Natural-Language Scenario Descriptions for Testing Core Language Models of Domain-specific Languages

Bernhard Hoisl[1,2], Stefan Sobernig[1], and Mark Strembeck[1,2]
[1]*Institute for Information Systems and New Media, WU Vienna, Austria*
[2]*Secure Business Austria Research (SBA Research), Austria*
{*firstname.lastname*}*@wu.ac.at*

Keywords: Domain-specific Modeling, Natural-Language Requirement, Scenario-based Testing, Metamodel Testing, Eclipse Modeling Framework, Xtext, Epsilon, EUnit.

Abstract: The core language model is a central artifact of domain-specific modeling languages (DSMLs) as it captures all relevant domain abstractions and their relations. Natural-language scenarios are a means to capture requirements in a way that can be understood by technical as well as non-technical stakeholders. In this paper, we use scenarios for the testing of structural properties of DSML core language models. In our approach, domain experts and DSML engineers specify requirements via structured natural-language scenarios. These scenario descriptions are then automatically transformed into executable test scenarios providing forward and backward traceability of domain requirements. To demonstrate the feasibility of our approach, we used Eclipse Xtext to implement a requirements language for the definition of semi-structured scenarios. Transformation specifications generate executable test scenarios that run in our test platform which is built on the Eclipse Modeling Framework and the Epsilon language family.

1 INTRODUCTION

A domain-specific language (DSL) provides tailored development support for a specific domain (Stahl and Völter, 2006). In model-driven development (MDD), a domain-specific *modeling* language (DSML) allows for developing tailored, platform-independent models. Their abstract syntax is typically defined using metamodeling and it is exposed to domain modelers in terms of a diagrammatic concrete syntax (Sendall and Kozaczynski, 2003).

The process of developing a DSML, either from scratch or by reusing existing DSMLs, involves an initial phase of domain analysis (Lisboa et al., 2010; Czarnecki and Eisenecker, 2000). A *domain analysis* aims at documenting the domain knowledge in terms of the domain vocabulary and the domain requirements (e.g., rules of applying the domain terms, normative procedural guidelines). In the *domain modeling* step, the data sources (e.g., code bases and application documentation available for the domain) are collected and reviewed to identify domain-specific entities, operations, and entity relationships. In a model-driven approach, this step yields a number of model artifacts (e.g., the core language model with accompanying optional constraints).

The *core language model* of a DSML captures the abstracted domain entities and their relationships (Strembeck and Zdun, 2009). This way, it defines the abstract syntax of a DSML. A DSML's core language model is often defined as a metamodel conforming to standards, such as the Meta Object Facility (Object Management Group, 2013) or as an extension to MOF-based general-purpose modeling languages such as the Unified Modeling Language (Object Management Group, 2011). Supplementary models can describe the commonalities and variations between domain entities defined by the core language model (variability models) and domain operations (e.g., relevant business processes). After domain modeling is finished, the resulting models are reviewed to validate the conformance of these domain models with the domain vocabulary and domain requirements.

In a domain modeling activity, domain requirements are frequently documented using natural-language descriptions (Neill and Laplante, 2003; Diethelm et al., 2005). This is due to the fact that domain requirements emerge and are often elicited during interviews, discussions, and meetings where different stakeholders are involved (Sutcliffe, 2002). Whereas such natural-language requirements descriptions are more accessible to non-technical stakeholders and stakeholders of diverse professional backgrounds (Dwarakanath and Sengupta, 2012), a natural-language description raises important issues when it comes to validating model artifacts such as the core language model against the domain requirements. A corresponding issue is, for example, the ambiguity of natural-language descriptions (Sutcliffe, 2002; Institute of Electrical and Electronics Engineers, 2011). This ambiguity can be caused by contextual details not being made explicit in the requirements narrative and/or by not being representable in a core language model, for instance, due to lacking expressiveness of the respective modeling language.

Adding to these issues, the core language model is typically created in several iterations and is therefore not a static artifact. Changes of the domain requirements trigger the evolution of the core language model (Wimmer et al., 2010). Requirements can change, for instance, due to additional functionality, a modified legal situation in the corresponding application domain, or the refactoring of software systems.

While the core language model is the primary artifact of a DSML, there are other DSML artifacts (such as model transformations or model constraints) which directly depend on the core language model (Strembeck and Zdun, 2009; Hoisl et al., 2013). An undetected requirement violation in the core language model may have severe effects on all dependent software artifacts.

The creation of a DSML and its core language model involves DSML engineers and domain experts. Here, a *domain expert* is a professional in a particular domain, such as a stock analyst in the investment banking domain or a physician in the health-care domain. While the domain expert provides the domain-specific knowledge, the *DSML engineer* is responsible for the domain model specification and implementation (e.g., the creation of model artifacts, their attributes, relationships). The challenge in the phase of domain analysis and modeling is the establishment of a common body of knowledge for both, the domain expert, and the DSML engineer (i.e., a shared vocabulary) and the correct abstraction and mapping of domain knowledge to a target modeling language.

A testing technique for evolving and heavily coupled language models would offer a means for selecting relevant, requirements-driven tests, to express and to maintain these tests, and to automate the testing procedure. Current testing approaches for metamodels fall short in a number of ways: For modeling-space sampling (Gomez et al., 2012; Merilinna et al., 2008), a sufficiently specified model under test is needed, which is not available in an iterative development and evolution of domain models. The same holds true for metamodel-test models (i.e., simulating a set of valid instance model alternatives), which require the full domain model under test to be specified (Sadilek and Weißleder, 2008; Cicchetti et al., 2011). Metamodel validation approaches (e.g., model-constraint evaluations) employ formal expression languages (e.g., OCL), but do not consider the structure of non-executable requirements specifications (Merilinna and Pärssinen, 2010). Moreover, approaches exist for tracing requirements (Winkler and Pilgrim, 2010) and testing natural-language statements (Gervasi and Nuseibeh, 2002; Yue et al., 2013), but lack an integrated model-driven tool-chain to combine both, domain modeling capabilities and requirements specification/validation.

We propose a testing approach that employs scenarios to describe system behavior, as well as to bridge the gap between informal natural-language requirements on the one hand and formal models (including source code implementations) on the other hand (Sutcliffe, 2002; Jarke et al., 1998; Uchitel et al., 2003). Thus, in our approach, domain requirements can directly serve as a normative specification for the core language model of a DSML. The paper makes the following contributions:

- *Semi-structured domain-requirements language*: We specify a semi-structured requirements language that can be used by domain experts to define scenarios via structured natural language. These natural-language scenario descriptions are then transformed into an executable scenario format. The executable test scenarios check the conformance of (evolving) DSML core language model definitions against the scenario-based requirements specification.
- *Traceable mapping of domain requirements*: In this way, our approach supports a systematic, semi-automated, and traceable mapping of domain requirements documented in natural language to executable test scenarios for requirements validation. The mapping conventions are defined in a reusable form applicable to different scenarios.
- *Participatory requirements validation*: In our approach, the domain expert uses scenarios to define domain requirements (Sutcliffe, 2002; Jarke et al., 1998). This way, the domain expert actively contributes to defining and to validating the DSML core language model, in close cooperation with the DSML engineer.

As a proof-of-concept prototype, we implemented a complete MDD-based tool chain in which the definition of natural-language requirements, scenario-based tests, and DSML model transformations are supported. The prototype builds on top of the Eclipse Modeling Framework (EMF) and the Epsilon language family. It is publicly available at `http://nm.wu.ac.at/modsec`.

The remainder of the paper is structured as follows: Section 2 presents a motivating example of DSML integration. DSML integration exemplifies the reuse of existing artifacts from two or more individual source DSMLs to implement a new DSML (Hoisl et al., 2012). Section 3 introduces our approach for transforming requirements into executable test scenarios. Section 4 specifies a language for requirements-level scenarios, whose applicability—in combination with transformation definitions—is shown in a DSML integration case in Section 5. Section 6 discusses how our approach contributes to address the issues of natural-language requirements testing raised in Section 2. At last, related work is reviewed in Section 7 and Section 8 concludes the paper.

2 MOTIVATING EXAMPLE

When modeling a domain, textual use-case scenarios are commonly employed for documenting domain requirements. Consider the example of *story-driven modeling* (Diethelm et al., 2005). First, use-case narratives are collected textually which are then refined into diagrammatic models. During so-called object-game sessions, the team of developers (e.g., DSML engineers analyzing the domain) draw up sketches of object diagrams cooperatively (e.g., using a whiteboard). These diagrams are then translated into UML collaboration diagrams and grouped into model sequences, the so-called story boards. From these story boards, structural and behavioral specifications (e.g., unit tests) can be derived (see Figure 1).

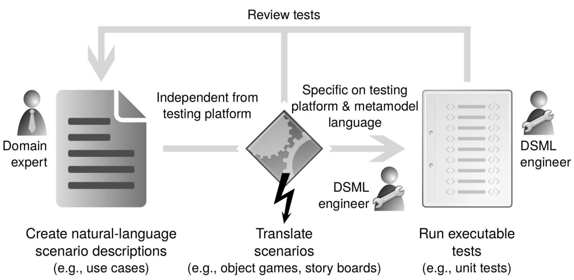

Figure 1: Translating requirements into executable tests.

The transitions between different types of requirement descriptions (e.g., text and collaboration diagrams), as well as between requirement descriptions and executable specifications (e.g., collaboration diagrams and unit tests) provide the opportunity to continuously elicit the requirements, by adding missing or by clarifying ambiguous details. At the same time, however, each transition risks introducing inconsistency between the requirements in their different representations (see also Figure 1). For example, a requirements detail documented in textual form in a use-case description, might be simply omitted accidentally when drawing up the collaboration diagrams. Then, once certain details have been clarified in terms of UML collaboration, a diagrammatically documented requirement might turn out to be conflicting with the early, textually recorded ones. Consequently, each requirements representation must be constantly maintained to reflect changes to other representations.

The risk of inconsistency between multiple requirements representations and the maintenance overhead, as exemplified for story-driven modeling above, motivated us to investigate means of automatically transforming natural-language scenarios at the requirements level into executable test scenarios. Throughout the paper, we will look at the development of a language model from existing ones, the case of DSML integration (Hoisl et al., 2012).

Consider the DSMLs `A` and `B` representing two technical domains: system auditing and distributed state-transition systems (see Figures 2 and 3). The integrated DSML `C` should cover a new and an integrated domain (i.e., auditable distributed state-transition systems). This example is taken from an integration case which is described in full detail in (Hoisl et al., 2013).

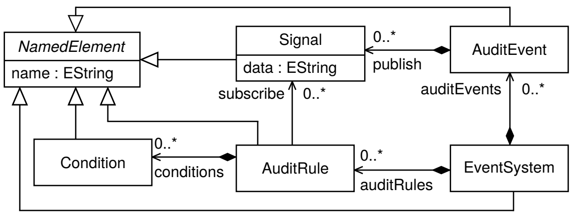

Figure 2: Auditing event-based systems (DSML `A`).

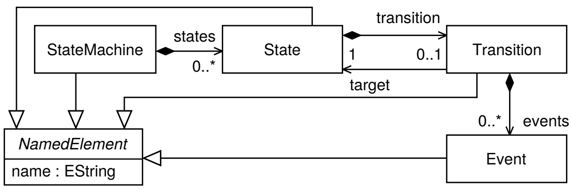

Figure 3: State/transitional behavioral system (DSML `B`).

The integration of the core language models of the DSMLs `A` and `B` is achieved by turning `Events` propagated in a distributed system into AuditableEvents that can be tracked for auditing purposes (e.g., through a corresponding system-monitoring facility). Table 1 shows an excerpt of a natural-language scenario description for the integrated DSML C which resulted from eliciting domain requirements during a domain analysis (see also Figure 1). The description is structured according to a one-column table format as suggested by (Cockburn, 2001). From the description of *test scenario 1* follows that the domain requires all events to be audited and each audited event shall issue a signal to the monitoring facility (see Table 1).

Table 1: Example requirements-level natural-language scenario description (excerpt).

Test case 1	Ascertain that each triggered AuditableEvent can be sensed by the monitoring facility.
Primary actors	System auditor, distributed-systems operator
Preconditions	All metamodel constraints for the source DSMLs shall hold for DSML C.
Trigger/Setup	The model-transformation workflow to integrate the metamodels of DSML A & DSML B is executed.
Test scenario 1	An AuditableEvent issued by a Transition shall publish at least one Signal.
Preconditions	AuditableEvent has all structural features of AuditEvent and Event.
Expected result	Instances of AuditableEvent shall refer to at least one Signal instance.

From this natural-language requirements description, a DSML engineer then derives concrete composition steps which are performed manually or via model-to-model transformations (Czarnecki and Helsen, 2006). The resulting relevant core language model fragment of the merged DSML C is shown in Figure 4.

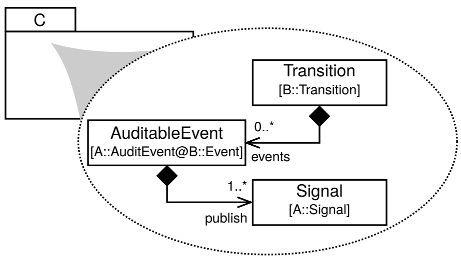

Figure 4: Excerpt from merged DSML C core language model.

When turning non-executable requirements descriptions into executable specifications (e.g., software tests, transformation definitions; see also Figure 1), evaluating the natural-language requirements against these evolving software artifacts poses important challenges (Sutcliffe, 2002; Institute of Electrical and Electronics Engineers, 2011).

Ambiguity of Requirements: In contrast to formal specifications, natural language is prone to misinterpretation (Sutcliffe, 2002). The ambiguity of natural-language statements may lead to erroneous or incomplete requirement implementations. For instance, the definition of the scenario trigger in Table 1 can lead to misinterpretations. It is ambiguous whether the scenario is meant to be enacted either at the beginning, at the end, or anywhere in between the integration of the two core language models.

Consistency of requirements: A DSML core language model may have to comply to a number of different requirements. When defining scenarios in natural language, it is difficult to check that the requirements are free of conflicts. Natural language allows for expressing identical requirements differently, for instance, the expected result from test scenario 1 in Table 1 can be rephrased: *The first structural feature of the metaclass AuditableEvent must never have a lower bound of zero.*

Singularity of Requirements: To allow for an easy to understand and testable requirements specification, a requirement statement should ideally include only one requirement with no use of conjunctions. For example, the test case 1 precondition from Table 1 demands that the constraints for the composed DSML C reference back to the individual constraint sets of DSMLs A and B. This backward-dependent relationship adds to the complexity of the requirements validation (i.e., constraints have to be checked twice for the two source core language models and a third time when applied in the context of the composed DSML C).

Traceability of Requirements: Requirements should be forward traceable (e.g., to DSML core language model source code artifacts; in our example the implemented metamodels of Figure 2 and Figure 3) and backward traceable (e.g., to specific stakeholder statements a requirement originates from; in our example the requirements described in Table 1). That is, all forward and backward relationships for a requirement are identified and recorded (Institute of Electrical and Electronics Engineers, 2011). This way, a requirement can be navigated from its source (e.g., the expected result in Table 1) to its implementation (e.g., a corresponding assertion statement in an automatable test specification); and vice versa. Tracing of natural-language requirements to their respective DSML artifact implementations (and vice versa) is non-trivial because of different abstraction levels and specification formats, for instance, natural language vs. source code (Marcus et al., 2005).

Validation of requirements: Requirements have to be validated in order to prove that they are satisfied by a corresponding DSML core language model implementation. Acceptance testing is a common method for the validation of software systems requirements (Institute of Electrical and Electronics Engineers, 2011). Nevertheless, tool-supported and automated testing of natural-language statements is difficult due to their ambiguity and the lack of a formal structure. Without adequate tool support, the validation of requirements from Table 1 in the context of the merged DSML `C` core language model (Figure 4) can only be done manually—a tedious and error-prone task whose complexity increases with the amount of involved metamodel elements, transformation statements, and requirement specifications.

These challenges increase with the number of requirements and scenario descriptions to be satisfied, the transformation rules involved, and the DSML artifacts created.

3 FROM REQUIREMENTS TO EXECUTABLE SCENARIO TESTS

Our approach of mapping natural-language requirements to executable scenario descriptions is sketched in Figure 5. In this process, the primary actors are the domain expert and the DSML engineer. In certain domains (e.g., software testing), both roles can be taken by one subject at the same time. First, the domain expert and the DSML engineer must agree on a requirements specification format (Sutcliffe, 2002). This is to assure that the requirements are captured in a format which can be further processed. For the specification of system behavior on the requirements level, natural-language scenarios are a suitable choice (Sutcliffe, 2002; Jarke et al., 1998; Uchitel et al., 2003). Scenarios can help to reduce the risk of omitting or forgetting relevant test cases, as well as the risk of describing important tests insufficiently. A requirements-level scenario description establishes the conditions under which it runs (Cockburn, 2001; Strembeck, 2011): A trigger corresponds to the event which sets off the scenario. Preconditions announce the system state expected by the use case before starting. The objective of the scenario defines the goal which should be achieved. Important persons involved are named as primary actors. Finally, a set of validation or action steps specifies the scenario's expected outcome. Examples of two semi-structured natural-language scenario descriptions on the requirements level are provided in Table 1 and in Listing 3.

Based on the scenario descriptions of the domain

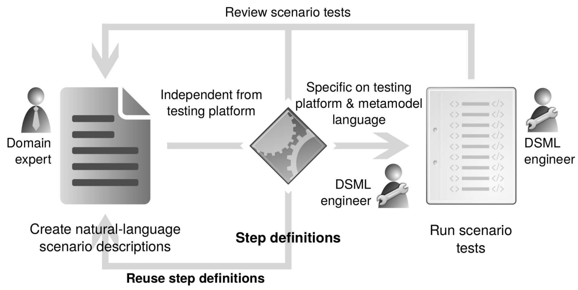

Figure 5: Transforming requirements into executable test scenarios via step definitions.

expert, the DSML engineer has to translate these scenarios into executable tests (see Figure 5). Executable tests allow for the automatic validation of DSML core language models against requirement-level scenarios. If each requirement-level scenario is checked via one (ore more) executable test scenario(s), a critical test coverage of the most relevant requirements can be achieved. The scenario tests are reviewed by the domain expert and the DSML engineer. This ensures that the executable scenario descriptions reflect the requirements sufficiently.

Each natural-language description of a scenario is called a *step*. In order to translate natural-language requirements into executable tests, for each scenario-step a corresponding *step definition* (Wynne and Hellesøy, 2012) must be defined (see Figure 5; an example is shown in Listing 4). We transform natural-language requirements into executable test scenarios via linguistic rule-based step definitions (Winkler and Pilgrim, 2010). A rule-based step definition deduces traces by applying rules to steps (in contrast to, e.g., approaches based on information retrieval). Linguistic rules overcome the limitation of structural approaches by extending the analysis of a step's structure to the analysis of its language (via natural-language processing techniques). In doing so, step definitions serve as the connecting link between the domain expert's vocabulary and the DSML engineer's vocabulary.

After the step definitions have been defined, the DSML engineer performs the domain modeling activity; i.e., the DSML engineer constructs (fragments of) the DSML core language model in a way that the design decisions comply with the requirement specifications. The test scenarios are then checked against the core language model (fragments). If all tests succeed, the core language model (part) adheres to the requirements. If a test fails, the domain expert and the DSML engineer review the respective test scenarios for their validity and iterate over the core language modeling artifacts.

This process facilitates the step-wise refinement of requirements as well as an iterative development of DSML core language models. At first, a requirement specification may not capture all needs for the whole DSML core language model (Sutcliffe, 2002). Nevertheless, by performing domain modeling actions, the DSML engineer constructs a requirements-conforming DSML core language model fragment. As the requirements evolve, so does the DSML core language model, for example, when integrating two DSMLs (Hoisl et al., 2012). This test-driven method of step-wise development and refinement of DSML core language models helps detect requirements violations at an early stage. Furthermore, step definition patterns are designed for reuse (e.g., in other core language modeling scenarios; see Figure 5).

4 A LANGUAGE FOR REQUIREMENTS-LEVEL SCENARIOS

To specify natural-language requirements via scenarios, we define a model-based scenario-description language. Using this language, the domain expert can express domain requirements on DSML core language models via semi-structured natural-language scenarios. At the same time, each scenario description can so be represented as a well-defined model to facilitate further-processing of the scenario description. Figure 6 shows the metamodel of our scenario-description language. The main concepts and concept relationships were identified by studying related work on requirements metamodels (Goknil et al., 2008; Somé, 2009) and acceptance testing (Wynne and Hellesøy, 2012).

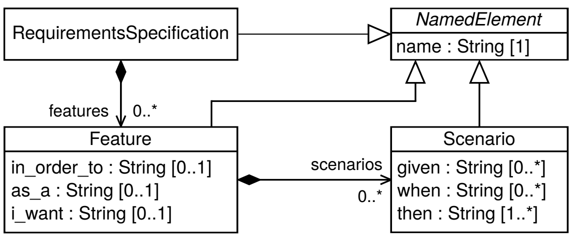

Figure 6: Scenario-based requirements specification language metamodel.

Abstract syntax: Requirements are specified in terms of characteristic functionality (`Features`) the stakeholders request and the DSML shall implement. A `Feature` is described textually via four properties: A `Feature` has a `name`, is specified in order to meet a certain goal (`in_order_to`), defines participating stakeholders (`as_a`), and describes the feature's purpose (`i_want`). A structural `Feature` of core language models may describe relationships of domain concepts or metaclass properties (e.g., inheritance relationships, metaclass attributes).

`Scenarios` describe important action and event sequences characteristic to a given `Feature`. We use `Scenarios` to determine the details of language model compositions. In addition to its `name`, a `Scenario` is associated with one or more conditions that trigger the scenario (`given`), define when alternative paths are chosen (`when`), and specify expected outcomes (`then`). A `RequirementsSpecification` has a `name` and documents as many `Features` as requested and a `Feature` consists of as many `Scenario` descriptions as needed.

Concrete syntax: Following related approaches to textual use-case modeling and acceptance testing (Wynne and Hellesøy, 2012; Goknil et al., 2008; Somé, 2009), we provide a textual concrete-syntax for the domain user of the requirements language. In this way, the domain expert is able to define scenario-based requirements via natural-language statements (an example is shown in Listing 1). The syntax rules allow for using synonyms for steps (e.g., `And`, `But`; see Listing 2). This way, the domain expert can, on the one hand, phrase requirements in a natural and readable way and, on the other hand, concatenate multiple steps into composite statements (i.e., adding multiple steps to each `Given`, `When`, or `Then` section). This allows the domain expert to define scenarios which are expressed over multiple, interrelated models.

Listing 1: A textual concrete syntax.

```
RequirementsSpecification: "..."
  Feature: "..."
    In order to "..."
    As a "..."
    I want "..."
    Scenario: "..."
      Given "..."
      When "..."
      Then "..."
```

In our proof-of-concept implementation, this textual concrete syntax is specified using an Eclipse Xtext grammar (see Listing 2). The style of the textual concrete syntax can easily be adjusted to the needs of a particular domain—we aligned our exemplary grammar definition with (Wynne and Hellesøy, 2012). Also note that this textual concrete syntax is just one option to define instance models of the requirements metamodel (Figure 6). Viable alternatives can be realized for our implementation with small effort, including tree-based views (e.g., via the Sample Reflective Ecore Model Editor), diagrammatic views (e.g., via the Eclipse Graphical Modeling Framework), or further textual views (e.g., via XML). In this way, our implementation allows, on the one hand, that the requirements specification document can be tailored to the best suitable format for the domain and, on the other hand, that formats and views can be switched interchangeably.

Listing 2: Xtext grammar definition for semi-structured scenario-based requirements.

```
grammar at.ac.wu.nm.dsml.sbt.SRL with org.eclipse.xtext.
      common.Terminals

import "http://requirementsspecification/0.1"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

RequirementsSpecification returns RequirementsSpecification
      :
  {RequirementsSpecification}
  'RequirementsSpecification:' name=EString
    (features+=Feature)*
  ;

Feature returns Feature:
  {Feature}
  'Feature:' name=EString
    ('In order to' in_order_to=EString)?
    ('As a' as_a=EString)?
    ('I want' i_want=EString)?
    (scenarios+=Scenario)*
  ;

Scenario returns Scenario:
  {Scenario}
  'Scenario:' name=EString
    ('Given' given+=EString
    (('Given'|'And'|'But') given+=EString)*)?
    ('When' when+=EString
    (('When'|'And'|'But') when+=EString)*)?
    'Then' then+=EString
    (('Then'|'And'|'But') then+=EString)*
  ;

EString returns ecore::EString:
  STRING | ID;
```

Platform integration: The requirements metamodel as shown in Figure 6 and the corresponding tool support provide the means to define non-executable, requirements-level scenarios which are independent from any metamodeling infrastructure (Ecore, MOF) and from a test-execution framework. To validate core language models against these descriptions in a systematic and automated manner, they must be transformed into executable test cases by the DSML engineer using step definitions. Scenario-based testing approaches (Strembeck, 2011; Sobernig et al., 2013) provide the necessary abstractions (e.g., test cases, test scenarios, pre- and post-conditions, setup and cleanup sequences) to represent scenario descriptions directly as executable tests. For our proof-of-concept implementation, we use our scenario-based testing framework published in (Sobernig et al., 2013) as test-execution platform. This execution platform for scenario tests is built on top of EMF and extends the Epsilon EUnit testing framework (Kolovos et al., 2013). In the step of platform integration, instance models of the requirements metamodel (see Figure 6) are transformed into executable scenario tests supported by this test framework. Table 2 shows the exemplary correspondences between the metamodel concepts, the scenario-based testing domain concepts (Strembeck, 2011), and the syntactical equivalents as provided by the test-execution platform (Kolovos et al., 2013; Sobernig et al., 2013). Note that integration with alternative validation platforms for the scenario descriptions (e.g., an OCL engine, a model-transformation engine) can be achieved by providing a dedicated set of step definitions. We adopted the scenario-based test framework (Sobernig et al., 2013) because of its matching test abstractions and for demonstration purposes.

Table 2: Correspondences between requirements language, scenario-testing concepts, and EUnit concrete syntax.

Requirements language	**Test concept**	**Epsilon syntax construct**
`RequirementsSpecification`	Test suite	`@TestSuite`
`Feature`	Test case	`@TestCase`
`Scenario`	Test scenario	`@TestScenario`
`Scenario.Given`	Precondition	`$pre`
`Scenario.When`	Test body	*operation's body*
`Scenario.Then`	Expected result	*EUnit assertion*

5 SCENARIO-BASED TESTING EXEMPLIFIED: A DSML INTEGRATION CASE

We demonstrate the scenario-based testing of DSML core language models via a DSML integration case. We build on the motivating example introduced in Section 2 and show how the transformation from natural-language requirements into executable scenario tests is achieved via our software prototype.

To recall, in our integration scenario from Section 2 we want to fully compose the core language models of two DSMLs `A` and `B` (Figures 2 and 3). The resulting merged DSML `C` covers an integrated domain established through the conceptual weaving of `AuditEvents` (from DSML `A`) and `Events` (from DSML `B`) into `AuditableEvents`. The process steps involved in this DSML integration are shown in Figure 7.

To scenario-test the DSML core language model composition, the domain expert and the DSML engineer first determine a requirements specification format (in our case through an Xtext grammar; see Listing 2). After the requirements specification format

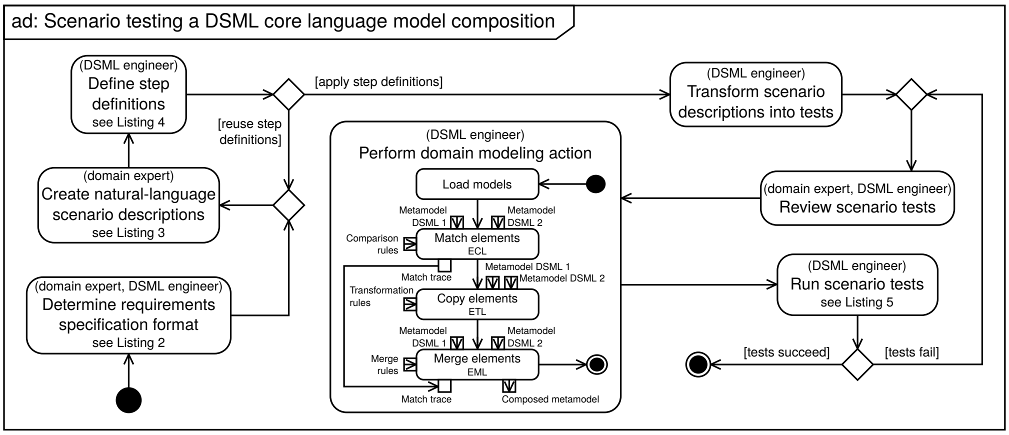

Figure 7: Scenario testing of DSML core language model integration.

has been determined, the domain expert can define natural-language scenarios tackling the domain requirements. Listing 3 shows an excerpt of such a requirements specification conforming to the syntax of our grammar. To support the domain expert in elaborating requirements, Xtext generates an editor supporting, for example, syntax highlighting, auto completion, or error reporting. Due to space limitations, we define a brief requirements-level scenario for only one DSML integration feature (more examples can be obtained from `http://nm.wu.ac.at/modsec`). Listing 3 shows a refinement from the initial Table 1 scenario and requires that all audited events shall issue a signal to the monitoring facility.

Listing 3: Scenario-based semi-structured requirements example.

```
RequirementsSpecification: "DSMLs A and B integration"
  Feature: "Monitor AuditableEvent"
    In order to "ascertain that each triggered
         AuditableEvent can be sensed by the monitoring
         facility"
    As a "system auditor and distributed-systems operator"
    I want "that AuditableEvents shall publish Signals"

    Scenario: "AuditableEvent shall publish at least one
         Signal"
      Given "that EventSystemStateMachine.AuditableEvent
           has all features of EventSystem.AuditEvent and
           StateMachine.Event"
      When "in metamodel EventSystemStateMachine metaclass
           AuditableEvent references metaclass Signal"
      Then "instances of EventSystemStateMachine.
           AuditableEvent shall refer to at least 1 Signal
            instance"
  ...
```

After this, the DSML engineer defines corresponding step definitions for every requirements-level scenario to allow for the translation into executable test scenarios (an example is shown in Listing 4). Our software prototype supports the specification of step definitions by employing token-matching patterns, i.e., string-sequences are recognized via regular expressions—a linguistic rule-based approach (Winkler and Pilgrim, 2010). In addition, a step definition can be composed of a unique and unordered collection of regular expression patterns (see lines 3–5 in Listing 4). This is to aid the DSML engineer in the process of matching steps which are not easily recognizable via a single regular expression statement. In this context, a step definition can match multiple scenario-steps of identical or different types (e.g., steps of types `Given` and `When`).

Listing 4: A scenario-transforming step definition.

```
1 var stepDef : Map = Map {
2   -- tests if multiplicity >= 1 between two classifiers
3   Set {
4     "^instances of (\\S+)\\.(\\S+) (?:shall|must) refer to
          at least (\\d+) (\\S+) instances?$"
5   }
6     = "assertFalse(\"An $1 shall publish at least $2 $3.\",
          $0!EClass.all->selectOne(c | c.name = \"$1\").
        eStructuralFeatures->first().lowerBound < $2);"
7   -- more step definitions
8 };
```

Our software prototype implements an EGL-based (Epsilon Generation Language) transformation from requirements-level scenarios into executable tests deployable in our extended EUnit testing framework (Kolovos et al., 2013; Sobernig et al., 2013). The transformation evaluates step definitions (Listing 4) against the requirements specification (Listing 3). The generated EUnit test scenario resulting from the requirement specification transformation is shown in Listing 5.

Listing 5: Generated EUnit scenario tests.

```
--DSMLs A and B integration
@TestSuite
operation dsmls_a_and_b_integration() {
  --Monitor AuditableEvent:
  --In order to ascertain that each triggered
       AuditableEvent can be sensed by the monitoring
       facility
  --As a system auditor and distributed-systems operator
```

```
--I want that AuditableEvents shall publish Signals
@TestCase
operation monitor_auditableevent() {
  --AuditableEvent shall publish at least one Signal
  @TestScenario
   --Given that EventSystemStateMachine.AuditableEvent
        has all features of EventSystem.AuditEvent and
        StateMachine.Event
  $pre EventSystemStateMachine!EClass.all->selectOne(ae |
        ae.name = "AuditableEvent").eStructuralFeatures.
       size() = EventSystem!EClass.all->selectOne(ae |
       ae.name = "AuditEvent").eStructuralFeatures.size
       () + StateMachine!EClass.all->selectOne(ae | ae.
       name = "Event").eStructuralFeatures.size()
  operation
       auditableevent_shall_publish_at_least_one_signal
       () {
    --When in metamodel EventSystemStateMachine metaclass
          AuditableEvent references metaclass Signal
    if (EventSystemStateMachine!EClass.all->selectOne(c |
          c.name = "AuditableEvent").eStructuralFeatures
         ->first().eType.name = "Signal") {
      --Then instances of EventSystemStateMachine.
           AuditableEvent shall refer to at least 1
           Signal instance
      assertFalse("An AuditableEvent shall publish at
           least 1 Signal.", EventSystemStateMachine!
           EClass.all->selectOne(c | c.name = "
           AuditableEvent").eStructuralFeatures->first()
           .lowerBound < 1);
    }
  }
}
...
}
```

Before the actual domain modeling composition is performed, the domain expert and the DSML engineer collaboratively review the executable test scenarios. This review is facilitated by maintaining the requirement statements along with the corresponding test cases (i.e., established trace links). For our example, the core language model integration is executed via an Epsilon-based workflow (e.g., matching, copying, merging core language model elements; see Figure 7).

Afterwards, the scenario tests (Listing 5) are run against the integrated DSML `C` core language model. If a test fails, the domain expert and the DSML engineer review the corresponding test scenario according to the error message shown via the EUnit reporting console to exclude an erroneous specification. Figure 8 shows a failing test scenario because of non-conforming multiplicity requirements in the integrated DSML `C` core language model. In order to get all scenario tests pass (Figure 9), the DSML engineer needs to patch the composition specification until it fully complies to the specified requirements (details are omitted here for brevity).

6 DISCUSSION

In Section 2, we argue that natural-language scenarios are useful to capture requirements, but have serious drawbacks when it comes to evaluating them against DSML software artifacts. In this section, we discuss how our work contributes to (partly) overcome this shortcomings.

Ambiguity of Requirements: Step definitions provide for the transformation of natural-language requirements into formal specifications. Those formal specifications render ambiguous natural-language requirements explicit. In this sense, DSML engineers play an important role as they serve as the interface of mapping natural-language requirements to executable scenarios. A correct mapping (i.e., accurate step definitions) can only be ensured in close collaboration with the domain expert. Still, the ambiguity of natural-language requirements remains, but formal specifications help to reduce the number of semantic variation points.

Consistency of requirements: Executable scenario tests add to the conflict-free definition of requirements. All test scenarios of a requirements specification document are executed in the same context of one test suite. This ensures, that each scenario is tested against identical language model artifacts. Inconsistent tests fail and are reported back to the DSML engineer. Consequently, the requirements need to be reviewed by the domain expert.

Singularity of Requirements: If a scenario step conjugates other steps of the requirements specification, this can be recognized in a pattern-based step definition approach. Composite steps match more than one step definition pattern which indicates that requirements singularity may not be satisfied.

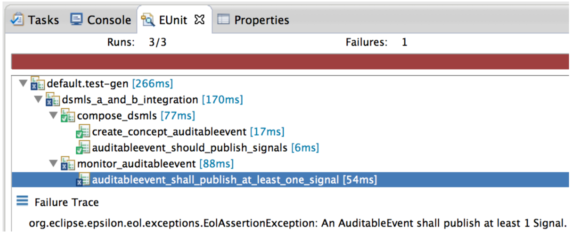

Figure 8: EUnit scenario-test report: one test fails.

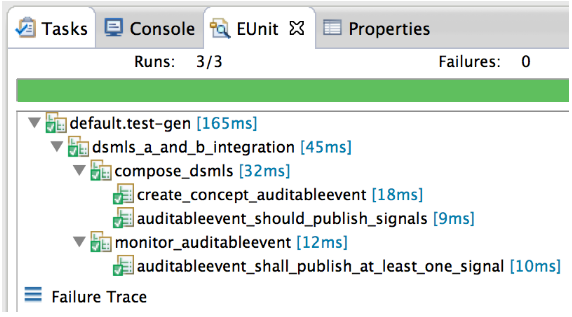

Figure 9: All scenario tests pass.

Traceability of Requirements: In our approach, on the one hand, requirements are forward traceable via transforming step definitions. With this, it is possible to keep track of the requirements-level scenarios to their executable test counterparts. On the other hand, we provide support for the backward traceability of executable test scenarios. In our transformation routines, the natural-language scenario steps are copied as comments besides their corresponding executable test scenarios (see Listing 5). Furthermore, the EUnit reporting console pairs passed/failed tests to their respective executable scenario implementations. This allows to trace the natural-language requirements from the scenario-test report.

Validation of requirements: In this paper, executable scenario tests are employed to collect evidence that proves that the system can satisfy the specified requirements. A test report is generated which shows the conformance status of the DSML core language model against the requirements specification. In this sense, the requirements specification serves as a documentation of the core language model development activities which can be validated. Furthermore, step definitions are an important documentation source as they provide for the generation of validation specifications (i.e., executable test scenarios) from the natural-language requirements.

7 RELATED WORK

Related work falls into three categories: testing of 1a) natural-language requirements and 1b) evolving metamodels, requirements 2a) metamodeling and 2b) traceability, and 3) available tool support.

Testing against natural-language requirements: In order to test natural-language statements, they need to be processed and transformed into an analyzable representation. Several natural-language processing techniques exists—keyword extraction, part of speech tagging etc. (Winkler and Pilgrim, 2010)—for instance, to derive model-based (Santiago Júnior and Vijaykumar, 2012) or functional test cases (Dwarakanath and Sengupta, 2012), to check model properties (Gervasi and Nuseibeh, 2002), and to generate analysis models from textual use cases (Yue et al., 2013). Our approach benefits from these documented experiences on processing natural-language requirements into processable and executable artifacts, in particular linguistic rule-based transformations (Winkler and Pilgrim, 2010). We realize this processing, in contrast to related work, using an integrated, model-driven tool chain. At the same time, testing DSML core language models and their integration has distinct requirements, for example, navigating between different metamodels to capture model transformations. Navigation between metamodels is supported by our approach, for instance, via a mapping of multi-metamodel requirements (e.g., line 8 in Listing 3) into executable test scenarios involving individual and integrated DSML core language models (e.g., precondition on line 13 in Listing 5).

Testing evolving metamodels: We distinguish between three current metamodel-testing approaches to test requirements-conformance objectives for evolving metamodels: 1) modeling-space sampling, 2) metamodel-test models, and 3) metamodel validation. Modeling-space sampling (1) adopts techniques of model-based testing, testing of model transformations, and model simulation to generate a sample of potential metamodel test instantiations (Gomez et al., 2012; Merilinna et al., 2008). Such a sample is produced in an automated manner by traversing the metamodel and creating metamodel instances according to the metamodel specification and pre-defined sampling criteria. Metamodel-test models (2) aim at the manual definition of potential metamodel instantiations by domain experts and DSML engineers. Such a procedure requires a generic, proxy metamodel from which the test models are instantiated. Metamodel validation approaches (3) employ model-constraint expressions (e.g., specified via the OCL) to express test cases on metamodels (e.g., specified as invariants), defined at the level of the corresponding meta-metamodel (Merilinna and Pärssinen, 2010). Our approach primarily extends metamodel validation techniques to provide an explicit scenario abstraction, both at the requirements and the testing level. Individual steps (e.g., `Given`) are transformed, for instance, into constraint-expressed preconditions evaluated over the metamodels under test.

Requirements metamodels: The requirements metamodel defined in Section 4 could be extended by integrating it with closely related metamodels (Goknil et al., 2008; Somé, 2009). A consolidated requirements metamodel using the proposal by (Goknil et al., 2008) would benefit from additional concepts such as requirements relations, status, and priority. In addition, there is a first SysML integration available for the metamodel in (Goknil et al., 2008). The requirements metamodel in (Somé, 2009) would allow for an alignment with UML-compliant use cases and a corresponding, alternative textual concrete syntax.

Requirements traceability: Requirements are traced, for example, to prove system adequateness, to validate artifacts, or to test a system (Winkler and Pilgrim, 2010). Using traceability links in MDD has its purpose, for instance, in supporting design decisions, in managing artifacts' dependencies, or in validating requirements via end-to-end traceability of MDD processes (Winkler and Pilgrim, 2010). Recent approaches (e.g., based on structural rules or information retrieval techniques) try to overcome the issues of tracing natural-language requirements (as discussed in Section 2). We contribute to the field of requirements traceability via a linguistic rule-based approach for testing DSML core language models (and their integration) and provide for accompanying tool support.

Tool support: Acceptance test approaches provide tool support for specifying executable test cases in the domain expert's language (Wynne and Hellesøy, 2012; Mugridge and Cunningham, 2005). These test cases are commonly defined via structured text or table formats. While our approach shares these design decisions, we built our proof-of-concept implementation on top of an existing model-management toolkit and a previously developed, general-purpose unit- and scenario-testing framework (Sobernig et al., 2013) to reuse their model management capabilities.

8 CONCLUSION

In this paper, we presented a linguistic, rule-based approach for a traceable translation of semi-structured natural-language requirements into executable test scenarios. This is motivated by the observed need to foster the cooperation of the domain expert and the language engineer when refining and validating the core language models, i.e., the abstract syntaxes of domain-specific modeling languages (DSMLs), iteratively. We exemplified the usage of our approach by presenting a case for the scenario-based testing of DSML integration. The feasibility of our approach is demonstrated via a dedicated software prototype. We discussed how our work can help to cope with problems that emerge when validating DSMLs against their requirements recorded in natural language.

A benefit of our work is its design for reuse (see also Figures 5 and 7). Step definitions provide a mapping convention for translating natural-language requirements into executable test scenarios. These mapping conventions are separated from the transformation routines. In order to provide for further scenario-based DSML core language model tests, the transformation routines do not change (as they are only dependent on the requirements specification language). The linguistic patterns as part of the step definitions can be reused, as well.

In future work, we plan to extend the requirements specification language (e.g., scenario outlines, nested steps), in particular to cover (iterative) metamodel development which differs from the coupled metamodel evolution under DSML integration. In addition, we will establish a repository of step definitions for testing DSML core language models and their integration.

ACKNOWLEDGEMENTS

This work has partly been funded by the Austrian Research Promotion Agency (FFG) of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) through the Competence Centers for Excellent Technologies (COMET K1) initiative and the FIT-IT program.

REFERENCES

Cicchetti, A., Ruscio, D. D., Kolovos, D. S., and Pierantonio, A. (2011). A test-driven approach for metamodel development. In *Emerging Tech. for the Evolution and Maintenance of Softw. Models*, pages 319–342. IGI Global.

Cockburn, A. (2001). *Writing Effective Use Cases*. Addison-Wesley.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley Longman Publishing Co., Inc., 6th edition.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

Diethelm, I., Geiger, L., and Zündorf, A. (2005). Applying story driven modeling to the Paderborn shuttle system case study. In *Proc. Int. Conf. Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 109–133. Springer.

Dwarakanath, A. and Sengupta, S. (2012). Litmus: Generation of test cases from functional requirements in natural language. In *Proc. 17th Int. Conf. Applications of Natural Language Processing and Information Systems*, pages 58–69. Springer.

Gervasi, V. and Nuseibeh, B. (2002). Lightweight validation of natural language requirements. *SP&E*, 32(2):113–133.

Goknil, A., Kurtev, I., and Berg, K. (2008). A metamodeling approach for reasoning about requirements. In *Proc. 4th European Conf. Model Driven Architecture: Foundations and Applications*, volume 5095 of *LNCS*, pages 310–325. Springer.

Gomez, J. J. C., Baudry, B., and Sahraoui, H. (2012). Searching the boundaries of a modeling space to test metamodels. In *Proc. 5th IEEE Int. Conf. Softw. Testing, Verification and Validation*, pages 131–140. IEEE.

Hoisl, B., Sobernig, S., and Strembeck, M. (2013). Higher-order rewriting of model-to-text templates for integrating domain-specific modeling languages. In *Proc. 1st Int. Conf. Model-Driven Eng. and Softw. Dev.*, pages 49–61. SciTePress.

Hoisl, B., Strembeck, M., and Sobernig, S. (2012). Towards a systematic integration of MOF/UML-based domain-specific modeling languages. In *Proc. 16th IASTED Int. Conf. on Softw. Eng. and Applications*, pages 337–344. ACTA Press.

Institute of Electrical and Electronics Engineers (2011). Systems and software engineering – life cycle processes – requirements engineering. Available at: `http://standards.ieee.org/findstds/standard/29148-2011.html`. ISO/IEC/IEEE 29148:2011.

Jarke, M., Bui, X. T., and Carroll, J. M. (1998). Scenario management: An interdisciplinary approach. *Requ. Eng.*, 3(3-4):155–173.

Kolovos, D., Rose, L., García-Domínguez, A., and Paige, R. (2013). The Epsilon book. Available at: `http://www.eclipse.org/epsilon/doc/book/`.

Lisboa, L. B., Garcia, V. C., Lucrédio, D., de Almeida, E. S., de Lemos Meira, S. R., and de Mattos Fortes, R. P. (2010). A systematic review of domain analysis tools. *Inform. Softw. Tech.*, 52(1):1–13.

Marcus, A., Maletic, J. I., and Sergeyev, A. (2005). Recovery of traceability links between software documentation and source code. *Int. J. Softw. Eng. and Knowledge Eng.*, 15(5):811–836.

Merilinna, J. and Pärssinen, J. (2010). Verification and validation in the context of domain-specific modelling. In *Proc. 10th Workshop Domain-Specific Modeling*, pages 9:1–9:6. ACM.

Merilinna, J., Puolitaival, O.-P., and Pärssinen, J. (2008). Towards model-based testing of domain-specific modelling languages. In *Proc. 8th Workshop Domain-Specific Modeling*, pages 39–44.

Mugridge, R. and Cunningham, W. (2005). *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall.

Neill, C. J. and Laplante, P. A. (2003). Requirements engineering: The state of the practice. *IEEE Softw.*, 20(6):40–45.

Object Management Group (2011). OMG Unified Modeling Language (OMG UML), Superstructure. Available at: `http://www.omg.org/spec/UML`. Version 2.4.1, formal/2011-08-06.

Object Management Group (2013). OMG Meta Object Facility (MOF) Core Specification. Available at: `http://www.omg.org/spec/MOF`. Version 2.4.1, formal/2013-06-01.

Sadilek, D. A. and Weißleder, S. (2008). Testing metamodels. In *Proc. 4th European Conf. Model Driven Architecture: Foundations and Applications*, volume 5095 of *LNCS*, pages 294–309. Springer.

Santiago Júnior, V. A. D. and Vijaykumar, N. L. (2012). Generating model-based test cases from natural language requirements for space application software. *Softw. Quality Control*, 20(1):77–143.

Sendall, S. and Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45.

Sobernig, S., Hoisl, B., and Strembeck, M. (2013). Requirements-driven testing of domain-specific core language models using scenarios. In *Proc. 13th Int. Conf. Quality Softw.*, pages 163–172. IEEE Computer Society.

Somé, S. (2009). A meta-model for textual use case description. *JOT*, 8(7):87–106.

Stahl, T. and Völter, M. (2006). *Model-Driven Software Development*. John Wiley & Sons.

Strembeck, M. (2011). Testing policy-based systems with scenarios. In *Proc. 10th IASTED Int. Conf. Softw. Eng.*, pages 64–71. ACTA Press.

Strembeck, M. and Zdun, U. (2009). An approach for the systematic development of domain-specific languages. *SP&E*, 39(15):1253–1292.

Sutcliffe, A. (2002). *User-Centred Requirements Engineering: Theory and Practice*. Springer.

Uchitel, S., Kramer, J., and Magee, J. (2003). Synthesis of behavioral models from scenarios. *IEEE Trans. Softw. Eng.*, 29(2):99–115.

Wimmer, M., Kusel, A., Schönböck, J., Retschitzegger, W., Schwinger, W., and Kappel, G. (2010). On using inplace transformations for model co-evolution. In *Proc. 2nd Int. Workshop Model Transformation with ATL*, volume 711, pages 65–78. CEUR Workshop Proceedings.

Winkler, S. and Pilgrim, J. (2010). A survey of traceability in requirements engineering and model-driven development. *SoSyM*, 9(4):529–565.

Wynne, M. and Hellesøy, A. (2012). *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Programmers.

Yue, T., Briand, L. C., and Labiche, Y. (2013). Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Trans. Softw. Eng. and Methodology*, 22(1):5:1–5:38.