Deriving Tailored UML Interaction Models from Scenario-Based 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
{thorsten.haendler,stefan.sobernig,mark.strembeck}@wu.ac.at

Abstract. Documenting system behavior explicitly using graphical models (e.g. UML activity or sequence diagrams) facilitates communication about and understanding of software systems during development and maintenance tasks. Creating graphical models manually is a time-consuming and often error-prone task. Deriving models from system-execution traces, however, suffers from resulting model sizes which render the models unmanageable for humans. This paper describes an approach for deriving behavior documentation from runtime tests in terms of UML interaction models. Key to our approach is leveraging the structure of scenario-based runtime tests to render the resulting interaction models and diagrams tailorable by humans for a given task. Each derived model represents a particular view on the test-execution trace. This way, one can benefit from tailored graphical models while controlling the model size. The approach builds on conceptual mappings (transformation rules) between a test-execution trace metamodel and the UML2 metamodel. In addition, we provide means to turn selected details of test specifications and of testing environment (i.e. test parts and call scopes) into views on the test-execution trace (scenario-test viewpoint). A prototype implementation called KaleidoScope based on a software-testing framework (STORM) and model transformations (Eclipse M2M/QVTo) is available.

Keywords: Test-based documentation · Scenario-based testing · Test-execution trace · Scenario-test viewpoint · UML interactions · UML sequence diagram

1 Introduction

Scenarios describe intended or actual behavior of software systems in terms of action and event sequences. Notations for defining and describing scenarios include different types of graphical models such as UML activity and UML interaction models. Scenarios are used to model systems from a user perspective and ease the communication between different stakeholders [3,17,18]. As it is almost impossible to completely test a complex software system, one needs an effective procedure to select relevant tests, to express and to maintain them, as well as to automate tests whenever possible. In this context, scenario-based testing is a

© Springer International Publishing Switzerland 2016
P. Lorenz et al. (Eds.): ICSOFT 2015, CCIS 586, pp. 326–348, 2016.
DOI: 10.1007/978-3-319-30142-6_18

means to reduce the risk of omitting or forgetting relevant test cases, as well as the risk of insufficiently describing important tests [21,32].

Tests and a system's source code (including the comments in the source code) directly serve as a documentation for the respective software system. For example, in Agile development approaches, tests are sometimes referred to as a *living documentation* [35]. However, learning about a system only via tests and source code is complex and time consuming. In this context, graphical models are a popular device to document a system and to communicate its architecture, design, and implementation to other stakeholders, especially those who did not author the code or the tests. Moreover, graphical models also help in understanding and maintaining a system, e.g., if the original developers are no longer available or if a new member of the development team is introduced to the system. Alas, authoring and maintaining graphical models require a substantial investment of time and effort. Because tests and source code are primary development artifacts of many software systems, the automated derivation of graphical models from a system's tests and source code can contribute to limiting documentation effort. Moreover, automating model derivation provides for an up-to-date documentation of a software system, whenever requested.

A general challenge for deriving (a.k.a. reverse-engineering) graphical models is that their visualization as diagrams easily becomes too detailed and too extensive, rendering them ineffective communication vehicles. This has been referred to as the problem of *model-size explosion* [1,33]. Common strategies to cope with unmanageable model sizes are filtering techniques, such as element sampling and hiding. Another challenge is that a graphical documentation (i.e. models, diagrams) must be captured and visualized in a manner which makes the resulting models *tailorable* by the respective stakeholders. This way, stakeholders can fit the derived models to a certain analysis purpose, e.g., a specific development or maintenance activity [9].

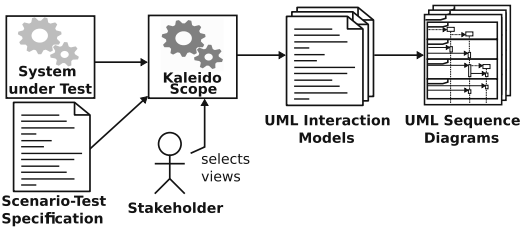

Fig. 1. Deriving tailored UML interaction models from scenario tests.

In this paper, we report on an approach for deriving behavior documentation (esp. UML2 interaction models depicted via sequence diagrams) from scenario-based runtime tests in a semi-automated manner (see Fig. 1). Our approach is independent of a particular programming language. It employs metamodel mappings between the concepts found in scenario-based testing, on the one hand, and the UML2 metamodel fragment specific to UML2 interactions [27], on the other hand. Our approach defines a *viewpoint* [4] which allows for creating different views on the test-execution traces resulting in partial interaction models

and sequence diagrams. Moreover, we present a prototypical realization of the approach via a tool called KaleidoScope[1]. This paper is a revised and extended version of our paper from ICSOFT 2015 [15]. This post-conference revision incorporates important changes in response to comments by reviewers and by conference attendees. For example, we included the OCL consistency constraints for the derived UML interaction models.

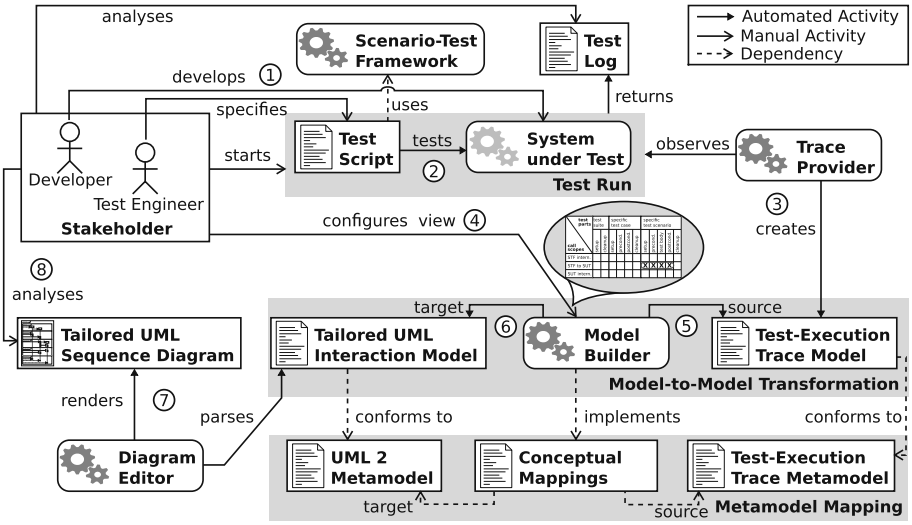

Fig. 2. Conceptual overview of deriving tailored UML interaction models from scenario-based runtime tests.

Figure 2 provides a bird's-eye view on the procedure of deriving tailored interaction models from scenario-based runtime tests. After implementing the source code of the SUT and specifying the scenario-test script, the respective tests are executed (see steps ① and ② in Fig. 2). A "trace provider" component instruments the test run (e.g. using dynamic analysis) and extracts the execution-trace data[2] for creating a corresponding scenario-test trace model (see step ③). After test completion, the test log is returned (including the test result). Based on a view configuration and on the extracted trace model (see steps ④ and ⑤), an interaction model (step ⑥) is derived. This transformation is executed by a "model builder" component, which implements the conceptual mappings between the test-execution trace metamodel and the UML2 metamodel. The concrete source and target models are instances of the corresponding metamodels. Notice that based on one trace model (reflecting one test run), multiple tailored interaction models can be derived in steps ④ through ⑥.[3] Finally,

[1] Available for download from our website [14].

[2] For the purposes of this paper, a *trace* is defined as a sequence of interactions between the structural elements of the system under test (SUT), see e.g. [38].

[3] The process described so far is supported by our KaleidoScope tool [14].

the models can be rendered by a diagram editor into a corresponding sequence diagram (step ⑦) to assist in analysis tasks by the stakeholders (step ⑧).

The remainder of this paper is structured as follows: In Sect. 2, we explain how elements of scenario tests can be represented as elements of UML2 interactions. In particular, we introduce in Sect. 2.1 our metamodel of scenario-based testing and in Sect. 2.2 the elements of the UML2 metamodel that are relevant for our approach. In Sect. 2.3, we present the conceptual mappings (transformation rules) between different elements of the scenario-test metamodel and the UML2 metamodel. Subsequently, Sect. 3 proposes test-based tailoring techniques for the derived interaction models. In Sect. 3.1, we explain the option space for tailoring interaction models based on a scenario-test viewpoint and illustrate a simple application example in Sect. 3.2. Section 3.3 explains how tailoring interaction models is realized by additional view-specific metamodel mappings. In Sect. 4, we introduce our prototypical implementation of the approach. Finally, Sect. 5 gives an overview of related work and Sect. 6 concludes the paper.

2 Representing Scenario Tests as UML2 Interactions

2.1 Scenario-Test Structure and Traces

We extended an existing conceptual metamodel of scenario-based testing [34]. This extension allows us to capture the structural elements internal to scenario tests, namely test blocks, expressions, assertions, and definitions of feature calls into the system under test (SUT; see Fig. 3). A trace describes the SUT's responses to specific stimuli [4]. We look at stimuli which are defined by an executable scenario-test specification and which are enacted by executing the corresponding scenario test. In the following, we refer to the combined structural elements of the scenario-test specifications and the underlying test-execution infrastructure as the scenario-test framework (STF). This way, an execution of a scenario-based `TestSuite` (i.e. one test run) is represented by a `Trace` instance. In particular, the respective trace records instances of `FeatureCall` in chronological order, describing the SUT feature calls defined by the corresponding instances of `FeatureCallDefinition` that are owned by a block. Valid kinds of `Block` are `Assertion` (owned, in turn, by `Pre-` or `Postcondition` block) or other STF features such as `Setup`, `TestBody` or `Cleanup` in a certain scenario test. In turn, each SUT `Feature` represents a kind of `Block`. A block aggregates definitions of SUT feature calls. Instances of `FeatureCall` represent one interaction between two structural elements of the SUT. These `source` and `target` elements are represented by instantiations of `Instance`. Every feature call maintains a reference to the calling feature (`caller`) and the corresponding called feature (`callee`), defined and owned by a given class of the SUT. `Features` are divided into structural features (e.g. `Property`) and behavioral features (e.g. `Operation`). Moreover, `Constructor` and `Destructor` owned by a class are also kinds of `Feature`. A feature call additionally records `Argument` instances that are passed into the

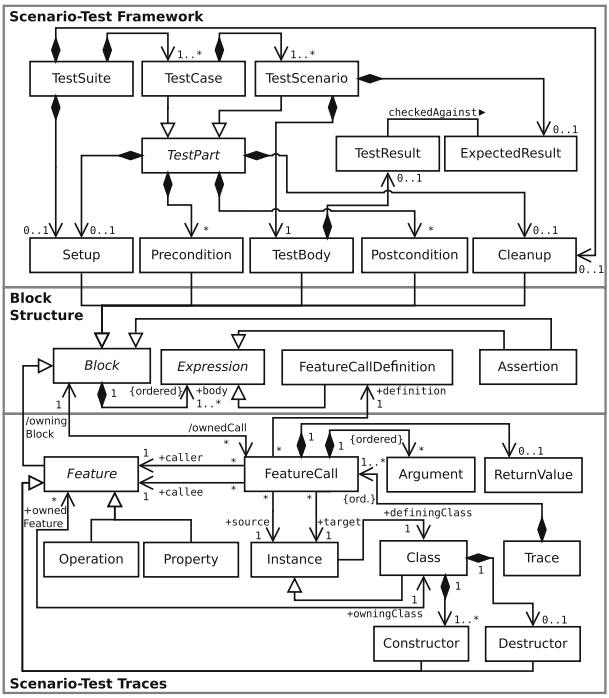

Fig. 3. Test-execution trace metamodel extends [34] to include test-block structure and scenario-test traces.

called feature, as well as the return value, if any. The sum of elements specific to a call is referred to as "call dependencies".

2.2 Interaction-Specific Elements of UML2

UML interaction models and especially sequence diagrams offer a notation for documenting scenario-test traces. A UML `Interaction` represents a unit of behavior (here the aforementioned trace) with focus on message interchanges between connectable elements (here SUT instances). In this paper, we focus on a subset of interaction-specific elements of the UML2 metamodel that specify certain elements of UML2 sequence diagrams (see Fig. 4). The participants in a UML interaction model are instances of UML classes which are related to a given scenario test. The sequence diagram then shows the interactions between these instances in terms of executing and receiving calls on behavioral features (e.g. operations) and structural features (e.g. properties) defined for these instances via their corresponding UML classes. From this perspective, the instances interacting in the scenario tests constitute the SUT. Instances which represent structural elements of the scenario-testing framework (STF; e.g. test cases, postconditions), may also be depicted in a sequence diagram; for example as

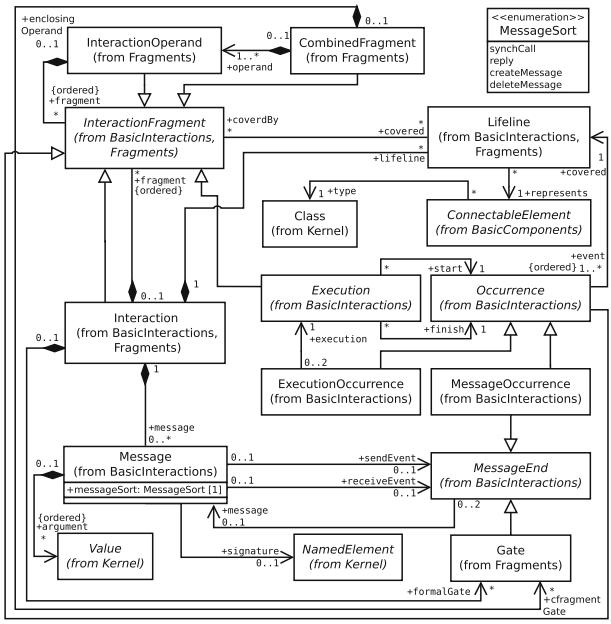

Fig. 4. Selected interaction-specific elements of the UML2 metamodel.

a test-driver lifeline [5]. The feature calls on SUT instances originating from STF instances rather than other SUT instances represent the aforementioned stimuli. This way, such feature calls designate the beginning and the end of a scenario-test trace.

2.3 Mapping Test Traces to Interactions

To formalize rules for transforming scenario-test traces into UML interactions, we define a metamodel mapping between the scenario-test trace metamodel, on the one hand, and the corresponding excerpt from the UML2 metamodel, on the other hand. For the purposes of this paper, we specified the corresponding mappings using transML diagrams [13], which represent model transformations in a tool- and technology- independent manner compatible with the UML. In total, 18 transML mapping actions are used to express the correspondences. These mapping actions (M1–M18) are visualized in Figs. 5, 6 and 12. The transML mapping diagrams are refined by OCL expressions [24] to capture important mapping and consistency constraints for the resulting UML interaction models. The mapping constraints are depicted below each related transML mapping. The mapping action represents the context for the OCL constraints and, this way, allows for navigating to elements of the source and target model. The OCL consistency constraints are fully reported in the Appendix of this paper.

In general, i.e. independent of a configured view, each `Trace` instance, which comprises one or several feature calls, is mapped to an instance of UML

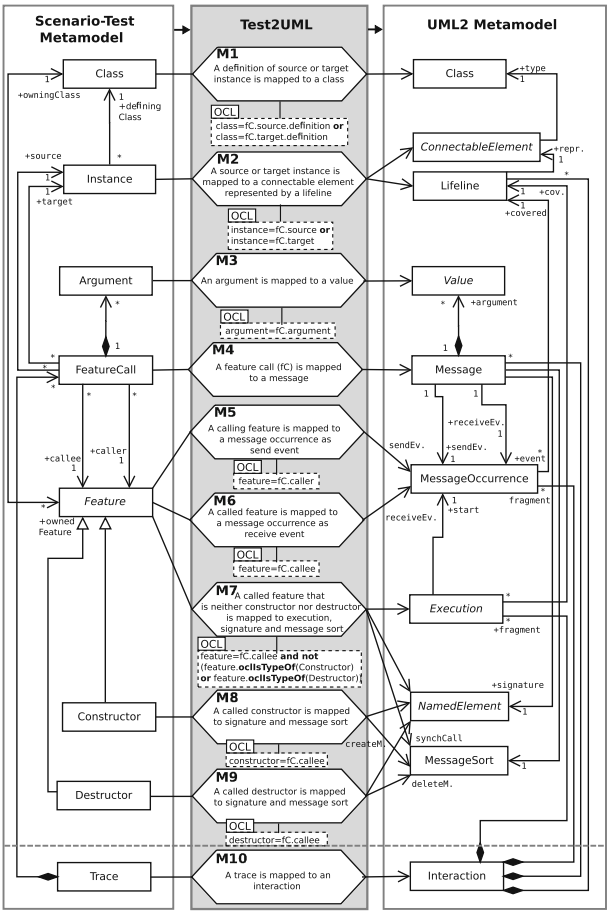

Fig. 5. Mapping elements of scenario-test traces (specific to a feature call `fC`) to UML2 elements.

`Interaction` (see `M10` in Fig. 5). This way, the resulting interaction model reflects the entire test-execution trace (for viewpoint mappings, see Sect. 3.3). However, each instance of `FeatureCall` (`fC`) contained by a given trace is mapped to one UML `Message` instance (see `M4`). Each of the mappings of the other trace elements (i.e. "call dependencies") depends on mapping `M4` and is specific to `fC`. Each instance that serves as `source` or `target` of a feature call is captured in terms of a pair of a `ConnectableElement` instance and a `Lifeline` instance. A `Lifeline`, therefore, represents a participant in the traced interaction, i.e., a `ConnectableElement` typed with the UML class of the participant. See the transML mapping actions `M1` and `M2` in Fig. 5. An instance of `MessageOccurrence` in the resulting interaction model represents the feature call at the calling feature's end as a `sendEvent` (see `M5`). Likewise,

at the called feature's end, the feature call maps to a `receiveEvent` (see `M6`). Depending on the kind of the feature call, the resulting `Message` instance is annotated differently. For constructor and destructor calls, the related message has a create or delete `signature`, respectively. In addition, the corresponding message is marked using `messageSort` `createMessage` or `deleteMessage`, respectively (see `M8` and `M9`). Note that in case of a constructor call, the `target` is represented by the class of the created instance and the created instance is the return value. This way, in this specific case, the return value is mapped to lifeline and connectable element typed by the `target` (see `M8`). Other calls map to synchronous messages (i.e. `messageSort` `synchCall`). In this case, the name of the `callee` feature and the names of the arguments passed into the call are mapped to the `signature` of the corresponding `Message` instance (see `M7`). In addition, an execution is created in the interaction model. An `Execution` represents the enactment of a unit of behavior within the lifeline (here the execution of a called feature). The resulting `Execution` instance belongs to the lifeline of the `target` instance and its `start` is marked by the message occurrence created by applying `M6`. For the corresponding OCL consistency constraints based on mapping `M4`, see Listing 1.3 in the Appendix.

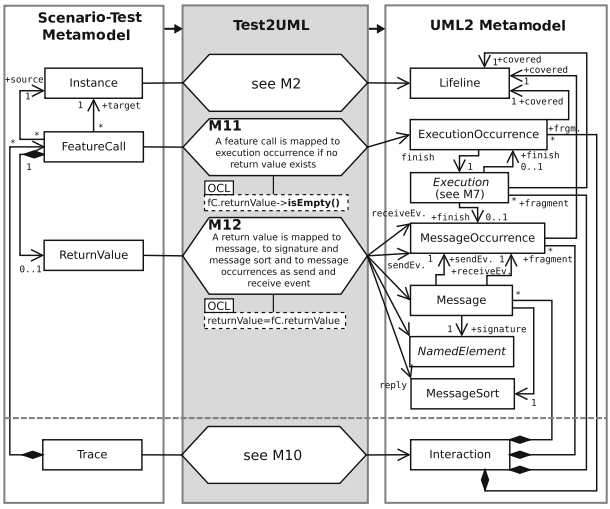

Fig. 6. Mapping return value (specific to a feature call `fC`) to UML2 elements.

If a given feature call `fC` reports a return value, a second `Message` instance will be created to represent this return value. This second message is marked as having `messageSort` `reply` (see `M12` in Fig. 6). Moreover, two instances of `MessageOccurrence` are created acting as the `sendEvent` and the `receiveEvent` (covering the lifelines mapped from `target` and `source` instance related to `fC`, respectively). An instance of `NamedElement` acts as the `signature` of this message, reflecting the actual return value (see `M12`). In case of a missing return value, an `ExecutionOccurrence` instance is

provided to consume the call execution (`finish`) at the called feature's end (see `M11`). Listing 1.4 in the Appendix provides the corresponding OCL consistency constraints based on mapping `M12`.

The chronological order of the `FeatureCall` instances in the recorded trace must be preserved in the interaction model. Therefore we require that the message occurrences serving as `send` and `receiveEvents` of the derived messages (see `M5`, `M6`, `M12`) preserve this order on the respective lifelines (along with the execution occurrences). This means, that after receiving a message (`receiveEvent`), the send events derived from called nested features are added in form of events covering the lifeline. In case of synchronous calls with owned return values, for each message, the receive event related to the reply message enters the set of ordered events (see `M12`) before adding the send event of the next call.

3 Views on Test-Execution Traces

In this section, we discuss how the mappings from Sect. 2 can be extended to render the derived interaction models tailorable. By tailoring, we refer to specific means for zooming in and out on selected details of a test-execution trace; and for pruning selected details. For this purpose, our approach defines a scenario-test viewpoint. A viewpoint [4] stipulates the element types (e.g. scenario-test parts, feature-call scopes) and the types of relationships between these element types (e.g. selected, unselected) available for defining different views on test-execution traces. On the one hand, applying the viewpoint allows for controlling model-size explosion. On the other hand, the views offered on the derived models can help tailor the corresponding behavior documentation for given tasks (e.g. test or code reviews) and/or stakeholder roles (e.g. test developer, software architect).

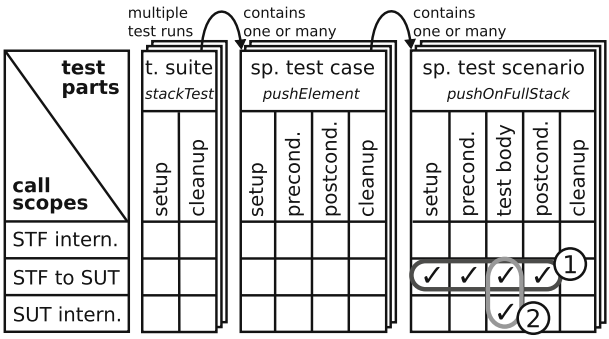

Fig. 7. Example of an option space for configuring views on test-execution traces by combining scenario-test parts and feature-call scopes.

3.1 Scenario-Test Viewpoint

To tailor the derived interaction models, two characteristics of scenario tests and the corresponding scenario-test traces can be leveraged: the whole-part structure of scenario tests and trackable feature-call scopes.

Scenario-Test Parts. Scenario tests, in terms of concepts and their specification structure, are composed of different parts (see Sect. 2.1 and Fig. 3):

- A *test suite* encompasses one or more test cases.
- A *test case* comprises one or more test scenarios.
- A test case, and a *test scenario* can contain assertion blocks to specify *pre-* and *post-conditions*.
- A test suite, a test case, and a test scenario can contain exercise blocks, as *setup*, or *cleanup* procedures.
- A test scenario contains a *test body*.

Feature-Call Scopes. Each feature call in a scenario-test trace is scoped according to the scenario-test framework (STF) and the system under test (SUT), respectively, as the source and the target of the feature call. This way, we can differentiate between three feature-call scopes:

- feature calls running from the STF to the SUT (i.e. test stimuli),
- feature calls internal to the SUT (triggered by test stimuli directly and indirectly),
- feature calls internal to the STF.

The scenario-test parts and feature-call scopes form a large option space for tailoring an interaction model. In Fig. 7, these tailoring options are visualized as a configuration matrix. For instance, a test suite containing one test case with just one included test scenario offers 14,329 different interaction-model views available for configuration based on one test run (provided that the corresponding test blocks are specified).[4]

3.2 Example

In this section, we demonstrate *by example* the relevance of specifying different views on the test-execution traces for different tasks and/or stakeholder roles. A stack-based dispenser component (one element of an exemplary SUT) is illustrated in Fig. 8. A `Stack` provides the operations `push`, `pop`, `size`, and `full` as well as the attributes `limit` and `element`, which are accessible via corresponding getter/setter operations (i.e. `getElements`, `getLimit` and `setLimit`).

[4] The number of views computes as follows: There are (2^3-1) non-empty combinations of the three feature-call scopes (SUT internal, STF internal, STF to SUT) times the $(2^{11}-1)$ non-empty combinations of at least 11 individual test parts (e.g. setup of test case, test body of test scenario).

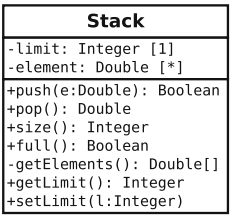

Fig. 8. Excerpt from a UML class diagram of an exemplary SUT.

Listing 1.1. Natural-language notation of scenario `pushOnFullStack`.

```
1 Given: 'that a specific instance of Stack contains elements of the size of 2
         and has a limit of 2'
2 When:  'an element is pushed on the instance of Stack'
3 Then:  'the push operation fails and the size of elements is still 2'
```

Listing 1.2. Excerpt from an exemplary test script specifying test scenario `pushOnFullStack`.

```
1  set fs [::STORM::TestScenario new -name pushOnFullStack -testcase pushElement]
2  $fs expected_result set 0
3  $fs setup_script set {
4    [::Stack info instances] limit set 2
5  }
6  $fs preconditions set {
7    {expr {[[::Stack info instances] size] == 2}}
8    {expr {[[::Stack info instances] limit get] == 2}}
9  }
10 $fs test_body set {
11   [::Stack info instances] push 1.4
12 }
13 $fs postconditions set {
14   {expr {[[::Stack info instances] size] == 2}}
15 }
```

Consider the example of a test developer whose primary task is to conduct a test-code review. For this review, she is responsible for verifying a test-scenario script against a scenario-based requirements description. The scenario is named `pushOnFullStack` and specified in Listing 1.1. The excerpt from the test script to be reviewed is shown in Listing 1.2. To support her in this task, our approach can provide her with a partial UML sequence diagram which reflect only selected details of the test-execution trace. These details of interest could be interactions triggered by specific blocks of the test under review, for example. Such a view provides immediate benefits to the test developer. The exemplary view in Fig. 9 gives details on the interactions between the STF and the SUT, i.e. the test stimuli observed under this specific scenario. To obtain this view, the configuration pulls feature calls from a combination of setup, precondition, test body and postcondition specific to this test scenario. The view from Fig. 9 corresponds to configuration ① in Fig. 7.

As another example, consider a software architect of the same SUT. The architect might be interested in how the system behaves when executing the test body of the given scenario `pushOnFullStack`. The architect prefers a behavior documentation which additionally provides details on the interaction between SUT instances. A sequence diagram for such a view is presented in Fig. 10. This second view effectively zooms into a detail of the first view in Fig. 9, namely the inner workings triggered by the message `push(1,4)`. The second view reflects configuration ② in Fig. 7.

3.3 Viewpoint Mappings

UML interaction models and corresponding sequence diagrams allow for realizing immediate benefits from a scenario-test viewpoint. For example, sequence

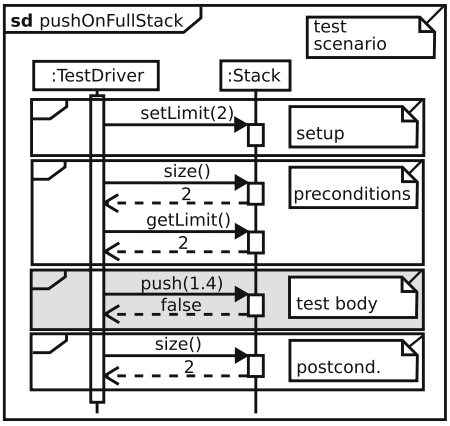

Fig. 9. Sequence diagram derived from `pushOnFullStack` highlighting calls running from STF to SUT.

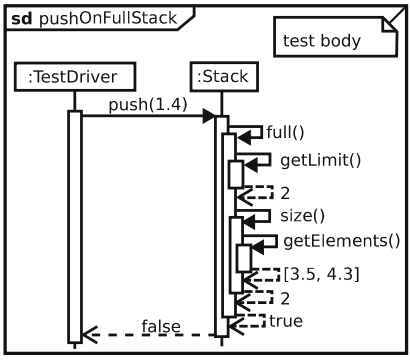

Fig. 10. Sequence diagram derived from `pushOnFullStack` zooming in on test body and representing both, calls running from STF to SUT and calls internal to the SUT.

diagrams provide notational elements which can help in communicating the scenario-test structure (suite, case, scenario) to different stakeholders (architects, developers, and testers). These notational features include combined fragments and references. This way, a selected part can be visually marked in a diagram showing a combination of test parts (see, e.g., Fig. 9). Alternatively, a selected part of a scenario test can be highlighted as a separate diagram (see Fig. 10). On the other hand, interaction models can be tailored to contain only interactions between certain types of instances. Thereby, the corresponding sequence diagram can accommodate views required by different stakeholders of the SUT. In Fig. 9, the sequence diagram highlights the test stimuli triggering the test scenario `pushOnFullStack`, whereas the diagram in Fig. 10 additionally depicts SUT internal calls.

Conceptually, we represent different views as models conforming to the view metamodel in Fig. 11. In essence, each view selects one or more test parts and feature-call scopes, respectively, to be turned into an interaction model. Generating the actual partial interaction model is then described by six additional transML mapping actions based on a view and a trace model (see `M13–18` in Fig. 12). In each mapping action, a given view model (`view`) is used to verify whether a given element is to be selected for the chosen scope of test parts and call scopes. Upon its selection, a feature call with its call dependencies is processed according to the previously introduced mapping actions (i.e. `M1–M9`, `M11`, and `M12`).

Mappings Specific to Call Scope. As explained in Sect. 3.1, a view can define any, non-empty combination of three call scopes: *STF internal*, *SUT internal*, and *STF to SUT*. In mapping action `M18`, each feature call is evaluated according

Fig. 11. View metamodel.

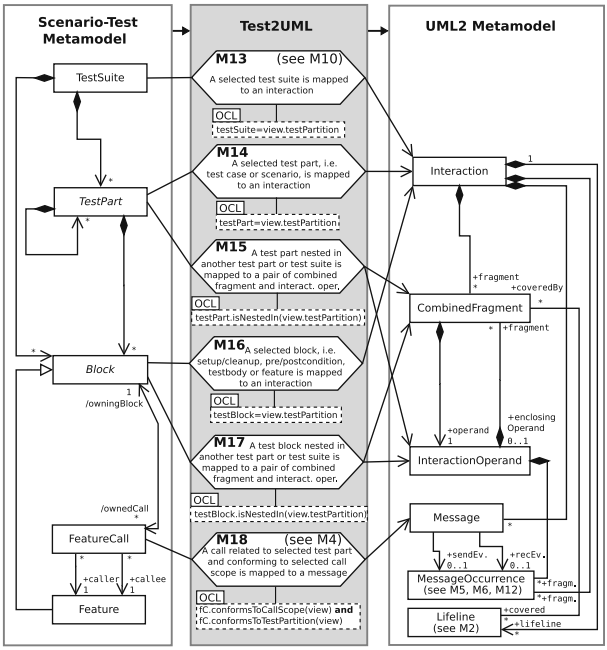

Fig. 12. Mappings specific to a given configured `view` with `callScope` and `testPartition`. For clarity, the case for configuring a view with one call scope and one test partition is depicted.

to the structural affiliations of the calling and the called feature, respectively. For details, see the OCL helper operation `conformsToCallScope(v:View)` in Listing 1.5 shown in the Appendix. Note that in case of explicitly documenting SUT behavior (i.e. *SUT internal* and *STF to SUT*), lifelines can alternatively just represent SUT instances. In this case, the `sendEvent` of each call running from STF to SUT (and, in turn, each `receiveEvent` of the corresponding `reply` message) is represented by a `Gate` instance (instead of `MessageOccurrence`) which signifies in the UML a connection point for relating messages outside with inside an interaction fragment.

Mappings Specific to Test Partition. The viewpoint provides for mapping structural elements of the STF to structural elements of UML interactions to highlight feature calls in their scenario-test context. Relevant contexts are the STF and scenario-test blocks (see `M13`–`M17` in Fig. 12). Feature calls relate directly to a test block, with the call definition being contained by a

block, or indirectly along a feature-call chain. This way, the STF and the respective test parts responsible for a trace can selectively enter a derived interaction as participants (e.g. as a test-driver lifeline). Besides, the scenario-test blocks and parts nested in the responsible test part (e.g. case, scenario, setup, precondition) can become structuring elements within an enclosing interaction, such as combined fragments. Consider, for example, a test suite being selected entirely. The trace obtained from executing the `TestSuite` instance is mapped to an instance of `Interaction` (`M13` in Fig. 12). Scenario-test parts such as test cases and test scenarios, as well as test blocks, also become instances of `Interaction` when they are selected as active partition in a given view (`M14, M16`). Alternatively, they become instances of `CombinedFragment` along with corresponding interaction operands (`M15, M17`), when they are embedded with the actually selected scenario-test part (see `isNestedIn(p:Partition)` in Listing 1.5 in the Appendix). Hierarchical ownership of one (child) test part by another (parent) part is recorded accordingly as `enclosingOperand` relationship between child and parent parts. The use of combined fragments provides for a general structuring of the derived interaction model according to the scenario-test structure. All feature calls associated with given test parts (see `M18` in Fig. 12 and the mapping constraint `conformsToTestPartition(v:View)` in Listing 1.5 in the Appendix) are effectively grouped because their corresponding message occurrences and execution occurrences (both being a kind of `InteractionFragment`) become linked to a combined fragment via an enclosing interaction operand. Combined fragments also establish a link to the `Lifeline` instances representing the SUT instances interacting in a given view. To maintain the strict chronological order of feature calls in a given trace, the resulting combined fragments must apply the `InteractionOperator` *strict* (see Sect. 2.1).[5]

4 Prototype Implementation

The KaleidoScope[6] tool can derive tailored UML2 interaction models from scenario-based runtime tests. Figure 13 depicts a high-level overview of the derivation procedure supported by KaleidoScope. The architectural components of KaleidoScope (STORM, trace provider, and model builder) as well as the diagram editor are represented via different swimlanes. Artifacts required and resulting from each derivation step are depicted as input and output pins of the respective action.

4.1 Used Technologies

The "Scenario-based Testing of Object-oriented Runtime Models" (STORM) test framework provides an infrastructure for specifying and for executing

[5] The default value *seq* provides weak sequencing, i.e. ordering of fragments just along lifelines, which means that occurrences on different lifelines from different operands may come in any order [27].

[6] Available for download from our website [14].

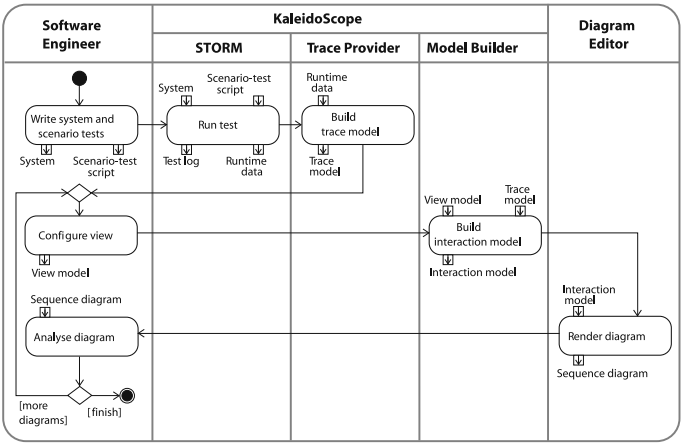

Fig. 13. Process of deriving tailored interaction models with KaleidoScope.

scenario-based component tests [34]. STORM provides all elements of our scenario-based testing metamodel (see Fig. 3). KaleidoScope builds on and instruments STORM to obtain execution-trace data from running tests defined as STORM test suites. This way, KaleidoScope keeps adoption barriers low because existing STORM test specifications can be reused without modification. STORM is implemented using the dynamic object-oriented language "Next Scripting Language" (NX), an extension of the "Tool Command Language" (Tcl). As KaleidoScope integrates with STORM, we also implemented KaleidoScope via NX/Tcl. In particular, we chose this development environment because NX/Tcl provides numerous advanced dynamic runtime introspection techniques for collecting execution traces from scenario tests. For example, NX/Tcl provides built-in method-call introspection in terms of message interceptors [36] and callstack introspection. KaleidoScope records and processes execution traces, as well as view configuration specifications, in terms of EMF models (Eclipse Modeling Framework; i.e. Ecore and MDT/UML2 models). More precisely, the models are stored and handled in their Ecore/XMI representation (XML Metadata Interchange specification [26]). For transforming our trace models into UML models, the required model transformations [6] are implemented via "Query/View/Transformation Operational" (QVTo) mappings [25]. QVTo allows for implementing concrete model transformations based on conceptual mappings in a straightforward manner.

4.2 Derivation Actions

Run Scenario Tests. For deriving interaction models via KaleidoScope, a newly created or an existing scenario-test suite is executed by the STORM engine. At this point, and from the perspective of the software engineer, this derivation-enabled test execution does not deviate from an ordinary one.

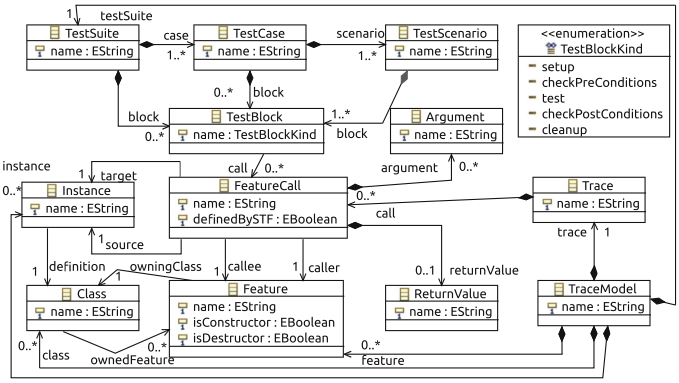

Fig. 14. Trace metamodel, EMF Ecore.

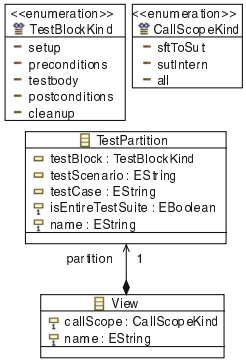

Fig. 15. View metamodel, EMF Ecore.

The primary objective of this test run is to obtain the runtime data required to build a trace model. Relevant runtime data consist of scenario-test traces (SUT feature calls and their call dependencies) and structural elements of the scenario test (a subset of STF feature calls and their call dependencies).

Build Trace Models. Internally, the trace-provider component of KaleidoScope instruments the STORM engine before the actual test execution to record the corresponding runtime data. This involves intercepting each call of relevant features and deriving the corresponding call dependencies. At the same time, the trace provider ascertains that its instrumentation remains transparent to the STORM engine. To achieve this, the trace provider instruments the STORM engine and the tests under execution using NX/Tcl introspection techniques. In NX/Tcl, method-call introspection is supported via two variants of message interceptors [36]: mixins and filters. Mixins [37] can be used to decorate entire components and objects. Thereby, they intercept calls to methods which are known a priori. In KaleidoScope, the trace provider registers a mixin to intercept relevant feature calls on the STF, i.e. the STORM engine. Filters [23] are used by the trace provider to intercept calls to objects of the SUT which are not known beforehand. To record relevant feature-call dependencies, the trace provider uses the callstack introspection offered by NX/Tcl. NX/Tcl offers access to its operation callstack via special-purpose introspection commands, e.g. `nx::current`, see [22]. To collect structural data on the intercepted STF and SUT instances, the trace provider piggybacks onto the structural introspection facility of NX/Tcl, e.g., `info` methods, see [22]. This way, structural data such as class names, feature names, and relationships between classes can be requested. The collected runtime data is then processed by the trace provider. In particular, feature calls at the application level are filtered to include only calls for the scope of the SUT. This way, calls into other system contexts (e.g., external components or lower-level host language calls) are discarded. In addition, the execution traces are reordered to report “invocations interactions” first and “return interactions”

second. Moreover, the recorded SUT calls are linked to the respective owning test blocks. The processed runtime data is then stored as a trace model which conforms to the `Trace` metamodel defined via Ecore (see Fig. 14). This resulting trace model comprises the relevant structural elements (test suite, test case and test scenario), the SUT feature calls and their call dependencies, each being linked to a corresponding test block.

Configure Views. Based on the specifics of the test run (e.g. whether an entire test suite or selected test cases were executed) and the kind of runtime data collected, different views are available to the software engineer for selection. In KaleidoScope, the software engineer can select a particular view by defining a view model. This view model must conform to the `View` metamodel specified using Ecore (see Fig. 15). KaleidoScope allows for configuring views on the behavior of the SUT by combining a selected call scope (SUT internal, STF to SUT, or both) and a selected test partition (entire test suite or a specific test case, scenario, or block), as described in Sect. 3.

Build Interaction Models. The model-builder component of KaleidoScope takes the previously created pair of a trace model and a view model as input models for a collection of QVTo model transformations. The output model of these QVTo transformations is the UML interaction model. The conceptual mappings presented in Subsects. 2.3 and 3.3 are implemented in QVT Operational mappings [25], including the linking of relationships between the derived elements. In total, the transformation file contains 24 mapping actions.

Display Sequence Diagrams. Displaying the derived interaction models as sequence diagrams and presenting them to the software engineer is not handled by KaleidoScope itself. As the derived interaction models are available in the XMI representation, they can be imported by XMI-compliant diagram editors. In our daily practice, we use Eclipse Papyrus [8] for this task.

5 Related Work

Closely related research can be roughly divided into three groups: reverse-engineering sequence diagrams from system execution, techniques addressing the problem of model-size explosion in reverse-engineered behavioral models and extracting traceability links between test and system artifacts.

Reverse-Engineering UML Sequence Diagrams. Approaches applying dynamic analysis set the broader context of our work [2,7,12,28]. Of particular interest are model-driven approaches which provide conceptual mappings between runtime-data models and UML interaction models. Briand et al. [2] as well as Cornelissen et al. [5] are exemplary for such model-driven approaches. In their approaches, UML sequence diagrams are derived from executing runtime tests. Both describe metamodels to define sequence diagrams and for capturing system execution in form of a trace model. Briand et al. define mappings between these two metamodels in terms of OCL consistency constraints.

Each test execution relates to a single use-case scenario defined by a system-level test case. Their approaches differ from ours in some respects. The authors build on generic trace metamodels while we extend an existing scenario-test metamodel to cover test-execution traces. Briand et al. do not provide for scoping the derived sequence diagrams based on the executed tests unlike Cornelissen et al. (see below). They, finally, do not capture the mappings between trace and sequence model in a formalized way.

Countering Model-Size Explosion. A second group of related approaches aims at addressing the problem of size explosion in reverse-engineered behavioral models. Fernández-Sáez et al. [10] conducted a controlled experiment on the perceived effects of derived UML sequence diagrams on maintaining a software system. A key result is that derived sequence diagrams do not necessarily facilitate maintenance tasks due to an excessive level of detail. Hamou-Lhadj and Lethbridge [16] and Bennett et al. [1] surveyed available techniques which can act as counter measures against model-size explosion. The available techniques fall into three categories: *slicing* and *pruning* of components and calls as well as *architecture-level filtering*. *Slicing* (or sampling) is a way of reducing the resulting model size by choosing a sample of execution traces. Sharp and Rountev [33] propose interactive slicing for zooming in on selected messages and message chains. Grati et al. [11] contribute techniques for interactively highlighting selected execution traces and for navigating through single execution steps. *Pruning* (or hiding) provides abstraction by removing irrelevant details. For instance, Lo and Maoz [20] elaborate on filtering calls based on different execution levels. In doing so, they provide hiding of calls based on the distinction between triggers and effects of scenario executions. As an early approach of *architectural-level filtering*, Richner and Ducasse [31] provide for tailorable views on object-oriented systems, e.g., by filtering calls between selected classes. In our approach, we adopt these techniques for realizing different views conforming to a scenario-test viewpoint. In particular, slicing corresponds to including interactions of certain test parts (e.g., test cases, test scenarios) only, selectively hiding model elements to pulling from different feature-call scopes (e.g., stimuli and internal calls). Architectural-level filtering is applied by distinguishing elements by their structural affiliation (e.g., SUT or STF).

Test-to-System Traceability. Another important group of related work provides for creating traceability links between test artifacts and system artifacts by processing test-execution traces. Parizi et al. [29] give a systematic overview of such traceability techniques. For instance, test cases are associated with SUT elements based on the underlying call-trace data for calculating metrics which reflect how each method is tested [19]. Qusef et al. [30] provide traceability links between unit tests and classes under test. These links are extracted from trace slices generated by assertion statements contained by the unit tests. In general, these approaches do not necessarily derive behavioral diagrams, however Parizi et al. conclude by stating the need for visualizing traceability links. These approaches relate to ours by investigating which SUT elements are covered by a specific part of the test specification. While they use this information, e.g.,

for calculating coverage metrics, we aim at visualizing the interactions for documenting system behavior. However, Cornelissen et al. [5] pursue a similar goal by visualizing the execution of unit tests. By leveraging the structure of tests, they aim at improving the understandability of reverse-engineered sequence diagrams (see above), e.g., by representing the behavior of a particular test part in a separate sequence diagram. While they share our motivation for test-based partitioning, Cornelissen et al. do not present a conceptual or a concrete solution to this partitioning. Moreover, we leverage the test structure for organizing the sequence diagram (e.g., by using combined fragments) and consider different scopes of feature calls.

6 Conclusion

In this paper, we presented an approach for deriving tailored UML interaction models for documenting system behavior from scenario-based runtime tests. Our approach allows for leveraging the structure of scenario tests (i.e. test parts and call scopes) to tailor the derived interaction models, e.g., by pruning details and by zooming in and out on selected details. This way, we also provide means to control the sizes of the resulting UML sequence diagrams. Our approach is model-driven in the sense that test-execution traces are represented through a dedicated metamodel. Conceptual mappings (transformation rules) between this metamodel and the UML metamodel are captured by transML diagrams refined by inter-model constraint expressions (OCL). To demonstrate the feasibility of our approach, we developed a prototype implementation called KaleidoScope. The approach is applicable for any software system having an object-oriented design and implementation, provided that suitable test suites and a suitable test framework are available. A test suite (and the guiding test strategy) is qualified if tests offer structuring abstractions (i.e. test parts as in scenario tests) and if tests trigger inter-object interactions. The corresponding test framework must offer instrumentation to obtain test-execution traces.

In a next step, we will investigate via controlled experiments how the derived interaction models can support system stakeholders in comprehension tasks on the tested software system and on the test scripts. From a conceptual point of view, we plan to extend the approach to incorporate behavioral details such as measured execution times into the interaction models. From a practical angle, we seek to apply the approach on large-scale software projects. For this, our KaleidoScope must be extended to support runtime and program introspection for other object-oriented programming languages and for the corresponding test frameworks.

Appendix

Listing 1.3. OCL consistency constraints based on mapping M4 in Fig. 5.

```
context M4 inv:
 message.name=featureCall.name and
 (featureCall.argument->notEmpty() implies message.
       argument.name=featureCall.argument.name) and
 message.sendEvent.oclIsTypeOf(
       MessageOccurrenceSpecification) and
 message.sendEvent.name=featureCall.caller.name and
 message.sendEvent.covered.name=featureCall.source.
       name and
 message.sendEvent.covered.represents.name=
       featureCall.source.name and
 message.sendEvent.covered.represents.type.name=
       featureCall.source.definingClass.name and
 message.receiveEvent.oclIsTypeOf(
       MessageOccurrenceSpecification) and
 message.receiveEvent.name=featureCall.callee.name
       and
 if(featureCall.callee.oclIsTypeOf(Constructor)) then
       {
  message.messageSort=MessageSort::createMessage and
  message.signature.name='create' and
  message.receiveEvent.covered.name=featureCall.
       returnValue.value and
  message.receiveEvent.covered.represents.name=
       featureCall.returnValue.value and
  message.receiveEvent.covered.represents.type.name=
       featureCall.target.name
 } else {
  message.receiveEvent.covered.name=featureCall.
       target.name and
  message.receiveEvent.covered.represents.name=
       featureCall.target.name and
  message.receiveEvent.covered.represents.type.name=
       featureCall.target.definingClass.name and
  if (featureCall.callee.oclIsTypeOf(Destructor))
       then {
   message.messageSort=MessageSort::deleteMessage and
   message.signature.name='delete'
  } else {
   message.messageSort=MessageSort::synchCall and
   message.signature.name=featureCall.callee.name and
   (featureCall.returnValue->isEmpty() implies
        message.receiveEvent.execution.finish.
        oclIsTypeOf(ExecutionOccurrence))
  } endif
 } endif
```

Listing 1.4. OCL consistency constraints based on mapping M12 in Fig. 6.

```
context M12 inv:
 message.messageSort=MessageSort::reply and
 message.name=returnValue.value and
 message.signature.name=returnValue.value and
 message.argument->isEmpty() and
 message.sendEvent.oclIsTypeOf(
       MessageOccurrenceSpecification) and
 message.sendEvent.name=returnValue.featureCall.
       callee.name and
 message.sendEvent.covered.name=returnValue.
       featureCall.target.name and
 message.sendEvent.covered.represents.name=
       returnValue.featureCall.target.name and
 message.sendEvent.covered.represents.type.name=
       returnValue.featureCall.target.definingClass.
       name and
 message.receiveEvent.oclIsTypeOf(
       MessageOccurrenceSpecification) and
 message.receiveEvent.name=returnValue.featureCall.
       caller.name and
 message.receiveEvent.covered.name=returnValue.
       featureCall.source.name and
 message.receiveEvent.covered.represents.name=
       returnValue.featureCall.source.name and
 message.receiveEvent.covered.represents.type.name=
       returnValue.featureCall.source.definingclass.
       name
```

Listing 1.5. OCL helper operations applied in mappings M15, M17 and M18 in Fig. 12.

```
context FeatureCall
def: conformsToCallScope(v:View) : Boolean =
 if (v.callScope='sutIntern') then {
  self.isDefinedByStfBlock=false and
  self.calleeOwnedByStfClass=false
 } else {
  if (v.callScope='stfToSut') then {
   self.isDefinedByStfBlock and
   self.calleeOwnedByStfClass=false
  } else {
   if (v.callScope='stfIntern') then {
    self.isDefinedByStfBlock and
    self.calleeOwnedByStfClass
   } else { false } endif
  } endif
 } endif
def: conformsToTestPartition(v:View) : Boolean =
 self.owningBlock.isNestedIn(v.testPartition)
def: isDefinedByTestBlock : Boolean =
 block.oclIsTypeOf(Setup) or
 block.oclIsTypeOf(TestBody) or
 block.oclIsTypeOf(Cleanup) or
 (block.oclIsTypeOf(Assertion)implies(block.block.
       oclIsTypeOf(Precondition) or
  block.block.oclIsTypeOf(Postcondition))
def: calleeOwnedByStfClass : Boolean =
 Set{TestSuite, TestCase, TestScenario, Setup,
       Precondition, TestBody, Postcondition,
       Cleanup}->includes(self.callee.owningClass.
       name)
def: block : Block = self.definition.Block

context TestPart
def: isNestedIn(p:TestPartition) : Boolean =
 if (p.oclIsTypeOf(TestSuite)) then {
  true
 } else {
  if (p.oclIsTypeOf(TestCase)) then {
   (self.oclIsTypeOf(TestCase) implies p.name=self.
        name) and
   (self.oclIsTypeOf(TestScenario) implies p.name=
        self.testCase.name) and
   (self.oclIsTypeOf(Block) implies (
    (self.testCase->notEmpty() and p.name=self.
         testCase.name) or
    (self.testScenario->notEmpty() and p.name=self.
         testScenario.testCase.name)))
  } else {
   if (p.oclIsTypeOf(TestScenario)) then {
    (not self.oclIsTypeOf(TestCase)) and
    (self.oclIsTypeOf(TestScenario) implies (
     p.name=self.name and
     p.testCase.name = self.testCase.name
    )) and
    (self.oclIsTypeOf(Block) implies (
     p.name=self.testScenario.name and
     p.testCase.name=self.testScenario.testCase.name
         ))
   } else {
    if (p.oclIsTypeOf(Block)) then {
     self.oclIsTypeOf(Block) and p.name=self.name
          and
      ((p.testCase->notEmpty() and self.testCase->
           notEmpty()) implies p.testCase.name =
           self.testCase.name) and
      (p.testScenario->notEmpty() and self.
           testScenario->notEmpty()) implies p.
           testScenario.name = self.testScenario.
           name))
    } else { false } endif
   } endif
  } endif
 } endif
```

References

1. Bennett, C., Myers, D., Storey, M.A., German, D.M., Ouellet, D., Salois, M., Charland, P.: A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. Softw. Maint. Evol. **20**(4), 291–315 (2008). doi:10.1002/smr.v20:4
2. Briand, L.C., Labiche, Y., Miao, Y.: Toward the reverse engineering of UML sequence diagrams. In: Proceedings of WCRE 2003, pp. 57–66. IEEE (2003). doi:10.1109/TSE.2006.96
3. Carroll, J.M.: Five reasons for scenario-based design. Interact. Comput. **13**(1), 43–60 (2000). doi:10.1016/S0953-5438(00)00023-0
4. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architecture: Views and Beyond. SEI, 2nd edn. Addison-Wesley, Boston (2011)
5. Cornelissen, B., Van Deursen, A., Moonen, L., Zaidman, A.: Visualizing testsuites to aid in software understanding. In: Proceedings of CSMR 2007, pp. 213–222. IEEE (2007). doi:10.1109/CSMR.2007.54
6. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: WS Proceedings of OOPSLA 2003, pp. 1–17. ACM Press (2003)
7. Delamare, R., Baudry, B., Le Traon, Y., et al.: Reverse-engineering of UML 2.0 sequence diagrams from execution traces. In: WS Proceedings of ECOOP 2006. Springer (2006)
8. Eclipse Foundation: Papyrus (2015). http://eclipse.org/papyrus/. Accessed 25 September 2015
9. Falessi, D., Briand, L.C., Cantone, G., Capilla, R., Kruchten, P.: The value of design rationale information. ACM Trans. Softw. Eng. Methodol. **22**(3), 21:1–21:32 (2013). doi:10.1145/2491509.2491515
10. Fernández-Sáez, A.M., Genero, M., Chaudron, M.R., Caivano, D., Ramos, I.: Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance? a family of experiments. Inform. Softw. Tech. **57**, 644–663 (2015). doi:10.1016/j.infsof.2014.05.014
11. Grati, H., Sahraoui, H., Poulin, P.: Extracting sequence diagrams from execution traces using interactive visualization. In: Proceedings of WCRE 2010, pp. 87–96. IEEE (2010). doi:10.1109/WCRE.2010.18
12. Guéhéneuc, Y.G., Ziadi, T.: Automated reverse-engineering of UML v2.0 dynamic models. In: WS Proceedings of ECOOP 2005. Springer (2005)
13. Guerra, E., Lara, J., Kolovos, D.S., Paige, R.F., Santos, O.M.: Engineering model transformations with transML. Softw. Syst. Model. **12**(3), 555–577 (2013). doi:10.1007/s10270-011-0211-2
14. Haendler, T.: KaleidoScope. Institute for Information Systems and New Media. WU Vienna (2015). http://nm.wu.ac.at/nm/haendler. Accessed 25 September 2015
15. Haendler, T., Sobernig, S., Strembeck, M.: An approach for the semi-automated derivation of UML interaction models from scenario-based runtime tests. In: Proceedings of ICSOFT-EA 2015, pp. 229–240. SciTePress (2015). doi:10.5220/0005519302290240
16. Hamou-Lhadj, A., Lethbridge, T.C.: A survey of trace exploration tools and techniques. In: Proceedings of CASCON 2004, pp. 42–55. IBM Press (2004). http://dl.acm.org/citation.cfm?id=1034914.1034918

17. Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach. ACM Press Series. ACM Press, New York (1992)
18. Jarke, M., Bui, X.T., Carroll, J.M.: Scenario management: an interdisciplinary approach. Requirements Eng. **3**(3), 155–173 (1998). doi:10.1007/s007660050002
19. Kanstrén, T.: Towards a deeper understanding of test coverage. Softw. Maint. Evol. **20**(1), 59–76 (2008). doi:10.1002/smr.362
20. Lo, D., Maoz, S.: Mining scenario-based triggers and effects. In: Proceedings of ASE 2008, pp. 109–118. IEEE (2008). doi:10.1109/ASE.2008.21
21. Nebut, C., Fleurey, F., Le Traon, Y., Jezequel, J.: Automatic test generation: a use case driven approach. IEEE Trans. Softw. Eng. **32**(3), 140–155 (2006). doi:10.1109/TSE.2006.22
22. Neumann, G., Sobernig, S.: Next scripting framework. API reference (2015). https://next-scripting.org/xowiki/. Accessed 25 September 2015
23. Neumann, G., Zdun, U.: Filters as a language support for design patterns in object-oriented scripting languages. In: Proceedings of COOTS 1999, pp. 1–14. USENIX (1999). http://dl.acm.org/citation.cfm?id=1267992
24. Object Management Group: Object Constraint Language (OCL) - Version 2.4 (2014). http://www.omg.org/spec/OCL/2.4/. Accessed 25 September 2015
25. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.2, February 2015. http://www.omg.org/spec/QVT/1.2/. Accessed 25 September 2015
26. Object Management Group: MOF 2 XMI Mapping Specification, Version 2.5.1, June 2015. http://www.omg.org/spec/XMI/2.5.1/. Accessed 25 September 2015
27. Object Management Group: Unified Modeling Language (UML), Superstructure, Version 2.5.0, June 2015. http://www.omg.org/spec/UML/2.5. Accessed 25 September 2015
28. Oechsle, R., Schmitt, T.: JAVAVIS: Automatic program visualization with object and sequence diagrams using the java debug interface (JDI). In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 176–190. Springer, Heidelberg (2002). doi:10.1007/3-540-45875-1_14
29. Parizi, R.M., Lee, S.P., Dabbagh, M.: Achievements and challenges in state-of-the-art software traceability between test and code artifacts. Trans. Reliab. IEEE **63**, 913–926 (2014). doi:10.1109/TR.2014.2338254
30. Qusef, A., Bavota, G., Oliveto, R., de Lucia, A., Binkley, D.: Recovering test-to-code traceability using slicing and textual analysis. J. Syst. Softw. **88**, 147–168 (2014). doi:10.1016/j.jss.2013.10.019
31. Richner, T., Ducasse, S.: Recovering high-level views of object-oriented applications from static and dynamic information. In: Proceedings of ICSM 1999, pp. 13–22. IEEE (1999). http://dl.acm.org/citation.cfm?id=519621.853375
32. Ryser, J., Glinz, M.: A scenario-based approach to validating and testing software systems using statecharts. In: Proceedings of ICSSEA 1999 (1999)
33. Sharp, R., Rountev, A.: Interactive exploration of UML sequence diagrams. In: Proceedings of VISSOFT 2005, pp. 1–6. IEEE (2005). doi:10.1109/VISSOF.2005.1684295
34. Strembeck, M.: Testing policy-based systems with scenarios. In: Proceedings of IASTED 2011, pp. 64–71. ACTA Press (2011). doi:10.2316/P.2011.720-021
35. Van Geet, J., Zaidman, A., Greevy, O., Hamou-Lhadj, A.: A lightweight approach to determining the adequacy of tests as documentation. In: Proceedings of PCODA 2006, pp. 21–26. IEEE CS (2006)
36. Zdun, U.: Patterns of tracing software structures and dependencies. In: Proceedings of EuroPLoP 2003, pp. 581–616. Universitaetsverlag Konstanz (2003)

37. Zdun, U., Strembeck, M., Neumann, G.: Object-based and class-based composition of transitive mixins. Inform. Softw. Tech. **49**(8), 871–891 (2007). doi:10.1016/j.infsof.2006.10.001
38. Ziadi, T., Da Silva, M.A.A., Hillah, L.M., Ziane, M.: A fully dynamic approach to the reverse engineering of UML sequence diagrams. In: Proceedings of ICECCS 2011, pp. 107–116. IEEE (2011). doi:10.1109/ICECCS.2011.18