*Proceedings of the IASTED International Conference*
*Advances in Computer Science (ACS 2013)*
*April 10 - 12, 2013 Phuket, Thailand*

# TOWARDS TESTING THE INTEGRATION OF MOF/UML-BASED DOMAIN-SPECIFIC MODELING LANGUAGES

Bernhard Hoisl
Institute for Information Systems and New Media, WU Vienna, Austria
Secure Business Austria Research (SBA Research), Austria
bernhard.hoisl@wu.ac.at

**ABSTRACT**
Domain-specific modeling languages (DSMLs) are commonly employed in the model-driven development (MDD) of software systems. As DSMLs are tailored for a narrow application domain, a software system needs to integrate multiple DSMLs for its complete specification. In this paper, we review the suitability of selected testing techniques for each phase of an MOF/UML-based DSML integration process. We exemplify every test technique by providing a motivating example of its application to the composition of existing, security-related DSMLs. As for evaluation, we provide for prototypical software implementations.

**KEY WORDS**
Domain-specific modeling language, language composition, integration test, model-driven development, UML

## 1 Introduction

In recent years, the model-driven development (MDD) of information systems attracts attention as an software engineering technique (see, e.g., [1]). In MDD, models are central artifacts and are used for the abstraction and description of the problem domain (e.g., requirements) as well as the solution techniques (e.g., implementation specifications) [2]. In the context of MDD, domain-specific (modeling) languages (DSLs/DSMLs) are frequently employed for the specification, GUI definition, and implementation of software systems for tailored application domains (see, e.g., [3]). Furthermore, model transformations provide for mappings of abstract high-level specifications (e.g., business process models) to system-level implementation artifacts (e.g., executable source code) [4].

DSMLs based on the Unified Modeling Language (UML [5]) are commonly applied, for instance, for the specification of security-related properties (see, e.g., [6]). A DSML that is based on the UML extends its specification with domain-specific constructs (notation, behavior, semantics). A UML-based DSML benefits from an integrated metamodeling architecture (defined via the Meta Object Facility, MOF [7]), standardized modeling extensions, and corresponding tool support.

Software systems are frequently subject to changing requirements and evolve over time (e.g., as a result of system maintenance, consolidation, or migration) [8]. In this context, the composition of DSMLs becomes an integral part of model-driven software evolution. As DSMLs are covering—by definition—a narrow problem domain, software systems may need to integrate multiple DSMLs for their full implementation [9]. DSML composition can be performed for all or selected language artifacts (e.g., language model, concrete syntax, or platform integration). For each composition task, one can pick from a broad range of integration techniques available as an informed decision (e.g., merging metamodels, extending concrete syntaxes, or pipelining DSML outputs on the host platform) [10].

The process of composing DSMLs puts special demands on testing the artifacts which produce an integrated DSML. The purpose of testing is to show that a certain software artifact operates as intended [11]. Thereby, the challenge of testing integrated DSMLs results from the vast number of DSML artifacts involved (e.g., different metamodels, model transformations, varying host platforms) [12]. For integrated DSMLs, we consider the individual DSML artifacts as having been tested at the unit and component levels. Thus, the composed artifacts of an integrated DSML need to be tested in combination by employing adequate integration and system testing techniques (see, e.g., [11]). Testing of composed DSMLs must be incorporated into all critical DSML development phases and, therefore, testing techniques should cover all process artifacts relating to an integrated DSML [10].

In this paper, we discuss the testing of composed MOF/UML-based DSMLs on the basis of the integration process defined in [10] (Section 2). We review a selected testing technique for each phase of the DSML integration process (by instrumenting integration test methods; Section 3, especially Sections 3.1–3.5). We further exemplify the system testing of model- and platform-level compliance of security properties for a complete MDD transformation process (Section 3.6). In addition, Section 4 discusses benefits, limitations, and transferability of our approach. At last, Section 5 classifies our work with respect to related approaches and Section 6 concludes the paper.

## 2 Background: DSML Integration Process

In [10], we define an integration process for MOF/UML-based DSMLs (Figure 1) which is adapted from [12]. The process results from our experiences in DSML

*DOI: 10.2316/P.2013.801-086*

engineering—over the past years we conducted 10+ MOF/UML-based DSML development projects—as well as from related literature [13, 14]. From the identified design decisions and corresponding decision options, we distilled commonly used integration techniques for MOF/UML-based DSMLs [10]. In this paper, the integration techniques are revisited and serve as the basis for evaluating testing techniques presented in Section 3.

Figure 1 shows the four main steps with accompanying input and output artifacts which form the DSML integration process. The DSML integration approach follows the language model-driven engineering process adopted from [12]: "first the core language model is defined to reflect all relevant domain abstractions, then the concrete syntax is defined along with the DSL's behavior, and finally the DSL is mapped to the platform/infrastructure on which the DSL runs".

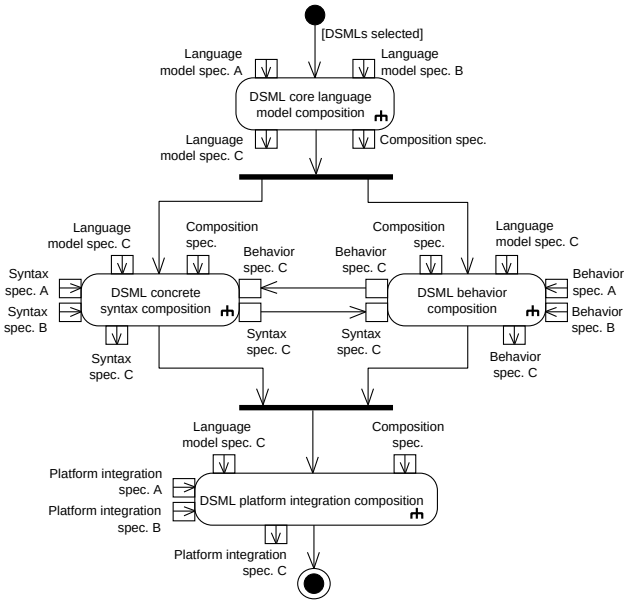


Figure 1: DSML integration process [10].

**DSML core-language-model composition.** The language model for UML-based DSMLs is defined via a MOF-compliant metamodel and via accompanying invariant constraints. As indicated by the rake in Figure 1, composing the core language model and its constraints is divided into several sub-activities (e.g., choosing the elements to be integrated, selecting a composition method).

**DSML concrete-syntax composition.** The concrete syntax serves as the user interface of a DSML. Thus, its representation must reflect the underlying concepts from the core language model as clearly as possible. The composed DSML's syntax is developed in parallel with the composed DSML behavior specification.

**DSML behavior composition.** The behavior specifications from each individual DSML need to be orchestrated to define a composition order which conforms to the integration purpose. The composition order dictates the enforcement of properties provided by each DSML and so contributes to a sound composition by, for instance, respecting functional dependencies between the concerns covered by the DSMLs.

**DSML platform-integration composition.** Platform integration is mainly driven by the feasibility and by the effort needed to compose the DSMLs at the system level. Hence, the platform integration specifications from the two source DSMLs establish the requirements on a composition technique (e.g., model-to-text (M2T) transformation specifications, glue-code artifacts in the host language).

## 3 DSML Testing Techniques

In this section, we discuss test methods which cover the four DSML composition stages introduced in Section 2. Our approach explicitly targets MOF/UML-based DSMLs and we limit the scope of testing techniques to the ones applicable according to the DSML design options identified in [13, 14]. In the context of DSML integration, the individual DSMLs represent parts of the whole system the composed DSML should cover. Our integration process focuses on pre-existing DSMLs. This means that the input DSMLs implement a specific encapsulated functionality and can be used in isolation. As these individual DSMLs are software systems on their own, we consider them independently tested using unit, component, integration, and system tests [11]. In the context of DSML composition, the input DSMLs represent subsystems and, thus, unit and component testing are already covered via the pre-existing tests of the individual DSMLs. Hence, we do only elaborate on testing methods relevant for DSML composition: integration and system testing. On the one hand, *integration testing* covers potential sources of defect which arise due to combining components. On the other hand, *system testing* is concerned with issues and behaviors that can only be exposed by testing the entire integrated system (i.e., the whole composed DSML) [11].

As can be seen in Figure 2, we perform integration testing for the individual composition steps separately. This means, the integrated artifacts of the input DSMLs are tested in combination for every phase of the composition process (Sections 3.1–3.5). If all integration tests succeed, the composed DSML will be tested system-wide (Section 3.6). Therefore, tests have to prove that all integrated artifacts work in combination and that the behavior of the composed DSML is as intended. Besides the testing process, Figure 2 displays required inputs and generated outputs of each employed testing technique as pins attached to the corresponding actions. A comment note shows, for each step, the presented testing technique which is discussed and exemplified in detail in the following sections.

### 3.1 Testing Core-Language-Model Composition

With their interwoven MDD specifications, the OMG provides the basis for an integrated model-driven architecture approach. All language-specific metamodels (such as the

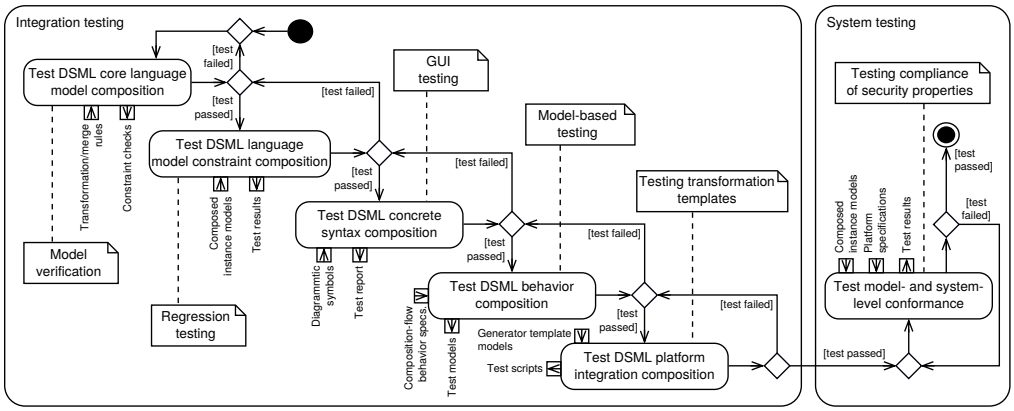


Figure 2: Integration and system testing of composed DSML artifacts.

UML) are defined via the MOF, facilitating interoperability and integration between different standards. A characteristic of MOF/UML-based DSMLs is that their core language model is formalized via diagrammatic MOF/UML constructs (e.g., UML profile definition, metamodel extension/modification; see [13]).

The method for composing core language models coming from different DSMLs is driven by the integration purpose and conceptual relationship of the two DSMLs (e.g., merge, refinement, alternative; see, e.g., [15]). For the purpose of model-to-model (M2M) transformations—regardless of the composition method used and the form of the produced output model—rule-based transformation languages are commonly employed: they are standard tool supported (e.g., Epsilon Transformation Language, ETL) and are loosely based on a common specification (i.e., MOF Query/View/Transformation, QVT). With these transformation languages, an M2M transformation is described by a set of declarative rules with optional imperative statements.

Testing the phase of the core language model integration is concerned with two software artifacts: (1) the transformation statements and (2) the composed output model. Testing the transformation language for syntactical correctness is tool supported, for instance, in the corresponding Eclipse-based ETL editor by providing syntax and error highlighting. Functional testing of transformation rules can be performed, for example, by providing test models and by predicting the expected outcome (specification- and model-based testing [16]).

To discuss the requirements of this testing phase, we look at an exemplary testing technique for M2M transformations based on model verification. By inspecting the applied transformation statements (Epsilon-based), a set of constraint expressions (defined via the Epsilon Validation Language, EVL [17]) are derived which are then evaluated in the context of the output language model of the composed DSML. With this, we want to establish whether the generated output model complies with the applied transformation rules.

To give an example, we illustrate the composition of two DSMLs for the generation of a new integrated DSML. The first source DSML models system audits (referred to as DSML `A`, hereafter) by providing abstractions for audit events and audit rules. The second input DSML (DSML `B`, hereafter) allows for modeling generic state machines. The example combines the two DSMLs into a composed DSML `C` capable of modeling a reactive distributed system with auditing support. Figure 3 illustrates the composition of these two DSMLs (DSML `A` and `B`, respectively) into the target DSML `C` via metamodel element merges. The elements `AuditEvent` and `Event` from DSMLs `A` and `B` are merged into the composed element `AuditEvent'` in the target DSML `C` (for more details see [18]). The newly generated element in the target DSML represents the union of all properties and features coming from both elements of the two source DSMLs. Figure 3 shows also the recording of transformation traces by associating corresponding merged source and target elements (via a dedicated composition trace model; see [18]).

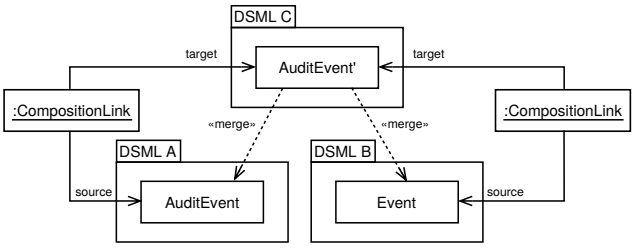


Figure 3: DSML composition via element merges [18].

Listing 1 shows an Epsilon Comparison Language (ECL [17]) statement which checks for the correspondent input elements to be merged (DSML `A` = `EventSystem`; DSML `B` = `StateMachine`). We compare the two names of the elements from both metamodels of DSMLs `A` and `B` and, if these names match the `compare` clause, the dedicated merge `rule` is invoked (shown in Listing 2).

Listing 1: Comparison of input elements to be merged.

```
1 rule Event2AuditEvent
2   match l : EventSystem!EClass
3   with r : StateMachine!EClass {
4     compare : l.name = "AuditEvent" and r.name = "Event"
5 }
```

Listing 2 merges the elements identified by the former ECL `rule` into a new `EClass` (because we base our developments on the Ecore metamodel) of DSML `C` (named `EventSystemStateMachine`). As can be seen in Listing 2, the merged element of the composed DSML `C` is named `AuditEvent'` and unions all properties and methods (`eStructuralFeatures`) as well as all inheritance relationships (`eSuperTypes`) from the source DSMLs.

Listing 2: EML merge rule for matched elements.

```
1 rule MergeAuditEvent
2   merge l : EventSystem!EClass
3   with r : StateMachine!EClass
4   into t : EventSystemStateMachine!EClass {
5     t.name = "AuditEvent'";
6     t.eStructuralFeatures ::= l.eStructuralFeatures + r.eStructuralFeatures;
7     t.eSuperTypes ::= l.eSuperTypes + r.eSuperTypes;
8 }
```

While executing the transformation process (comparison, transformation, merge), traces are recorded via a composition trace model (not displayed; for details see [18]). We use these traces to automatically generate EVL-based constraint expressions which verify the model transformation. We adopted EVL because it supports inter-model consistency checks [17]. Hence, we can base the evaluation of the correct merge behavior in the context of the output model (the composed DSML) on guards formulated for the two input models (the DSMLs to be merged).

Listing 3 shows the Epsilon Generation Language (EGL [17]) code which is used to automatically create EVL statements from the transformation traces. Therein, it is looped over the stored transformation traces (paired links of source and target elements; line 1). In our example (see Figure 3) the name of the merged source and target elements differ. Thus, we check if target and source names of the paired object links in the transformation trace do not match (line 2). If so, the corresponding EVL validation constraint is generated (saved as variable `evl`; lines 4–12) and printed to the Eclipse console (line 13).

Listing 3: EGL snippet for the creation of EVL constraints.

```
1 for (link in links) {
2   if (link.source.name <> link.target.name) {
3     if (targets->exists(t | t == link.target) == false) {
4       var evl = 'context EventSystemStateMachine!EPackage {\n' +
5                 'constraint Verify' + link.target.name + ' {\n' +
6                 'guard : ';
7       for (link in links->select(l | l.target == link.target)) {
8         evl = evl + 'EventSystem!EClass.all->exists(c | c.name = "' + link.source.name +
                  '") and\n';
9       }
10      evl = evl.substring(0, evl.length - 5) + '\n';
11      evl = evl + 'check : self.eClassifiers->exists(c | c.name = "' + link.target.name +
                '")\n' +
12            'message : "In the composed DSML there exist no merged classifier named ' +
                    link.target.name + '!"\n';
13      evl.print();
14    }
15    targets.add(link.target);
16  }
17 }
```

The so-generated EVL constraint is shown in Listing 4. As the DSML composition is based on element merges, we check for the existence of the elements named `AuditEvent` and `Event` (lines 3–4) in the corresponding DSMLs `A` and `B` (`EventSystem` and `StateMachine`). If the `guard` condition is satisfied, the `check` clause (line 5) is invoked. A successful merge implies the existence of an element named `AuditEvent'` in the target DSML `C` (`EventSystemStateMachine`); otherwise a warning message is printed to the console. The EGL code in Listing 3 could be extended to generate additional EVL constraints for validating the existence of all merged properties, methods, and inheritance relationships (see Listing 2) from the input DSMLs in the metamodel of the target DSML.

Listing 4: EVL constraint validating the metamodel merge.

```
1 context EventSystemStateMachine!EPackage {
2   constraint VerifyAuditEvent {
3     guard : EventSystem!EClass.all->exists(c | c.name = "AuditEvent") and
4             StateMachine!EClass.all->exists(c | c.name = "Event")
5     check : self.eClassifiers->exists(c | c.name = "AuditEvent'")
6     message : "In the composed DSML there exist no merged classifier named AuditEvent'!"
7   }
8 }
```

### 3.2 Testing Language-Model-Constraint Composition

The core language model of a DSML may not capture all restrictions which apply to the DSML. For the additional definition of semantics, constraints accompany a DSML's metamodel. These constraints can be expressed in various forms, for instance, via explicit expressions, via code annotations, or via constraining model transformations [13]. At the time of DSML composition, language model constraints must be adapted accordingly: Constraints can be rendered more restrictive, they can be declared as refinements or as extensions to existing constraints, or they can establish explicit and navigable links between metamodels [10]. The Object Constraint Language (OCL [19]) is commonly employed to describe explicit expressions (e.g., invariant constraints) on MOF/UML models.

We extend the example from the former Section 3.1 where a metaelement merge is applied for composing DSML `A` (`AuditEvent` element) and `B` (`Event` element) into DSML `C` (`AuditEvent'` element). We assume that both elements of the two input DSMLs are constrained by the OCL expressions defined in Listing 5. In case of an `AuditEvent` (from DSML `A`), a signal containing audit information is published for subscribed consumers to be received and processed (for more details see [20]). The first invariant constraint in Listing 5 requests that the data of the published signal must not be empty (lines 1–2). The second constraint concerns the `Event` element of DSML `B` and states that its name must be set (lines 4–5). The merge of the two DSMLs (as described in Section 3.1) implies an OCL refinement as shown in the third constraint in Listing 5 (lines 7–9). In essence, the metamodel merge requires that both individual constraints are combined (i.e., the constraint is more restrictive) and are always *true* for the `AuditEvent'` element.

Listing 5: Language model OCL expression refinement.

```
1 context AuditEvent inv:
2   self.publish->forAll(signal | signal.data->notEmpty())
3
4 context Event inv:
5   self.name->notEmpty()
6
7 context AuditEvent' inv:
8   self.publish->forAll(signal | signal.data->notEmpty()) and
9   self.name->notEmpty()
```

For testing the refined language model constraints, we use a variant of a regression test. Regression tests are used

to validate modified software parts and to ensure that no new errors are introduced into previously tested code—a technique frequently used during maintenance of evolving software [21]. As we assume that the constraints from the two input DSMLs are already tested (see Section 1), these constraints are considered free of errors when being evaluated over instance models of DSMLs `A` and `B`, respectively. We can now test the refined constraint for DSML `C` by providing an instance model of the composed DSML and by predicting the return value of the evaluated OCL statement. The test-case oracle must match the combined return values of the individual constraints and the evaluation must not abort due to any syntax errors. As invariant conditions must always hold for the system being modeled [19], the predicted return value must always be *true* when evaluating a composed DSML instance model. If a tested constraint returns *false* or aborts, either the model itself or the evaluated constraint expression are erroneous. But as testing the DSML core language model composition precedes constraint composition tests, failure in the language model composition and instantiation can be excluded.

By allowing a *false* return value when evaluating a constraint, the invariant condition's return value in our example must match the values given in Table 1 (due to the logical conjunction of the individual constraints in the composed OCL statement; line 8 in Listing 5).

Table 1: Return-value matrix for negative tests.

| DSML A constraint | *true* | *true* | *false* | *false* |
|---|---|---|---|---|
| DSML B constraint | *true* | *false* | *true* | *false* |
| DSML C constraint | *true* | *false* | *false* | *false* |

This setup can be used for expressing negative tests [11] on the composed language-model constraints. Figure 4 shows an excerpt from an exemplary, non-conforming instance model of DSML `C`. As the DSML C is composed via a full merge of DSMLs `A` and `B` (for details see [18]), their OCL constraints can be evaluated in the context of the displayed instance model. The evaluation of the constraint from DSML `A` (line 2 in Listing 5) returns *false* because the `data` attribute of the `Signal` occurrence is empty. The constraint from DSML `B` (line 5) returns *true* as the object of type `AuditEvent'` has a non-empty `name` attribute. Finally, evaluating the constraint from DSML `C` (lines 8–9) will result in a *false* return value (because of the logical conjunction of the two first constraints). Therefore, the negative test succeeds as all return values match the truth table (column 4 in Table 1).

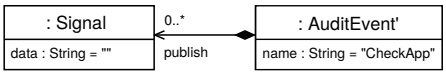


Figure 4: Non-conforming instance model of DSML C.

### 3.3 Testing Concrete-Syntax Composition

The concrete syntax definition of a DSML visualizes the abstract elements coming from the core language model for the end-user to be able to interact with the language [12]. It so visualizes the interface presented to the user. A DSML can have multiple concrete syntaxes and these can have various formats, for instance, text/table-based or graphical [14].

A MOF/UML-based DSML extends the UML symbol vocabulary with domain-specific visual notations (either via native methods defined in the UML specification [5] or via custom visual extensions [14]). Here, we consider only diagrammatic concrete syntaxes as the most widely used notational option for MOF/UML-based DSMLs (as their primary application area is the creation of models [5]).

As an example, we introduce a testing technique for visual DSML concrete-syntax composition which is agnostic about the DSML integration method used (e.g., syntax extension or integration; see [10]). We present a testing method based on the GUI which suits MOF/UML-based DSMLs and works with any diagrammatic model editor. In this context, we use an image-based testing approach to check for the symbol composition of two integrated concrete syntaxes of DSMLs. Hence, we test for the visual application of symbols coming from two different DSMLs and their usage in an integrated model.

For our example, we utilize a small symbol set taken from two existing MOF/UML-based DSMLs. These two DSMLs are both defined via UML profiles with accompanying stereotype and icon specifications (see the symbols on the right-hand side of Figure 5). The first DSML (`«profile» SOF`) supports the modeling of confidentiality and integrity properties for important objects that are passed between different participants in business processes [22]. The second DSML (`«profile» SecurityAudit`) is concerned with the modeling of system audits and was already introduced in Section 3.1 (DSML `A` [20]).

These two DSMLs are integrated in a way that the symbols from both DSMLs can be used in combination (for details see [10]). If we extend a UML model with properties from these DSMLs, we want to test if the concrete syntax represents the defined specifications. For instance, in the UML activity diagram on the left-hand side of Figure 5, the action *Check application form* requests features from both DSMLs (a secure input pin and object flow as well as audit support); i.e., both profiles must be applied in order for the integration to be fulfilled. These demands must be reflected via the symbols defined in each individual DSML (i.e., the *key* and *AES* symbols; see Figure 5). If any of these symbols is missing, the integration of the concrete syntax will be considered incorrect because it does not mirror the concrete-syntax specification of the composed DSML.

To test for the occurrence of symbol sets coming from different DSMLs, we deploy an image-based GUI testing technique. The output of the test is (1) a boolean flag if all requested DSML symbols are applied in the diagram and (2) a model in which the occurrences of the symbols found

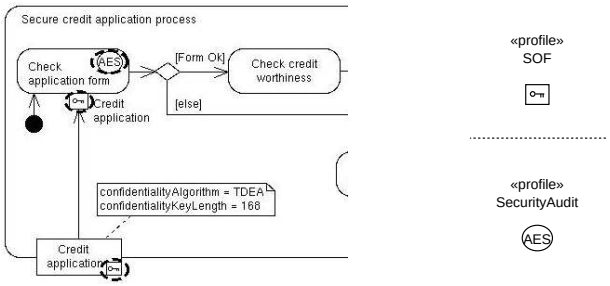


Figure 5: Image-based concrete syntax testing.

are marked (see the UML activity diagram in Figure 5). For this purpose, we utilize a GUI test automation tool named Ranorex [23] which is based on the .NET framework and which provides, amongst others, image-based testing capabilities. Hence, our testing technique is independent from any modeling tool support and works with any model editor (i.e., we do not need to trace tool- and vendor-specific APIs for object recognition and the programmatic identification of user interface elements).

Listing 6 shows an excerpt from the C# code to identify the *key* and *AES* symbols in a given model. First, we save a snapshot of the diagram representation under test taken from the employed modeling tool (lines 1–2). Then, we check whether the two symbols from Figure 5 are contained in this snapshot (lines 3–4). If the image comparison method does not find the right number of symbol occurrences, an error is logged in the test report (lines 7–8). Additionally, lines 10–19 draw a dotted ellipse around all found occurrences of the two symbols and append the image (Figure 5) to the test report. In Listing 6, we only check whether both symbols are at least used once in the diagram (lines 7–8). This test can, of course, be extended to more sophisticated checks (e.g., only an even number of *key* symbols is allowed or, when multi-model tests are employed, whether the audit-specific elements are tagged in all model representations, for instance, in a UML activity and corresponding class diagram).

Listing 6: C# snippet for integrated GUI testing in Ranorex.

```
1  var elem = repo.SecureCreditAppExampleWindows.ElementModel;
2  var model = elem.CaptureCompressedImage().Image;
3  var key_matches = Imaging.Find(model, ElementModel_key1);
4  var aes_matches = Imaging.Find(model, ElementModel_aes1);
5  var matches = key_matches; matches.AddRange(aes_matches);
6
7  if (key_matches.Count == 0 || aes_matches.Count == 0)
8    Report.Log(ReportLevel.Error, "The model does not contain both integrated symbols!");
9
10 using (var g = Graphics.FromImage(model)) {
11   var pen = new Pen(Color.Black, 3);
12   pen.DashPattern = new float[]{3.5F, 1.5F};
13   foreach (var match in matches) {
14     var rect = new Rectangle(match.Location, match.Size);
15     rect.Inflate(8, 8);
16     g.DrawEllipse(pen, rect);
17   }
18 }
19 Report.LogData(ReportLevel.Info, "FoundSymbols", model);
```

### 3.4 Testing Behavior-Specification Composition

The behavior specification of a DSML defines its dynamic characteristics at runtime. It can be defined in multiple ways, for instance, via control-flow diagrams (e.g., UML state machines), via process definitions (e.g., WS-BPEL), or via source code statements [10]. For the integration of DSMLs, the behavior specification is especially important for establishing a composition order between the individual DSMLs when deployed in combination. The behavior composition is used to define a timely order of event occurrences for properties of the integrated DSML.

Figure 6 shows an exemplary behavior composition of an integrated MOF/UML-based DSML via a UML state machine. The two merged DSMLs are defined via UML profiles and are already known: (1) the «SOF» profile (with a SOA-based extension: profile «SOF:Services»; see [22] for details) and (2) the «SecurityAudit» profile (see [20] for details).

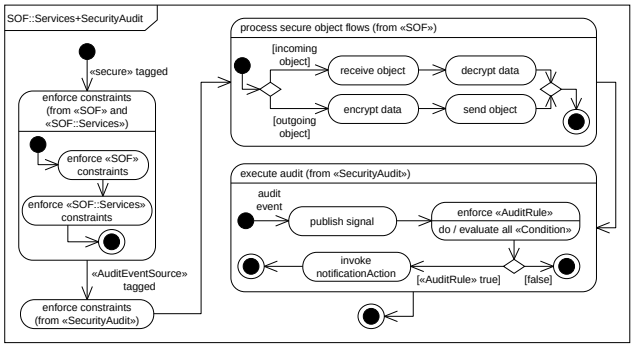


Figure 6: Integrated DSML behavior specification [10].

Here, we discuss a model-based testing technique for validating the composition order for the integrated DSML as defined in Figure 6. For this purpose, we utilize the UML Testing Profile (UTP [24]) for defining test aspects. We have opted for the UTP because it seamlessly integrates with our DSML specifications (all are UML profiles). Furthermore, the UTP is eligible to map models onto executable testing platforms (e.g., JUnit) [24].

Figure 7 shows a class diagram of relevant elements of the composed DSML under test. For our example, we consider a software subsystem managing credit applications of bank customers (see also Figure 5) which is integrated into a larger company-wide ERP system (packages CreditSystem and ERP-System). The credit system makes use of a SecurityLayer package for data confidentiality functionality. Interactions with the system are stored in an audit trail (package «AuditSystem») which is designed for distributed event-based systems. Our example consists of one test case (see Figure 9) which is used to check the correctness of the composition order as specified in Figure 6: After receiving a secured object flow, data are decrypted and processed; then an audit event is triggered.

Figure 8 shows the CreditApplicationTest package which contains all elements necessary to fully specify our test. We define emulator components for the test simulation as well as one test case. The test case (incompleteAppForm) checks the system behavior of handling incomplete credit application forms (this event should

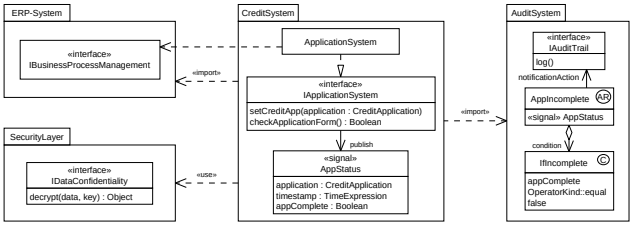


Figure 7: Elements of the composed DSML to be tested.

be logged; see Figure 7). We use this example to demonstrate the validation of the DSML composition order.

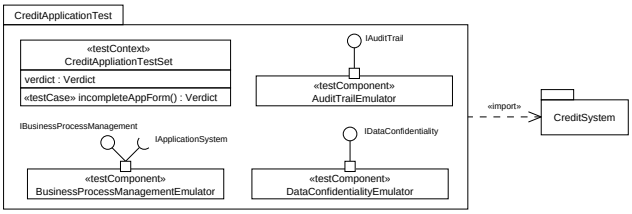


Figure 8: The `CreditApplicationTest` package.

The test case (Figure 9) is specified in terms of interactions between the test components (Figure 8). The system under test (SUT; `ApplicationSystem`) is stimulated via its public interface operations and signals by the test components. For testing the right composition order, the test case `incompleteAppForm` (Figure 9) must pass successfully. As the security-related functionality is inherently called at the time of receiving a credit application (`setCreditApp(app)`), the audited method `checkApplicationForm()` must be invoked afterwards. Although we do not check for successful audit logging, these two message sequences are sufficient for testing the composition order.

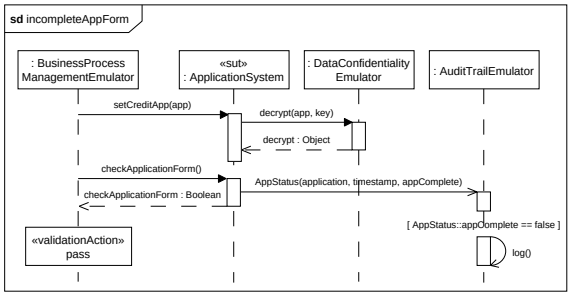


Figure 9: Test case for the DSML composition order.

### 3.5 Testing Platform-Integration Composition

The platform integration phase maps the various DSML artifacts to a dedicated software platform. This is achieved by defining, for instance, M2T transformations to convert a model into executable software artifacts (e.g., Java source code) or configuration specifications (e.g., web-service definitions) [14]. Approaches for the platform implementation range from pipelining, piggybacking, language extension, to front-end integration (see, e.g., [10]).

In MDD, M2T transformations are commonly applied using generator templates [25]. A composition of two DSMLs requires the adaption of these templates. In this section, we discuss a test technique applicable for composed M2T generator templates presented in [18]. Therein, M2T templates are considered as first-class models and, by reusing transformation traces, the approach enables syntactical template rewriting (see Figure 10). The representation of the M2T generator templates as models allows for an M2M transformation of these models (higher-order transformation, HOT [26]). The higher-order rewriting of M2T templates facilitates a platform integration composition via generator adaptations of DSMLs `A` and `B` by allowing for a certain level of reuse of each DSML's platform artifacts.

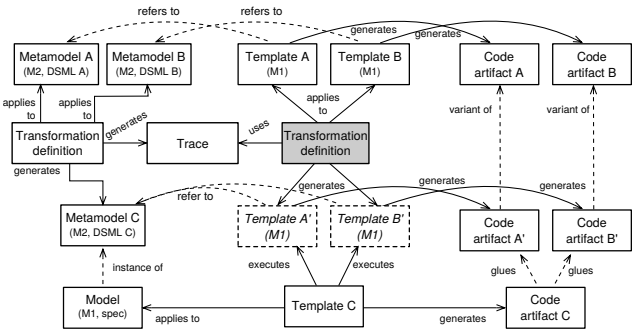


Figure 10: M2T template rewriting for DSML integration.

Listing 7 shows an example code snippet of an EGL template (from the system audit DSML introduced in Section 3.1; see [20]). By considering the composition scenario from Figure 3, the iterator variable `ae` must be modified to be of type `AuditEvent'` (line 1).

Listing 7: EGL code snippet with typed iterator.

```
1 [% for (ae : AuditEvent in EventSystem.auditEvents) {
2   for (signal in ae.publish) {
3     out.println('private Signal ' + signal.name + ';');
4   }
5 } %]
```

Rewriting the template shown in Listing 7 is achieved by creating a model representation of the EGL code and by applying M2M transformations (Listing 8). These ETL-based higher-order rewrite rules are automatically generated from the composition traces available for the composed language model (see Figure 10 and [18]). Listing 8 shows the so-created rewrite `rule` for altering the `type` properties of `ModelElementType` instances (from `AuditEvent` to `AuditEvent'`).

Listing 8: ETL higher-order rewrite rule.

```
1 rule renameAuditEvent2AuditEvent'
2   transform s : eglIn!ModelElementType
3   to t : eglOut!ModelElementType
4   extends Type {
5     guard : s.type == "AuditEvent"
6     t.type = "AuditEvent'";
7   }
```

Applying this `rule` to the EGL model results in a rewritten EGL model which is corrected for the changed

type name. Line 3 of Listing 9 shows that the `type` of the iterator variable named `ae` was effectively changed to `AuditEvent'`.

Listing 9: Rewritten EGL model representation.

```
1 <statements xsi:type="dom:ForStatement">
2   <iterator name="ae">
3     <type xsi:type="dom:ModelElementType" type="AuditEvent'"/>
4   </iterator>
5   <iterated xsi:type="dom:PropertyCallExpression" property="auditEvents">
6     <target xsi:type="dom:NameExpression" name="EventSystem"/>
7   </iterated>
```

To validate the rewriting of EGL models, we apply EUnit tests [17] to the process of ETL model transformations (Listing 10). After an EGL model rewrite (line 3), the output model (`eglOut`) is tested against the expected output model (`eglExp`). If any inconsistencies emerged from the model transformation, the test would fail signalling the error message shown in line 4.

Listing 10: Testing rewritten EGL model (EUnit).

```
1 @test
2 operation testTemplateRewrite() {
3   runTarget("templateRewrite");
4   assertEqualModels("Rewritten EGL model differs from expected!", "eglExp", "eglOut");
5 }
```

### 3.6 Testing Model- and System-Level Conformance

In this section, we exemplify the system testing of model- and platform-level compliance of security properties for a complete MDD transformation process. Thus, we rely on consistency checks between output artifacts at the modeling and at the system levels (functional testing [11]) rather than assessing every single DSML composition step.

Figure 11 shows the involved MDD-related artifacts and transformations of our example. At the modeling level, we compose two DSMLs to specify secure object flows for SOAs (packages `SecureObjectFlows` and `Services`; for more details see [22]). We provide M2M transformations to generate an intermediate object model (IOM) representation out of the merged `SOF::Services` package. The IOM structure can be mapped to multiple host platforms via M2T transformations. For our example, we generate web-service specification documents (i.e., WS-BPEL, WSDL, WS-SecurityPolicy).

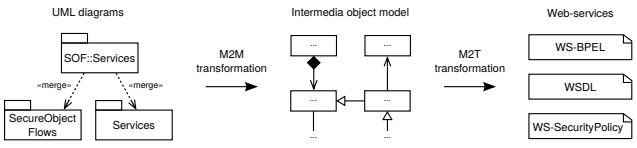


Figure 11: MDD artifacts and transformations.

Figure 12 shows the equivalences between concepts of the XMI-represented UML diagrams at the modeling level and the XML-expressed WS-* specifications at the system level. In this excerpt, the `CallOperationAction` `submitApplication` is mapped to a WSDL `operation` (`a`). The `argument` `content` represents an object of type string which should be secured. In the WS-* specification, this `content` element is identified in a SOAP message via an XPath expression (`b`). In the UML activity diagram, the `content` element is tagged via a stereotype (`SOF:secure`; `c`). This stereotype specifies an `integrityAlgorithm` attribute referencing a class which implements a specific cryptographic hash function (here: `Sha1`; `d`). The WS-SecurityPolicy standard [27] groups security-related properties into algorithm suites (here: `Basic192`). An attribute of the hash function element in the UML activity diagram references this algorithm suite (`e`) and it is mapped to the WS-* specification (`f`). The policy, which contains the XPath expression identifying the elements to be signed and the security-related algorithm suite, is referenced from the `input` object of the corresponding `operation` in the WS-* specification (`g`). In this way, security-enhanced UML diagrams at the modeling level are turned into WS-* specifications at the system level which can be deployed in a runtime engine (e.g., Apache ODE).

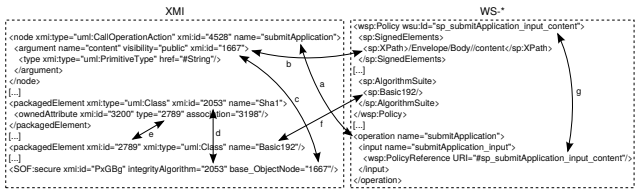


Figure 12: Equivalences between XMI and WS-* concepts.

Listing 11 shows an XQuery [28] snippet to test the output XML documents for their conformance. First, we loop over all «`secure`» stereotypes in the UML diagram (line 1) and declare a couple of helper variables. In lines 2–4, the «`secure`» tagged `argument` object node with its applied integrity algorithm and referenced algorithm suite are stored. Lines 5–6 find the corresponding WSDL `operation` and `Policy` nodes according to the UML `CallOperationAction`. We define two test cases (lines 7–8): (1) The XPath expression in the WS-* specification must identify the correct SOAP element by matching the name of the UML «`secure`» stereotyped object node; (2) the security property of the «`secure`» stereotyped object node must be enforced via the WS-* algorithm suite (i.e., declared algorithm suites in both XML definitions must be identical). The tests pass if the security definitions stated in the XMI and WS-* specifications conform to each other. If the tests fail, the transformation of security properties from the modeling- to the system-level is signalled incorrect.

Listing 11: XQuery snippet for XML conformance testing.

```
1  for $uml_sec in doc("uml.xmi")//SOF:secure
2    let $uml_arg := doc("uml.xmi")//node/argument[@xmi:id eq $uml_sec/@base_ObjectNode],
3        $uml_int := doc("uml.xmi")//packagedElement[@xmi:id eq $uml_sec/@integrityAlgorithm
                ],
4        $uml_alg := doc("uml.xmi")//packagedElement[@xmi:id eq $uml_int/ownedAttribute/
                @type],
5        $ws_op := doc("ws.xml")//operation[@name eq $uml_arg/../@name],
6        $ws_pol := doc("ws.xml")//wsp:Policy[@wsu:Id eq substring($ws_op/input/wsp:
                PolicyReference/@URI, 2)],
7        $test_path := ends-with($ws_pol/sp:SignedElements/sp:XPath, $uml_arg/@name),
8        $test_alg := $uml_alg/@name eq string(node-name($ws_pol/sp:AlgorithmSuite/*))
9    return if ($test_path = true() and $test_alg = true())
10         then concat("Test SOF:secure[@id=", $uml_sec/@xmi:id, "] passed!")
11         else concat("Test SOF:secure[@id=", $uml_sec/@xmi:id, "] failed!")
```

## 4 Discussion

The testing approach for language integration presented in this paper explicitly targets MOF/UML-based DSMLs. Nevertheless, the overall testing process for DSML integration applies to non-MOF/UML-based DSMLs, as well. We concentrate on testing DSML integration using dynamic techniques to validate their composition (inputs, outputs, transformations).

By providing testing techniques for the different phases of the MOF/UML-based DSML integration process, we contribute to preventing common DSML integration issues pertaining to *constraint adaptation* (Section 3.2), *symbol composition* (Section 3.3), *composition order* (Section 3.4), and *generator adaptation* (Section 3.5; for details see [10]).

All software prototypes—except the image-based concrete-syntax testing with Ranorex—are implemented using the Eclipse Modeling Framework, such as, the Ecore metamodel, the Epsilon language family, or the MDT OCL console. Furthermore, testing techniques can be transferred to other technologies, for instance, different metamodel implementations, transformation and constraint languages, modeling tools, or host-platform representations.

All DSMLs used for the composition test examples were developed by the author of this paper. Although the DSMLs were not built for this specific purpose, a methodical and technological bias may exist. This might have also influenced the selection of testing techniques.

## 5 Related Work

We identified research related to each testing technique presented above, falling into the categories of model verification, regression and GUI testing as well as model-based and transformation testing.

**Model verification.** Approaches to model verification resemble each other in the way models are translated into the native language of some model verification tool. For one, [29] presents a framework supporting formal verification of UML class, state, and communication diagrams. UML models are transformed into a formal specification language which supports model checking using linear temporal logic. Our model verification differs from these approaches as the objective is not to verify any correctness properties of finite-state systems, but to validate the correspondence of an output model with given composition specifications.

**Regression testing.** Our approach for testing the evolved language model constraints in the process of DSML composition is a variant of regression testing [21]. It differs from related work by testing OCL refinements at the time of integrating DSMLs.

**GUI testing.** In this paper, we employ image-based testing of the UML concrete syntax. This technique facilitates model evolution as it does not rely on object recognition strategies for identifying user interface elements [23].

**Model-based testing.** The UTP is frequently utilized for model-based testing. One related approach [30] presents a technique to automatically generate UTP models from systems described in the UML via QVT transformations. In a final step, these UTP models are transformed into JUnit tests. This generative approach can be combined with ours by running the QVT transformations for creating UTP models against the DSML behavior models under composition.

**Model transformation testing.** Various approaches exist to test model transformations including directly testing executable output models/source code, defining pre- and post-conditions for the transformation, establishing model-transformation contracts, and comparing generated with expected output models/files [31]. As for M2T transformations, testing the created artifacts (e.g., source code) is commonly employed via platform-specific frameworks (e.g., JUnit). Our approach differs from these by processing model representations of generator templates to test M2T transformations.

## 6 Conclusion

This paper presents an approach towards a definition of a testing process for each individual step of composing DSML artifacts. We discuss different methods and provide examples of integration and system testing for security-related, MOF/UML-based DSMLs. We base our work on existing DSMLs and so contribute to understand testing requirements for model-based software evolution in general as well as for DSML integration in particular.

As future work, we will refine and extend the presented techniques, to cover, for instance, platform-specific behavior tests via transformations from UTP models into JUnit code artifacts (to name just one example). We will review additional test techniques for inclusion to cover further DSML integration issues. Moreover, we will evaluate how our software prototypes can be transferred to alternative software technologies and platforms.

## 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

[1] T. Stahl and M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[2] B. Selic, "The Pragmatics of Model-driven Development," *IEEE Software*, vol. 20, pp. 19–25, Sept. 2003.

[3] M. Mernik, J. Heering, and A. Sloane, "When and How to Develop Domain-specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[4] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-driven Software Development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

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

[6] D. Basin, J. Doser, and T. Lodderstedt, "Model Driven Security: From UML Models to Access Control Infrastructures," *ACM Transactions on Software Engineering and Methodology*, vol. 15, pp. 39–91, Jan. 2006.

[7] Object Management Group, "OMG Meta Object Facility (MOF) Core Specification." Available at: `http://www.omg.org/spec/MOF`, August 2011. Version 2.4.1, formal/2011-08-07.

[8] M. Godfrey and D. German, "The Past, Present, and Future of Software Evolution," in *Proc. of Frontiers of Software Maintenance*, pp. 129–138, 2008.

[9] A. Vallecillo, "On the Combination of Domain Specific Modeling Languages," in *Proc. of the 6th European Conference on Modelling Foundations and Applications*, vol. 6138 of *LNCS*, pp. 305–320, Springer, 2010.

[10] B. Hoisl, M. Strembeck, and S. Sobernig, "Towards a Systematic Integration of MOF/UML-based Domain-specific Modeling Languages," in *Proc. of the 16th IASTED International Conference on Software Engineering and Applications*, pp. 337–344, ACTA Press, 2012.

[11] B. Beizer, *Software Testing Techniques*. Van Nostrand Reinhold, 2nd ed., 1990.

[12] M. Strembeck and U. Zdun, "An Approach for the Systematic Development of Domain-Specific Languages," *Software: Practice and Experience*, vol. 39, no. 15, pp. 1253–1292, 2009.

[13] B. Hoisl, S. Sobernig, S. Schefer-Wenzl, M. Strembeck, and A. Baumgrass, "Design Decisions for UML and MOF based Domain-specific Language Models: Some Lessons Learned," in *Proc. of the 2nd Workshop on Process-based approaches for Model-Driven Engineering*, pp. 303–314, 2012.

[14] B. Hoisl, S. Sobernig, S. Schefer-Wenzl, M. Strembeck, and A. Baumgrass, "A Catalog of Reusable Design Decisions for Developing UML- and MOF-based Domain-Specific Modeling Languages." Available at: `http://epub.wu.ac.at/3578`, 2012. Technical Reports / Institute for Information Systems and New Media (WU Vienna), 2012/01.

[15] M. Emerson and J. Sztipanovits, "Techniques for Metamodel Composition," in *Proc. of the 6th OOPSLA Workshop on Domain-Specific Modeling*, pp. 123–139, ACM, 2006.

[16] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in *Proc. of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pp. 31–36, ACM, 2007.

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

[18] B. Hoisl, S. Sobernig, and M. Strembeck, "Higher-Order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages," in *Proc. of the 1st International Conference on Model-Driven Engineering and Software Development*, forthcoming.

[19] Object Management Group, "OMG Object Constraint Language (OCL) Specification." Available at: `http://www.omg.org/spec/OCL`, January 2012. Version 2.3.1, formal/2012-01-01.

[20] B. Hoisl and M. Strembeck, "A UML Extension for the Model-driven Specification of Audit Rules," in *Proc. of the 2nd International Workshop on Information Systems Security Engineering*, pp. 16–30, Springer, 2012.

[21] M. J. Harrold, "Testing Evolving Software," *Journal of Systems and Software*, vol. 47, no. 2–3, pp. 173–181, 1999.

[22] B. Hoisl, S. Sobernig, and M. Strembeck, "Modeling and Enforcing Secure Object Flows in Process-driven SOAs: An Integrated Model-driven Approach," *Software and Systems Modeling*, DOI: 10.1007/s10270-012-0263-y, 2012.

[23] B. Peischl, R. Ramler, T. Ziebermayr, S. Mohacsi, and C. Preschern, "Requirements and Solutions for Tool Integration in Software Test Automation," in *Proc. of the 3rd International Conference on Advances in System Testing and Validation Lifecycle*, pp. 71–77, 2011.

[24] Object Management Group, "UML Testing Profile (UTP)." Available at: `http://www.omg.org/spec/UTP`, April 2012. Version 1.1, formal/2012-04-01.

[25] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," in *Proc. of the OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[26] M. Tisi, J. Cabot, and F. Jouault, "Improving Higher-Order Transformations Support in ATL," in *Proc. of the 3rd International Conference on Theory and Practice of Model Transformations*, vol. 6142 of *LNCS*, pp. 215–229, Springer, 2010.

[27] Organization for the Advancement of Structured Information Standards, "WS-SecurityPolicy 1.3." Available at: `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/os/ws-securitypolicy-1.3-spec-os.pdf`, 2009.

[28] World Wide Web Consortium, "XQuery 1.0: An XML Query Language (Second Edition)." Available at: `http://www.w3.org/TR/xquery/`, 2010.

[29] P. Gagnonand, F. Mokhati, and M. Badri, "Applying Model Checking to Concurrent UML Models," *Journal of Object Technology*, vol. 7, no. 1, pp. 59–84, 2008.

[30] B. P. Lamancha, P. R. Mateo, I. R. de Guzmán, M. P. Usaola, and M. P. Velthius, "Automated Model-based Testing using the UML Testing Profile and QVT," in *Proc. of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pp. 6:1–6:10, ACM, 2009.

[31] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu, "Barriers to Systematic Model Transformation Testing," *Communications of the ACM*, vol. 53, no. 6, pp. 139–143, 2010.