Deriving UML-based Specifications of Inter-Component Interactions from Runtime Tests

Thorsten Haendler, Stefan Sobernig and Mark Strembeck
Institute for Information Systems and New Media
Vienna University of Economics and Business (WU Vienna), Austria
{firstname.lastname}@wu.ac.at

ABSTRACT

In this paper, we present a model-driven approach for the derivation of inter-component-interaction specifications from runtime tests. In particular, we use test-execution traces to record interactions between architectural components based on testing object-oriented systems. The resulting models are specified via UML diagrams. In order to transform test executions to corresponding component and interaction models, we define conceptual mappings (transformation rules) between a test-execution metamodel and the UML2 metamodel. As a proof of concept, we integrated the approach into our tool KaleidoScope.

CCS Concepts

•**Software and its engineering** → **Object oriented architectures; Unified Modeling Language (UML); Software testing and debugging;** *Model-driven software engineering; Dynamic analysis; Documentation;*

Keywords

Component Interaction; Test-Execution Viewpoint; Scenario-based Runtime Tests; UML; Component-based Architecture

1. INTRODUCTION

A component-based architecture structures a software system in terms of components and connections between them [15]. (Re-)Structuring a system into components provides multiple advantages regarding system maintainability and code reusability at the large, e.g., by abstracting from details of object-oriented code structures [13]. Graphical models support a software system's stakeholders in understanding and communicating a specific software architecture [5]. Today, UML models are a *de facto* standard for graphically documenting software structures and processes. In recent years, researchers have found disadvantages resulting from a purely manual creation and maintenance of software architectures. For instance, given the need for having an up-to-date documentation of (possibly) large component-based architectures, manual documentation maintenance becomes time consuming and error-prone. In response, approaches have been proposed for recovering a component-based architecture (and architecture documentation) semi-automatically from implementation artifacts of object-oriented systems (see, e.g., [1, 14]). Architectural components are connected by provided and required component interfaces (e.g., specified via interface contracts [11]) which define how a component can be used by other components. Component interfaces provide important information for multiple system stakeholders: e.g., for system integrators or architects (design by reuse) or for developers of components (design for reuse; [5]). A critical facet of component interfaces is the documentation of intended inter-component interactions (e.g., specified via synchronization contracts [3] such as `Service-Interfaces` in SoaML).

In this paper[1], we propose a technique for semi-automatically deriving UML-based specifications of interactions between architectural components from testing object-oriented systems. Deriving such component interactions involves clustering component interfaces and setting up filters for interactions between these interfaces. The resulting test-based interactions are expressed via respective UML diagrams. Our approach builds on conceptual mappings (transformation rules) between a test-execution metamodel, on the one hand, and the UML2 metamodel, on the other hand. As a proof of concept, we extended our tool KaleidoScope[2] to support the proposed approach.

Conceptual Overview. Fig. 1 illustrates the suggested procedure. A software engineer (in the roles of a software developer or a test developer respectively) implements the system under test (SUT) and specifies the test script (step ①). Our approach requires the clustering of SUT classes to architectural components (step ②). This task can be performed manually by software architects (e.g., by structuring/annotating the source code using name spaces or packages, or

[1]An extended paper version is available for download from our website [7].
[2]Available for download from our website [7].

SAC 2016,April 04 - 08, 2016, Pisa, Italy

ACM 978-1-4503-3739-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851981

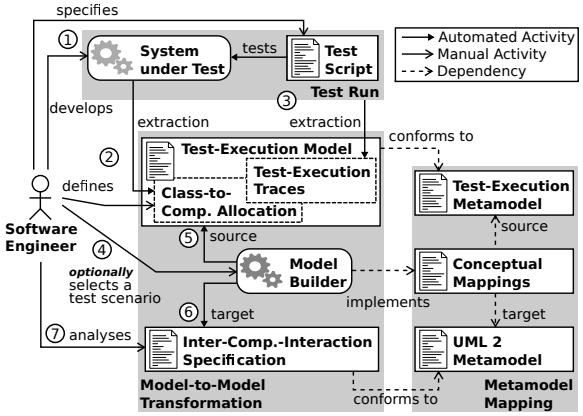

Figure 1: Conceptual overview of deriving test-based inter-component-interaction specifications.

by using a DSL) or automatically (e.g., based on extracted traits of cohesion and coupling of classes [10]). Next, based on instrumenting the test run (e.g., using dynamic analysis), a test-execution trace model is extracted automatically (step ③). Then, by default, the specifications of all relevant test scenarios are derived. Therefore, the test-execution model (including the class-to-component allocation and the test-execution traces; source model, step ⑤) is transformed automatically to inter-component-interaction specifications (target model, step ⑥). This transformation is executed by a model-builder engine which implements (e.g., in QVT operational) the conceptual mappings (transformation rules) between the test-execution metamodel and the UML2 metamodel. The concrete source and target models are instances of the corresponding metamodels. Optionally, a specific test scenario can be selected by the software engineer (step ④). In any case, the resulting UML model can be used for analysis of the system behavior by the software engineer (see step ⑦).[3]

2. A TEST-EXECUTION VIEWPOINT

To capture interactions between architectural components, we define and apply a *test-execution viewpoint* with the following three characteristics. *First*, the views document the *intended* behavior in terms of feature-call protocols (see Ⓒ in Fig. 2) of the system under test (SUT; see *process view* of the "4+1 view model" [9]); i.e. intended reactions of the SUT triggered by stimuli specified in the test script. *Second*, the views provide contextual information on the test script and the test environment (see *allocation viewtype* [5]). This context allows for bi-directional mappings between test (or test parts) and (architectural) elements of the SUT; similar to a *functional mapping* [2]:

- Links from a selected test part to covered SUT elements (Ⓐ in Fig. 2) and their behavior (see Ⓒ in Fig. 2; test-based slicing, e.g., for use-case-driven documentation [4])
- Links from a selected SUT element to covering test parts. For instance, by selecting a component, the owned ports indicate the covering test scenarios (see, e.g., `ComponentB` in Ⓐ in Fig. 2).

[3]The proposed procedure is fully supported by our tool KaleidoScope [7].

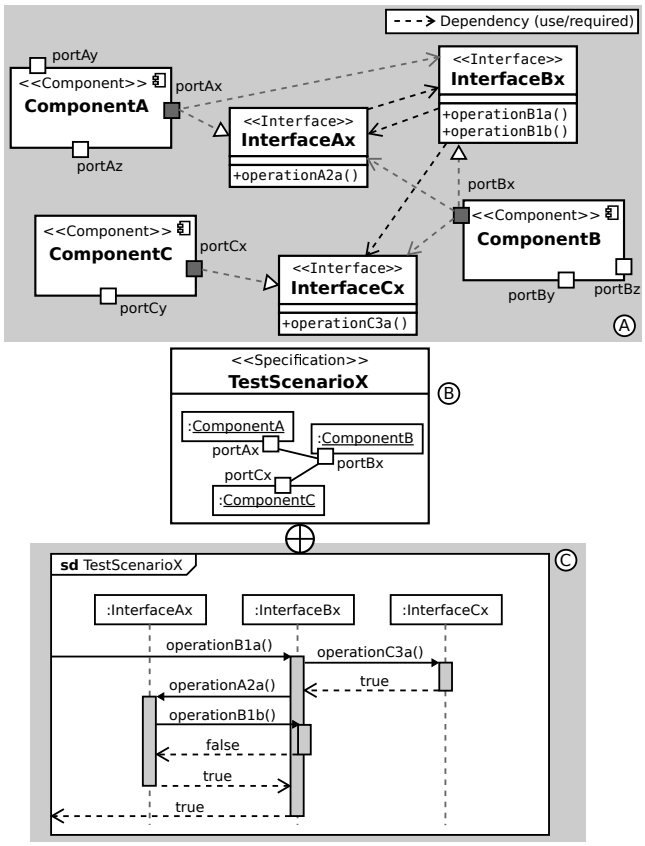

Figure 2: Example of a derived inter-component-interaction specification with SUT components and involved interfaces Ⓐ, the specification class Ⓑ, and owned interactions between interfaces Ⓒ.

Third, the views combine with those conforming to other viewpoints as additional slicing criteria, such as the component & connector viewpoint (e.g., represented by UML component models, Ⓐ in Fig. 2). This way, specifically tailored documentation can be obtained (model slicing).

Application Example. An exemplary object-oriented system consists of six classes allocated to three components (see Fig. 3). The owned class features are connected by multiple call dependencies (i.e. inter- vs. intra-component calls). A minimal test scenario for this SUT (`testScenarioX`) is depicted in Listing 1. Consider now, for instance, a software developer who intends to modify or to reuse `ComponentA`. She wants to identify the component's behavioral inter-dependencies to other components. Based on the derived specifications in Fig. 2, the participation of `ComponentA` in three test scenarios can be established; as indicated by the owned ports `portAx-z` (see Ⓐ). The developer can then decide to investigate one test scenario to review the corresponding inter-component interactions therein, such as called and calling features (e.g. `operationA2a` in fragment Ⓒ of Fig. 2).

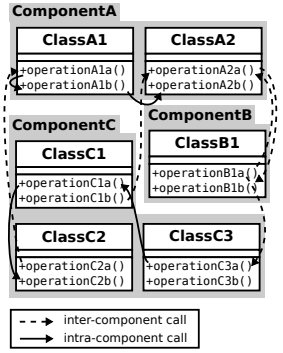

Figure 3: Classes of an exemplary SUT allocated to components.

Listing 1: Excerpt from an exemplary test script.

```
set sX [::STORM::TestScenario
     new -name testScenarioX
     -testcase cX]
$sX setup_script set {
  set a1 [::CompA::ClassA1 new]
  $a1 operationA1b
}
$sX test_body set {
  set b1 [::CompB::ClassB1 new]
  set c3 [::CompC::ClassC3 new]
  $b1 operationB1b
  $b1 operationB1a
}
$sX postconditions set {
  {expr {[[::CompC::ClassC3
       info instances]
       operationC3a] == true}}
}
```

3. DERIVING SPECIFICATIONS OF INTER-COMPONENT INTERACTIONS

Our approach applies scenario-based testing [12] to document the interplay between objects in the SUT. The SUT's runtime behavior (in terms of execution traces triggered by scenario tests) is constituted by the mutual interchange of messages between SUT objects. For specifying component interactions, we abstract from the concrete SUT elements (e.g., objects and class features) and their message exchanges by considering calls between components only. To transform test-execution trace models into UML interactions between components automatically, we define a set of conceptual mappings (transformation rules) between the test-execution metamodel, on the one hand, and the UML2 metamodel, on the other hand. These mappings are specified in transML [6] and refined by OCL mapping and consistency constraints. See the extended paper version for details [7].

4. CONCLUSION

This paper presents an approach for deriving UML-based specifications of interactions between architectural components from scenario-based runtime tests. The proposed viewpoint provides process documentation [9] in terms of sequences of (mutual) object-feature calls. The underlying execution traces are automatically extracted from test runs on object-oriented systems. Conceptual metamodel mappings between a test-execution metamodel and the UML metamodel render the approach generic.

We extended our tool KaleidoScope[4] [8] with support for the approach described above. In particular, KaleidoScope builds on the testing framework STORM [16] and model transformations (Eclipse M2M/QVTo). SUT classes can be allocated to components by defining packages in the system's source code. KaleidoScope then automatically derives the specifications by instrumenting the STORM test execution using message interceptors and callstack introspection.

Future work will seek to evaluate the approach in a larger project setting regarding its scalability and its benefits over manually creating and maintaining software-architecture documentation.

[4]Available for download from our website [7].

5. REFERENCES

[1] S. Allier, S. Sadou, H. Sahraoui, and R. Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *Proc. WICSA'11*, pages 214–223. IEEE, 2011.

[2] T. B. C. Arias, P. America, and P. Avgeriou. Defining execution viewpoints for a large and complex software-intensive system. In *Proc. WICSA/ECSA'09*, pages 1–10. IEEE, 2009.

[3] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[4] D. Bojic and D. Velasevic. A use-case driven method of architecture recovery for program understanding and reuse reengineering. In *Proc. CSMR'00*, pages 23–23. IEEE CS, 2000.

[5] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002.

[6] E. Guerra, J. Lara, D. S. Kolovos, R. F. Paige, and O. M. Santos. Engineering model transformations with transML. *Softw. Syst. Model.*, 12(3):555–577, 2013.

[7] T. Haendler. KaleidoScope. Institute for Information Systems and New Media. WU Vienna. http://nm.wu.ac.at/nm/haendler, 2015. Last accessed: 7 December 2015.

[8] T. Haendler, S. Sobernig, and M. Strembeck. An approach for the semi-automated derivation of UML interaction models from scenario-based runtime tests. In *Proc. ICSOFT-EA'15*, pages 229–240. SciTePress, 2015.

[9] P. B. Kruchten. The 4+ 1 view model of architecture. *Software, IEEE*, 12(6):42–50, 1995.

[10] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC'98*, pages 45–52, 1998.

[11] B. Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992.

[12] C. Nebut, F. Fleurey, Y. Le Traon, and J. Jezequel. Automatic test generation: A use case driven approach. *IEEE Trans. Softw. Eng.*, 32(3):140–155, 2006.

[13] T. Ravichandran and M. A. Rothenberger. Software reuse strategies and component markets. *Commun. ACM*, 46(8):109–114, 2003.

[14] A. Seriai, S. Sadou, H. Sahraoui, and S. Hamza. Deriving component interfaces after a restructuring of a legacy system. In *Proc. WICSA'14*, pages 31–40. IEEE, 2014.

[15] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[16] M. Strembeck. Testing policy-based systems with scenarios. In *Proc. IASTED'11*, pages 64–71. ACTA Press, 2011.