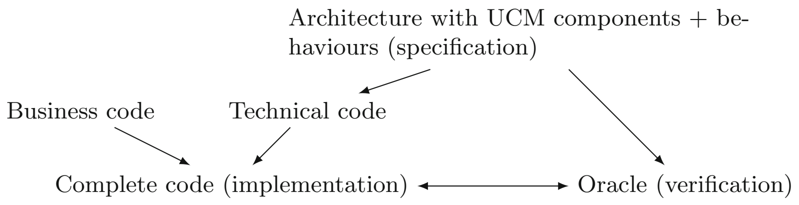

Architecture with UCM components + behaviours (specification)
Business code
Technical code
Complete code (implementation)
Oracle (verification)

Fig. 3. Production process

It is not possible to automate the complete generation of the implementation. This would mean the specifications gather all the implementation details (data structure, architecture and algorithms); that is, the specification would be the implementation. That is why we separate the business code (written by hand or produced by a third-party tool) and the technical code, which is generated from a specification of the architecture.

It is not possible to completely specify all possible tests either: this would imply a very complex and formal specification, difficult to create, maintain and understand for humans. In our work, we focus on testing the occurence of events in execution traces.

4 Model-Based Testing with Diversity and xLIA

Since the UCM standard is dedicated to the specification of real-time and embedded systems, it can be used to specify finite-state system, allowing model checking techniques to be used to verify the correctness of their implementation. Instead of requiring the user to create multiple models for a single application, our approach to model-based testing only requires a single model from which other models can be derived.

Diversity is a formal analysis tool based on symbolic execution developped by CEA LIST institute [1]. It supports the definition and symbolic execution of models and it can be used for automated test sets generation and for evaluating execution traces' conformance to models [7]. These models are concurrent communicating systems expressed in eXecutable Language for Interaction & Assemblage (xLIA).

The xLIA language offers a variety of primitives that allows it to encode classical semantics such as UML and SDL (Specification and Description Language). In this paper, we describe how to use xLIA as a pivot language to create timed input output labelled transition systems (TIOLTS) that can be used by Diversity.

In the xLIA models we produce, the different state machines can communicate with each other through rendez-vous interactions. State machines can also communicate with the environment (i.e. what is external to the system under test).

5 Mapping from UCM to xLIA

A UCM specification can contain execution scenarios attached to methods. An execution scenario contains execution steps that will be executed upon the invocation of the method. These scenarios are what will be translated into xLIA in order to automatically generate test oracles.

The translation follows these general principles: each scenario is translated to a state machine and each step contained by that scenario is translated to a state. Since a scenario's steps are ordered, the transitions from one state to the next follow that order.

Some steps are calls to other methods, they are named "call steps". Such steps are assigned a subscenario, which is a reference to the execution scenario of the method they call. The translation of a call step adds an instruction on the exiting transition of the state. This instruction makes the state machine that represents the scenario interact with the state machine that represents the subscenario through their ports. Most call steps have to wait for their subscenario to terminate before allowing the scenario to continue its execution. This is translated to a transition into a "waiting" state that waits for a new interaction between the two state machines.

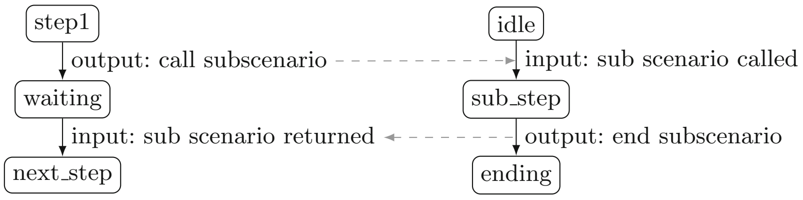


Fig. 4. Communication between state machines.

As scenarios can be executed multiple times throughout the execution of the application, the state machines that represent them have to be cyclic. Therefore, every state machine has an "idle" and an "ending" state that are respectively the first and the last step of one cycle of their execution. In the “idle” state, a state machine waits for a timed guard or an interaction from either another state machine or the environment to begin its execution. The execution traces will then be confronted to the outputs produced upon exiting both these states.

Other kinds of steps can be used to specify an application’s behaviour in UCM: alternative and loop steps. The alternative step offers a set of steps among which only one will be executed without specifying the condition used to make the decision. The loop step contains a list of steps that will be executed multiple times, the minimum and maximum number of iterations have to be specified.

An alternative step is translated into a state that has multiple nondeterministic exiting transitions, one for each possible next step. A loop state is translated to a state that has two transitions: one to the first state of the loop and one to the next state. Nondeterminism can be used to translate alternative and loop steps to xLIA because Diversity’s symbolic execution engine will explore every possibility. In the following example, i is initialized to 0 and the loop iterates between 2 and 5 times. This is nondeterministic for i $\in \{2, 3, 4\}$.

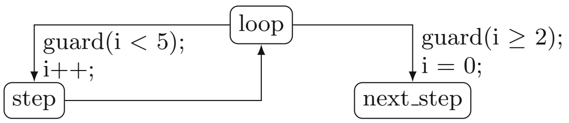


Fig. 5. Example of nondeterminism in loops.

6 Implementation and Experiments

We are currently developping a UCM code generator, Sigil-UCM [10], to demonstrate our approach. This tool is a code generator that implements the standard mapping from UCM to C++ and also the mapping from UCM to xLIA we outlined in the previous section.

Sigil-UCM produces the necessary calls to the log4cpp [6] library to trace the execution of all component port methods (method entry, method exit, method call, returning from call). Traces are compared with the xLIA test oracles by Diversity. Users get a test verdict that indicates if the execution traces conforms to the call sequences specified in the UCM components or not.

After generating the technical code with Sigil-UCM, users have to implement the methods for which they specified the behaviour in the UCM model. In our example, users have to implement method `push` whose behaviour is specified by the sequence diagram in Fig. 2.

Let us suppose users misunderstood the component specifications and wrote business code that systematically produces output to `norm_out`, even for erroneous data cases. The component execution produces the traces illustrated in listing 1.1.

Listing 1.1. Execution trace produced by an erroneous implementation

```
1 TRACE log_policy : filter.logger (raw_in__pe) entering push
2 TRACE log_policy : filter.logger (norm_out_pe) calling push
3 TRACE log_policy : filter.logger (norm_out_pe) returning from push
4 TRACE log_policy : filter.logger (raw_out_pe) calling push
5 TRACE log_policy : filter.logger (raw_out_pe) returning from push
6 TRACE log_policy : filter.logger (deviation_pe) calling push
7 TRACE log_policy : filter.logger (deviation_pe) returning from push
8 TRACE log_policy : filter.logger (raw_in_pe) exiting push
```

Users launch Diversity to compare the traces produced by the execution of the business code and the xLIA oracle generated from the specifications. The automatic analysis with Diversity produces the verdict *FAIL* because the traces do not match the test oracle. Indeed, after receiving a message, the business code implementation of `push` emits on all its outgoing ports instead of either emitting on `norm_out` or on `raw_out` and `deviation`.

The automatic analysis performed by Diversity indicates that line 4 in listing 1.1 caused the failure. This helps understand the implementation of the alternative is incorrect. After fixing the implementation of `push` and relaunching the execution, the component produces the execution trace shown in listing 1.2.

Listing 1.2. Execution trace produced by a correct implementation

```
1 TRACE log_policy : filter.logger (raw_in_pe) entering push
2 TRACE log_policy : filter.logger (raw_out_pe) calling push
3 TRACE log_policy : filter.logger (raw_out_pe) returning from push
4 TRACE log_policy : filter.logger (deviation_pe) calling push
5 TRACE log_policy : filter.logger (deviation_pe) returning from push
6 TRACE log_policy : filter.logger (raw_in_pe) exiting push
```

Now the automatic analysis with Diversity produces verdict *PASS*. This indicates the business code implementation of method `push` conforms to the specification.

7 Conclusion

In this paper, we gave the main lines of our work to ease the use of Model Based Testing in a software development process. We demonstrated Component Based software Engineering can be used to generate both technical code and test models. Having a unique, central model prevent inconsistencies between code and tests, especially for agile development processes in which application specifications are likely to evolve over time. From an industrial process point of view, it is convenient to manipulate a unique set of UML based models: it requires little additional effort for component designers to specify behaviours in order to get test oracles at almost no cost. This eases the smooth adoption of MBT techniques in industry.

References

1. Eclipse format modeling project. https://projects.eclipse.org/proposals/eclipse-formal-modeling-project
2. The agile manifesto (2001). http://agilemanifesto.org/principles.html
3. CORBA Component Model, version 4.0. OMG (2006). https://www.omg.org/spec/CCM/
4. Unified Modeling Language, version 2.5.1. OMG (2017). https://www.omg.org/spec/UML/
5. Unified Component Model for Distributed, Real-Time And Embedded Systems, version 1.1. OMG (2019). http://www.omg.org/spec/UCM
6. Bakker, B.: Log for c++ project. http://log4cpp.sourceforge.net
7. Bannour, B., Escobedo, J.P., Gaston, C., Le Gall, P.: Off-line test case generation for timed symbolic model-based conformance testing. In: Nielsen, B., Weise, C. (eds.) ICTSS 2012. LNCS, vol. 7641, pp. 119–135. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34691-0_10
8. Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.V.: A classification framework for software component models. IEEE Trans. Softw. Eng. **37**(5), 593–615 (2011). https://doi.org/10.1109/TSE.2010.83
9. Szyperski, C.A., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming. Addison-Wesley Component Software Series, 2nd edn. Addison-Wesley, Boston (2002). http://www.worldcat.org/oclc/248041840
10. Sigil-ucm. https://www.thalesforge.thalesgroup.com/projects/sigil-ucm/
11. Utting, M., Legeard, B., Bouquet, F., Fourneret, E., Peureux, F., Vernotte, A.: Recent advances in model-based testing. Adv. Comput. **101**, 53–120 (2016)