Building Test Harness from Service-based Component Models

Pascal André, Jean-Marie Mottu, and Gilles Ardourel

LINA CNRS UMR 6241 - University of Nantes
2, rue de la Houssinière, F-44322 Nantes Cedex, France
firstname.lastname@univ-nantes.fr

Abstract. In model-driven development, the correctness of models is essential. Testing as soon as possible reduces the cost of the verification and validation process. Considering separately PIM and PSM reduces the test complexity and helps the evolution of the system model. In order to test models before their final implementation, we propose an approach to guide the tester through the process of designing tests at the model level. We target model testing for Service-based Component Models. The approach produces test harness and provides information on how to bind the services of the components under test and how to insert the relevant data. The tests are then executed by plunging the harness into a technical platform.

Keywords: Test Harness, Model Conformance, Test Tool, MDD

1 Introduction

Testing as early as possible reduces the cost of Verification and Validation (V&V) [1]. In Model-Driven Development (MDD), V&V is improved by directly testing models for correctness [2]. Testing models helps to focus the effort on the early detection of *platform independent* errors, which are costly when detected too late. Testing models does not consider implementation specific errors and thus reduces the complexity of testing [3]. Designing the tests on models ease their adaptation when models are refactored. However, building and running tests on models remain a challenge.

We target service-based component models with behaviour, including data and communications. The specification detail level is sufficient enough to enable various kinds of test at the model level. While testing such software components, the encapsulation should be preserved. Therefore, finding where and how providing test data is difficult: variables are accessed through services, which are available from the interface of the component but also from other required services. Designing the *test fixture* of a service implies a sequence of service calls additionally to the component initialisation. In [4], Gross mentioned issues in Component-Based Software (CBS) testing: testing in a new context (i.e. development context one, deployment context ones), lack of access to the internal

working of a component. Ghosh et al. [5]. also identify several problems: the selection of subsets of components to be tested and the creation of testing components sequences. Our work is concerned with these issues.

We consider the building of *test harness* for unit and integration testing dedicated to Service-based Component Systems. Test harnesses are used to provide test data, to run the test, to get the verdict (fail or pass). We create test harness from the platform independent model (PIM) of the system under test (SUT) and from test intentions. Those intentions declare which services are tested in which context, with which data. The tester incrementally builds a harness to run the test in an appropriate context, especially without breaking the encapsulation. The tester needs assistance for this task.

In this paper, we propose an approach assisting the test of service-based component models. This approach integrates the tests at the model level and is provided with tools. *First*, we propose operating at the model level, we modelise the tests at the same abstraction level as the component model. We build *test harnesses* as component systems made of Components Under Test (CUT) and Test Components (TC), which are created to test. As a consequence, the assembly of test components and regular components can benefit from the development tools associated with the component model. Additionally, test components could be transformed into platform specific models, thus capitalising the early testing effort. *Second*, we propose a guided process to assist the tester to manage the complexity of the component under test. The test harness is obtained from transformations of the SUT, guided by the test intention, enriched by test components, and assisted by several analyses and heuristics proposing relevant information and ensuring model correctness through verifications. *Third*, we illustrate the approach on a motivating example, a platoon of vehicles. We have developed a set of prototype tools designed to experiment the proposals. It is implemented in COSTO, the tool support of the Kmelia component model [6].

2 Testing Service-based Component Model

This section introduces the testing of service-based component model and the challenges to be tackled. It describes our approach on a motivating example.

2.1 Service-based Component Model

A *component system* is an assembly of *components* which services are bound by *assembly links*. The interface of a component defines its *provided and required services*. The interface of a service defines a *contract* to be satisfied when calling it. The service may communicate, and the assembly links denote communication channels. The set of all the services needed by a service is called its *service dependency*. The required services can then be bound to provided services. These needs are either satisfied internally by other services of the same component, or specified as required services in the component's interface and satisfied by other components. A *composite* component encapsulates an assembly.

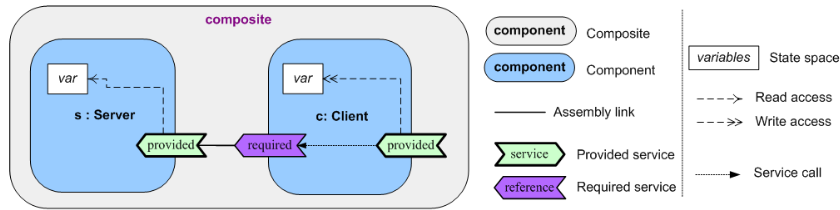

Fig. 1. Component and service notation

In the example of Figure 1, the component c:Client requires a service *provided* by the component s:Server, through an assembly link. This assembly is illustrated with the SCA notation [7]. We extended the SCA notation (right part of the legend of Figure 1) to make explicit the dependencies and data useful for testing: (i) typed data define the component state, (ii) services access these variables, (iii) provided services internally depends on required services.

Figure 2 represents a motivating example: a simplified platoon of several vehicles. Each vehicle computes its own state (speed and position) by considering its current state, its predecessor's state, and also a safety distance between vehicles. Each vehicle is a component providing its speed and position and requiring predecessor's speed and position. Three vehicles are assembled in that example. Each vehicle provides a configuration service conf initiating its state, a service run launching the platoon and requiring the service computeSpeed to calculate new position and speed. The leader is another component controlling its own values according to a position goal.

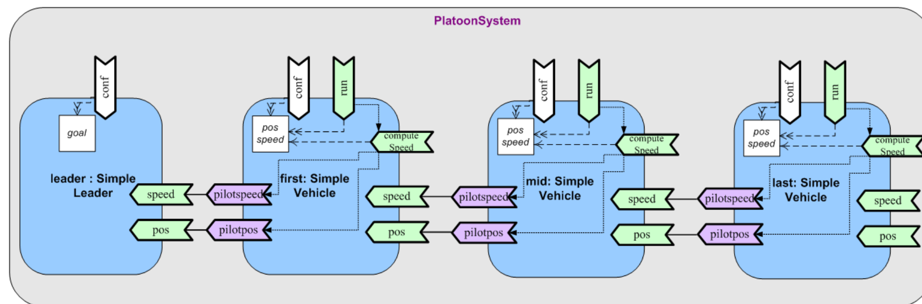

Fig. 2. Example of a component model of the Platoon system

2.2 Testing Components at the Modelling Stage

Conformance testing aims to control that system under test behave correctly according to their specification. With Service-based Components, we test that

the behaviour conforms the specification of their interfaces. In the context of Service-Based Component Applications, the test harness is designed by selecting the components under test, replacing their clients and their servers (the first with mocks, the last with test drivers), then providing test data to the test drivers and getting their verdicts [8]. Test harness should preserve component encapsulation. JUnit classes are an example of test harnesses for Java OO programs.

In MDD, Service-based Component System is modelised with a PIM, and part of its components (the SUT) is tested depending on a *test intention*. The test intention defines for each test which part of the system is tested (component(s)), with which entry point (a service), in which situation (SUT state), with which request (service call with parameters), expecting which results. At the beginning, the test intention is tester's knowledge, then she has to concretely prepare and run the test to get a verdict from oracles.

Testing such a SUT is not just creating test data and oracles, the difficulty is to consider components in an appropriate context (assembly of components), to run the test data on them, get the output to be checked with the oracles. The building of test harness to fulfil this task is an important work. Moreover, the process of creating a test harness is not trivial for models of components.

We consider several characteristics making the test of component models an original process: (i) Early specifications are often too abstract or incomplete to be executable. In order to test service-based component, we have to execute it in a consistent environment: its dependencies have to be satisfied by compatible services and every operation used in the environment has to be concrete enough or bound to a concrete operation. Furthermore, its environment must be able to run test cases. (ii) In strongly encapsulated components, finding where and how to insert data is challenging: variables are usually only accessible through services and some data is required from external services. (iii) At the model level, tests should be designed as models. Describing test artefacts like *mocks* and *drivers* in the component modelling language provides several benefits: it is easy to communicate with designers; tests follow the same validity rules as the model for both consistency and execution, allowing the use of the development tools associated with the component model. (iv) Models are subject to frequent refactoring or even evolutions, which can compromise the applicability of existing tests.

In this work, we are concerned with the building of test harness for service-based component models. The tester chooses the components and their services she tests (defined with the test intention). She orders the tests. She creates the test data based on existing works considering the specification of component: work of [13] with state machines, work of [14] performing LTS behaviours testing, etc. She creates oracles, for instance, based on the contracts usually available defining a service interface. Our approach assists the tester in building the test harness allowing her to run such test data and to apply such oracles.

3 Assisting the test harness building

The construction process takes as input the System Under Test (SUT) model which is a PIM (Platform Independent Model) and a test intention (Figure 3). It is made of three activities (transformations). The first one builds the *test harness* as an assembly with the SUT in a *Test Specific Model* (TSM) at the model level. The second composes the harness with a *Platform Description Model* (PDM) to get an executable model (code). During the execution, the concrete data providers give the test data needed to run the test, and the concrete assert functions return the verdicts. Designing the oracles and the data sets influences the operational level. This point is out of concern in the rest of the paper.

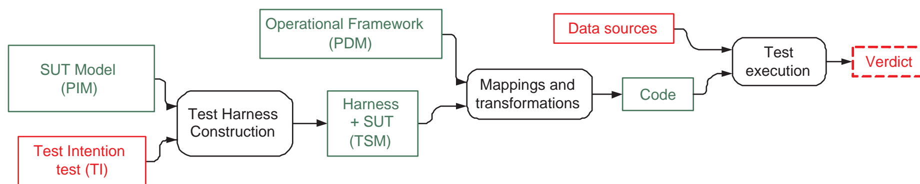

Fig. 3. Testing process overview

3.1 Test Harness Construction

This section explains how the tester is guided to build a consistent test harness. As noted in [5] (and filed under the issue C3), it is sometimes necessary to consider different subsets of the architecture when testing a component, because of the influence (e.g. race conditions) of the other components on the SUT. The test harness is designed by (i) selecting the relevant subset of original components and services to test, (ii) replacing all or parts of their clients (resp. servers) by test *drivers* (resp. *mocks*), (iii) providing data sources. The test intention serves as a guideline during the construction process.

As an example, we are intent on testing the conformance of the computeSpeed (safeDistance : Integer) : Integer service of the mid component with the following *safety property*: *the distance between two neighbour vehicles is greater than a value safeDistance*. The service behaviour depends on (a) the recommended safe distance from the predecessor, (b) the position and speed of the vehicle itself and its predecessor. Coming back to the example of Figure 2, testing the computeSpeed service of mid implies to give a value to the safeDistance parameter, to initialise the values of the pos and speed variables, which are used by the computeSpeed service and to find providers for pilotspeed et pilotpos which are required by computeSpeed. Only one component vehicle is under test here because other ones are not necessary to test the computeSpeed service, but a more complex architecture could have been retained.

The challenge for the tester is to manage the way these data can be provided. They can be provided by the test driver, by the configuration step, or by other

components (original components or mock components). Finding the service to be invoked in order to set the component in an acceptable state for the test is not trivial. Our goal is to help the tester to manage this complexity, in order to reduce testing effort. At this stage, assisting the harness building consists in:

1. Detecting missing features between the SUT and the test intention. This detection is achieved by the verification of two properties:
   (a) *Correctness*. All the service dependencies are satisfied in the scope of the test harness. No pending or incompatible dependencies remain.
   (b) *Testability*. All the formal test data are linked to values in the TSM (variables, parameters, results...).
2. Proposing candidates for the missing features: which service can configure data, can handle the test data or can fulfil missing required services?
3. Generating and linking test components (mocks, drivers as mirror services, bindings, abstract functions...).

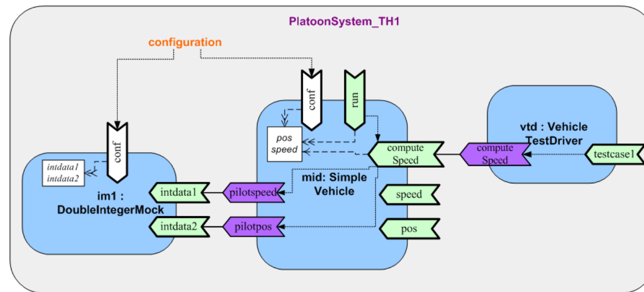

Fig. 4. A TSM : test harness for the mid component's service computeSpeed (SUT)

In the TSM of Figure 4, the vtd test driver is responsible to put the computeSpeed service in an adequate context (providing input data and oracle). The service testcase1 of vtd contains only a computeSpeed call and an oracle evaluation. To fix the computeSpeed requirements, the test designer could (1) ask for mock components (with random values), (2) reuse the first component coming from the SUT (and fulfil the requirements of first ...) or (3) design its own mock. This variability enables several kinds of test and answers to the issue C3 of Ghosh et al. [5]. Note that a single integer mock component im1 is used here because it offers a better control of the delivered speed (intdata1) and position (intdata2), but a harness with two independent integer mocks providing a single intdata service would also be possible.

The *data* of computeSpeed test harness are bound in the following way: (a) the safeDistance parameter is assigned in the call sequence of the test driver, (b) the position and speed (pos and speed variables) of the mid vehicle under test are bound to the conf service of mid, (c) the im1 mock component playing the role of the predecessor is configured to return the position and speed of the predecessor.

3.2 Mappings and Transformations

According to the MDD principles, the test will be executed by composing the test harness with the PDM (Figure 3). The test data and test primitives (oracle...) are provided by the PDM test support or implemented by the tester. The PIM may include primitive types and functions (numbers, strings, I/O...) that must also be mapped to the code level. These mappings are predefined in standard libraries or user-defined. High-level TSM primitives are connected to low level (PDM, code) functions, as illustrated by Figure 5. If the mapping is complete and consistent, then the model is executable.

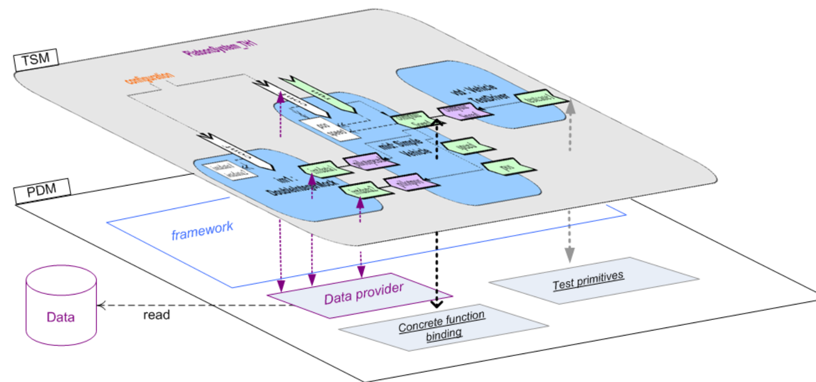

Fig. 5. Test harness Concrete data and Function Mapping

The goal of the *Mappings and Transformations* activity (Figure 3) is to fix this composition. Each service involved in the test must be executable in a consistent environment: its dependencies must be satisfied by compatible services, and every operation used in the environment must be sufficiently concrete or related to a specific operation. Every abstract type should be mapped to a concrete type. At this stage, assisting the harness building consists in (1) detecting missing mappings between the TSM and PDM (*Executability*), (2) proposing tracks for the missing mappings based on signatures, pre/post conditions... (all the primitive types and functions are mapped to concrete ones, the test data inputs have concrete entry points), (3) generating standard primitive fragments (idle functions, random functions...). The mappings are stored in libraries in order to be reused later and the entries can be duplicated to several PDM. This activity induces an additional cost at model level, but the errors detected are less expensive to solve than those of components and services. Moreover, if the component model is implemented several times targeting several PDM, the tests would be reused and part of the behaviour would have been checked before runtime, as proposed here.

4 Experimentation

The test process of Figure 3 has been instrumented using the COSTO/Kmelia tool[1]. Kmelia is a wide-spectrum language dedicated to the development of correct components and services [6]. Kmelia includes an abstract language for computations, instead of links to the source code as in related languages like SCA, AADL, Sofa or Fractal. This layer is useful to check models before transforming to PSM. Kmelia is supported by the COSTO tool, a set of Eclipse plugins including an editor, a type checker, several analysis tools.

The test harness construction has been experimented on the platoon example. The goal was to build a harness to test the conformance of the service computeSpeed as introduced in Section 3.

- The test intention is a special Kmelia component, where only the name and description are mandatory. This trick enables to reuse the tool facilities and to consider test design at the model level.

```
TEST_INTENTION PlatoonTestIntention
DESCRIPTION "the vehicle will stop if it is too close to the previous one"
INPUT VARIABLES
  pos, previous_pos, mindistance:Integer;
OUTPUT VARIABLES
  speed:Integer
ORACLE
  speed=0
```

- During the process, the building tool (1) starts from a test intention, (2) asks the user to select target system and services (or proposes some if the test intention is detailed), (3) displays the board to match the test intention with the SUT (Figure 6), (4) proposes candidates, (5) checks the correction and testability properties, (6) goes back to step (2) until the TSM is complete, (7) checks the executability properties and proposes candidates (from the libraries) for the missing elements, (8) generates the code.

A web-appendix[2] shows the details of this example.

The harness building activities (transformations, decisions, assistance and generation) are implemented and integrated as COSTO functionalities. We developed the PDM framework in Java (7 packages, 50 classes, 540 methods and 3400 LOC). The service instanciation uses Java threads with an ad-hoc communication layer based on buffered synchronous channels (monitors). We keep strong traceability links from the code level to the PIM level for a better feedback on errors and also for animating the specification at the right level (GUI). We add specific support for contract checking (assertions and oracles).

Compared with the Junit practice, the tester could focus on the "business" part and did not need to care with Java details of the implementation. While the specification of the components was available, the tester could not modify them to make the test case more easy to write. He discovers the specification elements on-the-fly when mapping them to the test intention.

[1] http://www.lina.sciences.univ-nantes.fr/aelos/projects/kmelia/

[2] http://www.lina.sciences.univ-nantes.fr/aelos/download/ModeVVa_app.pdf

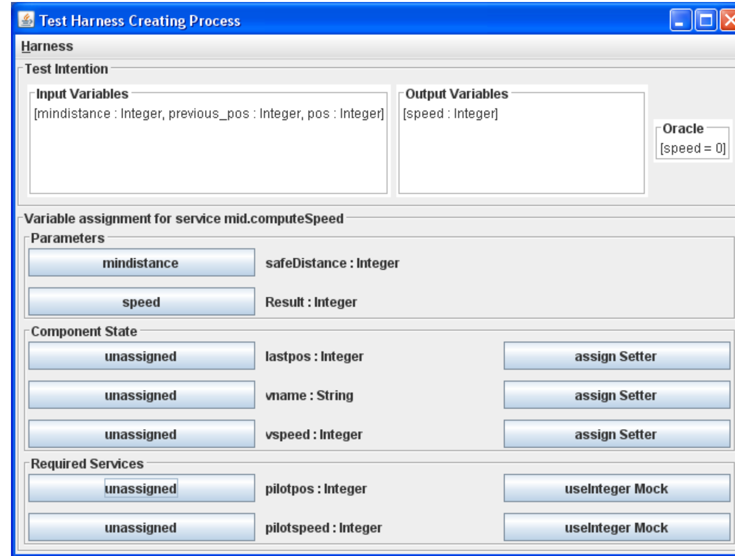

Fig. 6. Component and service notation

5 Related Work

There are several works interested in generating tests for testing components.

In [9], Mariani et al. propose an approach for implementing self-testing components. They move testing of component from development to deployment time. In [10], Heineman applies Test Driven Development to component-based software engineering. The component dependencies are managed with mocks, and tests are run once components can be deployed. In contrary, in our proposal we propose to test the components at the modelling phase, before implementation.

In [11], Edwards outlines a strategy for automated black-box testing of software components. Components are considered in terms of object-oriented classes, whereas we consider component as entity providing and requiring services.

In [12], Zhang introduces test-driven modeling to apply the XP test-driven paradigm to an MDD process. Their approach designs test before modelling when we design test after modelling. In [13], the authors target robustness testing of components using rCOS. Their CUT approach involves functional contracts and a dynamic contract. However, these approaches apply the tests on the target platform when we design them at the model level and apply them on simulation code.

6 Conclusion

In this paper, we described a method to integrate testing early in an MDD process, by designing test artefacts as models. The test designer is assisted in

building component test harnesses from the component model under test and test abstract information through a guided process. The COSTO/Kmelia framework enabled to implement the process activities and to run the tests on the target specific execution platform with a feedback at the model level. The benefits are a shorter test engineering process with early feedback.

In future work, we will study different kinds of oracle contracts in order to find which logics are best fitted to express them in a testable way as well as ensuring a good traceability of the verdict. We will apply *mutation analysis* on the models rather than on the generated code because such an approach can be integrated in MDD while respecting the hypotheses of the mutation analysis.

References

1. G. Shanks, E. Tansley, and R. Weber, "Using ontology to validate conceptual models," *Commun. ACM*, vol. 46, no. 10, pp. 85–89, Oct. 2003.
2. M. Gogolla, J. Bohling, and M. Richters, "Validating uml and ocl models in use by automatic snapshot generation," *Software and Systems Modeling*, vol. 4, no. 4, pp. 386–398, 2005.
3. M. Born, I. Schieferdecker, H.-g. Gross, and P. Santos, "Model-driven development and testing - a case study," in *First European Workshop on MDA with Emphasis on Industrial Application*. Twente Univ., 2004, pp. 97–104.
4. H.-G. Gross, *Component-based Software Testing With Uml*. SpringerVerlag, 2004.
5. S. Ghosh and A. P. Mathur, "Issues in testing distributed component-based systems," in *In First International ICSE Workshop on Testing Distributed Component-Based Systems*, 1999.
6. P. André, G. Ardourel, C. Attiogbé, and A. Lanoix, "Using assertions to enhance the correctness of kmelia components and their assemblies," *ENTCS*, vol. 263, pp. 5 – 30, 2010, proceedings of FACS 2009.
7. OSOA, "Service component architecture (sca): Sca assembly model v1.00 specifications," Open SOA Collaboration, Specification Version 1.0, March 2007, online http://www.osoa.org.
8. C. R. Rocha and E. Martins, "A method for model based test harness generation for component testing," *J. Braz. Comp. Soc.*, vol. 14, no. 1, pp. 7–23, 2008.
9. L. Mariani, M. Pezzè, and D. Willmor, "Generation of integration tests for self-testing components," in *FORTE Workshops*, ser. LNCS, vol. 3236. Springer, 2004, pp. 337–350.
10. G. Heineman, "Unit testing of software components with inter-component dependencies," in *Component-Based Software Engineering*, ser. LNCS. Springer Berlin / Heidelberg, 2009, vol. 5582, pp. 262–273.
11. S. H. Edwards, "A framework for practical, automated black-box testing of component-based software," *Softw. Test., Verif. Reliab.*, vol. 11, no. 2, pp. 97–111, 2001.
12. Y. Zhang, "Test-driven modeling for model-driven development," *IEEE Software*, vol. 21, no. 5, pp. 80–86, 2004.
13. B. Lei, Z. Liu, C. Morisset, and X. Li, "State based robustness testing for components," *Electr. Notes Theor. Comput. Sci.*, vol. 260, pp. 173–188, 2010.
14. B. Schätz and C. Pfaller, "Integrating component tests to system tests," *Electr. Notes Theor. Comput. Sci.*, vol. 260, pp. 225–241, 2010.