Requirements-driven Testing of Domain-specific Core Language Models using Scenarios

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

***Abstract*—In this paper, we present an approach for the scenario-based testing of the core language models of domain-specific modeling languages (DSML). The *core language model* is a crucial artifact in DSML development, because it captures all relevant domain abstractions and specifies the relations between these abstractions. In software engineering, scenarios are used to explore and to define (actual or intended) system behavior as well as to specify user requirements. The different steps in a requirements-level scenario can then be refined through detailed scenarios. In our approach, we use scenarios as a primary design artifact. Non-executable, human-understandable scenario descriptions can be refined into executable test scenarios. To demonstrate the applicability of our approach, we implemented a scenario-based testing framework based on the Eclipse Modeling Framework (EMF) and the Epsilon model-management toolkit.**

***Keywords*—Domain-specific modeling, scenario-based testing, language engineering, metamodel testing**

I. Introduction

In model-driven software development (see, e.g., [1]–[3]), a domain-specific modeling language (DSML) is a tailor-made software language for a specific problem domain. DSMLs are used as an abstraction and communication layer targeting software engineers and domain experts. Here, a *domain expert* is a human user who 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. DSMLs are built so that domain experts can understand and phrase domain-specific statements that can be processed by an information system. Thus, DSMLs aim at increasing the number of people who can actively participate in the specification, configuration, and management of software-based systems (see, e.g., [2]). However, in order to realize the benefits of DSMLs, we must ensure that the DSML is correctly implemented and behaves as specified. Moreover, because DSMLs evolve over time (see, e.g., [4]), we must be able to efficiently test the evolving language artifacts such as the core language model.

The DSML *core language model* captures all relevant domain abstractions and specifies the relations between these abstractions (see, e.g., [5]). Changes of the core language model often result from the iterative and collaborative DSML development process. Another example for DSML changes is the integration of two or more DSMLs (and their core language models) into a new DSML. In such an integration procedure, the derived DSML remains dependent on the source DSMLs (e.g., in terms of model transformations) because they represent system viewpoints or optional domain features such as security concerns (see [6] for some background).

In principle, any change to the core language model may result in defects. Because the core language model is a central DSML artifact and because many other artifacts depend on the core language model (such as model-transformation definitions or model constraints; see, e.g., [4], [6]) an undetected error in the core language model may have severe effects on all corresponding software artifacts. As many of such dependent artifacts are created late in the DSML development process (e.g., during platform integration, see, e.g., [5]), the cost-escalation factor of such defects can be considered significant.

In this context, the core language model of a DSML is defined as a metamodel compliant with, e.g., the Meta Object Facility (MOF) or Ecore. Recently, metamodel-testing approaches have been presented (see, e.g., [7]–[9]) to assist in the systematic development of DSMLs and to minimize the risk of late or post-release defects. While such approaches cover important metamodel-testing tasks, they fall short with respect to providing a testing procedure for evolving metamodels. Most importantly, existing approaches consider a metamodel as a given artifact from (and for) which instance models, test models, or test oracles are generated (and provided). Therefore, existing approaches usually fail in making changed metamodels testable against unchanged domain requirements.

In the context of software (systems) engineering, we often find the situation that requirements as well as corresponding solutions are best defined at a human-understandable, non-executable level of abstraction. In contrast to that, the software-based solution is designed and implemented at an executable level, using frameworks and programming languages. This results in a semantic gap between the human-level requirements and solution descriptions on the one hand, and the technical platform that is used to implement the respective software services on the other. The wider this semantic gap, the more difficult is the task to correctly specify and implement a system that behaves as desired by its human users. Scenarios are a natural means to describe (intended) system behavior both as a structured textual requirements definition (see, e.g., [10]) and as a source-code implementation for a software test (see, e.g., [11]–[13]).

To complement existing testing processes, we propose a scenario-driven metamodel-testing approach. In our work, scenarios (see, e.g, [11], [12], [14], [15]) are used to define domain requirements. The initial scenario descriptions can be defined at an abstract level and are specified by (or in collaboration with) domain experts (e.g., via structured text descriptions or UML use case diagrams). In a subsequent step, the requirements-level scenarios are refined and serve as input for the derivation of executable scenario test scripts which closely resemble the narrative structure of the scenarios at the requirements level. The executable scenario specifications are then used to test the evolving core language model for compliance with the corresponding domain requirements. In our approach, the specification and execution of the scenario-based tests of core language models are supported by a testing framework based on the Eclipse Modeling Framework (EMF) and the Epsilon model-management toolkit. The benefits of our approach are three-fold: First, it provides support for testing of changing core language models in different phases of a DSML life-cycle. Second, it facilitates the early establishment of an initial and requirements-based test library. Third, the executable scenario scripts provide an executable documentation of critical application scenarios.

The remainder is structured as follows: In Section II, we provide an overview of the drivers for language-model evolution and a synthesis of metamodel-testing approaches. Against the background on scenario-based testing in Section III, we lay out the notion of scenario-based test procedures and present the design of our prototypical testing framework in Section IV. Subsequently, we demonstrate our approach and prototype via a DSML integration example (see Section V). Finally, we discuss related work in Section VI and provide a concluding outlook in Section VII.

II. Testing Evolving Core Language Models

The language model of a DSML consists of a core language model to define its abstract syntax, constraint specifications to define additional static semantics, and behavior specifications for dynamic semantics (see, e.g., [5]). In DSML development, the core language model is defined as a metamodel which is specified using a metamodeling language (such as MOF or EMF Ecore). A domain engineer derives the metamodel from domain requirements established during a domain analysis and from the corresponding requirements artifacts (e.g., a variability model, a mockup language, or an existing system implementation). In the following, such a core language model is referred to as the Metamodel Under Test (MUT). The MUT is subject to continued change to maintain the high coupling between the DSML and the corresponding application domain (see, e.g., [4]). As a result, models can be instances of the changed MUT (MUT' in Fig. 1) and violate the DSML's domain requirements. Such requirement violations can result, for example, from both under-constraining and over-constraining an MUT (e.g., by tightening or loosening multiplicity constraints). At the same time, new domain requirements may contradict pre-existing requirements (requirements inconsistency, see, e.g., [16]).

Additionally, the *step-wise and iterative development* of a DSML language model ([4], [5]) may also result in requirements violations. Each development phase (e.g., the definition

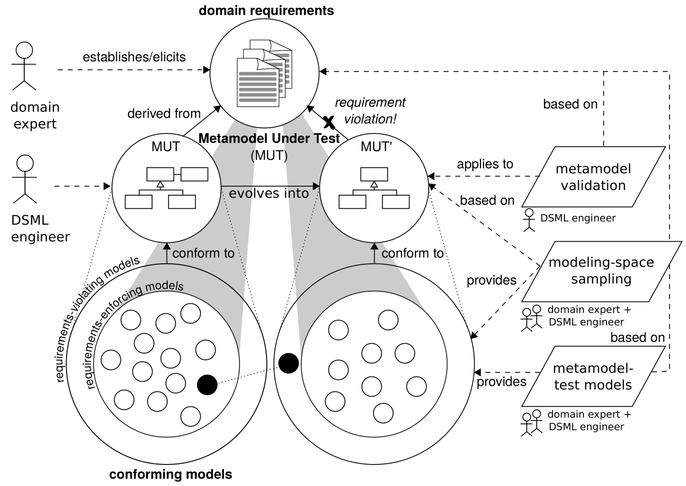

Fig. 1. Metamodel evolution, requirements violation, and metamodel testing.

of language-model constraints or behavior specifications) can require changes to an initially constructed MUT. The multi-phase procedure is often performed repeatedly, for example, in case of changing requirements or when applying a perfective metamodel refactoring by following metamodeling best-practices (see, e.g., [17]). Likewise, the phase of defining the MUT itself is also split into smaller, interrelated, and non-trivial working steps (see [5], [18], [19]), for example: 1) identifying the domain concepts and, subsequently, their relationships using canonical naming schemes; 2) mapping the domain onto metamodeling-language constructs including the concept-internal design (e.g., metaclass properties), concept partitioning (e.g., packaging, namespaces), or concept refactoring via auxiliary concepts (e.g., abstract metaclasses).

DSML integration enables the reuse of DSMLs by composing two or more languages into an integrated DSML to implement a new domain or to integrate domain viewpoints (see, e.g., [18], [20]). Integration should apply to all parts of the language model (e.g., core language model, model constraints; see [21]) as well as model transformations (see, e.g., [6]). When reusing and integrating DSML concepts to meet domain requirements, the DSML engineer must address syntactic and semantic mismatches between the source core language models (the core language models of the DSMLs that are to be integrated) that may cause conceptual defects in the target model (the core language model of the new, integrated DSML). In a coupled DSML integration, we first identify candidate concepts in the source DSML model(s), and then define links between the corresponding concepts in the source and target DSML models, for instance to propagate changes either way. If those links between source and target DSMLs are defined via model transformations, inter-model inconsistencies can easily emerge because of subtle changes to transformation definitions. Moreover, after integrating two DSMLs a domain expert must (re)validate the reused DSML concepts according to the source-domain and target-domain requirements. Finally, a DSML integration procedure (see [21]) resembles the characteristics of DSML development (several iterations, multiple steps per iteration; see above).

Testing metamodels that define the abstract syntax of DSMLs has a number of objectives (see, e.g., [22]). For a

DSML, the *requirements conformance* of the corresponding metamodel is critical. This conformance relation, however, can only be verified by the domain experts (see [7], [22]). Another important testing objective is assessing the *specification consistency* of the interrelated metamodel specifications, for example, consisting of a meta-metamodel instantiation and metamodel constraints expressed using a constraint language such as the Object Constraint Language (OCL) or the Epsilon Validation Language (EVL). An exemplary consistency defect is the risk of contradicting constraint expressions, such as conflicting invariant expressions in boundary cases (see [9]). An inconsistency defect, however, may also hint at *requirements inconsistencies* (see [16]). In the remainder, we concentrate on the requirements-conformance objective for evolving MUTs. Current metamodel-testing approaches address conformance checking differently (see also Fig. 1) and exhibit limitations concerning MUT evolution:

Modeling-space sampling: These approaches (see, e.g., [7], [8]) adopt techniques of model-based testing, testing of model transformations, and model simulation (see [23]) to generate a sample of potential MUT instantiations. 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 (e.g., coverage in terms of metamodel fragments, dissimilarity, boundary cases, custom structural constraints) to find both minimal and representative sets of instances. To verify the conformance relation, the generated models are then reviewed by the domain experts (see [7]) or processed via platform-specific, application-level input/output data to be tested against corresponding applications (see [8]). A first shortcoming with respect to evolving MUTs is the requirement of an existing and sufficiently specified MUT (see [9]) to generate potential instances. This requirement is not always met in the step-wise development of DSML models. A second barrier is that the derived models, at the time of model generation, cannot be considered requirements-conforming anymore. The sampling procedure operates on the changed MUT and so risks presenting the domain expert with a non-representative sample for review. Third, the sampling procedures (see [7]) have to be calibrated for an MUT to obtain both representative and manageable samples. Finally, there is the risk of perceptual misjudgements by the reviewers, for example, due to relatively large sample sizes or a high similarity between sampled models. Tool support for manual reviews is lacking, as well (see [24]).

Metamodel-test models: A second research direction (see, e.g., [9], [25]) is based on ideas from model simulation (see [26]) and aims at the manual definition of potential MUT instantiations by domain experts and DSML engineers (see Fig. 1). Such a procedure requires a generic, proxy metamodel from which the test models are instantiated. In practice, most often custom defined test metamodels (TMM in [9]) and extended UML object models are used for this task. Alternatively, the test specifications can be created as, for instance, external code models [25]. During test execution, the test models are bound late to the actual MUT (e.g., through just-in-time instantiation of the metamodel or entity resolution according to the test model details). Moreover, to limit the test-modeling effort, groups of related test models with some variation points can be defined (referred to as test specifications in [9]). Test models are suitable for deriving testable requirements specifications early. However, each test model must not only reflect the metamodel fragment relevant to the requirement tested, but also the context of this metamodel fragment to represent a *bindable* instance of the MUT (e.g., also auxiliary model types must be resolvable). This makes test models vulnerable to metamodel changes which do not directly affect the tested requirements (such as metamodel refactorings). This, again, requires the active maintenance of test models. Finally, establishing variation points for a test model manually (e.g., facing a complex multiplicity configuration) is not trivial.

Metamodel validation: A testing approach using model-constraint expressions (e.g., defined via OCL or EVL) specifies test cases in terms of collections of model constraints on MUTs (e.g., specified as invariants), defined at the level of the corresponding meta-metamodel (see [22]). The specification of metamodel constraints requires expertise in both the meta-modeling language and the underlying constraint language. However, the translation of requirements (e.g., a narrative text, a requirements catalog, or variability models) into well-defined constraint expressions is not trivial. Nevertheless, as the expressions are defined over the meta-metamodel instantiation structure (e.g., MOF or Ecore repository viewpoint) of the MUT, the resulting tests are widely decoupled from details of the evolving MUTs (e.g., navigation axes between model types, the domain of model-types). Model-constraint expressions are typically organized according to the built-in constructs of the specification languages (e.g., via operations, invariants, and query blocks). While the need for structuring of model constraints, for example, to match a certain testing level, has been acknowledged (see [27]), existing approaches do not consider the structure of non-executable requirements specifications, such as semi-structured textual or diagrammatic scenario descriptions (see, e.g., [10]). Such abstraction mismatches complicate the co-maintenance of the requirements description and the corresponding model constraints.

III. Scenario-based Testing

In software engineering, scenarios are used to specify user needs as well as to explore and to define (actual or intended) system behavior (see, e.g., [11]–[15]). Scenarios can be described in different ways at various abstraction levels, for example, via structured text, graphical models, or precise (and formal) textual specifications. For specifying a software system, they are typically defined using different types of models, such as UML interaction or activity models.

The different action steps in a non-executable scenario description can then be refined through detailed, executable scenario tests. Detailed scenarios are used to depict the dynamic runtime structures of a system, for instance, to show how a certain functionality is realized on the level of interacting software components. Therefore, scenarios are a natural source for behavior tests. Non-executable scenario descriptions for a DSML can directly be defined by domain experts to serve as an (additional) input for software engineers to implement integration and component tests at the implementation level (see, e.g., [28]).

As it is almost impossible to completely test a complex software system, effective means are needed to select relevant tests, to express and to maintain them, and to automate test procedures whenever possible. Scenarios can help to reduce the risk of omitting or forgetting relevant test cases, as well as the risk of describing important tests insufficiently. If each design-level scenario is checked via a corresponding scenario test, a critical test coverage of the most relevant requirements on the MUT can be achieved. Moreover, in a thorough engineering approach, changing domain requirements are first identified at the scenario level (see also [11], [12]). Hence, one can rapidly identify affected scenario tests and propagate the changes into the corresponding test specifications.

IV. Language-Model Testing using Scenarios

Performing a scenario-based testing and development process for metamodels involves planning activities (e.g., deciding on a test procedure) and the creation of a number of testing artifacts, such as non-executable scenario descriptions and executable test scenario specifications. To support such a testing process, we developed a prototype infrastructure as a scenario-oriented extension of the Epsilon EUnit testing framework [29]. The prototype is based on a scenario-test metamodel from [30] and realizes a concrete syntax to define scenario-based test specifications. Furthermore, it provides runtime and reporting support. Metamodel testing is so available for several metamodel types (e.g., EMF/Ecore, XML).

A. Metamodel-Testing Procedures with Scenarios

Several testing procedures can be supported by scenarios, including regression tests. Fig. 2 shows a process for the definition of an existing MUT. The collected scenario tests are then used to validate the MUT which is modified by a sequence of meta-modeling actions. Before a new action is performed, the scenario tests validate the changed MUT' for requirements conformance and a test report is issued. Upon successful completion of the corresponding test scenario, the next action can be performed by the DSML engineer. Otherwise, the MUT must be adjusted to comply with the test scenarios. Such a testing procedure is suitable for metamodel refactoring tasks (e.g., partitioning into sub-packages, restructuring of relationship representations; see [17]). This procedure may be repeated for the MUT', for example, when creating a revised metamodel version due to new domain requirements (see Fig. 2).

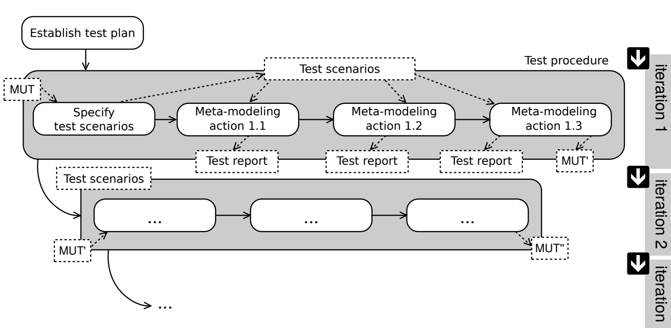

Fig. 2. Iterative language-model development and scenario-based testing.

Fig. 3 shows how a domain expert and the DSML engineer collaborate to complete the above testing and development procedure. Note that in some domains (e.g., software testing), the roles of the domain expert (e.g., the software tester) and of DSML engineer can be shared by single subjects (e.g., a software tester who develops and employs a testing DSML). The domain expert defines a guiding scenario description which is then mapped onto a suitable testing infrastructure by the DSML engineer. After having the domain expert and the DSML engineer collaboratively review the executable scenario test, the DSML engineer takes on the actual meta-modeling action (e.g., a refactoring or a DSML language-definition step). The results are then fed into the scenario tests relevant for the corresponding part of the core language model. If the tests fail, this part of the core language model does not meet the respective requirements and must be adjusted. If the tests succeed, the next action can be performed by the DSML engineer (see Fig. 2).

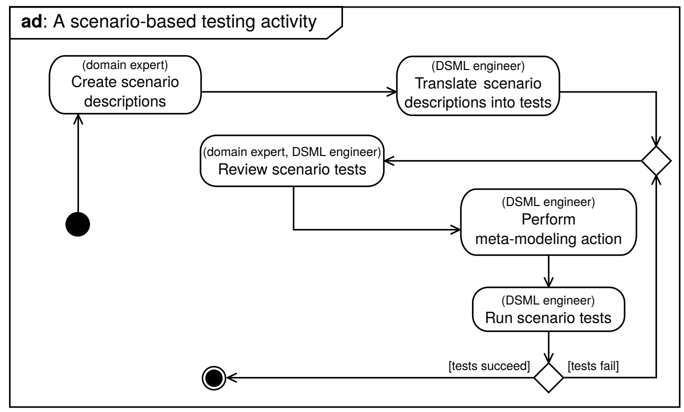

Fig. 3. Scenario-based testing for language-model fragments.

In an alternative procedure, there might be no pre-existing MUT. Each meta-modeling action adds to the incomplete MUT and creates a testable scenario specification (or fragments thereof). After having completed the last meta-modeling action, an initial MUT and a set of scenario tests are available. This procedure leans itself towards the initial and step-wise definition of a DSML model (see, e.g., story boarding in [31]). These two procedures and variants thereof can also be combined.

The non-executable scenario descriptions provided by the domain expert can be defined in different ways. For demonstration purposes, we adopt the one-column table format as found in [10]. Content-wise, a scenario description should establish the conditions under which it runs, i.e. a trigger and *preconditions*. In addition, the *scenario goal* which is to be achieved (e.g., testable postconditions) should be given. Finally, a set of validation or action steps which form the *scenario body* must be provided.

The conceptual model of the executable scenario-test specification available to the DSML engineer is depicted in Fig. 4. The corresponding specification syntax as realized by our EMF/Epsilon prototype is shown in Listing 1. A *test scenario* tests one particular facet of a system, here the MUT. In the first place, it represents one particular action and event sequence which is specified through the test body of the respective scenario. In addition to the test body, each test scenario includes an expected result and may include a number of preconditions and postconditions, as well as a setup sequence and a cleanup sequence (see Fig. 4). When a test scenario is executed, it first executes the corresponding setup sequence. A setup sequence includes an action sequence that is executed to set up an evaluation environment for the corresponding scenario, for example, a setup sequence may load and create several models as well as define helper operations required by the test body. Next, the preconditions of the scenario are checked. If at least one precondition fails, the test scenario is aborted and marked as incomplete. If all preconditions are fulfilled, the test body is executed. In particular, the action sequence in the test body produces a test result. This test result is then checked against the expected result using appropriate matcher and comparison operations. If the check fails, the test scenario is aborted and marked as incomplete. If the check is successful, the postconditions of the scenario are checked. Again, if at least one postcondition fails, the test scenario is aborted and marked as incomplete. If all postconditions are fulfilled, the cleanup sequence is called and the scenario is marked as complete. A cleanup sequence includes an action sequence that is executed to undo actions that were made during the test scenario. For example, the cleanup sequence can delete intermediate models and model elements created by the setup sequence. Note that the cleanup sequence is executed each time the respective test scenario is executed, even if the scenario is marked as incomplete.

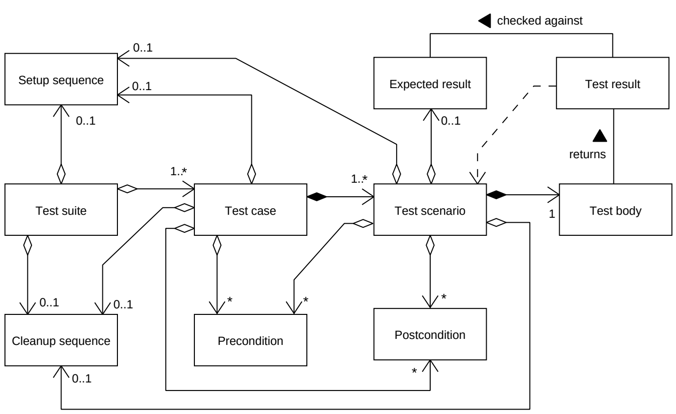

Fig. 4. Scenario-based testing domain model [30].

Each test scenario is part of a *test case*. In particular, a test case consists of one or more test scenarios and may include a number of preconditions and postconditions, as well as a setup sequence and a cleanup sequence (see Fig. 4). When a test case is run, it first executes the respective setup sequence. The runtime structures produced by the setup sequence are then available to all test scenarios of the corresponding test case. Subsequently, the preconditions of the test case are checked. Similar to test scenarios, a test case is aborted and marked as incomplete if one of its preconditions or postconditions fails. Next, each test scenario of the test case is executed as described above. If at least one test scenario is incomplete, the including test case is also marked as incomplete. After the test scenarios, the postconditions are checked before the test case cleanup sequence is executed. The test case cleanup sequence is executed each time the corresponding test case is performed.

Each test case is part of a *test suite* (see Fig. 4) and a test suite includes one or more test cases. Furthermore, a test suite may have a setup sequence and a cleanup sequence, as well. Again, the runtime structures produced by the test-suite setup sequence are available to all test cases of the corresponding suite.

```
@TestSuite
$setup    -- test suite A setup sequence
$cleanup  -- test suite A cleanup sequence
operation TestSuite_A() {
  @TestCase
  $pre      -- test case A precondition
  $post     -- test case A postcondition
  $setup    -- test case A setup sequence
  $cleanup  -- test case A cleanup sequence
  operation TestCase_A() {
    @TestScenario
    $pre      -- test scenario A precondition
    $post     -- test scenario A postcondition
    $setup    -- test scenario A setup sequence
    $cleanup  -- test scenario A cleanup sequence
    operation TestScenario_A() {
      -- test scenario A specification
    }
    @TestScenario
    $pre      -- test scenario B precondition
    -- ...    -- further pre-/postconditions, setup/cleanup
        sequences
    operation TestScenario_B() {
      -- test scenario B specification
    }
  }
  @TestCase
  operation TestCase_B() {
    -- test scenario specifications for test case B
  }
}
```

Listing 1. A concrete syntax for scenario-test specifications.

B. Scenario-based Metamodel-Testing Infrastructure

Our approach for scenario-based metamodel testing is implemented as an Eclipse-based prototype.[1] The prototype provides support for authoring, execution, and reporting of scenario-based test specifications as introduced in Section IV-A. Our prototype leverages the capabilities of the Epsilon family of model-management languages (see [32]). Among others, Epsilon provides the Epsilon Unit Testing Framework (EUnit, see [29]) which is designed to define tests for model-management tasks. EUnit itself is an embedded language which reuses constructs of the core Epsilon Object Language (EOL) to implement test-specific functionality (e.g., special annotations to define test operations). In our prototype, we developed a language extension of EOL. In this way, we reuse important EUnit and EOL features such as the built-in test annotations, guarding expressions, and the setup/cleanup operations while providing our own extensions to support scenario-based test specifications (see Fig. 4).

Our extension tackles four requirements which result from the conceptial metamodel introduced in Section IV-A: Scenario-based test specifications can include test suites, test cases, and test scenarios (see Sections III and IV). Therefore, our EUnit extension for scenario-based testing must be able to *explicitly distinguish between these test concepts* (R_1). Furthermore, a test case includes one or more test scenarios and a test suite groups one or more test cases. These *containment relationships* must unfold into a particular *sequencing of test execution*, as explained in Section IV-A (R_2). Test suites, test cases, and test scenarios each include *setup as well as cleanup*

[1]All software artifacts are publicly available at http://nm.wu.ac.at/modsec.

sequences (R_3). Finally, test cases and test scenarios must support *guard conditions which are evaluated before and after* test-case and test-scenario executions, respectively (R_4).

Language-model extensions: To address the four requirements defined above, we adapted the EOL model (i.e., the abstract syntax) and its behavioral specification accordingly. Fig. 5 shows an excerpt from the EOL language model as a UML class diagram. At the topmost level, EUnit tests are grouped into modules (`EOL Modules`) which are containers for `Statements` (e.g., any EOL logical expressions, conditional expressions) as well as annotation and operation definitions (`OperationDeclarationOrAnnotationBlock`). `Annotations` (grouped into `AnnotationBlocks`) add orthogonal metadata to `OperationDeclarations` and can be subdivided in two categories: An `ExecutableAnnotation` contains an EOL statement for evaluation while a `SimpleAnnotation` simply marks operations. Each EUnit test is implemented as a unit pair of an EOL operation and an attached `@test` annotation (see [29]). Operations and annotations are the only named structuring techniques available for EOL and EUnit, a notion of objects is not available. Test-implementing operations (`OperationDeclaration`) can contain an arbitrary EOL `StatementBlock` as test and operation body. In the operation and test bodies, a number of built-in assertion/matcher primitives of EUnit can be used along with model-management helpers (see [29]). The return value (`ReturnStatement`) of a test (i.e., an annotated `OperationDeclaration`; see Fig. 5) must be evaluated against an expected result to establish whether a test passes or fails. These return values can be of an Epsilon-internal type (e.g., `PrimitiveType`, `Collection` etc.), of an element type of a loaded model (`ModelElementType`), or of any Java type. For example, the return value can be accessed in postcondition blocks (`$post`) via the built-in `_result` variable (see [32]).

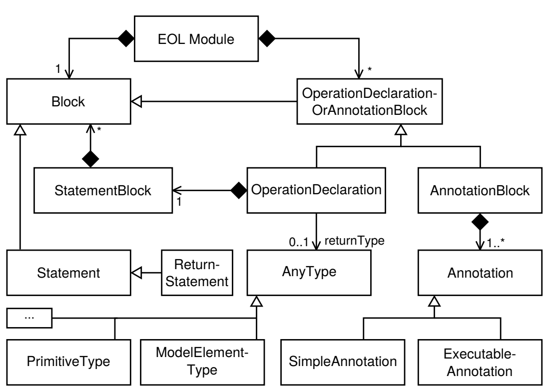

Fig. 5. Excerpt of the building blocks of our Epsilon prototype.

As an embedded EOL extension, EUnit makes heavy use of the `Annotation` feature (see [29], [32]). Therefore, we also used `Annotations` to extend the abstract syntax and the semantics (e.g., the composite execution modes). To provide test suites, test cases, and test scenarios as language constructs (see requirement R_1), we defined three `SimpleAnnotation` kinds. Similarly, annotations to specify setup/cleanup sequences at all three test levels (see requirement R_3) are provided. Moreover, by declaring additional `ExecutableAnnotations`, pre- and postconditions can be evaluated for all test levels (see requirement R_4).

To fully comply with requirement R_1 and to provide containment relationships between test suites, test cases, and test scenarios (see requirement R_2), we used nested EOL operations, with the following restrictions: EOL `Statements` which form operation bodies have been extended to allow for declaring `OperationDeclarationOrAnnotationBlocks`, i.e. nested operation declarations. Operations which are declared as nested operations have limited visibility and accessibility. For example, there is no lexical scoping at the script level, they cannot be called explicitly even from within the enclosing operation. Rather, the EUnit engine internally collects the nested operation declarations and performs their evaluation upon execution of the enclosing operation. Thus, it was necessary to alter both the EOL grammar specification and the EOL execution engine. We modified the corresponding ANTLR grammar to accept nested operation declarations and their attached annotations. The corresponding changes to the dispatcher of operations and (executable) annotations are implemented via refined methods which operate on the respective abstract-syntax-graph representations accordingly (see requirement R_2).

To sum up, Table I shows the correspondences between scenario-based testing domain concepts from Fig. 4, the extended EUnit concrete syntax (see also Listing 1), and the underlying EOL language-model entities (see also Fig. 5).

TABLE I. CORRESPONDENCES BETWEEN SCENARIO-TESTING CONCEPTS, EUNIT CONCRETE SYNTAX, AND EOL ABSTRACT SYNTAX.

Domain concept	**Epsilon syntax construct**	**Epsilon object**
`Test suite`	`@TestSuite`	`SimpleAnnotation`
`Setup sequence`	`$setup`	`ExecutableAnnotation`
`Cleanup sequence`	`$cleanup`	`ExecutableAnnotation`
`Test case`	`@TestCase`	`SimpleAnnotation`
`Precondition`	`$pre`	`ExecutableAnnotation`
`Postcondition`	`$post`	`ExecutableAnnotation`
`Test scenario`	`@TestScenario`	`SimpleAnnotation`
`Test body`	*operation's body*	`Statement`
`Test result`	*operation's return value*	`ReturnStatement`
`Expected result`	*as defined (data, model etc.)*	*of type* `AnyType`

Concrete-syntax extensions: The concrete syntax for scenario-based test specifications as shown in Listing 1 provides the textual interface for domain experts and DSML engineers. With respect to requirements R_1–R_4, the ANTLR grammar specification of EOL was adapted slightly, in a fully backward-compatible way. These adaptations allow for nested operation declarations (see Listing 2, lines 14–15)[2]. Besides this small modification, the EOL grammar is reused *as is*.

```
OperationDeclarationOrAnnotationBlock
        = AnnotationBlock | OperationDeclaration;
AnnotationBlock
        = Annotation { Annotation };
Annotation
        = SimpleAnnotation | ExecutableAnnotation;
OperationDeclaration
        = ("operation"|"function") [Type] Name
          "(" [FormalParameterList] ")" [":" Type]
             StatementBlock;
StatementBlock
        = "{" Block "}";
```

[2]Please note that only the modified grammar rules are shown, syntax entities relating to, for instance, model loading or model transformation are omitted for brevity. See [32] for the grammar details.

```
(* Allowing for nested operation and annotation declarations
    : *)
Block
        = { OperationDeclarationOrAnnotationBlock |
              Statement };

SimpleAnnotation
        = "@", Name [Value {"," Value}];
ExecutableAnnotation
        = "$", Name LogicalExpression;

(* The rules below are reused from the standard EOL grammar
    *)
Statement                  = ? ... ?;
Name                       = ? ... ?;
Value                      = ? ... ?;
LogicalExpression          = ? ... ?;
Type                       = ? ... ?;
FormalParameterList        = ? ... ?;
```

Listing 2. Excerpt from the extended EUnit grammar specification, in EBNF.

Advanced features: Non-executable scenario descriptions can include cross-references within the same scenario and between two or more scenarios (see [10]). For example, a scenario fragment or an extension scenario may refer to a superordinate scenario's goal as their end condition. Similarly, domain requirements may map to the equal pre- and postconditions in scenario tests, to establish the intention of invariance. To avoid code redundancy in scenario-based test specifications, constraint expressions and, more generally, EOL statements can be specified in two ways for reuse between scenario tests and/or between different testing levels (suite, case, scenario).

First, they can be defined as freestanding, helper EOL operations. This is possible because conditions (`$pre`, `$post`) and sequences (`$setup`, `$cleanup`) can refer to arbitrary EOL `LogicalExpressions` including operation-call statements (see Listing 2 and [32]). Second, to share statements for setup and cleanup sequences between test operations of an entire test level, helper operations can be associated with the following annotations `@SuiteSetup/@SuiteCleanup`, `@CaseSetup/@CaseCleanup`, and `@ScenarioSetup/@ScenarioCleanup`. These annotations register the annotated operations as the authoritative setup and cleanup sequences with the corresponding test level (i.e. test suite, test case, test scenario). Note that these global setup and cleanup sequences can be combined with local ones: During a test run, when executing setup sequences, the global sequences take precedence over the local ones. For cleanup sequences the precedence is inverse, with the global sequences being run after the local ones.

V. Scenario-based Testing Procedure for a DSML Integration Case

To demonstrate our approach, we now discuss a case for the integration of two DSMLs `A` and `B` which represent two narrow domains: system auditing and reactive distributed systems (see Figs. 6 and 7). The integrated DSML should cover a new and an integrated domain (i.e., auditable distributed systems). While a more detailed background on this application case is given in previous work (see [6]), it is important to note that this application case is about a coupled DSML integration. The derived DSML remains backward-dependent on the source DSMLs (e.g., to track perfective changes in the source languages). In this paper, we walk through a small case fragment.

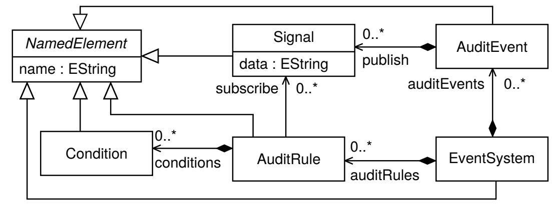

Fig. 6. Auditing event-based systems (DSML `A`).

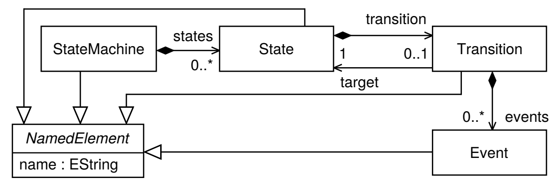

Fig. 7. State/transitional behavioral system (DSML `B`).

This fragment allows us to demonstrate a testing procedure as depicted in Fig. 8. This procedure resembles one of the characteristic procedures described in Section IV-A, with scenario tests being specified upfront. The new, derived DSML `C` is created in three meta-modeling steps: 1) defining entities and their internal structure, 2) establishing entity relationships, and 3) enforcing new domain-specific language-model constraints. During and between each step, the scenario tests are executed to check the requirements conformance.

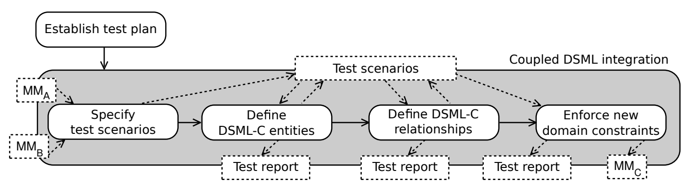

Fig. 8. A step-wise, scenario-assisted DSML integration process using model transformations.

Following a scenario-driven test plan (see Figs. 8 and 9), the main domain actors (e.g., a security-audit expert and the distributed-systems expert) draft non-executable scenario descriptions based on the agreed domain requirements. Table II exemplifies an excerpt from the resulting scenario descriptions in a one-column table format [10].

Scenario descriptions: In our example, the metamodels of the DSMLs `A` and `B` should be fully composed. The conceptual weaving is to be achieved by turning `Events` propagated in a distributed system into `AuditableEvents` that can be tracked for auditing purposes (e.g., through an appropriate system-monitoring facility). Furthermore, the domain requires all events to be audited, without exception. Also, each audited event must issue a `Signal` to the monitoring facility. These three goals are clearly documented in terms of the three scenario sections of Table II. In addition, the domain experts document the prerequisites for achieving these goals (e.g., presence conditions of certain entities in the source metamodels).

Scenario-test specifications: In a next step, the DSML engineer specifies the test cases based on the scenario descriptions. The DSML engineer maps certain document sections to

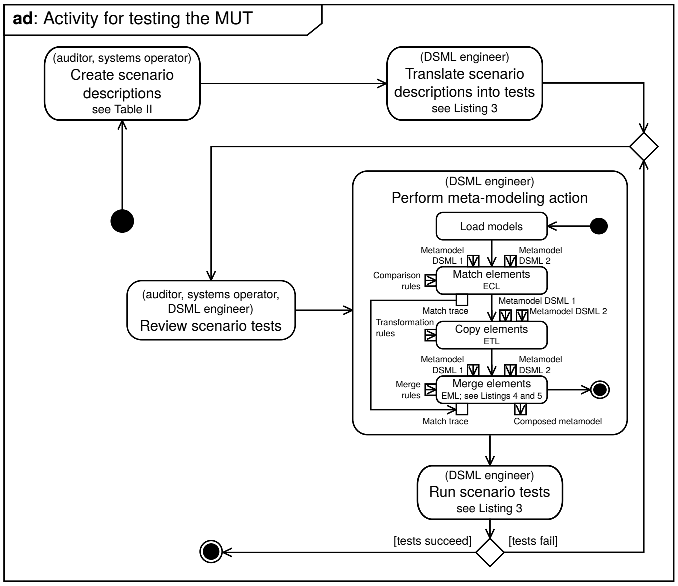

Fig. 9. Scenario-based DSML integration via model transformations.

TABLE II. REQUIREMENTS-LEVEL SCENARIO DESCRIPTION.

Test case 1	In order to support auditable events in a reactive distributed system, DSMLs `A` and `B` must be integrated. An auditable distributed system must be fully integratable into the current target software platforms, using existing code-generation templates. Therefore, a complete and structure-preserving DSML composition must be performed.
Primary actors	System auditor, distributed-systems operator
Trigger/Setup	The model-transformation workflow to integrate the metamodels of DSML `A` & DSML `B` is executed.
Test scenario 1	The central concept are `AuditableEvents` to be propagated and monitored. The new concept `AuditableEvent` must share all features of the `Event` (DSML `A`) and `AuditEvent` (DSML `B`) concepts.
Preconditions	The source concepts `Event` and `AuditEvent` must be available in the DSMLs `A` and `B`.
Expected result	A metaclass named `C::AuditableEvent` with the combined structural features of `A::Event` and `B::AuditEvent`.
Test scenario 2	In DSML `C`, each `AuditableEvent` publishes `Signals`.
Expected result	`AuditableEvents` must maintain a reference named `publish` to `Signal`.
Test case 2	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 must 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` must publish at least one `Signal`.
Preconditions	`AuditableEvent` has all structural features of `AuditEvent` and `Event`.
Expected result	Instances of `AuditableEvent` must refer to at least one `Signal` instance.

selected parts of an EUnit scenario-based test structure (e.g., preconditions in the document become `$pre` annotations) and operationalizes the requirements by translating them into constraint expressions over the source and the target metamodels (e.g., specific bound checks for multiplicity elements). One possible scenario-test specification is shown in Listing 3.

```
1  @TestSuite
2  $setup runTarget("merge")
3  operation TestSuite_1() {
4    @TestCase
5    operation TestCase_1() {
6      @TestScenario
7      $pre EventSystem!EClass.all->exists(c | c.name = "
           AuditEvent")
8      $pre StateMachine!EClass.all->exists(c | c.name = "Event
           ")
9      operation TestScenario_1() {
10       assertTrue("Missing composed classifier.",
             EventSystemStateMachine!EClass.all->exists(ae |
             ae.name = "AuditableEvent"));
11     }
12     @TestScenario
13     operation TestScenario_2() {
14       assertTrue("Firing event of a transition must be
             capable of publishing signals.",
             EventSystemStateMachine!EClass.all->selectOne(c |
              c.name = "Transition").eStructuralFeatures->
             selectOne(tsf | tsf.name = "events").eType.
             eStructuralFeatures->exists(aesf | aesf.name = "
             publish"));
15     }
16   }
17   @TestCase
18   $pre verifyEntities(StateMachine!EClass)
19   $pre verifyEntities(EventSystemStateMachine!EClass)
20   operation TestCase_2() {
21     @TestScenario
22     $pre EventSystemStateMachine!EClass.all->selectOne(ae |
           ae.name = "AuditableEvent").eStructuralFeatures.
           isEmpty() = false
23     operation TestScenario_1() {
24       assertFalse("An AuditableEvent in the context of a
             transition must publish at least one signal.",
             EventSystemStateMachine!EClass.all->selectOne(c |
              c.name = "Transition").eStructuralFeatures->
             selectOne(sf | sf.name = "events").eType.
             eStructuralFeatures->first().lowerBound = 0);
25     }
26   }
27 }
28 operation verifyEntities(eClass) {
29   -- Check for valid composition candidates (Note: details
         are omitted)
30 }
```

Listing 3. A possible mapping of the scenario descriptions to scenario tests.

The top-level test suite groups the two corresponding test cases and three test scenarios (lines 1–27). The first test case (lines 4–16) includes two scenarios. The first scenario (lines 6–11) requires two preconditions to be fulfilled (lines 7–8). The second test scenario (lines 12–15) verifies whether an event triggered by a transition is capable of publishing signals. The event must be of type `AuditableEvent`. The second test case (lines 17–26) utilizes the helper `verifyEntities` (lines 28–30) for the evaluation of two preconditions (lines 18–19), with each running the test on a different metamodel. The third test scenario (lines 21–25) checks for the mandatory signaling by system events (see above). At this stage, when executed, all tests will be reported failed.

Initial composition specifications: Once having the EUnit scenario-test specifications reviewed collaboratively by the domain experts and the DSML engineer, the DSML engineer specifies the actual metamodel composition. In this application case, this is achieved by devising an Epsilon-based composition workflow [6][3]. To provide an impression, Listing 4 shows the Epsilon Merging Language (EML [32]) rule for the creation of `AuditableEvent`. The merge procedure creates an `EClass` of the required name (line 5), establishes inheritance relationships, and incorporates the structural features from both source DSMLs (lines 6–7). Once performed, all except for one scenario test defined in Listing 3 pass. Fig. 10 shows the EUnit console reporting the failing test scenario.

[3] See also the meta-modeling action in Fig. 9.

```
rule MergeAuditEvent
  merge l : EventSystem!EClass
  with r : StateMachine!EClass
  into t : EventSystemStateMachine!EClass {
    t.name = "AuditableEvent";
    t.eSuperTypes ::= l.eSuperTypes + r.eSuperTypes;
    t.eStructuralFeatures ::= l.eStructuralFeatures + r.
         eStructuralFeatures;
}
```

Listing 4. EML merge rule for DSML composition.

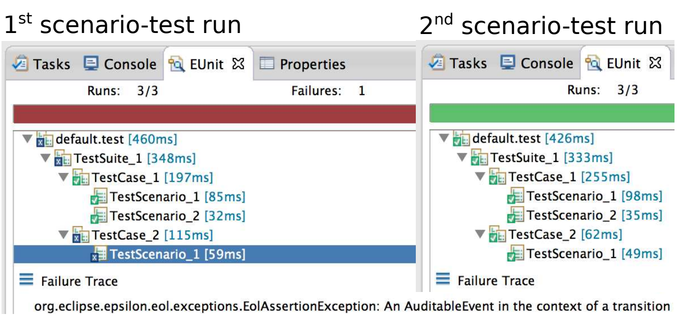

Fig. 10. Scenario-test reports in EUnit.

Patching composition specifications: Based on the test report, the DSML engineer reviews jointly with the domain experts the failing test scenario to exclude an erroneous specification. Once verified, the DSML engineer investigates the initial composition specification (Listing 4). The DSML engineer realizes that the source metaclass `A::AuditEvent` for the composed `AuditableEvent` does not conform to the requirement of the targeted domain because `A::AuditEvent` does not necessarily have to contain a `Signal` given the lower multiplicity bound of 0 of the `publish EReference`. To fix this, the DSML engineer adds a statement to the EML merge rule which modifies the lower bound accordingly (see line 9 in Listing 5). With this, all scenario tests pass (see Fig. 10). Fig. 11 documents the critical metamodel fragment of the final composed DSML.

```
1 rule MergeAuditEvent
2   merge l : EventSystem!EClass
3   with r : StateMachine!EClass
4   into t : EventSystemStateMachine!EClass {
5     t.name = "AuditableEvent";
6     t.eSuperTypes ::= l.eSuperTypes + r.eSuperTypes;
7     t.eStructuralFeatures ::= l.eStructuralFeatures +
8 r.eStructuralFeatures;
9     t.eStructuralFeatures->selectOne(sf | sf.name = "publish
         ").lowerBound = 1;
10 }
```

Listing 5. Refinement of EML merge rule.

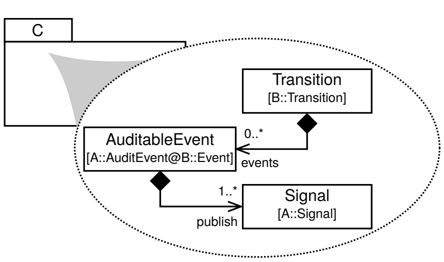

Fig. 11. Relevant excerpt from the final metamodel `C`.

VI. Further Related Work

In Section II, we have already iterated over closely related work on metamodel testing which falls into three categories: modeling-space sampling ([7], [8], [23]), metamodel-test models ([9], [25]), and metamodel validation [22].

Tort et al. ([16], [33]) have investigated testing support for conceptual modeling. In their approach, conceptual schemas are defined using UML models (at the M1 level) and OCL model constraints. We consider MUTs at the M2 level. Conceptual schemas cover both structural (entities, entity relationships) and behavioral aspects (events) while we look at an MUT as the structural specification of a core language model. Assisted by a dedicated Conceptual Schema Testing Language (CSLT) and runtime, executable test specifications can exercise a conceptual schema under test. State changes and state-based assertion checking as well as the temporal validation of event creation and occurrence can be tested. In a test-first application of the approach [16], test instantiations are specified to guide the development process. The runtime for model and test execution allows for testing UML models and the corresponding OCL model constraints to identify consistency defects and requirement inconsistencies.

The application case in Section V demonstrates that a testing facility which can refer to several metamodels at once is suitable for expressing test cases on model transformations [24]. For example, in our scenario-test format, preconditions expressed over the source metamodels and postconditions on the target metamodels establish a transformation contract. This closely resembles the idea of partial test oracles for model transformations (see, e.g., the basic precondition and postcondition contracts in [34]). Besides assertion checking, such contractual constraints can also be used as criteria for generating input test models (see, e.g., [35]). Moreover, requirements-level testing including non-executable requirements descriptions was also explored for model transformations [36].

Approaches to metamodel testing as ours apply to testing support of language-model and abstract-syntax design in isolation. A systematic alignment of testing activities to other phases is widely missing. Sadilek et al. [37] touch on all phases and their testing requirements, however, they do not analyze testing techniques other than their metamodel-testing approach (MMUnit [9]) in detail. Recently, one of the authors [38] evaluated the adequacy of general-purpose testing techniques for the various phases of DSML integration, including visual-syntax testing and testing of composed platform-integration artifacts (e.g., rewritten generator templates [6]). Nevertheless, metamodel testing affects indirectly other phases and language-model artifacts dependent on the MUTs: Merilinna et al. [8] provide for indirect testing of the platform-specific artifacts by generating and deploying them during metamodel testing. As metamodels can also be systematically derived from or transformed into corresponding grammar definitions (see, e.g., [39]), test-based validation can so extend partially to the grammar-based textual concrete syntaxes. The case of dependent model transformations is mentioned above.

VII. Conclusion

In this paper, we presented an approach for the scenario-based testing of core language models. The core language model is a metamodel that defines the abstract syntax of a DSML. Because the core language model is central to the proper implementation of a DSML, it is very important to ensure the correctness and consistency of this metamodel. Moreover, in case two (or more) DSMLs are integrated to define a composite DSML, it is also important to systematically check the corresponding composed core language model.

Our approach uses domain scenarios at the requirements level as primary artifacts. These non-executable scenario descriptions are refined into executable scenario tests. In this way, our approach integrates scenario descriptions on different abstraction layers. This is a first step towards providing forward- and backward-traceability for DSML test scenarios (also future work). To demonstrate our approach, we implemented a corresponding extension to the Eclipse Modeling Framework and Epsilon model-management toolkit.

Acknowledgment

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

[1] D. Schmidt, "Model-driven engineering – guest editor's introduction," *IEEE Comp.*, vol. 39, no. 2, Feb. 2006.

[2] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003.

[3] T. Stahl and M. Völter, *Model-Driven Software Development*. John Wiley & Sons, 2006.

[4] B. Meyers and H. Vangheluwe, "A framework for evolution of modelling languages," *Sci. Comput. Program.*, vol. 76, no. 12, pp. 1223–1246, 2011.

[5] M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," *SP&E*, vol. 39, no. 15, pp. 1253–1292, 2009.

[6] B. Hoisl, S. Sobernig, and M. Strembeck, "Higher-order rewriting of model-to-text templates for integrating domain-specific modeling languages," in *Proc. 1st Int. Conf. Model-Driven Eng. and Softw. Development*. SciTePress, 2013, pp. 49–61.

[7] J. Gomez, B. Baudry, and H. Sahraoui, "Searching the boundaries of a modeling space to test metamodels," in *Proc. 5th IEEE Int. Conf. Softw. Testing, Verification and Validation*. IEEE, 2012, pp. 131–140.

[8] J. Merilinna, O.-P. Puolitaival, and J. Pärssinen, "Towards model-based testing of domain-specific modelling languages," in *Proc. 10th Workshop Domain-Specific Modeling*, 2008. [Online]. Available: http://www.dsmforum.org/events/dsm08/Papers/7-Puolitaival.pdf

[9] D. A. Sadilek and S. Weißleder, "Testing metamodels," in *Proc. 4th European Conf. Model Driven Architecture – Foundations and Applications*, ser. LNCS, vol. 5095. Springer, 2008, pp. 294–309.

[10] A. Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2001.

[11] M. Jarke, X. Bui, and J. Carroll, "Scenario management: An interdisciplinary approach," *Requirements Eng. J.*, vol. 3, no. 3/4, 1998.

[12] A. Sutcliffe, *User-Centred Requirements Engineering*. Springer, 2002.

[13] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Trans. Softw. Eng.*, vol. 29, no. 2, February 2003.

[14] J. Carroll, *Scenario-Based Design: Envisioning Work and Technology in System Development*. John Wiley & Sons, 1995.

[15] ——, "Five reasons for scenario-based design," *Interacting with Computers*, vol. 13, no. 1, pp. 43–60, 2000.

[16] A. Tort, A. Olivé, and M.-R. Sancho, "An approach to test-driven development of conceptual schemas," *Data Knowl. Eng.*, vol. 70, no. 12, pp. 1088–1111, 2011.

[17] I. García-Magariño, R. Fuentes-Fernández, and J. J. Gómez-Sanz, "Guideline for the definition of EMF metamodels using an entity-relationship approach," *Inform. Softw. Tech.*, vol. 51, no. 8, pp. 1217–1230, 2009.

[18] F. Lagarde, H. Espinoza, F. Terrier, C. André, and S. Gérard, "Leveraging patterns on domain models to improve UML profile definition," in *Fundamental Approaches to Software Engineering*, ser. LNCS. Springer, 2008, vol. 4961, pp. 116–130.

[19] K. Czarnecki, "Overview of generative software development," in *Proc. 2004 Int. Conf. Unconventional Programming Paradigms*. Springer, 2005, pp. 326–341.

[20] A. Vallecillo, "On the combination of domain specific modeling languages," in *Proc. 6th European Conf. Modelling Foundations and Applications*, ser. LNCS. Springer, 2010, vol. 6138, pp. 305–320.

[21] B. Hoisl, M. Strembeck, and S. Sobernig, "Towards a systematic integration of MOF/UML-based domain-specific modeling languages," in *Proc. 16th IASTED Int. Conf. Softw. Eng. and Applications*. ACTA Press, 2012, pp. 337–344.

[22] J. Merilinna and J. Pärssinen, "Verification and validation in the context of domain-specific modelling," in *Proc. 10th Works. Domain-Specific Modeling*. ACM, 2010, pp. 9:1–9:6.

[23] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *SoSyM*, vol. 4, no. 4, pp. 386–398, 2005.

[24] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. L. Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Commun. ACM*, vol. 53, no. 6, pp. 139–143, Jun. 2010.

[25] A. Cicchetti, D. D. Ruscio, D. S. Kolovos, and A. Pierantonio, "A test-driven approach for metamodel development," in *Emerging Technologies for the Evolution and Maintenance of Software Models*, J. Rech and C. Bunse, Eds. IGI Global, 2011, pp. 319–342.

[26] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Sci. Comput. Program.*, vol. 69, no. 1–3, pp. 27–34, 2007.

[27] L. Hamann and M. Gogolla, "Improving model quality by validating constraints with model unit tests," *Proc. 2010 Works. Model-Driven Eng., Verification, and Validation*, pp. 49–54, 2010.

[28] R. Nord and J. Tomayko, "Software architecture-centric methods and agile development," *IEEE Softw.*, vol. 23, no. 2, March/April 2006.

[29] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo, "EUnit: A unit testing framework for model management tasks," in *Proc. 14th Int. Conf. Model Driven Eng. Languages and Systems*, ser. LNCS, vol. 6981. Springer, 2011, pp. 395–409.

[30] M. Strembeck, "Testing policy-based systems with scenarios," in *Proc. 10th IASTED Int. Conf. Softw. Eng.* ACTA Press, 2011, pp. 64–71.

[31] I. Diethelm, L. Geiger, and A. Zündorf, "Applying story driven modeling to the paderborn shuttle system case study," in *Proc. 2003 Int. Conf. Scenarios: Models, Transformations and Tools*, ser. LNCS, vol. 3466. Springer, 2005, pp. 109–133.

[32] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige, "The Epsilon book," Available at: http://www.eclipse.org/epsilon/doc/book/, 2013.

[33] A. Tort and A. Olivé, "An approach to testing conceptual schemas," *Data Knowl. Eng.*, vol. 69, no. 6, pp. 598–618, 2010.

[34] J.-M. Mottu, B. Baudry, and Y. L. Traon, "Reusable MDA components: A testing-for-trust approach," in *Proc. 9th Int. Conf. Model Driven Eng. Languages and Systems*, ser. LNCS, vol. 4199. Springer, 2006, pp. 589–603.

[35] M. Gogolla and A. Vallecillo, "Tractable model transformation testing," in *Proc. 7th European Conf. Modelling Foundations and Applications*, ser. LNCS, vol. 6698. Springer, 2011, pp. 221–235.

[36] P. Giner and V. Pelechano, "Test-driven development of model transformations," in *Proc. 12th Int. Conf. Model Driven Eng. Languages and Systems*, ser. LNCS, vol. 2795. Springer, 2009, pp. 748–752.

[37] D. A. Sadilek, M. Scheidgen, G. Wachsmuth, and S. Weißleder, "Towards agile language engineering," Humboldt University Berlin, Tech. Rep. 227, 2009.

[38] B. Hoisl, "Towards testing the integration of MOF/UML-based domain-specific modeling languages," in *Proc. 8th IASTED Int. Conf. Advances in Computer Science*. ACTA Press, 2013, pp. 314–323.

[39] M. Wimmer and G. Kramler, "Bridging grammarware and modelware," in *Proc. Satellite Events MoDELS 2005 Conf.*, ser. LNCS, vol. 3844. Springer, 2005, pp. 159–168.