COSTOTest: a tool for building and running test harness for service-based component models (demo)

Pascal Andre, Jean-Marie Mottu, Gerson Gerson Sunyé

Demo: COSTOTest: A Tool for Building and Running Test Harness for Service-based Component Models

Pascal André[1], Jean-Marie Mottu[1,2], and Gerson Sunyé[1,2]

[1] LINA - University of Nantes, France
[2]Inria, Mines Nantes
[pascal.andre,jean-marie.mottu,gerson.sunye]@univ-nantes.fr

April 26, 2017

Abstract

Early testing reduces the cost of detecting faults and improves the system reliability. In particular, testing component or service based systems during modeling frees the tests from implementation details, especially those related to the middleware. COSTOTest is a tool that helps the tester during the process of designing tests at the model level. It suggests the possibilities and the lacks when (s)he builds test cases. Building executable tests is achieved thanks to model transformations.

1 Introduction

Early testing reduces the cost of Verification and Validation (V&V) [1]. In Model-Driven Development (MDD), models describe the system and can be tested to validate its correctness [2]. Testing models reduces V&V complexity [3]: it helps to focus on platform-independent faults, which are costly corrected later (they can be spread in the code).

In [4], we considered early testing of Service-based Component (SBC) Models to detect platform-independent errors *i.e.* at the model level. Testing such applications' code is tricky due to the middleware information that implements communications. Nevertheless, testing such applications' model introduces some problematics, that we considered in [4]. In particular, as mentioned by Vincenzi et al. [5], *encapsulation* decreases testability (*i.e.* observability and controllability). Moreover, a tester should manage the *dependencies* between components, fulfilled by their services. Our contribution is an integrated and assisted

approach to build and to execute test cases on these models. Our approach provides assistance to explore the model elements still preserving encapsulation and dependencies.

Usually, a tester uses a *test harness* to provide the test data to the component under test, and to run the test cases. COSTOTest helps the tester in building such test harnesses. Our postulate is to consider a *test harness* like a plain SBC model: we therefore reuse the full SBC development tool kit (edition, validation, code generation). We promote *testing models by models*. This is different than following a model-based testing approach where models (e.g. UM models) are used to compute test suites that are implemented with deployment code (e.g. Java). This last approach has been considered by Rocha et al. [6]. Here the *model testing approach* considers the test on the models themselves.

This paper presents the *COSTOTest* tool, a practical answer to the problem of building and running test harnesses for Service-based Component Models. The demonstration overviews the approach and shows its feasibility. The tool provides facilities to bypass encapsulation in order to build executable test cases. The tool is available online at `http://costo.univ-nantes.fr/`.

2 Testing Models by Models

COSTOTest aims to assist testers during SBC testing [5, 7], assuming service's contracts to improve the quality [8, 9].

We depict a motivating example at the top of Figure 1 : a platoon of vehicles (illustrated with an extended SCA notation [10]). A *component system* (a platoon) is an assembly of *components* (vehicles) which services are bound by *assembly links*. A component encapsulates its state with internal variables (own speed and position of a vehicle).

The interface of a component defines its *provided and required services* (each vehicle provides its speed and position to its follower which requires them). 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.

Any provided service can require data:

- internally to internal variables,
- internally to its own provided services (e.g. the service run requires the computeSpeed service to update the speed of its component),
- externally to required services in the component's interface which are satisfied by other components (the service under test in the rest of the paper computeSpeed calculates the new speed of a component based on its speed, position, predecessor's speed and position).

Build test harness as a model

A tester may test the service computeSpeed, which is associated to the component named mid with the following *safety property*: *the distance between two*

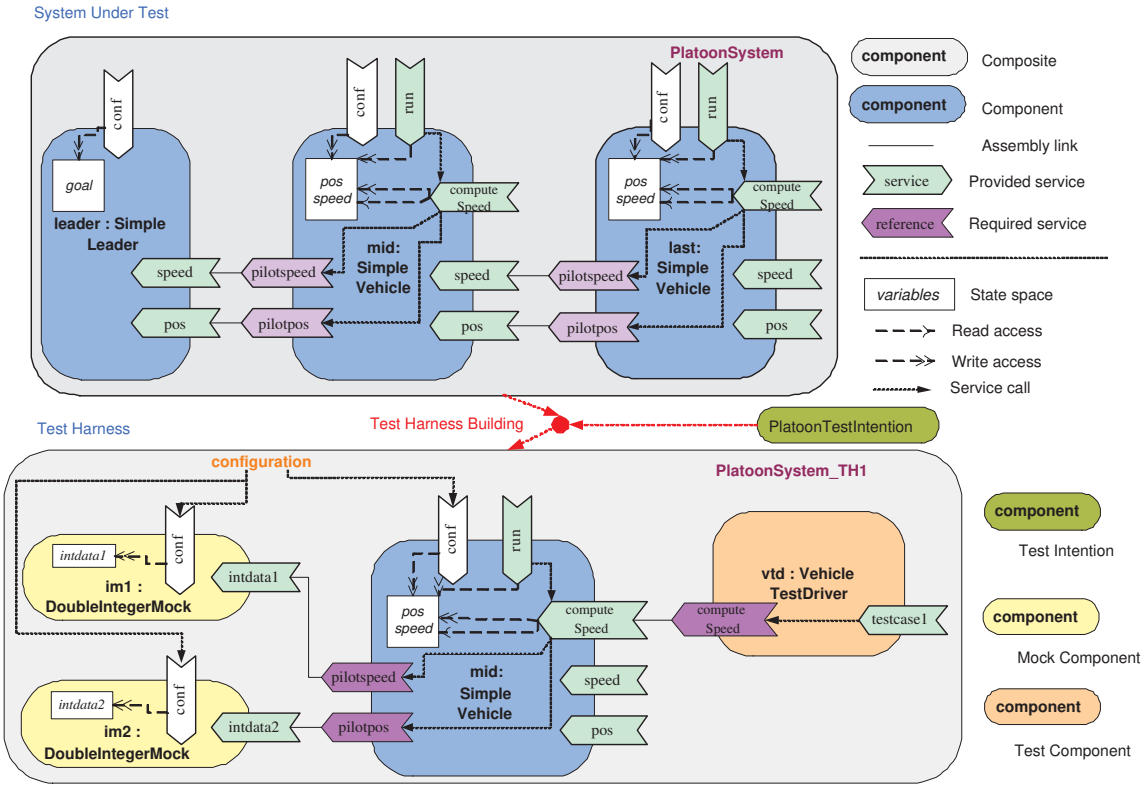

Figure 1: A test harness built from the Platoon system according to a test intention

neighbour vehicles is greater than a value safeDistance. The service behaviour is *dependent* on (i) the recommended safe distance from the predecessor, (ii) the position and speed of the vehicle itself and of its predecessor.

Testing the computeSpeed service of the component mid requires to affect *encapsulated* variables: give a value to its safeDistance parameter, initialise the values of the currentPos and currentSpeed variables, which are used by the compute Speed service, and find providers for pilotspeed and pilotpos which are required by computeSpeed.

The model of the test harness (bottom of Figure 1) is build from the *System Under Test (SUT) e.g.* the PlatoonSystem composite (top of Figure 1) and a *test intention* (middle of Figure 1). The test intention provides the list of which variables will be part of each test data (such as the above mentioned variables: position, speed, previous position, previous speed, and safe distance), and what would be the oracle data (such as the new own speed which is expected to have an expected value). It is provided by a TestIntention component.

COSTOTest helps the tester to (i) build the test harness (establish a consistent and complete context for testing) as illustrated by the bottom of Figure 1, and (ii) run the test cases with the test data values provided by the tester.

The advantages of testing models by models are those of early testing but also the possibility of reusing the rich modelling tools panel, and of providing an adequate framework for co-evolution between the System Under Test Model and the Test Harness model.

3 COSTOTest: a Testing Assistant

COSTOTest assists the tester in managing the way the test data can be provided: some of them by the configuration service, other ones by mock components, and the oracle by a test driver, as illustrated at the bottom of Figure 1. To achieve this, the tool helps the tester:

- to select the services and components from the SUT model according to a *test intention*;
- to check the test harness assembly correctness and completeness, satisfying assembly constraints;
- to bind required services to mocks provided in COSTOTest libraries;
- to check the test harness consistency and completeness regarding its test intention (that may be improved/completed during the test harness building);
- to generate a test component including the testcase services *e.g.* `vtd` in Figure 1;
- to launch the test harness with several test data values sets and to collect the verdicts.

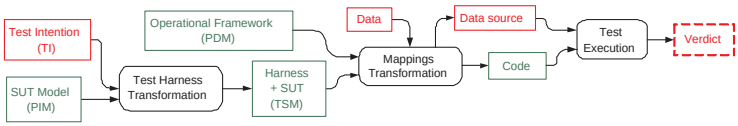

Figure 2: Testing process overview

Model transformation approach

The testing process is a sequence on model transformations which successively merge models, integrating features into them, as illustrated Figure 2. The input the System Under Test is a PIM (Platform Independent Model) of the SBC and a *Test Intention* is also a model described with a Domain Specific Language (DSL) *cf.* Figure 3. The process is made of two successive model transformations which return an executable code of the test harness.

The first model transformation is a model-to-model transformation. It builds the *test harness* as an assembly of selected part of the SUT with test components (mocks, test driver), and returns a *Test Specific Model* (TSM). It is semi-automatic: the test intention is provided by the tester and COSTOTest asks her/him to make choices, that are selected based on static analysis of the PIM. During this first step, the aim for the tester is to build a harness such as the one illustrated in the bottom of Figure 1 : PlatoonSystem_TH1.

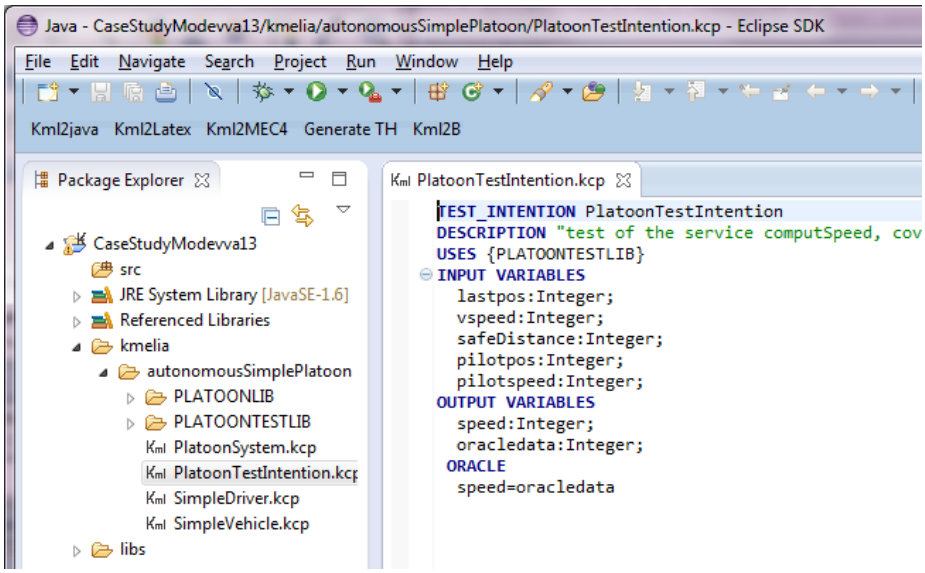

Figure 3: Test Intention

The second transformation is a model-to-code transformation, COSTOTest generates the code to simulate the behaviour of the harness. It merges the harness with a *Platform Description Model* (PDM) to get code (Java code in this case). It can be executed, because the model of the components describes the behaviour of the services, in the form of Communicating Finite State Machines. The test data and test oracle providers are designed in the PDM, thanks to the input "Data". A "data source" is generated, it is an XML file, with a structure corresponding to the test intention, and that the tester should fulfil with concrete values.

Implementation

In the current version of COSTOTool, the models (PIM and TSM) are described with the Kmelia modelling language [9], and the PDM framework is written in Java with an ad hoc communication layer for services and components. We can develop other PDM dedicated to different implementation languages, and to support different modelling languages.

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 defined by the user. High-level TSM primitives are auto-

matically connected to low level (PDM, code) functions, as illustrated in Figure 4. If the mapping is complete and consistent, then the model is executable.

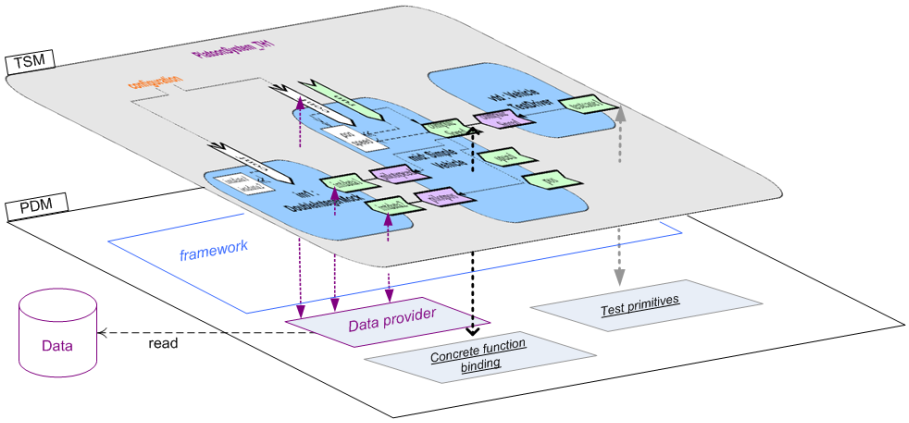

Figure 4: Test harness Concrete data and Function Mapping

COSTOTest exploits API features (1) to detect missing mappings between the TSM and the PDM, (2) to generate standard primitive fragments (e.g. 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.

Build such a test harness at the model level induces an additional effort (the models are not considered only to develop the service-based component's implementation), but the errors detected are less expensive to solve than those of components and services once deployed on specific platforms. Moreover, if the component model is implemented several times targeting several different PDM, the tests would be reused and part of the behaviour would have been checked before runtime.

Finally, *the test execution* consists in setting the test data and then "run" the test harness component. COSTOTest proposes interactive screen to enter all the data values into the XML file generated by the second model transformation. She/He can also provide the test data values in a CSV file which is transformed into the XML file.

We perform experiments to study the effectiveness of our tool in testing services into components assembly. We consider the test of computeSpeed service, covering its control flow graph to generate test data. We create 45 test cases and run them getting the verdicts. The data source XML file will also store the verdicts (*cf.* Figure 5).

4 Related Work

There are several research efforts interested in generating tests for testing components.

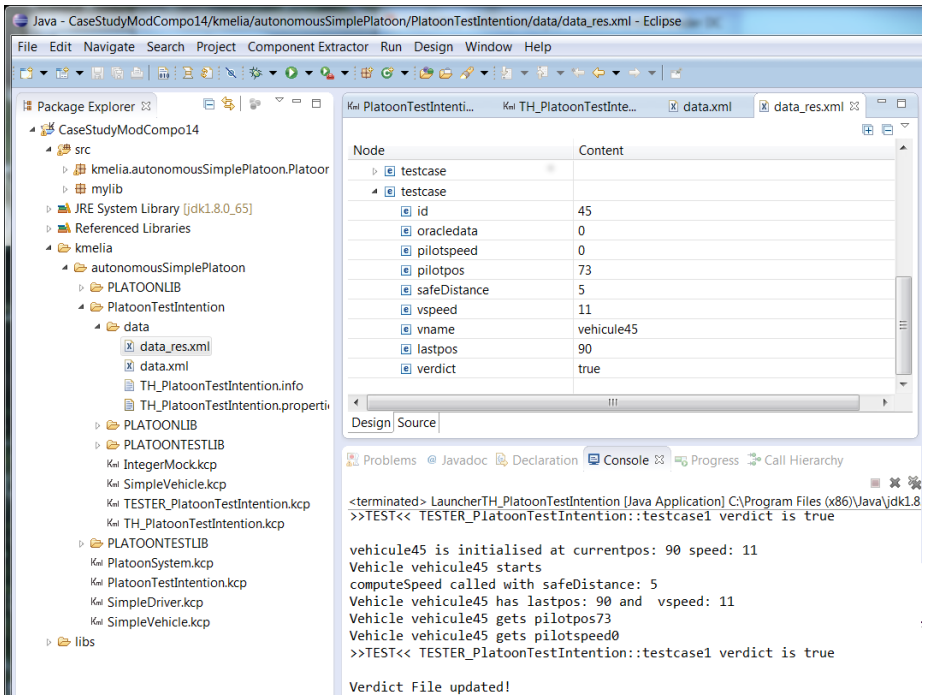

Figure 5: Test harness assignments: verdict stored in the XML file

In [11], Mariani et al. propose an approach for implementing self-testing components. They move testing of component from development to deployment time. In [12], 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 the components and services are tested at the modelling phase, before implementation.

In [13], Edwards outlines a strategy for automated black-box testing of software components. Components are considered in terms of object-oriented classes, whereas we consider components as entities providing and requiring services.

In [14], Zhang introduces test-driven modelling to apply the XP test-driven paradigm to an MDD process. Their approach designs test before modelling when we design test after modelling. In [15], 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 even if their are executed at the code level.

5 Conclusion

Testing SBC applications following MDD brings the advantages of early testing. We propose a demonstration of the COSTOTest tool which is a proof of concept of our previously published method for testing component models, from building the test harness to its execution. The test designer works on the test data and oracle, and the component interface, while the tool helps, checks, and builds the executable tests. This method applies when the modelling language includes detailed behaviour expression and is supported by a full IDE with code generation *e.g.* Sofa, rCOS, UML/AS...

The current tool, developed as an Eclipse plug-in, shows the feasibility of the approach and improvements are in prog-ress. First, we are developing mutation analysis facilities in COSTOTest. Second, we are experimenting the improvement of the approach compared with classical testing when models evolve.

References

[1] G. Shanks, E. Tansley, and R. Weber, "Using ontology to validate conceptual models," *Communications of the ACM*, vol. 46, no. 10, pp. 85–89, 2003.

[2] M. Gogolla, J. Bohling, and M. Richters, "Validating uml and ocl models in use by automatic snapshot generation," *Software & 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 *1st European Workshop on MDA-IA, in CTIT Technical Report Nr TR-CTIT-04-12*, Citeseer. Springer, 2004, pp. 97–104.

[4] P. André, J.-M. Mottu, and G. Ardourel, "Building test harness from service-based component models," in *proceedings of the Workshop MoDeVVa (Models2013)*, Miami, USA, Oct. 2013, pp. 11–20.

[5] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro, E. S. Spoto, and W. E. Wong, "Component-based software: An overview of testing," in *Component-Based Software Quality - Methods and Techniques*, vol. 2693. Springer, 2003, pp. 99–127.

[6] 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.

[7] S. H. Edwards, "A framework for practical, automated black-box testing of component-based software," *Software Testing, Verification and Reliability*, vol. 11, p. 2001, 2001.

[8] M. Messabihi, P. André, and C. Attiogbé, "Multilevel contracts for trusted components," in *WCSI*, ser. EPTCS, J. Cámara, C. Canal, and G. Salaün, Eds., vol. 37, 2010, pp. 71–85.

[9] 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.

[10] OSOA, "Service component architecture (sca): Sca assembly model v1.00 specifications," Open SOA Collaboration, Specification Version 1.0, March 2007.

[11] 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.

[12] 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.

[13] 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.

[14] Y. Zhang, "Test-driven modeling for model-driven development," *IEEE Software*, vol. 21, no. 5, pp. 80–86, 2004.

[15] 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.