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 KaleidoScope tool.

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 [34]. (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 [29]. Graphical models support a software system’s stakeholders in understanding and communicating a specific software architecture [9]. Today, UML models [27] 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 [31]. 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) semiautomatically from implementation artifacts of object-oriented systems (see, e.g., [1] [33]).

Architectural components are connected by provided and required component interfaces (e.g., specified via interface contracts [21]) 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 for reuse) or for developers of components (design for reuse; [9]). A critical facet of component interfaces is the documentation of intended component interactions (e.g., specified via synchronization contracts [4]). Therefore, in the field of modeling service-oriented architectures, such interaction-aware component interfaces can be captured as collaborations between components by using corresponding interaction models (e.g., ServiceInterfaces in SoaML [26]). There are existing approaches for recovering a component-based architecture (see, e.g., [1] [33]) and for reverse-engineering behavioral models (see, e.g., [19]) from object-oriented systems. However, they fall short in a) integrating specifications of component interactions and a corresponding component model as well as in b) deriving component interactions from calls between object features (e.g. properties and methods).

In this paper, we propose a derivation technique to close this gap. In our previous work [16], we presented an approach for deriving interaction models from testing object-oriented systems using scenario-based runtime tests. The resulting UML models reflect inter-object interactions. In this present approach, specifications of interactions between interfaces of architectural components are derived semi-automatically from runtime tests. Deriving such inter-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 [27]. 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 KaleidoScope tool to support the proposed approach.

The remainder of the paper is structured as follows. Section 1 gives a high-level overview of the suggested procedure. In Sect. 2, we explain the characteristics of the proposed test-execution viewpoint and illustrate a short application example. Section 3 focuses on the derivation of inter-component-interaction specifications. In particular, we provide the corresponding metamodels and the conceptual

1Manually performed by the software engineer are the tasks of allocating classes to components and, optionally, selecting a specific test scenario (see below).
2Available for download from our website [15].
metamodel mappings. In Sect. 4 we introduce our prototypical implementation. Section 5 gives an overview of related approaches and Sect. 6 concludes the paper.

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

1.1 Conceptual Overview

Fig. 1 illustrates the suggested procedure for deriving test-based specifications of inter-component interactions. 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 by using a DSL or automatically (e.g., based on extracted traits of cohesion and coupling of classes [20]). 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 transformation) 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 scenario can be selected by the software engineer (step ④). Finally, however, the resulting UML model can be used for analysis of the system behavior (see step ⑦).

2. A TEST-EXECUTION VIEWPOINT

An architectural view [9] represents an abstracted perspective on a software system’s elements, on relations between them, and, optionally, on the system context. Each view conforms to a viewpoint and reflects concerns of stakeholder roles. The system’s runtime behavior is one important facet for architecture documentation and is therefore included in multiple viewpoint models (see, e.g., [6, 9]).

The proposed procedure is fully supported by our KaleidoScope tool [15].

2.1 Characteristics

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 ③ in Fig. 2) of the system under test (SUT; see process view of the “4+1 view model” [18]); 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 ⑨). This context allows for bi-directional mappings between test (or test parts) and (architectural) elements of the SUT; similar to a functional mapping ⑧:

1. Links from a selected test part to covered SUT elements ( ⑧ in Fig. 2) and their behavior (see ③ in Fig. 2) test-based slicing, e.g., for use-case-driven documentation ④.

2. 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 ⑧ in Fig. 2).

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, ⑧ in Fig. 2). This way, specifically tailored documentation can be obtained (model slicing).
Test-based Component Interfaces. Component interfaces represent a form of contract which defines how to use a component (see, e.g., [21]). The proposed specifications put emphasis on component interfaces and their attached interaction as *synchronization contracts* [4]. A synchronization contract specifies the correct order (a.k.a. protocol) of mutual feature calls. The sequence of calls triggered by the stimuli of a test scenario represents such an intended order.

Partial Interaction Models. Runtime tests such as scenario-based tests (see below) provide a specific structure of nested/ordered test parts. This way, views can be derived from different hierarchical parts, e.g., from a specific test case or test block (see Fig. 4). Hereafter, we use test scenarios, since they conceptually correspond to usage scenarios that describe an intended interplay between components. This way, each derived partial model reflects interactions filtered and abstracted in two dimensions: horizontal (including only components and interfaces involved in a specific test scenario) as well as vertical (including only interactions between these interfaces triggered by the test scenario).

<table>
<thead>
<tr>
<th>FeatureCallDefinition</th>
<th>TestCase</th>
<th>TestScenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExpectedResult</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precondition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestResult</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Postcondition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cleanup</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 1: Excerpt from an exemplary test script.

```plaintext
set sx [:STORM::TestScenario
  -name scenarioX
  -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
      operationC3a] == true))
```

2.2 Application Example

An exemplary object-oriented system consists of six classes allocated to three components (see Fig. 3). The owned object 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 A). 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 C of Fig. 2).

3. DERIVING SPECIFICATIONS OF INTER-COMPONENT INTERACTIONS

3.1 Capturing Inter-Component Interactions

Test-Execution Metamodel. Our approach applies scenario-based testing [22, 23] to document the interplay between objects in the SUT. The metamodel of scenario-based testing, including the test-framework, the internal test-block structure and the test-execution traces, defined in [35, 16], is extended to capture the allocation of classes to architectural components (see Fig. 4).

Figure 3: Classes of an exemplary SUT allocated to components

Figure 4: Test-execution metamodel including scenario-test framework (STF; [35]), internal test-block structure, scenario-test traces [16], and class-to-component allocation.

Inter-Component Interactions. 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 that are calling object features (e.g., operations, property setters/getters). For specifying component interactions, we abstract from the concrete SUT elements (e.g., objects and object features) and their message exchanges by considering calls (message exchanges) between components only (see, e.g., the example in Fig. 3). Thus, our approach requires the application of integration tests which include calls between interfaces (sets of object features) of architectural components. In our metamodel representation (see Fig. 4), this criterion is expressed by requiring an ow—
Assumptions. In case that a class is not allocated to a component, this class is treated as a component. In case that a scenario does not include any inter-component call, a specification for this scenario can not be derived.

3.2 Applied Elements of UML2

For specifying inter-component interactions in the Unified Modeling Language (UML) [27], we apply a set of component- and interaction-specific elements of the UML2 metamodel (see Fig. 5). The example in Fig. 2 illustrates the notation of the derived specifications. A specification class (representing a relevant test scenario; see 5 in Fig. 2) contains instances of UML components (as ownedAttributes) participating in the corresponding test scenario. Moreover, it contains the corresponding interaction (as ownedBehavior). This way, the specification class connects parts A and C.

3.3 Conceptual Metamodel Mappings

To transform test-execution traces (and the corresponding class-to-component allocations) into UML inter-component-interaction specifications automatically, we define mappings between the test-execution metamodel (see Sect. 2 and Fig. 4) on the one hand, and the UML2 metamodel on the other hand (see Fig. 5). For this purpose, we define a set of conceptual mappings formalized by using a transML diagram [14], which represents transformation rules in a tool and technology independent manner compatible with the UML. The mappings are refined by OCL mapping constraints. In Fig. 6, we report on the 9 most important mappings (M1–9) [6].

Figure 6: The transML-based specification [14] of conceptual mappings between the test-execution metamodel and the UML2 metamodel.

Every Trace instance (reflecting one test run) is mapped to a Package instance which i.a. contains the partial interaction specifications (see M9 in Fig. 6). Each TestScenario instance that contains interactions between two or more components is mapped to a Model instance, to a Class instance (applying the stereotype Specification) and to an Interaction instance (see M8). For details of the mapping constraint isInterComponentInteractionTest, see Listing 2. The UML stereotype Specification indicates that the applying class specifies a domain of objects without defining the physical implementation of these objects. The resulting interaction represents the ownedBehavior of the

7For a complete set of applied mappings and for further details, see our prototype implementation [15].
specification. Both are owned by the resulting UML model. In UML, a model captures a view of a system. In our approach, each Model instance represents a partial scenario-based inter-component interaction specification.

**Deriving Component Elements.** At this point, we describe the metamodel mappings of the component-specific elements (as depicted by example in [A] in Fig. 2). A UML Component is a modular part of a system. Each Component instance is mapped to an instance of UML Component (see M1). In UML, a Port indicates an interaction point between a component and its environment. Each time a component is involved in a test scenario (i.e., public features are called or are calling), a Port instance is created (see M2) that is owned by the corresponding component.

A UML Interface represents a declaration of a set of coherent public component features. It specifies a syntactical contract for the realizing classifier. In our context, it owns the public features (e.g., operations, property setters/getters) of a specific component that are called during a test scenario by features of other components. This way, for each set of public features of a specific component that are used in the scope of the corresponding test scenario, an Interface instance (owning these features) is created (see M2–M4). In order to express the dependencies between interfaces, Usage relationships are applied. In UML, a Usage is a specialization of Dependency which indicates that a client element requires a supplier element. In our approach, the client is represented by the interface which contains the calling features and the supplier by the interface that owns the corresponding called features (see M5).

Each port (see above) is typed by the interface that owns the offered public features of the corresponding owning component. A port references required and provided interfaces, which are derived according to the value of isConjugated (by default false) from the type of the port (see [27]). This way, the required interfaces are derived from the set of interfaces that are used by the type of the port (see, e.g., portBx in Fig. 2). In turn, the provided interface is derived directly from the type of the port (since the type is an interface, see [27]).

**Deriving Interaction Elements.** In addition, we describe the transformation rules for interaction-specific metamodel elements related to the ownedBehavior of the test specification (for example, see [C] in Fig. 2). Each FeatureCall instance that represents a call between two components (isInterComponentCall, see Listing 2) is mapped to a Message instance owned by the corresponding interaction (see M3). Based on this feature call (Fc), other elements related to it (e.g., callee, arguments) are mapped. In particular, the return value is mapped to another instance of Message with messageSort reply (see M7). All calling or called features are mapped to instances of MessageOccurrence. Moreover, each Interface instance serves as a lifeline in the owned interaction (see M2).

### 4. PROTOTYPE IMPLEMENTATION

We extended our tool KaleidoScope with support for the approach described above. In particular, KaleidoScope builds on the testing framework STORM and model transformations (Eclipse M2M/QVTo). It can derive inter-component-interaction specifications from scenario-based runtime tests in a semi-automated manner. SUT classes can be allocated to components by defining packages in the system’s source code. KaleidoScope then automatically derives a corresponding interaction specification. By default, a UML model with specifications for all test scenarios is created that reflect the inter-component interactions resulting from the respective test run. Optionally, the software engineer can select a specific test scenario.

**STORM.** The “Scenario-based Testing of Object-oriented Runtime Models” (STORM) test framework provides an infrastructure for specifying and executing scenario-based tests. STORM provides all elements of our testing metamodel (see Fig. 1). KaleidoScope builds on and instruments STORM. It is implemented using the dynamic object-oriented language “Next Scripting Language” (NX), an extension of the “Tool Command Language” (Tcl).

KaleidoScope comprises a trace provider and a model builder component.

**Trace Provider.** The trace-provider component records the test-execution traces by intercepting all relevant method calls triggered by the STORM engine during test execution. For this purpose, NX/Tcl offers built-in method-call introspection in terms of message interceptors (see [30]) and call-stack introspection (see [23]). The class-to-component allocation (defined by namespaces/packages in the SUT’s source code) is also extracted using introspection techniques. The execution traces are subsequently stored as trace models in their XMI representation (XML Metadata Interchange specification), conforming to the Ecore trace metamodel extended by the class Component.

**Model Builder.** For transforming our trace models into UML models automatically, the respective model transformations are implemented via “Query/View/Transformation Operational” (QVTo) mappings. QVTo allows for implementing model transformations based on the conceptual mappings presented in Sect. 3 in a straightforward manner. In total, 21 mapping actions are executed. The resulting UML specifications are again persisted in their Ecore/XMI representation, which allows for import by XMI-compliant diagram editors (e.g., Eclipse Papyrus).

### 5. RELATED WORK

Our work is motivated by the observation that execution views significantly help practitioners to describe, to analyze, and to exchange information about a given software architecture. Arias et al. [3] discuss how runtime information can be reflected by execution viewpoints. They propose a conceptual model for defining and categorizing execution views and viewpoints. In particular, they distinguish functional mapping, deployment, concurrency, and resource usage viewpoints. In this context, our approach extends the viewpoint set to include a dedicated test-execution viewpoint.

Research closely related to our approach falls into three groups: (1) reverse-engineering interaction models from system executions, (2) specifying component interfaces and inter-component interactions and (3) recovering a software
architecture (or architecture documentation) from system implementations.

**Reverse-engineering Interaction Models.** Multiple approaches for (semi-)automatically reverse-engineering behavioral models from system-execution traces exist, in particular system behavior in terms of state machines, e.g., [6], [19]. Note that our approach does not cover this kind of behavior. Our approach compares with contributions which derive interaction models (e.g., UML sequence diagrams) reflecting execution paths (see, e.g., [18]) in object-oriented systems, for instance, mutual message exchanges. [28], [13], [11]. Among these, model-driven approaches, e.g., which provide a trace metamodel to represent the execution traces, are the most closely related ones [7], [10]. However, this group of related work does not consider interactions (in terms of feature-call sequences) between provided and required interfaces of architectural components.

**Specifying Inter-Component Interactions.** The second group of related work focuses on the semantic specification of component interfaces, especially interface interactions in a concrete documentation language such as the UML. Jonkers [17] introduces the ISpec approach for specifying interfaces compatible with the UML. A specification is defined as a multi-party contract between providers and users of services. The involved interfaces (called interface suite) are mutually dependent on each other. Service contracts of the Service-oriented architecture Modeling Language (SoaML) [25] provide a specification similar to the one applied in our approach. Service contracts include interactions (called service choreography; also a kind of UML behavior) between service providers and consumers (both typed by component interfaces). These contracts allow for specifying the mutual message exchange between the interfaces. In a SOA, services represent independent high-level software services (e.g., web services) provided for other services. In the context of testing object-oriented systems, consumer and provider might not be as clearly distinguishable as in a SOA context, since every software component can take on both roles in a given specification context (due to mutual dependencies, esp. callbacks). The UML-based specification proposed in our approach allows for reflecting components in arbitrary roles.

**Recovering Architecture Documentation.** Of particular interest are approaches that focus on interface identification and/or perform slicing based on call graphs built from execution traces. Allier et al. [1] propose a technique for assisting the transformation of object-oriented to component-based applications. Based on execution traces obtained from executing (self-defined) use-case scenarios, a set of abstraction and filter techniques are applied for component and interface identification. Identifying components and, especially, interface specification is based on execution traces, a static call graph, and statical class relationships. Seriai et al. [33] propose a process for identifying coherent component interfaces in object-oriented applications by using formal-concept analysis (FCA). Based on all possible interactions/dependencies between components (reflected by a conceptual lattice), features are clustered to component interfaces which in turn are aligned with the high-level component features. In contrast, we derive interfaces from test-driven system execution. In addition to Allier et al. and Seriai et al., who reflect component interfaces on a syntactic level (provided and required features), we derive specifications of ordered interactions (protocols) between these interfaces. Bojic and Velasevic [6] propose an approach for partially recovering elements of architectural views using FCA. Given an ad hoc selection of use-case scenarios that are refined into executable tests, they suggest slicing partial dynamic call graphs obtained from executing of the previously defined software tests. Based on collecting profiling information of test executions, they derive a conceptual lattice in form of an acyclic graph, which can serve as basis for deriving UML models. Our approach builds on a similar idea, however, we put a special emphasis on identifying interactions between component interfaces and on specifying them using UML.

6. CONCLUSION

In this paper, we presented an approach for deriving UML-based specifications of interactions between architectural components from scenario-based runtime tests. Therefore, we extract automatically execution traces from test runs on object-oriented systems. Our approach requires that the systems under test are organized into architectural components. Our derivation technique is rendered generic by offering conceptual metamodel mappings between a test-execution metamodel and the UML metamodel. As a proof of concept, we integrated the approach into our KaleidoScope tool.

**Limitations.** The proposed viewpoint provides process documentation [18] in terms of sequences of (mutual) object-feature calls. It reflects the concrete test-execution path filtered, abstracted and specified by the UML. Note that it is not meant to include complex behavior expressed by states and state transitions, e.g., as specified by finite state machines. Furthermore, the proposed approach requires an evaluation in a larger project setting. This is because both the approach’s scalability and its alleged benefits over manually creating and maintaining software-architecture documentation must be evaluated empirically.

**Future Work.** As a next step, we will investigate via controlled experiments to which extent the derived interaction specifications assist in architecture-related tasks such as the component refactoring by facilitating system comprehension. Moreover, we plan to extend the approach to incorporate behavior-related information from scenario-based runtime tests such as behavioral contracts (e.g., inferred pre- and post-conditions) and measured execution times (inferred time constraints for interactions). Furthermore, we will review techniques of comparing or merging the resulting specifications (e.g., with manually created architecture documentation). Finally, we seek to explore how the specifications can be applied for reviewing software tests (e.g., measuring code coverage and requirements conformance) and the SUT (e.g., regarding cohesion/coupling).

7. REFERENCES


