WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

EFMD
EQUIS
ACCREDITED

DOKTORAT DER SOZIAL- UND WIRTSCHAFTSWISSENSCHAFTEN

1. Beurteilerin/1. Beurteiler: **Prof. Mark Strembeck**

2. Beurteilerin/2. Beurteiler: **Prof. Gustaf Neumann**

Eingereicht am: ________________________________

WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

EFMD
EQUIS
ACCREDITED

Titel der Dissertation:

Integration and Test of MOF/UML-based Domain-specific Modeling Languages

Dissertation zur Erlangung des akademischen Grades

einer Doktorin/eines Doktors

der Sozial- und Wirtschaftswissenschaften an der Wirtschaftsuniversität Wien

eingereicht bei

1. Beurteilerin/1. Beurteiler: **Prof. Mark Strembeck**

2. Beurteilerin/2. Beurteiler: **Prof. Gustaf Neumann**

von **Bernhard Hoisl**

Fachgebiet: **Wirtschaftsinformatik**

Wien, im **November 2014**

Ich versichere:

1. dass ich die Dissertation selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

2. dass ich diese Dissertation bisher weder im In- noch im Ausland (einer Beurteilerin/ einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

3. dass dieses Exemplar mit der beurteilten Arbeit übereinstimmt.

Datum ____________________ Unterschrift ______________________________

Abstract

In model-driven development (MDD), domain-specific modeling languages (DSMLs) are used as tailor-made software languages targeting dedicated application domains. Due to the narrow domain coverage of DSMLs, demands to integrate their individual functionality into a consolidated DSML arise (e.g., developing a software product combining two or more pre-existing DSMLs). However, in order to realize the benefits of integrated DSMLs, it must be ensured that the integrated DSML is correctly implemented and behaves as specified. To support the integration and the test of DSMLs, this thesis presents an approach targeting the Meta Object Facility (MOF) and the Unified Modeling Language (UML)—a metamodeling infrastructure frequently employed for the MDD of software systems. The integration of DSMLs is based on a rewriting technique for model-to-text (M2T) transformations. This method allows for the reuse as well as for the automatic refactoring of M2T transformation templates to fix important syntactical mismatches between templates and the integrated DSML. To test an integrated DSML, scenarios are used to define domain requirements on an abstract level (via structured text descriptions). In a subsequent step, executable scenario tests are derived from the requirements-level scenarios. These executable scenario specifications are then employed to test the integrated DSML for compliance with corresponding domain requirements. Empirical evaluations of the approach (case studies, controlled experiment) demonstrate its successful application, collect evidence for its usefulness, and quantify its benefits. The integrated proof-of-concept implementations build on the Eclipse Modeling Framework (EMF), making use of and extending well-known Eclipse-based projects. All accompanying developments are placed into the public domain as free/libre open source software.

Within the framework of this thesis, research results were originally published as individual contributions (workshop, conference, and journal articles). All research contributions are results of applying a design science research approach.

Zusammenfassung

In der modellgetriebenen Softwareentwicklung (engl. model-driven development; MDD) werden domänenspezifische Sprachen (engl. domain-specific modeling languages; DSMLs) eingesetzt, die für eine bestimmte Anwendungsdomäne konzipiert wurden. DSMLs decken einen engen Anwendungsbereich ab, weshalb der Bedarf besteht, die Funktionalität verschiedener DSMLs miteinander zu integrieren (z.B. um ein Softwareprodukt aus der Kombination von zwei oder mehreren bestehenden DSMLs zu entwickeln). Um die Vorteile integrierter DSMLs zu realisieren, muss sichergestellt werden, dass die integrierte DSML korrekt implementiert wurde und sich wie spezifiziert verhält. Die vorliegende Arbeit präsentiert einen Ansatz, welcher die Integration und den Test von DSMLs unterstützt, die auf den Spezifikationen der Meta Object Facility (MOF) und der Unified Modeling Language (UML) beruhen – eine Metamodellierungsinfrastruktur, die häufig für die modellgetriebene Softwareentwicklung eingesetzt wird. Die Integration von DSMLs basiert auf einer Technik zur Adaption von Modell-zu-Text (engl. model-to-text; M2T) Transformationen. Diese Methode erlaubt die Wiederverwendung und die automatisierte Bearbeitung von M2T Transformationsvorlagen, um syntaktische Unterschiede zwischen den Vorlagen und der integrierten DSML zu beheben. Für das Testen einer integrierten DSML werden Szenarien eingesetzt, welche die Anforderungen an eine Domäne auf einer abstrakten Ebene beschreiben (mithilfe strukturierten Texts). Anschließend werden von diesen Szenarienbeschreibungen auf Ebene der Anforderungen ausführbare Szenarientests abgeleitet. Diese ausführbaren Szenarienspezifikationen überprüfen die Konformität der integrierten DSML mit den entsprechenden Anforderungen der Domäne. Durch empirische Evaluierungen des Ansatzes (Fallstudien, kontrolliertes Experiment) werden erfolgreiche Anwendungen demonstriert, Belege zum Nachweis der Nützlichkeit gesammelt und die Vorteile quantifiziert. Die integrierten prototypischen Implementierungen bauen auf das Eclipse Modeling Framework (EMF) auf und benutzen sowie erweitern bekannte auf Eclipse basierende Projekte. Der Quelltext der begleitenden Softwareentwicklungen ist offengelegt und frei verfügbar.

Die im Rahmen dieser Arbeit entstandenen Forschungsergebnisse sind ursprünglich als individuelle Beiträge publiziert worden (Workshop-, Konferenz- und Zeitschriftenartikel). Alle Beiträge resultieren aus der Anwendung eines Design Science Forschungsansatzes.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

In recent years, model-driven development (MDD) emerged as a software engineering technique for the systematic specification of software systems (see, e.g., [7, 78, 131]). The modeling of domain artifacts helps to understand complex problems and potential solutions by raising the level of abstraction in the software engineering process—comparable to high-level programming languages abstracting from a system's instruction set architecture (i.e. machine code; see, e.g., [122, 123]). Generally, in MDD, graphical or textual models residing on a higher-level of abstraction (and thereby omitting computational details) are transformed into models rendering implementation details of a software system more concrete (e.g., to be directly executable via a specific software platform). These model transformations are an integral part of MDD approaches and benefit from a high degree of automation, such as, tool-supported code generation (see, e.g., [123]).

In the context of MDD, domain-specific (modeling) languages (DSLs/DSMLs) are special-purpose (modeling) languages tailored to a particular application domain. DSMLs are a special kind of DSLs providing end users with a graphical/diagrammatic concrete syntax—in contrast to textual or form-/table-based DSLs (see, e.g., [73, 84, 130, 137, 182]). The development of DSMLs based on the Meta Object Facility (MOF [100]) and/or the Unified Modeling Language (UML [96]) are frequently applied in MDD (see, e.g., [3, 25, 26, 41, 65, 66, 114]). A MOF/UML-based DSML is characterized by utilizing the MOF/UML specifications where possible and by extending their definitions where necessary. Thereby, DSMLs that are based on the MOF/UML may directly benefit from standardized modeling extensions (see, e.g., [93, 97, 98, 99]), maintenance through the Object Management Group (OMG), and a variety of corresponding software tools (e.g., Sparx Systems Enterprise Architect, IBM Rational Software Architect, Eclipse Model Development Tools etc.). We have chosen the MOF/UML as the basis for our work because of its metamodeling infrastructure (e.g., the layers of abstraction resembles our MDD approach), its extension possibilities (e.g., native, MOF-based), the wide range of tool support available (e.g., based on Eclipse), and because we build on former work which uses the same metamodeling environment (see, e.g., [135]).

The advantages of DSMLs include reduced development times for DSML-based software products, an improved time-to-market, as well as reductions in development and delivery costs; for example, for developer or customer trainings (see, e.g., [6]). However, the development of a DSML and corresponding tool support most often requires substantial efforts that add to the overall costs of the underlying software development project. Thus, benefits of a domain-specific development approach usually realize over time (see, e.g., [74, 175]).

As a result, the costs of DSML development are strong drivers for reusing DSMLs as design artifacts, both during the life cycle of a single software product and for multiple software products (see, e.g., [74, 175]). To develop a software product using two or more pre-existing DSMLs, with each DSML defining a subsystem of the product, integrating the corresponding DSMLs into a new consolidated DSML is an important design option (see, e.g., [161]). Consider, for example, modeling the billing domain in a power supply company which covers company-specific accounting and branch-specific schedule management. Provided that compatible DSMLs for both tasks (i.e., accounting and schedule management) are available (e.g., based on the same metamodeling infrastructure), their integration is a feasible strategy, for example, via product line techniques (see, e.g., [175]).

However, in order to realize the benefits of (integrated) DSMLs, we must ensure that the (integrated) DSML is correctly implemented and behaves as specified. Furthermore, because DSMLs evolve over time (see, e.g., [85]), we must be able to efficiently test the evolving language artifacts (e.g., the integration of two or more DSMLs into a new DSML). In the context of defining test cases, a semantic gap exists between the human-level requirements and solution descriptions on the one hand, and the technical platform that is used to implement the respective DSML on the other. The wider this semantic gap, the more difficult is the task to correctly specify and implement a DSML that behaves as desired by its human users. For this task, scenarios are a natural means to describe (intended) system behavior both as structured textual requirements (see, e.g., [17]) and as executable software test (see, e.g., [63, 139, 159]) definitions.

In this context, this thesis presents an approach for the integration and the test of MOF/UML-based DSMLs. The next section provides an overview of our approach.

1.2 Approach Synopsis

Regarding the integration of DSMLs, this thesis presents an approach based on higher-order transformations for rewriting model-to-text (M2T) transformation specifications [53]. A higher-order transformation (HOT) is characterized by being "a model transformation such that its input and/or output models are themselves transformation models" [157]. This higher-order rewriting technique allows for the automated modification of transformation templates to fix important syntactical mismatches between templates and the integrated DSML (see, e.g., [177]). Our DSML integration approach uses techniques to combine two or more DSMLs which were not necessarily intended for integration at design time of each DSML. The integration process does not change the initial DSMLs, but provides techniques to transform and to compose the different artifacts for the creation of a new DSML (with the help of HOTs; see, e.g., [156, 157]). Therefore, reuse is facilitated in a way that all DSMLs can be used in parallel—either as standalone DSMLs or via an integrated DSML.

Furthermore, in this thesis, we propose a scenario-driven DSML testing approach to complement existing testing processes [57, 126]. In our work, scenarios (see, e.g, [12, 13, 63, 139]) are used to define domain requirements (e.g., requirements for the integration of DSMLs). 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 integrated DSML for compliance with the corresponding domain requirements.

Knowledge about the development process and possible application scenarios of MOF/UML-based DSMLs is a prerequisite for the creation of an integration and test approach. Besides the theoretical background of an extensive literature study [52, 54, 127], in the course of this thesis project, DSML experiences were also acquired by developing two MOF/UML-based DSMLs [58, 59]. Hence, we successfully applied our integration and testing approach on DSMLs developed by us and in combination with DSMLs developed by third-parties (e.g., [36, 77]).

This thesis project builds on and integrates with former research results on model-driven security engineering (see, e.g., [121, 134, 135]). As the two case studies for developing DSMLs [58, 59] and the integration and testing approach [53, 57, 126] complement former work, we have chosen security as application domain (e.g., to demonstrate the applicability of our approach via security-related examples). However, our approach is independent of an application domain and can be applied to other domains in an analogous manner (as long as the conceptual framework of the DSML conforms to the MOF/UML). Furthermore, parts of our results can be generalized and transferred to non-MOF/UML-based DSMLs, as well (e.g., the formal and generic metamodel definition for secure object flows [56]; for more information see Chapter 6).

Figure 1.1 shows a simplified process model of the approach presented in this thesis. It starts with the definition of domain requirements for the integration of two DSMLs (DSML `A` and DSML `B` in Figure 1.1, respectively). The initial DSMLs to be integrated are developed by DSML engineers, i.e. either by ourselves [58, 59] or by other developers. Both DSMLs as well as the integration requirements serve as input to our integration approach [53]. The integration is based on model merge definitions for the core language model of DSMLs. We use these merge definitions (stored in

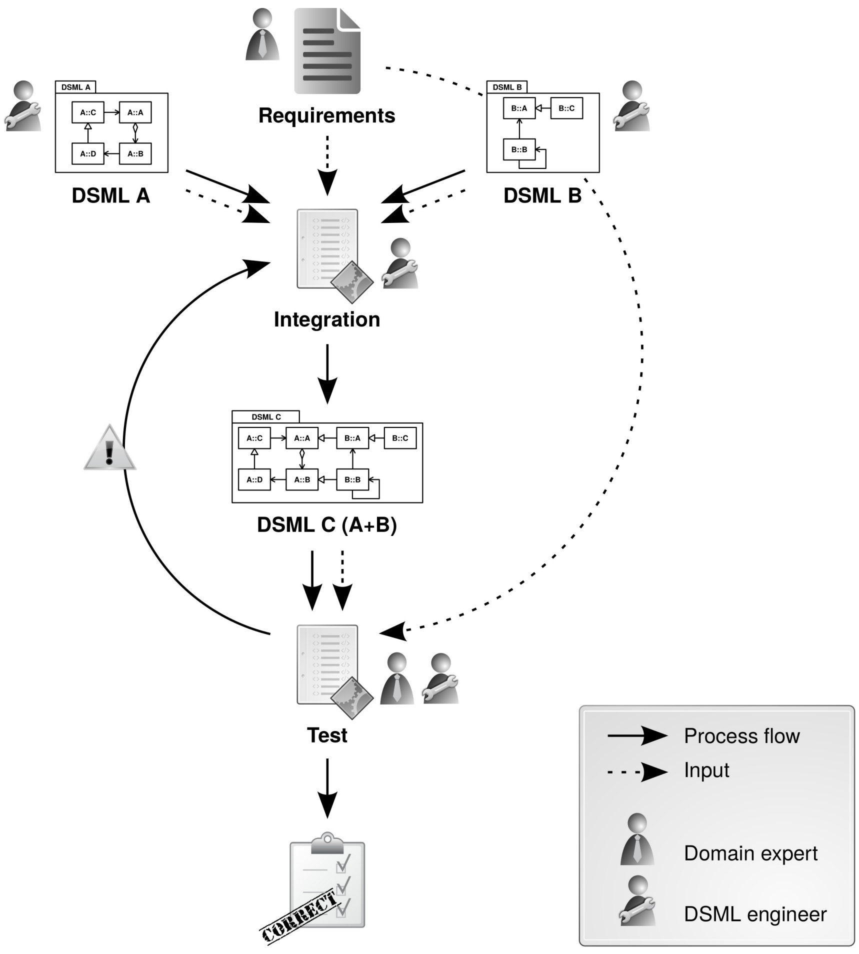


Figure 1.1: Simplified process of the presented approach

a trace model) for the automatic creation of rewrite rules for M2T templates of the individual DSMLs to support their platform-specific integration. The integration is administered by a DSML engineer and produces an integrated DSML (DSML `C` in Figure 1.1, composed of the individual DSMLs `A` and `B`). In order to test the correct integration of the DSML (i.e. that it conforms to the requirements specified at the time of integration), we apply our testing approach [57, 126]. The requirements initially specified for the integration are defined via natural-language scenarios by the domain expert. The DSML engineer creates transformations for these scenario descriptions to automatically generate scenario tests for the integrated DSML. If all scenario tests pass—i.e. the integrated DSML conforms to its requirements—, the individual DSMLs have been correctly integrated. Otherwise, if at least one scenario test fails, the domain expert in combination with the DSML engineer have to review the scenario test protocol to find discrepancies in the integration definitions with respect to the domain requirements and adjust the integration specifications accordingly. This iterative integration and testing process ends when all scenario tests pass.

To sum up, Table 1.1 provides an overview of our research contributions for the integration and test of MOF/UML-based DSMLs which were performed in and are results of this thesis project. In particular, in [60], we discuss issues that may arise when integrating MOF/UML-based DSMLs and present a process model for the systematic integration of DSMLs to address some of these issues. We showcase our integration process in a case study via an integrated model-driven approach for the specification and the enforcement of secure object flows in process-driven service-oriented architectures (SOAs [50, 56]). Our higher-order rewriting approach for integrating M2T templates is presented in [53]; its feasibility and benefits are empirically evaluated via a case study reported in Appendix B. With regard to testing DSMLs, in [47], we review selected testing techniques for each phase of a MOF/UML-based DSML integration process. We present an approach and framework for the requirements-driven and scenario-based testing of DSMLs in [126] and provide for an extension to specify structured natural-language scenarios in [57]. In [55], we evaluate our testing approach via a controlled experiment to understand how different scenario notations compare to each other with respect to accuracy and effort of comprehending scenario-test definitions, as well as with respect to the detection of errors in the models under test (MUT). All publications as well as accompanying materials (software artifacts, experimental material etc.) are publicly available at [48].

Table 1.1: Overview of research contributions for the integration and test of MOF/UML-based DSMLs

DSML integration	Integration process definition [60]; MDD-based integration case study [50, 56]; M2T template rewriting approach [53]; rewriting case study (see Appendix B)
DSML testing	Integration testing techniques [47]; scenario-based testing [126]; natural-language scenario testing [57]; experiment comparing scenario notations [55]

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 elaborates on the basics of model-driven software development (Section 2.1), MOF/UML-based domain-specific modeling (Section 2.2), higher-order model transformation (Section 2.3), and scenario-based model testing (Section 2.4). The problem statements addressed in this thesis, classified according to the two research areas (see above), are defined in Chapter 3: DSML integration (Section 3.1) and DSML testing (Section 3.2). In Chapter 4, our research approach applied in the course of this thesis is explained.

Research results emerging from this thesis project are discussed in Chapter 5. In particular, an overview of relevant publications is given (Section 5.1), the relations of the individual research contributions are shown (Section 5.2), details about accompanying artifacts are provided (Section 5.3), adopted formal specifications and mappings to developments constituting our integrated software support are explained (Section 5.4), and closely related additional research contributions are listed (Section 5.5). Chapter 6 concludes with a summary of the results of this thesis, discusses limitations of our approach, and points to further research directions. After the bibliography, the originally published individual research contributions (corresponding workshop, conference, and journal articles) are listed in Appendix A. The case study for evaluating our approach of higher-order rewriting of M2T templates [53] is reported in Appendix B.

Chapter 2

Background

2.1 Model-driven Software Development

In model-driven development (MDD), models are central artifacts and are frequently employed for the design of complex software systems. In this sense, MDD uses abstraction methods (i.e. models) familiar from traditional engineering disciplines, such as, architecture, construction, or civil and mechanical engineering. We define a *domain model* as a coherent set of formally specified elements describing (excerpts of) a particular domain via conceptual entities and relationships between those entities (see, e.g., [78, 123]). Concepts for the banking domain could be, for example, account number, credit transfer, and booking date. These concepts could relate to each other as, for example, a customer wants to transfer an amount of money from an account (identified by a dedicated account number) and wants it booked on another account on a specific date. These concepts as well as their relations can be defined in a dedicated banking domain model.

For the purpose of creating models for MDD, modeling languages support the representation of concepts and their relationships in a consistent and uniform way. Different modeling languages exist providing a formal syntax and semantics for the (tool-supported) creation of models, such as, UML [96], MOF [100], or entity-relationship (ER) models (see, e.g., [15]). We refer to a modeling language dedicated to a particular domain as *domain-specific modeling language* (DSML; see Section 2.2).

A model does not necessarily need to describe a domain completely. Rather, it will focus on one particular aspect of a domain (e.g., modeling the workflow of a bank customer's transaction and thereby neglecting the organizational structure of the bank). In doing so, models represent system *viewpoints* on a particular domain. A *view* is a representation of a system from the perspective of a viewpoint. Each view focuses on the elements relevant to that particular viewpoint, abstracting away all irrelevant details (see, e.g., [92, 161]). Viewpoint modeling allows to reduce the complexity of individual models, thereby increasing their readability, maintainability, and flexibility. To model a software system completely, several viewpoint models can be interwoven to represent all necessary domain aspects (see, e.g., [78]). The weaving of models is facilitated by using an integrated approach, i.e., by constructing models compatible with each other, for instance, by describing them in one modeling language (see, e.g., [56]). As an example, the UML provides 14 different structural as well as behavioral diagram types which can be used not only in isolation but which can also be combined with each other [96].

Furthermore, in MDD, models are used to describe a system from different abstraction levels (see, e.g., [78, 122, 123]). For example, the various UML diagram types can be employed to model different abstraction levels and, therefore, can be utilized for business process specifications (e.g., via activity diagrams) as well as software system implementation descriptions (e.g., via class and object diagrams). As a reference frame, the model-driven architecture (MDA) initiative of the Object Management Group (OMG) defines three abstraction levels [92]:

- *Computation independent model (CIM)*: A CIM (sometimes also referred to as *domain model*) omits details of the structure of a system and uses vocabulary familiar to the domain expert. The CIM plays an important role in bridging the gap between domain experts (those that are experts on domain requirements) on the one hand, and DSML engineers (those that are experts of the design and construction of the artifacts that together satisfy the domain requirements), on the other hand (see also Section 2.2). For example, a CIM can be provided as a generic metamodel that can be used to extend arbitrary modeling languages.

- *Platform independent model (PIM)*: A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type. For example, a PIM can be specified via technology-neutral UML models.

- *Platform specific model (PSM)*: The specification of the PSM may vary depending on its application on the target platform (e.g., model interpretation, source code generation [54]). In general, a PSM refines the specifications from the PIM by adding details that specify how that system uses a particular type of platform. If the PSM is not directly executable, platform-specific software artifacts need to be generated (e.g., source code or configuration and deployment specifications; see, e.g., [131]). For example, a PSM can be provided as a platform-specific UML profile used to generate Java source code.

One of the key challenges of MDD is the automated transformation of models on higher abstraction levels (CIMs, PIMs) to executable software artifacts (PSMs, source code etc.; see also Section 2.3). The objective is to increase productivity and quality by enabling development and using concepts closer to the problem domain, rather than those offered by programming languages. This is because models provide the benefit of being both, abstract and formal at the same time: the problem domain can be specified in a compact and reduced form. In this case, models are no longer only documentation, but parts of the software (i.e. constructing models become semantically equivalent to writing source code; see, e.g., [78, 124, 131]).

2.2 MOF/UML-based Domain-specific Modeling

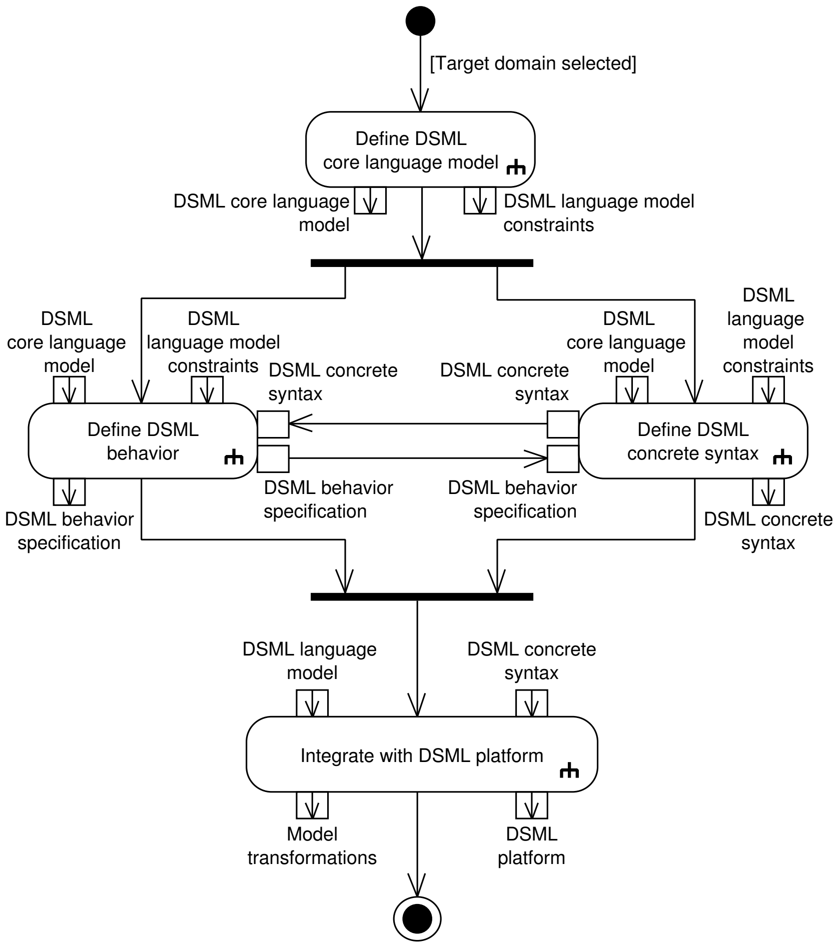
In MDD, a domain-specific modeling language (DSML) is a tailor-made software language for a specific problem domain. DSMLs are a special kind of domain-specific languages (DSLs). In contrast to textual or form-/table-based DSLs, DSMLs provide end users with a graphical or diagrammatic concrete syntax (see, e.g., [73, 84, 130, 137, 182]). DSLs/DSMLs are used as an abstraction layer to facilitate the communication between 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 (via models) that can be processed by an information system. Thus, DSMLs (as DSLs in general) aim at increasing the number of people who can actively participate in the specification, configuration, and management of software-based systems (see, e.g. [122, 123]).

According to [137], the development of DSMLs can be divided into four main tasks (see also Figure 2.1):

- *Define DSML core language model*: In this task, the DSML developer defines an initial *core language model* and corresponding *language model constraints* for the selected target domain. In particular, domain analysis methods (see, e.g., [23]) are used to identify abstractions and specify the core language model of a DSML. Because the core language model often cannot capture all restrictions and/or semantic properties of the DSML elements, language model constraints are added, if necessary. This phase results in the DSML core language model and an (optional) set of DSML language model constraints.

  The core language model of a DSML is typically defined using metamodeling and it is exposed to domain modelers in terms of a diagrammatic concrete syntax (see task *Define DSML concrete syntax* below and, e.g., [124]). A *metamodel* is a model of models; i.e. a metamodel is used to represent other models (see, e.g., [27, 92]). The relationship between metamodels and models can be illustrated via type/instance relations: models are instance specifications of their metamodel definitions (types); i.e. the metamodel defines the reference frame to which its conforming models must adhere to. As metamodels allow for classification abstraction mechanisms (models of metamodels are also metamodels; i.e. instances of types are also types), an infinite metamodeling hierarchy is derived from this recursive definition (see, e.g., [39]).

  In the context of DSMLs, a metamodel (with optional behavior and constraint specifications) defines the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts (see, e.g., [96, 161]). These three parts—the DSML core language model (i.e. metamodel), language model constraints, and behavior specifications (see task *Define DSML behavior* below)—make up the *language model* of



[Target domain selected]
Define DSML core language model
DSML core language model
DSML language model constraints
DSML core language model
DSML language model constraints
DSML concrete syntax
DSML concrete syntax
DSML core language model
DSML language model constraints
Define DSML behavior
Define DSML concrete syntax
DSML behavior specification
DSML behavior specification
DSML behavior specification
DSML concrete syntax
DSML language model
DSML concrete syntax
Integrate with DSML platform
Model transformations
DSML platform

Figure 2.1: Language model driven DSML engineering process (adapted from [137])

a DSML which is also often referred to as the *abstract syntax* of a DSML.

- *Define DSML concrete syntax*: The concrete syntax serves as the DSML's interface. In this task, graphical/diagrammatic (for DSMLs) or textual/form-/table-based (for DSLs) notation symbols as well as syntax-specific composition and production rules are defined. The different syntax types are tailored to the need of the modeler, for instance, different syntax styles may be chosen depending on the modeler's domain and/or software-technical proficiency. The DSML core language model and the DSML language model constraints serve as input to produce the *DSML concrete syntax specification.*

- *Define DSML behavior*: The behavior specification (also: *dynamic semantics*) of a DSML determines how the DSML elements interact to produce the behavior intended by the DSML designer. Moreover, the behavior specification defines how the DSML language elements can interact at runtime. Syntax and behavior of a DSML are usually defined in parallel. Again, the DSML core language model and the DSML language model constraints serve as input. The result of this task is the *DSML behavior specification* (e.g., by defining control flow models or formal textual specifications).

- *DSML platform integration*: All artifacts defined for a DSML are mapped to the features of a selected software platform by either extending an existing platform or by developing a new tool set. Often, platform integration is achieved by defining model transformations (see also Section 2.3 and, e.g., [81, 124]) to convert one model into another model (also called: model-to-model transformation, M2M) or into executable software artifacts (also called: model-to-text transformation, M2T). The DSML language model and the DSML concrete syntax specification serve as input to this task.

The order of these four DSML development tasks largely depends on the DSML's development context. In [137, 182], the following process variants have been identified: 1) *language model driven DSML development*, 2) *mockup language driven DSML development*, and 3) *extracting the DSML from an existing system*. In this thesis, all DSMLs are developed according to (1). Thus, here, we provide a characterization of the language model driven DSML development process variant only (see Figure 2.1): The DSML engineering process is driven by the language model definition (abstract syntax). First, the core language model with accompanying language model constraints is specified for the relevant domain concepts, then the concrete syntax and corresponding behavior specifications are defined. Finally, the DSML is mapped to a dedicated platform.

This thesis deals exclusively with MOF/UML-based DSMLs. The MOF "serves as the platform-independent metadata management foundation for MDA" [100]. With the metamodeling infrastructure provided by the MOF, the UML metamodel is defined. For the specification, the MOF reuses common core packages from the UML infrastructure library (via package imports/merges [95]) and extends it with additional packages. The UML infrastructure library defines a core language that can be reused to define a variety of metamodels (including UML and MOF). The MOF is described using both textual and graphic presentations. To precisely describe the abstract syntax and semantics of the MOF, a combination of languages is used: a subset of UML, the Object Constraint Language (OCL [99]), and precise natural language. Two alternatives are defined in the MOF specification: the essential MOF (EMOF) and the complete MOF (CMOF). They have different usage scenarios as the "primary goal of EMOF is to allow simple metamodels to be defined using simple concepts while supporting extensions [...] for more sophisticated metamodeling using CMOF" [100]. For example, the UML metamodel is specified using CMOF.

The MOF is often referred to as a four-layered metamodel architecture, although the reflection interfaces allow traversal across any number of metalayers recursively (i.e. an infinite number).[1] Four is the minimum number of layers (M3–M0; see

[1]The UML infrastructure library is defined reflective, i.e., it contains all the metaclasses required

Figure 2.2 for an example) needed for the specification of the MOF as well as for MOF-based modeling languages (e.g., the UML). The primary responsibility of the M3 layer is to define the language (i.e. a meta-metamodel) for specifying a metamodel. The MOF is an example of a metamodel on the M3 layer which is used to define a metamodel on the M2 layer (e.g., the UML metamodel). A metamodel on the M2 layer is an instance of a metamodel on the M3 layer, meaning that every element of the M2 metamodel is an instance of an element in the M3 metamodel (see Figure 2.2 and also Section 2.1). The primary responsibility of the M2 metamodel layer is to define a language for specifying models (on the M1 layer). An M1 model is an instance of an M2 metamodel. The primary responsibility of the M1 model layer (i.e., a user-generated model) is to allow users to model different problem domains. At the bottom of the metamodel hierarchy, the M0 layer contains runtime instances of model elements defined in an M1 user model (for further information see [95]).

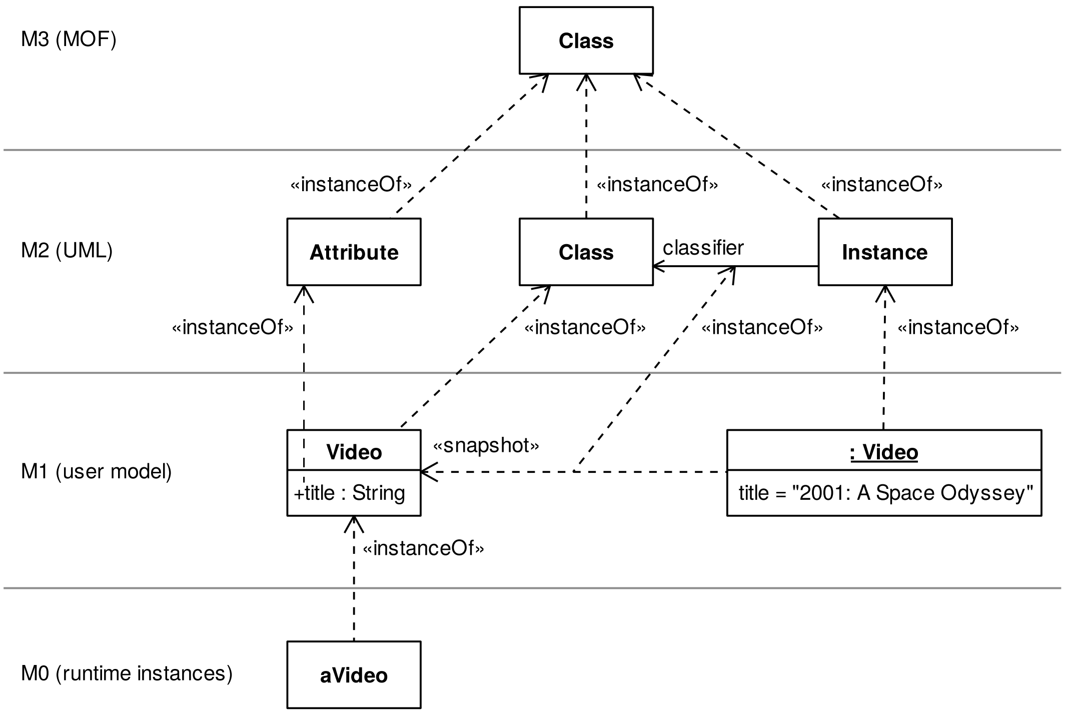


Figure 2.2: MOF four-layer metamodel hierarchy example (adapted from [95])

The UML is the de facto standard for modeling software-based systems and provides native support for all types of software models (structure, behavior, interaction diagrams). The UML serves as a solid foundation for DSML engineering (e.g., standardized modeling extensions, extensive tool support; see also Section 1.1) and in this thesis, we utilize the UML for DSML development in two ways: by 1) using UML profiles (native extension [96]) and by 2) introducing new or altering existing

to define itself. As the MOF is based on the UML infrastructure library, it is also reflective [95].

modeling concepts on the metamodel level (MOF-based extension).[2] UML profiles provide a mechanism for the extension of existing UML metaclasses to adapt them for non-standard purposes (e.g., define more restrictive constraints, provide a graphical icon as additional stereotype visualization). However, UML profiles are not a first-class extension mechanism [96]. They extend existing metaclasses of the UML metamodel, thereby being consistent with the semantics of the original UML metaclasses. For this reason, more complex extensions are defined on the level of the UML metamodel [96, 100]. A MOF-based extension allows to define new and specifically tailored UML elements (by defining new metaclasses), and allows to define a customized notation, syntax, and semantics for the new modeling elements. In our DSMLs, we employ a combination of both methods to take advantage of each mechanism. We have implemented our DSMLs via the Eclipse Modeling Framework (EMF; see, e.g., [132, 146]). In this technology projection, MOF-compliant DSML models are approximated by Ecore models.

2.3 Higher-order Model Transformation

MDD offers the potential for automatic transformations of concepts defined on an abstract level into running software systems (see, e.g., [78]). In general, the term *model transformation* is used when at least either the input or the output artifacts are graphical or textual models. In the literature this is often contrasted to the term *program transformation* (see, e.g., [109]), which indicates that input and/or output artifacts of a transformation are programs (source code, byte code etc.). However, executable program definitions (e.g., source code) can be seen as models, as well, because they are abstracted representations of lower-level machine structures and operations (see, e.g., [20, 81]). According to this definition, the term model transformation encompass program transformation since "a model can range from abstract analysis representations of the system, over more concrete design models, to very concrete models of source code" [81]. As the concept of models is more general than the concept of program code, we use the term model transformation to refer to transformations operating on a more diverse set of artifacts than program transformations (see, e.g., [20]). In this thesis, the term *model* refers to abstractions above program code (e.g., requirements and design specifications, domain concepts; see, e.g., [20]) as well as to different representations of program code (e.g., source code specifications of a transformation language stored in the XML Metadata Interchange format; XMI [101]). In the context of the MDD approach applied in this thesis project, the input and output models of model transformations can take various forms, for example, DSML core language models or model-to-text transformation specifications (see also further below).

As sketched in Figure 2.3, the output of model transformations are either again graphical models (in a model-to-model transformation; M2M) or text-based artifacts

[2] A more detailed discussion of the decision making process for customizing the UML is presented in [10].

(in a model-to-text transformation; M2T). According to the above definition, generated textual artifacts of a M2T transformation can represent models (e.g., program code, XMI specifications), as well, but also aims at producing textual representations not adhering to a dedicated metamodel definition in the technical space of MDD (e.g., reports, documents; see, e.g., [93]). A model transformation can take one or multiple graphical/textual models as input and can generate one or multiple graphical/textual models as output. A typical M2M transformation example is the generation of PSMs from PIMs—a transformation from a more abstract to a more concrete model. In a reverse engineering context, the transformation would be inverted; i.e. a more abstract model would be generated from a more concrete one (both are examples of vertical transformations, opposed to horizontal transformations; see, e.g., [81]). In a M2M transformation, the input and output instance models must not necessarily conform to the same metamodel (endogenous vs. exogenous transformation; see, e.g., [81]). As models are a means for abstraction, they may not capture enough implementation details to be directly executable. In such a case, M2T transformations may be used to generate textual models/artifacts (e.g., source code, configuration documents) which can be deployed and processed when integrated in a specific platform.

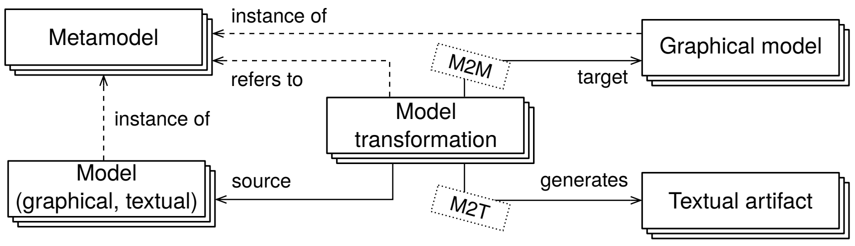


Figure 2.3: Model transformations in MDD-based engineering (adapted from [49])

Different methods exist to perform model transformations, such as, template-based, visitor-based, graph-transformation-based, relational, or hybrid approaches (see, e.g., [20]). Although we employ several model transformation methods, the main contribution of this thesis focus on template-based M2T generators. M2T *generator templates* receive a transformation definition and a set of source models as the input to produce a transformed representation of these models. In principle, a generator template consists of two kinds of code. On the one hand, there is template code to access and to select source model data by quantifying over the model structure that is specified in a metamodel. On the other hand, a template contains code to expand and to wrap the selected model data into string fragments. Template-based M2T transformations are a widely supported platform integration technique in contemporary MDD tool chains, and a variety of template language implementations exist, such as, Xpand, MOFScript, Epsilon Generation Language (EGL), Java Emitter Templates (JET), or Acceleo (see, e.g., [20, 70, 115, 116, 143, 150, 152])—the latter being the reference implementation of OMG's MOF Model to Text Transformation Language specification (MOFM2T [93]).

A higher-order transformation (HOT) is distinguished from a "common" model transformation as it "is a model transformation such that its input and/or output models are themselves transformation models. [...] This demands the representation of the transformation as a model conforming to a transformation metamodel" [157] (see also Figure 2.3). In this way, transformation specifications (e.g., generator templates) become first-class model entities, as well. In [156, 157], four transformation classes for HOTs are identified and characterized by their input and output models:

- *Transformation analysis*: HOTs are employed to process other transformations to extract meaningful data. They have at least one transformation as input model and no transformations as output models.
- *Transformation synthesis*: HOTs are employed to create new transformations from data modeled in other forms. They have no transformations as input models and at least one transformation as output model.
- *Transformation modification*: HOTs are employed to manipulate the logic of an input transformation to generate a modified version of the same transformation. They have one transformation as input model and one transformation as output model.
- *Transformation (de)composition*: HOTs are employed to merge or split other transformations, according to a (de)composition criterion. They have at least one transformation as input model, at least one as output model, and the input and/or the output models contain more than one transformation.

Our approach of higher-order rewriting M2T templates for integrating DSMLs [53] can be classified as a *transformation modification.* A model representation of a M2T transformation specification is provided as input to a HOT. The HOT modifies the input transformation model and generates an adapted transformation as output model.

2.4 Scenario-based Model 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., [12, 13, 63, 139, 159]). In recent years, scenarios have become a popular means for capturing requirements of software systems (see, e.g., [17, 129, 139]). Scenarios can be described in different ways at various abstraction levels, for example, via natural language, (informal) graphical models, or precise (and formal) textual specifications (see, e.g., [55]). In particular, different approaches have been proposed to provide notations for specifying scenarios, for example, in a table-based layout, as message sequence charts, or via formal methods (see, e.g., [1, 17, 160]). No generally accepted standard notation exists and the different scenario notations vary with respect to the corresponding application domain and with the professional background of the stakeholder involved in a particular development project (see, e.g, [17]).

Evaluating requirements-level scenarios poses important challenges with respect to, for instance, ambiguity, consistency, singularity, or traceability of defined requirements (see, e.g., [62, 139]). Furthermore, scenario descriptions are often not executable, thus, making the automatic validation of implemented software artifacts difficult. These issues are influenced not only but also by the notation employed for specifying scenarios. For example, consider the discrepancy of specifying scenarios in a natural language (may be simple to formulate, but difficult to validate automatically) or via formal methods (may be difficult to formulate, but simple to validate automatically). Hence, the choice of selecting a notation significantly influences the application of scenarios and trade-offs between different notation styles must be considered (e.g., scenarios as a means for documentation and communication purposes or as a means for formal verification).

To effectively validate scenarios, the different action steps in a non-executable scenario description (e.g., specified in a natural language) can 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., [91]).

In general, it is almost impossible to completely test a complex software system (see, e.g., [88, 133, 136]). Hence, the selection, description, maintenance, and automation of relevant test cases become an important factor. In this context, scenarios can help to reduce the risk of omitting or forgetting relevant test cases, as well as the risk of describing important tests insufficiently (see, e.g., [89, 118, 136]). Scenarios can contribute to achieve a critical test coverage of relevant requirements by checking each requirements-level scenario via corresponding scenario tests. Moreover, as in a scenario-driven engineering approach, changing domain requirements are first identified at the scenario level (see also [63, 139]), one can rapidly identify affected scenario tests and propagate the changes into corresponding test specifications (see, e.g., [57, 126, 133]).

In MDD, scenario specifications can be used to test models for compliance with corresponding domain requirements (see, e.g., [9, 89, 126]). For a DSML, the core language model, the behavior specification, or model transformations are examples of artifacts which may be specified via models (e.g., via structural or behavioral UML diagrams). Performing a scenario-based testing process for models 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. An example notation for non-executable scenario descriptions would be the one-column table format as suggested by Cockburn [17]. This format demands that a set of test scenarios is defined for every test case. Properties for each test case and each test scenario are specified via natural language, such as, pre- and postconditions, event triggers, or expected outcomes. These requirements-level scenarios are non-executable and can be translated into model tests in a next step, for example, into scenario tests specified in the Epsilon Unit Testing Framework (EUnit [33]) which is designed to define tests for model-management tasks. EUnit is a task-specific language provided by the extensible platform of integrated languages for model management (Epsilon [70]) which is based on EMF (see, e.g., [132]). In this way, the functionality for scenario-based model tests can be realized, including, the distinction of test concepts via containment relationships (e.g., test case, test scenario), the sequencing of test executions, or the support for guarding expressions (i.e. pre-/postconditions) as well as for setup/cleanup operations (see also [126]).

Chapter 3

Problem Areas and Related Work

In this chapter, we describe the research problems which are covered by this thesis. The problem statements are discussed with respect to related work and are classified according to the two categories this thesis provides research contributions for (see also Section 1.2, especially Table 1.1).

3.1 DSML Integration

Problem Statement 1. *When integrating two (or more) DSMLs, the reuse of M2T transformation templates for the integrated DSML would yield a number of benefits, such as, reduced testing and maintenance efforts. The automated and tool-supported syntactical adaptation of individual M2T templates for their reuse in DSML integration scenarios is currently not provided by existing approaches.*

In general, many issues relate to the integration of MOF/UML-based DSMLs, such as, establishing a consolidated domain space, developing a compatible formalization style, adapting core language model constraint sets, defining an executable integration workflow, specifying the conflict-free composition of symbol sets, arranging the integration order with respect to the functional properties of the integrated DSMLs, handling multiple host software platforms, providing integrated tool support, or adapting M2T generator templates (see, e.g., [14, 22, 52, 54, 60, 87, 137, 172, 182]).

The integration issues relate to the different phases of DSML development (define DSML core language model, define DSML concrete syntax, define DSML behavior, and DSML platform integration; see Section 2.2 and [137]). In this context, related work does not discuss issues arising at each phase of DSML integration as interdependent parts of a systematic integration process. In particular, research approaches limit themselves to individual aspects of, for example, integrating DSML core language models (see, e.g., [22, 28, 173]), integrating graphical concrete syntaxes (see, e.g., [110]), integrating model transformations (see, e.g., [156, 157, 172]), or integration at the level of the software platform (see, e.g., [130, 138, 182]). We contribute to this research by discussing integration issues at each of the DSML integration phases and drafting a systematic integration process for MOF/UML-based DSMLs [60].

In the context of a systematic DSML integration process, our integration approach presented in this thesis focuses on reuse aspects of M2T transformations. Related work regarding the syntactical adaptation of M2T transformations is especially concerned with reuse techniques for generator templates (e.g., HOTs, generic templates, adapter models [53]). For these M2T transformation templates, we identified problems with regard to their reuse when integrating DSMLs (related to the phase of DSML platform integration) [50, 56, 60], primarily because related work is mainly concerned with M2M transformations (e.g., transformation genericity has not been documented for M2T generator templates so far; see, e.g., [18, 19, 168]). Currently, the reuse and adaptation of metamodeling concepts for evolving M2T templates is mostly performed by a combination of standard-tool-supported cloning of existing templates (e.g., copy and paste) and manually identifying the concepts to be rewritten step by step. Only after manual identification of the concepts, their occurrences can automatically be searched and replaced across the generator templates via standard tools. Our approach benefits from recording traces from M2M transformations on the metamodel level (i.e. from the integration of the DSML core language models) and by using these traces to automatically rewrite the corresponding M2T templates (in one bulk operation). In this way, our approach supports a consistent evolution of M2M and M2T transformations. Furthermore, the evolution process is explicitly documented via the definition of M2M transformations and via the generated higher-order rewrite rules for the corresponding M2T transformations. We provide integrated tool-support for the automatic rewriting of M2M templates in MDD environments, thereby using native methods and tools (e.g., transformations represented as first-class models, applying M2M transformations on generator template models). In our approach, DSML engineers can use tools they are familiar with and the cognitive distance of the rewritten M2T templates is not increased, as is the case, for example, for adapter models because the generated platform artifacts do not reflect derived or newly introduced domain concepts (see, e.g., [113]). In this context, a further disadvantage of adapter models is that they restrict the generation of glue code using M2T transformations because concept correspondences between the integrated DSML and the source DSMLs cannot be leveraged in code generation (see, e.g., [113]).

Our approach is based on HOTs for the automatic rewriting of M2T templates (see Section 2.3). Current HOT approaches lack generalizability and transferability because they concentrate on specific transformation platforms only (e.g., Atlas Transformation Language; ATL) rather than on HOTs in a technology-independent manner (see, e.g., [156, 157]). To address this issue, we map the concepts of our approach to the MOFM2T specification [93] to facilitate transferability to other M2T transformation languages and environments. However, to use HOTs on M2T transformations, the generator templates need to be represented as first-class models. In related work, we could not identify a M2T transformation language explicitly providing both, a metamodel for its abstract syntax and an execution environment for the instance M2T transformation models. Our approach provides a metamodel as well as round-tripping functionality for EGL [70]—by adapting and extending work from [174]. Hence, our solution is capable of translating textually specified EGL templates into their model representation (and vice versa). The approach of not executing model representations directly has the advantage that existing EGL templates can be reused as is and they can be specified in both formats: textually and as models.

With regard to Problem Statement 1, results of our work concerning the integration of DSMLs are published as [50, 53, 56, 60] and are presented in Appendix B, respectively.

3.2 DSML Testing

Problem Statement 2. *In an integration scenario, the DSML core language models are not static artifacts, but evolve according to the requirements defined in the integration process. The automated and tool-supported validation of the integrated core language model for checking its conformance against corresponding DSML integration requirements specified via natural-language is currently not provided by existing approaches.*

In related work, different techniques are proposed for the testing of DSML artifacts, such as, model checking and verification (e.g., for the DSML core language model; see, e.g., [11, 32, 42, 183]), graphical user interface testing (e.g., for the DSML concrete syntax; see, e.g., [8, 79, 112]), or model transformation testing (e.g., for the DSML platform integration; see, e.g., [158, 162, 176]). However, existing test approaches neither take the different DSML development phases into account nor specifically target issues arising when integrating DSMLs (see, e.g., [47]).

In particular, this thesis presents a model testing approach based on scenario specifications applied in a DSML integration process. In this context, related work falls short with respect to providing requirements-driven and scenario-based testing procedures for evolving DSML core language models (as required in a DSML integration scenario). Existing approaches consider a metamodel (e.g., a DSML core language model) 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 (see, e.g, [38, 83, 119]). In contrast, our test approach explicitly supports the iterative development of language models. Every modification to the model under test (MUT; e.g., a sequence of metamodeling actions as performed when integrating DSML core language models) is validated by a set of corresponding test definitions to check for requirements conformance. Iterative development and testing is also facilitated in our approach via a seamless integration of the test infrastructure into the environment used for MDD [57, 126].

For evolving DSML core language models, requirements conformance is critical [126]. Current metamodel-testing approaches (e.g., modeling-space sampling, metamodel-test models, metamodel validation; see, e.g., [16, 38, 82, 83, 119]) address conformance checking differently. Shortcomings of related approaches are that in the context of modeling-space sampling, an existing and sufficiently specified MUT is required to generate potential instances (a requirement which is not always met in the step-wise development of DSML models; see, e.g., [119]). Our approach differs as it does not rely on sample model instantiations and can be employed to test complete models as well as model fragments of any size. Furthermore, related work shows that tool support for manual test reviews is currently not provided (see, e.g., [4]). In our approach, before the actual domain modeling integration is performed, the domain expert and the DSML engineer collaboratively review the executable test scenarios. This review is facilitated by maintaining the domain/integration requirement statements along with the corresponding test cases (i.e., trace links are established).

Moreover, to employ metamodel validation techniques, the translation of requirements (e.g., a narrative text, a requirements catalog, or variability models) into well-defined constraint expressions is not trivial and existing approaches do consider the structure of non-executable requirements specifications, such as, textual or diagrammatic scenario descriptions, only to a certain extent (see, e.g., [17]). In our approach, scenario-based requirements can be specified in a semi-structured natural-language format. As in related work, in order to be able to correctly map to executable scenario tests, the definition of natural-language scenarios in our approach must also conform to a pre-defined schema providing a basic structural skeleton. However, the concrete syntax of the scenario-based requirements specification language can easily be adapted. We provide a natural-language scenario notation by following related approaches to textual use-case modeling and acceptance testing (see, e.g., [37, 128, 180]). Nevertheless, this template syntax can be changed, for example, to conform to the notation requirements of the domain expert.

In addition, requirements distinct to testing the integration of DSML core language models, such as, to provide for the navigation between different metamodels to capture model transformations via, for example, an integrated, model-driven tool chain, are not met by related work in the context of testing against natural-language requirements (see, e.g., [21, 35, 120, 181]). In contrast, navigation between metamodels is supported by our approach, for instance, by mapping multi-metamodel requirements into executable test scenarios involving individual and integrated DSML core language models [57, 70].

In the context of testing DSMLs, research results concerning Problem Statement 2 are published as [47, 55, 57, 126].

Chapter 4

Research Approach

In this thesis project, research is performed in the context of integrating and testing MOF/UML-based DSMLs. The results of the research efforts manifest in the creation of novel, innovative, and useful artifacts (e.g., methods, models, software). In this sense, we apply a *design science* oriented research approach (see, e.g., [43, 44, 75, 111, 125]). In contrast to natural science which "tries to understand reality, design science attempts to create things that serve human purpose" [75]. In information system research, basically two complementary but distinct paradigms are recognized: behavioral science and design science (see, e.g., [44, 75]). The behavioral-science paradigm has its roots in natural science research methods and "seeks to develop and verify theories that explain or predict human or organizational behavior" [44]. In contrast, the design-science paradigm has its roots in engineering and the sciences of the artificial (see, e.g., [125]) and "seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts" [44]. Design science is fundamentally a problem-solving paradigm, technology-oriented, and its products are assessed against criteria of value and/or utility (see, e.g., [44, 75]).

Concerning the design-science aspects discussed in [75], the authors distinguish, on the one hand, two research activities: (*build* and *evaluate*). "Building is the process of constructing an artifact for a specific purpose; evaluation is the process of determining how well the artifact performs." [75] On the other hand, four different research outputs are presented: *constructs* ("constructs [...] constitute a conceptualization used to describe problems within the domain and to specify their solutions." [75]), *models* ("a model is a set of propositions or statements expressing relationships among constructs" [75]), *methods* ("a method is a set of steps (an algorithm or guideline) used to perform a task" [75]), and *instantiation* ("an instantiation is the realization of an artifact in its environment" [75]). To sum up, according to [75], design science is concerned with building and evaluating constructs, models, methods, and instantiations. In addition, the authors identify the last two research activities which make up their presented research framework (*theorize* and *justify*) as having natural science intent.

By building on and being consistent with prior research, the authors of [111] propose a methodology and provide a framework for carrying out design science research. In particular, a model for design science research processes is defined consisting of six activities in a nominal sequence. For this thesis project, we base our design science research approach on the process model presented in [111] (see Figure 4.1):

Identify problem and motivate. *In this phase, the research problems are defined and the value of solutions are justified.*

In this thesis, we are covering problems concerning the integration and testing of MOF/UML-based DSMLs (see also Chapter 3). In particular, we deal with the following two issues: Current approaches do not support the evolution of M2T generator templates as needed in an integration scenario (for details see Section 3.1) and integration requirements cannot automatically be validated (for details see Section 3.2).

Define objectives of a solution. *In this phase, the objectives of solutions are inferred from the problem definitions.*

In this thesis, objectives for MOF/UML-based DSML solutions comprise the (initial) definition of a systematic integration process, the automatic adaptation of M2T transformation templates for their reuse in an integration process, the discussion and review of testing techniques for each phase of integration, and the validation support for (natural-language) integration scenarios.

Design and development. *In this phase, artifacts are created, such as, constructs, models, methods, or software.*

In this thesis project, artifacts for MOF/UML-based DSMLs are designed to infer an integration process model from experience and related work as well as to provide integrated MDD-based tooling for the syntactical rewriting of M2T transformation templates and for the validation of requirements-level scenarios (defined via different scenario notations, such as, semi-structured natural-language scenario notation or fully-structured textual scenario notation).

Demonstration. *In this phase, the use of artifacts to provide for solutions to the particular problems is demonstrated.*

In this thesis, the use of MOF/UML-based DSML solutions is demonstrated by conducting an integration case study, by proof-of-concept software artifacts, and by example applications of developed methods and prototypes (e.g., an integration scenario to demonstrate the M2T generator template adaptation technique).

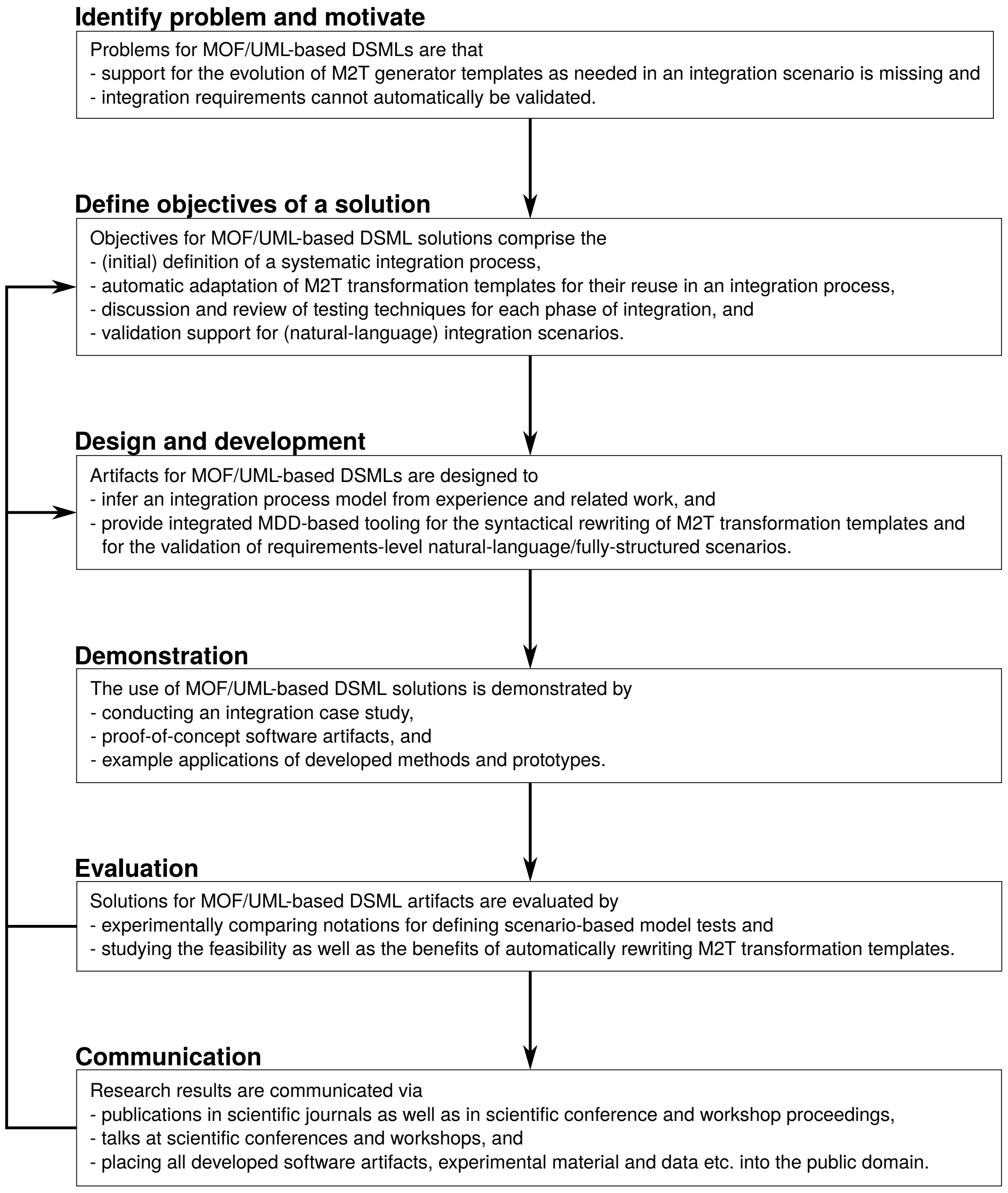

Identify problem and motivate
Problems for MOF/UML-based DSMLs are that
- support for the evolution of M2T generator templates as needed in an integration scenario is missing and
- integration requirements cannot automatically be validated.
Define objectives of a solution
Objectives for MOF/UML-based DSML solutions comprise the
- (initial) definition of a systematic integration process,
- automatic adaptation of M2T transformation templates for their reuse in an integration process,
- discussion and review of testing techniques for each phase of integration, and
- validation support for (natural-language) integration scenarios.
Design and development
Artifacts for MOF/UML-based DSMLs are designed to
- infer an integration process model from experience and related work, and
- provide integrated MDD-based tooling for the syntactical rewriting of M2T transformation templates and for the validation of requirements-level natural-language/fully-structured scenarios.
Demonstration
The use of MOF/UML-based DSML solutions is demonstrated by
- conducting an integration case study,
- proof-of-concept software artifacts, and
- example applications of developed methods and prototypes.
Evaluation
Solutions for MOF/UML-based DSML artifacts are evaluated by
- experimentally comparing notations for defining scenario-based model tests and
- studying the feasibility as well as the benefits of automatically rewriting M2T transformation templates.
Communication
Research results are communicated via
- publications in scientific journals as well as in scientific conference and workshop proceedings,
- talks at scientific conferences and workshops, and
- placing all developed software artifacts, experimental material and data etc. into the public domain.

Figure 4.1: Process of the design science research approach applied in this thesis (adapted from [111])

Evaluation. *In this phase, it is observed how well the artifacts support solutions to the problems.*

In this thesis, solutions for MOF/UML-based DSML artifacts are evaluated by experimentally comparing notations for defining scenario-based model tests (employed for the validation of integration requirements) and by studying the feasibility as well as the benefits of automatically rewriting M2T transformation templates (in comparison with manual refactorings).

Communication. *In this phase, the problems and their importance, the artifacts, their utility and novelty, and the rigor of their designs are communicated to the public.*

Research results of this thesis are communicated via publications in scientific journals as well as in scientific conference and workshop proceedings, via talks at scientific conferences and workshops, and via placing all developed software artifacts (e.g., tools, models, examples), experimental material and data etc. into the public domain (as free/libre open source software).

In this thesis, a problem-centered approach is applied which is the basis of the nominal sequence shown in Figure 4.1 (the phase of identifying the problems and justifying the value of solutions acts as research entry point [111]). However, the process was not carried out strictly in a sequential order. Iterations occurred in one phase and over different phases. For example, a prototypical implementation provided first insights into a possible solution for a specific problem and, after further research, the prototype was dismissed in favor of an improved design incorporating development experiences and merging them with new knowledge (an example iteration in the "design and development" phase). Furthermore, the evaluation of an artifact and discussions with colleagues elicited weaknesses which were used as a starting point for design improvements (e.g., optimizing the implementation, developing additional features). Moreover, evaluations and discussions also led to amended and extended objective definitions of a given solution which, in turn, also led to requests for, for example, additional features (these are examples of iterations over different phases, including "define objectives of a solution", "design and development", "evaluation", and "communication"; see Figure 4.1).

Chapter 5

Discussion of Results

In this chapter, we discuss the research contributions and results which emerged from this thesis project. We introduce the scientific publications, discuss the relations of the individual research contributions, and mention accompanying software artifacts. Moreover, we list further research contributions which emerged either as preparatory work to or as additional results of our main research findings.

5.1 Overview of Publications

This section provides an overview of research contributions published in the course of this thesis project.[3] The contributions are listed in their chronological order of publication and numbered accordingly. The full text of all papers is available in Appendix A.

P1 Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models. This paper presents an approach for incorporating data integrity and data confidentiality into the MDD of process-driven SOAs based on the SoaML specification [98] (see Section A.1 and [50]). Definitions for service interfaces are extended by UML activities to model invocation protocols. An invocation protocol makes the control and the object flows between service invocations explicit. Integrity and confidentiality attributes are used to annotate the object flows. The annotations serve for generating security-aware execution artifacts. We applied the approach prototypically in a web-services platform environment.

P2 Towards a Systematic Integration of MOF/UML-based Domain-specific Modeling Languages. This paper discusses issues that may arise when integrating MOF/UML-based DSMLs and present a process model for the systematic integration of DSMLs to address some of these problems (see Section A.2 and [60]). In particular, different integration techniques as well as challenges that may

[3]Classification of contributions: P = Paper, C = Chapter (of this thesis).

occur in the different phases of DSML integration are discussed. In addition, an example for the integration of two DSMLs from the security domain is provided.

P3 Higher-Order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages. This paper proposes a technology-independent approach to M2T generator template rewriting based on higher-order model transformations in order to reuse the original templates for, for example, an integrated DSML (see Section A.3 and [53]). In particular, the approach addresses syntactical mismatches between the templates and an integrated metamodel and enables syntactical template rewriting in an automated manner. To demonstrate the feasibility of this rewriting technique, a prototype for EMF and the Epsilon model-management toolkit was built.

P4 Towards Testing the Integration of MOF/UML-based Domain-specific Modeling Languages. This paper reviews the suitability of selected testing techniques for each phase of a MOF/UML-based DSML integration process (see Section A.4 and [47]). Every test technique is exemplified by providing a motivating example of its application to the integration of existing, security-related DSMLs. As for evaluation, prototypical software implementations are provided.

P5 Requirements-driven Testing of Domain-specific Core Language Models using Scenarios. This paper presents an approach for the scenario-based testing of core language models of DSMLs (see Section A.5 and [126]). The approach uses domain scenarios at the requirements level as primary artifacts. These non-executable, human-understandable scenario descriptions are refined into executable scenarios test (in the paper exemplified via a DSML integration case). To demonstrate the applicability of the approach, a scenario-based testing framework was implemented for EMF and the Epsilon language family.

P6 Natural-language Scenario Descriptions for Testing Core Language Models of Domain-Specific Languages. In this paper, a linguistic, rule-based approach for an automatic and traceable translation of semi-structured natural-language requirements into executable test scenarios is presented (see Section A.6 and [57]). In the approach, scenarios are used for the testing of structural properties of DSML core language models (in the paper exemplified via a DSML integration case). To demonstrate the feasibility of the approach, Eclipse Xtext [155] is employed to implement a requirements language for the definition of semi-structured scenarios. Transformation specifications generate executable test scenarios that run in a test platform which is built on EMF and Epsilon.

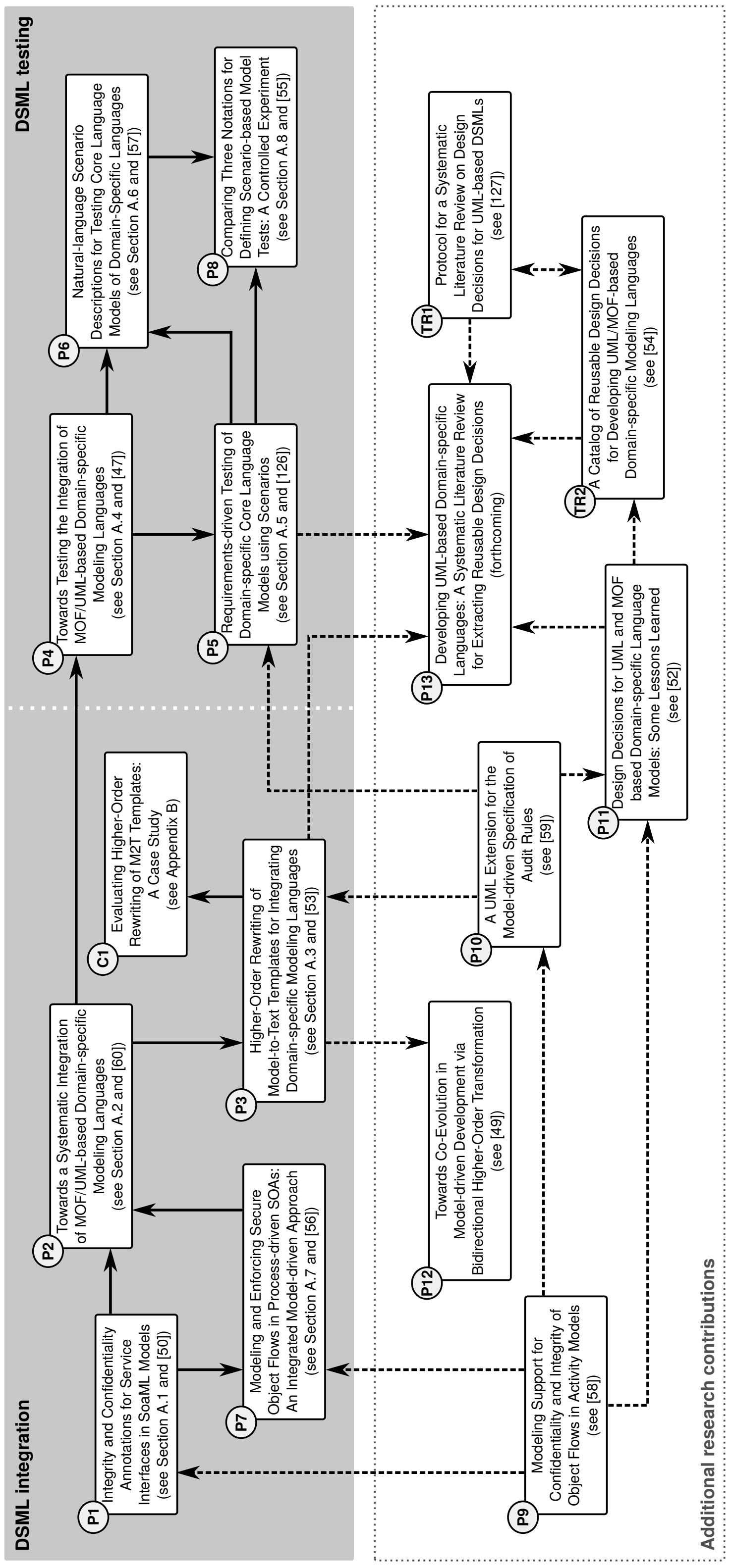
P7 Modeling and Enforcing Secure Object Flows in Process-driven SOAs: An Integrated Model-driven Approach. In this paper, an integrated model-driven approach for the specification and the enforcement of secure object flows in process-driven SOAs is presented (see Section A.7 and [56]). In this context, a secure object flow ensures the confidentiality and the integrity of important objects that are passed between different participants in SOA-based business processes. A formal and generic metamodel for secure object flows is specified that can be used to extend arbitrary process modeling languages. To demonstrate the approach, a UML extension for secure object flows is presented. Moreover, it is described how platform-independent models are mapped to platform-specific software artifacts via automated model transformations. In addition, the paper describes the integration of the approach with the Eclipse Model Development Tools (MDT [149]).

P8 Comparing Three Notations for Defining Scenario-based Model Tests: A Controlled Experiment. In this paper, three alternative notations to define scenario-based tests on structural models are considered: a semi-structured natural-language, a diagrammatic, and a fully-structured textual notation (see Section A.8 and [55]). A controlled experiment to understand how these three notations compare to each other with respect to accuracy and effort of comprehending scenario-test definitions, as well as with respect to the detection of errors in the MUT was performed. The results show that the participants of our study spent comparatively less time and completed the tasks more accurately when using the natural-language notation compared to the other two notations. Moreover, the participants of the study explicitly expressed their preference for the natural-language notation.

C1 Evaluating Higher-Order Rewriting of M2T Templates: A Case Study. In Appendix B, we report on a case study performed in order to evaluate our approach of higher-order rewriting of M2T Templates [53]. In particular, we apply our approach to a publicly available, Epsilon-based open-source project to evaluate its feasibility. Via the case study, we show how our approach facilitates the reuse of M2T templates and reduces the need of manually performed refactorings in metamodel evolution scenarios. We measure the benefits of our approach, for example, via the ratio of automatically to manually executed rewriting operations on EGL-based templates. Furthermore, we present improvements and extensions as well as discuss limitations of our approach and the corresponding software prototype which resulted from the execution of the case study.

5.2 Relations of Research Contributions

In this section, we discuss the relations of the individual research publications contributed to this thesis project. An overview of the established relations between the publications is provided in Figure 5.1. Arrows denote that research contributions from the publication the arrow originates from are influencing contributions to which the arrow is pointing, for example, research results (models, methods, or software artifacts etc.) are used as input to subsequent publications. Research contributions located in the gray filled rectangle contributed to answer the research



DSML integration
DSML testing
Additional research contributions
P1 Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models (see Section A.1 and [50])
P2 Towards a Systematic Integration of MOF/UML-based Domain-specific Modeling Languages (see Section A.2 and [60])
P3 Higher-Order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages (see Section A.3 and [53])
C1 Evaluating Higher-Order Rewriting of M2T Templates: A Case Study (see Appendix B)
P4 Towards Testing the Integration of MOF/UML-based Domain-specific Modeling Languages (see Section A.4 and [47])
P5 Requirements-driven Testing of Domain-specific Core Language Models using Scenarios (see Section A.5 and [126])
P6 Natural-language Scenario Descriptions for Testing Core Language Models of Domain-Specific Languages (see Section A.6 and [57])
P7 Modeling and Enforcing Secure Object Flows in Process-driven SOAs: An Integrated Model-driven Approach (see Section A.7 and [56])
P8 Comparing Three Notations for Defining Scenario-based Model Tests: A Controlled Experiment (see Section A.8 and [55])
P9 Modeling Support for Confidentiality and Integrity of Object Flows in Activity Models (see [58])
P10 A UML Extension for the Model-driven Specification of Audit Rules (see [59])
P11 Design Decisions for UML and MOF based Domain-specific Language Models: Some Lessons Learned (see [52])
P12 Towards Co-Evolution in Model-driven Development via Bidirectional Higher-Order Transformation (see [49])
P13 Developing UML-based Domain-specific Languages: A Systematic Literature Review for Extracting Reusable Design Decisions (forthcoming)
TR1 Protocol for a Systematic Literature Review on Design Decisions for UML-based DSMLs (see [127])
TR2 A Catalog of Reusable Design Decisions for Developing UML/MOF-based Domain-specific Modeling Languages (see [54])

Figure 5.1: Relations of individual research contributions

problems identified in Chapter 3 and, therefore, are included in this thesis project (see Appendix A). In contrast, research contributions framed by the dotted light gray rectangle do not primarily target the research problems of this thesis and, thus, are not considered essential parts of this thesis project (i.e. they represent additional contributions). Nevertheless, these additional research contributions are briefly discussed in Section 5.5. Please note that solid and dashed arrows in Figure 5.1 have identical meanings. The syntactic difference serves only as a visual cue to better distinguish relations to additional research contributions. As Section 5.1 lists the research contributions in their chronological order of publication, the ascending numbering of contributions in Figure 5.1 indicates their publication date (please note that this is not the case for additional research contributions). We classify individual research contributions according to the two research areas DSML integration and DSML testing introduced in Section 1.2 and Chapter 3.

Regarding our research process, we started by designing a UML modeling extension for the support of security properties (i.e. confidentiality and integrity of object flows for activity models; P9 [58]; see Section 5.5). Experiences and knowledge gained from the development of the initial security-related MOF/UML-based DSML influenced the design of a second DSML (i.e. UML extension for the specification of audit rules; P10 [59]; see Section 5.5). For example, knowledge on how to define formal constraints via the OCL or on how to benefit from the employment of different UML extension methods in combination (e.g., MOF-based extension and UML profile definition) was applied. Furthermore, all developed DSMLs are compatible with each other, designed to be used in combination, and are based on and integrated in the same MDD tool chain. This requires to build on former decisions and to incorporate existing designs into development decisions of newly created DSMLs.

The two DSMLs served as first case studies and are considered preparatory work for investigating DSML integration aspects (for more information on the DSMLs see Section 5.5). In a second step, we extended our initially developed DSML (P9) to be employed in process-driven SOAs by integrating it with an existing third-party DSML. For this, we designed integration packages on the modeling-level to render our DSML compatible with SOA-related specifications (P1 [50]). These early DSML developments and prototypical applications matured into an integrated model-driven approach covering all MDD layers via a generic metamodel definition, UML (SOA) extensions (e.g., SoaML, UML4SOA [76]), formal specification and constraints (e.g., first-order logic, OCL), platform-specific artifacts (e.g., WSDL [179], WS-BPEL [102], WS-SecurityPolicy [107]), model transformations (e.g., MDD4SOA [77]), and corresponding tool support (P7 [56]). The case study allowed us to gain insights into several MOF/UML-based DSML integration aspects (e.g., modeling-level integration with SoaML and UML4SOA specifications, integration with MDD4SOA model transformations etc.).

Experiences gained from the development and the integration of DSMLs directed us towards a first definition of a systematic MOF/UML-based DSML integration approach (P2 [60]). By inferring a process model covering all phases of DSML integration, we identified—besides other issues—problem areas concerning the reuse of M2T transformation templates in an integration scenario. We address these shortcomings in P3 [53] and provide a solution to rewrite M2T generator templates by employing HOTs. Our approach is demonstrated by integrating a DSML developed by us (P10) with a generic applicable state/transition-pattern-based DSML. The integrated MOF/UML-based DSML is capable of covering a reactive distributed system with auditing support. An evaluation of the approach published as P3 is performed via a case study presented in C1 (see Appendix B). In the case study, we apply our approach to a third-party project to evaluate its feasibility and to show how our approach facilitates the reuse of M2T templates as well as how it reduces the demand of manually performed refactorings in metamodel evolution scenarios.

In order to support the testing of integrated DSMLs, we reviewed testing techniques for each phase of a MOF/UML-based DSML integration process (as discussed in P2). In P4 [47], we present our results towards testing methods for the integration of MOF/UML-based DSMLs. In doing so, we identified problem areas concerning the validation of DSML integration requirements. Therefore, we developed an approach for the requirements-driven testing of DSML core language models using (integration) scenarios (P5 [126]). Again, we exemplified our approach with the same DSML integration case as in P3. The developed prototype in P5 could validate requirements specified via fully-structured scenarios, but not via natural-language scenarios (i.e. the transformation from natural-language scenarios into executable test scenarios had to be performed manually). Thus, we built an extension to be capable of validating DSML core language models against requirements-level natural-language scenario descriptions (P6 [57]). As before, we motivated our work with the same DSML integration case as in P3 and in P5. An empirical study was performed to evaluate our approach published as P5 and as P6. We conducted a controlled experiment to compare three notations for defining scenario-based model tests (P8 [55]). The results of the experiment support our approach.

In addition to the main research contributions, we also documented our experiences and lessons learned from the development, the integration, and the testing of MOF/UML-based DSMLs in a structured manner (for more information see Section 5.5). For this, we provide a catalog of structured decision descriptions (*decision records*) to support the development of DSMLs. The first collection of extracted design decisions for MOF/UML-based DSMLs was predominantly influenced by lessons learned from developing our own DSMLs. In P11 [52], we documented design decisions and options as well as associations between options for the two phases of defining domain-specific language models (i.e. language model formalization and language model constraints decision points). We expanded this initial collection into a first version of a catalog of reusable decision records for MOF/UML-based DSMLs [51]. This catalog recorded additional DSML decision points and corresponding options (e.g., for the phase of the concrete syntax definition, platform integration) as well as updated decision associations. In addition, we performed a systematic literature review (SLR) on design decisions in MOF/UML-based DSMLs (TR1 [127]). We encoded design decisions gathered via the SLR according to our catalog. In doing so, we were able to broaden our DSML project base and to revise, extend, and validate the decision records in the catalog. The new version of the catalog was published as TR2 [54]. Research results regarding reusable design decision extraction (via a SLR) and documentation (via a catalog of decision records) for the development of MOF/UML-based DSMLs are presented in P13 (the publication is forthcoming).

According to our design-science-oriented research approach (see Chapter 4), an iterative development process for research artifacts was inherent. The evaluation and communication of results triggered further developments of an artifact. For example, our prototype for the scenario-based testing of models (P5) was extended with capabilities to cope with natural-language scenario descriptions after receiving feedback from the research community. Results which emerged from the extension were communicated in a subsequent publication (P6).

5.3 Accompanying Artifacts

Research publications part of this thesis project (see Section 5.1) are accompanied by software developments (e.g., models, examples, prototypes), experimental and case study material, collected data, as well as calculated results. The following artifacts are publicly available at [48]:

- P1
    - The `SOF` UML profile (with accompanying OCL constraints) specified in Eclipse Papyrus [34] providing EMF/XMI serializations of the UML models, graphical information, and tool metadata.
    - The EMF/XMI serialization of the `SOF::Services` UML profile (with accompanying OCL constraints).
    - The credit application example discussed in the paper as EMF/XMI serializations for No Magic's MagicDraw [90], for the EMF (created via a MagicDraw export), and of intermediate object models (specific to the MDD4SOA tool chain [77]) as well as WSDL/WS-BPEL specification documents.
- P3
    - The Eclipse projects `org.eclipse.epsilon.eol.dom`, `org.eclipse.epsilon.eol.dom.ast2dom`, and `org.eclipse.epsilon.eol.dom.printer` which provides code/model round-tripping for the Epsilon Object Language (EOL [70]). This is a branch of the `EpsilonStaticAnalysis` project [174] hosted at EpsilonLabs. Changes made to the `EpsilonStaticAnalysis` source code are provided as a patch.
    - The Eclipse projects `org.eclipse.epsilon.egl.dom`, `org.eclipse.epsilon.egl.dom.ast2dom`, and `org.eclipse.epsilon.egl.dom.printer`

which extends the `org.eclipse.epsilon.eol.dom*` projects (see above) with code/model round-tripping functionality for EGL.

  - The integration scenario discussed in the paper as an Eclipse project providing metamodels, instance models, EGL specifications (as textual and model representations), and the generated Java source code for the two individual DSMLs as well as an Apache Ant [141] orchestration workflow definition, Epsilon Comparison Language (ECL [70]), Epsilon Transformation Language (ETL [70]), and Epsilon Merge Language (EML [70]) integration specifications, metamodels (generated core language model of the integrated DSML, trace metamodel), instance models (of the integrated DSML core language model and the trace metamodel), EGL rewrite specifications, generated ETL rewrite rules, rewritten EGL specifications (as textual and model representations), and the generated Java source code for the integrated DSML.

- P5

  - The patch for the Epsilon Eclipse projects `org.eclipse.epsilon.eol.engine` and `org.eclipse.epsilon.eunit.engine` to extend EUnit with scenario-based testing functionality.
  - The integration scenario discussed in the paper (the same example as in P3) as an Eclipse project providing metamodels (core language models of two individual DSMLs, generated core language model of the integrated DSML), an Apache Ant orchestration workflow definition, ECL, ETL, and EML integration specifications, and EUnit scenario tests.

- P6

  - The Eclipse Xtext grammar specification to define semi-structured natural-language scenarios on the requirements level.
  - The Eclipse project `at.ac.wu.nm.dsml.sbt` which, in combination with an Eclipse Xtext project deployed with the grammar specification mentioned above, supports the definition of semi-structured natural-language test scenarios. The project provides all software artifacts from the integration scenario discussed in P5 (as the paper exemplifies the prototype with the same scenario as P5 does) and, additionally, a requirements specification language metamodel (to be used with the Xtext grammar), natural-language scenario definitions (conforming to the Xtext grammar), the EGL specification for EUnit scenario test generation, the EOL step definitions specification, and generated EUnit scenario tests.
  - The Eclipse project `at.ac.wu.nm.dsml.sbt.regexp` which provides helper functionality in Java to translate steps via regular-expression-based pattern matching methods not natively supported in Epsilon.

- P7
  - The revised EMF/XMI serialization of the `SecureObjectFlows` UML metamodel extension (with revised accompanying OCL constraints) from P9.
  - The OCL constraints for the `SecureObjectFlows::Services` UML metamodel extension.
  - The revised `SOF` UML profile (with revised accompanying OCL constraints) specified in Eclipse Papyrus from P1.
  - The revised EMF/XMI serialization of the `SOF::Services` UML profile (with revised accompanying OCL constraints) from P1.
  - The patch for MDD4SOA [77] Eclipse projects (`eu.mdd4soa.smm`, `eu.mdd4soa.trans.smm2bpel`, `eu.mdd4soa.trans.uml2smm`) to support model transformations for secure object flows.
  - The revised software artifacts from P1 for the credit application example discussed in the paper.
- P8
  - The materials used in the controlled experiment, including the experience questionnaire, the introductory presentation, the notation references, example tasks, and the ex-post questionnaire as PDF documents as well as the Eclipse project `at.ac.wu.nm.dsml.eval.sbt` providing the Ecore-based MUT, the scenario descriptions in the three different notations, and the corresponding questions (with correct answers and, when applicable, line/message numbers in the scenario descriptions responsible for a failure) for the six tasks performed in the experiment.
  - The collected data from the experience/ex-post questionnaires and from the tasks conducted in the experiment as well as the calculated group allocations (based on data gathered from the experience questionnaires) and the analysis of data gathered from performing the experiment's tasks provided in the Open Document Format for Office Applications (i.e. OpenDocument [103, 104, 105, 106]).
- C1
  - The Eclipse project `com.googlecode.pongo.uml` which provides the rewritten UML-compliant M2T transformation specifications as Epsilon scripts (i.e. the resulting files from performing the case study). For comparison, the original Ecore-based M2T transformation definitions are included in the project, as well. With the EGX (a coordination language for EGL templates), EGL, and EOL files included in this project, an Ecore/UML model can be transformed into Java classes for interacting with MongoDB.

- The Eclipse project `com.googlecode.pongo.examples.miniblog.uml` which provides the UML-compliant *Pongo* tutorial files used in the case study: a UML domain model and the generated Java classes as well as a test Java application (i.e. the resulting files from performing the case study). The UML model serves as input to the `com.googlecode.pongo.uml` project for its transformation into Java classes.
- The revised Eclipse projects `org.eclipse.epsilon.egl.dom`, `org.eclipse.epsilon.egl.dom.ast2dom`, and `org.eclipse.epsilon.egl.dom.printer` from P3 which provide extended and improved code/model round-tripping functionality for EGL.
- The Eclipse project `at.ac.wu.nm.dsml.eval.hotm2t` which provides the case-study-specific files of our rewriting approach. This includes the Ecore-to-UML M2M transformation definitions (ECL, EML, ETL, EOL), the trace metamodel and its instance model, the `UMLExcerpt` metamodel, the rewrite rule generator (EGL), the generated rewrite rules (ETL), the original and the rewritten Pongo M2T transformation definitions (EGL specifications as model representations), and the orchestrating Apache Ant workflow definitions.
- The measured execution times (in milliseconds) of the different transformations repeatedly performed (ten times) during the case study provided in the OpenDocument format.

5.4 Adopted Specifications and Software Support

This section describes the formal specifications the approach presented in this thesis builds upon as well as the developed integrated tool support. Furthermore, we show how concepts defined via formal specifications are mapped to our prototypical proof-of-concept implementations.

Figure 5.2 sketches adopted formal specifications relevant to our approach of integrating and testing MOF/UML-based DSMLs. All of the specifications are formally published and maintained by the OMG with the exception of the scenario-based testing domain model which is defined in [133] and the UML4SOA metamodel defined in [76]. The UML Infrastructure specification (UML IS [95]) constitutes the main structural modeling capabilities which are reused by the Essential Meta Object Facility (EMOF [100]), the Complete Meta Object Facility (CMOF [100]), and the UML Superstructure specifications (UML SS [96]). The MOF—which consists of two main packages, EMOF (designed to match the concepts of object oriented programming languages) and CMOF (providing the full metamodeling capabilities of MOF)—as well as the UML Infrastructure specification are defined reflective; i.e., they contain all the metaclasses required to define themselves. The complete UML specification consists of all packages located in the UML Infrastructure and the UML Superstructure specifications which are defined using constructs from the CMOF specification (i.e. both UML specifications are instances of the CMOF metamodel). The definitions of the MOF and the UML metamodels are accompanied by constraints specified in OCL [99] (again an instance of the CMOF). The MOFM2T specification [93] reuses constructs from both, CMOF and OCL and is defined as a CMOF-based model (i.e. an instance-of relationship exists). The XMI specification [101] is also an instance of CMOF as well as the domain model [133] used as basis for the scenario-based testing approach presented in this thesis project. The SoaML specification [98] as well as the UML4SOA metamodel are defined in two ways, as a CMOF-compliant model and as a UML profile, for which both provide mappings between the two model representations.

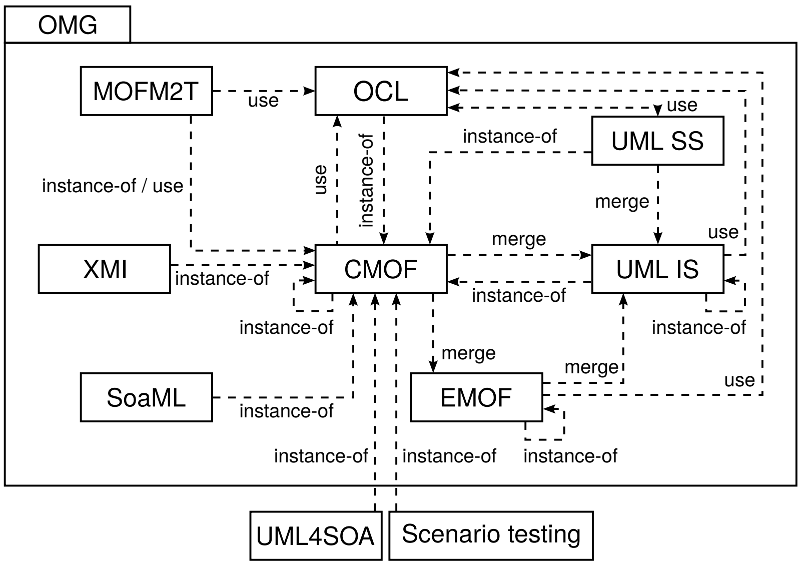


Figure 5.2: Adopted formal specifications (UML IS = UML Infrastructure specification, UML SS = UML Superstructure specification)

Our approach of integrating and testing DSMLs builds on all specifications shown in Figure 5.2. We define all models (e.g., DSML core language models, transformation models) using the MOF and the UML specifications (e.g., MOF-based models, UML profiles). Language model constraints are expressed as OCL statements (accompanied by natural language). Our rewriting technique for integrating DSMLs is applicable to M2T transformation languages which support a subset of the MOFM2T specification. The conceptual model of our scenario-based testing approach is defined in [133]. In a case study [50, 56], we employ the SoaML specification and the UML4SOA modeling extension. The created modeling artifacts (all instances of MOF models) are persistently stored in the XMI format.

The generally defined concepts (using the specifications shown in Figure 5.2) are technically projected into proof-of-concept implementations. In particular, our software artifacts are based on Eclipse to provide for an integrated development environment (see Figure 5.3). Table 5.1 summarizes the correspondences between the specifications from Figure 5.2 and the software artifacts (i.e. technologies, projects) used for their implementation (see Figure 5.3). In essence, we use Ecore (from the EMF project [146]) to approximate EMOF models. The UML and the CMOF specifications are defined as Ecore models in the UML2 project of Eclipse MDT [149]. Eclipse MDT provides also an implementation of the OCL specification. The EMF project supports persistence storage of models by serializing to the XMI format. We use EGL as a MOFM2T-compliant implementation of a M2T transformation language for our rewriting approach [53]. The concepts of scenario-based testing are transformed into a requirements language defined via Xtext (part of the Eclipse Textual Modeling Framework, TMF [153]) as well as into an Epsilon-based syntax by extending the EUnit testing framework [57, 126]. The UML4SOA metamodel is specified as an Ecore model in the `MDD4SOA` project; the SoaML and UML4SOA profiles are provided as Ecore-based UML models.

Table 5.1: Correspondences between specifications and implementations

Formal specification	Implementation vehicle
`EMOF`	`EMF::Ecore`
`CMOF`	`EMF::Ecore`, `MDT::UML2`
`UML IS`, `UML SS`	`EMF::Ecore`, `MDT::UML2`
`OCL`	`MDT::OCL`
`MOFM2T`	`Epsilon::EGL`
`XMI`	`EMF::XMI`
`Scenario testing`	`TMF::Xtext`, `Epsilon::EUnit`
`SoaML`	`EMF::Ecore`, `MDT:UML2`
`UML4SOA`	`EMF::Ecore`, `MDT:UML2`, `MDD4SOA`

Figure 5.3 provides an overview of implementation techniques employed for realizing the integrated software support for our approach of integrating and testing MOF/UML-based DSMLs presented in this thesis. Central to our implementation is the EMF project with its Ecore modeling infrastructure and XMI persistence storage support. We developed our approach of higher-order rewriting of M2T transformations on top of Epsilon using a variety of Epsilon-based language dialects (e.g., the Epsilon Validation Language; EVL [70]). For our approach, we had to extend the `EpsilonStaticAnalysis` project [174] to provide code/model round-tripping for EGL-based M2T templates. The EGL models are stored as Ecore models and are accessible from Epsilon via the Epsilon Model Connectivity (EMC) layer [70]. Our approach for scenario-based testing implements the semi-structured natural-language notation in Xtext [155] and executable scenario tests by extending the EUnit testing framework. The grammar definitions for both syntaxes are using the ANTLR parser generator [108]. Our MDD-based integration case study [50, 56] is based on the `MDD4SOA` project and generates WS-BPEL and WSDL documents as platform-specific artifacts (using the Eclipse BPEL Designer [144] as well as the Web Standard

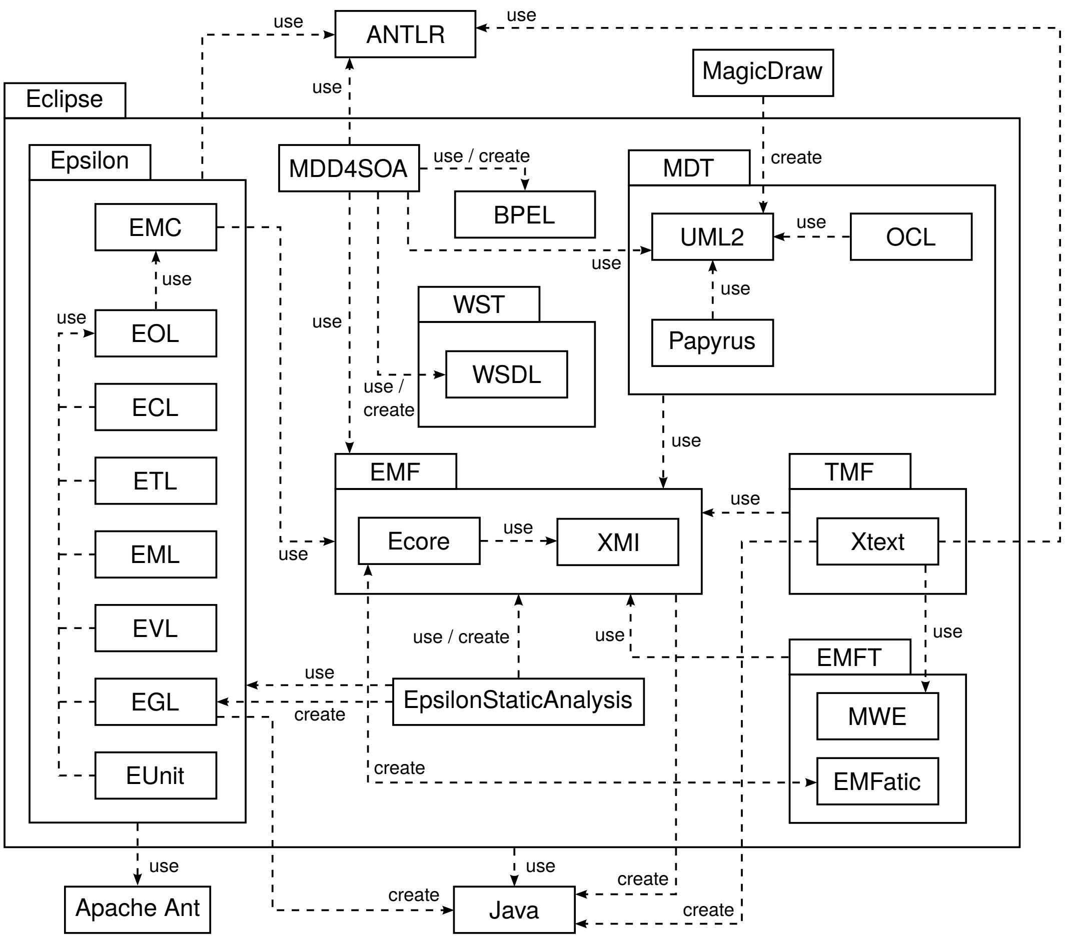


Figure 5.3: Integrated software support based on Eclipse

Tools project; WST [154]). Ecore models were created using editors provided by the EMF project and, for the evaluation of our rewriting approach (see Appendix B), also by using EMFatic—"a textual syntax for EMF Ecore (meta-)models" [148]—from the Eclipse Modeling Framework Technology project (EMFT [147]). For UML models, we employed Eclipse Papyrus [34] (from the MDT project) as well as No Magic's MagicDraw [90] (e.g., to design UML sequence diagrams as one of the three alternative scenario notations used in [55]) both allowing for an EMF/XMI conform serialization export. A console for the interactive evaluation of OCL expressions on models is provided by the MDT project. Orchestration workflows are defined using Apache Ant [141] (e.g., for Epsilon-based model management tasks) and the Modeling Workflow Engine (MWE [151]) from the EMFT project (e.g., to configure the Xtext code generator). The Eclipse integrated development environment (IDE) as well as Eclipse-based projects are implemented, to a large extent, using Java as programming language. Our developments also significantly rely on Java and we generate Java-based platform-specific artifacts in our demonstrator examples and case studies, as well.

5.5 Additional Research Contributions

In addition to the papers presented in Section 5.1, the following closely related contributions emerged during the course of this thesis project.[4] However, the contributions are not included in this thesis because they do not primarily target the research problems identified in Chapter 3.

P9 Modeling Support for Confidentiality and Integrity of Object Flows in Activity Models. This paper is concerned with confidentiality and integrity of object flows [58]. In particular, a UML extension to model confidentiality and integrity of object flows in activity models is presented (named `SecureObjectFlows`). Moreover, the semantics of secure object flows with respect to control nodes is discussed and a formal definition of the corresponding semantics via the OCL is provided.

P10 A UML Extension for the Model-driven Specification of Audit Rules. This paper presents a UML extension for the specification of audit properties [59]. The extension is generic and can be applied to a wide variety of UML elements. In this sense, the approach supports the definition of structural and behavioral perspectives to model different aspects of system audits. In addition to graphical model elements, a fully equivalent textual syntax is provided, as well. In a MDD approach, the extension can be used to generate corresponding audit rules by employing model transformations.

P11 A Catalog of Reusable Design Decisions for Developing UML- and MOF-based Domain-Specific Modeling Languages. This paper documents some of our experiences gathered from developing then MOF/UML-based DSMLs and presents our experiences in a reusable manner via decision templates [52]. In particular, this paper focuses on design decisions for the initial phase of the DSML development process, i.e. the definition of the DSML core language model.

P12 Towards Co-Evolution in Model-driven Development via Bidirectional Higher-Order Transformation. This paper presents first ideas towards an approach to overcome MDD-based problems of co-evolution [49]. In MDD, metamodels, models, and model transformations are interdependent. A change in one artifact must be reflected in all other related artifacts. Regardless of their dependencies, (meta)models and transformations can evolve autonomously rendering referenced artifacts invalid. Coupling the evolution of models to their corresponding metamodels tries to prevent such mismatches, but is currently limited to one-way adaptations

[4]Classification of contributions: P = Paper, TR = Technical Report.

and does not take model transformations into account. To eliminate these shortcomings, the paper discusses the combination of first-class transformation models with bidirectional transformations (BX; see, e.g., [45, 46]). A generic approach is proposed by integrating BX into well-established Eclipse-based MDD tools, thereby neither being restricted to a specific modeling nor model transformation language.

P13 Developing UML-based Domain-specific Languages: A Systematic Literature Review for Extracting Reusable Design Decisions. This paper reports on a research effort to compile and to empirically validate a catalog of structured decision descriptions (decision records) for MOF/UML-based DSMLs (the publication is forthcoming). The catalog (TR2) is based on design decisions extracted from 90 DSML projects. These projects were identified—among others—via an extensive SLR (TR1). The extracted data were evaluated quantitatively (e.g. by running a frequent-item-set analysis to obtain characteristic combinations of design decisions) and qualitatively (e.g. to document recurring documentation issues for UML-based DSMLs).

TR1 Protocol for a Systematic Literature Review on Design Decisions for UML-based DSMLs. This technical report[5] records the process of conducting a SLR of scientific publications documenting the design of MOF/UML-based DSML development projects [127]. The aim of this systematic review is to collect a data set on actual design decisions and decision sequences, to document them, and to evaluate the catalog of decision records (TR2). Results of the SLR are published as P13 and are incorporated into the catalog of decision records. The protocol of the SLR (TR1) as well as the corresponding catalog (TR2) are provided as supplemental materials to P13.

TR2 A Catalog of Reusable Design Decisions for Developing UML/MOF-based Domain-specific Modeling Languages. This technical report documents recurring design decisions collected from 90 MOF/UML-based DSML projects [54]. The findings were gained, on the one hand, from developing ten DSML projects by ourselves. On the other hand, a SLR on the development of MOF/UML-based DSMLs was performed (TR1) to complement and to validate our design decisions. The design decisions are presented in the form of reusable decision records, with each decision record corresponding to a decision point in DSML development processes. The decision records documented in the catalog serve as data basis for analyses conducted in P13. The catalog of decision records (TR2) as well as the corresponding protocol of the SLR (TR1) are provided as supplemental materials to P13.

[5]Please note that technical reports were not subject to an independent peer-review process (i.e. colleagues within the same organization reviewed and revised the reports).

Accompanying Artifacts

These additional research contributions are also accompanied by software developments. The following artifacts are publicly available at [48], as well:

- P9
  - The EMF/XMI serialization of the `SecureObjectFlows` UML metamodel extension (with accompanying OCL constraints). The model was created with the EMF-based implementation of the UML metamodel for the Eclipse platform (provided by Eclipse MDT).
- P12
  - The Eclipse project `org.biglab.groundtram.bx.unqlplus` providing an Ecore-based UnQL$^+$ metamodel and an Eclipse Xtext grammar specification for the UnQL$^+$ textual syntax. With this, the development of UnQL$^+$ BX is supported via textual and model-based syntax editors.[6]

[6]The accompanying software artifact is publicly available at both, [48] and [142].

Chapter 6

Conclusion

In this thesis, an approach for the integration and the testing of MOF/UML-based DSMLs is presented. The integration of DSMLs is based on a rewriting technique for M2T transformations and allows for the automatic refactoring of EGL generator templates to fix important syntactical mismatches between templates and the integrated DSML. The approach is based on HOTs applied on model representations of generator templates and uses trace information generated from metamodel transformations. In this sense, reuse is facilitated in a way that all DSMLs can be used in parallel—either as standalone DSMLs or as an integrated DSML. For our approach of testing an integrated DSML, we use scenarios to define domain requirements on an abstract level (via structured text descriptions). In a subsequent step—by employing refinements and automatic transformations via step definitions—executable scenario tests are derived from the requirements-level scenarios. These executable scenario specifications are then used to test the integrated DSML for compliance with the corresponding domain requirements. Our approach of DSML integration and testing builds on Eclipse EMF and makes use of as well as extends Eclipse-based projects, such as, Epsilon or Xtext.

As a prerequisite for integrating and testing DSMLs, we developed two security-related MOF/UML-based DSMLs. Furthermore, we report our experiences and lessons learned from the development, the integration, and the testing of DSMLs in a structured manner; i.e. in the form of a catalog of reusable decision records to support the development of MOF/UML-based DSMLs. The collected decision options as well as associations between options of one and between options of two decision points in the different phases of the DSML development process were complemented and validated by incorporating results from a SLR. These additional research contributions emerged during the course of this thesis project and are closely related to the presented results (e.g., as preparatory work or as further findings). However, as the publications do not primarily target the research problems this thesis is concerned with (see Chapter 3), they are not included in Appendix A.

This thesis is composed of individual scientific contributions (workshop, conference, and journal articles; P1–P8; see Appendix A). Via the combined research contributions of these individual publications, we identify the problem this thesis is concerned with, define objectives of a solution, present the design and the development of our approach, demonstrate and evaluate its feasibility and usefulness, and communicate the results. The case study report for the evaluation of our higher-order rewriting approach for M2T templates is the only exception, as it was not published besides this thesis, but in Appendix B. All research contributions—whether individually published or as part of this thesis—are results of applying a design science research approach as presented in Chapter 4 and relate to each other as shown in Section 5.2.

In addition, research results are accompanied by software developments (e.g., models, examples, prototypes), experimental and case study material, including collected data as well as calculated results. We placed all accompanying artifacts into the public domain as free/libre open source software (available at [48]).

To summarize the findings of this thesis project, with the development and the evaluation of our approach for rewriting generator templates, we could demonstrate that the reuse of M2T transformation definitions is satisfactory supported by our approach and facilitates the integration of DSMLs (addressing Problem Statement 1; see Chapter 3). Similarly, we showed the feasibility of our approach of using requirements-level scenarios for testing integrated DSMLs. The results of the experimental evaluation indicates that the natural-language-based notation of our approach is recommended as well as most favored by the users who worked with it (by comparing it with two alternative notations; addressing Problem Statement 2; see Chapter 3).

However, this thesis deals exclusively with—and thus, the findings mentioned in the former paragraph are primarily applicable to—MOF/UML-based DSMLs and their technological projection as approximated Ecore-based models, respectively. Although a limiting factor (by definition), we argue that to some extent our research results can be generalized and are transferable to other (non-MOF/UML-based) modeling languages, as well. As the theoretical underpinning of our approach is neither based nor relies on any characteristic implementation—we have chosen suitable development environments for demonstration purposes only—, we could abstract the problem and the solution domains from the concrete software artifacts in our research contributions. For example, we can sufficiently transfer the formalization of the DSML's core language model to generic object-oriented modeling constructs, such as classes, inheritance relationships between classes, references, and attributes, thereby, completely suppressing the MOF/UML as a modeling language.[7] As an example, we provide a definition for a formal and generic metamodel for secure object flows in [56]. This generic metamodel can be used to extend arbitrary modeling languages. Although in the paper we instantiate the generic metamodel as a UML extension, in principle, it is neither specific to the UML nor the MOF. Regarding particular implementation details, for instance, the EMC layer provides abstraction

[7]Of course, some concepts, such as UML profiles, are specific to a modeling language and corresponding equivalents may not exist in other modeling environments.

facilities over concrete modeling technologies (such as, EMF, XML) allowing to interact with models conforming to these technologies in a uniform manner [70]. Thus, our developments can be based on different modeling technologies without any problems. Furthermore, our approach does not rely on any characteristics implied by the chosen domain (here: security). As the domain serves demonstration purposes only, our approach can be applied to other domains in an analogous manner. In addition, we have detailed in Appendix B that the scenario of DSML integration is a special case of model-driven software evolution. Thus, we generalized the problem statement and showed that our rewriting approach supports further (evolutionary) scenarios, as well. Please note in this context that our approach for testing core language models of DSMLs is not limited to any application scenario per se.

In the course of this thesis, we conducted research to achieve objectives derived from the problems identified at the beginning of the project. Beyond that, we recognized various additional areas of research to extend our work. Albeit already partially discussed in other sections of this thesis (and in the individual research contributions) further research directions might also include, but are not restricted to the following.

The current rewriting prototype implementing our approach for *DSML integration* does neither address semantic heterogeneity in metamodel-model relations nor types of syntactical heterogeneity between source and target metamodels which cannot be resolved in an automated manner. In this context, the limitations of the currently employed HOTs as well as recorded transformation trace information must be evaluated. In a further step, the refactoring support may be extended to cover not only M2T generators, but also parts of the generated code (e.g., by adopting concepts from *language workbenches*; see, e.g., [30, 170, 171]), for instance, for the generation of glue code. Quality aspects of our prototype neglected so far, such as, usability or stability, need empirical evaluation, for example, by conducting experiments and/or case studies.

In the area of requirements-driven *DSML testing*, our approach uses semi-structured text artifacts defined via natural-language as scenario notation. The development of an equivalent graphical notation (e.g., similar to UML use cases) may be a benefit. Currently, no established repository of step definitions exists to map natural-language scenarios to executable scenario tests. Such a repository may facilitate the uptake of requirements-driven and natural-language-based test approaches for DSML core language models. In this context, the review and development of additional test techniques to cover other artifacts than the DSML core language model (e.g., to test generated code) is also a candidate for further research. To deepen the understanding on how different scenario notations compare to each other, a replication study of [55] with participants from a different professional background may be conducted to be able to draw more general conclusions (including extended quantitative and qualitative analyses of obtained data).

In this thesis, we focus on a language model driven DSML engineering process [137, 182], thereby factoring out other development variants (e.g., mockup language driven DSML development or extracting the DSML from an existing system). Studying the implications and consequences of employing variants of DSML engineering processes on all research areas discussed in this thesis project is a candidate for further research, as well.

Bibliography

[1] D. Amyot and A. Eberlein. An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. *Telecommunication Systems*, 24(1):61–94, 2003.

[2] V. Basili and D. Weiss. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, Nov. 1984.

[3] D. Basin, J. Doser, and T. Lodderstedt. Model Driven Security: From UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, Jan. 2006.

[4] 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*, 53(6):139–143, June 2010.

[5] I. Benbasat, D. K. Goldstein, and M. Mead. The Case Research Strategy in Studies of Information Systems. *MIS Quarterly*, 11(3):369–386, Sept. 1987.

[6] J. Bettin. Measuring the Potential of Domain-Specific Modelling Techniques. In *Proceedings of the 2nd Workshop on Domain-Specific Visual Languages*, pages 39–44. Helsinki School of Economics, 2002.

[7] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[8] P. A. Brooks and A. M. Memon. Automated GUI Testing Guided by Usage Profiles. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 333–342. ACM, 2007.

[9] P. Brosch, U. Egly, S. Gabmeyer, G. Kappel, M. Seidl, H. Tompits, M. Widl, and M. Wimmer. Towards Scenario-Based Testing of UML Diagrams. In A. Brucker and J. Julliand, editors, *Tests and Proofs*, volume 7305 of *Lecture Notes in Computer Science*, pages 149–155. Springer, 2012.

[10] J. Bruck and K. Hussey. Customizing UML: Which Technique is Right for You? Available at: `http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html`, 2008.

[11] E. Cariou, C. Ballagny, A. Feugas, and F. Barbier. Contracts for Model Execution Verification. In R. France, J. Kuester, B. Bordbar, and R. Paige, editors, *Modelling Foundations and Applications*, volume 6698 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2011.

[12] J. Carroll. Five Reasons for Scenario-based Design. *Interacting with Computers*, 13(1):43–60, 2000.

[13] J. M. Carroll, editor. *Scenario-based Design: Envisioning Work and Technology in System Development.* John Wiley & Sons, 1995.

[14] W. Cazzola and D. Poletti. DSL Evolution Through Composition. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 6:1–6:6. ACM, 2010.

[15] P. P.-S. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, Mar. 1976.

[16] A. Cicchetti, D. D. Ruscio, A. Pierantonio, and D. S. Kolovos. A Test-Driven Approach for Metamodel Development. In J. Rech and C. Bunse, editors, *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 319–342. IGI Global, 2012.

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

[18] J. S. Cuadrado, E. Guerra, and J. de Lara. Generic Model Transformations: Write Once, Reuse Everywhere. In J. Cabot and E. Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2011.

[19] J. S. Cuadrado, E. Guerra, and J. de Lara. Flexible Model-to-Model Transformation Templates: An Application to ATL. *Journal of Object Technology*, 11(2):4:1–28, Aug. 2012.

[20] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, July 2006.

[21] A. Dwarakanath and S. Sengupta. Litmus: Generation of Test Cases from Functional Requirements in Natural Language. In G. Bouma, A. Ittoo, E. Métais, and H. Wortmann, editors, *Natural Language Processing and Information Systems*, volume 7337 of *Lecture Notes in Computer Science*, pages 58–69. Springer, 2012.

[22] M. Emerson and J. Sztipanovits. Techniques for Metamodel Composition. In *Proceedings of the 6th Workshop on Domain-Specific Modeling*, pages 123–139. ACM, 2006.

[23] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.

[24] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Company, 2nd edition, 1997.

[25] E. Fernández-Medina, J. Trujillo, R. Villarroel, and M. Piattini. Access Control and Audit Model for the Multidimensional Modeling of Data Warehouses. *Decision Support Systems*, 42(3):1270–1289, Dec. 2006.

[26] T. Fink, M. Koch, and K. Pauls. An MDA Approach to Access Control Specifications Using MOF and UML Profiles. *Electronic Notes in Theoretical Computer Science*, 142:161–179, Jan. 2006.

[27] R. G. Flatscher. Metamodeling in EIA/CDIF—Meta-Metamodel and Metamodels. *ACM Transactions on Modeling and Computer Simulation*, 12(4):322–342, Oct. 2002.

[28] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A Generic Approach for Automatic Model Composition. In H. Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 7–15. Springer, 2008.

[29] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[30] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Available at: `http://martinfowler.com/articles/languageWorkbench.html`, June 2005.

[31] M. Fowler and R. P. und Josh MacKenzie. POJO. Available at: `http://www.martinfowler.com/bliki/POJO.html`, 2000.

[32] P. Gagnonand, F. Mokhati, and M. Badri. Applying Model Checking to Concurrent UML Models. *Journal of Object Technology*, 7(1):59–84, 2008.

[33] A. García-Domínguez, D. Kolovos, L. Rose, R. Paige, and I. Medina-Bulo. EUnit: A Unit Testing Framework for Model Management Tasks. In J. Whittle, T. Clark, and T. Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2011.

[34] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic. Papyrus: A UML2 Tool for Domain-Specific Language Modeling. In H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, pages 361–368. Springer, 2010.

[35] V. Gervasi and B. Nuseibeh. Lightweight Validation of Natural Language Requirements. *Software: Practice and Experience*, 32(2):113–133, 2002.

[36] S. Gilmore, L. Gönczy, N. Koch, P. Mayer, M. Tribastone, and D. Varró. Non-Functional Properties in the Model-driven Development of Service-oriented Systems. *Software & Systems Modeling*, 10(3):287–311, 2011.

[37] A. Goknil, I. Kurtev, and K. van den Berg. A Metamodeling Approach for Reasoning about Requirements. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 310–325. Springer, 2008.

[38] J. J. C. Gomez, B. Baudry, and H. Sahraoui. Searching the Boundaries of a Modeling Space to Test Metamodels. In *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation*, pages 131–140. IEEE, 2012.

[39] C. Gonzalez-Perez and B. Henderson-Sellers. *Metamodelling for Software Engineering*. John Wiley & Sons, 2008.

[40] E. Guerra, J. de Lara, D. Kolovos, R. Paige, and O. dos Santos. Engineering Model Transformations with transML. *Software & Systems Modeling*, 12(3):555–577, 2013.

[41] M. Hafner and R. Breu. *Security Engineering for Service-Oriented Architectures*. Springer, 2009.

[42] D. Hatebur and M. Heisel. Making Pattern- and Model-Based Software Development More Rigorous. In J. Dong and H. Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2010.

[43] A. Hevner and S. Chatterjee. *Design Research in Information Systems: Theory and Practice*, volume 22 of *Integrated Series in Information Systems*. Springer, 2010.

[44] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, Mar. 2004.

[45] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 480–483, Nov. 2011.

[46] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. *Progress in Informatics*, 10:131–148, Mar. 2013.

[47] B. Hoisl. Towards Testing the Integration of MOF/UML-based Domain-specific Modeling Languages. In *Proceedings of the 8th IASTED International Conference on Advances in Computer Science*, pages 314–323. ACTA Press, 2013.

[48] B. Hoisl. Domain-Specific Languages for Model-Driven Security Engineering (ModSec). Available at: `http://nm.wu.ac.at/modsec`, 2014.

[49] B. Hoisl, Z. Hu, and S. Hidaka. Towards Co-Evolution in Model-driven Development via Bidirectional Higher-Order Transformation. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 466–471. SciTePress, 2014.

[50] B. Hoisl and S. Sobernig. Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models. In *Proceedings of the International Workshop on Security Aspects of Process-aware Information Systems*, pages 673–679. IEEE, 2011.

[51] 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 University of Economics and Business, 2012/01.

[52] 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 *Proceedings of the 2nd Workshop on Process-based Approaches for Model-Driven Engineering*, pages 303–314. Technical University of Denmark (DTU), 2012.

[53] B. Hoisl, S. Sobernig, and M. Strembeck. Higher-Order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages. In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pages 49–61. SciTePress, 2013.

[54] B. Hoisl, S. Sobernig, and M. Strembeck. A Catalog of Reusable Design Decisions for Developing UML/MOF-based Domain-specific Modeling Languages. Available at: `http://epub.wu.ac.at/4312/`, 2014. Technical Reports / Institute for Information Systems and New Media, WU Vienna University of Economics and Business, 2014/03.

[55] B. Hoisl, S. Sobernig, and M. Strembeck. Comparing Three Notations for Defining Scenario-based Model Tests: A Controlled Experiment. In *Proceedings of the 9th International Conference on the Quality of Information and Communications Technology*, pages 95–104. IEEE, 2014.

[56] B. Hoisl, S. Sobernig, and M. Strembeck. Modeling and Enforcing Secure Object Flows in Process-driven SOAs: An Integrated Model-driven Approach. *Software & Systems Modeling*, 13(2):513–548, 2014.

[57] B. Hoisl, S. Sobernig, and M. Strembeck. Natural-language Scenario Descriptions for Testing Core Language Models of Domain-Specific Languages. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 356–367. SciTePress, 2014.

[58] B. Hoisl and M. Strembeck. Modeling Support for Confidentiality and Integrity of Object Flows in Activity Models. In *Proceedings of the 14th International Conference on Business Information Systems*, pages 278–289. Springer, 2011.

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

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

[61] Institute of Electrical and Electronics Engineers (IEEE). Systems and Software Engineering – Vocabulary. Available at: `http://standards.ieee.org/findstds/standard/24765-2010.html`, 2010. ISO/IEC/IEEE 24765:2010.

[62] Institute of Electrical and Electronics Engineers (IEEE). Systems and Software Engineering – Life Cycle Processes – Requirements Engineering. Available at: `http://standards.ieee.org/findstds/standard/29148-2011.html`, 2011. ISO/IEC/IEEE 29148:2011.

[63] M. Jarke, X. T. Bui, and J. M. Carroll. Scenario Management: An Interdisciplinary Approach. *Requirements Engineering*, 3(3-4):155–173, 1998.

[64] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting Experiments in Software Engineering. In F. Shull, J. Singer, and D. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008.

[65] J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[66] G. M. Kapitsaki, D. A. Kateros, G. N. Prezerakos, and I. S. Venieris. Model-driven Development of Composite Context-aware Web Applications. *Information and Software Technology*, 51(8):1244–1260, 2009.

[67] L. Kapová, T. Goldschmidt, S. Becker, and J. Henss. Evaluating Maintainability with Code Metrics for Model-to-Model Transformations. In G. Heineman, J. Kofron, and F. Plasil, editors, *Research into Practice – Reality and Gaps*, volume 6093 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2010.

[68] B. Kitchenham, H. Al-Khilidar, M. Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu. Evaluating Guidelines for Reporting Empirical Software Engineering Studies. *Empirical Software Engineering*, 13(1):97–121, 2008.

[69] D. Kolovos. Pongo: 5 Minute Tutorial. Available at: `https://code.google.com/p/pongo/wiki/5MinuteTutorial`, 2014.

[70] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige. The Epsilon Book. Available at: `http://www.eclipse.org/epsilon/doc/book/`, 2014.

[71] D. Kolovos and J. R. Williams. Pongo: Java POJO generator for MongoDB. Available at: `https://code.google.com/p/pongo/`, 2014.

[72] D. S. Kolovos. An Overview of the Epsilon Profiling Tools. Available at: `https://www.eclipse.org/epsilon/doc/EpsilonProfilingTools.pdf`, 2007. Technical Report, The University of York.

[73] J. Krogstie. *Model-Based Development and Evolution of Information Systems: A Quality Approach*. Springer, 2012.

[74] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[75] S. T. March and G. F. Smith. Design and Natural Science Research on Information Technology. *Decision Support Systems*, 15(4):251–266, 1995.

[76] P. Mayer, N. Koch, A. Schroeder, and A. Knapp. The UML4SOA Profile. Available at: `http://www.uml4soa.eu/wp-content/uploads/uml4soa.pdf`, 2010. Technical Report, LMU München.

[77] P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-Driven Service Orchestration. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 203–212, Sept. 2008.

[78] S. Mellor, A. Clark, and T. Futagami. Model-Driven Development – Guest Editor's Introduction. *IEEE Software*, 20(5):14–18, Sept. 2003.

[79] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a Goal-driven Approach to Generate Test Cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 257–266. ACM, 1999.

[80] A. Meneely, B. Smith, and L. Williams. Validating Software Metrics: A Spectrum of Philosophies. *ACM Transactions on Software Engineering and Methodology*, 21(4):24:1–24:28, Feb. 2013.

[81] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, Mar. 2006.

[82] J. Merilinna and J. Pärssinen. Verification and Validation in the Context of Domain-specific Modelling. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, pages 9:1–9:6. ACM, 2010.

[83] J. Merilinna, O.-P. Puolitaival, and J. Pärssinen. Towards Model-Based Testing of Domain-Specific Modelling Languages. In *Proceedings of the 8th Workshop on Domain-Specific Modeling*. ACM, 2008.

[84] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, Dec. 2005.

[85] B. Meyers and H. Vangheluwe. A Framework for Evolution of Modelling Languages. *Science of Computer Programming*, 76(12):1223–1246, Dec. 2011.

[86] MongoDB, Inc. MongoDB. Available at: `http://www.mongodb.org/`, 2014.

[87] D. Moody and J. van Hillegersberg. Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams. In D. Gašević, R. Lämmel, and E. Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2009.

[88] G. J. Myers, T. Badgett, and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[89] C. Nebut, F. Fleurey, Y. Le-Traon, and J.-M. Jezequel. Automatic Test Generation: A Use Case Driven Approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, Mar. 2006.

[90] No Magic, Inc. MagicDraw. Available at: `http://www.nomagic.com/products/magicdraw.html`, 2014.

[91] R. Nord and J. E. Tomayko. Software Architecture-Centric Methods and Agile Development. *IEEE Software*, 23(2):47–53, Mar. 2006.

[92] Object Management Group (OMG). MDA Guide. Available at: `http://www.omg.org/cgi-bin/doc?omg/03-06-01`, June 2003. Version 1.0.1, omg/2003-06-01.

[93] Object Management Group (OMG). MOF Model to Text Transformation Language. Available at: `http://www.omg.org/spec/MOFM2T`, Jan. 2008. Version 1.0, formal/2008-01-16.

[94] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Available at: `http://www.omg.org/spec/QVT`, Jan. 2011. Version 1.1, formal/2011-01-01.

[95] Object Management Group (OMG). OMG Unified Modeling Language (OMG UML), Infrastructure. Available at: `http://www.omg.org/spec/UML`, Aug. 2011. Version 2.4.1, formal/2011-08-05.

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

[97] Object Management Group (OMG). OMG Systems Modeling Language (OMG SysML). Available at: `http://www.omg.org/spec/SysML`, June 2012. Version 1.3, formal/2012-06-01.

[98] Object Management Group (OMG). Service oriented architecture Modeling Language (SoaML) Specification. Available at: `http://www.omg.org/spec/SoaML`, May 2012. Version 1.0.1, formal/2012-05-10.

[99] Object Management Group (OMG). Object Constraint Language. Available at: `http://www.omg.org/spec/OCL`, Feb. 2014. Version 2.4, formal/2014-02-03.

[100] Object Management Group (OMG). OMG Meta Object Facility (MOF) Core Specification. Available at: `http://www.omg.org/spec/MOF`, Apr. 2014. Version 2.4.2, formal/2014-04-03.

[101] Object Management Group (OMG). XML Metadata Interchange (XMI) Specification. Available at: `http://www.omg.org/spec/XMI/`, Apr. 2014. Version 2.4.2, formal/2014-04-04.

[102] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Business Process Execution Language. Available at: `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`, Apr. 2007. Version 2.0.

[103] Organization for the Advancement of Structured Information Standards (OASIS). Open Document Format for Office Applications (OpenDocument). Available at: `http://docs.oasis-open.org/office/v1.2/os/OpenDocument-v1.2-os.pdf`, Sept. 2011. Version 1.2.

[104] Organization for the Advancement of Structured Information Standards (OASIS). Open Document Format for Office Applications (OpenDocument), Part 1: OpenDocument Schema. Available at: `http://docs.oasis-open.org/office/v1.2/os/OpenDocument-v1.2-os-part1.pdf`, Sept. 2011. Version 1.2.

[105] Organization for the Advancement of Structured Information Standards (OASIS). Open Document Format for Office Applications (OpenDocument), Part 2: Recalculated Formula (OpenFormula) Format. Available at: `http://docs.oasis-open.org/office/v1.2/os/OpenDocument-v1.2-os-part2.pdf`, Sept. 2011. Version 1.2.

[106] Organization for the Advancement of Structured Information Standards (OASIS). Open Document Format for Office Applications (OpenDocument), Part 3: Packages. Available at: `http://docs.oasis-open.org/office/v1.2/os/OpenDocument-v1.2-os-part3.pdf`, Sept. 2011. Version 1.2.

[107] Organization for the Advancement of Structured Information Standards (OASIS). WS-SecurityPolicy 1.3. Available at: `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/errata01/os/ws-securitypolicy-1.3-errata01-os-complete.pdf`, Apr. 2012. Version 1.3.

[108] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[109] H. Partsch and R. Steinbrüggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3):199–236, Sept. 1983.

[110] L. Pedro, M. Risoldi, D. Buchs, B. Barroca, and V. Amaral. Composing Visual Syntax for Domain Specific Languages. In J. Jacko, editor, *Human-Computer Interaction. Novel Interaction Methods and Techniques*, volume 5611 of *Lecture Notes in Computer Science*, pages 889–898. Springer, 2009.

[111] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3):45–77, Dec. 2007.

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

[113] G. Perrouin, G. Vanwormhoudt, B. Morin, P. Lahire, O. Barais, and J.-M. Jézéquel. Weaving Variability into Domain Metamodels. *Software & Systems Modeling*, 11(3):361–383, 2012.

[114] A. Rodríguez, E. Fernández-Medina, J. Trujillo, and M. Piattini. Secure Business Process Model Specification Through a UML 2.0 Activity Diagram Profile. *Decision Support Systems*, 51(3):446–465, June 2011.

[115] L. Rose, N. Matragkas, D. Kolovos, and R. Paige. A Feature Model for Model-to-Text Transformation Languages. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 57–63, June 2012.

[116] L. Rose, R. Paige, D. Kolovos, and F. Polack. The Epsilon Generation Language. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[117] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[118] J. Ryser and M. Glinz. A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts. In *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications*, 1999.

[119] D. Sadilek and S. Weißleder. Testing Metamodels. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2008.

[120] V. d. Santiago Júnior and N. Vijaykumar. Generating Model-based Test Cases from Natural Language Requirements for Space Application Software. *Software Quality Journal*, 20(1):77–143, 2012.

[121] S. Schefer and M. Strembeck. Modeling Support for Delegating Roles, Tasks, and Duties in a Process-Related RBAC Context. In C. Salinesi and O. Pastor, editors, *Advanced Information Systems Engineering Workshops*, volume 83 of *LNBIP*, pages 660–667. Springer, 2011.

[122] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, Feb. 2006.

[123] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, Sept. 2003.

[124] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, Sept. 2003.

[125] H. A. Simon. *The Sciences of the Artificial.* MIT Press, 3rd edition, 1996.

[126] S. Sobernig, B. Hoisl, and M. Strembeck. Requirements-driven Testing of Domain-specific Core Language Models using Scenarios. In *Proceedings of the 13th International Conference on Quality Software*, pages 163–172. IEEE, 2013.

[127] S. Sobernig, B. Hoisl, and M. Strembeck. Protocol for a Systematic Literature Review on Design Decisions for UML-based DSMLs. Available at: `http://epub.wu.ac.at/4311/`, 2014. Technical Reports / Institute for Information Systems and New Media, WU Vienna University of Economics and Business, 2014/02.

[128] S. S. Somé. A Meta-Model for Textual Use Case Description. *Journal of Object Technology*, 8(7):87–106, Nov. 2009.

[129] I. Sommerville. *Software Engineering.* Addison-Wesley, 9th edition, 2010.

[130] D. Spinellis. Notable Design Patterns for Domain-specific Languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.

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

[132] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework.* Addison-Wesley, 2nd edition, 2008.

[133] M. Strembeck. Testing Policy-based Systems with Scenarios. In *Proceedings of the 10th IASTED International Conference on Software Engineering*, pages 64–71. ACTA Press, 2011.

[134] M. Strembeck and J. Mendling. Generic Algorithms for Consistency Checking of Mutual-Exclusion and Binding Constraints in a Business Process Context. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems*, volume 6426 of *Lecture Notes in Computer Science*, pages 204–221. Springer, 2010.

[135] M. Strembeck and J. Mendling. Modeling Process-related RBAC Models with Extended UML Activity Models. *Information and Software Technology*, 53(5):456–483, 2011.

[136] M. Strembeck and U. Zdun. Scenario-based Component Testing Using Embedded Metadata. In *Proceedings of the Workshop on Testing of Component-based Systems*, pages 31–45. Lecture Notes in Informatics, Sept. 2004.

[137] M. Strembeck and U. Zdun. An Approach for the Systematic Development of Domain-specific Languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.

[138] A. Sujeeth, T. Rompf, K. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and Reuse with Compiled Domain-Specific Languages. In G. Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2013.

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

[140] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. A Fundamental Approach to Model Versioning based on Graph Modifications: From Theory to Implementation. *Software & Systems Modeling*, 13(1):239–272, 2014.

[141] The Apache Software Foundation. The Apache Ant Project. Available at: `http://ant.apache.org/`, 2014.

[142] The BiG Team. The BiG Project: A Grand Challenge Project on Bidirectional Graph (Model) Transformation at NII in Japan. Available at: `http://www.biglab.org/`, 2014.

[143] The Eclipse Foundation. Acceleo. Available at: `https://www.eclipse.org/acceleo/`, 2014.

[144] The Eclipse Foundation. BPEL Designer Project. Available at: `https://www.eclipse.org/bpel/`, 2014.

[145] The Eclipse Foundation. Eclipse Community Forums. Available at: `https://www.eclipse.org/forums/`, 2014.

[146] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF). Available at: `https://www.eclipse.org/modeling/emf/`, 2014.

[147] The Eclipse Foundation. Eclipse Modeling Framework Technology (EMFT). Available at: `https://www.eclipse.org/modeling/emft/`, 2014.

[148] The Eclipse Foundation. EMFatic: A Textual Syntax for EMF Ecore (Meta-)Models. Available at: `https://www.eclipse.org/emfatic/`, 2014.

[149] The Eclipse Foundation. Model Development Tools (MDT). Available at: `https://www.eclipse.org/modeling/mdt/`, 2014.

[150] The Eclipse Foundation. Model To Text (M2T). Available at: `https://www.eclipse.org/modeling/m2t/`, 2014.

[151] The Eclipse Foundation. Modeling Workflow Engine. Available at: `https://www.eclipse.org/modeling/emft/?project=mwe`, 2014.

[152] The Eclipse Foundation. MOFScript. Available at: `https://www.eclipse.org/gmt/mofscript/`, 2014.

[153] The Eclipse Foundation. Textual Modeling Framework. Available at: `https://www.eclipse.org/modeling/tmf/`, 2014.

[154] The Eclipse Foundation. Web Standard Tools Subproject. Available at: `https://www.eclipse.org/webtools/wst/main.php`, 2014.

[155] The Eclipse Foundation. Xtext – Language Development Made Easy. Available at: `https://www.eclipse.org/Xtext/`, 2014.

[156] M. Tisi, J. Cabot, and F. Jouault. Improving Higher-Order Transformations Support in ATL. In L. Tratt and M. Gogolla, editors, *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2010.

[157] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In R. Paige, A. Hartman, and A. Rensink, editors, *Model Driven Architecture – Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.

[158] A. Tiso, G. Reggio, and M. Leotta. A Method for Testing Model to Text Transformations. In B. Baudry, J. Dingel, L. Lucio, and H. Vangheluwe, editors, *Proceedings of the 2nd Workshop on the Analysis of Model Transformations*, volume 1077 of *CEUR-WS.org.* CEUR Workshop Proceedings, 2013.

[159] S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, Feb. 2003.

[160] A. Ulrich, E.-H. Alikacem, H. Hallal, and S. Boroday. From Scenarios to Test Implementations via Promela. In A. Petrenko, A. Simão, and J. Maldonado, editors, *Testing Software and Systems*, volume 6435 of *Lecture Notes in Computer Science*, pages 236–249. Springer, 2010.

[161] A. Vallecillo. On the Combination of Domain Specific Modeling Languages. In T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, editors, *Modelling Foundations and Applications*, volume 6138 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2010.

[162] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann. Formal Specification and Testing of Model Transformations. In M. Bernardo, V. Cortellessa, and A. Pierantonio, editors, *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 399–437. Springer, 2012.

[163] M. van Amstel. The Right Tool for the Right Job: Assessing Model Transformation Quality. In *Proceedings of the 34th Annual IEEE Computer Software and Applications Conference Workshops*, pages 69–74, July 2010.

[164] M. van Amstel, C. Lange, and M. van den Brand. Using Metrics for Assessing the Quality of ASF+SDF Model Transformations. In R. Paige, editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 239–248. Springer, 2009.

[165] M. van Amstel and M. van den Brand. Quality Assessment of ATL Model Transformations using Metrics. In M. D. D. Fabro, F. Jouault, and I. Kurtev, editors, *Proceedings of the 2nd International Workshop on Model Transformation with ATL*, volume 711 of *CEUR-WS.org.* CEUR Workshop Proceedings, 2010.

[166] M. van Amstel and M. van den Brand. Using Metrics for Assessing the Quality of ATL Model Transformations. In I. Kurtev, M. Tisi, and D. Wagelaar, editors, *Proceedings of the 3rd International Workshop on Model Transformation with ATL*, volume 742 of *CEUR-WS.org.* CEUR Workshop Proceedings, 2011.

[167] R. van Solingen and E. Berghout. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development.* McGraw-Hill, 1999.

[168] D. Varró and A. Pataricza. Generic and Meta-transformations for Model Transformation Engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, *«UML» 2004 – The Unified Modeling Language. Modeling Languages and Applications*, volume 3273 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 2004.

[169] A. Vignaga. Metrics for Measuring ATL Model Transformations. Available at: `http://swp.dcc.uchile.cl/TR/2009/TR_DCC-20090430-006.pdf`, 2009. Technical Report, Universidad de Chile.

[170] M. Völter. From Programming to Modeling – and Back Again. *IEEE Software*, 28(6):20–25, Nov. 2011.

[171] M. Völter, D. Ratiu, B. Kolb, and B. Schätz. mbeddr: Instantiating a Language Workbench in the Embedded Software Domain. *Automated Software Engineering*, 20(3):339–390, 2013.

[172] D. Wagelaar. Composition Techniques for Rule-Based Model Transformation Languages. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2008.

[173] T. Walter and J. Ebert. Combining DSLs and Ontologies Using Metamodel Integration. In W. Taha, editor, *Domain-Specific Languages*, volume 5658 of *Lecture Notes in Computer Science*, pages 148–169. Springer, 2009.

[174] W. Wei. EpsilonLabs: Epsilon Static Analysis. Available at: `https://code.google.com/p/epsilonlabs/wiki/EpsilonStaticAnalysis`, 2012.

[175] J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale, and D. C. Schmidt. Improving Domain-Specific Language Reuse with Software Product Line Techniques. *IEEE Software*, 26(4):47–53, July 2009.

[176] M. Wimmer and L. Burgueño. Testing M2T/T2M Transformations. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 203–219. Springer, 2013.

[177] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Towards an Expressivity Benchmark for Mappings Based on a Systematic Classification of Heterogeneities. In *Proceedings of the 1st International Workshop on Model-Driven Interoperability*, pages 32–41. ACM, 2010.

[178] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Fact or Fiction – Reuse in Rule-Based Model-to-Model Transformation Languages. In Z. Hu and J. de Lara, editors, *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2012.

[179] World Wide Web Consortium (W3C). Web Services Description Language (WSDL). Available at: `http://www.w3.org/TR/wsdl`, Mar. 2001. Version 1.1.

[180] M. Wynne and A. Hellesøy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.

[181] T. Yue, L. C. Briand, and Y. Labiche. Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments. *ACM Transactions on Software Engineering and Methodology*, 22(1):5:1–5:38, Mar. 2013.

[182] U. Zdun and M. Strembeck. Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Development. In *Proceedings of the 14th European Conference on Pattern Languages of Programs*, 2009.

[183] Y. Zhang, Y. Liu, L. Zhang, Z. Ma, and H. Mei. Modeling and Checking for Non-Functional Attributes in Extended UML Class Diagram. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*, pages 100–107, July 2008.

Appendix A

Publications

Within the scope of this thesis, research results were originally published as individual contributions (see Table A.1). The main research findings were published as one workshop article, six conference articles, and one journal article (see Sections A.1–A.8).[8] This chapter lists the individual research contributions in their chronological order of publication.

Table A.1: Overview of publications

#	Section	Publication
P1	A.1	B. Hoisl and S. Sobernig. Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models. In *Proceedings of the International Workshop on Security Aspects of Process-aware Information Systems*, pages 673–679. IEEE, 2011 (see [50]).
P2	A.2	B. Hoisl, M. Strembeck, and S. Sobernig. Towards a Systematic Integration of MOF/UML-based Domain-specific Modeling Languages. In *Proceedings of the 16th IASTED International Conference on Software Engineering and Applications*, pages 337–344. ACTA Press, 2012 (see [60]).
P3	A.3	B. Hoisl, S. Sobernig, and M. Strembeck. Higher-Order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages. In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pages 49–61. SciTePress, 2013 (see [53]).
P4	A.4	B. Hoisl. Towards Testing the Integration of MOF/UML-based Domain-specific Modeling Languages. In *Proceedings of the 8th IASTED International Conference on Advances in Computer Science*, pages 314–323. ACTA Press, 2013 (see [47]).

[8] In addition, one main research contribution (C1) was not submitted to any scientific workshop, conference, or journal and is published in Appendix B.

#	Section	Publication
P5	A.5	S. Sobernig, B. Hoisl, and M. Strembeck. Requirements-driven Testing of Domain-specific Core Language Models using Scenarios. In *Proceedings of the 13th International Conference on Quality Software*, pages 163–172. IEEE, 2013 (see [126]).
P6	A.6	B. Hoisl, S. Sobernig, and M. Strembeck. Natural-language Scenario Descriptions for Testing Core Language Models of Domain-Specific Languages. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 356–367. SciTePress, 2014 (see [57]).
P7	A.7	B. Hoisl, S. Sobernig, and M. Strembeck. Modeling and Enforcing Secure Object Flows in Process-driven SOAs: An Integrated Model-driven Approach. *Software & Systems Modeling*, 13(2):513–548, 2014 (see [56]).
P8	A.8	B. Hoisl, S. Sobernig, and M. Strembeck. Comparing Three Notations for Defining Scenario-based Model Tests: A Controlled Experiment. In *Proceedings of the 9th International Conference on the Quality of Information and Communications Technology*, pages 95–104. IEEE, 2014 (see [55]).

A.1 Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models

The following paper was published as:

B. Hoisl and S. Sobernig. Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models. In *Proceedings of the International Workshop on Security Aspects of Process-aware Information Systems*, pages 673–679. IEEE, 2011 (see [50]).

2011 Sixth International Conference on Availability, Reliability and Security

Integrity and Confidentiality Annotations for Service Interfaces in SoaML Models

Bernhard Hoisl[1,2]
[1] *Secure Business Austria Research Center*
Vienna, Austria
bernhard.hoisl@sba-research.org

Stefan Sobernig[2]
[2] *Institute for Information Systems and New Media*
Vienna University of Economics and Business
stefan.sobernig@wu.ac.at

***Abstract*—This paper presents an approach for incorporating data integrity and data confidentiality into the model-driven development (MDD) of process-driven service-oriented architectures (SOAs) based on the OMG SoaML. Specifications for service interfaces are extended by UML activities to model invocation protocols. An invocation protocol makes the control and the object flows between service invocations explicit. Integrity and confidentiality attributes are used to annotate the object flows. The annotations serve for generating security-aware execution artefacts (e.g., interface description documents, deployment descriptors, and middleware configurations). We applied the approach prototypically in a Web Services platform environment (WS-BPEL, WSDL, WS-SecurityPolicy).**

***Keywords*-Service-Oriented Architecture, Security Engineering, UML, Web Services, SoaML, Model-Driven Development**

I. Introduction

A service-oriented architecture (SOA) is an architectural style which organises a software system as a composition of distributed software components, or services. Services require and provide callable functions at announced network endpoints. Processable interface descriptions serve for implementing the service interfaces in consumer and provider applications; independent from the communication middleware, the invocation protocols, and the transport protocols used. Modelling support for systems adhering to the SOA architectural style is provided, e.g., by the UML extension SoaML [1]. In a process-driven SOA, one or more components act as process engines and orchestrate service interactions in order to implement business processes [2].

Business processes executed by a process-driven system involve data assets (e.g., personnel or financial records) requiring protection against, e.g., unauthorized or inadvertent disclosure and modification. The need for enforcing security properties required by data assets (e.g., data confidentiality, data integrity) arises also from compliance requirements with legislation (e.g., privacy laws), with industry regulations (e.g., the Basel II Accord), and with security engineering frameworks (e.g., the SOA Security Compendium). Therefore, representing security properties explicitly in business process models based on EPCs [3], BPMN [4], and UML ([5]–[7]) have been proposed. In an earlier contribution [8], we introduced first-class support for expressing integrity and confidentiality properties of object flows in UML activities.

Nevertheless, business process models including security properties must be integrated with design and implementation models of the process-executing software system. For the scope of this paper, we look at the various invocation data processed for realising interactions among services and process engines: endpoint references, the operation names, as well as input and output parameters [9]. Process execution turns data assets into serialised invocation data exchanged between service endpoints. For example, if data assets in a business process require the integrity property, a modeller must express corresponding message integrity constraints over service interfaces. With this, integrity-verifying code (e.g., message interceptors for fingerprinting and signing) and/or configuration data (e.g., for a security component) can be generated. Such a multi-stage mapping helps maintain consistency with corresponding business process models and contributes to a compliant system implementation.

To facilitate the complex mapping task, model-driven development (MDD) approaches ([3]–[7]) have combined modelling support for security properties in business process models, model integration with structural and behavioural views of a process-driven SOA, and automated generation of execution models (and code). However, many existing MDD approaches fall short in two respects:

Common multi-view meta-model — The support for SOA, process, and security property modelling is not based on a meta-modelling environment providing several modelling views (e.g., a process flow view, a composition view, a data flow view) for reducing the overall mapping complexity [10]. Even if adopting a multi-view environment based on a common meta-model, such as the UML, standard extensions providing or refining these views are not adopted; e.g., the SoaML for the UML.

View accuracy — Security concerns are orthogonal to concerns such as service composition, transport handling etc. [9]. Modelling data integrity and data confidentiality properties should not lead to model interactions violating this separation of concerns. For instance, expressing security properties of invocation data at the level of process execution models (e.g., service activities [11]) bears the risk to interweave process flow and invocation handling details which are otherwise orthogonal to each other.

In this paper, we present an approach for modelling integrity and confidentiality properties of invocation data for standard SoaML models. For this, we constrain service interfaces through UML activities. These activities specify invocation data dependencies as object flows with integrity and confidentiality annotations [8]. By adopting the standard UML and the SoaML extension, existing tool support can be reused for generating execution models (WS-BPEL, WSDL). For the integration of the UML package `SecureObjectFlows` [8] with the SoaML, we provide both, a UML meta-model extension (`SecureObjectFlows::Services`) and a profile extension (`SOF::Services`), with bi-directional mappings available.[1]

The paper is structured as follows: In Section II, we give an overview of MDD for process-driven SOAs and the UML extensions used. The integration steps necessary for modelling secure flows of invocation data in SoaML are elaborated on in Sections III and IV. We proceed by contrasting our approach with closely related work on model-driven security in Section V. In Section VI, we conclude by summarising our contribution and by pointing to future work.

II. Overview

In relevant MDD approaches ([3]–[7]) computational independent models (CIMs) are considered first (e.g., informal process descriptions, structured architectural descriptions, and security engineering guidelines). In a next step, CIMs are formalised into platform independent models (PIMs), providing structural and behavioural views on the technical services and their process-driven composition. PIMs offer different, yet integrated views to capture a SOA, a process description, and security properties. From these different views, model transformations produce a set of platform-specific models (PSMs).

In our approach, we employ the UML for modelling various PIM views on a process-driven SOA. A SOA's structure (e.g., services, service interfaces) is depicted as a set of SoaML/UML models (see Section II-A). Data dependencies between invocations, including their integrity and confidentiality properties, are defined for a given service interface by secure object flows [8] (see Section II-B). Rule-based PIM-to-PSM translation is achieved by operating on the models' XMI representations. The XMI documents are processed into an intermediate object model, to bridge between the graph-based PIMs and the block-based PSMs [12]. Process-oriented transformation steps are supported by the existing Eclipse-based MDD4SOA plugin [13]. The targeted PSMs are interface descriptions (i.e., WSDL) and process execution descriptions (i.e., WS-BPEL 2.0).

[1]All modelling and implementation artefacts are available from http://nm.wu.ac.at/bhoisl.

For the transformation of the security property elements, we extended the MDD4SOA plugin. Additional transformation steps add WS-SecurityPolicy fragments to the generated interface descriptions and deployment descriptors. In addition, our approach allows for specifying parts of invocation data (e.g., single parameters or message elements) to be annotated. These parts turn into selection expressions over messages in WS-SecurityPolicy descriptions (e.g., `EncryptedElements`, `SignedParts`). The attributes of secure property elements specifying the integrity and confidentiality details map to identifiers for algorithm suites as defined by the WS-SecurityPolicy specification. Due to the space limitations, we do not elaborate further on PSM generation in this paper.

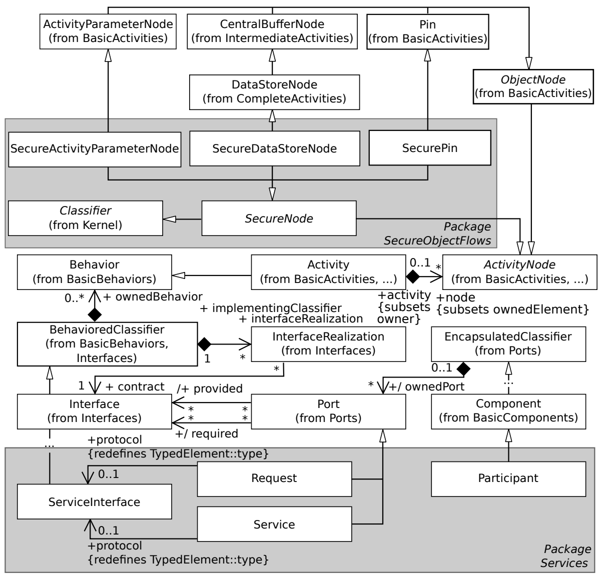

ActivityParameterNode (from BasicActivities)
CentralBufferNode (from IntermediateActivities)
Pin (from BasicActivities)
DataStoreNode (from CompleteActivities)
ObjectNode (from BasicActivities)
SecureActivityParameterNode
SecureDataStoreNode
SecurePin
Classifier (from Kernel)
SecureNode
Package SecureObjectFlows
Behavior (from BasicBehaviors)
Activity (from BasicActivities, ...)
0..1
*
ActivityNode (from BasicActivities, ...)
0..*
+ ownedBehavior
+ implementingClassifier
+ interfaceRealization
+activity {subsets owner}
+node {subsets ownedElement}
BehavioredClassifier (from BasicBehaviors, Interfaces)
InterfaceRealization (from Interfaces)
EncapsulatedClassifier (from Ports)
1
*
*
0..1
1
+ contract
/+ provided
*
+/ ownedPort
Interface (from Interfaces)
Port (from Ports)
Component (from BasicComponents)
+/ required
+protocol {redefines TypedElement::type}
0..1
Request
Participant
ServiceInterface
Service
0..1
+protocol {redefines TypedElement::type}
Package Services

Figure 1: Relevant excerpts from the SoaML and `Secure-ObjectFlows` meta-models

A. SoaML Concepts

The Service-oriented architecture Modeling Language (SoaML) extends the UML to model SOAs from structural and behavioural viewpoints (see Figure 1). As for the composition of service consumers and providers, the SoaML describes a SOA as a set of interacting components referred to as `Participants`, each announcing interaction capabilities and needs by means of `Service` and `Request` ports, respectively. For this, `Service` and `Request` ports expose required and provided `Interfaces` realising the port's protocol. This port protocol can be further specified by a `ServiceInterface`. `Participants` are connected via their protocol-compliant ports. Protocol compliance is expressed by either sharing a `ServiceInterface` between corresponding `Request` and `Service` ports; or by mating their required and provided `Interfaces` directly.

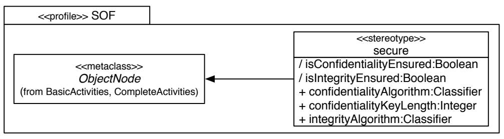
ServiceInterfaces allow the modeller to express behavioural details of port protocols explicitly. Protocol roles taken by two or more interacting ports for realising a ServiceInterface can be modelled, along with behavioural specifications such as UML activities. An excerpt of a SoaML model is depicted in Figure 3 for later reference.

B. *SecureObjectFlows Concepts*

The SecureObjectFlows meta-model extends UML activity models with abstract syntax and semantics for modelling data integrity and data confidentiality properties for object flows. Essentially, the SecureObjectFlows package specialises three types of ObjectNodes with integrity and confidentiality properties: SecurePin, SecureDataStoreNode, and SecureActivityParameterNode (see also Figure 1). These specialised nodes inherit all behaviour from their corresponding ObjectNodes and the SecureNode classifier. The abstract SecureNode meta-class allows for specifying additional properties (e.g., cryptographic hash functions, encryption algorithms, encryption key lengths) for its indirect instances and provides model integrity constraints.

III. A Profile for Secure Object Flows

In [8], the SecureObjectFlows package is introduced as a UML meta-model extension to the UML CompleteActivities package. For integration with the SoaML, it is necessary to provide a UML profile variant of the SecureObjectFlows meta-model. This is because, on the one hand, the SoaML is provided both as a meta-model and as a profile, with explicit correspondences defined for them. Both variants are binding compliance points for modellers, CASE tool providers, and extension engineers [1]. On the other hand, a UML profile extension provides immediate advantages. Most importantly, the CASE tool integration available for SoaML can be reused.



<<profile>> SOF
<<metaclass>>
ObjectNode
(from BasicActivities, CompleteActivities)
<<stereotype>>
secure
/ isConfidentialityEnsured:Boolean
/ isIntegrityEnsured:Boolean
+ confidentialityAlgorithm:Classifier
+ confidentialityKeyLength:Integer
+ integrityAlgorithm:Classifier

Figure 2: The «secure» stereotype

The profile variant of the SecureObjectFlows package provides a single stereotype «secure» extending the ObjectNode meta-class (see Figure 2). This stereotype provides all integrity and confidentiality attributes available for the SecureNode meta-class. The OCL constraints originally defined for the SecureObjectFlows meta-model were adapted for the context of the «secure» stereotype. The correspondences between the meta-model and the profile are depicted in Table I as instance specifications at the UML level M1.

M1 model (profile extension)	M1 model (meta-model extension)
:Pin —extension_secure→ :secure; :secure —base_ObjectNode→ :Pin	:SecurePin
:DataStoreNode —extension_secure→ :secure; :secure —base_ObjectNode→ :DataStoreNode	:SecureDataStoreNode
:ActivityParameterNode —extension_secure→ :secure; :secure —base_ObjectNode→ :ActivityParameterNode	:SecureActivityParameterNode

Table I: Mappings between the SOF profile and the SecureObjectFlows meta-model extension

IV. Secure Interfaces in SoaML

The SecureObjectFlows extension permits modelling the integrity and confidentiality properties as annotations for object nodes in UML activities. We aim at modelling invocation data (e.g., input and output parameters) requiring the integrity and the confidentiality property. In the compositional view of a SoaML model, service invocations are represented by ServiceInterfaces. In Figure 3, AService stipulates the permissible service invocations (e.g., OperationB1) between a Request port (requestor) and a Service port (provider). To describe the behavioural pattern of service invocations between two (or more) ports, a ServiceInterface as a kind of BehavioredClassifier can hold instances of Behavior and, hence, instances of Activities (see the UML excerpt in Figure 1 and AnInvocationProtocol in Figure 3).

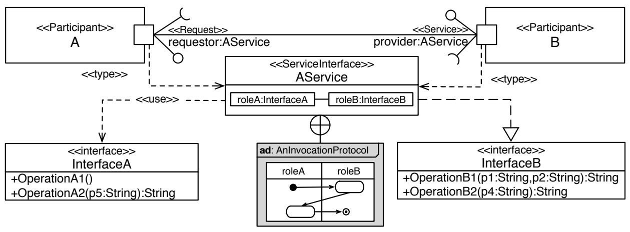


Figure 3: Activities on ServiceInterfaces

The availability of an Activity as owned behaviour of a ServiceInterface provides an important extension point[2] for defining security properties in invocation protocols:

- *Invocation data as object nodes*: An Activity, with its actions denoting invocations, allows for object flows to reflect input and output parameter streams for invocations. This is the major representational prerequisite for

[2]Note that the SoaML specification points to the usage of Activities for modelling *control flows* between operations which are required or provided by Interfaces defined on a ServiceInterface [1].

applying the concepts of the SecureObjectFlows package.

- *Protocol roles*: ActivityPartitions can be used for modelling *protocol roles*. Each ActivityPartition represents an interface-realising role, abstracting from the actual Participants using or implementing the interfaces. Their compositional correspondences are the parts defined for the activity-owning ServiceInterface (see roleA and roleB in Figure 3). ServiceInterfaces may refer to more than two parts (or protocol roles) and can so model multi-directional invocation flows.
- *Duality of invocations*: The roles are typed by the Interfaces required and implemented by the ServiceInterface (i.e., InterfaceA and InterfaceB in Figure 3). With ActivityPartitions denoting these roles, ActivityPartitions reflect the provider and consumer sides of invocations in a single model element. The integrity and confidentiality annotations in the Activity can so capture consumer- and provider-side capabilities (e.g., signature mechanisms).
- *Standalone invocation protocol*: A refined Activity owned by a ServiceInterface and the security properties specified for its object flows are modelled independently from the concrete Participants consuming or implementing the service endpoints. In Figure 3, for instance, AService and so AnInvocationProtocol apply to any pair of Participants, whether process engines or service providers.

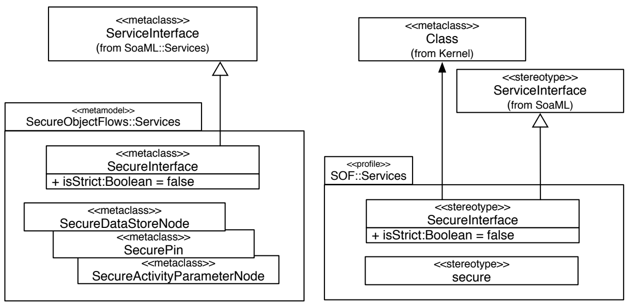


Figure 4: The UML packages for integration

To connect the SecureObjectFlows extension to ServiceInterfaces in SoaML, we provide two UML packages (see Figure 4). The SecureObjectFlows::Services package applies to the SecureObjectFlows meta-model extension. The SOF::Services package glues the SOF profile package and the SoaML profile. In the following, we outline their additions to the abstract syntax and to the semantics of secure object flows.

A. Abstract Syntax

As for the abstract syntax, we introduce a specialised ServiceInterface named SecureInterface. A SecureInterface adds to the ServiceInterface's capabilities by requiring exactly one Activity to be set as owned behaviour. At the SoaML meta-model level, SecureInterface extends the ServiceInterface meta-class. In the profile mapping, the SecureInterface meta-class is represented as a distinct stereotype specialising the ServiceInterface stereotype; and extending the Class meta-class (see Figure 4; see also OCL Constraint 1 in the Appendix).

A SecureInterface contracts either a strict or a permissive mode. In strict mode, all object flows qualifying as invocation data flows (as specified further below) must be annotated. The permissive mode, the default, does not impose a minimum number of secured object flows. The reader is referred to OCL Constraint 2 for the profile realisation of the strict/permissive mode.

B. Constraints

The following constraints impose integrity requirements upon the Activity owned by a SecureInterface. In addition, they add semantics to model flows of invocation data more accurately. For the meta-model integration, the OCL constraints are defined over the meta-classes SecureDataStoreNode, SecurePin, and SecureActivityParameterNode. As for the profile, they apply to the context of the «secure» stereotype. The OCL constraints for the SOF profile package are given as listings in the Appendix.

1) Traceability between Invocations and Interfaces: An Activity may only contain Actions representing Operations owned by the Interfaces implemented or used by a given ServiceInterface. We realise this by requiring all Actions to be instances of CallOperationAction. In addition, all CallOperationActions in an ActivityPartition must link to an Operation of the Interface represented by this ActivityPartition (see OCL Constraint 3).

2) Cross-Interface Invocations only: Depending on the partitioning of an Activity, object flows may occur *within* a single partition or *between* two partitions. In Figure 5, for instance, the object flow between OperationB1 and OperationB2 depicts an output/input dependency between operations owned by the same Interface. Such service invocations are traded within the same Service or Request port and do not travel between two service endpoints (see also Figure 3). We consider such service invocations bypassing most steps of invocation processing and being served within process or machine boundaries [9].

Therefore, we limit the applicability of secured object nodes to cross-interface invocations (see OCL Constraint

4). To distinguish between inner- and cross-interface object flows throughout an invocation protocol activity, all object nodes must be assigned to a single partition (see OCL Constraint 5). According to these constraints, the object flow between OperationB2 and OperationA2 in Figure 5 can be annotated, for instance.

3) Activity Parameters for Initial and Intermediary Inbound Data: An invocation protocol activity captures data dependencies between invocations, i.e., output data of one invocation serving as input data for a subsequent invocation. In two important cases, however, input data originates from the outside. These cases are *initial* and *intermediary* inbound data.

Initial inbound data is provided by the consumer triggering the execution of the Activity (see e.g. p1 in Figure 5). Intermediary inbound data is not the result of previous invocations within the same protocol. The input is rather provided from the outside, such as from a process engine holding process control data (see, e.g., p5).

For specifying secure object flows, however, it is mandatory to model pairs of secure object nodes [8]. This is because the security properties required at either end of an object flow might deviate from each other. Consider, for instance, two Participants providing different confidentiality algorithm suites. In order to model this heterogeneity while maintaining consistency between endpoints, an additional object node as an explicit counterpart must be modelled (see OCL Constraint 6). By using ActivityParameterNodes as counterparts, external input and output dependencies of the Activity are expressed.

4) Activity Parameters for Intermediary and Final Outbound Data: Analogous to initial and intermediary inbound data, output data can describe external data dependencies, i.e., dependencies which do not manifest within the invocation protocol. For instance, an invocation's output is to be stored as a process-persistent variable by a process engine. If secured, such object nodes require a corresponding object node, e.g., an ActivityParameterNode (see OCL Constraint 6).

5) Streaming-only Activity Parameters: As a result of the prior two constraint sets, the ActivityParameterNodes used to depict the counterparts of intermediary inbound data (e.g., AInParam2 in Figure 5) and output data must stand for *streaming* activity parameters (see OCL Constraint 7). Streaming parameters represent data which become available in the context of a given activity, or which leave this context, during execution of the Activity. Note that the streaming mode is only mandatory for cases of secure intermediary Input- and OutputPins (in the sense of OCL Constraint 6). Corresponding object nodes of Pins (e.g., AnInParam1) held by the first and the last CallOperationActions are exempted so that they can model global start and stop conditions for the invocation protocol activity.

6) Same Origin for Input Data Flows: Input data for service invocations, represented by InputPins on CallOperationActions, must have corresponding object nodes which all reside in the same ActivityPartition. By corresponding, we mean the initial source nodes of an object flow. Different partitions as origins for input data for an operation are invalid (see OCL Constraint 8). In Figure 5, for example, we find that the InputPins of OperationA2 have corresponding object nodes in the roleB partition.

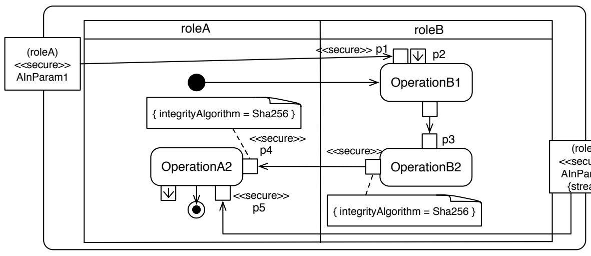


Figure 5: An invocation protocol activity with secure object flows

V. Related Work

In [3], Jensen and Feja extend a proprietary MDD software tool for modelling SOA security properties (access control, data integrity, and confidentiality). The target PSMs are WS-SecurityPolicy specifications. EPCs are used at the business process modelling level. The security model view, while sharing the EPC meta-model, is separate from the process model view. Both are maintained separately and mapped to each other to form an amalgam model. As for view accuracy, the security properties are only captured for the scope of a single process engine (rather than for a collaboration of service partners).

Wolter et al. present in [4] an approach for modelling security goals visually, including model transformations into corresponding security policy specifications. Security-annotated BPMN process models are mapped to WS-BPEL service descriptions and WS-Security policies. As the security models are specified in the UML, model transformations must be applied between the UML and the BPMN process description. A common meta-model for all views is, therefore, not realised. The security properties covered are access control (authentication and authorisation), confidentiality, and integrity at a per-message level.

Another approach is presented by Basin and Doser [5]. Basing the work on UML 1.4 class models, the authors integrate their security modelling language (SecureUML) with a custom defined process language. The SecureUML is extended to adopt an RBAC scheme, with RBAC constraints being expressed over process model instances. The PSM target is code for Java Servlet containers, instrumenting the container's access control mechanisms. The approach is limited in its extensibility because UML M1 class models define the shared meta-model. The state-transition semantics of the process models only cover a single view on a process-driven system. SOA-related views are not provided at all.

Nakamura et al. describe in [6] a toolkit for generating web services security configurations, covering properties such as authentication, integrity, non-repudiation, and confidentiality. UML class models provide a structural view on a SOA, with stereotypes representing selected security properties. Although using the UML meta-model has potential for adopting existing UML extensions, the authors limit themselves to a custom, ad hoc profile definition. Process views are not considered. The target PSMs are IBM WS-Security specifications.

In [7], Hafner et al. present a model-driven security approach for incorporating security requirements (integrity, confidentiality, and non-repudiation) into PIMs. The UML is used, on the one hand, to model process descriptions as activity diagrams and, on the other hand, to add security annotations to `ObjectNodes` using OCL-like statements. Two orthogonal views are presented: a workflow and an interface view. Although the PIMS are based on a common, i.e., the UML, meta-model, SOA-related extensions are not reused. Secure document flows between two participants can be expressed in the workflow view only, security properties of object flows between service invocations are not covered. From the PIMs, transformations generate BPEL, WSDL, and XACML artefacts.

VI. Concluding

In this paper, we outlined an approach for model-driven security of invocation data in process-driven SOAs. With an extension of SoaML service interfaces based on UML activities, we provide means to model integrity and confidentiality in invocation protocols. We emphasise the reuse of existing modelling extensions (SoaML, SecureObjectFlows), as well as existing MDD software artefacts (MDD4SOA). A UML profile (`SOF::Services`), formally described by a suite of OCL constraints, is available for adoption.

In contrast to likeminded MDD approaches, we provide separate views for security properties of invocation data, on the one hand, and process descriptions, on the other hand. As security requirements are orthogonal to, for instance, service orchestration requirements, these views should not interfere with each other. This separation of concerns is achieved for the UML as a common meta-model.

Future work will focus on integrating the invocation data view with the process flow and business process views. Candidates are *service activities* in UML4SOA [11] and the *process flow models* in [14]. Moreover, we plan to mature and document our adaptations of the Eclipse plugin MDD4SOA; and then place them into the public domain.

References

[1] Object Management Group, "Service oriented architecture Modeling Language (SoaML) – Specification for the UML Profile and Metamodel for Services (UPMS) – Version 1.0, Beta 2," Available at: http://www.omg.org/spec/SoaML/1.0/Beta2/PDF, 2009.

[2] C. Hentrich and U. Zdun, "A Pattern Language for Process Execution and Integration Design in Service-Oriented Architectures," *Transactions on Pattern Languages of Programming*, vol. 1, pp. 136–191, 2009.

[3] M. Jensen and S. Feja, "A Security Modeling Approach for Web-Service-based Business Processes," in *Proceedings of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 340–347.

[4] C. Wolter, M. Menzel, A. Schaad, P. Miseldine, and C. Meinel, "Model-Driven Business Process Security Requirement Specification," *Journal of Systems Architecture*, vol. 55, no. 4, pp. 211–223, 2009.

[5] 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, January 2006.

[6] Y. Nakamura, M. Tatsubori, T. Imamura, and K. Ono, "Model-Driven Security Based on a Web Services Security Architecture," in *Proceedings of the IEEE International Conference on Services Computing*. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 7–15.

[7] M. Hafner, R. Breu, B. Agreiter, and A. Nowak, "SECTET: An Extensible Framework for the Realization of Secure Inter-Organizational Workflows," *Journal of Internet Research*, vol. 16, no. 5, pp. 491–506, 2006.

[8] B. Hoisl and M. Strembeck, "Modeling Support for Confidentiality and Integrity of Object Flows in Activity Models," in *Proceedings of the 14th International Conference on Business Information Systems (BIS2011)*. Lecture Notes in Business Information Processing (LNBIP), Springer, forthcoming.

[9] S. Sobernig and U. Zdun, "Invocation Assembly Lines: Patterns of Invocation and Message Processing in Object Remoting Middleware," in *Proceedings of 14th Annual European Conference on Pattern Languages of Programming (EuroPLoP 2009)*, A. Kelly and M. Weiss, Eds. Irsee, Germany, July 8-12, 2009: CEUR-WS.org (Vol-566), March 2009.

[10] H. Tran, U. Zdun, and S. Dustdar, "VbTrace: Using View-based and Model-driven Development to Support Traceability in Process-driven SOAs," *Software & System Modeling*, vol. 10, no. 1, pp. 5–29, 2009.

[11] P. Mayer, N. Koch, A. Schröder, and A. Knapp, "The UML4SOA Profile," Available at: http://www.uml4soa.eu/wp-content/uploads/uml4soa.pdf, 2010.

[12] J. Mendling, K. B. Lassen, and U. Zdun, “On the Transformation of Control Flow between Block-Oriented and Graph-Oriented Process Modeling Languages,” *International Journal of Business Process Integration and Management*, vol. 3, no. 2, pp. 96–108, 2008.

[13] P. Mayer, A. Schroeder, and N. Koch, “MDD4SOA: Model-Driven Service Orchestration,” in *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*. IEEE Computer Society, 2008, pp. 203–212.

[14] U. Zdun, C. Hentrich, and S. Dustdar, “Modeling Process-Driven and Service-Oriented Architectures Using Patterns and Pattern Primitives,” *ACM Trans. Web*, vol. 1, no. 3, pp. 23–50, September 2007.

APPENDIX

CONSTRAINTS FOR THE SOF::SERVICES PROFILE

The following OCL expressions are specific to the Eclipse 3.6.2 MDT/OCL engine.

OCL Constraint 1: A `SecureInterface` must own an `Activity` instance as its owned behaviour.

```
context SOF::Services::SecureInterface
inv:    self.base_Class.ownedBehavior->one(oclIsKindOf(Activity))
```

OCL Constraint 2: In strict mode all cross-interface object flows must be secured.

```
context SOF::Services::SecureInterface
def:    allPredecessors(objNode : ActivityNode) : Set(ActivityNode) =
        objNode.incoming.source->collect(x |
          allPredecessors(x))->asSet()->union(objNode.incoming.source)
inv:    self.isStrict implies
          self.base_Class.ownedBehavior.oclAsType(Activity).node->select(
          oclIsKindOf(ObjectNode))->forAll(objNode |
            allPredecessors(objNode)->select(incoming->isEmpty())->forAll(s |
              s.inPartition <> objNode.inPartition implies
                s.getAppliedStereotype('SOF::Services::secure') <> null and
                objNode.getAppliedStereotype('SOF::Services::secure') <> null))
```

OCL Constraint 3: All `Actions` must be instances of `CallOperationAction` and each `CallOperation-Action`’s operation enclosed by a given partition must correspond to an `Operation` owned by the `Interface` denoted by this partition.

```
context SOF::Services::SecureInterface
inv:    self.ownedBehavior.oclAsType(Activity).node->
        select(oclIsKindOf(Action))->forAll(a |
          a.oclIsKindOf(CallOperationAction) and
          self.part->any(name = a.inPartition->any(true).name).type.
          oclAsType(Interface).ownedOperation->
          includes(a.oclAsType(CallOperationAction).operation))
```

OCL Constraint 4: Only corresponding object nodes residing in different partitions may be tagged by the «`secure`» stereotype.

```
context SOF::Services::secure
def:    allPredecessors(objNode : ActivityNode) : Set(ActivityNode) =
        objNode.incoming.source->collect(x | allPredecessors(x))->asSet()->
        union(objNode.incoming.source)
inv:    allPredecessors(self.base_ObjectNode)->select(
          incoming->isEmpty() and
          oclIsKindOf(ObjectNode) and
          getAppliedStereotype('SOF::Services::secure') <> null)->forAll(s |
            s.inPartition <> self.inPartition)
```

OCL Constraint 5: All activity nodes must be assigned to and must be contained by exactly one and only one activity partition.

```
context SOF::Services::SecureInterface
inv:    self.base_Class.ownedBehavior.oclAsType(Activity).node->forAll(
          inPartition->size() = 1)
```

OCL Constraint 6: All secured `InputPins` must have an incoming object flow; all secured `OutputPins` must have an outgoing object flow.

```
context SOF::Services::secure
inv:    self.base_ObjectNode.oclIsKindOf(InputPin) implies
          self.base_ObjectNode.incoming->notEmpty()
inv:    self.base_ObjectNode.oclIsKindOf(OutputPin) implies
          self.base_ObjectNode.outgoing->notEmpty()
```

OCL Constraint 7: All `ActivityParameterNodes` which are not initial or final nodes in a control and data flow but counterparts of intermediary `Input-` and `Out-putPins` must refer to a streaming `Parameter`.

```
context SOF::Services::SecureInterface
def:    isFirstNode(a : ActivityNode) : Boolean =
        a.owner.oclAsType(Activity).node->select(
          oclIsKindOf(InitialNode))->exists(outgoing.target->any(true) = a) or
        a.owner.oclAsType(Activity).node->select(
          oclIsKindOf(ActivityNode) and incoming->isEmpty())->includes(a)
def:    isLastNode(a : ActivityNode) : Boolean =
        a.owner.oclAsType(Activity).node->select(
          oclIsKindOf(ActivityFinalNode))->exists(
          incoming.source->any(true) = a) or
        a.owner.oclAsType(Activity).node->select(
          oclIsKindOf(ActivityNode) and outgoing->isEmpty())->includes(a)
def:    allSuccessors(objNode : ActivityNode) : Set(ActivityNode) =
        objNode.outgoing.target->collect(x |
          allSuccessors(x))->asSet()->union(objNode.outgoing.target)
inv:    self.base_Class.ownedBehavior.oclAsType(Activity).node->select(
        oclIsKindOf(ActivityNode))->forAll(an |
          (not isFirstNode(an) implies
            an.input->forAll(ipin |
              allPredecessors(ipin)->select(
              oclIsKindOf(ActivityParameterNode))->forAll(
                oclAsType(ActivityParameterNode).parameter.isStream))) and
          (not isLastNode(an) implies
            an.output->forAll(opin |
              allSuccessors(opin)->select(
              oclIsKindOf(ActivityParameterNode))->forAll(
                oclAsType(ActivityParameterNode).parameter.isStream))))
```

OCL Constraint 8: All source object nodes of a set of `InputPins` owned by a `CallOperationAction` must be assigned to the same activity partition.

```
context SOF::Services::secure
inv:    self.base_ObjectNode.oclIsKindOf(InputPin) implies
          self.base_ObjectNode.oclAsType(InputPin).owner.oclAsType(
          CallOperationAction).input->forAll(ipin |
            allPredecessors(ipin)->select(
              incoming->isEmpty() and
              oclIsKindOf(ObjectNode) and
              getAppliedStereotype('SOF::Services::secure') <> null)->forAll(
                inPartition = self.inPartition))
```

A.2 Towards a Systematic Integration of MOF/UML-based Domain-specific Modeling Languages

The following paper was published as:

B. Hoisl, M. Strembeck, and S. Sobernig. Towards a Systematic Integration of MOF/UML-based Domain-specific Modeling Languages. In *Proceedings of the 16th IASTED International Conference on Software Engineering and Applications*, pages 337–344. ACTA Press, 2012 (see [60]).

Proceedings of the IASTED International Conference
Software Engineering and Applications (SEA 2012)
November 12 - 14, 2012 Las Vegas, USA

TOWARDS A SYSTEMATIC INTEGRATION OF MOF/UML-BASED DOMAIN-SPECIFIC MODELING LANGUAGES

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

ABSTRACT
In model-driven development (MDD), UML-based domain-specific modeling languages (DSMLs) are frequently used for specifying software systems. The integration of corresponding DSMLs is an important part of model-driven software evolution and maintenance. However, due to a wide variety of DSML design options, integrating DSMLs is a non-trivial task. In this paper, we discuss issues that may arise when integrating MOF/UML-based DSMLs and present a process model for the systematic integration of DSMLs to address some of these issues. In particular, we discuss different composition techniques as well as challenges that may occur in the different phases of DSML integration. In addition, we provide an example for the integration of two DSMLs from the security domain. With our process model we aim to provide a conceptual framework for the systematic integration of MOF/UML-based DSMLs.

KEY WORDS
Model-driven development, domain-specific modeling language, language composition, integration model, UML

1 Introduction

In recent years, model-driven software development (MDD) emerged as a software engineering technique for the specification of tailored domain-specific software systems (see, e.g., [1, 2]). The modeling of complex domain artifacts helps to understand these problems and potential solutions through abstraction [3]. In this sense, MDD raises the level of abstraction in the software engineering process—as high-level programming languages have done in the past [4]. Thereby, MDD helps to enhance the understanding of a problem and solution domain and benefits from a high degree of automation (e.g., tool-supported code generation) [3].

In the context of MDD, domain-specific (modeling) languages (DSLs/DSMLs) are special-purpose (modeling) languages tailored for a particular domain (see, e.g., [5, 6, 7]). The development of DSMLs based on the Meta Object Facility (MOF, [8]) and/or the Unified Modeling Language (UML, [9]) are commonly applied in MDD, for instance, for the specification of security-related properties (see, e.g., [10, 11]). A MOF/UML-based DSML is characterized by utilizing the MOF/UML specifications where possible and by extending their definitions where necessary. Thereby, DSMLs that are based on the MOF/UML can directly benefit from maintenance through the Object Management Group (OMG), standardized modeling extensions, and a variety of corresponding software tools.

Software systems are frequently subject to changing requirements and evolve over time [12]. Thus, the composition of DSMLs becomes an integral part of model-driven software evolution and maintenance. DSML composition refers to techniques to combine two or more DSMLs which were not intended for integration at design time of each DSML. The integration process does not change the initial DSMLs, but provides techniques to transform and to compose the different artifacts for the creation of a new DSML. Therefore, reuse is facilitated in a way that all DSMLs can be used in parallel. However, DSML integration is a non-trivial task due to the variety of design options (even if we focus on MOF/UML-based DSMLs) and a number of composition issues (e.g., composition order).

The many facets of DSL/DSML development (such as, development guidelines and patterns, development processes, design decisions) are discussed by a number of recent publications (see, e.g., [5, 7, 13, 14]. Fewer publications contribute to evolutionary aspects of DSL development, such as, model versioning (see, e.g., [15]), composition of metamodels (see, e.g., [16]), composition of DSLs (see, e.g., [17, 18]), or composition of transformation rules (see, e.g., [19, 20]). Although these publications discuss selected DSML integration techniques, (1) they do not take the overall DSML development process into account and (2) they do not target on MOF/UML-based DSML composition in particular.

In order to integrate DSMLs, the composition process must ensure the correct integration of all aspects a DSML consists of (e.g., language model, behavior, concrete syntax). In this paper, we adopt the language model-driven process model from [14] to structure the integration of MOF/UML-based DSMLs. We discuss composition decisions and techniques, inputs and outputs, as well as challenges that may occur in each DSML integration phase. We provide a process model for DSML composition which can be used as a template when integrating MOF/UML-based DSMLs. An example shows the integration of two security-related DSMLs.

The remainder of this paper is structured as follows. Section 2 discusses issues at the various stages of DSML integration. Section 3 presents a process model for the systematic integration of DSMLs. An example DSML composition is sketched in Section 4. Related work is reviewed in Section 5 and Section 6 concludes the paper.

2 DSML Integration Issues

In the process of DSML integration a couple of issues arise. In this section, we briefly review important challenges of integrating MOF/UML-based DSMLs which we extracted from the literature (see, e.g., [7, 13, 14, 16, 17, 20, 21, 22]).

Consolidated domain space. If the DSMLs do not only need to reference each other, but aim at a tighter integration (e.g., one DSML refining the other), their concepts must be aligned. It must be assured that, for instance, equally named metaclasses are representing the same domain concepts. Thus, a transformation into a consolidated solution domain space is essential (e.g., via a composition of MOF-compliant metamodels).

Compatible formalization. The MOF/UML-based language models of DSMLs can be formalized using different modeling techniques (see, e.g., [13, 21]). A common formalization style is a prerequisite for a consistent composition (e.g., if both language models are described as UML extensions or as profiles). This can be achieved, for instance, with automatic model transformations (see, e.g., [23, 24]); e.g., via transformation languages, such as, the Atlas Transformation Language (ATL [25]) or the Epsilon Transformation Language (ETL [26]). In this context, rule-based transformation templates provide implicit trace links to the original metamodels (which can be used, for example, to adjust platform integration templates).

Constraint adaptation. According to the integration strategy and to the composition purpose, the constraint sets defined over the DSML models must be adapted. For instance, refinements of a metaclass can be further restricted using explicit constraint expressions; e.g., via Object Constraint Language (OCL [27]) or Epsilon Validation Language (EVL [26]) expressions.

Composition workflow. The composition process for the elements of the language models can include several composition techniques and a number of interdependent composition tasks (e.g., selecting the elements, choosing the composition operation, and adapting the constraints). It is essential to apply suitable means to define an executable composition workflow; e.g., via build files, such as, Apache Ant scripts for the Epsilon Merge Language (EML [26]).

Packaging. Composition-specific constructs (e.g., merge and import references, new metaclasses, OCL constraints) should be defined in a way which preserves the modeling artifacts of the integrated DSMLs. With this, the DSMLs remain usable both in their uncomposed and in their composed forms. The UML provides constructs for grouping and for qualifying composition-only model elements (i.e., packages, profiles, the containment relationship).

Symbol composition. The integrated DSMLs might come with symbol additions to the UML symbol set [21]. Under composition, the symbol sets must be integrated. If two metaclasses were merged into a single one, one of the diagram symbols would have to be dropped. This symbol composition is non-trivial. The combined symbol set must be consistent with the original ones (e.g., unchanged concepts continue to be represented by one icon) and the resulting symbol set must not suffer from cognition-critical deficiencies (e.g., synographs [22]). This challenge is amplified because the UML specification does not provide standardized means for extending the base UML symbol set.

Composition order. The order in which the source DSML models enter the composition operation is of utmost importance. This order must avoid any contradictory composition results in terms of the functional properties of both integrated DSMLs. Constraints on the composition ordering must also be addressed in the behavioral formalization of the composed DSML (as represented by, e.g., UML M1 behavioral models, such as, state machines or sequence diagrams). The composition ordering must also be enforced at the platform integration stage (e.g., by instrumenting an appropriate language-level composition technique accordingly).

Host platform. The target platform of the composed DSML is crucial. Both DSMLs are integrated either into one of the two already targeted platforms or into a new, third platform specific to the composed DSML. Alternatively, pipelining [6] can be used to operate between different platforms.

Generator adaptation. In MDD, model-to-text (M2T) transformations are commonly applied using generator templates (see, e.g., [28]). A composition of two DSMLs requires the adaption of these templates. Depending on the technology, this can be achieved in various ways, for instance, with aspect orientation, higher-order templates, or automatic evaluation of trace links (see, e.g., [19]).

Modeling tool support. If two DSMLs serve modeling purposes only, it is needed to integrate or provide a new tool which supports the composed DSML; e.g., by creating a graphical editor for the composed DSML based on the Eclipse Graphical Editing Framework (GEF).

Composition times. The stages of DSML composition are performed at different times. Examples are the generation time of an intermediate model (if an indirect model transformation is applied), the direct M2T transformation time (irrespective of the transformation technique used; e.g., API-based), and the runtime (e.g., by using pipelining between DSML-derived programs).

3 DSML Integration Process

In this section, we present a process model for the integration of MOF/UML-based DSMLs (see Figure 1). The process model identifies the four core phases of DSML composition. In the language model-driven engineering process, "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" [14] (see Figure 1). Below, we discuss how the issues documented in Section 2 are addressed in our integration process.

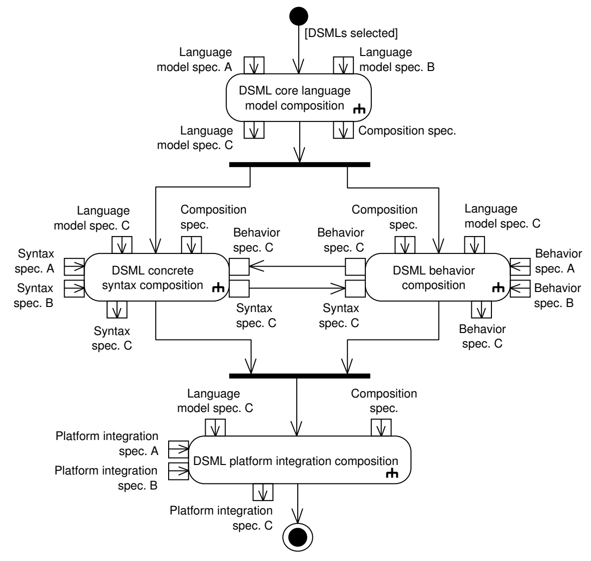


Figure 1. DSML integration process (based on [14]).

3.1 Language Model and Constraints

For UML-based DSMLs, the language model is specified via a MOF-compliant metamodel and, if needed, via accompanying invariant constraints. In the context of DSML integration, both language model specifications act as the inputs for the core language model composition phase (see Figure 1). Composing the core language model and its constraints is divided into several sub-activities, as indicated with the rake in Figure 1. These include selecting the elements from the core language models to be integrated, choosing a composition method, defining the composition workflow etc. Outputs of this phase are, on the one hand, composed language model specifications (i.e., the composed core language model and its constraints), and, on the other hand, composition specifications. The form of the composed language model specifications relies on the applied composition technique. The composition specifications can, for instance, have a rule-based format (e.g., model-to-model (M2M) transformation rules) or composition traces can be stored in a separate model (e.g., as a weaving model). The subsequent integration phases depend on all outputs from this first phase.

There are several techniques for integrating elements from two different metamodels which can be used exclusively or in combination. Elements from both metamodels can be *merged* into a third (existing) element or into a new one (created as an output element of the composed language model specification). This technique is favorable if both metamodels overlap and partly provide the same functionality. A *refinement* of a metaclass (to be preserved) using another metaclass is implemented as a specialization. That is, DSML 1 refines the functionality from DSML 2, for instance, with platform-specific methods (e.g., modeled as a generalization relationship). DSML 1 can also *extend* features from DSML 2. Thereby, DSML 1 provides new functionality that does not exist in DSML 2. Another composition method extends DSML 1 by *referencing* features from DSML 2 (e.g., via new associations or via a separate weaving model). Functionality from DSML 1 can also act as an *alternative* to features from DSML 2, with the modeler having to choose between one of the two, finally. Furthermore, 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.

3.2 Concrete Syntax

The concrete syntax of the DSML acts as the interface presented to the user. Therefore, the symbols must reflect their underlying concepts (from the core language model; see Section 3.1) as clearly as possible. Inputs to the phase of the DSML concrete syntax composition are the individual syntax specifications coming from the two DSMLs, the composed language model, and the composition specifications resulting from the core language model composition phase (see Figure 1). The composed DSML's syntax is developed in parallel with the DSML behavior specification (see Section 3.3). This is because details of the behavior specification can be reflected in the concrete syntax (e.g., states of a language model element become cues in the symbol design). Output of this phase is a composed syntax specification, comprising symbolic elements (in any form, e.g., diagrammatic or textual) and their mapping to elements of the core language model.

For composing different graphical elements there are basically two options: *syntax extension* and *syntax integration*[1]. The graphical syntax of DSML 1 can be extended by elements of DSML 2 (e.g., when language model elements were composed using reference or extend techniques). Elements from DSML 1 can also be fully integrated into DSML 2. This means that new graphical elements are created which combine the syntactical styles of both DSMLs (e.g., when language model elements were composed using merge or refine techniques).

3.3 Behavior Specification

The behavior specification of the composed DSML must conform to the integration purpose and is critical for defining a composition order. The composition order dictates the enforcement of properties provided by each DSML and so contributes to a sound composition by, e.g., respecting functional dependencies between the concerns covered by the DSMLs. The composed language model and composition specifications as well as the two behavior specifications from the integrated DSMLs are input to the phase of behavior composition (see Figure 1). The phase of composing the DSMLs' behaviors is performed in parallel with the composition of the concrete syntaxes (see Section 3.2). Output is the behavior specification of the composed DSML.

Behavior definitions can be created, for instance, informally using *text artifacts*, *formal or informal (control-flow) models* (e.g., petri nets or UML state machines), *examples* (e.g., usage or model examples), and *executable code* specifications (e.g., algorithms). Depending on the specification of the composed language model, the behavior definition represents a *refinement*, an *extension*, a *restriction*, or an *execution order* on the integrated DSMLs.

[1]For this paper, we do not elaborate on further concrete syntax options, such as, non-diagrammatic, tree-based, tabular, or hybrid forms (see, e.g., [21]).

3.4 Platform Integration

Platform integration is not only determined by the integration purpose, but also by the feasibility and by the effort needed to compose the DSMLs at the system level. Again, the language model and composition specifications as well as the two platform integration specifications serve as inputs to this composition phase. Output is a specification for the composed platform integration of the DSMLs (see Figure 1). This specification can take the form of mere textual integration descriptions, M2T transformation specifications, or code artifacts in the host language.

The composition techniques are loosely dependent on the integration strategy applied in the phase of composing the core language model. Approaches for the platform implementation range from pipelining, piggybacking, language extension, to front-end integration. A *pipeline* takes the output from one DSML-derived program and feeds it into the second DSML-derived program for further processing without structurally integrating both code bases. The *piggyback* approach reuses the capabilities of two DSML-derived programs for building a new DSML-derived program. Using an *extension*, the DSML-derived programs are extended by means of the host language (e.g., class hierarchy). *Front-end integration* provides a common interface and facade for both DSMLs, with instruction calls being forwarded to the freestanding DSML-derived programs (see, e.g., [6, 7]).

4 Example DSML Integration

In this section, we apply the process model presented in the former Section 3 for integrating two security-related DSMLs (see [11, 29]). The first DSML [29] models system audits (referred to as *DSML A*, hereafter). Therein, a domain-specific UML extension is defined for the specification of audit events, audit rules, and notifications that are triggered via audit events. With this generic extension, audit requirements can be modeled from multiple views. The second DSML project (*DSML B*, [11]) presents an approach for the specification and the enforcement of secure object flows in process-driven service-oriented architectures (SOAs). In this context, a secure object flow (SOF) ensures the confidentiality and the integrity of important objects (e.g., business contracts) that are passed between different participants in SOA-based business processes. While DSML B provides means for message security in SOAs, DSML A supports accountability of event data via audit trails.

Audit logging in distributed environments, such as in Web-Service-based infrastructures, is a challenge and is not sufficiently supported by current approaches (only via limited specifications of context-insensitive log levels of runtime engines). Moreover, many security standards for Web Services exist (e.g., WS-Security, WS-Trust, SAML), but they all lack extensions to audit logging [30]. Thus, integrating message-level security and event-based audit log facilities at the model and at the application level presents a benefit.

Table 1 summarizes the techniques applied for both DSMLs at each development phase. At the stage of defining the *DSML core language model*, both DSMLs provide new packages at the level of the UML metamodel. For DSML A, the package consists of both, a UML stereotype specialization (contained in a UML profile) and MOF-based extensions. Besides the UML metamodel extension, DSML B provides also a complete mapping to a UML profile. Both DSMLs provide new diagrammatic elements (i.e., novel graphical notations) and UML stereotypes as their *concrete syntaxes*. The stereotype definition of DSML A is complementing the newly defined graphical elements; a textual notation is provided as an alternative visualization option. In contrast, the UML stereotypes of DSML B can be used as a replacement of the novel diagrammatic elements. The *behavior specifications* of both DSMLs are provided as textual descriptions with accompanying example models. For the *DSML platform integration*, DSML A provides direct M2T transformations into Java code using Epsilon Generator Language (EGL [26]) templates. In contrast, API-based generators are defined in DSML B. A first transformation generates an intermediate object model, which is transformed into BPEL, WSDL, and WS-SecurityPolicy documents in a second step.

Development phase	DSML A [29]	DSML B [11]
DSML core language model and constraints	UML metamodel extension & UML profile, OCL constraints	UML metamodel extension, UML profile, OCL constraints
DSML concrete syntax	New diagrammatic elements & UML stereotype, textual notation	New diagrammatic elements, UML stereotypes
DSML behavior	Textual descriptions, example models	Textual descriptions, example models
DSML platform integration	EGL generator templates; direct transformation; Java code	API-based generator; indirect transformation; BPEL, WSDL, WS-SecurityPolicy specifications

Table 1. Applied techniques at each DSML development phase.

4.1 Language Model and Constraints

The core language model for both DSMLs are defined at the level of the UML metamodel (*consolidated domain space*[2]; see Table 1). Additionally, for DSML B, mappings to a UML profile exist. This is to comply with the SoaML specification and to facilitate tool support (see [11, 31]). Thus, conceptual composition is performed via a metamodel-based integration; UML/SoaML compliance can be achieved at the level of a profile integration (*compatible formalization*). As there exists a consolidated domain space (MOF-constructs), we provide mappings for all DSML A elements to UML stereotypes (see [11] and Figure 2). The composition is done via a dedicated integration profile named `SOF::Services+SecurityAudit` which merges the corresponding profiles from both DSMLs (*packaging*, see Figure 2). In addition to the integration of the DSML language models, the profile merge also implies the application of corresponding language model constraint specifications (as defined in [11, 29]). Moreover, the integration profile provides the following OCL constraint as a composition refinement (*constraint adaptation*): *Every «`secure`» stereotyped `ObjectNode` must also be tagged as an «`AuditEventSource`».*

```
context SOF::Services+SecurityAudit::secure inv:
  self.base_ObjectNode.getAppliedStereotype('SOF::
       Services+SecurityAudit::AuditEventSource') <>
       null
```

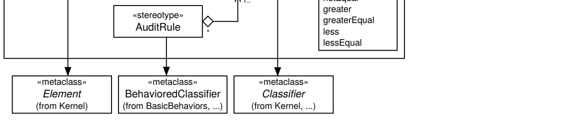


Figure 2. Example language model composition via UML profile merges.

4.2 Concrete Syntax

The concrete syntaxes are provided as UML stereotypes with accompanying icons for the defined profiles (Figure 2). The concrete syntax specifications for the `SOF` and `SOF::Services` profiles (i.e., DSML B) are defined in [11]. Figure 3 specifies the corresponding icons for the stereotypes of the `SecurityAudit` profile (i.e., DSML A; *symbol composition*). Icons can be used as full replacements of stereotyped elements (see [9]). A

[2]Italic phrases indicate the DSML integration issues discussed in Section 2.

sample application of a stereotyped classifier is shown in Figure 3. On the left hand side, textual stereotypes are written inside guillemets, on the right hand side, the same stereotypes are applied using corresponding symbols. As several stereotypes (and icons) can be applied to an element (see [9]), we do not need to define extra graphical specifications for the integration profile `SOF::Services+SecurityAudit`.

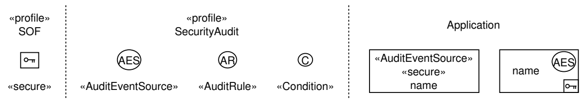


Figure 3. Stereotype and icon definitions for the concrete syntax.

4.3 Behavior Specification

For the integration profile (see Section 4.1), we specify a UML state machine to define the composed behavior and the behavioral *composition order* of the integrated security-related DSML aspects (Figure 4). For each individual DSML, OCL constraints formally specify their semantics (see [11, 29]). The integration profile `SOF::Services+SecurityAudit` merges these constraints as well as the language models (see Section 4.1). We define that for the `SOF::Services+SecurityAudit` profile, first, the constraints from the composed profiles are enforced; i.e., constraints from the «`SOF`», «`SOF::Services`», and «`SecurityAudit`» profiles—in this order (see Figure 4). Then, the incoming/outgoing secure object flows are processed, and, last, the audit is performed (details are shown in Figure 4).

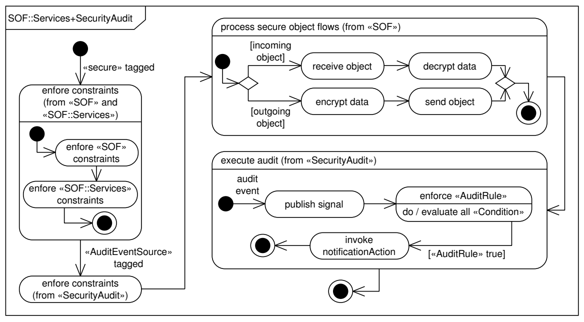

SOF::Services+SecurityAudit
«secure» tagged
enfore constraints (from «SOF» and «SOF::Services»)
enfore «SOF» constraints
enfore «SOF::Services» constraints
«AuditEventSource» tagged
enfore constraints (from «SecurityAudit»)
process secure object flows (from «SOF»)
[incoming object]
receive object
decrypt data
encrypt data
send object
[outgoing object]
execute audit (from «SecurityAudit»)
audit event
publish signal
enforce «AuditRule»
do / evaluate all «Condition»
invoke notificationAction
[«AuditRule» true]

Figure 4. Integrated DSML behavior specification as a UML state machine.

4.4 Platform Integration

As can be seen in Figure 5, we use a pipeline approach by calling audit features of a program derived from DSML A within a program generated by DSML B (*host platform*). Auditing is done via the facilities of DSML A and respective calling routines are added to DSML B. Therefore, we adapted the API-based generator of DSML B to issue a Web Service call every time a secure object flow is instantiated (see Figure 5[3]). This meant changing the generator to include audit-based service endpoints in the WSDL as well as the pipeline audit logic in the BPEL process (*generator adaptation*). Another approach would have been, for instance, using tools such as Java2WSDL to migrate the output of the M2T transformation from DSML A to be Web Service compliant and to integrate it into DSML B specifications (different *composition times*). Native UML *modeling tool support* is provided for all aspects of the composed DSML.

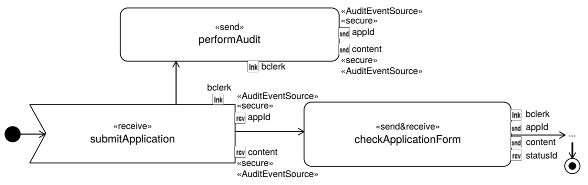


Figure 5. Example of a pipeline call to the audit DSML A within the DSML B workflow.

5 Related Work

In [14], we present an approach for developing DSLs systematically. In particular, we propose a process model for DSL development. The different phases can be tailored to the respective DSL engineering project. This process model forms the basis of the flow of DSML integration tasks presented in this paper. The reusable architectural decisions on designing DSLs in [7] influenced the discussion of the integration issues as well as the process model for DSML integration. However, the reflection on DSL design decisions in [7] relates primarily to programmatic DSLs (and so, e.g., to concrete textual syntaxes) and we extend their reach to MOF/UML-based embedded DSMLs (e.g., their concrete graphical syntaxes).

The composition of DSML models (i.e., of metamodels) has been frequently addressed. Emerson and Sztipanovits [16] review current metamodel composition methods (e.g., merge, refinement, interfacing). This research strand is limited to the phase of integrating the language model, the remaining phases (e.g., DSML behavior specification or platform integration) and their interdependence are not covered.

Another group of researchers (see, e.g., [20]) investigates composing model transformation

[3]The other stereotypes are coming from the SoaML and UML4SOA profiles and are not of interest in this context (for details see [11]).

templates—an important instrument for DSML integration at the platform integration stage. Tisi et al. [19], to name just one, present transformation templates as first-class models themselves. These models can then be used as inputs for templates defined using the same transformation language (i.e., higher-order model transformation). While mostly applied to M2M transformations (i.e., at the level of the DSML language model), higher-order transformations can be adopted for M2T, as well.

Reuse strategies for DSLs/DSMLs are discussed by a number of contributions (for instance, through software product-line techniques; see, e.g., [17, 18]). Taentzer et al. [15] present an approach for model versioning based on graph transformations. Although composition techniques are the key factor, each of these contributions either discusses a specific DSL/DSML development phase (i.e., the core language model) or system-level, textual DSLs only.

6 Conclusion

In this paper, we discussed composition aspects for MOF/UML-based DSMLs in the context of MDD. We reflected on the issues arising during DSML composition and presented a process model for the systematic integration of DSMLs. Moreover, we discussed an integration of two security-related DSMLs to provide a practical example.

Our approach explicitly focuses on MOF/UML-based DSMLs, although the overall integration process and its composition aspects may be applicable to other DSML formats, as well. For instance, the core composition phases with their inputs and outputs (see Section 3) are most likely the same for every DSML and are not MOF/UML-specific. Furthermore, several composition decisions and techniques as well as integration challenges can be applied to non-MOF/UML-based DSMLs or—when not directly applicable—can be transferred to other modeling languages.

Both DSMLs used for the composition examples were developed by the authors of this paper. Although they were not built for integration, a methodical and technological bias may exist. This bias may also had an influence on the critical discussion of the composition aspects. Due to the page limitations, we were restricted to one integration example for every phase of the process model. In our future work, we will consider more composition options.

This contribution is a first step towards the definition of a systematic DSML integration approach. Still, there is need for abstracting and expanding the findings presented in this paper. Hence, as an outlook, we will extend our work, collect more evidence, and provide for an evaluation.

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] S. Mellor, A. Clark, and T. Futagami, “Model-driven Development,” *IEEE Software*, vol. 20, pp. 14–18, Sept. 2003.

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

[4] D. Schmidt, “Model-Driven Engineering – Guest Editor’s Introduction,” *IEEE Computer*, vol. 39, pp. 25–31, Feb. 2006.

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

[6] D. Spinellis, “Notable Design Patterns for Domain-specific Languages,” *Journal of Systems and Software*, vol. 56, no. 1, pp. 91–99, 2001.

[7] U. Zdun and M. Strembeck, “Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Development,” in *Proceedings of the 14th European Conference on Pattern Languages of Programs (EuroPLoP)*, pp. 1–36, 2009.

[8] 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.

[9] 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.

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

[11] 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*, accepted for publication, 2012, DOI: 10.1007/s10270-012-0263-y.

[12] M. Godfrey and D. German, "The Past, Present, and Future of Software Evolution," in *Proceedings of Frontiers of Software Maintenance (FoSM)*, pp. 129–138, 2008.

[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 *Proceedings of the 2nd Workshop on Process-based approaches for Model-Driven Engineering (PMDE)*, pp. 303–314, 2012.

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

[15] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer, "A Fundamental Approach to Model Versioning based on Graph Modifications: From Theory to Implementation," *Software and Systems Modeling*, pp. 1–34, Apr. 2012.

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

[17] W. Cazzola and D. Poletti, "DSL Evolution through Composition," in *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, pp. 6:1–6:6, ACM, 2010.

[18] J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale, and D. C. Schmidt, "Improving Domain-specific Language Reuse with Software Product Line Techniques," *IEEE Software*, vol. 26, pp. 47–53, July 2009.

[19] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the Use of Higher-Order Model Transformations," in *Proceedings of the 5th European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, pp. 18–33, Springer, 2009.

[20] D. Wagelaar, "Composition Techniques for Rule-Based Model Transformation Languages," in *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT)*, pp. 152–167, Springer, 2008.

[21] 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.

[22] D. Moody and J. van Hillegersberg, "Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams," in *Proceedings of the 1st International Conference on Software Language Engineering (SLE)*, pp. 16–34, Springer, 2009.

[23] T. Mens and P. v. Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.

[24] 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.

[25] F. Jouault and I. Kurtev, "On the Architectural Alignment of ATL and QVT," in *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, pp. 1188–1195, ACM, 2006.

[26] D. Kolovos, L. Rose, R. Paige, and A. García-Domínguez, "The Epsilon Book." Available at: `http://www.eclipse.org/epsilon/doc/book/`, 2012.

[27] 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.

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

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

[30] A. Chuvakin and G. Peterson, "Logging in the Age of Web Services," *IEEE Security and Privacy*, vol. 7, pp. 82–85, May 2009.

[31] Object Management Group, "Service oriented architecture Modeling Language (SoaML) Specification." Available at: `http://www.omg.org/spec/SoaML`, May 2012. Version 1.0.1, formal/2012-05-10.

A.3 Higher-Order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages

The following paper was published as:

B. Hoisl, S. Sobernig, and M. Strembeck. Higher-Order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages. In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pages 49–61. SciTePress, 2013 (see [53]).

Higher-order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages

Bernhard Hoisl[1,2], Stefan Sobernig[1], and Mark Strembeck[1,2]
[1]*Institute for Information Systems and New Media, WU Vienna, Vienna, Austria*
[2]*Secure Business Austria Research (SBA Research), Vienna, Austria*

Keywords: Domain-specific Modeling Language, M2T, Code Generation, Higher-order Transformation, Eclipse Modeling Framework, Epsilon.

Abstract: Domain-specific modeling languages (DSMLs) are commonly used in model-driven development projects. In this context, model-to-text (M2T) transformation templates generate source code from DSML models. When integrating two (or more) DSMLs, the reuse of such templates for the composed DSML would yield a number of benefits, such as, a reduced testing and maintenance effort. However, in order to reuse the original templates for an integrated DSML, potential syntactical mismatches between the templates and the integrated metamodel must be solved. This paper proposes a technology-independent approach to template rewriting based on higher-order model transformations to address such mismatches in an automated manner. By considering M2T generator templates as first-class models and by reusing transformation traces, our approach enables syntactical template rewriting. To demonstrate the feasibility of this rewriting technique, we built a prototype for Eclipse EMF and Epsilon.

1 INTRODUCTION

A domain-specific modeling language (DSML) provides modeling abstractions and notations to describe the concepts and activities in a business domain (e.g., health care or banking) or a technical domain (e.g., access control or workflow specification). DSMLs commonly focus on narrow domain fragments and system concerns only, such as schedule management for power suppliers or security properties of business process data (Spinellis, 2001).

The benefits of DSMLs include reduced development times for DSML-based software products, an improved time-to-market, as well as reductions in development and delivery costs; e.g., for developer or customer trainings (Bettin, 2002). However, the development of a DSML and corresponding tool support most often requires substantial efforts that add to the overall costs of the underlying software development project (White et al., 2009; Krueger, 1992). Thus, benefits of a domain-specific development approach only realize over time.

As a result, the costs of DSML development are strong drivers for reusing DSMLs as design artifacts, both during the life cycle of a single software product and for multiple software products (White et al., 2009; Krueger, 1992). For a single software product, the development of a tailored DSML can be justified if the underlying software product is subject to frequent modifications or if the respective project demands for multiple and rapidly available prototypes. While a DSML would generate significantly more benefits if it was used in the development of different software products, this reuse is often barred by the narrow domains covered by DSMLs. In this situation, one option is to start from a joint metamodel and to refine this metamodel, the corresponding structural and behavioral semantics, as well as the DSML notation to cover an extended domain.

To develop a software product using two or more pre-existing DSMLs, with each DSML defining a subsystem of the product, integrating the corresponding DSMLs into a new consolidated DSML is an important design option (Vallecillo, 2010). Consider, for example, modeling the billing domain in a power supply company which covers company-specific accounting and branch-specific schedule management. Provided that compatible DSMLs for both tasks (i.e., accounting and schedule management) are available (e.g., based on the same metamodeling infrastructure), their integration is a viable strategy; e.g., via product line techniques (White et al., 2009). Simi-

Hoisl B., Sobernig S. and Strembeck M. (2013).
Higher-order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages.
In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pages 49-61
DOI: 10.5220/0004321100490061
Copyright © SCITEPRESS

larly, in security-critical domains integrating security-related DSMLs can support the systematic composition of different security concerns (Hoisl et al., 2012).

In addition to reusing the domain-specific language models through metamodel composition (Kalnina et al., 2010; Object Management Group, 2008), the model-to-text (M2T) transformations available for the source DSMLs could be applied to models of the new DSML. This way, generator artifacts (such as platform code, configuration specifications, and deployment descriptors) could be reused for the new DSML.

From a DSML integration perspective, however, there are major barriers to reusing model transformations (Wimmer et al., 2012). One barrier for M2T generator templates, in particular, is the conformance relation between M2T transformations and a given metamodel. Consider the template sketch in Figure 1. The transformation template which accompanies class `B` contains a variable assignment expression, with the right-hand side calling on `propertyB` of an instance of `B` (assuming that `y` stores an instance of `B`). This way, the template is confined to a metamodel containing a corresponding class `B`.

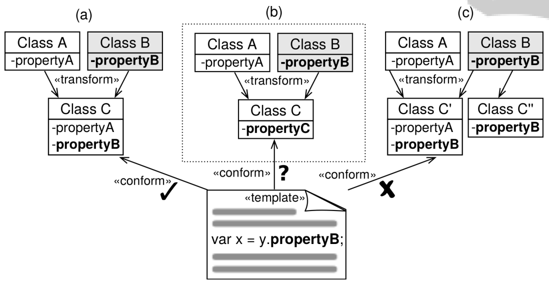


Figure 1: Syntactical metamodel conformance.

This conformance relation is affected by the metamodel composition applied for DSML integration. In the model composition scenarios (`a`) and (`b`) in Figure 1, class `B` is composed into new classes, using different composition operations. In scenario (`a`) the conformance relation is preserved by the composition operation; e.g., a full-property-preserving composition (Herrmann et al., 2007). Hence, the template still applies and can be reused over instances of class `C`. In scenario (`b`), however, the naming difference between `propertyB` and `propertyC`, while structurally equivalent, breaks the conformance relation. The template cannot be applied to instances of class `C`. Refactoring a template clone would be required, for example.

Existing reuse techniques for M2T generator templates fall short on addressing such syntactical mismatches. For example, language-level reuse techniques for M2T transformation languages only provide reuse capabilities at the block level (e.g. for entire template or function blocks) rather than at the expression level. Furthermore, most M2T transformation languages lack the ability to use generic transformations to abstract from different, but structurally compatible metamodels. Generic transformation techniques, as available for some model-to-model (M2M) transformation languages, require an upfront definition in terms of parametric templates and explicit bindings (Cuadrado et al., 2011). However, the demand for such an upfront definition also adds to the development overhead of a DSML. Adapter models mimic the metamodels of the source DSMLs. This, however, impedes the definition of additional M2T transformations that are specific to the integrated DSML. Yet, additional transformations are required to glue the different artifacts that are generated for the integrated DSML.

In this paper, we suggest an approach to overcome the above limitations of M2T transformation templates for DSML integration. Our *transformation rewriting technique* allows for the automated modification of transformation templates to fix important syntactical mismatches between templates and the composed DSML (Wimmer et al., 2010). In scenario (`b`) from Figure 1, for example, the naming difference would be tackled by rewriting the template expression based on tracing data of the class composition, using the new property name `propertyC`.

Our current prototype supports three higher-order rewriting operations (retyping, association retargeting, and property renaming). Note, however, that our approach allows for arbitrary rewriting operations. Semantic heterogeneity in metamodel-model relations (Wimmer et al., 2010) and types of syntactical heterogeneity between source and target metamodels which cannot be resolved in an automated manner are currently not addressed by our approach. This is, for instance, the case for m:n source/target cardinality in scenario (`c`) in Figure 1. Our rewriting technique is applicable to M2T transformation languages which support a subset of the meta object facility (MOF) M2T transformation language (OMG MOFM2T), higher-order transformations (HOTs), and basic model transformation tracing (Object Management Group, 2008; Tisi et al., 2009). All implementation artifacts are available from http://nm.wu.ac.at/modsec.

In Section 2, we give an overview on DSML integration, M2T transformations, and generator templates. Subsequently, we introduce our generic template rewriting approach in Section 3. In Section 4, we describe how our approach is then mapped to the EMF and Epsilon infrastructures. A proof-of-concept implementation for the Epsilon Generation Language (EGL) is introduced. Section 5 gives an example of how our prototype environment can be applied. We discuss limitations as well as the pros and cons of our approach when compared to alternative techniques in Section 6. Section 7 provides an overview of related work and Section 8 concludes the paper.

2 TRANSFORMATIONS AND TRACES FOR DSML INTEGRATION

In general, the process of integrating two or more *source* DSMLs involves four major activities which may be repeated a number of times to derive an integrated *target* DSML (Hoisl et al., 2012). The *language model composition* activity uses the language models (e.g., the MOF or Ecore metamodels including corresponding metamodel-level constraints) of the source DSMLs as input for the definition of a target metamodel. An important output of this activity is a composition specification that includes, for instance, correspondence rules and/or M2M transformations. In the *behavior composition* activity the behavioral semantics attached to the source metamodels are composed to match the target metamodel. Depending on the behavior definition corresponding composition operations are applied (such as M2M transformations or code-to-code transformations). The *concrete syntax composition* activity integrates the concrete syntaxes (e.g., textual, tree-based, tabular, or diagrammatic) of the source DSMLs. Finally, the *platform integration composition* activity integrates the software platforms of the source DSMLs. In particular, this activity integrates artifacts such as M2M and M2T transformations or model interpreters of the source DSMLs. The output of this activity is a set of platform integration specifications which conform to the target metamodel. Sometimes this composition step requires the generation of glue artifacts to realize a system-level composition; e.g., via pipelining, language extension, or front-end integration (Spinellis, 2001).

The artifacts defined in the language model composition activity serve as the input for platform integration. In the remainder of this paper, we focus on M2T generator templates, M2M transformations, and transformation traces. Below, we discuss each of these artifact types in more detail.

M2T Generator Templates as Models. Platform integration as described above (Hoisl et al., 2012) includes the generation of platform-specific artifacts (such as, platform-specific source code or platform configuration and deployment documents; see Figure 2). The generation of these artifacts can be supported via M2T generators which receive a transformation definition and a set of source models as the input to produce a transformed representation of these models. For the remainder of this paper, we especially focus on template-based M2T generators and the corresponding generator templates (Czarnecki and Helsen, 2006). In principle, a generator template consists of two kinds of code. On the one hand, there is template code to access and to select source model data by quantifying over the model structure that is specified in a metamodel (see also Figure 2). On the other hand, a template contains code to expand and to wrap the selected model data into string fragments.

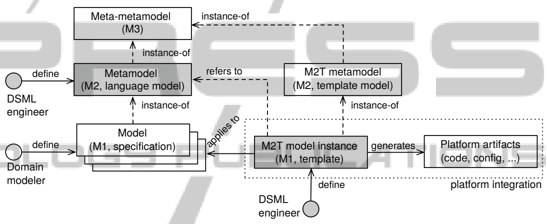


Figure 2: M2T template models.

Template-based M2T transformations are a widely supported platform integration technique in contemporary MDD tool chains, and a variety of template language implementations exist, such as, Eclipse Xpand, Xtend2, EGL, JET, or Acceleo (Rose et al., 2012; Czarnecki and Helsen, 2006). For each of these languages, M2T generators and generator templates can be implemented in different ways. For example, one option is to use specification documents with a textual abstract syntax and a tree-based intermediate representation which is interpreted by the generator. In an alternative approach, we can use a DSL for M2T transformations that is embedded in a general-purpose scripting engine to realize generator templates via scripts. Such "generator template scripts" are then evaluated by the language interpreter (Zdun, 2010).

To abstract from such implementation details and to benefit from generator templates as first-class modeling elements, our approach focuses on the model representations of generator templates. In other words, we consider generator templates as instances of a conceptual M2T template metamodel (see Figure 2). Rather than including all features that are available in different template languages, our approach requires only a generic subset of the features defined through the MOFM2T specification (Object Management Group, 2008). In this way, the approach is portable to contemporary M2T template languages.

M2M Transformations. In our approach we consider M2M transformations at the M1 and M2 modeling levels (see Figure 3). First, we compose the core language models of two (or more) source metamodels into a target metamodel. This composition is achieved via transformations that refer to the corresponding metamodel structures (M3). Second, M1 models of respective generator templates (see also Figure 2) are transformed into new M1 template models to adapt them to metamodel changes that result from the DSML integration. These M2M transformations are higher-order model transformations (HOTs): Transformations receiving input/output models which are themselves model representations of transformations; probably even expressed in the same transformation language (Tisi et al., 2010).

Note that the programming model that is used by a certain M2M transformation language (e.g. relational, operational mappings, hybrid) is transparent to our approach (Czarnecki and Helsen, 2006). Nevertheless, to demonstrate our approach we use hybrid transformation rules as supported by, for instance, the Atlas Transformation Language (ATL) and the Epsilon Transformation Language (ETL) in our examples.

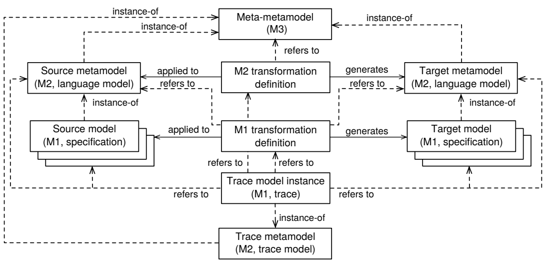

Meta-metamodel (M3)
instance-of
instance-of
instance-of
refers to
Source metamodel (M2, language model)
applied to
refers to
M2 transformation definition
generates
refers to
Target metamodel (M2, language model)
instance-of
instance-of
Source model (M1, specification)
applied to
M1 transformation definition
generates
Target model (M1, specification)
refers to
refers to
Trace model instance (M1, trace)
refers to
refers to
instance-of
Trace metamodel (M2, trace model)

Figure 3: M2M transformations and traces.

Transformation Traces. Even though our approach is generic and does not require a specific variant of M2M transformations, we assume that a transformation history is available. This history includes transformation traces that document the M2M transformations between language models (the M2 level in Figure 3). In particular, each transformation trace establishes a persistent link between a source and a target model element which are connected via a model transformation operation (merging, extension, renaming). Moreover, each transformation trace refers to a corresponding transformation rule. In order to introspect on these traces (for example, to identify the kind of transformation operation performed), transformation traces must be represented as first-class models. In other words, each trace must be an instance of a dedicated trace metamodel that complies with the transformed models at the M3 level (see Figure 3). Note, however, that we neither define restrictions on the time the traces are recorded (allowing, for example, partial evaluation of transformation rules, and runtime tracing) nor do we require a specific tracing engine: Built-in tracing, traces generated by transformation rules, as well as internal or external trace stores are supported.

3 TEMPLATE SYNTAX REWRITING

Before introducing our template rewriting approach, we must first review the types of potential mismatch between different DSML metamodels in more detail. We consider MOF-compliant metamodels. Regarding M2T template language concepts, we use the corresponding MOFM2T terminology for explanatory purposes (Object Management Group, 2008).

Figure 4 provides a sketch of DSML `A` and DSML `B` being composed into DSML `C` using an M2M transformation definition. The mismatch problem can then be restated as question: *How can we make the generator templates `A` and `B` apply to instances of the composed metamodel `C` rather than to instances of metamodels `A` and `B`, respectively?* To answer this question, we must establish some background: First, it is important to identify the types of structural differences encountered during metamodel composition. More specifically, we must collect the details about the structural differences and make them accessible to the template transformation. Then, the elements of a generator template which are affected by these structural differences must be highlighted. Finally, corresponding transformations of the generator templates must be defined. The objective of template rewriting is the transformation of the source templates (`A`, `B`) into derivatives (`A′`, `B′`) which refer to metamodel `C` directly (see Figure 4).

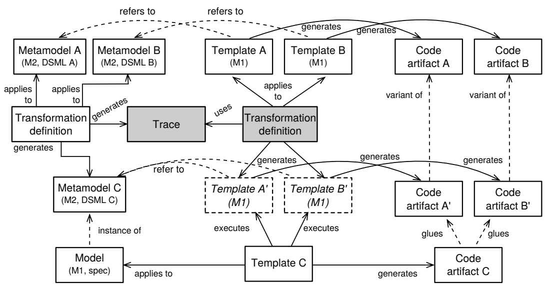


Figure 4: M2T template rewriting for DSML integration.

Metamodel Composition. When composing the source metamodels (`A` and `B` in Figure 4), the DSML engineer can choose from a variety of model composition operations when defining the transformation, only limited by the M2M transformation language's capacity (Vallecillo, 2010). These include *model mergers* using merge operators with different precedence and conflict resolution schemes, operating at different granularity levels (package, metaclass). In a *model extension*, subsets of either language model enter the new metamodel as disjoint sets to complement each other. *Model refinements* realize is-kind-of dependencies between all or selected elements of the source metamodels. *Model interfacing* involves introducing model elements specific to the new metamodel as a structural glue between elements merged from the source metamodels. A *hybrid composition* can involve any combination of the above operations.

In our approach, we restrict the composition operations so that structural semantics of the source metamodels are preserved. Thus, it is assured that the code artifacts generated based on the composed metamodel are semantically equivalent to the code artifacts generated with M2T templates of the individual source metamodels. The source-target cardinality can be either *1:1* or *n:1*: Either the element (class, attribute etc.) is preserved in the target model or a set of elements is merged into one new element.

Our template rewriting approach considers the effects of metamodel composition as changes between two model states across predefined model correspondences, rather than as a sequence of transformation operations. State-based differentiation refers to computing the changes between the source and target models after the completed composition, by contrasting the source and target metamodels at the M2 level. Listing 1 presents two examples of state-based differentiation to identify element name changes, defined as OCL expressions over two couples of source and target elements of two MOF-compliant metamodels. The two introspection definitions `isRenamedClass` and `isRenamedProperty` reflect important cases for M2T template rewriting, to be reconsidered later in this section.

Listing 1: State-based differentiation.

```
def: isRenamedClass(source : NamedElement,
                    target : NamedElement) : Boolean
    =  source.oclIsKindOf(Class) and
       target.oclIsKindOf(Class) and
       target.name != source.name

def: isRenamedProperty(source : NamedElement,
                       target : NamedElement) : Boolean
    =  source.oclIsKindOf(Property) and
       target.oclIsKindOf(Property) and
       target.name != source.name
```

A state-based technique has a number of advantages. To begin with, it is minimalistic and implementable for several M2M transformation engines. In addition, it turns our template rewriting approach agnostic about the actual composition operations used.

Trace Models. To make state differences between source and target metamodels computable, the correspondences between metamodel elements established during the metamodel composition must be preserved at metamodel composition time (Paige et al., 2011). Alternative tracing techniques, such as, implicit tracing or model annotation (Drivalos et al., 2008), are not suitable for state-based differentiation. The tracing scheme can be achieved by storing the transformed element couples as instances of a simplistic trace model (see Figure 5). Instances of `Trace` refer to a `source` metamodel element and a `target` metamodel element which is paired during metamodel composition. In addition, a reference to the `Transformation` is stored along with the element couple. The `Trace` concept provides the context for the introspection expressions in Listing 1. This tracing scheme is suitable for detecting unsupported heterogeneity situations in the context of model composition (i.e., composition operations violating defined structural semantic preservation conditions, such as, m:n cardinality of source/-target element mappings). Traces can be used to interrupt the rewriting process and to aid the debugging of allowed composition operations (Amar et al., 2008).

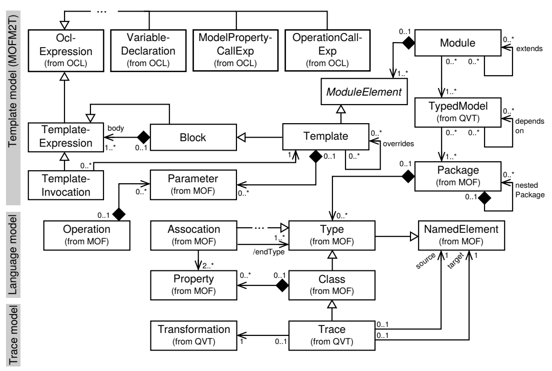


Figure 5: Excerpts from MOF and MOFM2T.

Template Models. Based on the transformation data gained from introspecting the transformation traces, the templates are to be adapted (e.g., a template `A'` is to be derived from template `A` in Figure 4). To represent M2T generator templates as first-class models, we consider them instances of a subset of the MOFM2T reference language (see Figure 5).

In MOFM2T, transformations are structured in `Modules` (e.g., for namespace or public/private part definitions) which can contain a number of `Templates` (see Figure 5). A `Template` as a specialized `Block` contains string production expressions (`TemplateExpressions`) with placeholders for data to be extracted from models. `Templates` can be invoked with `Parameters` and can return `Parameters`. They realize function-style language elements (Wimmer et al., 2012). A `Template` can override other templates, with the overriding template being invoked in place of the shadowed template. A `TemplateExpression`, among others, represents expressions for calls to model elements, for declaring and managing variables, for declaring control statements, and for expanding to strings. A `TemplateInvocation` specifies an invocation of a `Template`. As `TemplateInvocation` is a specializing classifier of `TemplateExpression`, `Templates` can be invoked from within other `Templates`. A `TypedModel` specifies an input model (a `Package`) to be referenced and accessed throughout the `TemplateExpressions`. The `Types` contained by the `Package` represent the domain of model element types available to the `Templates`. These type references are primary rewriting targets.

Template Model Transformations. Metamodel changes as identified by state-based differentiation (`isRenamedClass` and `isRenamedProperty` in Listing 1) affect the M2T generator templates syntactically in two ways. First, block expressions (variants of `OclExpressions`) maintain references to `Types` (as can be learned from Figure 5). This is the case for `Parameters` of `Templates` and for type-aware template expressions, in particular type annotations in `VariableDeclarations` and type annotations for parameters of operation calls (`OperationCallExp`). Second, navigating `OclExpressions`, such as, `ModelPropertyCallExp` may refer to renamed metamodel elements (e.g., `Property`). From these syntactical dependencies, three rewriting requirements follow for pairs of source and target metamodel elements, which are represented by a set of `Trace` instances (see Figure 5):

Retyping: References to a `Type` named after a renamed source `Class` must be replaced by the target `Class` name.

Association Retargeting: When an `Association` between two `Classes` is redirected and receives another target (i.e., a `Property` owned by another `Class`), the corresponding `endTypes` must be modified. Thus, the return `Type` of the expression must be adapted and set to the new `endType`. Note that the name of a corresponding navigation reference (e.g., a property call to retrieve an element) in navigating `OclExpressions` is not affected.

Property Renaming: A renamed `Property` (`isRenamedProperty`) affects navigating `OclExpressions` as the navigation path in these expressions changes. To rewrite these navigation paths according to the renaming, the `property` attributes of these `OclExpressions` are adjusted.

All three rewriting operations may occur repeatedly for identical pairs of transformed source and target metamodel element types, depending on the number of state-based differences computed from the set of `Traces`. Corresponding transformations must be defined in an M2M transformation definition which operates on the source templates (`A`, `B`) based on a set of `Traces` to produce syntactically adapted template model instances (`A′`, `B′`; see Figure 4).

4 EPSILON/EMF PROTOTYPE

Based on our notion of template rewriting (see Section 3), we introduce a prototypical realization of this rewriting technique for the EMF and the Epsilon family of model transformation languages. In this technology projection, MOF-compliant DSML models are approximated by Ecore metamodels. To perform language model composition, as described in Section 3, we use transformation definitions expressed in Epsilon task languages. M2T generator templates are represented by EGL transformations. To obtain model representations of EGL templates, we map the respective MOFM2T concepts to their corresponding language concepts in the Epsilon language family. The Epsilon language runtime provides a built-in tracing facility for capturing transformation correspondences between Ecore metamodels as required for our rewriting approach (Kolovos et al., 2012).

Ecore Metamodel Composition. In the EMF/Epsilon toolkit, metamodel composition is divided into three tasks: (1) matching, (2) copying, and (3) merging metamodels. The first task (matching) is performed via the Epsilon Comparison Language (ECL) and has the source metamodels, as well as comparison rules provided by the DSML engineer as inputs. During copying, the unmodified metamodel elements (i.e., which do not match any comparison rule) are copied into the target metamodel of the composed DSML. This can be achieved using the ETL. The third step in an Epsilon-specific DSML composition applies merge rules defined by the DSML engineer on element triples of the source metamodels and the target metamodel. The output of this 3-pass transformation is a composed DSML metamodel and transformation traces, to be used for state-based differentiation (see Section 3).

The introspection operations `isRenamedClass` and `isRenamedProperty` translate into the Epsilon

Object Language (EOL) equivalents for Ecore metamodels in Listing 2, to compute the state-based differences between the source and the target metamodels.

Listing 2: State-based differentiation for Ecore metamodels.

```
operation isRenamedClass(source : Ecore!EObject,
                         target : Ecore!EObject) :
                               Boolean {
        return  source.isKindOf(Ecore!EClass) and
                target.isKindOf(Ecore!EClass) and
                target.name <> source.name;
        }

operation isRenamedProperty(source : Ecore!EObject,
                            target : Ecore!EObject) :
                                  Boolean {
        return  source.isKindOf(Ecore!EReference) and
                target.isKindOf(Ecore!EReference) and
                target.name <> source.name;
        }
```

Minimal Ecore Trace Model. To provide a model representation of transformation traces (as sketched in Figure 5 in Section 3), we realized a custom trace metamodel for our prototype. Listing 3 gives the metamodel definition. `CompositionLink` is the corresponding concept of `Trace` in Figure 5.

Listing 3: Trace metamodel in EMFatic textual syntax.

```
@namespace(uri="CompositionTrace", prefix="
      CompositionTrace")
package CompositionTrace;

class CompositionLink {
   ref EObject source;
   ref EObject target;
}
```

During the metamodel composition step (see above), transformation correspondences obtained from the Epsilon tracing machinery are stored as `CompositionLink` instances. Each `CompositionLink` stores a pair of source and target `EObjects` extracted from the Epsilon tracing sources (ECL match, ETL transformation, and EML merge traces). In this custom metamodel, references to the respective merge and transformation rules are omitted for brevity.

Ecore Metamodel for EGL Templates. EGL templates are natively processed by an ANTLR-based parser and transformed into an abstract syntax tree (AST) structure. Currently, Epsilon neither provides EMF metamodel representations of its language family, nor the tooling to perform round-tripping between ASTs and any model representation. Therefore, we extended an early prototype for EOL model representations (Wei, 2012) to cover EGL language concepts. We extracted the language abstractions from (Kolovos et al., 2012) and by screening the Epsilon code base for missing details.

Figure 6 presents an excerpt of the extended Ecore metamodel. In this technology projection, the M2T template model concepts introduced in Figure

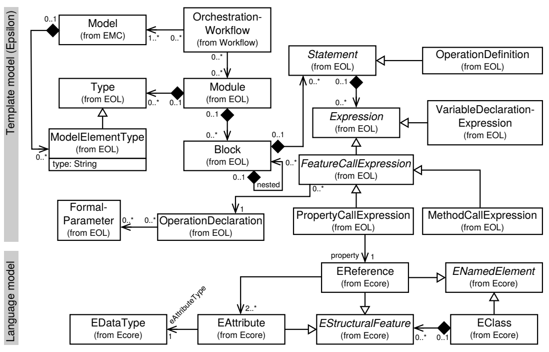

Template model (Epsilon)
Language model
Model (from EMC)
Orchestration-Workflow (from Workflow)
Statement (from EOL)
OperationDefinition (from EOL)
Type (from EOL)
Module (from EOL)
Expression (from EOL)
VariableDeclaration-Expression (from EOL)
ModelElementType (from EOL)
type: String
Block (from EOL)
nested
FeatureCallExpression (from EOL)
Formal-Parameter (from EOL)
OperationDeclaration (from EOL)
PropertyCallExpression (from EOL)
MethodCallExpression (from EOL)
property
EReference (from Ecore)
ENamedElement (from Ecore)
eAttributeType
EDataType (from Ecore)
EAttribute (from Ecore)
EStructuralFeature (from Ecore)
EClass (from Ecore)
0..1
1..*
0..*
2..*
1

Figure 6: Excerpt from the Ecore metamodel for the Epsilon language family.

5 must be mapped to their Epsilon correspondences in Figure 6. Some Epsilon concepts such as `Module` and `Block` are directly compliant with MOFM2T. Nevertheless, some EGL concepts deviate from the MOFM2T metamodel structure: For example, Epsilon distinguishes between `Statements` and `Expressions`. In contrast, the MOFM2T specification (Object Management Group, 2008) summarizes text production rules of `Templates` as specialized expressions (`TemplateExpression`). Generally speaking, the complete EOL/EGL metamodel exceeds the domain coverage of the MOFM2T metamodel because Epsilon provides an integrated collection of several task-specific languages. For our prototype, however, only a subset of the Epsilon metamodel was relevant (see Figure 6). The resulting concept mapping used for our prototype is shown in Table 1.

Table 1: Mappings between MOFM2T and Epsilon.

MOFM2T	**Epsilon**
`Association`	`EReference`
`Block`	`Block`
`Class`	`EClass`
`ModelPropertyCallExpr`	`PropertyCallExpression`
`Module`	`Module`
`NamedElement`	`ENamedElement`
`OperationCallExpr`	`MethodCallExpression`
`Parameter`	`FormalParameter`
`Property`	`EAttribute`
`Template`	`OperationDefinition`
`TemplateExpression`	`Statement`, `Expression`
`TemplateInvocation`	`FeatureCallExpression`
`Type`	`Type`, `EDataType`
`TypedModel`	`Model`, `ModelElementType`
`VariableDeclaration`	`VariableDeclaration-Expression`

ETL Transformations on EGL Template Models. Finally, assuming the availability of model representations of source EGL templates and of transformation traces (`CompositionLink` instances), equivalents to the template-to-template transformations as defined in Section 3 must be defined based on the mappings shown in Table 1. Note that these transformations apply to any occurrence of EGL templates to be adapted. The definitions below are generic in this sense and must be bound to the concrete templates under transformation. These M2M transformations are expressed in ETL.

Retyping, Association Retargeting. In Epsilon, the type of a model element acting as an input to a transformation is represented by the `ModelElementType` metaclass. To change the type reference owned by a template element (e.g., an expression) or to retarget an association (i.e., an `EReference` in Ecore), the respective `type` attribute of the corresponding `ModelElementType` element must be changed to match the name of the target metamodel metaclass (see Figure 6). The corresponding ETL rule is depicted in lines 10–20 of Listing 4.

Property Renaming. An `EReference` in Ecore represents a navigation axis from one `EClass` to another by pairing the opposite metaclasses. Therefore, to reflect a renamed `EReference` in the template model, the `property` attribute of a `PropertyCallExpression` element must be adapted (see Figure 6). The ETL rule for this transformation is defined in lines 22–32 of Listing 4.

Listing 4: EGL snippet creating ETL rewrite rules.

```
 1 for (l in links) {
 2   var src : Ecore!EObject; var trgt : Ecore!EObject;
 3   src = l.source;
 4   trgt = l.target;
 5   if (src.name <> trgt.name) {
 6     etl = etl + TemplateFactory.prepare(renameElement(
           src, trgt)).process();
 7   }
 8 }
 9
10 @template
11 operation renameElement(src : Ecore!EClass, trgt : Ecore
       !EClass) { %]
12   rule retype[%=src.name%]2[%=trgt.name%]
13     transform s : egl_in!ModelElementType
14     to t : egl_out!ModelElementType
15     extends Type
16     {
17       guard : s.type == "[%=src.name%]"
18       t.type = "[%=trgt.name%]";
19     }
20 [% }
21
22 @template
23 operation renameElement(src : Ecore!EReference, trgt :
       Ecore!EReference) { %]
24   rule rename[%=src.name%]2[%=trgt.name%]
25     transform s : egl_in!PropertyCallExpression
26     to t : egl_out!PropertyCallExpression
27     extends FeatureCallExpression
28     {
29       guard : s.property == "[%=src.name%]"
30       t.property = "[%=trgt.name%]";
31     }
32 [% }
```

As mentioned above, the two M2M transformation rules must be instantiated for a concrete set of EGL template models (e.g., to reflect the concrete element names). As shown in Listing 4, the ETL rules are themselves generated by instantiating an M2T EGL template for a given set of transformation traces. For demonstration purposes, the top-level EGL script processes the available traces retrieved from the preceeding metamodel composition in lines 1–8, to dispatch to the expanded ETL transformation rules for each pair of source and target elements with name mismatches (see line 5 in Listing 4).

Epsilon Composition and Rewriting Procedure. A process flow view of the overall composition and transformation steps is presented in Figure 7. This Eclipse-specific process flow realizes the abstracted scheme shown in Figure 4. Two activities must be performed as the prerequisites for applying the actual rewriting to the M2T EGL templates: 1) The DSML metamodel composition in three Epsilon-specific steps (matching, copying, merging) and 2) the transformation of M2T EGL templates into their model representations—that is, the instances of the metamodel depicted in Figure 6. The trace model generated during the metamodel composition and the instantiated ETL rewrite rules enter the actual template-to-template transformation along with the EGL template models. The rewritten EGL template models are finally serialized into EGL script representations to be applied to the composed metamodel at the end of this process (this last step is not shown in Figure 7).

This process flow can be automated in Epsilon by providing a specific build script which turns the flow into a sequence of Epsilon-specific Apache Ant tasks (Kolovos et al., 2008). Such an `OrchestrationWorkflow` defines the sequence of tasks, such as, `Model` loading or `Module` invocation (see Figure 6). Alternatively, such a process flow can be realized by instrumenting the Epsilon and EMF APIs in a piece of Java glue code.

5 AN INTEGRATION SCENARIO

In this section, we describe a composition scenario of two DSMLs to exemplify the integration process. We run through the whole process of applying one higher-order rewrite rule[1]. The first DSML models system audits (referred to as DSML `A`, hereafter) by providing abstractions for audit events and audit rules. The second DSML (DSML `B`, hereafter) allows for modeling generic state machines. The scenario integrates the two DSMLs into a composed DSML `C` capable of

[1]All software artifacts as well as the complete example can be obtained from http://nm.wu.ac.at/modsec.

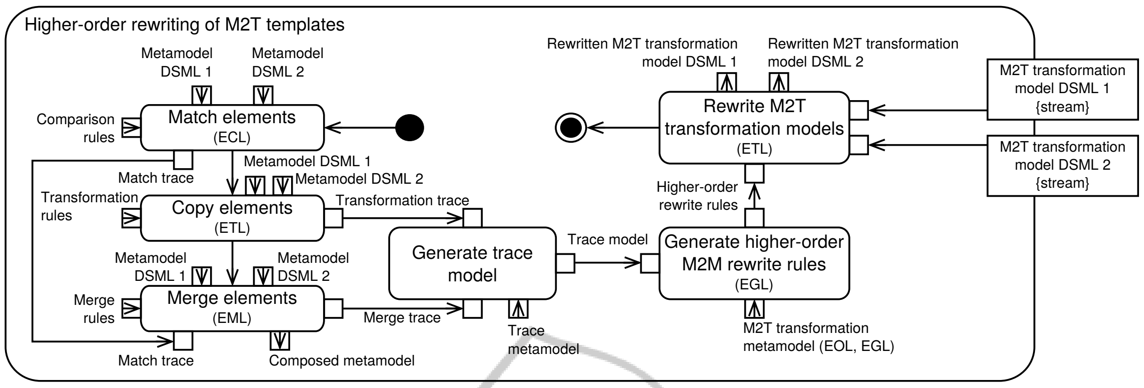

Higher-order rewriting of M2T templates
Metamodel DSML 1
Metamodel DSML 2
Comparison rules
Match elements (ECL)
Match trace
Metamodel DSML 1
Metamodel DSML 2
Transformation rules
Copy elements (ETL)
Transformation trace
Metamodel DSML 1
Metamodel DSML 2
Merge rules
Merge elements (EML)
Merge trace
Match trace
Composed metamodel
Generate trace model
Trace metamodel
Trace model
Generate higher-order M2M rewrite rules (EGL)
M2T transformation metamodel (EOL, EGL)
Higher-order rewrite rules
Rewrite M2T transformation models (ETL)
Rewritten M2T transformation model DSML 1
Rewritten M2T transformation model DSML 2
M2T transformation model DSML 1 {stream}
M2T transformation model DSML 2 {stream}

Figure 7: Process of higher-order rewriting of M2T templates.

modeling a reactive distributed system with auditing support. Both DSMLs provide M2T generator templates written in EGL to generate Java code. The objective is to reuse these EGL templates for models of DSML `C` through syntax rewriting. For this scenario, we explain the application of a particular higher-order rewriting rule to a template specific to DSML `A`.

In DSML `A` for system audits, an `AuditRule` subscribes to a `Signal` type and, when an `AuditEvent` is triggered, checks the corresponding `Conditions` against the published `Signal` occurrence (see Figure 8). If all `Conditions` evaluate to true, a notification action will be executed to perform audit-related tasks, such as generating an entry in an audit trail or notifying the system administrator—not displayed in Figure 8; for details see (Hoisl and Strembeck, 2012).

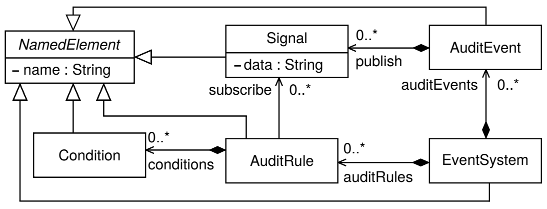


Figure 8: DSML `A`—auditing in event-based systems.

For DSML `B`, we have chosen a state/transition pattern (see Figure 9) for its communicability in an example. In a state machine, `Transitions` model the change from one `State` to another. A `Transition` is triggered by an `Event`.

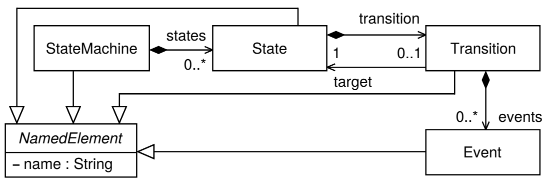


Figure 9: DSML `B`—a state/transition behavioral system.

DSML Metamodel Composition. In this step, we merge the `AuditEvent` element from DSML `A` and the `Event` element from DSML `B` into a unified `AuditElement'` element of the composed DSML `C` (see Figure 10). Thereby, we connect both DSMLs structurally by merging these two core concepts into one concept of DSML `C`. Otherwise, the metamodel composition preserves all structural semantics present in the source DSMLs (inheritance, attributes, references).

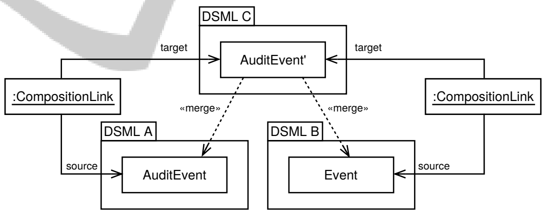


Figure 10: DSML composition via element merge.

This concept merge is defined by the ECL comparison rule shown in Listing 5. Therein, a match is defined iff the corresponding metamodel elements of the two DSMLs are named *AuditEvent* and *Event*, respectively (line 5)[2].

Listing 5: ECL comparison rule for `AuditEvent'`.

```
1 rule AuditEventandEvent2AuditEvent '
2   match l : EventSystem!EClass
3   with r : StateMachine!EClass {
4   compare :
5     l.name = 'AuditEvent' and r.name = 'Event'
6 }
```

For all elements missed by the rule in Listing 5, a direct copy operation into the target metamodel is defined via an ETL transformation (not shown). All elements matching the above ECL rule are processed

[2]Please note that we show only relevant code parts in the example listings (excerpts).

by the merge operation in Listing 6. Therein, a new element name is constructed (line 5) and all properties, references, and inheritance relations (lines 6–7) from both the DSML `A` and the DSML `B` metamodels are transferred into the newly created element in the target metamodel. This preserves the n:1 source/target cardinality (see Section 3).

Listing 6: EML merge rule for `AuditEvent'`.

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

Ecore-based Trace Model. The merge and the transformation yield an instance of the trace metamodel (see Listing 3, Section 4). In Figure 10, the two resulting instances of `CompositionLink` are illustrated, recording pairs of transformation sources and transformation targets: (`AuditEvent`, `AuditEvent'`) and (`Event`, `AuditEvent'`).

Ecore-based Template Model. Listing 7 shows an example code snippet of an EGL template. For now, only line 1 is of interest: A loop is defined iterating over all `AuditEvents` in an `EventSystem`. The return type of the reference `EventSystem.auditEvents` is defined as `AuditEvent`. In the composed DSML `C`, the corresponding concept is `AuditEvent'`. To reuse this snippet for DSML `C`, the type annotation of the iterator variable `ae` must be modified to `AuditEvent'`.

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 } %]
```

For applying syntactical rewrite rules, the EGL template (Listing 7) needs to be transformed into its model representation. Listing 8 shows the corresponding instance model representation of line 1 of Listing 7 (simplified).

Listing 8: 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>
```

EGL Template Model Transformation. The abstracted higher-order rewrite rules documented in Listing 4, Section 4, must be instantiated using the trace model shown in Figure 10. The ETL rewrite rule generated by this template instantiation for the DSML `A` element `AuditEvent` is reproduced in Listing 9. All other rewrite rules are omitted due to space limitations. The rule in Listing 9 resets the `type` properties of `ModeElementType` instances, which equal to `AuditEvent`, to the value `AuditEvent'`.

Listing 9: ETL higher-order rewrite rule.

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

Applying this rule to the EGL model as shown in Listing 8 results in an EGL model which is corrected for the changed type name. Line 3 of Listing 10 shows that the type of the iterator variable named `ae` was effectively changed to `AuditEvent'`.

Listing 10: 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 be able to execute the rewritten EGL template, in a last step, the EGL model representation in Listing 10 is serialized back into EGL template code (see Listing 11). Line 1 shows the changed type of the loop iterator named `ae`. This type conforms to the composed DSML metamodel `C` (see Figure 10). The rewritten EGL code template can be executed over models of DSML `C`.

Listing 11: EGL snippet with changed iterator type.

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

6 DISCUSSION

Our approach to rewriting M2T generator templates syntactically is motivated by examining barriers to reusing DSMLs, in general, and to reusing DSML-based M2T transformations for platform integration, in particular. An important barrier results from M2T transformation languages lacking the capacity of abstracting from certain structural conditions of a concrete metamodel (Wimmer et al., 2012).

While variants of template genericity (Cuadrado et al., 2011; Varró and Pataricza, 2004) help decouple from early bound references to concrete model element types, naming differences affecting navigational axes are not addressed, for example. Therefore, our approach can complement M2T template genericity. Given that generic transformations can also be implemented using HOTs on M2T templates, there is even a shared implementation vehicle.

In addition, our approach contributes to capturing M2T transformation logic independently from a particular transformation language or engine. This platform abstraction (Wimmer et al., 2012) contributes to the reusability of M2T transformations, as they can be migrated to another language environment. By providing a precise definition of our approach in terms of the MOFM2T specification (see Sections 3 and 4), we establish such an M2T platform abstraction.

Another barrier to M2T transformation reuse is the lack of contextual information about the conditions and requirements of reuse (Wimmer et al., 2012). While not fully elaborated in this paper, we enumerate working assumptions on the structural semantics of metamodel transformations (e.g., cardinality classes supported) in Section 3. These working assumptions can be formalized into executable pre- and post-conditions (e.g., OCL expressions) stored at the model-level. The conditions can then be evaluated based on the transformation traces generated during metamodel composition to establish whether the rewriting transformations are applicable.

One critique of using HOTs (Tisi et al., 2009) is that they expose the engineer to the internals of the transformation language (Tisi et al., 2010) and thus hinder reuse. In the case of M2T transformation models, this model complexity bears the risk of derailing the widely opaque text production expressions so that the platform artifacts are emitted malformed. In our approach, the M2T generator templates are represented by comparatively small metamodel domains (i.e., subsets of MOFM2T and the corresponding EGL mapping). On top, the HOTs remain completely hidden from the DSML engineer because they are themselves generated by template instantiation on the tracing data (see Section 3). This is a compromise balancing between automation and a limited support for metamodel heterogeneity.

The degree of DSML and M2T transformation reuse is directly related to the relative effort caused by the generative environment (transformation adjustments, manual configuration, automation of the generation tasks). To improve the reuse degree, this extra effort must be minimal. In our approach, most of the artifacts are only specified or generated once upon composing a DSML (e.g., the rewrite rules). Only when the source DSMLs are modified, the composing transformation definition must be updated. M2T transformations specific to the integrated DSML can vary independently from the generated M2T transformations. This allows for generating different kinds of patch code; e.g., pipelining or language extension (Spinellis, 2001).

7 RELATED WORK

The approach presented in this paper relates to existing work in two areas. First, we distinguish between three relevant language- and model-level **reuse techniques for generator templates**.

Higher-Order Transformation (HOT). Our syntax rewriting approach takes two M2T transformation models (EGL templates) as input and produces two modified M2T transformations (EGL templates) by applying an M2M transformation (via ETL) over these two transformation models. Hence, we apply HOTs for *transformation modifications* (Tisi et al., 2009). In recent years, a variety of alternative HOT application scenarios have attracted attention, including transformation analysis, transformation generation, and transformation composition (Tisi et al., 2009). In addition, language-level support for HOTs has been improved (Oldevik and Haugen, 2007; Tisi et al., 2010). However, related work has concentrated on specific transformation platforms, for instance, ATL (Tisi et al., 2010), rather than on HOTs in a technology-independent manner.

Generic Templates. Generic templates abstract from the underlying metamodel and contain transformation rules which refer to abstracted metamodel types in terms of type variables (Cuadrado et al., 2011; Varró and Pataricza, 2004). The type variables are then late-bound to specific model element types at transformation runtime. This form of type parametrization must be employed by the DSML engineer right from the beginning to construct the generator templates in a reusable manner. Some approaches also require the explicit definition of bindings for type variables and structural adapters against a concrete metamodel to cope with types of structural heterogeneity in metamodels. This certainly adds to the initial effort of constructing the supporting transformations for a DSML. Our approach differs as we do not base the rewriting of templates on placeholder variables or adapters, but we rather extract the changes from a trace model. This offers the benefit of automation and of unanticipated reuse of generator templates. At the same time, our approach is limited in its expressiveness to handle metamodel heterogeneity (see Section 3). Because both approaches use HOTs as implementation vehicle, they can complement each other. Transformation genericity has not been documented for M2T generator templates so far.

Adapter Models. To establish metamodel conformance for generator templates, another strategy is the use of intermediate models which adapt model accesses by the generator template to match the original metamodel structure. To mimic the original metamodel, an adapter model consists of relational expressions which bind to the transformed metamodel and return model values according to the declared correspondences (Morin et al., 2009). Adapter models provide for unanticipated template reuse, however, as for M2T transformations the generated platform artifacts would not reflect the derived or newly introduced domain concepts. This increases the cognitive distance for the integrating DSML engineer. The generation of glue code using M2T transformations is restricted because concept correspondences between the integrated DSML and the source DSMLs can not be leveraged in code generation.

The second related area is the **encoding of tracing data** to be used in model transformations.

Modeled Traces. Traces captured along the transformation process can be stored in a *separate* trace model or can be *attached* to the source/target model, e.g., via model annotations (Paige et al., 2011; Amar et al., 2008). The complexity of traces depends on their scope (e.g., only selected or all rules) and the tracing data needed. For our approach, it is sufficient to store trace links between source and target elements. Besides, the trace metamodel can be defined for more general or very specific purposes, such as our template rewriting scenario (Drivalos et al., 2008).

Delta Models. Delta models are generated by comparing the input and output models of an M2M transformation (ex-post). The creation of delta models (or difference models) is comparable to diff tools for text artifacts. In contrast to model traces, delta models are an indirect method. Traces are not directly captured at transformation time, the actual transformation correspondences at the element level cannot be reconstructed. This black-box encoding of tracing data (Diskin et al., 2011) is not suitable for a rewriting approach which requires exact knowledge of source and target correspondences.

8 CONCLUSIONS

In this paper, we present an approach to rewriting M2T generator templates syntactically for reusing them in DSML integration. By considering M2T generator templates as first-class models and reusing M2M transformation traces, we developed a rewriting approach based on higher-order model transformations (HOTs). This approach is independent from a concrete transformation platform and the documentation in terms of the MOFM2T specification facilitates uptake in MOFM2T-compliant transformation languages. To demonstrate the feasibility of this rewriting technique, we provide a prototype implementation and a DSML integration example based on the Eclipse EMF project and the Epsilon language family. As a side product, we so contributed to constructing a metamodel for the M2T-specific parts of the Epsilon language infrastructure.

In our future work, we will extend our prototype to support a wider range of metamodel-level composition operations (e.g., extends, alternatives). Moreover, we will evaluate the applicability of our ideas to other transformation languages with HOT support (e.g., ATL).

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.

The authors would like to thank William Wei and Dimitris Kolovos for their valuable advice on various Epsilon issues.

REFERENCES

Amar, B., Leblanc, H., and Coulette, B. (2008). A Traceability Engine Dedicated to Model Transformation for Software Engineering. In *Proc. of the 4th ECMDA Traceability Workshop*, volume WP09-09 of *CTIT Workshop Proceedings*, pages 7–16. Centre for Telematics and Information Technology (CTIT), University of Twente.

Bettin, J. (2002). Measuring the Potential of Domain-Specific Modelling Techniques. In *Proc. of the 2nd Domain-Specific Modelling Languages Workshop*, pages 39–44. Helsinki School of Economics.

Cuadrado, J. S., Guerra, E., and de Lara, J. (2011). Generic Model Transformations: ”Write Once, Reuse Everywhere". In *Theory and Practice of Model Transformations*, volume 6707 of *LNCS*, pages 62–77. Springer.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646.

Diskin, Z., Xiong, Y., and Czarnecki, K. (2011). From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology*, 10:6:1–25.

Drivalos, N., Paige, R. F., Fernandes, K. J., and Kolovos, D. S. (2008). Towards Rigorously Defined Model-to-Model Traceability. In *Proc. of the 4th ECMDA Traceability Workshop*, volume WP09-09 of *CTIT Workshop Proceedings*, pages 17–26. Centre for Telematics and Information Technology (CTIT), University of Twente.

Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., and Völkel, S. (2007). An Algebraic View on the Semantics of Model Composition. In *Proc. of the 3rd European Conference on Model Driven Architecture—Foundations and Applications*, volume 4530 of *LNCS*, pages 99–113. Springer.

Hoisl, B. and Strembeck, M. (2012). A UML Extension for the Model-driven Specification of Audit Rules. In *Proc. of the 2nd International Workshop on Information Systems Security Engineering*, volume 112 of *LNBIP*, pages 16–30. Springer.

Hoisl, B., Strembeck, M., and Sobernig, S. (2012). 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*, pages 337–344. ACTA Press.

Kalnina, E., Kalnins, A., Celms, E., and Sostaks, A. (2010). Graphical Template Language for Transformation Synthesis. In *Software Language Engineering*, volume 5969 of *LNCS*, pages 244–253. Springer.

Kolovos, D., Rose, L., Paige, R., and García-Domínguez, A. (2012). The Epsilon Book. Available at: http://www.eclipse.org/epsilon/doc/book/. Last accessed: 30.11.2012.

Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2008). A Framework for Composing Modular and Interoperable Model Management Tasks. In *Proc. of the 1st ECMFA Workshop on Model Driven Tool and Process Integration*, pages 79–90. Frauenhofer IRB.

Krueger, C. W. (1992). Software Reuse. *ACM Computing Surveys*, 24(2):131–183.

Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., and Jézéquel, J.-M. (2009). Weaving Variability into Domain Metamodels. In *Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 690–705. Springer.

Object Management Group (2008). MOF Model To Text Transformation Language. Available at: http://www.omg.org/spec/MOFM2T. Version 1.0, formal/2008-01-16. Last accessed: 30.11.2012.

Oldevik, J. and Haugen, Ø. (2007). Higher-Order Transformations for Product Lines. In *Proc. of the 11th International Software Product Line Conference*, pages 243–254. IEEE Computer Society.

Paige, R., Drivalos, N., Kolovos, D., Fernandes, K., Power, C., Olsen, G., and Zschaler, S. (2011). Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. *Software and Systems Modeling*, 10:469–487.

Rose, L. M., Matragkas, N., Kolovos, D. S., and Paige, R. F. (2012). A Feature Model for Model-to-Text Transformation Languages. In *Proc. of the 2012 ICSE Workshop on Modeling in Software Engineering*, pages 57–63. IEEE Computer Society.

Spinellis, D. (2001). Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software*, 56(1):91–99.

Tisi, M., Cabot, J., and Jouault, F. (2010). Improving Higher-Order Transformations Support in ATL. In *Proc. of the 3rd International Conference on Theory and Practice of Model Transformations*, volume 6142 of *LNCS*, pages 215–229. Springer.

Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J. (2009). On the Use of Higher-Order Model Transformations. In *Proc. of the 5th European Conference on Model Driven Architecture—Foundations and Applications*, volume 5562 of *LNCS*, pages 18–33. Springer.

Vallecillo, A. (2010). On the Combination of Domain Specific Modeling Languages. In *Proc. of the 6th European Conference on Modelling Foundations and Applications*, volume 6138 of *LNCS*, pages 305–320. Springer.

Varró, D. and Pataricza, A. (2004). Generic and Metatransformations for Model Transformation Engineering. In *Proc. of the 7th International UML Conference Modelling Languages and Applications*, volume 3273 of *LNCS*, pages 290–304. Springer.

Wei, W. (2012). EpsilonLabs: Epsilon Static Analysis. Available at: http://code.google.com/p/epsilonlabs/wiki/EpsilonStaticAnalysis. Last accessed: 30.11.2012.

White, J., Hill, J. H., Gray, J., Tambe, S., Gokhale, A. S., and Schmidt, D. C. (2009). Improving Domain-Specific Language Reuse with Software Product Line Techniques. *IEEE Software*, 26(4):47–53.

Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., and Schwinger, W. (2010). Towards an Expressivity Benchmark for Mappings based on a Systematic Classification of Heterogeneities. In *Proc. of the 1st International Workshop on Model-Driven Interoperability*, pages 32–41. ACM.

Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., and Schwinger, W. (2012). Fact or Fiction—Reuse in Rule-Based Model-to-Model Transformation Languages. In *Proc. of the 2nd International Conference on Model Transformations*, volume 7307 of *LNCS*, pages 280–295. Springer.

Zdun, U. (2010). A DSL Toolkit for Deferring Architectural Decisions in DSL-based Software Design. *Information & Software Technology*, 52(7):733–748.

A.4 Towards Testing the Integration of MOF/UML-based Domain-specific Modeling Languages

The following paper was published as:

B. Hoisl. Towards Testing the Integration of MOF/UML-based Domain-specific Modeling Languages. In *Proceedings of the 8th IASTED International Conference on Advances in Computer Science*, pages 314–323. ACTA Press, 2013 (see [47]).

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 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".

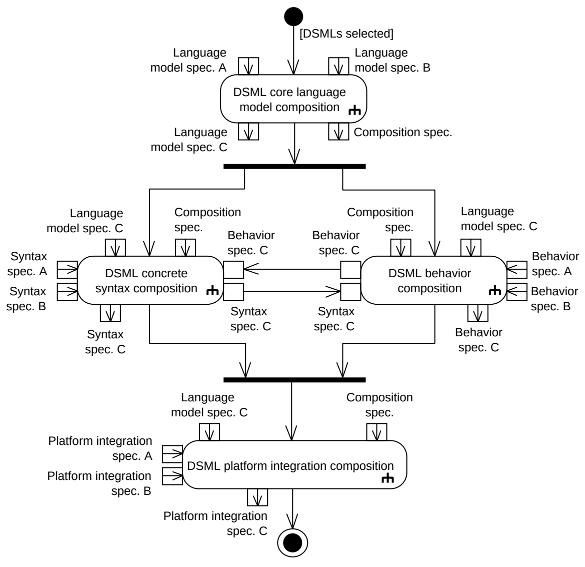


Figure 1: DSML integration process [10].

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

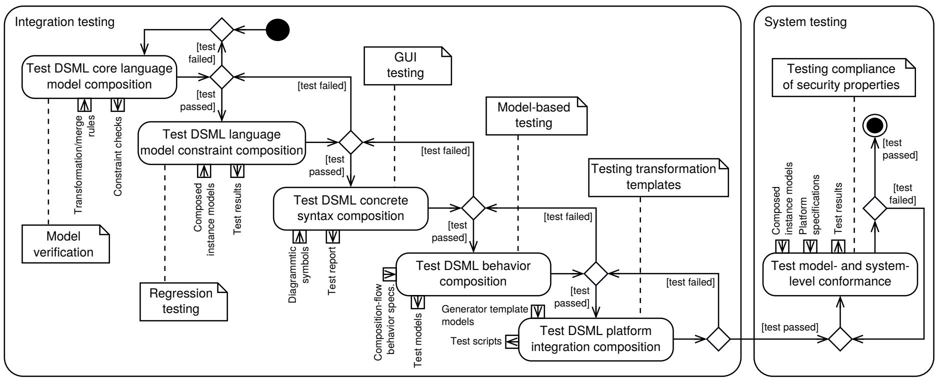

Integration testing
System testing
Test DSML core language model composition
[test failed]
[test passed]
Transformation/merge rules
Constraint checks
Model verification
Test DSML language model constraint composition
Composed instance models
Test results
Regression testing
GUI testing
[test failed]
Test DSML concrete syntax composition
Diagrammatic symbols
Test report
Model-based testing
Test DSML behavior composition
Composition-flow behavior specs.
Test models
Generator template models
Test scripts
Testing transformation templates
Test DSML platform integration composition
[test passed]
Testing compliance of security properties
Composed instance models
Platform specifications
Test results
Test model- and system-level conformance

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]).

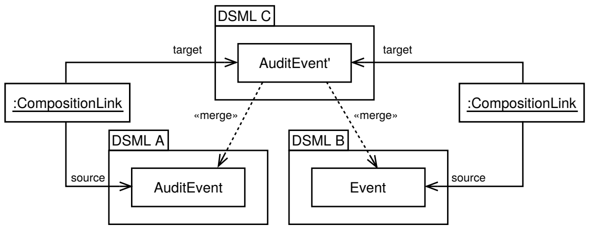


Figure 3: DSML composition via element merges [18].

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.

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

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).

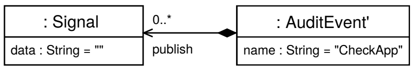


Figure 4: Non-conforming instance model of DSML C.

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

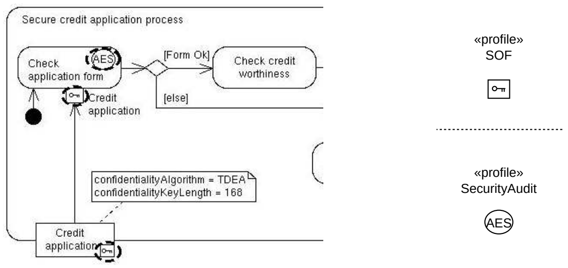


Figure 5: Image-based concrete syntax testing.

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).

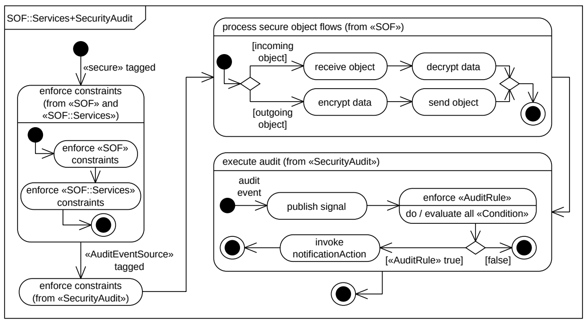

SOF::Services+SecurityAudit
«secure» tagged
enforce constraints (from «SOF» and «SOF::Services»)
enforce «SOF» constraints
enforce «SOF::Services» constraints
«AuditEventSource» tagged
enforce constraints (from «SecurityAudit»)
process secure object flows (from «SOF»)
[incoming object]
receive object
decrypt data
encrypt data
send object
[outgoing object]
execute audit (from «SecurityAudit»)
audit event
publish signal
enforce «AuditRule»
do / evaluate all «Condition»
invoke notificationAction
[«AuditRule» true]
[false]

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

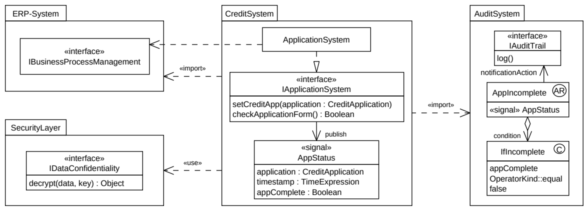


Figure 7: Elements of the composed DSML to be tested.

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

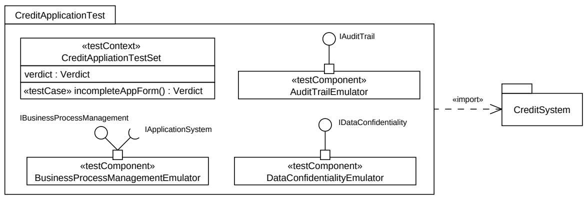


Figure 8: The `CreditApplicationTest` package.

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.

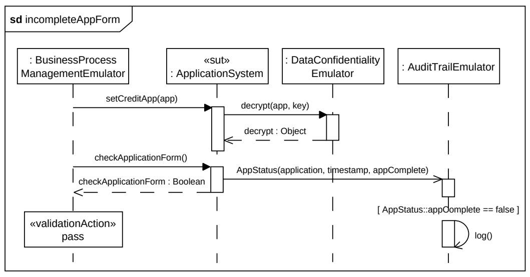

sd incompleteAppForm
: BusinessProcess ManagementEmulator
«sut» : ApplicationSystem
: DataConfidentiality Emulator
: AuditTrailEmulator
setCreditApp(app)
decrypt(app, key)
decrypt : Object
checkApplicationForm()
AppStatus(application, timestamp, appComplete)
checkApplicationForm : Boolean
[AppStatus::appComplete == false]
«validationAction» pass
log()

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.

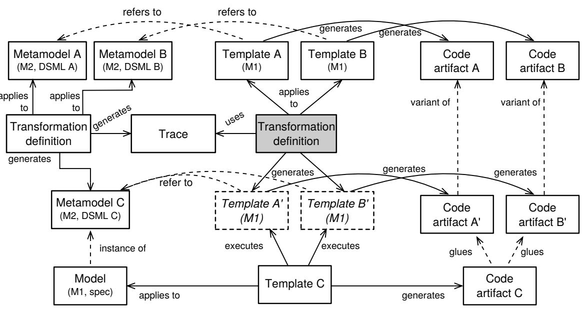


Figure 10: M2T template rewriting for DSML integration.

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.

```
rule renameAuditEvent2AuditEvent'
  transform s : eglIn!ModelElementType
  to t : eglOut!ModelElementType
  extends Type {
    guard : s.type == "AuditEvent"
    t.type = "AuditEvent'";
  }
```

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).

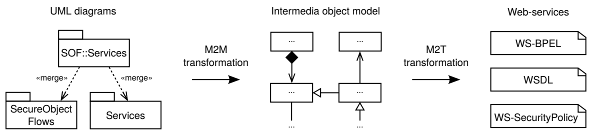
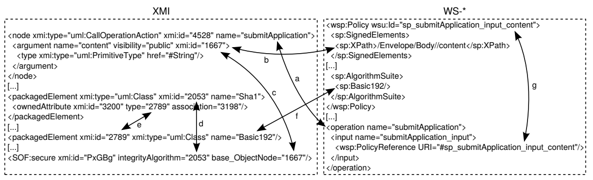


Figure 11: MDD artifacts and transformations.

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).



```
<node xmi:type="uml:CallOperationAction" xmi:id="4528" name="submitApplication">
  <argument name="content" visibility="public" xmi:id="1667">
    <type xmi:type="uml:PrimitiveType" href="#String"/>
  </argument>
</node>
[...]
<packagedElement xmi:type="uml:Class" xmi:id="2053" name="Sha1">
  <ownedAttribute xmi:id="3200" type="2789" association="3198"/>
</packagedElement>
[...]
<packagedElement xmi:id="2789" xmi:type="uml:Class" name="Basic192"/>
[...]
<SOF:secure xmi:id="PxGBg" integrityAlgorithm="2053" base_ObjectNode="1667"/>
```

```
<wsp:Policy wsu:Id="sp_submitApplication_input_content">
  <sp:SignedElements>
    <sp:XPath>/Envelope/Body//content</sp:XPath>
  </sp:SignedElements>
[...]
  <sp:AlgorithmSuite>
    <sp:Basic192/>
  </sp:AlgorithmSuite>
</wsp:Policy>
[...]
<operation name="submitApplication">
  <input name="submitApplication_input">
    <wsp:PolicyReference URI="#sp_submitApplication_input_content"/>
  </input>
</operation>
```

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.

A.5 Requirements-driven Testing of Domain-specific Core Language Models using Scenarios

The following paper was published as:

S. Sobernig, B. Hoisl, and M. Strembeck. Requirements-driven Testing of Domain-specific Core Language Models using Scenarios. In *Proceedings of the 13th International Conference on Quality Software*, pages 163–172. IEEE, 2013 (see [126]).

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]).

978-0-7695-5039-8/13 $26.00 © 2013 IEEE
DOI 10.1109/QSIC.2013.56

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

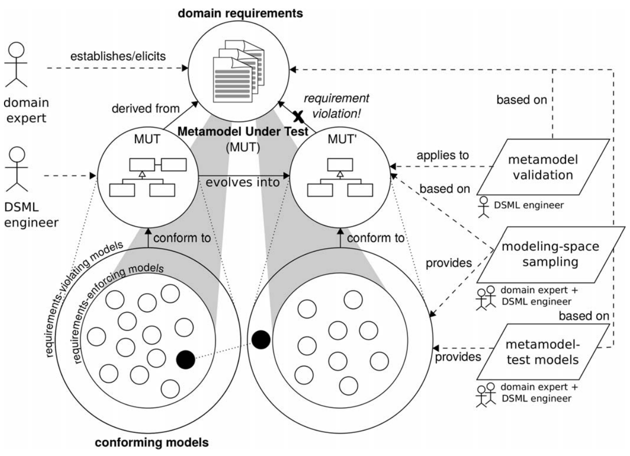


Fig. 1. Metamodel evolution, requirements violation, and metamodel testing.

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).

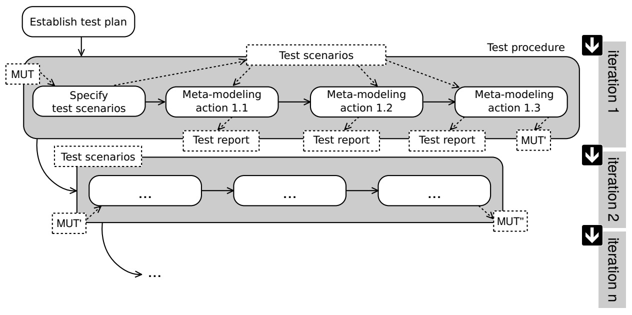
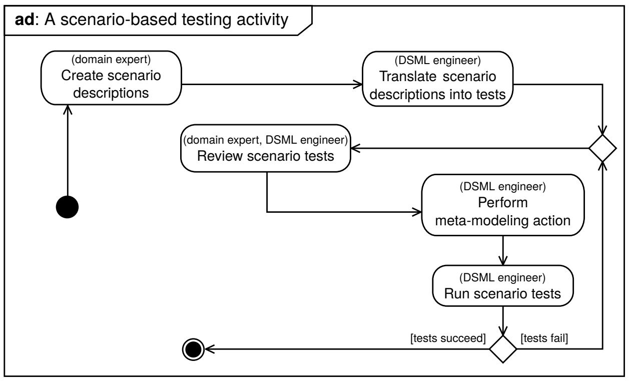


Fig. 2. Iterative language-model development and scenario-based testing.

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).



ad: A scenario-based testing activity
(domain expert) Create scenario descriptions
(DSML engineer) Translate scenario descriptions into tests
(domain expert, DSML engineer) Review scenario tests
(DSML engineer) Perform meta-modeling action
(DSML engineer) Run scenario tests
[tests succeed]
[tests fail]

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.

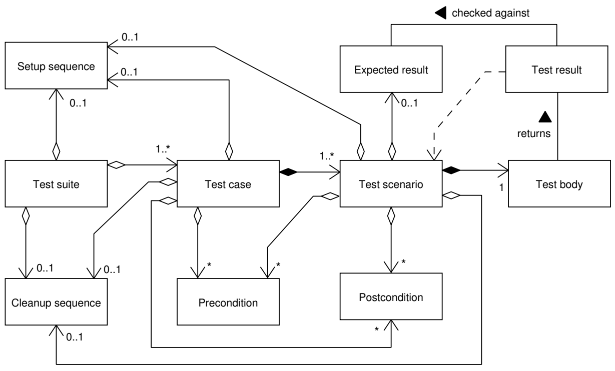


Fig. 4. Scenario-based testing domain model [30].

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]).

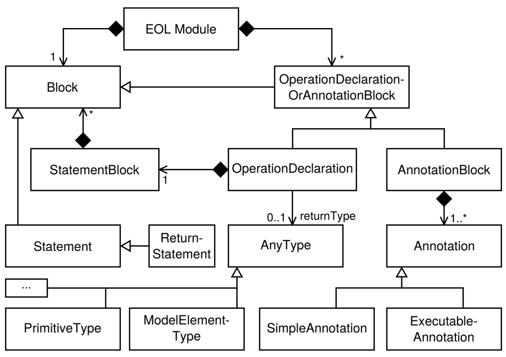


Fig. 5. Excerpt of the building blocks of our Epsilon prototype.

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*.

```
 1 OperationDeclarationOrAnnotationBlock
 2        = AnnotationBlock | OperationDeclaration;
 3 AnnotationBlock
 4        = Annotation { Annotation };
 5 Annotation
 6        = SimpleAnnotation | ExecutableAnnotation;
 7 OperationDeclaration
 8        = ("operation"|"function") [Type] Name
 9          "(" [FormalParameterList] ")" [":" Type] StatementBlock;
10 StatementBlock
11        = "{" Block "}";
12
13 (* Allowing for nested operation and annotation declarations: *)
14 Block
15        = { OperationDeclarationOrAnnotationBlock | Statement };
16
```

[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.

```
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.

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

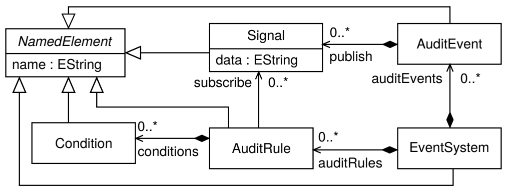


Fig. 6. Auditing event-based systems (DSML `A`).

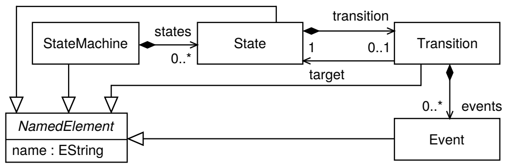


Fig. 7. State/transitional behavioral system (DSML `B`).

3) enforcing new domain-specific language-model constraints. During and between each step, the scenario tests are executed to check the requirements conformance.

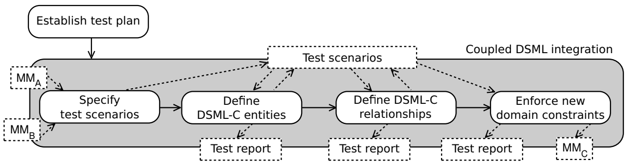


Fig. 8. A step-wise, scenario-assisted DSML integration process using model transformations.

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].

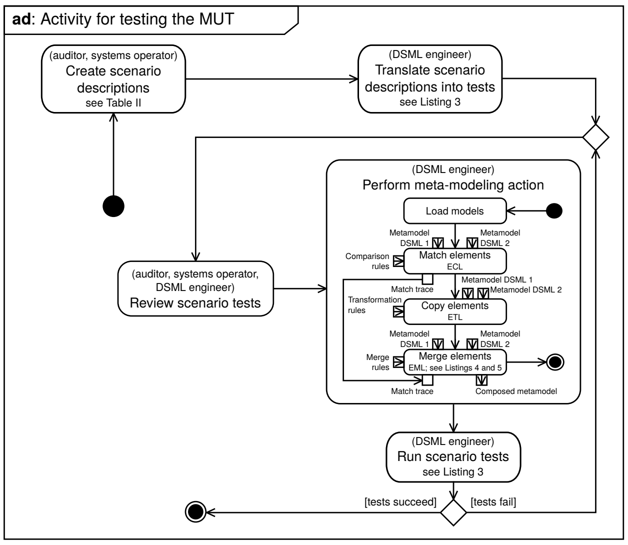


Fig. 9. Scenario-based DSML integration via model transformations.

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.

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 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.

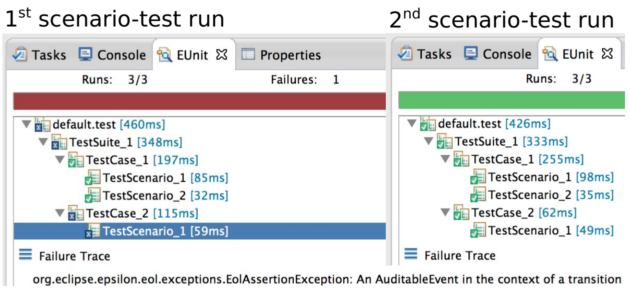
Initial composition specifications: Once having the EUnit scenario-test specifications reviewed collaboratively by the domain experts and the DSML engineer, the DSML engineer specifies the actual metamodel composition. In this application case, this is achieved by devising an Epsilon-based composition 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.

```
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 + r.
            eStructuralFeatures;
8 }
```

Listing 4. EML merge rule for DSML composition.

Patching composition specifications: Based on the test report, the DSML engineer reviews jointly with the domain

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


1st scenario-test run

Tasks Console EUnit Properties

Runs: 3/3 Failures: 1

```
default.test [460ms]
  TestSuite_1 [348ms]
    TestCase_1 [197ms]
      TestScenario_1 [85ms]
      TestScenario_2 [32ms]
    TestCase_2 [115ms]
      TestScenario_1 [59ms]
```

Failure Trace

org.eclipse.epsilon.eol.exceptions.EolAssertionException: An AuditableEvent in the context of a transition

2nd scenario-test run

Tasks Console EUnit

Runs: 3/3

```
default.test [426ms]
  TestSuite_1 [333ms]
    TestCase_1 [255ms]
      TestScenario_1 [98ms]
      TestScenario_2 [35ms]
    TestCase_2 [62ms]
      TestScenario_1 [49ms]
```

Failure Trace

Fig. 10. Scenario-test reports in EUnit.

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.

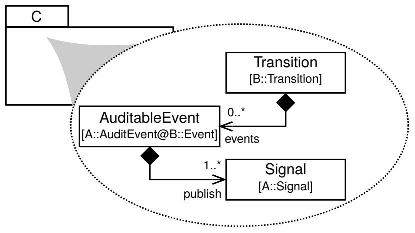


Fig. 11. Relevant excerpt from the final metamodel `C`.

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.

A.6 Natural-language Scenario Descriptions for Testing Core Language Models of Domain-Specific Languages

The following paper was published as:

B. Hoisl, S. Sobernig, and M. Strembeck. Natural-language Scenario Descriptions for Testing Core Language Models of Domain-Specific Languages. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 356–367. SciTePress, 2014 (see [57]).

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

Bernhard Hoisl[1,2], Stefan Sobernig[1], and Mark Strembeck[1,2]

[1]*Institute for Information Systems and New Media, WU Vienna, Austria*

[2]*Secure Business Austria Research (SBA Research), Austria*

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

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

1 INTRODUCTION

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

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

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

In a domain modeling activity, domain requirements are frequently documented using natural-language descriptions (Neill and Laplante, 2003; Diethelm et al., 2005). This is due to the fact that domain requirements emerge and are often elicited during interviews, discussions, and meetings where different stakeholders are involved (Sutcliffe, 2002). Whereas such natural-language requirements descriptions are more accessible to non-technical stake-

Hoisl B., Sobernig S. and Strembeck M. (2014).
Natural-language Scenario Descriptions for Testing Core Language Models of Domain-Specific Languages.
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 356-367
DOI: 10.5220/0004713703560367
Copyright © SCITEPRESS

holders and stakeholders of diverse professional backgrounds (Dwarakanath and Sengupta, 2012), a natural-language description raises important issues when it comes to validating model artifacts such as the core language model against the domain requirements. A corresponding issue is, for example, the ambiguity of natural-language descriptions (Sutcliffe, 2002; Institute of Electrical and Electronics Engineers, 2011). This ambiguity can be caused by contextual details not being made explicit in the requirements narrative and/or by not being representable in a core language model, for instance, due to lacking expressiveness of the respective modeling language.

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

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

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

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

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

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

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

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

2 MOTIVATING EXAMPLE

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

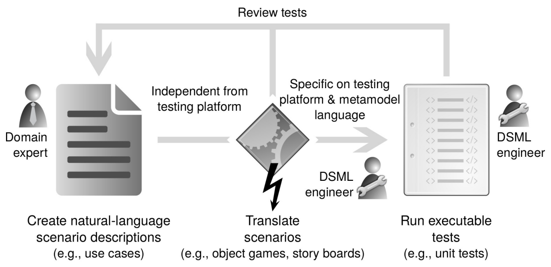

Review tests
Independent from testing platform
Specific on testing platform & metamodel language
Domain expert
DSML engineer
DSML engineer
Create natural-language scenario descriptions (e.g., use cases)
Translate scenarios (e.g., object games, story boards)
Run executable tests (e.g., unit tests)

Figure 1: Translating requirements into executable tests.

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

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

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

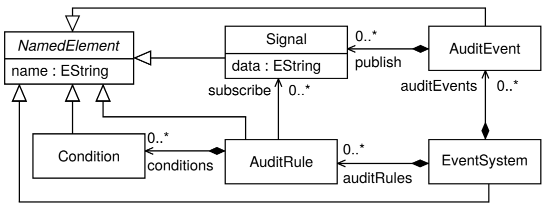


Figure 2: Auditing event-based systems (DSML `A`).

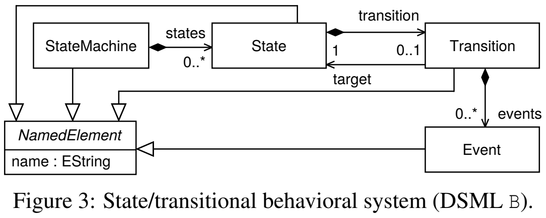


Figure 3: State/transitional behavioral system (DSML `B`).

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

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

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

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

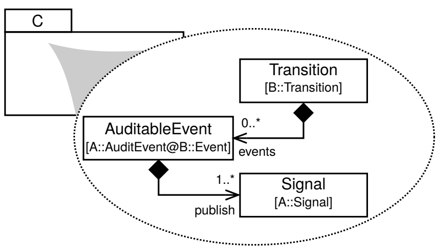


Figure 4: Excerpt from merged DSML C core language model.

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

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

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

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

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

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

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

3 FROM REQUIREMENTS TO EXECUTABLE SCENARIO TESTS

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

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

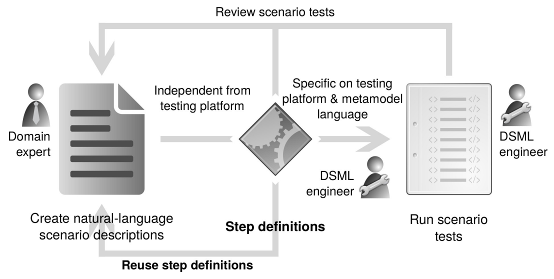

Review scenario tests
Independent from testing platform
Specific on testing platform & metamodel language
Domain expert
DSML engineer
DSML engineer
Create natural-language scenario descriptions
Step definitions
Run scenario tests
Reuse step definitions

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

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

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

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

4 A LANGUAGE FOR REQUIREMENTS-LEVEL SCENARIOS

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

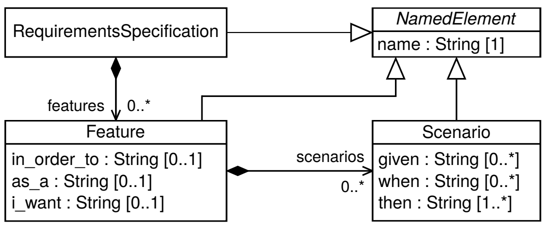


Figure 6: Scenario-based requirements specification language metamodel.

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

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

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

Listing 1: A textual concrete syntax.

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

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

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

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

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

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

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

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

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

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

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

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

5 SCENARIO-BASED TESTING EXEMPLIFIED: A DSML INTEGRATION CASE

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

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

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

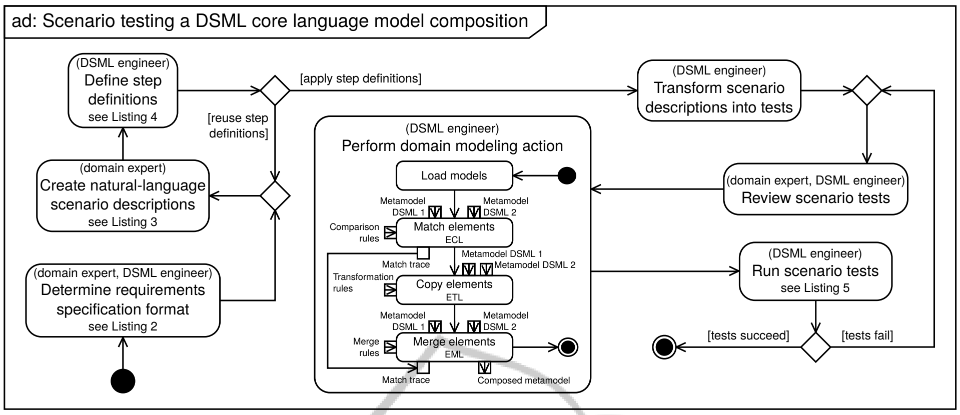


Figure 7: Scenario testing of DSML core language model integration.

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

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

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

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

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

Listing 4: A scenario-transforming step definition.

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

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

Listing 5: Generated EUnit scenario tests.

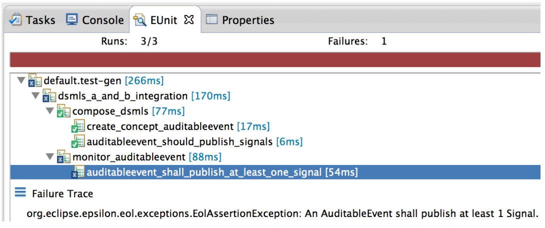
```
--DSMLs A and B integration
@TestSuite
operation dsmls_a_and_b_integration() {
  --Monitor AuditableEvent:
  --In order to ascertain that each triggered
        AuditableEvent can be sensed by the monitoring
        facility
  --As a system auditor and distributed-systems operator
--I want that AuditableEvents shall publish Signals
@TestCase
operation monitor_auditableevent() {
  --AuditableEvent shall publish at least one Signal
  @TestScenario
   --Given that EventSystemStateMachine.AuditableEvent
        has all features of EventSystem.AuditEvent and
        StateMachine.Event
  $pre EventSystemStateMachine!EClass.all->selectOne(ae |
        ae.name = "AuditableEvent").eStructuralFeatures.
       size() = EventSystem!EClass.all->selectOne(ae |
       ae.name = "AuditEvent").eStructuralFeatures.size
       () + StateMachine!EClass.all->selectOne(ae | ae.
       name = "Event").eStructuralFeatures.size()
  operation
       auditableevent_shall_publish_at_least_one_signal
       () {
    --When in metamodel EventSystemStateMachine metaclass
          AuditableEvent references metaclass Signal
    if (EventSystemStateMachine!EClass.all->selectOne(c |
          c.name = "AuditableEvent").eStructuralFeatures
         ->first().eType.name = "Signal") {
      --Then instances of EventSystemStateMachine.
           AuditableEvent shall refer to at least 1
           Signal instance
      assertFalse("An AuditableEvent shall publish at
           least 1 Signal.", EventSystemStateMachine!
           EClass.all->selectOne(c | c.name = "
           AuditableEvent").eStructuralFeatures->first()
           .lowerBound < 1);
    }
  }
 }
 ...
}
```

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

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

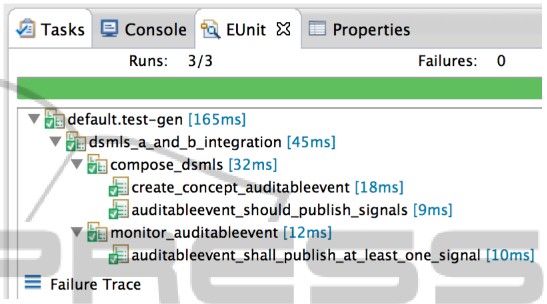
6 DISCUSSION

In Section 2, we argue that natural-language scenarios are useful to capture requirements, but have serious



```
Runs:  3/3                    Failures:  1
default.test-gen [266ms]
  dsmls_a_and_b_integration [170ms]
    compose_dsmls [77ms]
      create_concept_auditableevent [17ms]
      auditableevent_should_publish_signals [6ms]
    monitor_auditableevent [88ms]
      auditableevent_shall_publish_at_least_one_signal [54ms]
Failure Trace
org.eclipse.epsilon.eol.exceptions.EolAssertionException: An AuditableEvent shall publish at least 1 Signal.
```

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



```
Runs:  3/3                    Failures:  0
default.test-gen [165ms]
  dsmls_a_and_b_integration [45ms]
    compose_dsmls [32ms]
      create_concept_auditableevent [18ms]
      auditableevent_should_publish_signals [9ms]
    monitor_auditableevent [12ms]
      auditableevent_shall_publish_at_least_one_signal [10ms]
Failure Trace
```

Figure 9: All scenario tests pass.

drawbacks when it comes to evaluating them against DSML software artifacts. In this section, we discuss how our work contributes to (partly) overcome this shortcomings.

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

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

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

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

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

7 RELATED WORK

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

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

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

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

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

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

8 CONCLUSION

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

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

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

ACKNOWLEDGEMENTS

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.7 Modeling and Enforcing Secure Object Flows in Process-driven SOAs: An Integrated Model-driven Approach

The following paper was published as:

B. Hoisl, S. Sobernig, and M. Strembeck. Modeling and Enforcing Secure Object Flows in Process-driven SOAs: An Integrated Model-driven Approach. *Software & Systems Modeling*, 13(2):513–548, 2014 (see [56]).

Softw Syst Model (2014) 13:513–548
DOI 10.1007/s10270-012-0263-y

THEME SECTION PAPER

Modeling and enforcing secure object flows in process-driven SOAs: an integrated model-driven approach

Bernhard Hoisl · Stefan Sobernig · Mark Strembeck

Received: 7 October 2011 / Revised: 25 April 2012 / Accepted: 13 July 2012 / Published online: 5 October 2012

Abstract In this paper, we present an integrated model-driven approach for the specification and the enforcement of secure object flows in process-driven service-oriented architectures (SOA). In this context, a secure object flow ensures the confidentiality and the integrity of important objects (such as business contracts or electronic patient records) that are passed between different participants in SOA-based business processes. We specify a formal and generic metamodel for secure object flows that can be used to extend arbitrary process modeling languages. To demonstrate our approach, we present a UML extension for secure object flows. Moreover, we describe how platform-independent models are mapped to platform-specific software artifacts via automated model transformations. In addition, we give a detailed description of how we integrated our approach with the Eclipse modeling tools.

Communicated by Dr. Juan M. Vara, Mike Papazoglou and Il-Yeol Song.

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.

B. Hoisl (✉) · S. Sobernig (✉) · M. Strembeck (✉)
New Media Lab, Institute for Information Systems,
Vienna University of Economics and Business (WU Vienna),
Vienna, Austria
e-mail: bernhard.hoisl@wu.ac.at

S. Sobernig
e-mail: stefan.sobernig@wu.ac.at

B. Hoisl · M. Strembeck
Secure Business Austria Research (SBA Research),
Vienna, Austria
e-mail: mark.strembeck@wu.ac.at

Keywords Process modeling · Secure object flows · Security engineering · Service-oriented architecture · Model-driven development · UML · SoaML · Web services

1 Introduction

1.1 Motivation

Business processes define an organization's operational procedures and are performed to reach operational goals. In recent years, service-oriented architectures (SOA; see, e.g., [29,62,64]) are increasingly used in the area of business process management. In this context, a process-driven SOA (see, e.g., [96]) is specifically built to support the definition, the execution, and monitoring of intra-organizational and cross-organizational business processes. The widespread use of service-oriented technologies also led to demands for a thorough integration of security features in the development process of service-oriented systems.

In particular, IT systems must comply with certain laws and regulations, such as the Basel II Accord, the International Financial Reporting Standards (IFRS), or the Sarbanes-Oxley Act (SOX). For example, adequate support for the definition and the enforcement of process-related security policies is one important part of SOX compliance (see, e.g., [7,9,43]). Corresponding compliance requirements also arise from security recommendations and standards, such as the NIST security handbook [45], the NIST recommended security controls [49], or the ISO 27000 standard family [16–18] (formerly ISO 17799). Legally binding agreements, such as business contracts, or company-specific (internal) rules and regulations do also have a direct impact on corresponding information systems (see, e.g., [87]).

Yet, modeling process-related security properties yields different types of problems. First, contemporary modeling languages such as Business Process Model and Notation (BPMN, [52]), Event-driven Process Chain (EPC, [70]), or Unified Modeling Language (UML) activity models [59] do not provide native language constructs to model security features. A second problem is that the language used for process modeling is often different from (or not integrated with) the system modeling language that is used to specify the corresponding software system. This, again, may result in problems because different modeling languages provide different language abstractions that cannot easily be mapped to each other. In particular, such semantic gaps may involve significant efforts when conceptual models from different languages need to be integrated and mapped to a software platform (see, e.g., [4,34,95]). However, a complete and correct mapping of process definitions and related security properties to the corresponding software system is essential in order to assure consistency between the modeling-level specifications on the one hand, and the software system that actually manages corresponding process instances and enforces the respective security properties, on the other hand.

In this paper, we are especially concerned with the confidentiality and the integrity of object flows in process-driven systems. *Confidentiality* ensures that important/classified objects (such as court records, business contracts, or electronic patient records) which are used in a business process can only be read by designated subjects (see, e.g., [8,49]). *Integrity* ensures that important objects are in their original/intended state, and enables the straightforward detection of accidental or malicious changes (see, e.g., [45,50,69]). At the modeling-level, an *object flow* defines that an object is passed from one node in a business process model to another. In a process-driven SOA, the corresponding object flow is then implemented via different messages that are passed between different software services. In the remainder of this paper, we use the term *secure object flow* to refer to an object flow whose confidentiality and/or integrity is ensured via cryptographic mechanisms.

1.2 Approach synopsis

We use model-driven development (MDD) techniques (see, e.g., [77,79,82]) to provide an integrated, tool-supported approach for the definition, for the deployment, and for the execution of secure object flows in process-driven SOAs. In the context of MDD, a computation-independent model (CIM) defines a certain domain (or sub-domain) at a generic level. The CIM is independent of a particular modeling language or technology. A CIM can be used to build a platform-independent model (PIM) of the corresponding domain. While it is independent of any platform, and thereby neutral from an implementation point of view, the PIM is typically

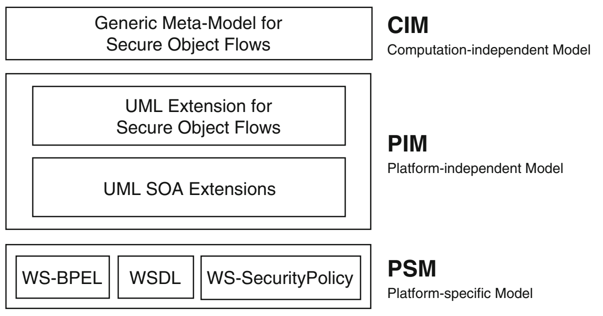

Generic Meta-Model for Secure Object Flows
CIM
Computation-independent Model
UML Extension for Secure Object Flows
UML SOA Extensions
PIM
Platform-independent Model
WS-BPEL
WSDL
WS-SecurityPolicy
PSM
Platform-specific Model

Fig. 1 The secure object flows approach covers the CIM, PIM, and PSM layers

specified in a particular modeling language (such as BPMN or UML) and describes the structure of a system, the elements/results that are produced by a system, or the control and object flow in a system. Finally, a platform-specific model (PSM) describes the realization/implementation of a software system via platform-specific technologies and tools.

Our work on *secure object flows* presented in this paper is an integrated approach which covers the CIM, PIM, and PSM layers (see Fig. 1). At the CIM layer, we provide a generic metamodel for secure object flows that can be used to extend arbitrary process modeling languages. At the PIM layer, we provide a UML extension that allows to model secure object flows via extended activity diagrams. Moreover, we integrate our extension with the SoaML [58] and UML4SOA [38] to enable the definition of secure object flows for process-driven SOAs. At the PSM layer, we generate WS-BPEL [61], WSDL [92], and WS-SecurityPolicy [63] specifications from the PIMs.

To enable the specification and the implementation of secure object flows in process-driven SOAs, we provide an integrated tool support for our approach based on the Eclipse IDE [10]. Figure 2 gives an overview of our tool support on different abstraction levels for the definition and for the implementation of secure object flows. In particular, we apply model transformations [42,80] to automatically generate executable, platform-specific service descriptions that are deployed in a SOA process engine. At the topmost layer, our tool supports the definition of security-enhanced *Business Process* models via UML activity diagrams (see Fig. 2). At the *SOA Models* level, the service-oriented architecture is modeled via component structures, service activities, message types, as well as service and invocation protocols. *Web Service Artifacts* (such as WS-BPEL, WSDL, or WS-SecurityPolicy specifications) are derived from SOA models through automatic model transformations. Finally, these artifacts are deployed for execution in a process-driven *Runtime Environment*.

Our contribution is based on previous publications concerning the modeling of secure object flows [27] and its adop-

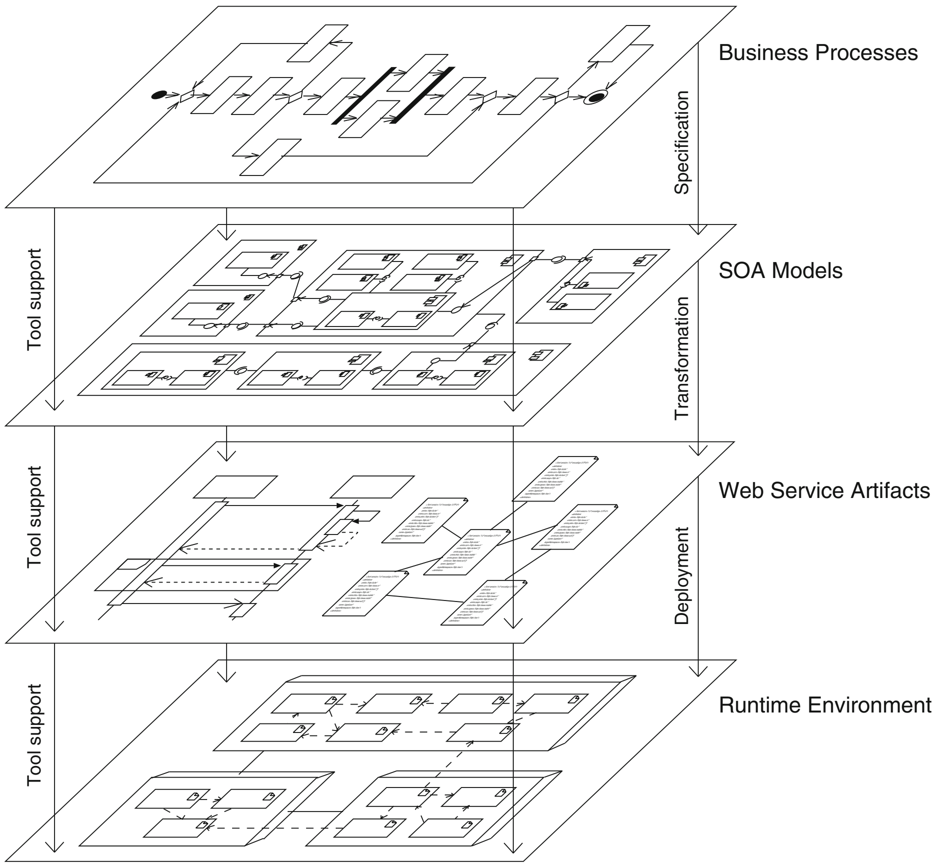


Fig. 2 Integrated tool support for the definition and implementation of secure object flows

tion for SOA modeling [26]. These previous contributions do, however, only discuss specific and limited modeling options at the PIM level. They do neither provide a generic CIM nor integrated PIM models, or tool support. In this paper, we extend our previous contributions via MDD techniques to build an integrated approach (see Figs. 1, 2) for the specification and for the enforcement of secure object flows in process-driven SOAs. We provide a thorough description of the capabilities of our approach to model secure object flows in SOAs, both at a generic and at the modeling language level. In addition, we present our tool support (including automated model transformations) for the specification and for the deployment of secure object flows.

The remainder of this paper is structured as follows. Section 2 discusses the general characteristics of secure object flows and Sect. 3 defines a formal and generic CIM. Next, we describe our PIM which consists of a generic UML extension for secure object flows (see Sect. 4) as well as an integration of secure object flows with SOA-based modeling primitives (see Sect. 5). Subsequently, Sect. 6 presents tool support for our approach and describes automated model transformations that produce PSM artifacts from corresponding PIMs. Finally, Sect. 7 discusses related work and Sect. 8 concludes the paper.

For the sake of readability, we moved the formal constraints that define the semantics of our UML extension to Appendices A and B.

2 Characteristics of secure object flows

Process models typically have (implicit or explicit) token semantics, and object tokens are passed along object flow edges. Thus, to ensure the consistency of the corresponding process models, it is especially important to thoroughly specify the semantics of secure object flows with respect to control nodes (such as fork, join, decision, and merge nodes).

In general, a *secure object flow* consists of one or more arcs in a business process model that transport important, security-sensitive workflow objects (e.g., electronic patient records or business contracts) between two *secure nodes* of the respective process model. In particular, we have to ensure

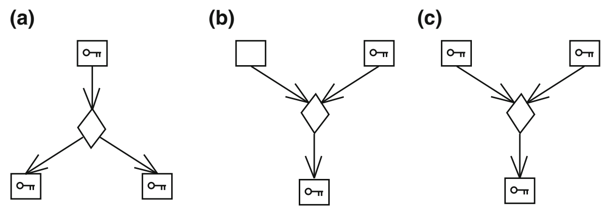


Fig. 3 Secure object flows with decision and merge nodes

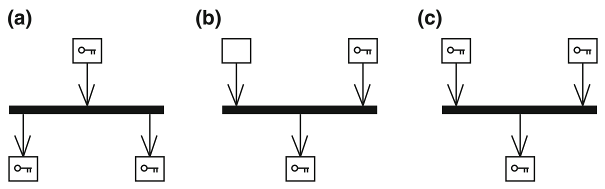


Fig. 4 Secure object flows with fork and join nodes

that the security attributes determined by the source node of a secure object flow (such as the confidentiality algorithm used to encrypt the corresponding objects) are understood by the respective target node(s). In this context, control nodes (fork, join, decision, merge) are of special importance because they influence the semantics of secure object flows. Below, we give an overview of the impact that different configurations of control nodes have on the corresponding secure object nodes. Subsequently, Sect. 3 provides generic definitions that formally define the semantics of secure object nodes at the CIM level.

Figure 3 shows examples for the different configuration options of secure object flows that include decision or merge nodes.[1] In the subsequent figures, a rectangle including a key symbol represents a secure object node, while a blank rectangle represents an ordinary object node.

Figure 3a shows a configuration in which a decision node has an incoming secure object flow and presents the corresponding object tokens to multiple outgoing edges. As the source of the incoming object flow is a secure node both target nodes must also be secured. Otherwise, a secure object flow could have a secure node as its source and an ordinary object node as its target, which would result in an inconsistency because ordinary object nodes cannot ensure the confidentiality or the integrity of object tokens. Furthermore, target nodes of a secure object flow must support the same security properties as the respective source node. This constraint ensures that (1) security properties cannot be lost when traversing a decision node and that (2) the target node(s) are able to check and to ensure the corresponding security properties.

Figure 3b shows a configuration where a merge node brings together different flows, one of which is a secure object flow. For such a configuration, we define that if a merge node receives at least one secure object flow, the target node of this merge node must also be a secure node. This constraint guarantees that each secure object token passing a merge node can be checked and processed by the corresponding target node.

[1] For the sake of simplicity, Figs. 3 and 4 show only two incoming/outgoing flows for the respective control nodes. However, the corresponding discussion equally applies to an arbitrary number of incoming/outgoing edges, of course.

Figure 3c shows a configuration where a merge node brings together different secure object flows. In this case, the target must also be a secure node. Furthermore, we define that all source nodes must provide compatible security properties, i.e., the nodes must support the same confidentiality and/or integrity algorithms. In addition, the target node must support all security properties of the respective source nodes. Otherwise, incompatibilities could emerge if the security properties supported by the source nodes are different from the security properties supported by the target node.

Figure 4 shows examples for the different configuration options of secure object flows that include fork or join nodes. Figure 4a shows a configuration where a fork node splits a secure object flow into multiple concurrent flows. Because the tokens arriving at a fork node are duplicated, all target nodes must be secure nodes. Furthermore, the target nodes must support the same security properties as the corresponding source node. This constraint ensures (1) that security properties cannot be lost when traversing a fork node and (2) that the target node(s) are able to check and ensure the corresponding security properties.

Figure 4b shows a configuration where a join node synchronizes multiple object flows, one of which is a secure object flow. We define that if a join node receives at least one secure object flow, then the target node of this join node must also be a secure node. This constraint guarantees that each secure object token passing a join node can be checked and processed by the corresponding target node.

Figure 4c shows a configuration where a join node synchronizes multiple secure object flows. In such a situation, the target must also be a secure node. Furthermore, all source nodes and the target node must support compatible security properties. Otherwise, inconsistencies could emerge if the security properties supported by the source nodes are different from the security properties supported by the target node.

The examples from Figs. 3 and 4 only include a single control node, respectively. However, in principle, the path from one secure object node to another secure object node may include an arbitrary number of control nodes. Figure 5 shows examples of such configurations. In case a path between two secure object nodes includes two or more intermediate control nodes, we also have to ensure that the source and the

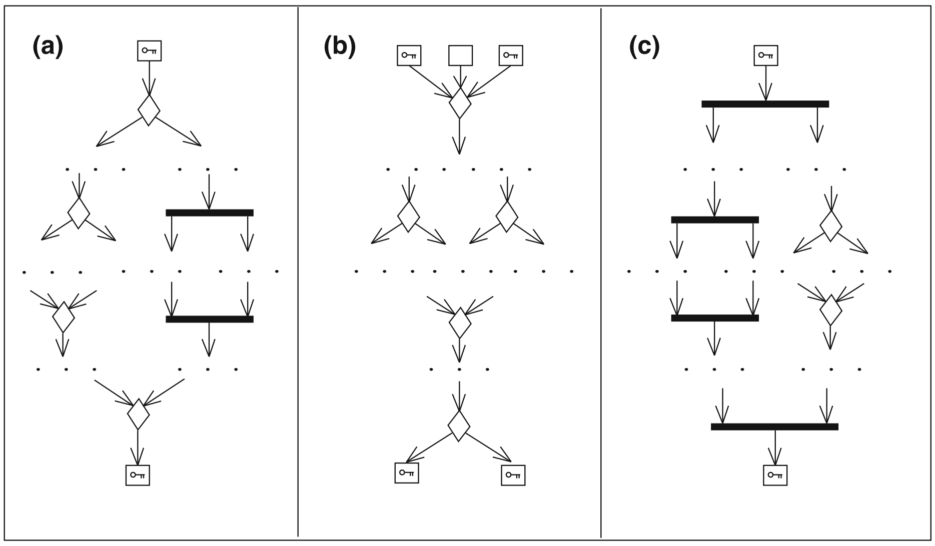


Fig. 5 Secure object flows with an arbitrary number of intermediate control nodes

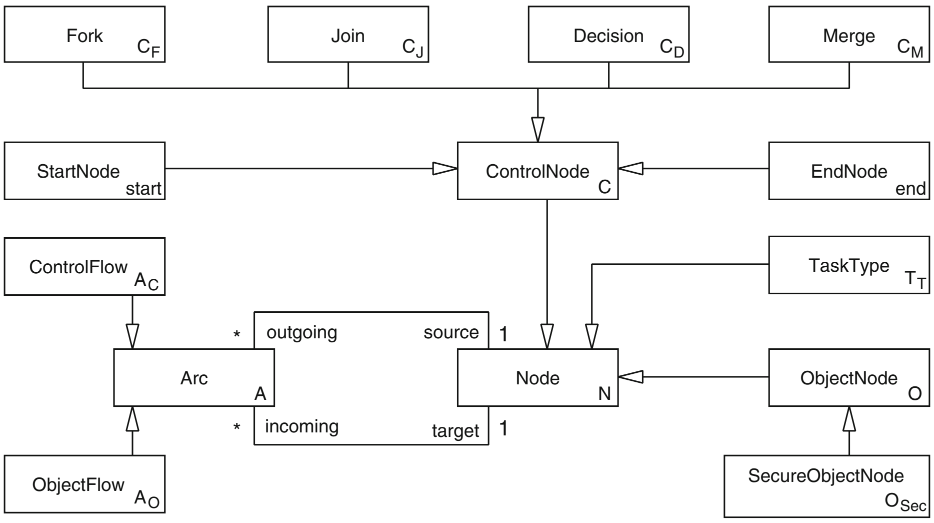


Fig. 6 Conceptual overview: main elements of Business Activity process flows (see also [85])

target nodes of the respective path provide compatible security features.

After discussing the constraints for secure object flows on the above examples, Sect. 3 now provides a formal and generic metamodel (CIM) for secure object flows.

3 A formal and generic metamodel for secure object flows

Figure 6 shows the basic elements of Business Activity process flows [85] and the main relations between these elements as a MOF-compliant structural diagram [53]. While this graphical model only gives an overview, we now provide a formal specification of the process flow model. The formal definitions below complement the definitions from [85].

Definition 1 *(Business Activity Process Flow Model)* A Process Flow Model $PFM = (N, A, S, Y)$ where $N = T_T \cup C_F \cup C_J \cup C_D \cup C_M \cup O \cup O_{Sec} \cup \{start, end\}$ and $A = A_C \cup A_O$ refer to pairwise disjoint sets of the metamodel, $A \subseteq N \times N$ refers to a set of arcs that connect nodes, $S = S_C \cup S_I$ refers to pairwise disjoint sets of security attributes, and $Y = in \cup out \cup source \cup target \cup ofpath \cup successors \cup predecessors \cup ca \cup ia$ refers to mappings that establish relationships such that

- an element of N is called *node* and an element of A is called *arc*;
- an element of A_C is called *control flow* and an element of A_O is called *object flow*;
- an element of S_C is called *confidentiality algorithm* and an element of S_I is called *integrity algorithm*;
- an element of T_T is called *task type*;
- an element of O is called *object node* and an element of O_{Sec} is called *secure object node* with $O_{Sec} \subseteq O$;
- an element of $C = C_F \cup C_J \cup C_D \cup C_M$ is called *control node*. An element of C_F is called *fork*, an element of C_J *join*, an element of C_D *decision*, and an element of C_M *merge*;
- $start$ is called *start node* and end is called *end node*;
- all nodes $n \in N$ are on a path from $start$ to end.

Below, we iteratively define the partial mappings of the Business Activity Process Flow Model and provide corresponding formalizations ($\mathcal{P}$ refers to the power set):

1. The mapping $in : N \mapsto \mathcal{P}(A)$ is called **incoming arc**. For $in(n) = A_{in}$ with $n \in N$ and $A_{in} \subseteq A$ we call each $a \in A_{in}$ an incoming arc of node n.
2. The mapping $out : N \mapsto \mathcal{P}(A)$ is called **outgoing arc**. For $out(n) = A_{out}$ with $n \in N$ and $A_{out} \subseteq A$ we call each $a \in A_{out}$ an outgoing arc of node n.
3. The mapping $source : A \mapsto N$ is called **source node**. For $source(a) = n$ with $a \in A$ and $n \in N$ we call n the source node of arc a.
4. The mapping $target : A \mapsto N$ is called **target node**. For $target(a) = n$ with $a \in A$ and $n \in N$ we call n the target node of arc a.
5. The mapping $ofpath : (O \times O) \mapsto \mathcal{P}(A_O)$ is called **object flow path**. For $ofpath(o_s, o_t) = A_{path}$ with $o_s, o_t \in O$ and $A_{path} \subseteq A_O$ we call o_s source node, o_t target node, and each $a \in A_{path}$ is an arc on the path from o_s to o_t. Thus, an object flow path between two object nodes o_s and o_t must only include arcs or control nodes, it must not include intermediary tasks or (other) object nodes. Therefore, the following consistency requirements must hold for each object flow path:
   - An object flow path connects the source node o_s and the target node o_t via an arbitrary number of arcs and intermediary control nodes, therefore: $\forall a \in A_{path} : source(a) = o_s \vee source(a) \in C$ and $\forall a \in A_{path} : target(a) = o_t \vee target(a) \in C$.
   - The first arc a_{first} in an object flow path is an outgoing arc of the source node o_s, therefore: $\exists a_{first} \in A_{path} : source(a_{first}) = o_s$.
   - Each object flow path includes exactly one outgoing arc of the source node o_s, therefore: $\forall a_1, a_2 \in A_{path} : a_1 \in out(o_s) \wedge a_2 \in out(o_s) \Rightarrow a_1 = a_2$.
   - In an object flow path, the source node o_s has no incoming arcs, therefore: $\forall a \in A_{path} : a \notin in(o_s)$.
   - The last arc a_{last} in an object flow path is an incoming arc of the target node o_t, therefore: $\exists a_{last} \in A_{path} : target(a_{last}) = o_t$.
   - Each object flow path includes exactly one incoming arc of the target node o_t, therefore: $\forall a_1, a_2 \in A_{path} : a_1 \in in(o_t) \wedge a_2 \in in(o_t) \Rightarrow a_1 = a_2$.
   - In an object flow path, the target node o_t has no outgoing arcs, therefore: $\forall a \in A_{path} : a \notin out(o_t)$.
6. The mapping $successors : O \mapsto \mathcal{P}(O)$ is called **succeeding object nodes**. For $successors(o_s) = O_{succ}$ with $o_s \in O$ and $O_{succ} \subseteq O$ we call o_s source node and each $o_t \in O_{succ}$ a direct successor of o_s. In particular, O_{succ} is the set of object nodes for which a path exists between o_s and each $o_t \in O_{succ}$. Formally: $\forall o_s \in O, o_t \in successors(o_s) : ofpath(o_s, o_t) \neq \emptyset$.
7. The mapping $predecessors : O \mapsto \mathcal{P}(O)$ is called **preceding object nodes**. For $predecessors(o_t) = O_{pre}$ with $o_t \in O$ and $O_{pre} \subseteq O$ we call o_t target node and each $o_s \in O_{pre}$ a direct predecessor of o_t. In particular, O_{pre} is the set of object nodes for which a path exists between each $o_s \in O_{pre}$ and o_t. Formally: $\forall o_t \in O, o_s \in predecessors(o_t) : ofpath(o_s, o_t) \neq \emptyset$.
8. The mapping $ca : O_{Sec} \mapsto S_C$ is called **confidentiality algorithm**. For $ca(o_s) = s_c$ with $o_s \in O_{Sec}$ and $s_c \in S_C$ we call s_c the confidentiality algorithm used by o_s.
9. The mapping $ia : O_{Sec} \mapsto S_I$ is called **integrity algorithm**. For $ia(o_s) = s_i$ with $o_s \in O_{Sec}$ and $s_i \in S_I$ we call s_i the integrity algorithm used by o_s.

A *direct object flow* consists of a single arc that directly connects two object nodes without intermediary control nodes, i.e., $A_{DO} = \{a \in A_O | source(a) \in O \wedge target(a) \in O\}$. A *transitive object flow* consists of two or more arcs which connect two object nodes via an object flow path, i.e., $A_{TO} = \{a \in A_O | a \in ofpath(o_s, o_t)\}$ with $o_s, o_t \in O$. If the source of a (direct or transitive) object flow is a secure object node, i.e., $a \in A_O \wedge source(a) \in O_{Sec}$, we call this object flow a *secure object flow*.

Definition 2 Let $PFM = (N, A, S, Y)$ be a Business Activity Process Flow Model. PFM is said to be correct if the following requirements hold:

1. Each secure object node ensures either or both confidentiality and integrity: $\forall o_s \in O_{Sec} : ca(o_s) \cup ia(o_s) \neq \emptyset$.
2. The successor of a secure object node must also be a secure object node: $\forall o_s \in O_{Sec}, o_t \in successors(o_s) : o_t \in O_{Sec}$.
3. The successor of a secure object node must support the same confidentiality algorithm as the respective source

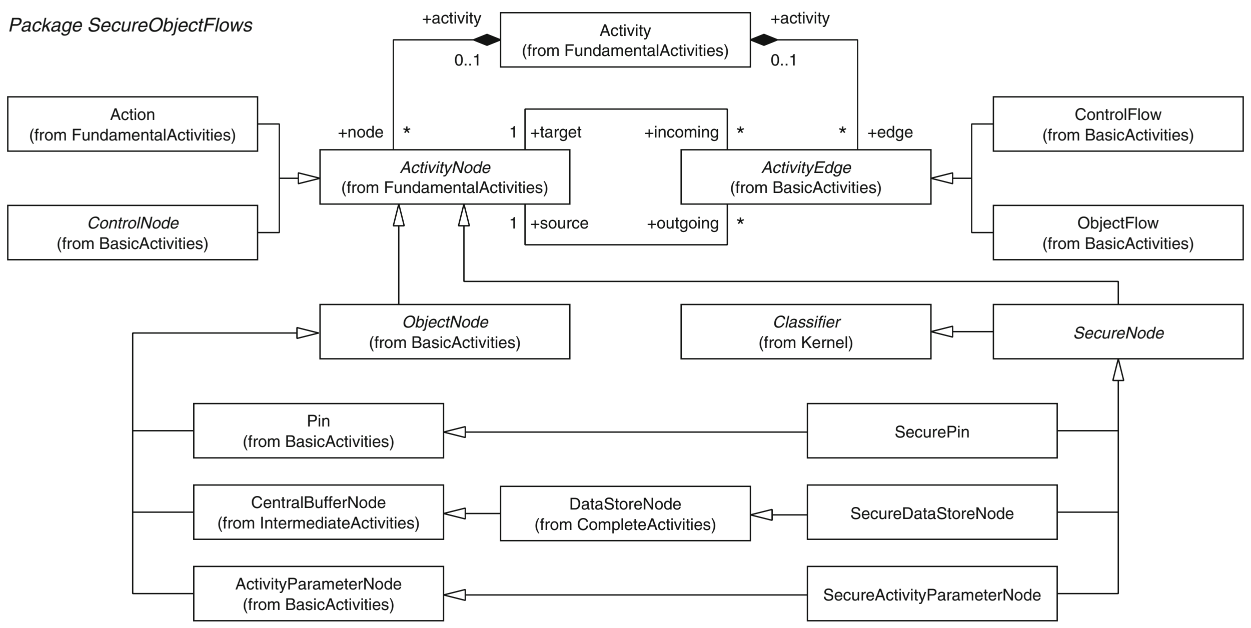

Package SecureObjectFlows
Activity (from FundamentalActivities)
+activity
0..1
+activity
0..1
Action (from FundamentalActivities)
ControlNode (from BasicActivities)
+node
*
1
+target
+incoming
*
*
+edge
ActivityNode (from FundamentalActivities)
ActivityEdge (from BasicActivities)
1
+source
+outgoing
*
ControlFlow (from BasicActivities)
ObjectFlow (from BasicActivities)
ObjectNode (from BasicActivities)
Classifier (from Kernel)
SecureNode
Pin (from BasicActivities)
SecurePin
CentralBufferNode (from IntermediateActivities)
DataStoreNode (from CompleteActivities)
SecureDataStoreNode
ActivityParameterNode (from BasicActivities)
SecureActivityParameterNode

Fig. 7 UML metamodel extension for secure object flows

node: $\forall o_s \in O_{Sec}, o_t \in successors(o_s) : ca(o_s) = ca(o_t)$.

4. The successor of a secure object node must support the same integrity algorithm as the respective source node: $\forall o_s \in O_{Sec}, o_t \in successors(o_s) : ia(o_s) = ia(o_t)$.
5. The security attributes of two secure object nodes $o_{s1}, o_{s2} \in O_{Sec}$ may influence each other, even if they are not connected via a direct or via a transitive object flow. In particular, this is the case if o_{s1} and o_{s2} are predecessors of a common target object node $o_t \in O_{Sec}$. In other words, if a secure object node o_t has two or more predecessors that are also secure object nodes, then each predecessor must support the same confidentiality algorithm as the respective target node: $\forall o_t \in O_{Sec}, o_s \in predecessors(o_t) : o_s \in O_{Sec} \Rightarrow ca(o_t) = ca(o_s)$.
6. If a secure object node o_t has two or more predecessors that are also secure object nodes, then each predecessor must support the same integrity algorithm as the respective target node: $\forall o_t \in O_{Sec}, o_s \in predecessors(o_t) : o_s \in O_{Sec} \Rightarrow ia(o_t) = ia(o_s)$.

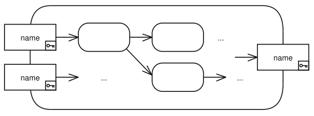
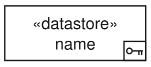
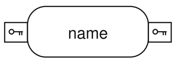
4 UML extension for secure object flows

To provide modeling support for confidentiality and integrity properties of object flows at the PIM level, we define a new package called *SecureObjectFlows* as an extension to the UML metamodel (see Fig. 7). In particular, we introduce `SecureNode`, `SecurePin`, `SecureDataStoreNode`, and `SecureActivityParameterNode` as new modeling elements. A secure object flow is defined as an object flow between two of the above mentioned secure object nodes. The `SecureNode` element is defined as an abstract node, and the `SecurePin`, `SecureDataStoreNode`, and `SecureActivityParameterNode` represent specialized secure nodes. In particular, these three node types inherit the properties from their corresponding parent object nodes as well as the security related properties from `SecureNode` (see Fig. 7).

Below, we specify the attributes of the `SecureNode` elements defined via the metamodel extension. In addition, we use the Object Constraint Language (OCL, [57]) to formally specify the semantics of the SecureObjectFlows package. For the sake of readability, we decided to move the associated OCL constraints to Appendix 8. However, these OCL constraints are a significant part of our UML extension, because they formally define the semantics of the new modeling elements. Therefore, each UML model that uses the SecureObjectFlows package must conform to these OCL constraints.[2]

[2] For some of our OCL constraints, Appendix A provides two optional OCL statements expressing identical constraints, where each of these optional constraints complies with a different version of the OCL standard. OCL Constraints 4a and 6a comply with OCL version 2.2 [56], while OCL Constraints 4b and 6b use new language constructs from the OCL 2.3.1 standard [57]. The changes affect only the `allSuccessors()` and `allPredecessors()` definitions which are interchangeable.

Table 1 Notation of elements for modeling secure objects

Node type	Notation	Explanation
`SecurePin` (attached to an action)		A `SecurePin` attached to an action is shown as a UML `Pin` element that includes a key symbol
`SecureDataStoreNode`		A `SecureDataStoreNode` is shown as a UML `DataStoreNode` element with a key symbol in the lower right corner surrounded by a small rectangle
`SecureActivityParameterNode`		A `SecureActivityParameter Node` is shown as a UML `ActivityParameterNode` element with a key symbol in the lower right corner surrounded by a small rectangle

- *confidentialityAlgorithm : Classifier [0..1]*
  - References a classifier that provides methods to ensure confidentiality properties of the object tokens that are sent or received by a `SecureNode`, e.g., a class implementing Data Encryption Standard (DES, [46]) or Advanced Encryption Standard (AES, [47]) functionalities.
- *confidentialityEnsured : Boolean [0..1]*
  - This attribute is derived from the attribute *confidentialityAlgorithm*. It evaluates to “true” if a `SecureNode` supports confidentiality-related security properties (see OCL Constraint 1 in Appendix A).
- *integrityAlgorithm : Classifier [0..1]*
  - References a classifier that provides methods to ensure integrity properties of the object tokens that are sent or received by a `SecureNode`, e.g., a class implementing SHA-1 or SHA-384 (Secure Hash Algorithm, [48]) functionalities.
- *integrityEnsured : Boolean [0..1]*
  - This attribute is derived from the attribute *integrityAlgorithm*. It evaluates to “true” if a `SecureNode` supports integrity-related security properties (see OCL Constraint 2).

With respect to the attributes defined above, we specify that a secure object node supports either or both confidentiality and integrity properties (see OCL Constraint 3). Table 1 shows the graphical elements for `SecureNodes`. Table 2 gives an overview of how each of the generic (CIM) definitions from Sect. 3 is mapped to our UML extension (PIM) for secure object flows.

4.1 Example processes with secure object flows

Below, we show two examples that model secure object flows. In Sect. 4.1.1, we present a radiological image reading process that is conducted in a hospital. In Sect. 4.1.2, we show a simple credit application process in a bank.

4.1.1 Radiological examination process

Figure 8 shows a radiological examination process, modeled via a UML activity diagram that uses elements of the SecureObjectFlows package. The process starts with a *Radiological examination* action that produces images which are read in a next step. The corresponding `SecurePins` enforce the security properties defined in Table 3 for all *Image* object tokens traveling between the *Radiological examination* and *Image reading* actions. The attributes are derived from the `SecureNode` classifier defined via the SecureObjectFlows package. Note that the different attributes are properties of the corresponding `SecureNodes` and exist independent of their visualization in a model.[3] If the images are of sufficient quality, the activity continues with two concurrent flows: the images are annotated and the patient record is fetched. Both actions produce output tokens of type *Image* and *Patient*

[3] For example, an alternative visualization of SecureObjectFlows attributes would use comments/constraints attached to secure object nodes directly in an activity diagram.

Table 2 Consistency of the generic metamodel and the SecureObjectFlows UML extension

Generic definitions	Covered through
Definition 1.1: $in : N \mapsto \mathcal{P}(A)$	Implicitly defined via our metamodel extension and the specification of UML activity models (see Fig. 7; [59])
Definition 1.2: $out : N \mapsto \mathcal{P}(A)$	Implicitly defined via our metamodel extension and the specification of UML activity models (see Fig. 7; [59])
Definition 1.3: $source : A \mapsto N$	Implicitly defined via our metamodel extension and the specification of UML activity models (see Fig. 7; [59])
Definition 1.4: $target : A \mapsto N$	Implicitly defined via our metamodel extension and the specification of UML activity models (see Fig. 7; [59])
Definition 1.5: $ofpath : (O \times O) \mapsto \mathcal{P}(A_O)$	Implicitly defined via our metamodel extension, the specification of UML activity models (see Fig. 7; [59]), as well as the usage of the OCL quantifiers `collect` (see [56,57]) or `closure` (see [57])
Definition 1.6: $successors : O \mapsto \mathcal{P}(O)$	Implicitly defined via our metamodel extension, the specification of UML activity models (see Fig. 7; [59]), and helper OCL operations (see, e.g., `allSuccessors` in Constraints 4a and 4b in Appendix A)
Definition 1.7: $predecessors : O \mapsto \mathcal{P}(O)$	Implicitly defined via our metamodel extension, the specification of UML activity models (see Fig. 7; [59]), and helper OCL operations (see, e.g., `allPredecessors` in OCL Constraints 6a and 6b in Appendix A)
Definition 1.8: $ca : O_{Sec} \mapsto S_C$	Metamodel extension `SecureNode` and corresponding sub-types (see Fig. 7)
Definition 1.9: $ia : O_{Sec} \mapsto S_I$	Metamodel extension `SecureNode` and corresponding sub-types (see Fig. 7)
Definition 2.1: $\forall o_s \in O_{Sec} : ca(o_s) \cup ia(o_s) \neq \emptyset$	OCL Constraints 1, 2, and 3 in Appendix A
Definition 2.2: $\forall o_s \in O_{Sec} : \forall o_t \in successors(o_s) : o_t \in O_{Sec}$	OCL Constraints 4a and 4b in Appendix A
Definition 2.3: $\forall o_s \in O_{Sec} : \forall o_t \in successors(o_s) : ca(o_s) = ca(o_t)$	OCL Constraint 5 in Appendix A
Definition 2.4: $\forall o_s \in O_{Sec} : \forall o_t \in successors(o_s) : ia(o_s) = ia(o_t)$	OCL Constraint 5 in Appendix A
Definition 2.5: $\forall o_t \in O_{Sec} : \forall o_s \in predecessors(o_t) : o_s \in O_{Sec} \Rightarrow ca(o_t) = ca(o_s)$	OCL Constraints 6a and 6b in Appendix A
Definition 2.6: $\forall o_t \in O_{Sec} : \forall o_s \in predecessors(o_t) : o_s \in O_{Sec} \Rightarrow ia(o_t) = ia(o_s)$	OCL Constraints 6a and 6b in Appendix A

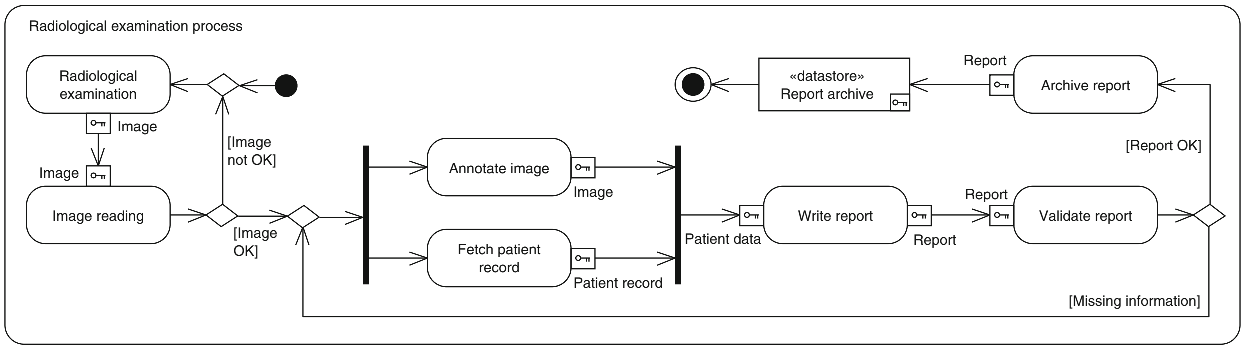


Fig. 8 Radiological examination process with secure object flows

record, respectively. Note that *Image* and *Patient record* are specialized classifiers of type *Patient data* (see Fig. 9) and, therefore, support the same SecureObjectFlows attributes as defined in Table 3.

After the report has been written, it is validated by a senior physician. If the report is incomplete, the corresponding actions have to be repeated. Otherwise, the report is archived via a `SecureDataStoreNode` (see Fig. 8).

Table 3 SecureObjectFlows attributes for the radiological examination process

Object type	SecureObjectFlows attributes
Patient data	`confidentialityAlgorithm = Aes256; integrityAlgorithm = Sha512`
Report	`confidentialityAlgorithm = Aes256; integrityAlgorithm = Sha512`

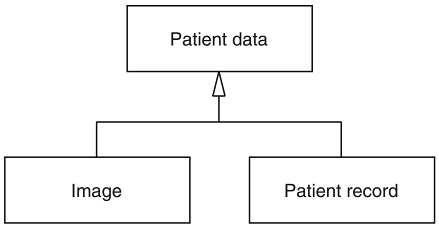


Fig. 9 Patient data object types

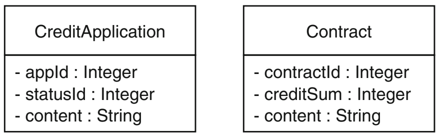


Fig. 11 Data items of the credit application process

4.1.2 *Credit application process*

Figure 10 shows a credit application process that uses the elements of the SecureObjectFlows package. The model contains swimlanes representing a customer and a bank clerk. Figure 11 shows a UML class diagram that describes the data items used in the credit application process. In addition, Table 4 documents the attribute-value pairs of the corresponding secure object nodes. The activity starts when the `SecureActivityParameterNode` named *Credit application* passes an object token to the *Check application form* action (see Fig. 10). In this example, the *Credit application* `SecureActivityParameterNode` is ensuring the data confidentiality and the data integrity of the corresponding object tokens via the AES-192 and SHA-1 algorithms, respectively (see Table 4). Remember that the formal semantics of the respective modeling elements are defined via the OCL constraints from Appendix A.

After completing the *Check application form* action, the creditworthiness of the applicant is checked. If the check fails, the credit application is rejected and the process ends (see Fig. 10). If the creditworthiness check is passed, however, the bank offers a contract to the respective customer. If the credit sum does not exceed the amount of 5000, the applicant is offered a standard contract. Otherwise, a customized contract is negotiated with the client. Because the contents of this contract are confidential, both output pins of the *Standard contract* and *Negotiate contract* actions as well as the input pin of the subsequent action *Approve contract* support confidentiality properties (see also Table 4). In Sects. 5 and 6, we use the credit application example to describe the modeling of (secure) process-driven SOAs and to illustrate our tool support for secure object flows.

5 Modeling process-driven SOAs with UML

In the context of (Web) service modeling, identifying and categorizing services that are based on business process artifacts is an important modeling task. It provides the input for

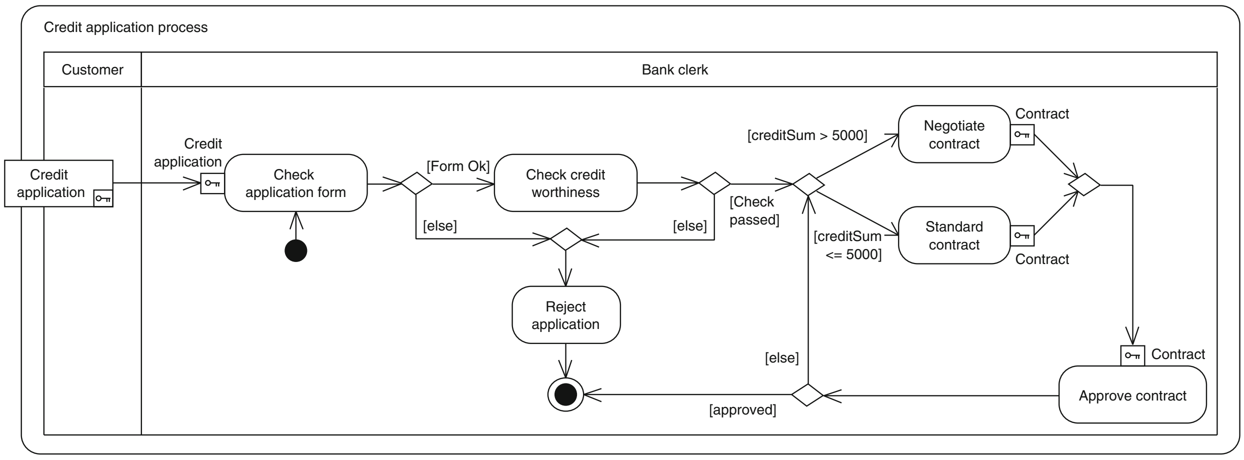


Fig. 10 Credit application process with secure object flows

Table 4 SecureObjectFlows attributes for the credit application process

Object type	SecureObjectFlows attributes
Credit application	`confidentialityAlgorithm = Aes192; integrityAlgorithm = Sha1`
Contract	`confidentialityAlgorithm = Aes192`

specifying the services, the service architecture, and the technical (i.e., executable) process descriptions. In the remainder of this paper, we focus on the specification of services and a corresponding service architecture using extended UML activities (see Sect. 4) and other UML models. Below, we describe how we derive the structural specification of a SOA (in terms of a distributed system architecture) from a business process modeled as a UML activity model. The examples given below refer to the credit application process from Fig. 10. For the sake of simplicity, we make the following assumptions:[4]

- Business units and actors involved in a process are modeled as `ActivityPartitions` (swimlanes). Thus, each swimlane indicates task ownership by a single unit or by an actor. In the service modeling step, units and actors may be modeled as SOA participants that provide and consume services. For example, the credit application activity from Fig. 10 contains two swimlanes (and thereby two task owners): The `BankClerk` receives and evaluates the credit application filed by the `Customer`.
- A process model identifies the object flows between tasks. This means that the model specifies the data items (objects) that serve as the input for and as the output of the tasks. For example, the credit application process describes object flows for a credit application and for the respective contract (see Figs. 10, 11). The corresponding data items (objects) enter a service model as the invocation data that are exchanged between the services (e.g., via messages, see [81]).
- A process model may define object flows between tasks owned by a single actor as well as object flows which occur between tasks owned by two (or more) distinct actors. Thus, an object flow within the same swimlane is executed by a single actor (i.e., it does not involve interaction between actors). In a service architecture, such an actor becomes a SOA participant which is responsible for executing a macroflow or a microflow (see also [25]). On the other hand, object flows crossing swimlane boundaries identify interactions between services that are provided and consumed by two (or more) actors (SOA participants). Such interactions effectively turn into service invocations, and the object flow details (such as object types and security attributes) are contracted via respective service interfaces.
- The data integrity and data confidentiality properties expressed via our SecureObjectFlows extension apply to both intra-swimlane and inter-swimlane object flows (of course). In the subsequent step of modeling a service architecture, the secure object flows therefore map to either or both the macroflow/microflow specifications and the service interfaces.

[4] Note, however, that we only make these assumptions to simplify the following explanations, our approach is independent of these assumptions, of course.

A service specification includes the definition of structural and behavioral views of a service architecture. In particular, we use the Service-oriented architecture Modeling Language (SoaML, [58]) and the SoaML extension UML4SOA (see [38]). The SoaML provides essential modeling primitives for structural views of a service architecture (including participants, collaborations, service contracts and interfaces, as well as messages). The UML4SOA extension is used for modeling macroflow/microflow specifications for the participants of a service architecture. Moreover, we describe how we integrated the SecureObjectFlows extension (see Sect. 4) with the SoaML and UML4SOA, respectively. Thereby, we provide a seamless mapping of integrity and confidentiality properties specified at the business process level to the structural and behavioral views of a service architecture.

5.1 Modeling the structure of a process-driven SOA

The SoaML offers an extension of the UML composite structure metamodel: Fig. 12 shows an excerpt from the SoaML metamodel extension. This extension enables a composition of service consumers and providers via a set of interacting entities, referred to as `Participants`. A participant announces its interaction capabilities and requirements through `Service` and `Request` ports, respectively (see Fig. 12). Because the `Service` and `Request` elements are derived from the UML `Port` metaclass, they specify required and provided `Interfaces`.

`Interfaces` can directly be attached to `Service` and `Request` ports or they can be specified for a port via an intermediary construct: a so called `ServiceInterface` (see also [14]). A SoaML `ServiceInterface` is derived from the `Class` metaclass (see Fig. 12) and can be used to define a port protocol. `Participants` are connected via ports. The context of such a connection is modeled

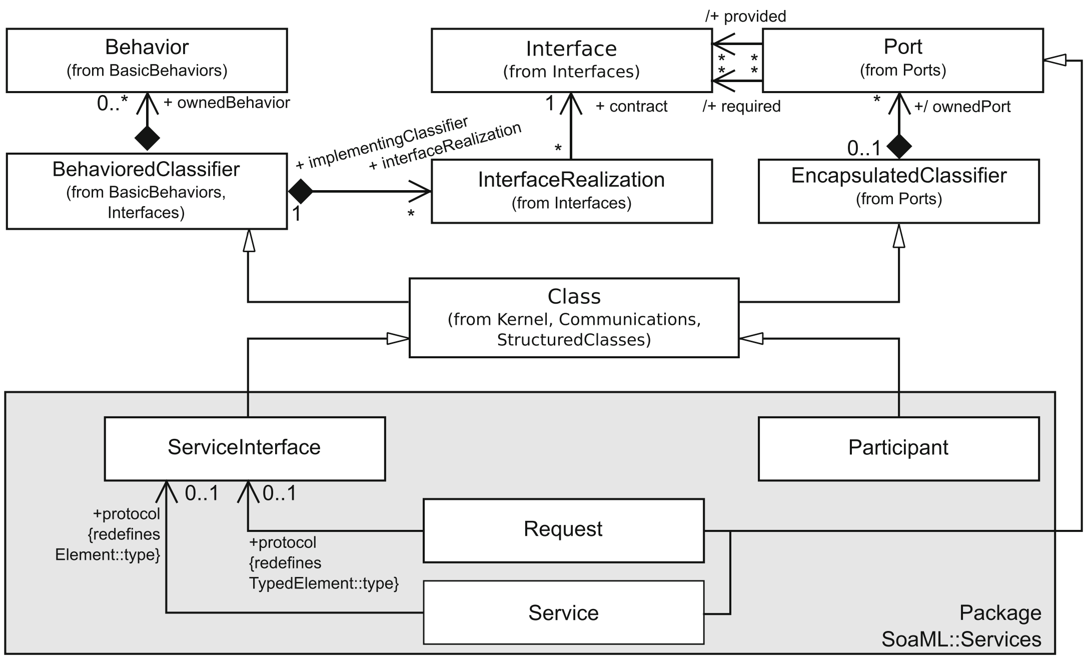


Fig. 12 Excerpt from the SoaML metamodel extension (see [58])

Table 5 Selected SoaML modeling elements

SoaML::Services metaclass	SoaML stereotype	Specialized/Extended metaclass	Description
Participant	«Participant»	Class	Represents a software system, component, or application which provides or consumes services (including process engines)
Request	«Request»	Port	Defines the service interaction point of a participant for consuming services offered by other participants, according to the port's required interfaces
Service	«Service»	Port	Defines the service interaction point of a participant for providing services to other participants, according to the port's provided interfaces
ServiceInterface	«ServiceInterface»	Class	Defines the structural (e.g., the operational interface) and behavioral properties (e.g., the invocation protocol) of a service. It is shared between a pair of request and service ports

as a `ServiceChannel`. A SoaML `ServiceChannel` ensures protocol compliance between a pair of `Service` and `Request` ports. In the structural view, protocol compliance is either expressed by sharing a `ServiceInterface` between corresponding `Request` and `Service` ports, or by directly connecting their required and provided `Interfaces`. Table 5 gives an overview of selected SoaML elements for the definition of composite structures (see also Fig. 12).

Figure 13 shows an example of a composite SOA structure modeled using the SoaML. The example includes two `Participants` (`A` and `B`) which define interacting subsystems of the SOA. Participant `B` acts as the service provider. This is modeled via its «`Service`» port. Participant `A` is a service consumer modeled via a «`Request`» port. The `requestor` and `consumer` ports are connected through a service channel. The structural dependency realized by the two ports via mutually provided and required operations is specified by the service interface `AService` (see Fig. 13). This service interface describes two participating roles (`roleA` and `roleB`) with each role referring to a corresponding `Interface`. In the context of the given service channel, the `requestor` port binds `roleA` while the `provider` port binds `roleB`. This binding indicates that the «`Service`» port implements the provided interfaces (i.e., `InterfaceB`) and uses the required interfaces (i.e., `InterfaceA`; see Fig. 13). The «`Requestor`» port of participant `A` is also typed by the `AService` interface and

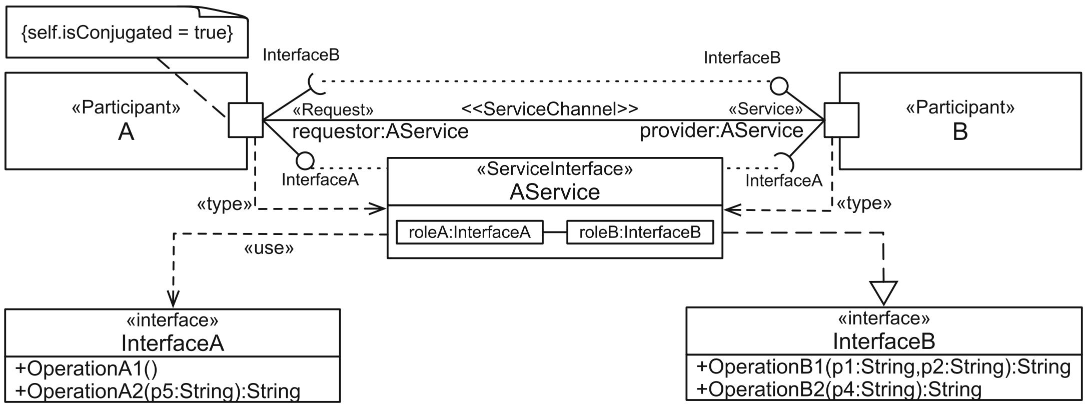


Fig. 13 Example of a composite SOA structure in the SoaML

because it is defined as a *conjugated* port, the meaning of the required and provided interfaces linked to the port is inversed (see Fig. 13). The `requestor` port provides an interface realization for `InterfaceA` and expects `InterfaceB` to be implemented by its port `provider` (for further details see [58]).

5.2 Modeling the behavior of a process-driven SOA

After defining a SoaML structure model, we require means to express the object flows resulting from service invocations (as well as their integrity and confidentiality properties). In general, we can distinguish two types of object flows in a process-driven SOA. First, object flows occurring during execution of macroflows/microflows (see [25]). These object flows are internal to a process engine and will be referred to as *process execution data* in the following. Second, object flows resulting from service invocations. Note, however, that neither business nor control data are exclusive to one of these object flow categories. They may rather be involved in both types of object flows. Consider, for example, that the credit application data from Fig. 10 are first reified as a data structure that is associated with a certain process instance and stored by the respective process engine. In a subsequent step, the data are marshaled into a message which is then delivered to a remote service endpoint. Because of this dual character of business and control data in a process-driven SOA, we require two different, yet complementary, behavioral viewpoints to specify secure object flows for these data assets. In particular, it is necessary to incorporate secure object flows in *service orchestration* specifications as well as *service choreography* specifications.

The SoaML provides explicit extension points for attaching behavioral specifications to elements of a composite SOA structure. As far as participants and service interfaces are concerned, the SoaML recommends the use of UML activities and interactions (see, e.g., [13,58]). However, the SoaML does not provide any normative guidance for specifying SOA behaviors such as service choreography and service orchestration (see, e.g., [19]).

5.2.1 Specifying a choreography via UML activities

Object flows that realize service invocations across ports are constrained by the ports' protocol. This also applies for object flows between corresponding `Service` and `Request` ports that are provided through the same `Participant`. The behavioral part of a port protocol stipulates the choreography of service invocations between the `Interfaces` and it defines the respective service invocation patterns (such as "fire and forget" or "result callbacks"). While the structural part of a port protocol is specified by a `ServiceInterface`, the `ServiceInterface` can be extended via an owned `Behavior` to express the details of a corresponding behavioral protocol. To represent secure object flows at the level of such a choreography specification, we use an extended UML activity. For example, the use of UML activities allows us to integrate the protocol viewpoint with the orchestration viewpoint in terms of `ServiceActivityNodes` (see Fig. 14).

In particular, we use UML activities as owned behavior of `ServiceInterfaces` to model flows of invocation objects. This choice permits us to model the characteristics of service invocation data (see, e.g., [81]) through UML object flows:

- *Invocation data as object nodes:* An `Activity` can model input and output parameter streams of service invocations. This is a prerequisite for applying the SecureObjectFlows extension to invocation data at this level.

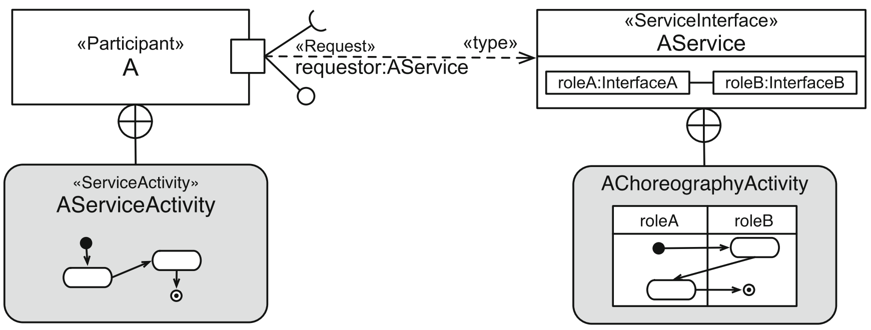


Fig. 14 Activities as behavioral specifiers

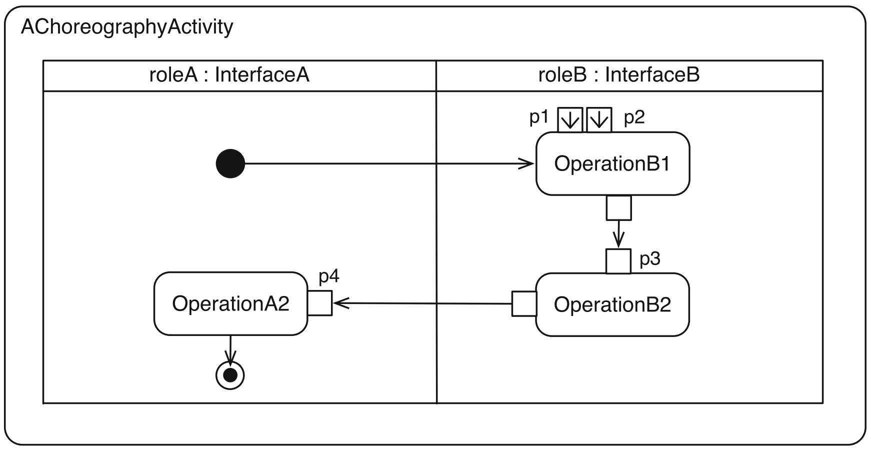


Fig. 15 Example of a choreography activity

- *Choreography roles:* ActivityPartitions can be used for modeling *choreography roles*.[5] Thus, each ActivityPartition represents an "interface-realizing role" and thereby abstracts from the Participants that use or implement the interfaces. In the compositional view, they correspond to the respective ServiceInterfaces (see roleA and roleB in Figs. 13 and 14). Note that ServiceInterfaces may refer to more than two parts (or choreography roles) and can be used to model multi-directional invocation flows.
- *Duality of invocations:* Choreography roles are typed through the Interfaces that are required and implemented by the corresponding ServiceInterface (for example InterfaceA and InterfaceB in Fig. 13). ActivityPartitions model the providers and consumers of invocations. Thus, the SecureObjectFlows elements in an Activity model consumer-side and provider-side security properties (such as signature mechanisms for messages).

[5] Note that such "choreography roles" do only model which participant provides and/or requests specific functions/interfaces. They do not model access control roles. For the definition of process-related access control models, the SecureObjectFlows extension is integrated with the extension presented in [85].

- *Standalone choreography specification:* An Activity that is owned by a ServiceInterface and the security properties that are specified for its object flows are modeled independently of the Participants which consume or implement the respective service endpoints. For example, in Fig. 13 AService applies to any pair of Participants A and B, regardless of whether they act as process engine or service providers.

Figure 15 shows an example choreography activity, where AChoreographyActivity further specifies the AService interface from Fig. 14. Through its requestor port, participant A consumes the AService interface (represented via roleA, see Figs. 13, 14). Thus, the activity defines the operation calls between the Interfaces via actions and a corresponding control flow. Object flows model the flow of input and output parameters between the operation calls (see parameters p1–p4 in Fig. 15).

SecureObjectFlows are independent of different types of invocation patterns. Therefore, we do not further elaborate on the definition of invocation patterns for distributed systems in general (see, e.g., [81]), or for process-driven SOAs (see, e.g., [97]). Note, however, that UML activities can be used to model service invocation patterns such as fire-and-forget invocations, sync-with-server invocations, request-

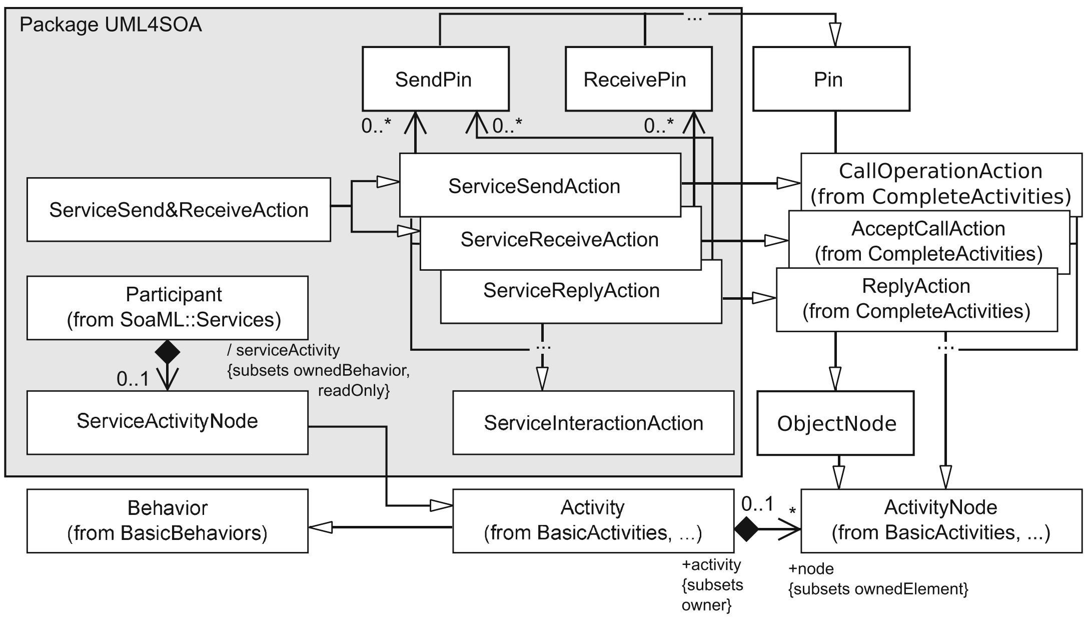

Package UML4SOA
SendPin
ReceivePin
Pin
0..*
0..*
0..*
ServiceSend&ReceiveAction
ServiceSendAction
ServiceReceiveAction
ServiceReplyAction
CallOperationAction (from CompleteActivities)
AcceptCallAction (from CompleteActivities)
ReplyAction (from CompleteActivities)
Participant (from SoaML::Services)
/ serviceActivity {subsets ownedBehavior, readOnly}
0..1
ServiceActivityNode
ServiceInteractionAction
ObjectNode
Behavior (from BasicBehaviors)
Activity (from BasicActivities, ...)
0..1
*
ActivityNode (from BasicActivities, ...)
+activity {subsets owner}
+node {subsets ownedElement}
...

Fig. 16 Excerpt from the UML4SOA metamodel (see [38])

reply invocations, or result callbacks, of course (see, e.g., [97]).

5.2.2 Specifying service orchestrations via UML activities

We use the UML4SOA [38] to model object flows as an integral part of service orchestrations. UML4SOA is a SoaML extension to model process-driven service compositions through orchestration specifications that are defined with UML activities. In a process-driven SOA, one or more `Participants` act as process engines that invoke the functions of service providers to execute tasks. The corresponding tasks are defined via a composite activity. In UML4SOA, such a composite activity (controlled by a single, orchestrating `Participant`) is modeled through a `ServiceActivityNode` (also referred to as a "service activity"). A `ServiceActivityNode` is owned by the orchestrating `Participant` (see Fig. 16 and the example from Fig. 14).

In particular, a service activity defines the control flow through `ServiceInteractionActions` and corresponding UML protocol state machines. For example, a fire-and-forget invocation is modeled using a `ServiceSendAction`. Accessing and changing the process engine's state can be modeled via `DataHandling-Actions`. The internal data flow and data dependencies of a service activity (e.g., the process engine state or the invocation data) are expressed using a set of refined object nodes (`SendPins` and `ReceivePins`) that are linked to `ServiceInteractionActions`. These `SendPins` and `ReceivePins` represent the object nodes which form object flows (see Fig. 16). Table 6 provides an overview of the UML4SOA model elements that are relevant for the remainder of this paper.

Figure 17 shows an example of a `ServiceActivityNode`. The orchestration specification `AServiceActivity` (see also Fig. 14) is modeled via a `ServiceActivityNode` that is registered as an owned behavior of the corresponding `Participant` (participant `A` from Fig. 14). Process execution starts by sending a call request to an operation `OperationB1` via the consumer port (`requestor`) of participant `A`. The process instance must provide two input parameters `p1` and `p2` as `SendPins` to the call request (see Fig. 17). The call request does not return out parameters to the process instance (i.e., it models a fire-and-forget invocation). The participant then enters a waiting state, until an inbound call request for the operation `OperationA2` is signalled through the `requestor` port. This inbound call request provides invocation data (`p4`) via a `ReceivePin` (see Fig. 17).

The examples from Figs. 15 and 17 show the complementary nature of the invocation activities and the service activities. Service invocations that are issued or expected by the orchestration specification (see Fig. 17) are reflected in the corresponding choreography specification (see Fig. 15) as the required or the provided operations. While the orchestration specification only considers consuming or providing roles (i.e., the required or implemented `Interfaces`) of a *single* participant, the choreography specification integrates

Table 6 Selected UML4SOA/SoaML modeling elements

SoaML::Services metaclass	SoaML/UML4SOA stereotype	Specialized/Extended metaclass	Description
ServiceActivityNode	«ServiceActivity»	Activity	A service orchestration specification; specific to a single SoaML participant
SendPin	«Snd»	Pin	Represents outbound invocation data provided to a service in a send action
ReceivePin	«Rcv»	Pin	Represents inbound invocation data received from a service in a receive action
ServiceSendAction	«Send»	CallOperationAction	Models a non-blocking service invocation
ServiceReceiveAction	«Receive»	AcceptCallAction	Models a blocking message listener
ServiceReplyAction	«Reply»	ReplyAction	Represents an invocation which completes a receive action; i.e., the provider-side of a result-callback invocation
ServiceSend&ReceiveAction	«Send&Receive»	ServiceSendAction, ServiceReceiveAction	Models a blocking request-reply invocation
LinkPin	«Lnk»	Pin	A reference to a service endpoint (i.e., a service or a request port)

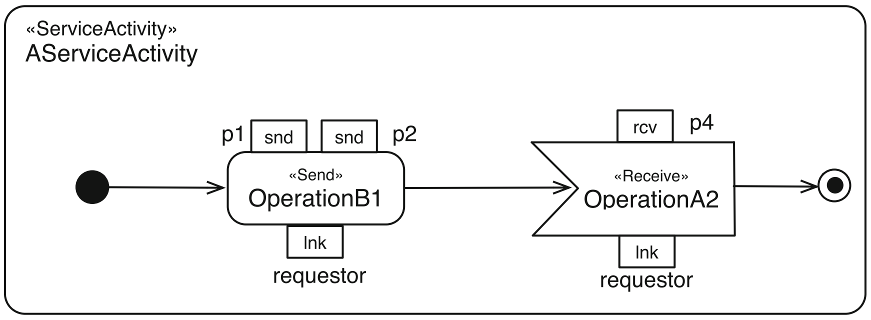


Fig. 17 Example of a service activity

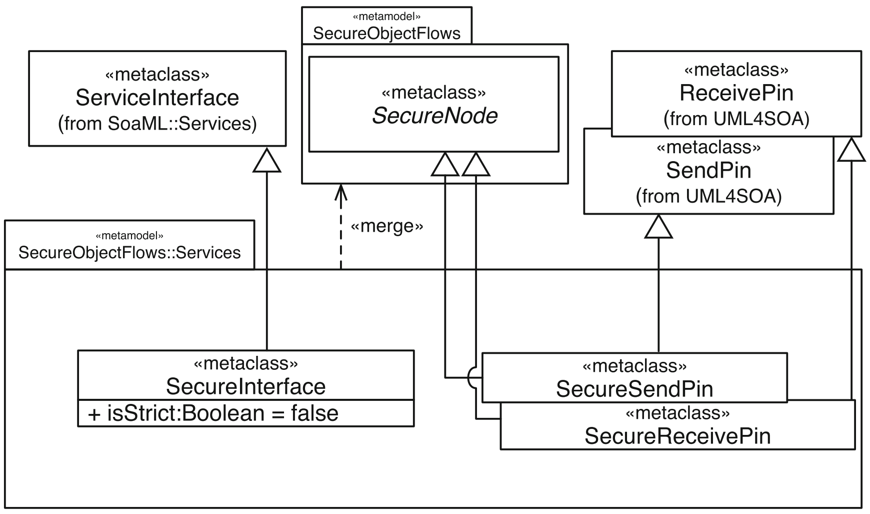
the consuming and providing roles of two or more participants for the scope of a single service channel.

5.3 A SoaML extension for secure object flows

Security concerns such as message confidentiality and message integrity are crosscutting in nature and must be addressed in different types of SOA models (see, e.g., [25,81]). Business process data (e.g., business and control objects) are exchanged in terms of invocation data. Invocation data include service endpoint references, operation names, input and output parameters, as well as exception data (see, e.g., [81]). At runtime, a process engine controls process instances that include corresponding data objects. Confidentiality and integrity properties of invocation data and process execution data affect data transformation steps at various layers of a SOA. As a result, the invocation processing infrastructure as well as the respective transport handling must be adapted. For example, if we need to ensure the integrity of data assets in a business process, a modeler must define message integrity constraints over the corresponding service interfaces. Afterwards, corresponding source code (such as message interceptors for message signing) and/or configuration data (e.g. for a security component) can be generated. Therefore, multiple views must be considered to support the definition of secure object flows in a SOA context. In this context, the choreography specifications for service interfaces and the corresponding orchestration specifications are of special importance:

- *Choreography specifications for secure flows of invocation data:* This view includes modeling support for invocation data (such as input and output parameters) which require integrity and/or confidentiality properties. In the compositional view of a SoaML model, the structural characteristics (e.g., the interface signature) of service invocations are specified via `ServiceInterfaces` (see also Fig. 12).
- *Orchestration specifications for secure flows of process execution data:* This view includes modeling support for secure object flows of process execution data as well as invocation data. A UML4SOA `ServiceActivityNode` allows to model specifying flows of process execution and invocation data for a single entity, e.g., a `Participant` or a role in the sense of a `ServiceInterface` (see also Fig. 16).

Fig. 18 The SecureObjectFlows::Services package



Modeling integrity and confidentiality properties in the choreography specifications of `SecureInterfaces` and `ServiceActivityNodes` is complementary. If the `SecureInterface` view was not available, the modeling effort would have to be duplicated for the `ServiceActivityNodes` of all `Participants` which share one or more `ServiceInterfaces`. This is because a `ServiceActivityNode` captures service invocations as patterns of `ServiceInteractionActions` from either the consumer or provider side *only* (see also Sect. 5.2). Moreover, a `Participant` may represent a process engine as well as simple service providers. As a consequence, the `ServiceActivityNode` element may not be available for all `Participants`.

To integrate the SecureObjectFlows extension with the SoaML, we provide a UML integration package (see Fig. 18). Our SecureObjectFlows::Services package adds SoaML-specific constraints for secure object flows.

5.3.1 SecureObjectFlows::Services abstract syntax

The SecureObjectFlows::Services package introduces a specialized `ServiceInterface` called `SecureInterface`. At the SoaML metamodel level, `SecureInterface` extends the `ServiceInterface` metaclass (see Fig. 18; and OCL Constraint 7 in Appendix B). A `SecureInterface` contracts either a strict or a permissive mode. The *permissive mode* is the default mode (i.e., `isStrict` is set to `false`) which allows to include secure object flows as well as ordinary object flows. In contrast, the *strict mode* (i.e., `isStrict` is set to `true`) defines that all invocation data flows (as specified further below) must be secure object flows (see OCL Constraint 8).

To model secure object flows in UML4SOA `ServiceActivityNodes`, we provide two additional metaclasses: `SecureSendPin` and `SecureReceivePin` (see Fig. 18). They integrate the capabilities of the `SecureNode` metaclass (see Sect. 4) and the `ReceivePin` and `SendPin`, respectively. For all metaclasses provided by the SecureObjectFlows::Services package, the notation for the `SecureNode` metaclass applies (see Sect. 4; [27]).

5.3.2 Constraints for the SecureObjectFlows::Services package

In this section, we discuss constraints for an `Activity` that is owned by a `SecureInterface` and for `ServiceActivityNodes` of the `Participants` which are connected by `SecureInterfaces`. The OCL constraints for the `SecureInterface` metaclass are defined over the metaclasses `SecurePin` and `SecureActivityParameterNode`. The OCL constraints for `ServiceActivityNodes` refer to the `SecureSendPin` and `SecureReceivePin` metaclasses (see Figs. 12, 16, 18).

Explicit links between invocations and interfaces: An `Activity` that specifies a choreography must only describe service invocations between `Operations` that are provided or required by the `Interfaces` referenced by the corresponding `SecureInterface` (see OCL Constraint 9). This allows the modeler to express explicit links between `Actions` in a choreography activity and the `Operation` repository represented by these `Interfaces`.

Cross-interface invocations only: `ActivityPartitions` represent "interface-realizing" and "interface-providing" roles with respect to a `SecureInterface` (see also Sect. 5.2). Object flows may occur *within* a single partition or *between* two partitions. Thus, an object flow between two `CallOperationActions`, which are modeled in different `ActivityPartitions`, depicts an output/input dependency between operations which represent an actual service invocation (see Figs. 12, 16, 18). If two adjacent CallOperationActions reside within the same ActivityPartition, they are required and provided by the same Interface. Thus, secure object flows in the SecureObjectFlows::Services package apply to cross-interface invocations (see OCL Constraint 10), and each activity node is assigned to exactly one partition (see OCL Constraint 11).

Activity parameters for initial and intermediary inbound data: A choreography activity captures data dependencies between invocations, i.e., the output data of one invocation serve as the input data for a subsequent invocation. In two important cases, however, the input data may originate from the outside. These cases are *initial* and *intermediary* inbound data.

Initial inbound data are provided through the required Interface of the consumer role. These data are contracted by the operation of a SecureInterface that triggers the execution of an Activity (see Figs. 12, 16, 18). Intermediary inbound data are not the result of previous invocations within the same choreography. The input is rather provided from the outside, such as from a process engine holding process control data which are then used as the input parameters for an operation call.

For specifying secure object flows, however, it is mandatory to model pairs of secure object nodes (see Sect. 4; [27]). This is because the security properties required at either end of an object flow must be compatible (see also Sect. 3). Thus, each secure pin and each secure activity parameter node must be connected to (at least) one object flow (see OCL Constraint 12).

Activity parameters for intermediary and flow-final outbound data: Analogous to initial and intermediary inbound data, output data can describe external data dependencies, i.e., dependencies which do not manifest within the choreography activity alone. For instance, an invocation's output may be stored in a process-persistent variable by a process engine. In this context, each secure pin and each secure activity parameter node must be connected to (at least) one corresponding object flow (see OCL Constraint 12).

Streaming-only intermediary activity parameters: ActivityParameterNodes that are used to model intermediary inbound data, and output data represent *streaming* activity parameters (see OCL Constraint 13). Streaming parameters model data which become available in the context of an activity, or which leave this context during execution of the Activity. Note that the streaming mode is only mandatory for cases of secure intermediary InputPins and OutputPins (in the sense of OCL Constraint 12).

Same origin for input data flows: Input data for service invocations, which are represented by InputPins on CallOperationActions, must have related object nodes which reside in the same ActivityPartition. Different partitions as the origins for input data of an operation are not valid (see OCL Constraint 14). This requirement follows from the intention to model input/output data dependency along a path of operation calls. The input parameters of an operation, i.e., the InputPins modeled for a CallOperationAction, must either be related to output parameters (OutputPins) of the preceding operation call, or to (initial or intermediary) inbound parameters of the choreography activity. In either case, these source object nodes must share their ActivityPartition origin.

Explicit links between orchestration and choreography activities: The overlap between the views provided by choreography activities for SecureInterfaces and ServiceActivityNodes becomes evident through the dual appearance of business and control data—once in terms of process execution data items and once as invocation data items. On the one hand, data returned from service invocations, e.g., out-parameter values, enter the process execution view as input data (e.g., for result callbacks). On the other hand, process execution data that are passed as parameter to invocation requests become in-parameters traveling across object flows in the choreography view. To avoid inconsistent models, it is important to verify these dependencies in the model specification phase. Therefore, OCL constraints 15 and 16 define consistency constraints between the two model views (see also Figs. 12, 16, 18).

5.4 UML profiles for secure object flows

Sections 4 and 5.3 introduced the SecureObjectFlows package and the SecureObjectFlows::Services package, respectively. Both packages specify UML metamodel extensions at the PIM level and provide native UML elements for the definition of secure object flows in general, and for secure object flows in SOAs in particular. An extension of the UML metamodel allows to define new and specifically tailored UML elements (defined via *new* metaclasses), and it allows to define a customized notation, syntax, and semantics for the new modeling elements. However, the integration of a metamodel extension with software tools most often results in a significant development effort. Therefore, a metamodel extension can be seen as a medium-term and long-term option to extend the UML (or another modeling language) as well as corresponding software tools.

In contrast, UML profiles provide a mechanism for the extension of *existing* UML metaclasses to adapt them for non-standard purposes. However, UML profiles are *not* a first-class extension mechanism (see [59, p. 660]) and are less powerful than metamodel extensions. In particular, UML profiles do not allow for modifying existing metamodels. Nevertheless, most UML tools directly support the definition of profiles. Therefore, it is comparatively easy to integrate

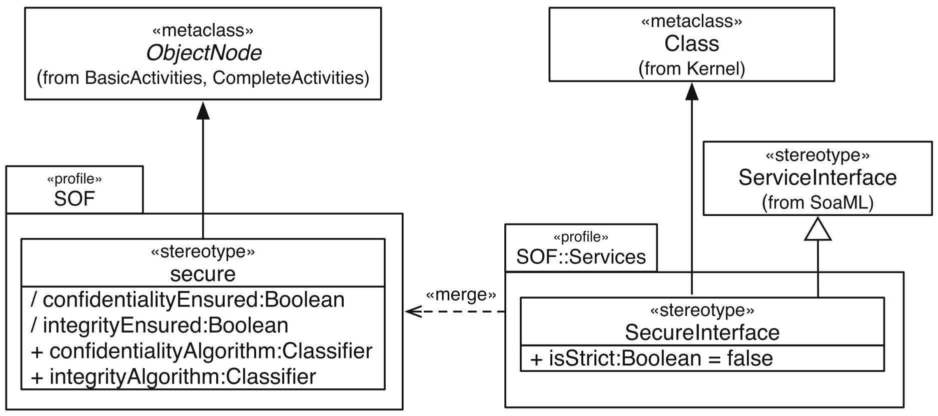
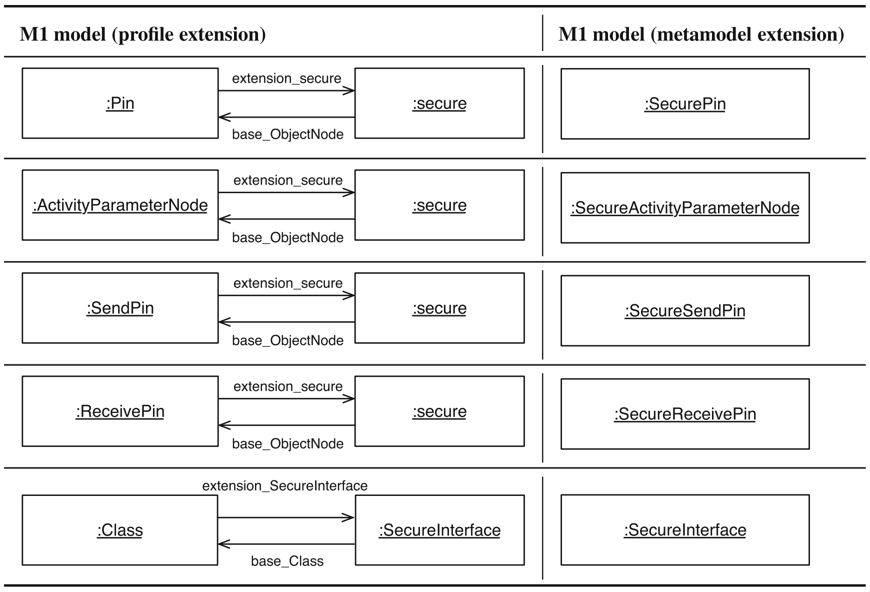

«metaclass»
ObjectNode
(from BasicActivities, CompleteActivities)
«profile»
SOF
«stereotype»
secure
/ confidentialityEnsured:Boolean
/ integrityEnsured:Boolean
+ confidentialityAlgorithm:Classifier
+ integrityAlgorithm:Classifier
«merge»
«metaclass»
Class
(from Kernel)
«stereotype»
ServiceInterface
(from SoaML)
«profile»
SOF::Services
«stereotype»
SecureInterface
+ isStrict:Boolean = false

Fig. 19 The UML profile packages SOF and SOF::Services

Table 7 Mappings between the SOF::Services profile and the SecureObjectFlows::Services metamodel extension


M1 model (profile extension)
M1 model (metamodel extension)
:Pin
extension_secure
base_ObjectNode
:secure
:SecurePin
:ActivityParameterNode
extension_secure
base_ObjectNode
:secure
:SecureActivityParameterNode
:SendPin
extension_secure
base_ObjectNode
:secure
:SecureSendPin
:ReceivePin
extension_secure
base_ObjectNode
:secure
:SecureReceivePin
extension_SecureInterface
:Class
base_Class
:SecureInterface
:SecureInterface

UML profiles in a software tool. For this reason, we introduce two UML profiles for secure object flows. By these means, we provide both a long-term option based on metamodel extensions and a short-term option based on UML profiles.

Below, we describe the UML profile packages Secure Object Flows (SOF) and SOF::Services (see Fig. 19). In Sect. 6, we present our tool support for the definition of secure object flows in process-driven SOAs based on these two profile packages.

The SOF package provides a profile for a (simplified) variant of the SecureObjectFlows package (see Sect. 4), and the SOF::Services package provides a (simplified) variant of the SecureObjectFlows::Services package (see Sect. 5.3). The `secure` stereotype provides the integrity and confidentiality attributes of the `SecureNode` metaclass (see Fig. 19). The OCL constraints for the SecureObjectFlows metamodel (see Appendix A) were adapted for the context of the secure stereotype. Table 7 shows mappings between the SOF::Services profile and the SecureObject Flows::Services metamodel extension. In particular, the following transformation rules exist: Instances of `Pin`, `ActivityParameterNode`, `SendPin`, and `ReceivePin` tagged with the «`secure`» stereotype map to instances of `SecurePin`, `SecureActivityParameterNode`, `SecureSendPin`, and `SecureReceivePin`, respectively.

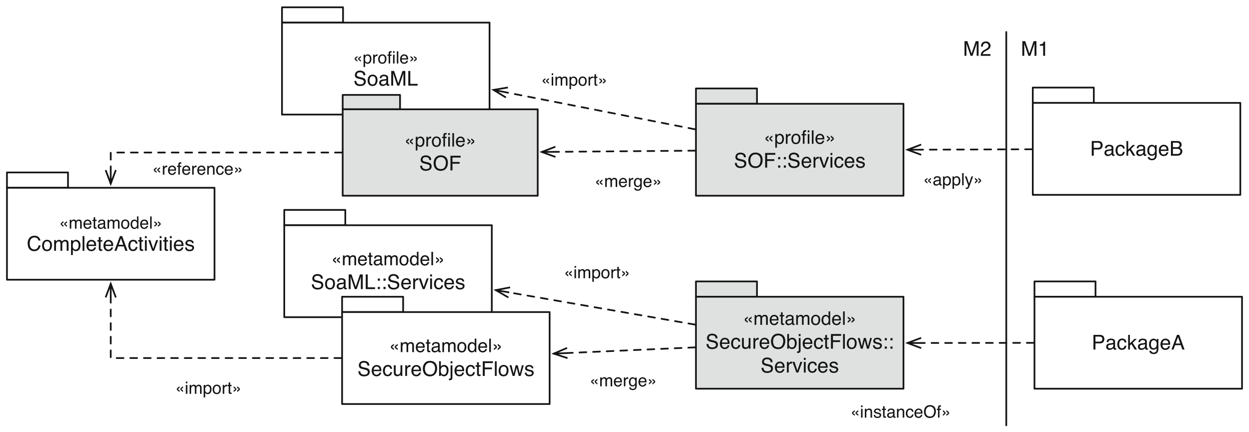


Fig. 20 The metamodel and profile packages for secure object flows

The `SecureInterface` metaclass is represented via the `SecureInterface` stereotype (see Fig. 19). In particular, this stereotype identifies a tagged `ServiceInterface` as a secure interface, and allows for specifying the strict or the permissive mode (see Sect. 5.3.1). The `SecureReceivePin` and `SecureSendPin` metaclasses are included in the SOF profile. However, an important limitation applies: `ServiceActivitiyNodes` may also contain `LinkPins` for identifying the ports of a given `ServiceInteractionAction`. Because they also represent `ObjectNodes` which are in principle extensible via the «`secure`» stereotype, we define that the «`secure`» stereotype must not be applied to object nodes tagged as «`lnk`» pins:

```
context SOF::Services::secure
inv: self.base_ObjectNode.getAppliedStereotype('UML4SOA::Services
    ::lnk') = null
```

The definition of secure object flows for the UML and their integration with the SoaML/UML4SOA via a metamodel extension as well as two profiles open up two integration paths (see Fig. 20). If we use the metamodel extension to define secure object flows, we essentially instantiate the corresponding metamodel (see “instanceOf” relation from `PackageA` to the SecureObjectFlows::Services package in Fig. 20). In contrast, if we use the profile extension to define secure object flows, we apply the profile to the corresponding UML model (see “apply” relation from `PackageB` to the SecureObjectFlows::Services package in Fig. 20). For details concerning the “instanceOf” and “apply” relations see [60].

5.5 An integrated example

Figure 21 shows an example that uses the SOF and SOF::Services profile packages—it extends the example from Sects. 5.1 and 5.2. Remember that in this example, participant `A` acts as process engine and is specified via `AServiceActivity`. The service channel between the process engine and the service provider (participant `B`) is defined through the `AService` interface which owns `AChoreographyActivity` (see also Figs. 13, 14, 15, 17).

`AChoreographyActivity` is tagged with the «`SecureInterface`» stereotype, requesting the permissive mode (see Fig. 21). Thus, not all invocation object flows in the choreography activity need to be specified as secure object flows (see also Sect. 5.3.1). The choreography activity defines that the object flow pointing towards `OperationB1` realizes a secure object flow that uses the *Aes192* as confidentiality mechanism. Moreover, the choreography activity specifies another secure object flow between `OperationB2` and `OperationA2`. This invocation object flow establishes end-to-end message integrity via the *Sha256* integrity algorithm.

`AServiceActivity` defines the orchestration specification of participant `A` (see Fig. 21). It includes the corresponding secure object flows from the perspective of the process engine (participant `A`). The two `SendPins` from the orchestration specification map to the input parameters `p1` and `p2` defined on `OperationB1` in the choreography activity. Moreover, the input parameter `p4` from the choreography activity matches the respective `ReceivePin` on the `ServiceReceiveAction` in the orchestration specification.

6 Tool support for secure object flows in SOAs

Our tool support for the definition of secure object flows in process-driven SOAs is based on the Eclipse 3.6 Model Development Tools (MDT; [11]) and the Eclipse Papyrus visual UML editor [12]. Moreover, we use the SoaML and

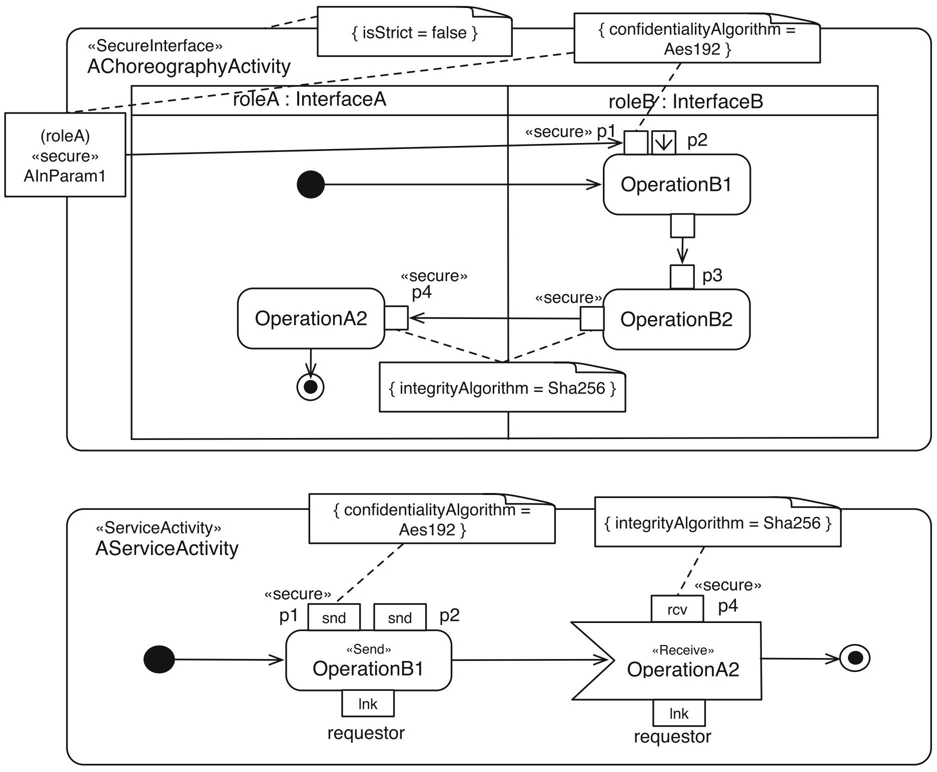


Fig. 21 Integrated views on secure object flows: orchestration and choreography specifications

UML4SOA profile definitions for MagicDraw 17.0 [51] to define service specifications. Model integrity checking based on the OCL constraints for our UML packages (see Appendices A, B) is performed in the Eclipse MDT environment.[6]

Our tool support enables automated model transformations for secure object flows that are defined via platform-independent models (PIMs). The model transformations produce corresponding platform-specific models (PSMs). The generated PSM artifacts include WSDL interface descriptions [92] and WS-BPEL process descriptions [61]. A major challenge of this model transformation step was to bridge the gap between the graph-based PIMs (defined via extended UML activities; see Sect. 5) and the block-based PSMs (defined via BPEL specifications; see, e.g., [41]). In particular, we extended the MDD4SOA Eclipse plugin [36] to support the corresponding model transformations for secure object flows. Additional transformation steps add WS-SecurityPolicy statements [63] to the generated interface descriptions and deployment descriptors. Moreover, our approach allows to add security properties to invocation data (e.g., single parameters or message elements). These security properties are transformed into WS-SecurityPolicy descriptions (e.g., `EncryptedElements`, `SignedParts`). Modeling elements that specify integrity and confidentiality requirements of object flows map to identifiers for algorithm suites (as defined by the WS-SecurityPolicy specification [63]).

In the following, we use the credit application example from Sect. 4.1 to describe our extension for the Eclipse MDT tool chain in detail. Figure 22 gives an overview of the different steps and the resulting artifacts.

For the sake of simplicity, and in order to emphasize the details relevant for this paper, we focus on the object flow which triggers the processing of a credit application (see Sect. 4.1.2). The corresponding object flow is also depicted in the topmost diagram of Fig. 22 (PIM, Business Level, Business Activity). This object flow is defined as a secure object flow and models the submission of a credit application document, that is, it specifies integrity and confidentiality properties for the transferred document.

6.1 Modeling the SOA structure

The credit application is submitted by a customer software component and received by a bank clerk software component

[6] All modeling and implementation artifacts are available from http://nm.wu.ac.at/modsec.

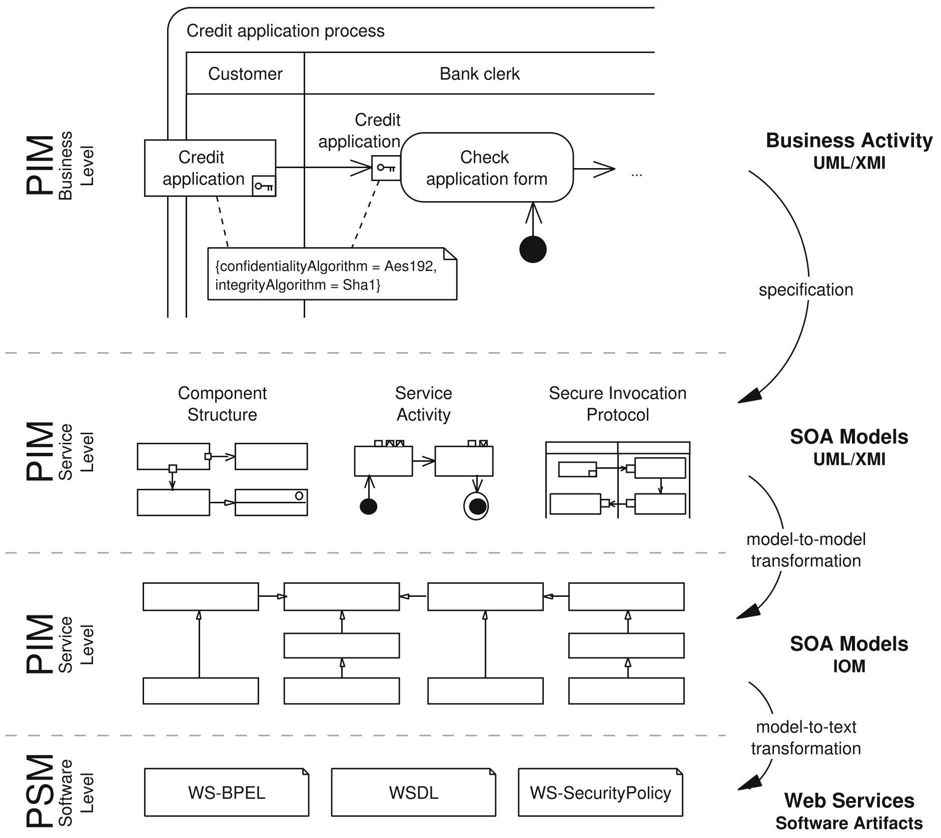


Fig. 22 Different modeling levels supported by the tool chain

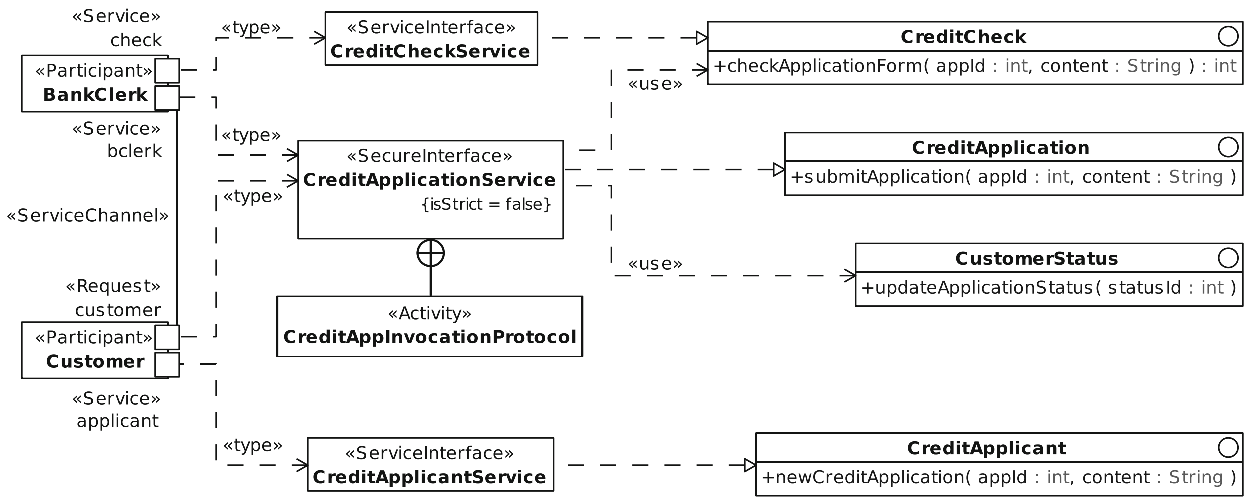

«Service» check
«Participant» BankClerk
«Service» bclerk
«ServiceChannel»
«Request» customer
«Participant» Customer
«Service» applicant
«type»
«ServiceInterface» CreditCheckService
«type»
«SecureInterface» CreditApplicationService {isStrict = false}
«type»
«Activity» CreditAppInvocationProtocol
«type»
«ServiceInterface» CreditApplicantService
«use»
«use»
CreditCheck
+checkApplicationForm(appId : int, content : String) : int
CreditApplication
+submitApplication(appId : int, content : String)
CustomerStatus
+updateApplicationStatus(statusId : int)
CreditApplicant
+newCreditApplication(appId : int, content : String)

Fig. 23 Static structure of the credit application SOA

(see Fig. 22). In the structural viewpoint of a SOA, this translates into two interacting participants named `BankClerk` and `Customer`. A corresponding SoaML composite structure model is shown in Fig. 23.

The `Customer` and `BankClerk` participants are connected through a `ServiceChannel` which carries credit application submissions. The channel connects the two participants through their `Request` and `Service` ports.

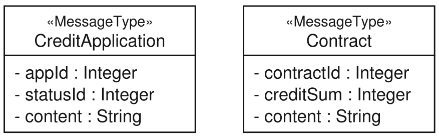


Fig. 24 Business objects as SoaML message types

The `customer` request port represents the `Customer`'s consuming interaction point. The `bclerk` service port represents the corresponding providing interaction point of the `BankClerk` participant (see Fig. 23).

The details of service invocations for submitting credit applications are negotiated by the ports' protocol, i.e., via an instance of a `SecureInterface` named `CreditApplicationService`. This `SecureInterface` defines the required and provided interfaces for service invocations traveling through the `customer` and `bclerk` ports. The structural part of a respective protocol identifies a single operation `submitApplication` that is defined via the `CreditApplication` interface. This operation realizes the application submission and must be implemented by the `BankClerk` participant via its `bclerk` port (see Fig. 23).

While this section highlights the specification of one secure object flow only, the composite structure model from Fig. 23 includes additional `ServiceInterfaces` that result from other tasks of the credit application process (see Fig. 10). For example, the `BankClerk` participant provides the `CreditCheckService` interface via its `check` service port.

The `CreditApplicationInterface` is defined as a `SecureInterface`. Corresponding confidentiality and integrity properties are specified via a UML activity attached to the `SecureInterface`. The «`SecureInterface`» slot value for `isStrict` is set to `false` (see Fig. 23). Therefore, the `SecureInterface` is said to be *permissive* (see also Sect. 5.3.1). This is necessary because service invocations other than the secured application submission are specified via the same service interface (e.g., the `updateApplicationStatus` function provided through the `CustomerStatus` interface, see Fig. 23).

In addition, we must define a structural model of the business objects via SoaML message type specifications. In the credit application example, we have two types of business objects (the `CreditApplication` and the `Contract`, see also Sect. 4.1.2). Figure 24 shows the respective SoaML message type specification for these object types. Note that in the subsequent steps of our example, we do not use a document-centric service invocation style, but rather a procedural invocation style (also referred to as RPC-style). As a result, message type features (such as `appId` or `statusId`) are included in the operation signatures rather than in the messages.

6.2 Modeling the SOA behavior

After defining the structural model, the control flow and the object flow for processing credit applications are added to the SoaML model. In Sect. 6.2.1, we specify the choreography of the `CreditApplicationService` interface. Subsequently, Sect. 6.2.2 defines the corresponding orchestration specification.

6.2.1 Choreography specification

`ServiceInterface` instances can include behavioral protocol specifications for the corresponding ports (see also Sect. 5). A behavioral protocol specification defined through a UML activity can describe various characteristics of a service channel. It identifies the `Interfaces` (grouped by a `ServiceInterface`) which depend on each other, the order and the targets of consecutive operation calls, as well as the respective invocation patterns (i.e., the choreography). In addition, input and output dependencies between the operation calls can be specified via object flows.

For the `CreditApplicationService`, we must define a choreography over four `Interfaces` and their operations: `CreditCheck`, `CreditApplication`, `CustomerStatus`, and `CreditApplicant` (see Fig. 23). Figure 25 shows a UML activity that models the corresponding choreography. The choreography identifies five data-level dependencies in terms of object flows between the call operation actions. When invoking the `newCreditApplication` operation, the `appId` and `content` parameters must be provided. Furthermore, the output parameters of the `newCreditApplication` (`content` and `appId`) are expected as input parameters for the subsequent `submitApplication` call. The fifth dependency specifies that the output parameter of the `checkApplicationForm` (`statusId`) is expected as the input parameter for the `updateApplicationStatus` action.

To model secure object flows, we tag the corresponding input and output pins of `newCreditApplication` and of `submitApplication` (see Fig. 25). The other object flows in the example are ordinary object flows. According to the constraints for the SecureObjectFlows package (see Appendix A), any secure target node must support the same security properties as its source node(s) (see OCL Constraint 5). Therefore, both secure object flows from Fig. 25 (and all four secure object nodes) support the same confidentiality and integrity algorithms (in this example AES-192 and SHA-1).

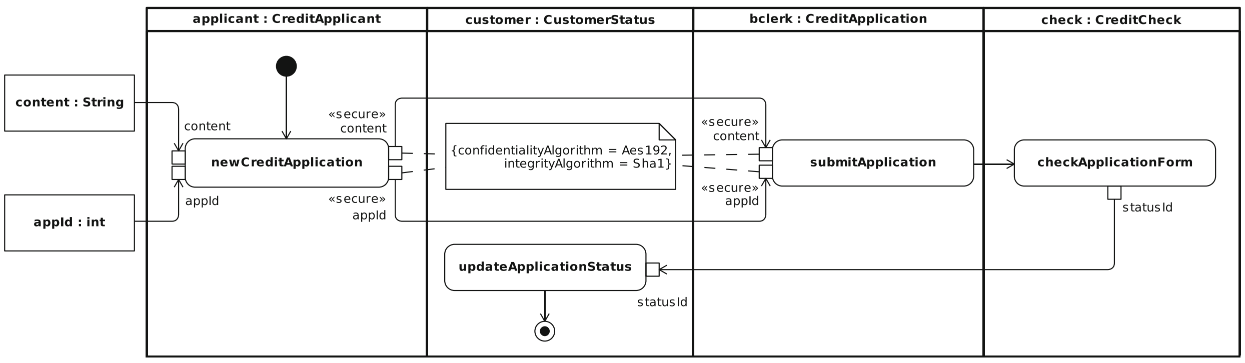


Fig. 25 Secure choreography activity

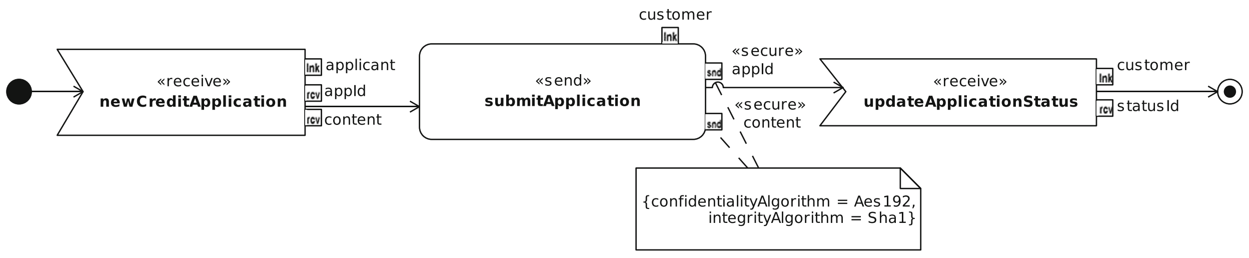

customer
«receive» newCreditApplication
applicant
appId
content
«send» submitApplication
«secure» appId
«secure» content
«receive» updateApplicationStatus
customer
statusId
{confidentialityAlgorithm = Aes192, integrityAlgorithm = Sha1}

Fig. 26 Consumer (customer) service activity

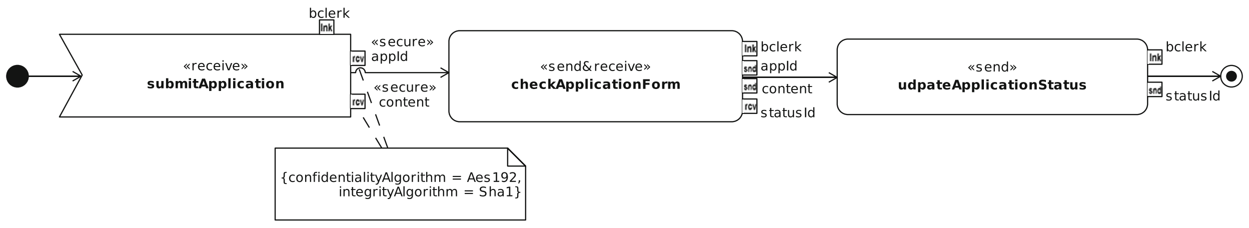

bclerk
«receive» submitApplication
«secure» appId
«secure» content
«send&receive» checkApplicationForm
bclerk
appId
content
statusId
«send» udpateApplicationStatus
bclerk
statusId
{confidentialityAlgorithm = Aes192, integrityAlgorithm = Sha1}

Fig. 27 Provider (bank clerk) service activity

6.2.2 Orchestration specification

A `ServiceActivityNode` models the behavior of a `Participant` across all service and request ports (see Sect. 5; [38]). While the choreography for `ServiceInterfaces` lays out the order of service invocations for a service channel, a `ServiceActivityNode` orchestration specification defines the control and object flow between service invocations that are controlled by a particular `Participant`. In Figs. 26 and 27, we show the `ServiceActivities` (i.e., the orchestration specifications) of the `BankClerk` and `Customer` participants.

The `Customer` service activity is triggered through the `newCreditApplication` operation call (see Fig. 26). As a result of the call request, two variables are stored in the corresponding `ReceivePins` (`appId` and `content`). After receiving the `newCreditApplication` call, the `Customer` executes the `submitApplication` operation. The `submitApplication` action specifies two `SendPins` (`appId` and `content`) which hold the input parameters that are transmitted in the corresponding operation call. The participant then enters a waiting state, until it receives the `updateApplicationStatus` request through its `customer` port (see Fig. 26).

To ensure consistency between the models, the Secure ObjectFlows extension requires that `SendPins` and `ReceivePins`, which model `SecureNodes` in the choreography (see Fig. 25), also include corresponding security properties in the respective service activity (see Fig. 26). Therefore, the `SendPins` of the `submitApplication` action (`appId` and `content`) include confidentiality and integrity properties which refer to the `OutputPins` of the `newCreditApplication` action in the associated invocation protocol (see Figs. 25, 26).

The orchestration specification for the `BankClerk` participant is shown in Fig. 27. This specification stipulates a blocking `Send&ReceiveAction` on the `checkApplicationForm` operation. Thus, after sending the request data (i.e., the `appId` and `content`), the action waits for a response which is then stored in the `ReceivePin` (`statusId`). Note that the `submitApplication` `ReceiveAction` includes secure object flow annotations (see Fig. 27). Again, the security properties for the `submitApplication` action establish a consistency link between the choreography activity and the respective service activity (see Figs. 25, 27).

6.3 Intermediary model transformations

After modeling the SOA structure and behavior (see Sects. 6.1, 6.2), the respective models are transformed into an intermediary PIM. We extended the MDD4SOA Eclipse plug-in [36,39] to automate this step for models that include secure object flows.

The corresponding processing steps perform model-to-model transformations via customized rule-based translations (see, e.g., [42]). In particular, the XMI representation (XML Metadata Interchange [55]) of the SoaML models is transformed into an intermediate object model (IOM). The primary objective of the SoaML-to-IOM transformation is to bridge between the graph-based UML PIMs (i.e., extended UML activities, see Sects. 4, 5) and the block-based PSMs defined via WS-BPEL (see, e.g., [41]). For example, in the UML-based PIMs loops are modeled via specific control nodes and control flow edges between nodes. However, because our PSM process execution format (WS-BPEL) does not allow for the definition of graphs that contain cycles, control flow loops at the PIM-level must be translated to block-structured loops in the PSMs (see, e.g., [35]).

The XMI representation of the SoaML structure and behavior models (see Sects. 6.1, 6.2) serves as input for the intermediary model transformation. Listing 1 shows an excerpt from the XMI representation for the secure choreography from Fig. 25. In particular, it shows the assignment of the activity as `ownedBehavior` to the `CreditApplicationService` class. In addition, Listing 1 includes the `appID` and `content` `ActivityParameterNodes` (lines 12–17). An example of a «`secure`» stereotype instance and the respective slot values are shown in line 30. It refers to the `ActivityParameterNode` named `content` (referenced via `id = 4149`, see lines 15–17).

The transformation maps the XMI representation of the SoaML model to the IOM which is implemented on top of Eclipse's Ecore facility. The Ecore metamodel is based on the essential meta object facility (EMOF, [53]) standard and supported by the Eclipse Modeling Framework (EMF, [83]) project.

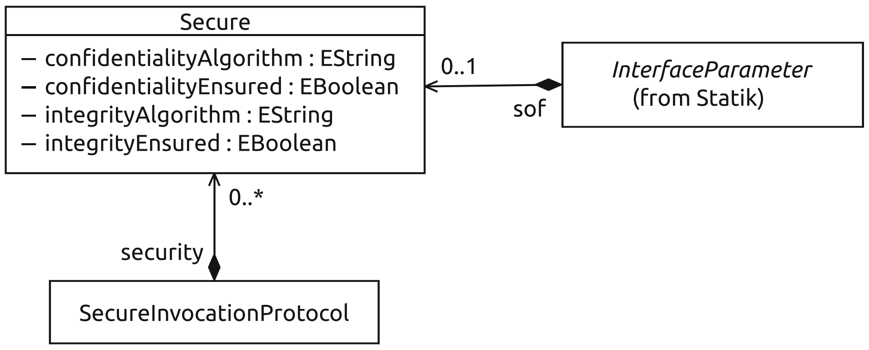
The Ecore model of the MDD4SOA IOM provides all stereotypes that are required by the UML4SOA profile via Ecore classes (`EClass`). The MDD4SOA infrastructure defines three Ecore packages for the intermediary metamodel: `Statik`, `Behaviour`, and `Data` (see [37]). The `Statik` Ecore package contains intermediary abstractions corresponding to SoaML's composite structure metamodel. The structural abstractions include Ecore classes for participants, service endpoints, and so on. Similarly, the `Behaviour` Ecore package provides an EClass for service activities. In order to transform `SecureInterfaces` (see Sects. 5.3, 5.4) and their choreography specifications into any IOM representation we had to extend the `Statik` package. The `Data` and `Behaviour` Ecore packages did not need to be changed. In particular, we had to address the following requirements to integrate the SOF package (see Sect. 5) into the IOM.

```
1  <xmi:XMI [...]>
2   [...]
3   <packagedElement xmi:type="uml:Class" xmi:id="3709" name="
         CreditApplicationService" clientDependency="7807"
         classifierBehavior="3841">
4     <ownedBehavior xmi:type="uml:Activity" xmi:id="3841" name="
           SecureInvocationProtocol" isReentrant="true" partition="
           1975">
5       [...]
6       <ownedParameter xmi:id="4081" name="appId" visibility="public
            " isStream="true">
7         <type xmi:type="uml:PrimitiveType" href="pathmap://
              UML_LIBRARIES/JavaPrimitiveTypes.library.uml#int"/>
8       </ownedParameter>
9       <ownedParameter xmi:id="4150" name="content" visibility="
            public" isStream="true">
10        <type xmi:type="uml:PrimitiveType" href="pathmap://
              UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>
11      </ownedParameter>
12      <node xmi:type="uml:ActivityParameterNode" xmi:id="4080" name
            ="appId" visibility="public" outgoing="1817"
            inPartition="3917" parameter="4081">
13        <type xmi:type="uml:PrimitiveType" href="pathmap://
              UML_LIBRARIES/JavaPrimitiveTypes.library.uml#int"/>
14      </node>
15      <node xmi:type="uml:ActivityParameterNode" xmi:id="4149" name
            ="content" visibility="public" outgoing="1801"
            inPartition="3917" parameter="4150">
16        <type xmi:type="uml:PrimitiveType" href="pathmap://
              UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>
17      </node>
18      <node xmi:type="uml:InitialNode" xmi:id="4295" name=""
            visibility="public" outgoing="1831" inPartition="3917"/
            >
19      [...]
20      <node xmi:type="uml:CallOperationAction" xmi:id="1593" name="
            " visibility="public" outgoing="1767" inPartition="3917
            " operation="4236">
21        <argument xmi:id="1604" name="statusId" visibility="public"
               incoming="2359" inPartition="3917">
22          <type xmi:type="uml:PrimitiveType" href="pathmap://
                UML_LIBRARIES/JavaPrimitiveTypes.library.uml#int"/>
23        </argument>
24      </node>
25      [...]
26    </ownedBehavior>
27     [...]
28  </packagedElement>
29   [...]
30  <SOF:secure xmi:id="q1kQ" confidentialityAlgorithm="2366"
         confidentialityEnsured="true" integrityAlgorithm="2053"
         integrityEnsured="true" base_ObjectNode="4149"/>
31   [...]
32 </xmi:XMI>
```

Listing 1 XMI excerpt from a secure choreography activity

Transformation Requirement 1 *Make all UML stereotypes defined in the SOF profile package available as model elements in the Ecore-based IOM.*

To meet this requirement, we added the EClass `Secure` to the `Statik` Ecore package (see Fig. 28). The `Secure` EClass represents the «`secure`» stereotype (see Sect. 5.4). Instances of the `Secure` EClass describe instances of the IOM's `InterfaceParameter` (see Fig. 28). The `InterfaceParameter` EClass represents input and output parameters for `InterfaceOperations`. Because



Secure
– confidentialityAlgorithm : EString
– confidentialityEnsured : EBoolean
– integrityAlgorithm : EString
– integrityEnsured : EBoolean
0..1
sof
InterfaceParameter
(from Statik)
0..*
security
SecureInvocationProtocol

Fig. 28 Ecore-based IOM of the SOF extension

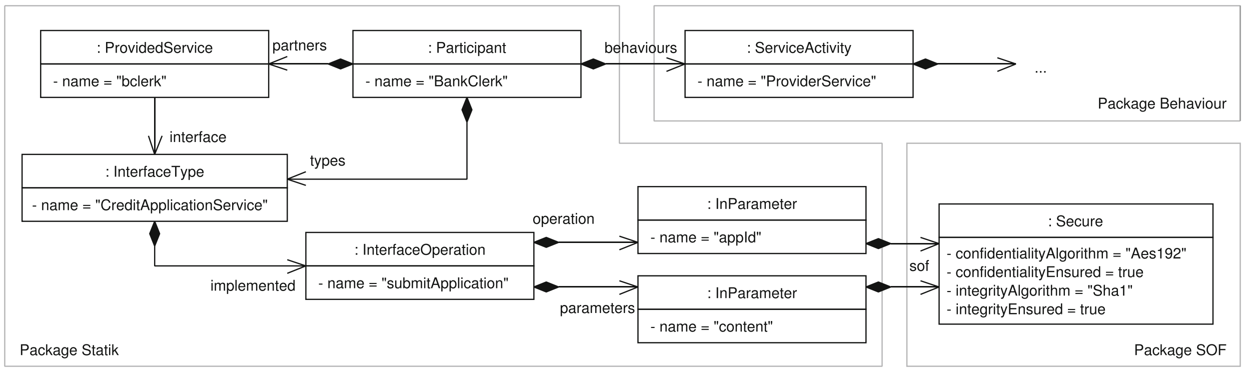


Fig. 29 Excerpt from the IOM object model for the credit application example

the `Pin` types used in `ServiceActivityNodes` and `SecureInterface` choreography activities map to the input and output parameters of the corresponding UML operations, the `InterfaceParameter` EClass qualifies as an appropriate extension point for attaching secure object flow properties at the IOM level.

Transformation Requirement 2 *Define transformation rules for converting an XMI-based into an IOM-based representation of secure object flows.*

We implemented a converter component to transform secure object flows into an IOM structure. Figure 29 shows an excerpt from the IOM model that the converter produced for the credit application example. In principle, the conversion from SoaML/XMI artifacts into IOM instances involves three steps: First, we identify the `SecureInterfaces` and their owned activities (i.e., the choreography specifications) from the SoaML/XMI document. Second, the secure object flows from the choreography specifications that involve an invocation between two entities (e.g., two distinct ports) are selected for further processing. Then, the `SecureNodes` included in these object flows are mapped to IOM instances (in particular instances of type `InterfaceParameter`, see also Fig. 28). Third, the resulting IOM is serialized into its Ecore/XMI representation.

6.4 Platform-specific model transformations

After the intermediary model transformations, our extended MDD4SOA plug-in creates platform-specific models (PSMs) for a selected execution platform. While our approach is not limited to this platform family, the MDD tool chain described in this paper integrates with Web services communication middleware. In particular, the IOM representations of the SOA composite structure (i.e., the service interfaces and message type specifications) are transformed into WSDL interface descriptions [92], and the behavioral parts (i.e., the service activities) are translated into WS-BPEL execution specifications [61]. For our credit application example, we generate WS-BPEL specifications for the Apache orchestration director engine (ODE, [2]). The corresponding secure object flow properties are transformed into WS-SecurityPolicy statements [63]. Finally, supplemental artifacts such as deployment descriptors for Apache ODE are created.

Transformation Requirement 3 *Secure object flow properties of the PIM must be mapped to the PSM.*

The WS-SecurityPolicy standard [63] allows to define a number of security binding properties. In particular, it provides a list of security algorithm suites for cryptographic operations with symmetric or asymmetric encryption mech-

Table 8 Examples for WS-SecurityPolicy algorithm suites

Encryption	Aes128	Aes192	Aes256	TripleDes
Digest	Sha1	Sha1	Sha1	Sha1
Algorithm suite	*Basic128*	*Basic192*	*Basic256*	*TripleDes*

anisms. To be compliant with the WS-SecurityPolicy standard, Web service communication middleware must provide software support for these security bindings (message-level encryption). Each algorithm suite specifies the actual algorithm and the respective key lengths [63]. Table 8 shows encryption algorithms and hash functions which can be applied in the credit application example.

Thus, our extended MDD4SOA plug-in must perform an automated mapping of secure object flow properties (i.e., confidentiality and integrity attributes) to the algorithm suites specified by the WS-SecurityPolicy standard. In the credit application example, the following security properties have been defined (see also Figs. 25, 26, 27): `confidentialityAlgorithm = Aes192` and `integrityAlgorithm = Sha1`. Both, encryption algorithm and integrity algorithm, correspond to the *Basic192* algorithm suite of the WS-SecurityPolicy standard (see Table 8).

Transformation Requirement 4 *Implement transformation rules for converting the IOM/XMI representation into the PSMs.*

Because our PSMs are Web service artifacts, we map the Ecore representation of the `Statik` IOM package (e.g., the EClasses `Service`, `InterfaceOperation`, `InterfaceParameter`; see Sect. 6.3) to WSDL descriptions. The artifacts from the `Behaviour` package are translated to WS-BPEL definitions (e.g., `ServiceActivityNode` or `ServiceProtocol`; see [37]). The EClass from our security extension in the IOM SOF package is transformed into WS-SecurityPolicy fragments.

The WS-SecurityPolicy specification allows to define nested policy assertions (see [94]). In principle, the WS-SecurityPolicy standard defines three attachment points for policies (called *policy subjects*, see [93]):

Endpoint policy subject A policy that applies to the service at the endpoint level.
WSDL attachment points: `wsdl:binding` and `wsdl:port`.

Operation policy subject A policy on a per-operation basis. WSDL attachment points: `wsdl:binding/wsdl:operation`.

Message policy subject A policy at the message level.
WSDL attachment points: `wsdl:binding/wsdl:operation/wsdl:input`, `wsdl:binding/wsdl:operation/wsdl:output`, and `wsdl:binding/wsdl:operation/wsdl:fault`.

A benefit of the SecureObjectFlows extension is its ability to model data security properties at the level of individual object flows. Therefore, we are able to model confidentiality and integrity requirements of single invocation parameters, if necessary. This level of detail can be expressed through the *Message Policy Subject* of the WS-SecurityPolicy specification (see above).

We implemented a converter which adds WS-SecurityPolicy statements to the WSDL interface descriptions (see Listing 2 for an example). In particular, we need to find all `InterfaceParameters` from the IOM which include confidentiality and integrity properties. For every secured `InterfaceParameter`, we must identify the corresponding WS-SecurityPolicy algorithm suite (see Table 8). Next, we have to generate a policy assertion for each interface parameter with the corresponding `SignedElements`, `EncryptedElements`, `ContentEncryptedElements`, and `AlgorithmSuite` (see Listing 2, lines 3–22). Subsequently, each message parameter in the binding definition of the WSDL document is extended to contain a reference to the corresponding policy assertion in terms of a `PolicyReference` statement (see Listing 2, line 30).

```
1  <definitions [...] name="bclerk" [...]>
2   [...]
3   <wsp:Policy wsu:Id="sp_submitApplication_input_appId">
4     <wsp:ExactlyOne>
5       <wsp:All>
6         <sp:SignedElements>
7           <sp:XPath>/Envelope/Body//appId</sp:XPath>
8         </sp:SignedElements>
9         <sp:EncryptedElements>
10          <sp:XPath>/Envelope/Body//appId</sp:XPath>
11        </sp:EncryptedElements>
12        <sp:ContentEncryptedElements>
13          <sp:XPath>/Envelope/Body//appId</sp:XPath>
14        </sp:ContentEncryptedElements>
15        <sp:AlgorithmSuite>
16          <wsp:Policy>
17            <sp:Basic192/>
18          </wsp:Policy>
19        </sp:AlgorithmSuite>
20      </wsp:All>
21    </wsp:ExactlyOne>
22  </wsp:Policy>
23  [...]
24  <binding name="bclerk_binding" type="this:bclerk">
25    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org
            /soap/http"/>
26    <operation name="submitApplication">
27      <soap:operation soapAction="submitApplication"/>
28      <input name="submitApplication_input">
29        <soap:body namespace="http://www.mdd4soa.eu/generated/
                BankClerk/" use="literal"/>
30        <wsp:PolicyReference URI="#sp_submitApplication_input_appId
                " required="true"/>
31        [...]
32      </input>
33    </operation>
34    [...]
35  </binding>
36  [...]
37 </definitions>
```

Listing 2 Excerpt from a WSDL document with WS-SecurityPolicy assertions

Transformation Requirement 5 *Execution of generated PSMs in a selected environment.*

The credit application example was deployed in the Apache ODE 1.3.5. Apache Axis2 [1] serves as the integration layer for the communication over Web services. Apache Rampart (the security module of Axis2; [3]) enforces the corresponding WS-SecurityPolicy specifications. With the automated creation of deployment descriptors, we realize the integrated model-driven specification of secure object flows from the CIM and PIM levels down to the execution of the PSM in a specific platform environment (here: a SOA environment).

7 Related work

We use three main characteristics to review related approaches: 1) whether a *formal and generic metamodel* (CIM) is provided, 2) whether *modeling support* (for PIMs) is provided, and 3) whether an *integrated tool chain* is available (including automated model transformations into PSMs). For each of these characteristics, we further consider multiple sub-criteria. Table 9 shows an overview of related work on modeling of secure object flows for process-driven SOAs. With respect to the concepts and artifacts specified in Sects. 3–6, we use a $\surd$ if a related approach provides similar or comparable support for a certain concept; and a $\triangle$ if a related approach provides at least partial support for a particular concept.

Wolter et al. [89,90] presented an approach to model security goals in business processes. The security goals are mapped to a workflow model, and finally to executable Web service specifications. Wolter et al. provide generic metamodels for security properties that are expressed via concept diagrams which loosely resemble UML class diagrams, whereas the workflow models use BPMN (see also [91]). Informal semantics can be derived from UML-like security policy diagrams and the assisting comments. However, they do not provide an integrated metamodel for the business process view and crosscutting security views (neither at the CIM nor at PIM level). As a result, Wolter et al. only sketch their model transformation framework. In particular, they do not describe how the model transformations actually merge the security goals and the BPMN process descriptions to produce platform-specific Web service artifacts. Furthermore, the behavioral model view (BPMN) only provides model annotations for security properties without processable or formalized semantic constraints. Therefore, the approach does not support the specification of security properties for object flows at various abstraction levels (such as process assets, process execution data, invocation data, messages). Regarding tool support, the authors provide an overview of a transformation framework based on annotated XMI representations of BPMN models. However, a detailed discussion of how the security properties are integrated with the XMI representation is missing.

Gilmore et al. [20] discuss modeling of non-functional aspects for service-oriented systems. They define a UML profile based on the SoaML that supports the definition of non-functional properties (performance, reliable messaging, and security). Similar to our work, the approach of Gilmore et al. uses the UML4SOA extension [36,38,39]. However, in contrast to [20] our work is based on a formal and generic metamodel (CIM) that integrates the process flow view with our security extension (see also [85]). Thus, in principle, our CIM for secure object flows can be used to extend arbitrary process modeling languages. Moreover, Gilmore et al. treat security specifications along with non-functional properties such as QoS and emphasize a (predominantly) structural view on security properties. The structural view especially includes non-functional contracts that are associated with SoaML interfaces. However, this design choice also limits the expressiveness, in particular, security properties cannot be specified at different abstraction levels (such as message parts, scope of single invocations, or context dependencies). Gilmore et al. define non-functional properties as part of a service interface. Therefore, a contract specifies requirements for every interaction between the corresponding participants. Thus, it is not possible to define security properties at the level of individual service invocations or at the level of separate message parameters. In contrast, our secure object flows provide means for a much more fine-grained specification of security properties from the CIM to the PSM level. However, the approach from [20] could be integrated with our approach. The level of service contracts could add an additional, global specification scope for integrity and confidentiality properties.

Jürjens et al. present UMLsec [32,33], a UML profile that supports the definition of various security properties. For example, UMLsec is used to define and verify cryptographic protocols. Moreover, data confidentiality and integrity stereotypes can define dependencies in static structure or component diagrams. While UMLsec does not explicitly target SOAs, it could be integrated with other UML-based approaches such as the SoaML with moderate effort. Jürjens et al. offer tool support for running static and behavioral checks and a permission analyzer for access control mechanisms (see [78]). The original UMLsec approach was defined using UML 1.5 [32]. Recently, they introduced a corresponding UML 2.3 profile [78] as well as Eclipse-MDT-based tool support [88]. However, regarding model-driven development UMLsec does not (yet) provide a model transformation framework and, consequently, does not address platform integration issues. Therefore, UMLsec is well-suited to be integrated with our approach. For example, UMLsec models can be supported by our model transformation framework to target other (non-SOA) platforms.

Table 9 Secure object flows for process-driven SOAs: related work

	Integrity and confidentiality	Formal and generic metamodel (CIM)	Integrated metamodel (CIM, PIM, PSM)	Formal semantics	Separation of concerns	Multiple model views	Multiple refinement levels	Tool support	Model transformation framework
Model-driven sec. requirement spec. [89,90]	√	△						△	√
Non-functional properties in MDD [20]	√		√	△	△	△		△	√
Model-driven security [5]		√	△	√	△			√	△
Secure sys. development with UML [32,33]	√	√	△	√	△	√	√	△	
Inter-org. workflow sec. [21–24,40]	√	△	√	△	△	√	√	√	√
Sec. for WS-based business processes [31]	√					√		△	√
Secure business process model spec. [66,67]	√	△	△	△				△	△
MDD of security aspects [65]		△	△		△			△	√
MDA approach to access control spec. [15]		△	√	△				△	△
Generative arch. for model-driven sec. [68]	√	△				△	√	△	√
Model-driven security based on WS [44,86]	√		△					△	△
Secure object flows (our approach)	√	√	√	√	√	√	√	√	√

The SECTET framework [21–24,40] aims at supporting the design and implementation of security-critical inter-organizational workflows based on Web services. The approach includes a metamodel consisting of multiple UML class diagrams. However, SECTET is not based on a formal CIM. Yet, because SECTET is based on the UML, the respective PIMs are (implicitly) based on a common metamodel (i.e., the UML metamodel). Nevertheless, SECTET does not use SOA-related UML extensions (such as the SoaML). Formal semantics for SECTET models are specified via a custom, OCL-like constraint language. A global workflow view is used for specifying message exchange contracts between service providers and service consumers. However, security properties of object flows between service invocations are not covered.

In [31], Jensen and Feja extend a proprietary MDD software tool for modeling SOA-based security properties. The approach uses event-driven process chains (EPCs), but does not provide a formal and generic CIM. Security models are defined separately from process models and add security-related notation elements to different workflow elements. Security and process models are integrated via a model transformation step. However, the approach does not provide formal semantics for the process-related security properties. In addition, the security properties are defined for the scope of a single process engine (rather than for a collaboration of service partners).

Rodríguez et al. [66,67] present a UML profile to model security requirements via UML activities. Separation of concerns and multi-level specification of security properties are not supported. The approach provides a single process-related view, yet, it could be extended with additional views provided in our approach. Rodríguez et al. describe PIM transformations to use case and class diagrams. However, they do not define PSM transformations. The PIM transformation rules are specified via QVT [54], but they are not integrated with the overall tool chain. In [67], they describe the modeling of security requirements in BPMN and the corresponding model-to-model transformations into UML activity diagrams. Rodríguez et al. do not provide a formal and generic metamodel (CIM).

Reznik et al. [65] address the automatic generation of security-critical applications for different middleware platforms. The approach defines a UML profile for an adapted subset of the UML metamodel. Corresponding PIMs are mapped to a security-extended CORBA component model. While their PIMs are based on a UML subset, Reznik et al. redefine some parts of the UML metamodel. As a result, the corresponding subset it is not compliant with the UML standard. Moreover, because their UML profile is based on the adapted UML subset it is not supported by standard UML tools. The approach of Reznik et al. does neither provide a CIM nor does it discuss multi-view modeling or refinement of security properties.

In [68], Sanchez et al. present a model-driven approach for the definition of different security properties. The approach is based on multi-stage, automatic model transformations. The generic metamodel of the approach is defined as a UML M1 class diagram. Sanchez et al. do not provide a CIM, they emphasize that their approach is limited to a specific modeling language. In [68], they describe a custom modeling language that uses the Eclipse Ecore facility. Hence, regarding syntax and semantic, their generic UML-based metamodel does not integrate seamlessly with their custom modeling language.

Nakamura et al. [44] and Tatsubori et al. [86] present a toolkit for generating Web service security configurations. UML class models are applied to provide a structural view on a SOA. In addition, they provide stereotypes to define selected security properties. However, the corresponding UML profile is defined in an ad-hoc fashion and does not conform to the UML standard. Process views are not considered, but could be added via an extension of their approach. Nakamura et al. and Tatsubori et al. neither define a formal or generic metamodel (CIM) nor do they discuss formal semantics or issues regarding separation of concerns.

In addition to the approaches described above, multiple other model-driven security approaches exist that do not address the modeling of secure object flows but focus on other process-related security aspects. For example, Fink et al. [15] propose to generate access control specifications from MOF-based models, and Basin et al. [5] specify access control properties via domain-specific UML profiles.

8 Conclusion

In this paper, we presented an integrated approach to model and to enforce secure object flows in process-driven SOAs. In particular, we provide a generic metamodel (CIM) for secure object flows, a corresponding UML extension to define platform-independent models (PIM), a model transformation framework for PIM-to-PSM transformations for Web service artifacts, as well as corresponding tool support. Our approach enables the continuous specification and the enforcement of confidentiality and integrity properties for object flows in business processes that are executed in a distributed system. Moreover, secure object flows are a part of the Business Activities framework and are thereby directly integrated with extensions for the specification, for checking, and for the enforcement of other security properties, such as access control or audit rules (see, e.g., [6,28,30,71–76,84,85]).

Our approach follows the model-driven development paradigm. At the PIM level, we provide both a metamodel extension for secure object flows (see Sect. 4) as well as a corresponding UML profile (see Sect. 5). For both extensions, we provide OCL constraints that define consistency requirements for the corresponding modeling artifacts. Therefore, each valid secure object flow must conform to the respective OCL constraints. Our metamodel extension provides specifically tailored UML elements. It can be used to extend future versions of UML tools with native modeling support for secure object flows. In contrast, a UML profile adapts *existing* UML metaclasses for non-standard purposes. In our case, it extends the SoaML/UML4SOA with modeling support for secure object flows. Because most UML tools directly support the definition of profiles, it is comparatively easy to integrate UML profiles in a software tool. Our tool support for secure object flows in process-driven SOAs is based on the Eclipse Model Development Tools (see Sect. 6). Our model transformation framework generates WS-BPEL, WSDL, and WS-SecurityPolicy artifacts. The generated artifacts were deployed in the Apache orchestration director engine (ODE). Apache Axis2 serves as the integration layer for the communication over Web services. Apache Rampart enforces the corresponding WS-SecurityPolicy specifications. However, note that our approach is generic and does neither depend on the UML nor on a specific software tool or runtime environment. Thus, it can also be applied to extend other modeling languages or other software tools.

In addition to the main contributions of this paper, we also gained numerous other experiences and insights. Some of which are specific to certain design artifacts. For example, during the tool integration, it turned out that the UML4SOA extension [38] deviates from the UML standard by using an `AcceptCallAction` with the «Receive» stereotype for defining an event or message listener (i.e., a `ServiceReceiveAction`). Moreover, it demands that a specific stereotype is defined on the corresponding input pins. However, the resulting constraint is not compliant with the UML standard. This small yet important anomaly directly influenced the tool integration and corresponding PIM-to-PSM transformations. In general, such small anomalies may always occur if one integrates different modeling extensions in a consolidated tool environment.

In our future work, we will extend the Business Activities framework to provide an integrated environment for the model-driven specification, checking, deployment, and enforcement of secure business processes in distributed systems.

Appendix A: Constraints for the SecureObjectFlows package

This section provides the complete list of OCL-expressions for the UML extension specified in Sect. 4.

OCL Constraint 1 *The confidentialityEnsured attribute of the* `SecureNode` *classifier is derived from the confidentialityAlgorithm attribute and evaluates to true if a confidentiality-related security property is supported.*

```
context SecureObjectFlows::SecureNode::confidentialityEnsured :
      Boolean
derive: confidentialityAlgorithm->notEmpty()
```

OCL Constraint 2 *The integrityEnsured attribute of the* `SecureNode` *classifier is derived from the integrityAlgorithm attribute. It evaluates to true if an integrity-related security property is supported.*

```
context SecureObjectFlows::SecureNode::integrityEnsured : Boolean
derive: integrityAlgorithm->notEmpty()
```

OCL Constraint 3 *A secure object node must ensure either or both the confidentiality and the integrity.*

```
context SecureObjectFlows::SecureNode
inv:    self.confidentialityEnsured or
        self.integrityEnsured
```

OCL Constraint 4a *The successor object node of a secure object flow must also be a secure object node.*[7]

```
context SecureObjectFlows::SecureNode
def:    allSuccessors(node : ActivityNode) : Set(ActivityNode) =
     node.outgoing.target->collect(x |
         allSuccessors(x))->asSet()->union(node.outgoing.target)
inv:    allSuccessors(self)->select(oclIsKindOf(ObjectNode))->
     forAll(node |
         node.oclIsKindOf(SecureNode))
```

OCL Constraint 4b *The successor object node of a secure object flow must also be a secure object node.*[8]

```
context SecureObjectFlows::SecureNode
def:    allSuccessors(node : ActivityNode) : Set(ActivityNode) =
     node.outgoing.target->closure(x |
         x.outgoing.target)->asSet()->union(node.outgoing.target)
inv:    allSuccessors(self)->select(oclIsKindOf(ObjectNode))->
     forAll(node |
         node.oclIsKindOf(SecureNode))
```

OCL Constraint 5 *The successor secure object nodes must support the same security properties as the corresponding source secure object node.*

```
context SecureObjectFlows::SecureNode
inv:    allSuccessors(self)->select(oclIsKindOf(SecureNode))->
     forAll(node |
         node.oclAsType(SecureNode).confidentialityEnsured implies
           node.oclAsType(SecureNode).confidentialityAlgorithm =
                 self.confidentialityAlgorithm)
inv:    allSuccessors(self)->select(oclIsKindOf(SecureNode))->
     forAll(node |
         node.oclAsType(SecureNode).integrityEnsured implies
           node.oclAsType(SecureNode).integrityAlgorithm = self.
                 integrityAlgorithm)
```

OCL Constraint 6a *All secure object nodes having the same target secure object node must support identical security properties.* [9]

```
context SecureObjectFlows::SecureNode
def:    allPredecessors(node : ActivityNode) : Set(ActivityNode) =
      node.incoming.source->collect(x |
         allPredecessors(x))->asSet()->union(node.incoming.source)
inv:    allPredecessors(self)->select(oclIsKindOf(SecureNode))->
     forAll(node |
         node.oclAsType(SecureNode).confidentialityEnsured implies
           node.oclAsType(SecureNode).confidentialityAlgorithm =
                 self.confidentialityAlgorithm)
inv:    allPredecessors(self)->select(oclIsKindOf(SecureNode))->
     forAll(node |
         node.oclAsType(SecureNode).integrityEnsured implies
           node.oclAsType(SecureNode).integrityAlgorithm = self.
                 integrityAlgorithm)
```

OCL Constraint 6b *All secure object nodes having the same target secure object node must support identical security properties.* [10]

[7] This constraint conforms to the OCL standard version 2.2 [56].

[8] This constraint conforms to the OCL standard version 2.3.1 [57].

[9] This constraint conforms to the OCL standard version 2.2 [56].

[10] This constraint conforms to the OCL standard version 2.3.1 [57].

```
context SecureObjectFlows::SecureNode
def:    allPredecessors(node : ActivityNode) : Set(ActivityNode) =
          node.incoming.source->closure(x |
            x.incoming.source)->asSet()->union(node.incoming.source)
inv:    allPredecessors(self)->select(oclIsKindOf(SecureNode))->
          forAll(node |
            node.oclAsType(SecureNode).confidentialityEnsured implies
              node.oclAsType(SecureNode).confidentialityAlgorithm =
                self.confidentialityAlgorithm)
inv:    allPredecessors(self)->select(oclIsKindOf(SecureNode))->
          forAll(node |
            node.oclAsType(SecureNode).integrityEnsured implies
              node.oclAsType(SecureNode).integrityAlgorithm = self.
                integrityAlgorithm)
```

Appendix B: Constraints for the SecureObjectFlows::Services package

This section provides the complete list of OCL constraints for the UML extension specified in Sect. 5.

OCL Constraint 7 *A* `SecureInterface` *must own an* `Activity` *instance as its owned behavior.*

```
context SecureObjectFlows::Services::SecureInterface
inv:    self.ownedBehavior->one(oclIsKindOf(Activity))
```

OCL Constraint 8 *In strict mode all cross-interface object flows must be secured.*[11]

```
context SecureObjectFlows::Services::SecureInterface
def:    allPredecessors(node : ActivityNode) : Set(ActivityNode) =
          node.incoming.source->collect(x |
            allPredecessors(x))->asSet()->union(node.incoming.source)
inv:    self.isStrict implies
          self.ownedBehavior.oclAsType(Activity).node->select(
            oclIsKindOf(ObjectNode))->forAll(node |
              allPredecessors(node)->select(incoming->isEmpty())->
                forAll(s |
                  s.inPartition <> node.inPartition implies
                    s.oclIsKindOf(SecureNode) and node.oclIsKindOf(
                      SecureNode)))
```

OCL Constraint 9 *All* `Actions` *must be instances of* `CallOperationAction` *and each* `CallOperationAction`*'s operation enclosed by a given partition must correspond to an* `Operation` *owned by the* `Interface` *denoted by this partition.*

```
context SecureObjectFlows::Services::SecureInterface
inv:    self.ownedBehavior.oclAsType(Activity).node->select(
          oclIsKindOf(Action))->forAll(a |
            a.oclIsKindOf(CallOperationAction) and
            self.part->any(name = a.inPartition->any(true).name).type
              .oclAsType(Interface).ownedOperation->
            includes(a.oclAsType(CallOperationAction).operation))
```

OCL Constraint 10 *Corresponding secure object nodes must reside in different partitions.*[12]

```
context SecureObjectFlows::Services::SecureNode
def:    allPredecessors(node : ActivityNode) : Set(ActivityNode) =
          node.incoming.source->collect(x |
            allPredecessors(x))->asSet()->union(node.incoming.source)
inv:    allPredecessors(self)->select(incoming->isEmpty() and
        oclIsKindOf(SecureNode))->forAll(s | s.inPartition <> self
              .inPartition)
```

OCL Constraint 11 *All activity nodes must be assigned to and must be contained by exactly one and only one activity partition.*

```
context SecureObjectFlows::Services::SecureInterface
inv:    self.ownedBehavior.oclAsType(Activity).node->forAll(
          inPartition->size() = 1)
```

OCL Constraint 12 *Only* `InputPins`*,* `OutputPins`*, and* `ActivityParameterNodes` *can be secured. All secured* `InputPins` *must have an incoming object flow; all secured* `OutputPins` *must have an outgoing object flow. Secured* `ActivityParameterNodes` *must either be connected to an incoming object flow, to an outgoing object flow, or to both; depending on the parameter direction.*

```
context SecureObjectFlows::Services::SecureNode
inv:    self.oclIsKindOf(InputPin) or self.oclIsKindOf(OutputPin)
          or self.oclIsKindOf(ActivityParameterNode)
inv:    self.oclIsKindOf(InputPin) implies self.incoming->notEmpty
          ()
inv:    self.oclIsKindOf(OutputPin) implies self.outgoing->
          notEmpty()
inv:    self.oclIsKindOf(ActivityParameterNode) implies
          (self.oclAsType(ActivityParameterNode).parameter.
              direction = ParameterDirectionKind::in or
          self.oclAsType(ActivityParameterNode).parameter.direction
              = ParameterDirectionKind::inout implies
            self.incoming->notEmpty()) and
          (self.oclAsType(ActivityParameterNode).parameter.
              direction = ParameterDirectionKind::out or
          self.oclAsType(ActivityParameterNode).parameter.direction
              = ParameterDirectionKind::inout or
          self.oclAsType(ActivityParameterNode).parameter.direction
              = ParameterDirectionKind::return implies
            self.outgoing->notEmpty())
```

OCL Constraint 13 *All* `ActivityParameterNodes` *which are not initial or final nodes in a control and data flow but counterparts of intermediary* `InputPins` *and* `OutputPins` *must refer to a streaming* `Parameter`*.*[13]

```
context SecureObjectFlows::Services::SecureInterface
def:    isFirstNode(a : ActivityNode) : Boolean =
          a.owner.oclAsType(Activity).node->select(oclIsKindOf(
              InitialNode))->exists(outgoing.target->any(true) =
              a) or
          a.owner.oclAsType(Activity).node->select(oclIsKindOf(
              ActivityNode) and incoming->isEmpty())->includes(a)
def:    isLastNode(a : ActivityNode) : Boolean =
          a.owner.oclAsType(Activity).node->select(oclIsKindOf(
              ActivityFinalNode))->exists(incoming.source->any(
              true) = a) or
          a.owner.oclAsType(Activity).node->select(oclIsKindOf(
              ActivityNode) and outgoing->isEmpty())->includes(a)
def:    allSuccessors(node : ActivityNode) : Set(ActivityNode) =
        node.outgoing.target->collect(x |
          allSuccessors(x))->asSet()->union(node.outgoing.target)
inv:    self.ownedBehavior.oclAsType(Activity).node->select(
        oclIsKindOf(ActivityNode))->forAll(an |
          (not isFirstNode(an) implies
            an.input->forAll(ipin | allPredecessors(ipin)->select(
                oclIsKindOf(ActivityParameterNode))->forAll(
              oclAsType(ActivityParameterNode).parameter.isStream))
              ) and
          (not isLastNode(an) implies
            an.output->forAll(opin | allSuccessors(opin)->select(
                oclIsKindOf(ActivityParameterNode))->forAll(
              oclAsType(ActivityParameterNode).parameter.isStream))
              ))
```

OCL Constraint 14 *All source object nodes of a set of* `InputPins` *owned by a* `CallOperationAction` *must be assigned to the same activity partition.*

```
context UML::ObjectNode
inv:    self.activity.owner.oclIsKindOf(SecureInterface) and
        self.oclIsKindOf(InputPin) implies
          self.oclAsType(InputPin).owner.oclAsType(
              CallOperationAction).input->forAll(ipin |
            allPredecessors(ipin)->select(incoming->isEmpty() and
                oclIsKindOf(ObjectNode))->forAll(on1,on2 |
              on1.inPartition = on2.inPartition))
```

OCL Constraint 15 *If provided for a* `Participant`*, the* `ServiceActivityNode` *must contain a corresponding and compatible* `SecureSendPin` *for each secured* `InputPin` *in a choreography activity; provided that a) there is a choreography activity in*

[11] Here, an OCL 2.3.1 compliant definition is omitted. For an OCL 2.3.1 compliant definition of `allPredecessors()` see OCL Constraint 6b in Appendix A.

[12] Here, an OCL 2.3.1 compliant definition is omitted. For an OCL 2.3.1 compliant definition of `allPredecessors()` see OCL Constraint 6b in Appendix A.

[13] Here, an OCL 2.3.1 compliant definition is omitted. For an OCL 2.3.1 compliant definition of `allSuccessors()` see OCL Constraint 4b in Appendix A.

the first place, and that b) the CallOperationAction *owning the* InputPin *and the* ServiceInteractionAction *owning the* SecureSendPin *share the* Operation *(required from the same* Interface*).*

```
context UML4SOA::ServiceActivityNode
inv:     self.node->select((oclIsKindOf(ServiceSendAction) or
       oclIsKindOf(ServiceReplyAction)) and
         oclAsType(ServiceInteractionAction).input->select(
              oclIsKindOf(SecureSendPin)))->forAll(sa |
          let p : Port =
            if oclIsKindOf(ServiceSendAction)
            then sa.oclAsType(ServiceSendAction).target.oclAsType(
                  Port)
            else sa.oclAsType(ServiceReplyAction).returnInformation
                  .oclAsType(Port)
            endif
          in
            p.type.oclAsType(SecureInterface) and
            p.type.oclAsType(SecureInterface).ownedBehavior.
                  oclAsType(Activity).node->select(
              let op : Operation =
                if oclIsKindOf(ServiceSendAction)
                then sa.oclAsType(ServiceSendAction).operation
                else sa.oclAsType(ServiceReplyAction).replyToCall.
                      oclAsType(CallEvent).operation
                endif
              in
                oclIsKindOf(CallOperationAction) and
                oclAsType(CallOperationAction).operation = op)->
                      forAll(
                  oclAsType(CallOperationAction).input->select(
                        oclIsKindOf(SecureNode))->forAll(i |
                    sa.input->exists(
                      name = i.name and
                      type = i.type and
                      (i.oclAsType(SecureNode).integrityEnsured =
                            oclAsType(SecureNode).integrityEnsured)
                             implies
                        i.oclAsType(SecureNode).integrityAlgorithm
                              = oclAsType(SecureNode).
                              integrityAlgorithm and
                      (i.oclAsType(SecureNode).
                            confidentialityEnsured = oclAsType(
                            SecureNode).confidentialityEnsured)
                            implies
                        i.oclAsType(SecureNode).
                              confidentialityAlgorithm = oclAsType(
                              SecureNode).confidentialityAlgorithm)
                              )))
```

OCL Constraint 16 *If provided for a* Participant, *the* ServiceActivityNode *must contain a corresponding and compatible* SecureReceivePin *for each secured* OutputPin *in a choreography activity; provided that a) there is a choreography activity in the first place, and that b) the* CallOperationAction *owning the* OutputPin *and the* ServiceReceiveAction *owning the* SecureReceivePin *share the* Operation *(required from the same* Interface*).*

```
context UML4SOA::ServiceActivityNode
inv:     self.node->select(oclIsKindOf(ServiceReceiveAction) and
         oclAsType(ServiceReceiveAction).output->select(oclIsKindOf
              (SecureReceivePin)))->forAll(sa |
          sa.oclAsType(ServiceReceiveAction).returnInformation.
                oclAsType(Port).type.oclAsType(SecureInterface) and
          sa.oclAsType(ServiceReceiveAction).returnInformation.
                oclAsType(Port).type.oclAsType(SecureInterface).
                ownedBehavior.oclAsType(
            Activity).node->select(
              oclIsKindOf(CallOperationAction) and
              oclAsType(CallOperationAction).operation = sa.
                    oclAsType(ServiceReceiveAction).trigger->
                any(true).oclAsType(CallEvent).operation)->forAll(
                  oclAsType(CallOperationAction).input->select(
                        oclIsKindOf(SecureNode))->forAll(i |
                    sa.output->exists(
                      name = i.name and
                      type = i.type and
                      (i.oclAsType(SecureNode).integrityEnsured =
                            oclAsType(SecureNode).integrityEnsured)
                             implies
                        i.oclAsType(SecureNode).integrityAlgorithm
                              = oclAsType(SecureNode).
                              integrityAlgorithm and
                      (i.oclAsType(SecureNode).
                            confidentialityEnsured = oclAsType(
                            SecureNode).confidentialityEnsured)
                            implies
                        i.oclAsType(SecureNode).
                              confidentialityAlgorithm = oclAsType(
                              SecureNode).confidentialityAlgorithm)
                              )))
```

References

1. Apache Software Foundation (ASF): Apache Axis2. http://axis.apache.org/axis2/java/core/ (2012)
2. Apache Software Foundation (ASF): Apache ODE. http://ode.apache.org (2012)
3. Apache Software Foundation (ASF): Apache Rampart—Axis2 Security Module. http://axis.apache.org/axis2/java/rampart/ (2012)
4. Axenath, B., Kindler, E., Rubin, V.: AMFIBIA: a meta-model for the integration of business process modelling aspects. In: Leymann, F., Reisig, W., Thatte, S., van der Aalst, W. (eds.) The Role of Business Processes in Service Oriented Architectures, Dagstuhl Seminar Proceedings (2006)
5. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: from UML models to access control infrastructures. ACM Transact. Softw. Eng. Methodol. (TOSEM) **15**(1), 39–91 (2006)
6. Baumgrass, A., Baier, T., Mendling, J., Strembeck, M.: Conformance checking of RBAC policies in process-aware information systems. In: Proceedings of the Workshop on Workflow Security Audit and Certification (WfSAC), Lecture Notes in Business Information Processing (LNBIP), vol. 100. Springer, Berlin (2011)
7. Cannon, J., Byers, M.: Compliance deconstructed. ACM Queue **4**(7), 30–37 (2006)
8. Committee on National Security Systems (CNSS): National Information Assurance (IA): glossary. http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf (2010)
9. Damianides, M.: How does SOX change IT? J. Corp. Account. Finance **15**(6), 35–41 (2004)
10. Eclipse Foundation: Eclipse IDE. http://www.eclipse.org (2012)
11. Eclipse Foundation: Eclipse model development tools (MDT). http://www.eclipse.org/modeling/mdt/ (2012)
12. Eclipse Foundation: Eclipse Papyrus. http://www.eclipse.org/modeling/mdt/papyrus/ (2012)
13. Elvesæter, B., Berre, A.-J., Sadovykh, A.: Specifying services using the service oriented architecture modeling language (SoaML)—a baseline for specification of cloud-based services. In: Proceedings of the 1st International Conference on Cloud Computing and Services Science (CLOSER'11), pp. 276–285. SciTePress (2011)
14. Elvesæter, B., Carrez, C., Mohagheghi, P., Berre, A.-J., Johnsen, S., Solberg, A.: Model-driven service engineering with SoaML. In: Service Engineering—European Research Results, pp. 25–54. Springer, Berlin (2011)
15. Fink, T., Koch, M., Pauls, K.: An MDA approach to access control specifications using MOF and UML profiles. In: Electronic Notes in Theoretical Computer Science, pp. 161–179 (2006)
16. International Organization for Standardization (ISO): Information technology: security techniques—code of practice for information security management, ISO/IEC 27002:2005, Stage: 90.92. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50297 (2008)
17. International Organization for Standardization (ISO): Information technology: security techniques—information security management systems—requirements, ISO/IEC 27001:2005, Stage: 90.92. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=42103 (2008)
18. International Organization for Standardization (ISO): Information technology—security techniques—information security management systems—overview and vocabulary, ISO/IEC 27000:2009, Stage: 60.60. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=41933 (2009)
19. Foster, H., Gönczy, L., Koch, N., Mayer, P., Montangero, C., Varró, D. UML extensions for service-oriented systems. In: Wirsing, M., Hölzl, M. (eds.) Rigorous Software Engineering for Service-

Oriented Systems, Lecture Notes in Computer Science (LNCS), pp. 35–60. Springer, Berlin (2011)
20. Gilmore, S., Gönczy, L., Koch, N., Mayer, P., Tribastone, M., Varró, D.: Non-functional properties in the model-driven development of service-oriented systems. Softw. Syst. Model. **10**(3), 287–311 (2011)
21. Hafner, M., Alam, M., Breu, R.: Towards a MOF/QVT-based domain architecture for model driven security. In: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MODELS 2006), Lecture Notes in Computer Science (LNCS), pp. 275–290. Springer, Berlin (2006)
22. Hafner, M., Breu, R.: Security Engineering for Service-Oriented Architectures, 1st edn. Springer, Berlin (2009)
23. Hafner, M., Breu, R., Agreiter, B., Nowak, A.: SECTET: an extensible framework for the realization of secure inter-organizational workflows. Internet Res. **16**(5), 491–506 (2006)
24. Hafner, M., Memon, M., Alam, M.: Modeling and enforcing advanced access control policies in healthcare systems with SECTET. In: Giese, H. (ed.) Models in Software Engineering, pp. 132–144. Springer, Berlin (2008)
25. Hentrich, C., Zdun, U.: A pattern language for process execution and integration design in service-oriented architectures. In: Noble, J., Johnson, R. (eds.) Transactions on Pattern Languages of Programming I, Lecture Notes in Computer Science (LNCS), pp. 136–191. Springer, Berlin (2009)
26. Hoisl, B., Sobernig, S.: Integrity and confidentiality annotations for service interfaces in SoaML models. In: Proceedings of the International Workshop on Security Aspects of Process-aware Information Systems (SAPAIS2011), pp. 673–679. IEEE (2011)
27. Hoisl, B., Strembeck, M.: Modeling support for confidentiality and integrity of object flows in activity models. In: Proceedings of the 14th International Conference on Business Information Systems (BIS2011), Lecture Notes in Business Information Processing (LNBIP), pp. 278–289. Springer, Berlin (2011)
28. Hoisl, B., Strembeck, M.: A UML extension for the model-driven specification of audit rules. In: Proceedings of the 2nd International Workshop on Information Systems Security Engineering (WISSE), Lecture Notes in Business Information Processing (LNBIP). Springer, Berlin (2012)
29. Huhns, M., Singh, M.: Service-oriented computing: key concepts and principles. IEEE Internet Comput. **9**, 75–81 (2005)
30. Hummer, W., Gaubatz, P., Strembeck, M., Zdun, U., Dustdar, S.: An integrated approach for identity and access management in a SOA context. In: Proceedings of the 16th ACM Symposium on Access Control Models and Technologies (SACMAT) (2011)
31. Jensen, M., Feja, S.: A security modeling approach for web-service-based business processes. In: Proceedings of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, pp. 340–347. IEEE (2009)
32. Jürjens, J.: UMLsec: extending UML for secure systems development. In: Proceedings of the 5th International Conference on The Unified Modeling Language, pp. 412–425. Springer, Berlin (2002)
33. Jürjens, J.: Secure Systems Development with UML. Springer, Berlin (2005)
34. Kim, S., Burger, D., Carrington, D.: An MDA approach towards integrating formal and informal modeling languages. In: Proceedings of the International Symposium of Formal Methods Europe, Lecture Notes in Computer Science (LNCS), vol. 3582, pp. 448–464. Springer, Berlin (2005)
35. Kopp, O., Martin, D., Wutke, D., Leymann, F.: The difference between graph-based and block-structured business process modelling languages. Enterp. Model. Inf. Syst. **4**(1), 3–13 (2009)
36. Mayer, P.: Model-driven development for service-oriented computing—transformers. http://mdd4soa.eu/transformers/ (2008)
37. Mayer, P.: MDD4SOA—model-driven development for service-oriented architectures. PhD thesis, Ludwig Maximilian University of Munich, Faculty of Mathematics, Computer Science and Statistics (2010)
38. Mayer, P., Koch, N., Schröder, A., Knapp, A.: The UML4SOA profile. http://www.uml4soa.eu/wp-content/uploads/uml4soa.pdf (2010)
39. Mayer, P., Schröder, A., Koch, N.: MDD4SOA: model-driven service orchestration. In: Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 203–212. IEEE (2008)
40. Memon, M., Hafner, M., Breu, R.: SECTISSIMO: a platform-independent framework for security services. In: Proceedings of the Modeling Security Workshop in Association with MODELS 2008 (2008)
41. Mendling, J., Lassen, K., Zdun, U.: On the transformation of control flow between block-oriented and graph-oriented process modeling languages. Int. J. Business Process Integr. Manag. **3**(2), 96–108 (2008)
42. Mens, T., van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
43. Mishra, S., Weistroffer, H.: A framework for integrating Sarbanes-Oxley compliance into the systems development process. Commun. Assoc. Inf. Systems (CAIS) **20**(1), 712–727 (2007)
44. Nakamura Y., Tatsubori M., Imamura T., Ono K.: Model-driven security based on a web services security architecture. In: Proceedings of the IEEE International Conference on Services Computing, pp. 7–15. IEEE (2005)
45. National Institute of Standards and Technology (NIST): An Introduction to Computer Security: The NIST Handbook. Special Publication 800–12. http://csrc.nist.gov/publications/nistpubs/800-12/handbook.pdf (1995)
46. National Institute of Standards and Technology (NIST): Data Encryption Standard (DES). Federal Information Processing Standards Publication 46–3. http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf (1999)
47. National Institute of Standards and Technology (NIST): Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (2001)
48. National Institute of Standards and Technology (NIST): Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180–3. http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf (2008)
49. National Institute of Standards and Technology (NIST): Recommended Security Controls for Federal Information Systems and Organizations. NIST Special Publication 800–53, Revision 3. http://csrc.nist.gov/publications/nistpubs/800-53-Rev3/sp800-53-rev3-final_updated-errata_05-01-2010.pdf (2009)
50. National Security Agency (NSA): Information assurance technical framework. http://handle.dtic.mil/100.2/ADA393328 (2000)
51. No Magic, Inc.: MacigDraw. https://www.magicdraw.com (2012)
52. Object Management Group: OMG Business Process Model and Notation (BPMN) Specification, Version 2.0, formal/2011-01-03. http://www.omg.org/spec/BPMN (2011)
53. Object Management Group: OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, formal/2011-08-07. http://www.omg.org/mof (2011)
54. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, formal/2011-01-01. http://www.omg.org/spec/QVT (2011)
55. Object Management Group: OMG MOF 2 XMI Mapping Specification, Version 2.4.1, formal/2011-08-09. http://www.omg.org/spec/XMI (2011)

56. Object Management Group: OMG Object Constraint Language (OCL) Specification, Version 2.2, formal/2010-02-01. http://www.omg.org/spec/OCL (2010)
57. Object Management Group: OMG Object Constraint Language (OCL) Specification, Version 2.3.1, formal/2012-01-01. http://www.omg.org/spec/OCL (2012)
58. Object Management Group: OMG Service oriented architecture Modeling Language (SoaML) Specification, Version 1.0 Beta 2, ptc/2009-12-09. http://www.omg.org/spec/SoaML (2009)
59. Object Management Group: OMG Unified Modeling Language (OMG UML): superstructure, Version 2.4.1, formal/2011-08-06. http://www.omg.org/spec/UML (2011)
60. Object Management Group: OMG Unified Modeling Language (OMG UML): infrastructure, Version 2.4.1, formal/2011-08-05. http://www.omg.org/spec/UML (2011)
61. Organization for the Advancement of Structured Information Standards (OASIS): Web Services Business Process Execution Language, Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf (2007)
62. Organization for the Advancement of Structured Information Standards (OASIS): Reference Architecture Foundation for Service Oriented Architecture, Version 1.0. http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-cd-02.pdf (2009)
63. Organization for the Advancement of Structured Information Standards (OASIS): WS-SecurityPolicy 1.3. http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/os/ws-securitypolicy-1.3-spec-os.pdf (2009)
64. Papazoglou, M., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: state of the art and research challenges. IEEE Comput. **40**, 38–45 (2007)
65. Reznik, J., Ritter, T., Schreiner, R., Lang, U.: Model driven development of security aspects. Electron. Notes Theo. Comput. Sci. **163**, 65–79 (2007)
66. Rodríguez, A., Fernández-Medina, E., Trujillo, J., Piattini, M.: Secure business process model specification through a UML 2.0 activity diagram profile. Decis. Support Syst. **51**(3), 446–465 (2011)
67. Rodríguez, A., García-Rodríguez de Guzmán, I., Fernández-Medina, E., Piattini, M.: Semi-formal transformation of secure business processes into analysis class and use case models: an MDA approach. Inform. Softw. Technol. **52**, 945–971 (2010)
68. Sánchez, Ó., Molina, F., García-Molina, J., Toval, A.: ModelSec: a generative architecture for model-driven security. J. Univ. Comput. Sci. **15**(15), 2957–2980 (2009)
69. Sandhu, R.: On five definitions of data integrity. In: Proceedings of the IFIP WG11.3 Working Conference on Database Security VII (1993)
70. Scheer, A.-W.: ARIS: Business Process Modeling. Springer, Berlin (2000)
71. Schefer, S., Strembeck, M.: Modeling process-related duties with extended UML activity and interaction diagrams. In: Proceedings of the International Workshop on Flexible Workflows in Distributed Systems, Electronic Communications of the EASST (2011)
72. Schefer, S., Strembeck, M.: Modeling support for delegating roles, tasks, and duties in a process-related RBAC context. In: Proceedings of the International Workshop on Information Systems Security Engineering (WISSE), Lecture Notes in Business Information Processing (LNBIP), vol. 83. Springer, Berlin (2011)
73. Schefer, S., Strembeck, M., Mendling, J.: Checking satisfiability aspects of binding constraints in a business process context. In: Proceedings of the Workshop on Workflow Security Audit and Certification (WfSAC), Lecture Notes in Business Information Processing (LNBIP), vol. 100. Springer, Berlin (2011)
74. Schefer, S., Strembeck, M., Mendling, J., Baumgrass, A.: Detecting and resolving conflicts of mutual-exclusion and binding constraints in a business process context. In: Proceedings of the 19th International Conference on Cooperative Information Systems (CoopIS), Lecture Notes in Computer Science (LNCS), vol. 7044. Springer, Berlin (2011)
75. Schefer-Wenzl, S., Strembeck, M.: An approach for consistent delegation in process-aware information systems. In: Proceedings of the 15th International Conference on Business Information Systems (BIS), Lecture Notes in Business Information Processing (LNBIP). Springer, Berlin (2012)
76. Schefer-Wenzl, S., Strembeck, M.: Modeling context-aware RBAC models for business processes in ubiquitous computing environments. In: Proceedings of the 3rd International Conference on Mobile, Ubiquitous and Intelligent Computing (MUSIC) (2012)
77. Schmidt, D.: Model-driven engineering: guest editor's introduction. IEEE Comput. **39**(2), 25–31 (2006)
78. Schmidt, H., Jürjens, J.: Connecting security requirements analysis and secure design using patterns and UMLsec. In: Proceedings of the 23rd International Conference on Advanced Information Systems Engineering (CAiSE), Lecture Notes in Computer Science (LNCS), pp. 367–382. Springer, Berlin (2011)
79. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003)
80. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
81. Sobernig, S., Zdun, U.: Invocation assembly lines: patterns of invocation and message processing in object remoting middleware. In: Kelly, A., Weiss, M. (eds.) Proceedings of 14th Annual European Conference on Pattern Languages of Programming (EuroPLoP 2009), CEUR-WS.org, vol. 566. (2009)
82. Stahl, T., Völter, M.: Model-Driven Software Development. Wiley, New York (2006)
83. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley, Boston (2008)
84. Strembeck, M., Mendling, J.: Generic algorithms for consistency checking of mutual-exclusion and binding constraints in a business process context. In: Proceedings of the 18th International Conference on Cooperative Information Systems (CoopIS), Lecture Notes in Computer Science (LNCS), vol. 6426. Springer, Berlin (2010)
85. Strembeck, M., Mendling, J.: Modeling process-related RBAC models with extended UML activity models. Inform. Softw. Technol. **53**(5), 456–483 (2011)
86. Tatsubori, M., Imamura, T., Nakamura, Y.: Best-practice patterns and tool support for configuring secure web services messaging. In: Proceedings of the IEEE International Conference on Web Services, pp. 244–251. IEEE (2004)
87. Warner, J., Atluri, V.: Inter-instance authorization constraints for secure workflow management. In: Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT) (2006)
88. Wenzel, S.: CARiSMA. http://vm4a003.itmc.tu-dortmund.de/carisma/web/doku.php (2012)
89. Wolter, C., Menzel, M., Meinel, C.: Modelling security goals in business processes. In Modellierung 2008, Lecture Notes in Informatics (LNI), pp. 197–212 (2008)
90. Wolter, C., Menzel, M., Schaad, A., Miseldine, P., Meinel, C.: Model-driven business process security requirement specification. J. Systems Archit. **55**(4), 211–223 (2009)
91. Wolter, C., Schaad, A.: Modeling of task-based authorization constraints in BPMN. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) Proceedings of the 5th International Conference on Business Process Management (BPM), volume 4714 of Lecture Notes in Computer Science (LNCS), pp. 64–79. Springer, Berlin (2007)
92. World Wide Web Consortium (W3C): Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl (2001)
93. World Wide Web Consortium (W3C): Web Services Policy 1.5, Attachment. http://www.w3.org/TR/ws-policy-attach/ (2007)

94. World Wide Web Consortium (W3C): Web Services Policy 1.5, Framework. http://www.w3.org/TR/ws-policy/ (2007)
95. Zdun, U.: Patterns of component and language integration. In: Manolescu, D., Völter, M., Noble, J. (eds.) Pattern Languages of Program Design 5 (2006)
96. Zdun, U., Dustdar, S.: Model-driven and pattern-based integration of process-driven SOA models. Int. J. Business Process Integr. Manag. (IJBPIM) **2**(2), 109–119 (2007)
97. Zdun, U., Hentrich, C., Dustdar, S.: Modeling process-driven and service-oriented architectures using patterns and pattern primitives. ACM Transact. Web **1**(3), 14:1–14:44 (2007)

Author Biographies

Bernhard Hoisl holds a MSc degree in Information Systems from the Vienna University of Economics and Business (WU Vienna) and a MSc degree in Computer Science Management from the Vienna University of Technology. Currently, he works at the New Media Lab, Institute for Information Systems, WU Vienna on a dissertation fellowship project. Since April 2010 he is also part-time researcher at Secure Business Austria (SBA). His PhD research interests are focused on model-driven software development, domain-specific languages, and process modeling with a special emphasis on the specification of security aspects.

Stefan Sobernig is a Postdoc Researcher and Lecturer at the New Media Lab, Institute for Information Systems, Vienna University of Economics and Business (WU Vienna). In his research, he works in the fields of feature-oriented software development, domain-specific language engineering, software patterns, and communication middleware.

Mark Strembeck is an Associate Professor of Information Systems at the Vienna University of Economics and Business (WU Vienna), Austria. His research interests include secure business systems, business process management, model-driven software development, and the modeling and management of dynamic software systems. He received his doctoral degree as well as his Habilitation degree (venia docendi) from WU Vienna. He is a key researcher at Secure Business Austria (SBA), and the Vice Institute Head of the Institute for Information Systems at WU Vienna.

A.8 Comparing Three Notations for Defining Scenario-based Model Tests: A Controlled Experiment

The following paper was published as:

B. Hoisl, S. Sobernig, and M. Strembeck. Comparing Three Notations for Defining Scenario-based Model Tests: A Controlled Experiment. In *Proceedings of the 9th International Conference on the Quality of Information and Communications Technology*, pages 95–104. IEEE, 2014 (see [55]).

2014 9th International Conference on the Quality of Information and Communications Technology

Comparing Three Notations for Defining Scenario-based Model Tests: A Controlled Experiment

Bernhard Hoisl*†, Stefan Sobernig*, 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*—Scenarios are an established means to specify requirements for software systems. Scenario-based tests allow for validating software models against such requirements. In this paper, we consider three alternative notations to define such scenario tests on structural models: a semi-structured natural-language notation, a diagrammatic notation, and a fully-structured textual notation. In particular, we performed a study to understand how these three notations compare to each other with respect to accuracy and effort of comprehending scenario-test definitions, as well as with respect to the detection of errors in the models under test. 20 software professionals (software engineers, testers, researchers) participated in a controlled experiment based on six different comprehension and maintenance tasks. For each of these tasks, questions on a scenario-test definition and on a model under test had to be answered. In an ex-post questionnaire, the participants rated each notation on a number of dimensions (e.g., practicality or scalability). Our results show that the choice of a specific scenario-test notation can affect the productivity (in terms of correctness and time-effort) when testing software models for requirements conformance. In particular, the participants of our study spent comparatively less time and completed the tasks more accurately when using the natural-language notation compared to the other two notations. Moreover, the participants of our study explicitly expressed their preference for the natural-language notation.**

I. Introduction

In model-driven development (MDD), models are used to define a software system's problem domain as well as corresponding solutions. Thereby, models abstract from the underlying implementation technology (see, e.g., [1], [2]). In this context, models are not only used for documentation purposes, but are first-class artifacts from which executable code is generated. Consequently, the quality of a software product considerably relies on the quality of the corresponding models. To ensure such high quality models, the predominantly code-centric testing strategies must be complemented with approaches for testing at the model-level. Such model-level tests help detect errors at an early development stage (see, e.g., [3], [4]).

In recent years, scenarios have become a popular means for capturing a software system's requirements (see, e.g., [5]–[7]). In MDD, scenario specifications can be used to test models for compliance with corresponding domain requirements (see, e.g., [3], [4], [8]). Different approaches have been proposed to provide notations for specifying scenarios, for example in a table-based layout, as message sequence charts, or via formal methods (see, e.g., [7], [9], [10]). However, no generally accepted standard notation exists and the different scenario notations vary with respect to the corresponding application domain and with the professional background of the stakeholder involved in a particular development project (see, e.g., [7]). Nevertheless, independent of the visualization technique, three cognitive activities are fundamental to determine the usability and quality of different notations: learnability, understandability, and changeability (see, e.g., [11]). These aspects have a significant influence on the productivity (see, e.g., [12]): the costs increase the longer a domain expert or software engineer needs to understand a scenario and the less software errors are corrected. Hence, by ensuring easy-to-understand scenario tests, the software-development costs can be decreased.

In this paper, we report on a controlled experiment for the comparison of three frequently used alternatives for scenario notations (see, e.g., [7]) to test Ecore-based structural models[1]. In particular, we compare a semi-structured natural-language notation, a diagrammatic notation, and a fully-structured textual notation. As the comprehension of a notation is a cognitive process, it cannot be observed directly (see [13]). Therefore, we compare the three notations with respect to the accuracy and the required effort for comprehending scenario-test definitions, as well as with respect to the detection of errors in the models under test (MUT). The evaluation consists of comprehension (learnability, understandability) and maintenance (changeability) tasks (see, e.g., [14]). As concrete measures, we use the response time and the correctness of a solution (see, e.g., [12]). We report our experiment according to the guidelines for empirical software studies defined by Jedlitschka et al. [15] and by Kitchenham et al. [16].

The remainder of this paper is structured as follows. In Section II, we describe the three notations that we compare in our study. Subsequently, Section III gives an overview of related work, before Section IV explains the planning and design of the experiment. Section V describes the execution of the experiment and Section VI provides an analysis of the study's results. Next, Sections VII and VIII interpret the results and discuss threats to validity. Section IX concludes the paper and points to future work. In the Appendix, we showcase a

[1]All materials used in the experiment as well as collected data, calculations, and computed results can be downloaded from http://nm.wu.ac.at/modsec.

978-1-4799-6133-7/14 $31.00 © 2014 IEEE
DOI 10.1109/QUATIC.2014.19

complete task actually performed during the experiment.

II. Three Notations for Scenario-based Testing

In this section, we provide an overview of the three scenario notations. In particular, we show an example how the same test scenario is described in each of the three notations. The example is taken from the introductory tutorial given to the participants of our study at the beginning of each experiment session.

For the experiment, each scenario test consists of three parts: (1) preconditions setting up the scenario context, (2) events triggering an action, and (3) expected outcomes of the action (see, e.g., [7], [17]). All three parts of a scenario test must evaluate to true in order to make a scenario test pass as a whole. In the example, and as required by the design of our experiment, the MUT is an Ecore model[2].

N-notation: The semi-structured natural-language notation is adopted from Cucumber's set of syntax rules for scenario tests (called Gherkin; see [17]). The natural-language statements follow a declarative sentence structure. In Listing 1, line 1 initializes a new scenario (`Scenario`). The keywords `Given`, `When`, and `Then` define the start of a precondition, an event, and an expected outcome, respectively (lines 2–4). This natural language scenario definition can be automatically transformed into executable test scenarios which can be processed by the framework presented in [19]. The scenario test passes iff all test expressions are fulfilled, otherwise it fails.

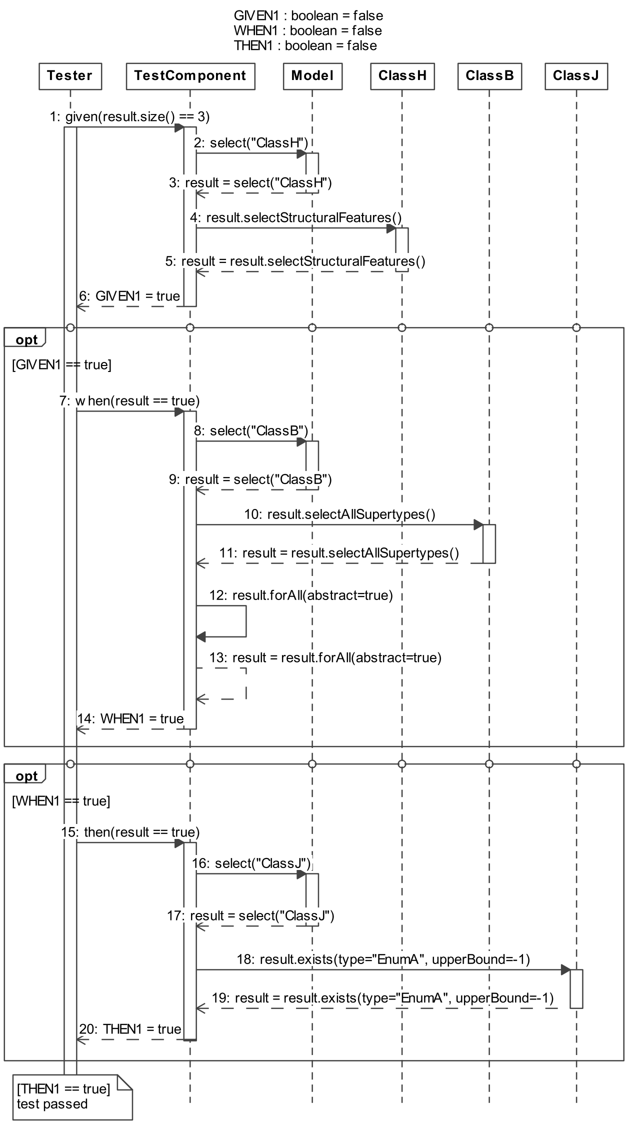
```
1 Scenario:
2   Given "that ClassH owns exactly three structural features"
3   When "all supertype classes of ClassB are abstract"
4   Then "in ClassJ there shall be at least one attribute of type EnumA
          with an upper bound multiplicity of -1"
```

Listing 1. Semi-structured natural-language scenario notation (N-notation) exemplified.

D-notation: The second scenario-test notation is based on UML sequence diagrams [20]. It is inspired by related model-based scenario-test approaches (see, e.g., [9], [21]). Fig. 1 shows the example scenario test defined via this diagrammatic representation—it is semantically equivalent to the natural-language representation in Listing 1. Lifelines represent instances of a test runner, a test component, the MUT, and classes contained in the MUT which collaborate to realize the tested scenario. Interactions are shown as sequences of call and return messages between the corresponding instances on the lifelines. Events and expected outcomes are declared optional (see the `CombinedFragments` with `opt InteractionOperatorKind` in Fig. 1 [20]). The different scenario-test expressions are nested, so that an event can only be triggered iff the precondition(s) are fulfilled (see `InteractionConstraint` guards of `Combined-Fragments` in Fig. 1). In the same way, an expected outcome is only tested iff all preconditions and events are fulfilled.

E-notation: The third notation defines scenario tests through an extension to a fully-structured language for model management tasks (see Listing 2), called Epsilon [22]. In Epsilon, model testing is supported by the Epsilon unit testing

[2]In this paper, we do not provide a background on Ecore models; for the remainder, it is sufficient to refer to generic object-oriented modeling constructs such as classes, inheritance relationships between classes, references, and attributes. For details on Ecore, see, e.g., Steinberg et al. [18].



GIVEN1 : boolean = false
WHEN1 : boolean = false
THEN1 : boolean = false
Tester
TestComponent
Model
ClassH
ClassB
ClassJ
1: given(result.size() == 3)
2: select("ClassH")
3: result = select("ClassH")
4: result.selectStructuralFeatures()
5: result = result.selectStructuralFeatures()
6: GIVEN1 = true
opt
[GIVEN1 == true]
7: when(result == true)
8: select("ClassB")
9: result = select("ClassB")
10: result.selectAllSupertypes()
11: result = result.selectAllSupertypes()
12: result.forAll(abstract=true)
13: result = result.forAll(abstract=true)
14: WHEN1 = true
opt
[WHEN1 == true]
15: then(result == true)
16: select("ClassJ")
17: result = select("ClassJ")
18: result.exists(type="EnumA", upperBound=-1)
19: result = result.exists(type="EnumA", upperBound=-1)
20: THEN1 = true
[THEN1 == true]
test passed

Fig. 1. Diagrammatic scenario notation (D-notation) exemplified.

framework (EUnit) which we extended to define scenario tests (see [4]). With this notation, scenario tests are imperatively defined via textual sequences of commands. Listing 2 shows our scenario-test example (again, semantically equivalent to Listing 1 and Fig. 1) including a precondition (line 6), an event (lines 8–10), and an expected outcome (line 11). Epsilon shares similarities with OCL [23]. For example, regarding operations on collections, a universal quantification can be expressed identically (`forAll()`; see line 9 in Listing 2).

```
1 @TestSuite
2 operation testSuite() {
3   @TestCase
4   operation testCase() {
5     @TestScenario
6     $pre Model!EClass.all().selectOne(x|x.name="ClassH").
            eStructuralFeatures.size() = 3
7     operation testScenario() {
8       if (
9         Model!EClass.all().selectOne(x|x.name="ClassB").closure(x|x.
            eSuperTypes).forAll(x|x.abstract=true)
        ) {
          assertTrue(Model!EClass.all().selectOne(x|x.name="ClassJ").
              eStructuralFeatures.exists(x|x.isTypeOf(EAttribute) and x.
              eType.name="EnumA" and x.upperBound=-1));
        } else {
          assertTrue(false);
        }
      }
    }
}
```

Listing 2. Fully-structured textual scenario notation (E-notation) exemplified.

III. Related Work

For this paper, we consider two categories of related work: (1) studies regarding the comprehension of the three notation types used in our experiment and (2) (empirical) evaluations of requirements-driven scenario notations.

Our first notation uses a text-based format that closely resembles natural-language expressions. Studies to measure text comprehension via problem solving tasks—comparable to the tasks performed in the experiment reported in this paper—have a long history (see, e.g., [24]). Emerging from this background, program comprehension experiments try to measure factors influencing the understandability of source code, such as scenario tests defined via EUnit. For instance, Kosar et al. [11] compare the accuracy and efficiency of domain-specific languages with general-purpose languages and Feigenspan et al. [12] evaluate the improved correctness in the development of software-product lines by introducing background-color-enhanced preprocessor directives. In both studies, a family of experiments was conducted in which participants had to perform comprehension-related tasks. As an example of a model-based notation, Mendling et al. [25] examine factors influencing how model viewers comprehend the syntactical information in process models (UML sequence diagrams fall in this category) in a four-part series of experiments. All related studies have in common that comprehension-influencing factors (i.e. dependent variables) are measured via response time and correctness of a solution.

As a representative for a non-empirical evaluation of requirements-driven scenario notations, Amyot and Eberlein [10] define evaluation criteria (e.g. decomposition and abstraction) and review fifteen scenario notations with respect to their suitability for the telecommunication domain. Ricca et al. [26] performed two controlled experiments to empirically evaluate a table-based notation for defining scenario tests. In particular, they investigate whether the notation contributes to the clarification of domain requirements and whether any additional comprehension effort is required. For requirements elicitation, scenarios are frequently documented via use cases. Gemino and Parker [27] explore the effectiveness of integrating use case models [20] with a set of textual use case descriptions (see, e.g., [7]). A study is conducted including comprehension, retention, and problem solving tasks measuring the level of understanding developed by the participants who either used textual uses cases or textual use cases with a supporting use case diagram. Likewise, Ottensooser et al. [28] report on an experiment for comparing the success of participants in interpreting business process descriptions for a graphical notation (BPMN) and for a textual notation (based on written use cases).

Our study complements existing contributions with a systematic comparison of three notations for scenario-based model testing.

IV. Experiment Planning

A. Goals and Variables

As mentioned above, the objective of our experiment is to compare three notations (see Section II) regarding their applicability for the scenario-based testing of models. For this task, we selected three notations from the body of previously screened literature on model-testing notations (see, e.g., [4], [17], [19], [20], [22]). The MUTs were defined using Ecore, a metamodeling language that provides the foundation of the Eclipse Modeling Framework (EMF; see, e.g., [18]).

Results obtained in related studies indicate that different scenario notations have an influence on the accuracy and effort of fulfilling model-testing tasks. For our experiment, the participants had to solve tasks which were identical except for the alternate usage of different scenario notations (our independent variable). Similar to related studies (see Section III), we use the response time and the correctness of the solution as dependent variables to measure the comprehension of a particular notation. A change in the independent variable is expected to have a measurable effect on one or both dependent variables.

To ensure that the scenario notation is the only independent variable, we had to control important confounding variables. In particular, we focused on maximizing the internal validity of our experimental setting to be able to draw sound conclusions from our results. Prior to the experiment, we asked all participants to fill in a questionnaire regarding their experience with topics that could have an influence on the tasks performed in the experiment (e.g., experience with different testing methods and testing notations relevant to the experiment). In addition, we collected the basic demographic profile of the participants. The process of building homogeneous groups was based on these information to keep the influence of confounding parameters on notation comprehension constant (see Section IV-B).

B. Participants of the Study

For the experiment, we contacted 30 software professionals (software engineers, testers, researchers) from different software companies, universities, and research institutions. For their participation, we offered a small reward in the form of a voucher for a well-known e-commerce company. 24 individuals agreed to contribute to the experiment, the remaining six did not respond to our request. We selected three participants to take part in a pilot study to pre-test the experimental setting (see also [12]). From the remaining 21, one participant could not take part due to illness. As we conducted the experiment at companies in different cities in Austria, we had to exclude the ill participant due to travel requirements. Hence, our data set consisted of 20 participants.

In our study we had five female and 15 male participants with an average age of 30.5±2.95[3] years. Each participant has obtained a computer-science degree or a related academic

[3]The value after ± shows the standard deviation (σ) from the arithmetic mean ($\bar{x}$).

degree and currently works in the IT industry. The experience questionnaire consisted of five demographic and ten experience questions. To measure the level of confounding parameters, participants had to estimate their own experience level on a six-point Likert scale[4]. The ten questions collected evidence about the participants' experiences with regard to: scenario-based testing, Ecore modeling, natural-language testing, model-based testing, and test-case programming. We grouped and classified the questions according to a pre-defined schema (five groups, one to three questions per group). In a discussion with colleagues, we allocated weights to each question in a group. We classified general questions to be less important than specific ones. For example, experience in writing software in any programming language counted less than experience with the Epsilon language or model constraint/validation languages (such as OCL or EVL). The reason for this weighting was that we explicitly used dialects of the Epsilon language for validating models in our experiment. We assume, for instance, that general programming knowledge does not help as much in comprehending EUnit scenario tests as does knowledge in a language specifically designed for model management tasks (e.g. model navigation or element selection).

Our research design required to divide the participants into three groups (see Section IV-E). According to the participants' self estimations on the Likert scale, zero to six points ("no experience" to "very high experience") were allocated per question. Then, questions were grouped and weighted according to the schema explained above. The group building process homogeneously spread the participants so that the combined score of the weighted experience questions in each of the five question groups was similar. After the allocation, each group contained seven participants[5].

Via the experience questionnaire, we informed the participants that the collected data will be used solely for the process of conducting the experiment. Furthermore, we guaranteed that all published results will be anonymized and that personal data is neither made public nor given to any third-parties. In order to ask clarification questions after the experiment, we needed to know the participants' names as well as the name of their employer. As the collected data can be linked to the participants, we consulted the Center of Empirical Research Methods of WU Vienna which approved the ethical correctness of our design. The participants could opt-out of the experiment at any time, but no one did so.

C. Experimental Material

For the experiment, we prepared six different scenarios[6]. For each scenario, we provide an Ecore-based MUT, a scenario description specified in each of the three notations, and questions to be answered (see also the example task in the appendix). All materials were printed and fitted on one A4 sheet respectively. The Ecore models were created using the EMF project of Eclipse 4.2 [18]. The models were taken from real-world examples (e.g., the ATL code generator metamodel, the Ant metamodel), were obtained from the AtlanMod Metamodel Zoo[7], and adapted to fit the experimental setting (in terms of size and naming conventions). The adapted size of the six models were similar to each other and had been chosen to ensure a certain degree of complexity. In particular, it was intended that the participants should frequently consult the models during their tasks and should not be able to memorize their structure easily. Regarding model measures, on average, each model contained 23±1.17 classes, 8±1.03 of them declared abstract. These classes contained on average 44±3.21 structural features consisting of 18±3.51 attributes and 26±2.1 references. 12±1.67 of these references were defined as containment references. In a model, 19±3.14 inheritance relationships between classes were specified on average.

All scenario descriptions included the definition of two preconditions, two events, and two expected outcome specifications. The creation of all scenarios was supported by tools. The natural-language notation was defined via the infrastructure described in Hoisl et al. [19], the fully-structured notation via the infrastructure described in Sobernig et al. [4], and the diagrammatic notation by utilizing the UML sequence diagram editor of No Magic's MagicDraw 17.05. Each line of the two textual scenario descriptions were numbered on the left-hand side (see Listings 1 and 2 in Section II). The UML sequence diagram shows scenario interactions via messages passed between object instances. All call and return messages were sequentially numbered from top to bottom and these message numbers were displayed in front of each message expression (see Fig. 1 in Section II).

The experimental material also included language references for each of the three scenario notations as well as for Ecore models. The language references explained the syntactical format and the semantics of each notation and served as a documentation for the study participants (e.g., for the fully-structured notation, method calls and return values were explained).

For each scenario, five questions were asked with "yes", "no", and "don't know" answer choices. For each question, the possibility to insert line and message numbers was given. At the top of the answer sheet, participants had to insert their start time. An equivalent field was provided at the bottom for the end time. The task, the participants had to carry out, was explained on the sheet. At the bottom, space was reserved for comments.

To correctly answer the questions in the experiment the participants had to understand whether a scenario test passes or fails when elements in the corresponding model are changed. We classified the model changes referred to by the questions according to Wimmer et al. [30]. On the one hand, solutions to the questions required changes to existing modeling concepts: attributes (context, multiplicity, datatype), references (context, multiplicity, direction, containment), and inheritance relationships (concreteness, depth, inheritance type). On the other hand, we also asked for creation and deletion of classes, attributes, references, and inheritance relationships (source-target-concept cardinality differences).

[4]Self estimation is expected to be a reliable way to judge experiences and is recommended, for example, in Siegmund et al. [29].

[5]Note that the group allocation was done before we learned that one of the individuals was not able to participate because of illness.

[6]All materials used in the experiment as well as collected data, calculations, and computed results can be downloaded from http://nm.wu.ac.at/modsec.

[7]http://www.emn.fr/z-info/atlanmod/index.php/Zoos

At the end of the experiment, an ex-post questionnaire had to be filled out. The participants should rate different criteria on a five point scale (1=worst, 5=best) for all three scenario-test notations. In particular, we asked the participants to evaluate the notations according to eight different criteria:

- *Clarity*: Is the notation understandable regarding its syntax and semantics?
- *Completeness*: Does the notation describe all necessary information to review the scenario test without consulting other information sources (e.g., the notation reference)?
- *Conciseness*: Are the resulting scenario tests precise (e.g., no ambiguities in the description, no misinterpretations when reviewing the description)?
- *Expressiveness*: Is the notation eligible to express model tests (e.g., tests on Ecore-based models)?
- *Generalizability*: Is the notation eligible to express tests other than model tests (e.g., code-unit tests)?
- *Practicality*: Does the notation require specific preparation (e.g., an extensive tutorial) and/or auxiliary materials (e.g., frequent use of notation reference)?
- *Scalability*: Does the notation support increasingly complex scenario tests well (e.g., increasing test and/or model size, increasing number of test conditions)?
- *General rating*: Overall, how did you like working with the notation?

Furthermore, participants should elaborate on their ratings (e.g. why did they favor a particular notation). Finally, we asked if they would like to add anything else (related to the scenario tests, improvements to the experiment etc.).

D. Tasks performed by the Participants

In total, each participant had to perform six tasks: one comprehension and one maintenance task per notation (i.e. for each of the three notations). All participants worked on the same six scenarios, but each group received a different sampling of scenario notations (according to the groups built via the experience questionnaire; see Section IV-B). Every task consisted of a scenario as described in Section IV-C: an Ecore-based MUT, a scenario description in one of the three notations, and five questions to be answered (see also the example task in the appendix). For every notation, the participants worked on the comprehension task first, followed by the maintenance task for the same notation.

Both tasks measure the participants' understanding of a scenario. The questions were designed in a way such that they could only be answered correctly if a subject fully understood the scenario. For the comprehension task, the participants were faced with a correct (i.e. passing) scenario test. We asked the participants to look at the scenario description and the corresponding MUT in order to answer five questions ("Does the scenario test fail if ..."). The second task was a maintenance task—here we provided an incorrect (i.e. failing) scenario test. The participants were asked to look at the scenario description and the corresponding MUT and find the line/message numbers which are responsible for the failure of the scenario test (for the maintenance task, an additional line/message number field was inserted in the answer sheet). After the initial error was found, the MUT had to be corrected in order for the scenario test to pass. To achieve this, the participants had to answer five questions whether changes in the model made the scenario test pass ("Does the scenario test pass if ..."). All questions were independent from each other (i.e. the participants had to answer each question on its own).

The process and the design of the tasks resemble the usual workflow of developing model-based software systems via scenarios. In a first step, requirements-based scenario descriptions are developed and executed to validate the MUT. When the requirements change, so do the scenario tests. Now, a discrepancy exists between the new scenario tests and the unchanged MUT. As a consequence, the scenario tests fail. To solve the problem, the MUT needs to be adapted in order to meet the new requirements. Again, the conformance of the (new) requirements is ensured via correct (i.e. passing) scenario tests.

E. Design of the Experiment

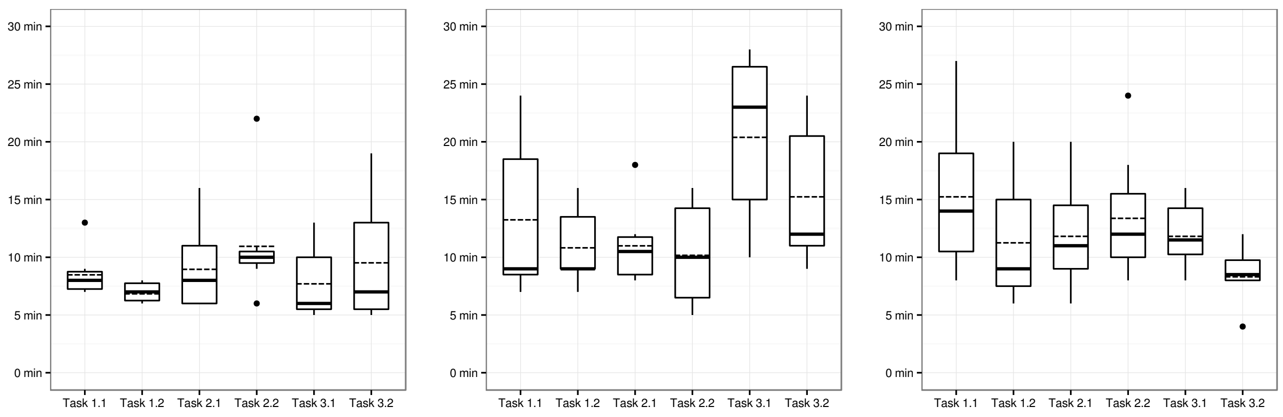
Our controlled experiment has a between-subject design (see, e.g., [31]). In particular, every participant worked with all three notations (our independent variable) and on all six scenarios, but not with every notation alternative on every scenario (to exclude learning effects). The three homogeneous groups built via the experience questionnaire determined the notations which where given to each participant for a particular task. For instance, Group 1 used the natural-language notation for comprehension Task 1.1 and maintenance Task 1.2, Group 2 the diagrammatic notation, and Group 3 the fully-structured notation for the same tasks. For the remaining tasks (2.1, 2.2, 3.1, 3.2) the notations alternated between the groups so that every participant in each group worked with every notation and that every task had to be solved with every notation.

Every scenario was independently designed and did not reuse any elements from another scenario (i.e. different model, scenario description, and questions). To reduce potential learning effects, none of the participants worked on the same scenario twice. Weariness effects were eliminated by randomizing the order in which participants had to work on the different tasks. For further control, the questions per task were also randomly arranged for each participant individually. As domain knowledge has been shown beneficial for comprehension tasks (see, e.g., [32]), we factored out any domain bias by neutrally naming all model elements (e.g. `ClassA`, `refB`, `attC`). Hence, all participants had to use the same bottom-up approach (increasing the internal validity of our experiment), which means that the participants had to analyze the scenario descriptions statement by statement. Thus, the participants first had to understand a scenario statement and then gradually build up to an understanding of groups of statements until the whole scenario was understood completely (see, e.g., [12], [32]).

Before the participants worked on the different tasks, an introductory presentation was given. The introduction explained the objectives of scenario-based testing, the structure of the scenario tests, and the syntax and semantics of Ecore models

TABLE I. Participants' time spent and answers given per task and per notation.

	Task 1.1			Task 1.2			Task 2.1			Task 2.2			Task 3.1			Task 3.2		
	N	D	E	N	D	E	N	D	E	N	D	E	N	D	E	N	D	E
Number of participants	6	7	7	6	7	7	7	6	7	7	6	7	7	7	6	7	7	6
Mean response time (in min.)	8.67	13.43	15.43	7.00	11.00	11.43	9.14	11.17	12.00	11.14	10.33	13.57	7.86	20.57	12.00	9.71	15.43	8.50
Standard deviation (in min.)	2.25	6.80	6.73	0.89	3.32	5.38	3.80	3.71	5.03	5.05	4.68	5.59	3.13	7.14	3.03	5.38	6.05	2.66
Min./max. response time (in min.)	7/13	7/24	8/27	6/8	7/16	6/20	6/16	8/18	6/20	6/22	5/16	8/24	5/13	10/28	8/16	5/19	9/24	4/12
Median response time (in min.)	8	9	14	7	9	9	8	10.5	11	10	10	12	6	23	11.5	7	12	8.5
Lower quartile (in min.)	7.25	8.5	10.5	6.25	9	7.5	6	8.5	9	9.5	6.5	10	5.5	15	10.25	5.5	11	8
Upper quartile (in min.)	8.75	18.5	19	7.75	13.5	15	11	11.75	14.5	10.5	14.25	15.5	10	26.5	14.25	13	20.5	9.75
Number of correct answers	29	29	32	27	21	28	35	26	30	27	23	22	29	28	26	26	27	19
Number of incorrect answers	1	5	1	3	5	7	0	4	5	8	7	8	5	4	4	7	8	11
Number of don't know answers	0	1	2	0	9	0	0	0	0	0	0	5	1	3	0	2	0	0
Percentage of correct answers	96.67	82.86	91.43	90.00	60.00	80.00	100.00	86.67	85.71	77.14	76.67	62.86	82.86	80.00	86.67	74.29	77.14	63.33



(a) N-notation (b) D-notation (c) E-notation

Fig. 2. Box plots of participants' time spent per notation and per task. Please note that the dashed horizontal line indicates the mean response time and the solid horizontal line the median response time.

as well as of the three notations. To train the participants, three example tasks were presented—one for each notation (see Section II). Two questions were asked per task, the participants were motivated to answer the questions, and the solutions were presented. At the end of the introduction, we explained what the participants had to do for the experiment (see Section IV-D).

V. Execution and Deviations

To evaluate our experimental setting (time needed, difficulty of tasks etc.), we conducted a pilot study on March 4, 2014 (one participant) and on March 11, 2014 (two participants), respectively. In each round, we received feedback from the participants and revised the experimental material and the process. The experience questionnaires for building groups were submitted as editable PDFs to the participants on March 9, 2014 via e-mail. The participants had to return the questionnaires by e-mail until March 12, 2014 (the participants were assigned to groups on the same day). As we conducted the study at different locations and due to participants' time constraints, we had to execute the experiment multiple times (between March, 14 and March, 28 2014). The dates were arranged via e-mail communication. However, in every installment the experimental setting was identical (i.e. amount of introductory information presented, room with enough space for all participants and a projector for the introductory presentation etc.)—the experimental process did not differ as well. First, the participants were seated and told that they only need a pen to write and a clock for measuring time. After all participants had arrived, we gave the introductory presentation and performed the three warming up tasks which together took about 20 minutes. Then, each participant received a copy of their personal experimental material (according to the groups they belonged to and with randomized task and question orders; see Sections IV-D and IV-E) and were told that they had to work on the tasks in their predefined order. No supporting material or additional equipment was allowed. Subsequently, the participants worked on their tasks as explained above. Finally, each participant filled out the ex-post questionnaire and handed-in the material.

After the experiment, we had to contact four participants via e-mail and two personally because of ambiguous answers and were able to clarify all issues (e.g., problems reading the handwriting, unclear answer selections). After we evaluated the experiment's data, we informed the participants via e-mail that they could review their personal and, for comparison, the average results (correct answers, time needed etc.). Six participants responded and we transmitted the results to them.

Deviations from the plan occurred as one participant was ill and thus Group 1 consisted only of six participants instead of seven. Furthermore, it was hard to find suitable time-slots for the participants. Some participants took the experiment in the morning, some in the afternoon, and some in the evening. Although, the time of the day may influence the attention and concentration of participants, the different groups as well as the randomized tasks should compensate this deviation. The UML

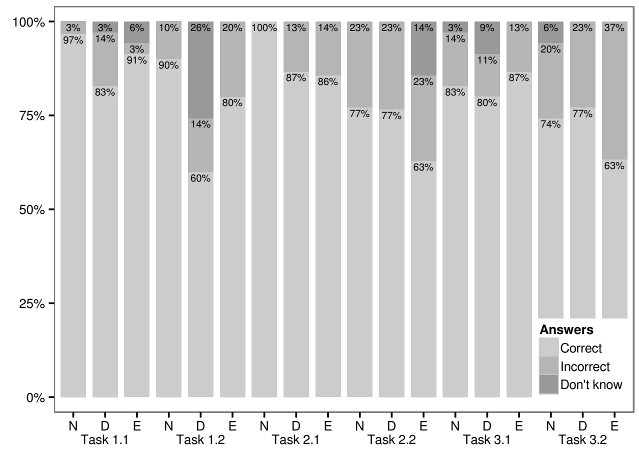


Fig. 3. Percentage of participants' correct, incorrect, and don't know answers per task and per notation.

sequence diagrams needed the most space of all notations, but we fitted each of the diagrams on one A4 page (otherwise participants would have had to turn pages which would have required additional time). Five participants commented that the font sizes of the diagrammatic notation were too small and hard to read. This deviation could have an effect on the comprehension of the diagrammatic scenario notation.

VI. Analysis

Table I as well as Figs. 2 and 3 summarize our collected data in terms of the participants' time spent on the individual tasks and the answers given per task and per notation. The participants measured their time to complete a task in minutes (see Section IV-C). The correctness of a solution is derived from the ratio of correct answers to the sum of all answers (correct, incorrect, don't know answers). Table II shows the participants' average ratings per notation and per dimension from the ex-post questionnaire. The average least task-solving time and highest percentage of correct answers per task as well as the highest average rating per criterion are underlined in Tables I and II, respectively.

On average (arithmetic mean) the participants spent the following times for completing the six tasks: 53.52 min. for the natural-language notation (N), 81.93 min. for the diagrammatic notation (D), and 72.93 min. for the fully-structured notation (E). Furthermore, the percentage of correct answers over all tasks is 86.5% for the natural-language notation, 77% for the diagrammatic notation, and 78.5% for the fully-structured notation. On average, the participants spent 70.05±17.06 min. on the six tasks. As each task contained five questions, each participant had to answer 30 questions in total. On average, each participant answered 80.67% of the questions correctly (24.2±3.55 questions), 15.5% of the questions incorrectly (4.65±2.18 questions), and the participants did not know the answer to 3.83% of the questions (1.15±2.21 questions).

VII. Interpretation

Table I and Fig. 2 show that, except for Tasks 2.2 and 3.2, the participants required the least time to solve a task using the natural-language notation. Regarding these two tasks, the task-solving times of the natural-language notation were close to the times of the other notations: the diagrammatic notation in Task 2.2 (participants required on average 0.81 min. less time; an improvement by 7.27%) and the fully-structured notation in Task 3.2 (participants required on average 1.21 min. less time; an improvement by 12.46%). By using the natural-language notation, the participants required on average 28.4 min. less time to finish all six tasks than with the diagrammatic notation (an improvement by 34.7%) and 19.4 min. less time than with the fully-structured notation (an improvement by 26.6%). Using the fully-structured notation, it took the participants 9 min. less time than working on the tasks in diagrammatic notation (an improvement by 11%).

TABLE II. Ex-post average rating per notation (1–5, 5=best).

	N	D	E
Clarity	4.30±0.80	3.10±1.02	3.05±1.10
Completeness	4.40±0.82	3.45±0.89	3.15±1.04
Conciseness	4.00±0.79	3.35±1.14	3.80±1.15
Expressiveness	3.85±1.09	3.10±1.25	3.35±1.09
Generalizability	3.40±1.05	3.05±0.94	3.95±1.10
Practicality	4.65±0.59	3.00±0.97	2.50±1.00
Scalability	4.15±0.93	2.30±1.03	3.35±0.81
General	4.40±0.60	2.85±1.14	3.00±0.97

For the four Tasks 1.1–2.2, the participants reached most correct answers by using the natural-language notation (see Table I and Fig. 3). The fully-structured and the diagrammatic notations are comparable regarding answer correctness (78.5% and 77%). There was one participant who had a very low score of only 14 correct answers (46.7%). When removing this outlier from the data set, the natural-language notation accounts for the highest amount of correct answers in all tasks, including Tasks 3.1 and 3.2. The respective notation of the least response time remains unchanged, both per task and overall.

In total, participants using the natural-language notation required the least time to solve the tasks and answered the most questions correctly. These results in favor of the natural-language notation are also supported by the participants ex-post rating (see Table II). Except for its *generalizability*, the participants ranked the natural-language notation first in all remaining dimensions.

Via the ratings and comments in the ex-post questionnaire (see Table II), the study participants criticized the poor *scalability* of the diagrammatic notation (quote: "just overhead", "bloated"). Both the structure of the diagrammatic notation and the general syntax of UML sequence diagrams need comparably more space than the other two notations. Thus, for complex scenarios, the diagrammatic notation might incur the risk of comparatively large scenario-test definitions which, thereby, become hard to comprehend. Furthermore, the participants stated that the fully-structured notation requires the most specific preparation and/or auxiliary materials to be understood. Its *practicality* was therefore rated lowest (see Table II). However, despite its lower ranked practicality, the task-solving time was improved by 11% and the percentage of correct answers by 1.5% when using the fully-structured rather than the diagrammatic notation.

VIII. Threats to Validity

Some threats to internal validity are caused by the deviations that occurred (see Section V). Furthermore, our group building process focused on a homogeneous distribution of participants with the same experience level. We tried to equally distribute demographic characteristics as well, and managed to do so for the participants' education and their age. Nevertheless, the data set did not allow for a homogeneous distribution of female participants (n=5) which is also a threat to internal validity. Our focus on internal validity limits external validity. The participants were all software professionals working with a selection of notations on a set of neutralized Ecore models to solve specific scenario tasks. In order to generalize our results and to improve external validity, we would need to repeatedly conduct the experiment with different participants from diverse professional backgrounds, different notation alternatives, different scenario descriptions and so forth.

IX. Conclusion and Future Work

In this paper, we presented an experiment to evaluate three notations for their applicability to describe scenario-based model tests. In particular, the experiment compared the comprehensibility of three scenario-based notations and showed that the choice of a specific notation has an effect on the productivity when testing models for requirements conformance. The results of our experiment indicate that a natural-language-based approach for scenario-based model tests is recommended, as it required the least task-solving time and was the most accurate alternative as well as most favored by the participants who worked with it.

As future work, we will repeat the experiment with participants from a different professional background to be able to draw more general conclusions. For the replication study, we will revise the experimental setting and materials based on the feedback collected so far (e.g., conducting the experiment in one session, increasing the font size of the diagrammatic notation). Furthermore, we will run extended quantitative and qualitative analyses. For example, we will evaluate the line/message numbers recorded for the failing scenario tests by the participants.

Appendix
Experiment Task 2.1

To illustrate the actual tasks performed during the experiment, we present the comprehension Task 2.1 of our experiment. For completing a task, we provided each participant with an Ecore-based MUT (for Task 2.1 the MUT is shown in Fig. 4) and a scenario-test definition in one of the three notations. For Task 2.1, the scenario was either specified as in Listing 3, as in Listing 4, or as in Fig. 5. Five questions were to be answered on each task, the ones about Task 2.1 are shown later in this section.

The MUT of Task 2.1 (see Fig. 4) is based on the Ant metamodel and was adapted to fit the experimental setting (in terms of size and naming conventions). The MUT of Task 2.1 contains 24 classes, 8 of them declared abstract. These classes contain 46 structural features consisting of 22 attributes and 24 references. 15 of theses references are defined as containment references. In the MUT of Task 2.1, 19 inheritance relationships between classes are specified.

The scenario descriptions of Task 2.1 for each notation are shown in Listing 3, Listing 4, and Fig. 5, respectively. Please note that all three scenario-test definitions are semantically equivalent; i.e. all three scenario representations per task describe the same scenario-based model test. Each participant of the experiment was provided with exactly one of these notations, depending on their group affiliation.

```
Scenario:
  Given "that it is possible to navigate from ClassO to ClassA via
        containment references refQ and refJ"
  And "that ClassY includes ClassF, ClassP, and ClassE in its hierarchy
        of supertypes"
  When "ClassD, ClassR, and ClassV are not abstract"
  And "exactly two attributes of ClassA are of type EString with a
        multiplicity of 0..1"
  Then "ClassY shall own exactly six structural features"
  And "ClassG shall have any kind of reference pointing to ClassR named
        refG"
```

Listing 3. Natural-language scenario notation of Task 2.1.

```
@TestSuite
operation testSuite() {
  @TestCase
  operation testCase() {
    @TestScenario
    $pre Model!EClass.all().selectOne(x|x.name="ClassO").
          eStructuralFeatures.selectOne(x|x.name="refQ").eType.
          eStructuralFeatures.selectOne(x|x.name="refJ").eType.name="
          ClassA"
    $pre Model!EClass.all().selectOne(x|x.name="ClassY").closure(x|x.
          eSuperTypes).includesAll(Model!EClass.all().select(x|x.name="
          ClassF" or x.name="ClassP" or x.name="ClassE"))
    operation testScenario() {
      if (Model!EClass.all().select(x|x.name="ClassD" or x.name="ClassR"
            or x.name="ClassV").forAll(x|x.abstract=false)
        and
        Model!EClass.all().selectOne(x|x.name="ClassA").
              eStructuralFeatures.select(x|x.isTypeOf(EAttribute) and x.
              eType.name="EString" and x.lowerBound=0 and x.upperBound
              =1).size() = 2
      ) {
        assertTrue(Model!EClass.all().selectOne(x|x.name="ClassY").
              eStructuralFeatures.size() = 6);
        assertTrue(Model!EClass.all().selectOne(x|x.name="ClassG").
              eStructuralFeatures.selectOne(x|x.name="refG").eType.name
              = "ClassR");
      } else {
        assertTrue(false);
      }
    }
  }
}
```

Listing 4. Fully-structured scenario notation of Task 2.1.

For the comprehension Task 2.1, the participants had to answer the following five questions with either "yes", "no", or "don't know" (please note that the questions were randomly ordered for each participant):

1) Does the scenario test fail if ClassF, ClassT, and ClassD are declared abstract?
2) Does the scenario test fail if the reference refG of ClassG is not a containment reference and has a multiplicity of 1..1?
3) Does the scenario test fail if in ClassI the navigability of reference refJ is inverted?
4) Does the scenario test fail if in ClassA the lower bound of attribute attR is changed to 1?
5) Does the scenario test fail if the reference refB of ClassY is inverted?

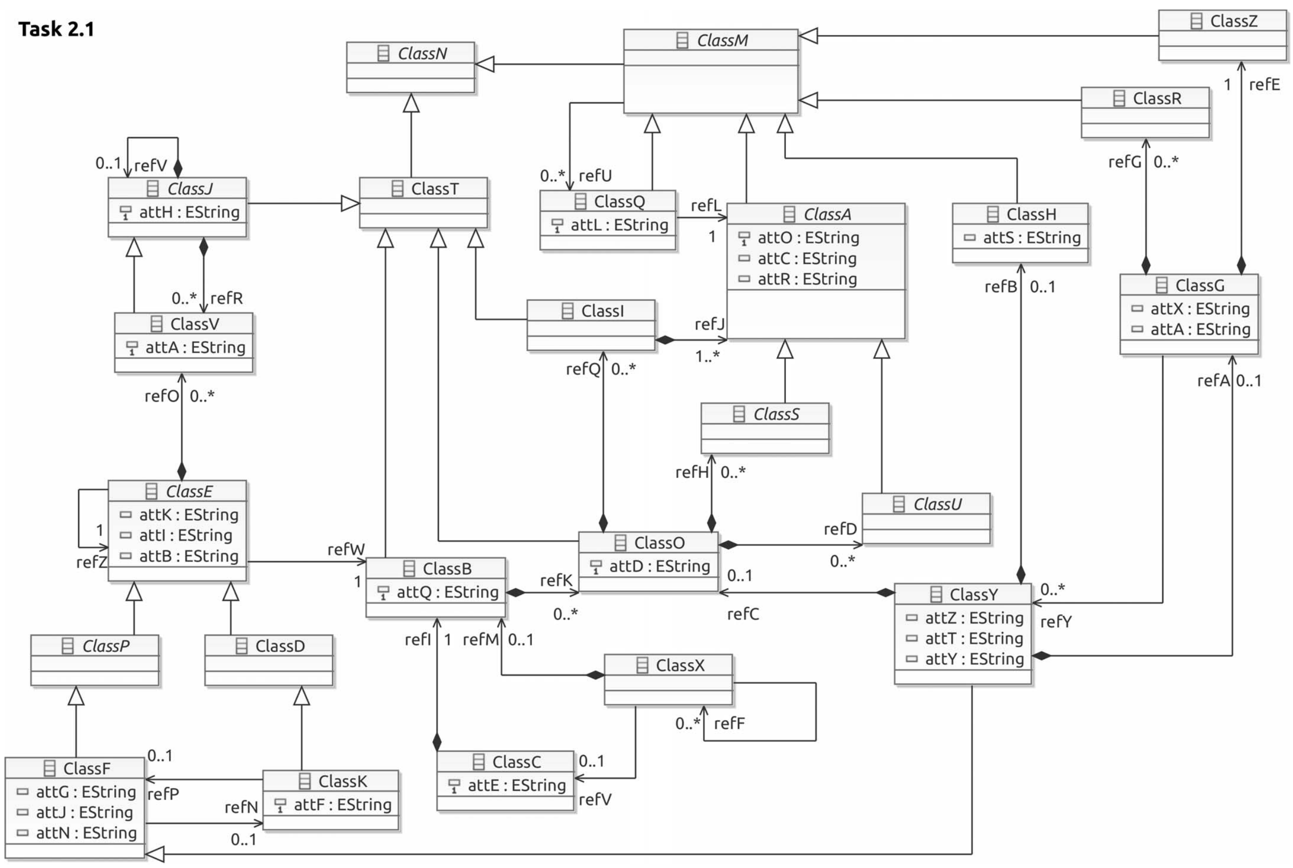


Fig. 4. Ecore-based MUT of Task 2.1.

If a scenario test was deemed failed (i.e. a question was answered with "yes"), the participant had to mention the line number(s) (in case of the natural-language or fully-structured notation) or message number(s) (in case of the diagrammatic notation) in the scenario description considered responsible for failing the test. Furthermore, the participants had to fill in their start and end time per task (in minutes). At the end of the answer sheet, space was reserved for comments.

Acknowledgment

Thanks are due to Janet Siegmund for her advice on planning the experiment. 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, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, Feb 2006.

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

[3] P. Brosch, U. Egly, S. Gabmeyer, G. Kappel, M. Seidl, H. Tompits, M. Widl, and M. Wimmer, "Towards scenario-based testing of UML diagrams," in *Tests and Proofs*, ser. LNCS. Springer, 2012, vol. 7305, pp. 149–155.

[4] S. Sobernig, B. Hoisl, and M. Strembeck, "Requirements-driven testing of domain-specific core language models using scenarios," in *Proc. 13th Int. Conf. Quality Softw.* IEEE Computer Society, 2013, pp. 163–172.

[5] I. Sommerville, *Software Engineering*, 9th ed. Addison-Wesley, 2010.

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

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

[8] C. Nebut, F. Fleurey, Y. Le-Traon, and J.-M. Jézéquel, "Automatic test generation: A use case driven approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 140–155, 2006.

[9] A. Ulrich, E.-H. Alikacem, H. Hallal, and S. Boroday, "From scenarios to test implementations via Promela," in *Test. Softw. Syst.*, ser. LNCS. Springer, 2010, vol. 6435, pp. 236–249.

[10] D. Amyot and A. Eberlein, "An evaluation of scenario notations and construction approaches for telecommunication systems development," *Telecommun. Syst.*, vol. 24, no. 1, pp. 61–94, 2003.

[11] T. Kosar, M. Mernik, and J. Carver, "Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments," *Emp. Softw. Eng.*, vol. 17, no. 3, pp. 276–304, 2012.

[12] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the #ifdef hell?" *Emp. Softw. Eng.*, vol. 18, no. 4, pp. 699–745, 2013.

[13] J. Koenemann and S. P. Robertson, "Expert problem solving strategies for program comprehension," in *Proc. Conf. Human Factors Comp. Sys.* ACM, 1991, pp. 125–130.

[14] A. Dunsmore and M. Roper, "A comparative evaluation of program comprehension measures," University of Strathclyde, Tech. Rep., 2000.

[15] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in

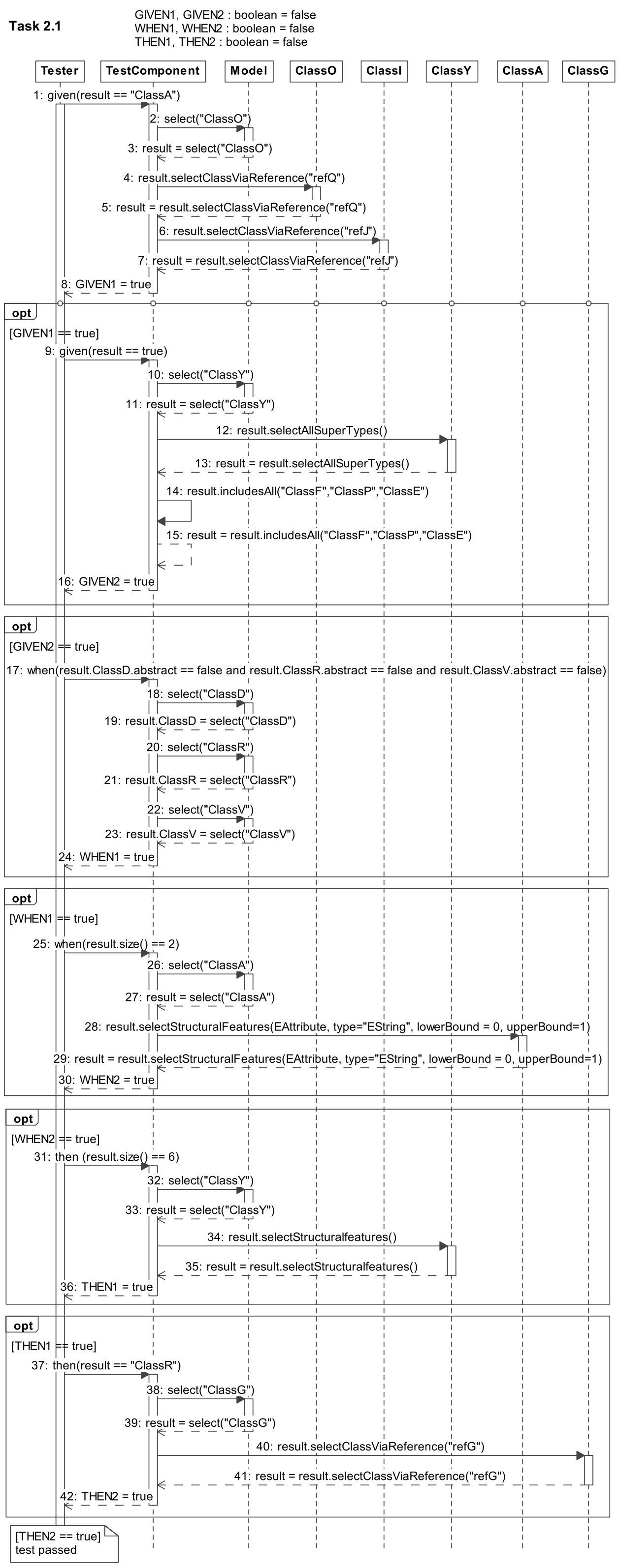

Task 2.1
GIVEN1, GIVEN2 : boolean = false
WHEN1, WHEN2 : boolean = false
THEN1, THEN2 : boolean = false
Tester
TestComponent
Model
ClassO
ClassI
ClassY
ClassA
ClassG
1: given(result == "ClassA")
2: select("ClassO")
3: result = select("ClassO")
4: result.selectClassViaReference("refQ")
5: result = result.selectClassViaReference("refQ")
6: result.selectClassViaReference("refJ")
7: result = result.selectClassViaReference("refJ")
8: GIVEN1 = true
opt
[GIVEN1 == true]
9: given(result == true)
10: select("ClassY")
11: result = select("ClassY")
12: result.selectAllSuperTypes()
13: result = result.selectAllSuperTypes()
14: result.includesAll("ClassF","ClassP","ClassE")
15: result = result.includesAll("ClassF","ClassP","ClassE")
16: GIVEN2 = true
opt
[GIVEN2 == true]
17: when(result.ClassD.abstract == false and result.ClassR.abstract == false and result.ClassV.abstract == false)
18: select("ClassD")
19: result.ClassD = select("ClassD")
20: select("ClassR")
21: result.ClassR = select("ClassR")
22: select("ClassV")
23: result.ClassV = select("ClassV")
24: WHEN1 = true
opt
[WHEN1 == true]
25: when(result.size() == 2)
26: select("ClassA")
27: result = select("ClassA")
28: result.selectStructuralFeatures(EAttribute, type="EString", lowerBound = 0, upperBound=1)
29: result = result.selectStructuralFeatures(EAttribute, type="EString", lowerBound = 0, upperBound=1)
30: WHEN2 = true
opt
[WHEN2 == true]
31: then (result.size() == 6)
32: select("ClassY")
33: result = select("ClassY")
34: result.selectStructuralfeatures()
35: result = result.selectStructuralfeatures()
36: THEN1 = true
opt
[THEN1 == true]
37: then(result == "ClassR")
38: select("ClassG")
39: result = select("ClassG")
40: result.selectClassViaReference("refG")
41: result = result.selectClassViaReference("refG")
42: THEN2 = true
[THEN2 == true]
test passed

Fig. 5. Diagrammatic scenario notation of Task 2.1.

software engineering," in *Guide Advanced Emp. Softw. Eng.* Springer, 2008, pp. 201–228.

[16] B. Kitchenham, H. Al-Khilidar, M. Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu, "Evaluating guidelines for reporting empirical software engineering studies," *Emp. Softw. Eng.*, vol. 13, no. 1, pp. 97–121, 2008.

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

[18] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Addison-Wesley, 2008.

[19] B. Hoisl, S. Sobernig, and M. Strembeck, "Natural-language scenario descriptions for testing core language models of domain-specific languages," in *Proc. 2nd Int. Conf. Model-Driven Eng. Softw. Dev.* SciTePress, 2014, pp. 356–367.

[20] Object Management Group, "OMG unified modeling language (OMG UML), superstructure," available at: http://www.omg.org/spec/UML, Aug. 2011, v2.4.1, formal/2011-08-06.

[21] S. K. Swain, D. P. Mohapatra, and R. Mall, "Test case generation based on use case and sequence diagram," *Int. J. Softw. Eng.*, vol. 3, no. 2, pp. 21–52, July 2010.

[22] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige, "The Epsilon book," Available at: http://www.eclipse.org/epsilon/doc/book/, 2014.

[23] Object Management Group, "Object constraint language," available at: http://www.omg.org/spec/OCL, Feb. 2014, v2.4, formal/2014-02-03.

[24] W. Kintsch, "Learning from text," *Cognition Instruct.*, vol. 3, no. 2, pp. 87–108, 1986.

[25] J. Mendling, M. Strembeck, and J. Recker, "Factors of process model comprehension—findings from a series of experiments," *Decision Support Syst.*, vol. 53, no. 1, pp. 195–206, 2012.

[26] F. Ricca, M. Torchiano, M. D. Penta, M. Ceccato, and P. Tonella, "Using acceptance tests as a support for clarifying requirements: A series of experiments," *Inform. Softw. Tech.*, vol. 51, no. 2, pp. 270–283, 2009.

[27] A. Gemino and D. Parker, "Use case diagrams in support of use case modeling: Deriving understanding from the picture," *J. Database Mgmt.*, vol. 20, no. 1, pp. 1–24, 2009.

[28] A. Ottensooser, A. Fekete, H. A. Reijers, J. Mendling, and C. Menictas, "Making sense of business process descriptions: An experimental comparison of graphical and textual notations," *J. Syst. Softw.*, vol. 85, no. 3, pp. 596–606, 2012.

[29] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience," *Emp. Softw. Eng.*, 2013.

[30] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger, "Towards an expressivity benchmark for mappings based on a systematic classification of heterogeneities," in *Proc. 1st Int. Workshop Model-Driven Interoperability*. ACM, 2010, pp. 32–41.

[31] A. Dix, J. Finlay, G. Abowd, and R. Beale, *Human-Computer Interaction*, 3rd ed. Pearson Education, 2004.

[32] T. M. Shaft and I. Vessey, "The relevance of application domain knowledge: The case of computer program comprehension," *Inform. Syst. Res.*, vol. 6, no. 3, pp. 286–299, 1995.

Appendix B

Evaluating Higher-Order Rewriting of M2T Templates: A Case Study

In this chapter, we report on a case study performed in order to evaluate our approach to higher-order rewrite M2T templates developed in this thesis project to support the integration of MOF/UML-based DSMLs (see Section A.3 and [53]). In particular, we apply our approach to a publicly available, Epsilon-based open-source project (*Pongo*; see Section B.1.2) to evaluate its feasibility. Via the case study, we show how our approach facilitates reusing M2T templates and reducing the need of manually performed refactorings in metamodel-evolution scenarios (we apply our rewriting technique for the evolution of the Pongo project to support not only Ecore-, but also UML-based models). We measure the benefits of our approach, for example, via the ratio of automatically to manually executed rewriting operations on EGL-based templates. Furthermore, we present improvements and extensions as well as discuss limitations of our approach and the corresponding software prototype which resulted from conducting the case study.[9]

In this chapter, we apply the guidelines for conducting and documenting case study research in software engineering [117] which adapt the proposed reporting structure for empirical software studies defined in [64] and in [68]. Accordingly, the subsections cover the design of the case study (problem statement, research objectives, case selection, conceptual framing, methods; see Section B.1), its results (case description, execution, analysis, interpretation; see Section B.2), limitations and threats (see Section B.3), as well as a concluding summary (Section B.4).

[9]All software artifacts used in and developed for the case study as well as the improved prototypical implementation of our approach for higher-order rewriting of M2T templates are publicly available at [48].

B.1 Case Study Design

The context of the case study is set by the definition of the problem statement and the research objectives (including the definition of two research questions; Section B.1.1). We explain the process of selecting a suitable case for answering the research questions in Section B.1.2. The theoretical frame of reference of this study and the study methods (e.g., data collection and analysis procedures) are detailed in Sections B.1.3 and B.1.4, respectively.

B.1.1 Problem Statement and Research Objectives

To recap, in Section A.3 (see also [53]) we present an approach to rewriting M2T generator templates syntactically for reusing them for evolving metamodels (e.g., as in DSML integration scenarios). By considering M2T templates as first-class models and by reusing M2M transformation traces, we developed a rewriting approach based on higher-order model transformations (HOTs) for *transformation modifications* [156, 157]. To demonstrate the feasibility of this rewriting technique, we provide a prototype implementation and a DSML integration example based on the EMF project and the Epsilon language family. Hence, with our rewriting approach of M2T templates, we provide a solution to deal with structural mismatches between different DSML metamodels in an integration scenario. In [53], we defined the mismatch problem as a question specifically targeting the case of integrating DSMLs: "How can we make the generator templates $\mathtt{T_A}$ and $\mathtt{T_B}$ apply to instances of the composed metamodel $\mathtt{M_C}$ rather than to instances of metamodels $\mathtt{M_A}$ and $\mathtt{M_B}$, respectively?"

DSML integration is a special case of model-driven software evolution and maintenance (see, e.g., [14, 22, 60, 140, 172, 175]). For example, in [53], the integrated DSML $\mathtt{C}$ is fully composed of both DSMLs $\mathtt{A}$ and $\mathtt{B}$ (i.e. $\mathtt{C} \widehat{=} \mathtt{A} \cup \mathtt{B}$). Both individual DSML metamodels are connected structurally by merging two concepts (one from each DSML) into a single concept of DSML $\mathtt{C}$. Otherwise, the integration preserves all structural semantics present in the two source metamodels (e.g., inheritance, attributes, references). From the perspective of DSML $\mathtt{A}$, an integration with DSML $\mathtt{B}$ into DSML $\mathtt{C}$ conforms to an evolution of DSML $\mathtt{A}$ into DSML $\mathtt{A} \cup \mathtt{B}$ (equally, DSML $\mathtt{B}$ evolves into DSML $\mathtt{B} \cup \mathtt{A}$). In this sense, we can generalize the problem statement of [53] and reformulate it as the following question: "How can we make the generator template $\mathtt{T_X}$ apply to instances of an evolved metamodel $\mathtt{M_Y}$ rather than to instances of the original metamodel $\mathtt{M_X}$?"

Rewriting M2T generator templates to conform to a different metamodel requires dealing with structural heterogeneity of these metamodels [177]. Problems for the automatic rewriting emerge from the considerable amount of potential metamodel change operations (e.g., conflicting source-target-concept cardinalities; multiplicity, direction, and containment differences of references; or context, order, and datatype differences of class attributes). Our current approach supports three syntactical higher-order rewriting operations (retyping, association retargeting, and property renaming [53]), however, it allows for arbitrary rewriting operations. This case study was performed in order to (1) evaluate the feasibility of our approach in a real-life context[10] and to (2) quantify the alleged benefits from applying the proposed rewriting technique. Hence, our case study was guided by the following two research questions indicating its exploratory and improving characteristics:

Research Question 1. *To which extent can our current approach be used in a real-life MDD-based software development project?*

Research Question 2. *What are the observable and quantifiable advantages of applying our approach?*

B.1.2 Case Selection

The two guiding research questions establish different requirements on the case to be selected—especially with regard to technical requirements. Our approach for higher-order rewriting of M2T templates builds on EMF and the Epsilon language family. In particular, we employ Ecore-based models, perform M2M transformations with ECL, EML, and ETL, define rewrite rules over model representations of EGL templates, and provide helper operations in EOL. In principle, our approach does not depend on any of these implementation techniques and models as well as model-management languages are exchangeable (e.g., ATL-based rewrite rules can be generated instead of ETL-based ones or via the EMC layer [70], it can be switched between concrete modeling technologies). Regardless of the chosen implementation technique, the application of our approach demands that the source code of the software artifacts the case study is based on is accessible (e.g., M2T transformation templates). We also require that all artifacts are publicly available to make our case study and its results reproducible. Furthermore, the case must be non-trivial and fit our generalized definition of an evolutionary scenario as outlined in Section B.1.1 (i.e. the evolution process should not be too simplistic, e.g., not only a change of one or two metamodel elements, and the M2T transformation templates should not be too short, e.g., contain not only a couple of source lines of code; SLOC).

In order to find a suitable case, we asked the MDD community via postings in relevant Eclipse sub-forums for hints [145] and contacted research and industry peers. One of our colleagues at the Department of Computer Science at the University of York (Dimitris Kolovos) pointed us to the Pongo project [71] which describes itself as "a template-based Java POJO[11] generator for MongoDB [86]. Instead of using low-level `DBObjects` to interact with your MongoDB database, with Pongo you can define your data/domain model using EMFatic and then generate strongly-typed Java classes you can then use to work with your database at a more convenient level of abstraction" [71]. In Pongo, the M2T transformations from Ecore models to Java source code is implemented in Epsilon (EGL templates with EOL helper operations and EGX as coordination language for EGL templates). As the Pongo project

[10]For an overview and discussion of evaluation methods in design science research, see, e.g., [43, 44].

[11]Plain Old Java Object [31].

fulfilled our requirements (open-source, publicly available, non-trivial M2T transformation definitions), we selected it for our case study. Currently, Pongo supports Ecore-based domain models only (e.g., specified via the EMFatic textual syntax or via Ecore model/diagram editors). For the definition of an evolutionary process, we adopted a scenario to migrate Pongo to be able to work with UML models (in particular, class diagrams), as well.[12] The case study was performed with data (domain model, test application) obtained from the Pongo tutorial (a blogging system) published on the project's website [69].

B.1.3 Conceptual Framing

Our work adopts the notion of HOTs as presented for ATL by Tisi et al. [156, 157]. We employ HOTs to rewrite M2T templates (transformation modifications [156, 157]) for their reuse in evolutionary MDD-based scenarios (e.g., for the integration of DSMLs [53]). Our template syntax rewriting technique allows for reusing transformation logic independently from the scenario [178]. It is applicable to M2T transformation languages which support a subset of the MOFM2T specification [93], HOTs, and basic model transformation tracing (e.g., as is the case for EGL [53]). Our approach implements three syntactical higher-order rewriting operations to deal witch structural heterogeneity between metamodels [177]: retyping, association retargeting, and property renaming [53]. Other approaches using HOTs as implementation vehicle (e.g., generic templates [18, 19, 168]), differ, for instance, in their expressiveness to handle metamodel heterogeneity as well as in their level of automation and of unanticipated reuse of generator templates. The case study was designed to qualitatively evaluate our approach with regard to its observed practical feasibility in a real-life context and to provide for quantitative evidence of the advantages of its application (methods employed in the case study are detailed in the next section).

B.1.4 Methods

We do not only want to apply our rewriting approach in a real-life setting, but also gain insights to develop possible extensions and improvements (see Research Question 1). This requires in-depth knowledge of our rewriting approach (structure, functionality, limitations) as well as of the employed technology stack (EMF, Epsilon). Therefore, the author himself designed, planned, and conducted the case study. This way, corrections, extensions, and improvements emerging from the case study can be integrated into our approach in a timely manner.

To answer both research questions, we need to collect qualitative (Research Question 1) and quantitative data (Research Question 2). Qualitative data is gathered based on the observations while applying our approach (e.g., rewrite problems, missing functionality etc.; see Section B.2.2). We base our definition for obtaining quantitative data on a goal-oriented measurement technique by adopting the Goal Question

[12]The scenario was inspired by the discussion at `https://www.eclipse.org/forums/index.php/t/488742/`.

Metric (GQM) method (see, e.g., [2, 167]). In the GQM method, goals are formulated first (conceptual level). Then, a set of questions is defined to characterize the way the assessment of a specific goal is going to be performed (operational level). At last, metrics are specified and associated with every question in order to answer it in a quantitative way (quantitative level). Table B.1 shows our GQM model for quantifying the advantages of our approach from the viewpoint of a DSML engineer. By applying the GQM method, we defined four questions with corresponding metrics which all contribute to answer Research Question 2 (our goal). Please note that Table B.1 serves only as an overview of questions and metrics which are going to be explained in detail in the following paragraphs.

Table B.1: GQM model for measuring advantages of our approach

Goal	Purpose	Quantifying
	Issue	the advantages of
	Object	applying our rewriting approach
	Viewpoint	from the DSML engineer's viewpoint.
Question		To which extent does our approach reduce the need to manually perform template-rewriting operations?
Metrics		1. Number of automatically/manually rewritten concepts with/without employing our approach.
Question		Is it feasible to employ standard-tool-supported refactorings?
Metrics		2. Number of unique concepts (frequent patterns).
Question		Are additional helper operations used and/or does the amount of calls to helper operations increase/decrease?
Metrics		3. Number of called helper operations. 4. Number of calls to helper operations.
Question		Does the size of rewritten generator template(s) differ when compared with the original one(s)?
Metrics		5. Transformation size in SLOC.

As introduced in Section B.1.1, in a metamodel evolution scenario, generator templates must be adapted to apply to instances of the evolved metamodel (e.g., to conform to changed metamodel elements). However, these adaptations must not change the semantics of M2T transformations and, thus, the functionality of the generated code stays the same (i.e. behavior of generator templates are reused as-is). Hence, in our metamodel evolution scenario, we apply structural *refactorings* to the initial generator template(s) to establish conformance with the evolved metamodel (e.g., retyping or property renaming). In this thesis, we adopt the definition of refactoring code as "to restructure software by applying a series of refactorings without changing its observable behavior" [29].

Our case study consists of refactoring the M2T transformations of the Pongo project to work not only with Ecore, but also with UML models. Therefore, we focus on one observable part of Pongo's software artifacts only: the EGL-based generator templates (and accompanying Epsilon scripts used for the M2T transformation, such as, included in EGX and EOL files). In order to answer Research Question 2 (and all questions defined in Table B.1 accordingly), we need to define *metrics* for measuring the amount of refactorings applied on the generator templates to conform to the evolved UML metamodel. In general, a (software/quality) metric "is a term that embraces many activities, all of which involve some degree of software measurement" [24]; it is "a quantitative measure of the degree to which a system, component, or process possesses a given attribute" [61]. For the case study, we adopt metrics inspired by related work on model transformation metrics, such as, for MOF Query/View/-Transformation (MOF QVT [94]) or ATL (see, e.g., [67, 164, 165, 166, 169]):

1. *Number of rewritten concepts*: This metric counts every adapted concept (e.g., types, expressions) required to render M2T transformations compliant with the evolved metamodel (see, e.g., [24]). The metric measures both, concepts that are *automatically* rewritten by our approach and concepts which need *manual* adaptation.

2. *Number of unique concepts (frequent patterns)*: This metric counts the unique concepts rewritten in M2T transformations (see, e.g., [67]).

3. *Number of called helpers*: This metric counts the number of helper operations (e.g., in EOL files) which are used (i.e. called at least once) during M2T transformation (see, e.g., [165, 169]).

4. *Number of calls to helpers*: This metric sums up the total number of calls to helper operations during a M2T transformation (see, e.g., [165, 169]).

5. *Transformation size*: This metric counts the SLOC of M2T transfromation definitions; i.e. of EGL templates and accompanying Epsilon scripts (see, e.g., [24, 67]).

The metric (1.) *number of rewritten concepts* (and, thus, the corresponding question defined in Table B.1) is the main source for providing an answer to Research Question 2. A high number of automatically rewritten concepts performed via our approach would establish evidence for the advantage of fewer manual adaptations. The other three questions with their corresponding four metrics (2.–5.) provide additional insights into the rewriting process and are operationalized as a means of control. For example, if only a few (2.) *unique concepts* are found during template rewriting, employing other approaches not relying on transformation traces (e.g., tool-specific search-and-replace functionality) may be a viable option. However, as no explicit trace information between source and target concepts of involved metamodels would be available, the correct rewriting operations have to be discovered manually (e.g., establishing links between metamodel elements by finding equally named model instance specifications). Furthermore, the (3. and 4.) *number of called/calls to helpers* are metrics to check if a potentially high number of automatically rewritten concepts are not based on extensive use of (newly defined) helper operations. Additional helper operations would also increase the (5.) *transformation size* of the generator templates. Thus, at last, we want to know if and how much the transformation size of the rewritten generator templates differ when compared with their original versions.

All five metrics are measuring internal attributes of a M2T transformation (see, e.g., [24, 163]). The last three metrics (3.–5.; number of called helpers, number of calls to helpers, and transformation size) can be measured directly. In contrast, the first two metrics (1. and 2.; number of rewritten concepts and number of unique concepts) can only be measured indirectly by comparing the original generator templates with the rewritten ones (see, e.g., [24]). The measurement of the various metrics is tool-supported, for example, by the profiling mechanism of the Epsilon language (see, e.g., [72, 116]).

B.2 Results

Results of the case study are reported by describing the studied software artifacts (Section B.2.1), by explaining the process of executing the study (Section B.2.2), as well as by a detailed analysis (Section B.2.3) and interpretation (Section B.2.4) of observations and collected data.

B.2.1 Case Description

As explained in Section B.1.2, our case study is based on the Pongo project [71] and utilizes on data obtained from its introductory tutorial (a blogging system [69]). For the evolutionary scenario of migrating from Ecore to UML class models, the case study explores the refactoring process of the Epsilon-based M2T transformation definitions. The rewritten M2T templates are applied on the evolved UML-based domain model of the blogging system to compare the generated platform-specific software artifacts (i.e. Java classes) with the ones originally created from the Ecore domain model to evaluate the successful transformation. Regarding software artifacts, the case study combines the following Eclipse projects:[13]

- `com.googlecode.pongo`: This project provides the M2T transformation definitions as Epsilon scripts. With the EGX, EGL, and EOL files included in this project, an Ecore model can be transformed into Java classes for interacting with MongoDB (the project can be obtained from [71]).

- `com.googlecode.pongo.examples.miniblog`: This project provides the Pongo tutorial files used in the case study: an EMFatic-specified domain model, the

[13]Please note that we only describe the Eclipse projects directly involved in the case study and that some of the projects depend on additional resources.

generated Java classes as well as a test Java application. The Ecore model serves as input to the `com.googlecode.pongo` project for its transformation into Java classes. The Pongo tutorial covers a scenario for defining a blogging system including authors, posts, and comments as data entities (the project can be obtained from [71]).

- `com.googlecode.pongo.uml`: This project provides the rewritten UML-compliant M2T transformation specifications as Epsilon scripts (i.e. the resulting files from performing the case study). For comparison, the original Ecore-based M2T transformation definitions are included in the project, as well (i.e. the same files as in `com.googlecode.pongo`; the project can be obtained from [48]).
- `com.googlecode.pongo.examples.miniblog.uml`: This project provides the UML-compliant Pongo tutorial files used in the case study: a UML domain model and the generated Java classes as well as a test Java application (i.e. the resulting files from performing the case study; the project can be obtained from [48]).
- `org.eclipse.epsilon.egl.dom`, `org.eclipse.epsilon.egl.dom.ast2dom`, and `org.eclipse.epsilon.egl.dom.printer`: In combination, these projects provide code/model round-tripping functionality for EGL. They extend (and depend on) the `org.eclipse.epsilon.eol.dom*` projects which provide code/-model round-tripping for EOL (see Section 5.3; the projects can be obtained from [48]).
- `at.ac.wu.nm.dsml.eval.hotm2t`: This project provides the case-study-specific files of our rewriting approach. This includes the Ecore-to-UML M2M transformation definitions (ECL, EML, ETL, EOL), the trace metamodel and the instance model, the `UMLExcerpt` metamodel (see below), the rewrite rule generator (EGL), the generated rewrite rules (ETL), the original and the rewritten Pongo M2T transformation definitions (EGL specifications as model representations), and the orchestrating Apache Ant workflow definitions (the project can be obtained from [48]).

The domain model of the blogging system used as tutorial for the Pongo project and also for this case study is specified in the EMFatic textual syntax (see Listing B.1).[14] The model defines four `EClasses` (`Blog`, `Post`, `Comment`, and `Author`) as well as corresponding attributes and references to represent the blogging domain. At execution time of the M2T transformation, an Ecore model is generated from the EMFatic representation which is further processed by the generator components (see Figure B.1). Please note that the EMFatic textual syntax (Listing B.1) and

[14]Please note that the EMFatic specification shown in Listing B.1 is a corrected version of the definition originally provided at [69] (see lines 14 and 19).

the Ecore diagram (Figure B.1) specifications are fully interchangeable and—in this case—define identical domain models for the blogging system.

Listing B.1: EMFatic-specified model of the blogging system tutorial

```
package com.googlecode.pongo.examples.miniblog.model;

@db
class Blog {
    val Post[*] posts;
    val Author[*] authors;
}

class Post {
    @searchable
    attr String title;
    attr String body;
    ref Author author;
    val Comment[*] comments;
}

class Comment {
    attr String from;
    attr String text;
    val Comment[*] replies;
}

class Author {
    @searchable
    attr String name;
    attr String email;
}
```

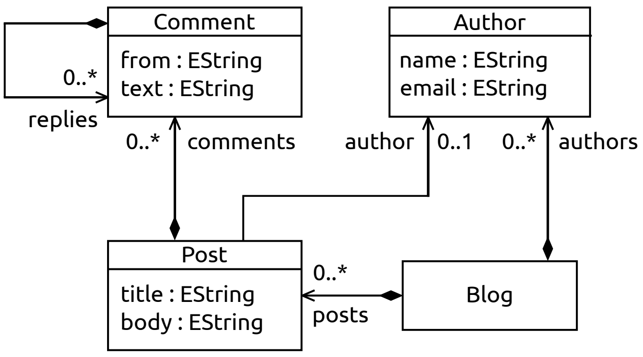


Figure B.1: Ecore model of the blogging system tutorial

The definition of the M2T transformations for the blogging system are based on EGX, EGL, and EOL files. By executing the transformations, six Java files are generated. These Java classes represent the domain model (see Listing B.1 and Figure B.1) and define helper methods (e.g., getter and setter) to conveniently work with the MongoDB database (e.g., for reading and writing data). The Pongo tutorial [69] provides a Java application used for testing the generated Java classes (see Listing B.2).[15] If the M2T transformation from the Ecore-based domain model into Java classes was successfully, the application should execute without any errors.

Listing B.2: Test application for the generated Java classes

```java
1  package com.googlecode.pongo.examples.miniblog;
2
3  import java.util.*;
4  import com.mongodb.*;
5  import com.googlecode.pongo.runtime.*;
6  import com.googlecode.pongo.examples.miniblog.model.*;
7
8  public class App {
9
10     public static void main(String[] args) throws Exception {
11         App app = new App();
12         app.writeStuff();
13         app.readStuff();
14     }
15
16     public void writeStuff() throws Exception {
17         Mongo mongo = new Mongo();
18         mongo.dropDatabase("blog");
19         Blog blog = new Blog(mongo.getDB("blog"));
20
21         Post post = new Post();
22         post.setTitle("A post");
23
24         Author author = new Author();
25         author.setName("Joe Doe");
26         blog.getAuthors().add(author);
27         post.setAuthor(author);
28
29         Comment comment = new Comment();
30         comment.setText("A comment");
31         post.getComments().add(comment);
32
33         Comment reply = new Comment();
34         reply.setText("A reply");
35         comment.getReplies().add(reply);
36
37         blog.getPosts().add(post);
38
39         // Syncs with the underlying database
40         blog.sync(true);
41     }
42
43     public void readStuff() throws Exception {
44
```

[15]Please note that the originally provided App Java class [69] contains a typing error in line 1 which is corrected in Listing B.2.

```
        Mongo mongo = new Mongo();
        Blog blog = new Blog(mongo.getDB("blog"));

        System.out.println(blog.getPosts().size());

        for (Post post : blog.getPosts()) {
            System.out.println(post.getTitle() + " - " + post.getAuthor().getName());
            System.out.println(post.getComments().size() + " comment(s)");
        }

        // Generated from the @searchable title attribute of Post
        System.out.println(blog.getPosts().findOneByTitle("A post").getAuthor().getName
            ());
    }

}
```

B.2.2 Execution

This section details how we proceeded with the case study by applying our approach to rewrite M2T transformations to provide Pongo with the functionality to operate on UML models. During the case study, we applied our rewriting approach in two iterations (*UML-Pv1* and *UML-Pv2*). Issues which emerged from the first iteration were handled by improving and extending our approach and by executing the evolutionary scenario a second time.

First Iteration (UML-Pv1)

In order to allow for automatic rewriting of generator templates, we have to specify the evolutionary process from the Ecore to the UML metamodel in terms of M2M transformations. To show the necessary transformation operations, we adopt the notion of *mapping diagrams* from [40]. Mapping diagrams provide "a high-level design view" [40] on transformations, thereby abstracting from concrete transformation languages. Figure B.2 shows the *Ecore2UML* transformation operations implemented while performing the case study.[16]

[16]The notation of mapping diagrams defined in this thesis can be summarized as follows: vertical rectangles on the left and right with a gray background color represent the Ecore and UML metamodels; the vertical rectangle in the middle with a slightly darker gray background color groups the transformation rules; model elements are represented as rectangles; transformation rules are represented as hexagons; the transformation is executed in the direction of the arrow; arrow ends represent input and output elements; a label near an arrow represents a property of an element (i.e. the value of the property named as the label is set to the corresponding element the arrow points to); a rectangle with a dashed border represents conditions (defined as simplified Epsilon statements); a guard constraints output elements; a set-condition defines values for properties of output elements where the value type is neither an element of the Ecore or the UML metamodel (e.g., boolean types specified in a dedicated model omitted for clarity; the semantics of a labeled arrow is identical for elements defined in either the Ecore or the UML metamodel).

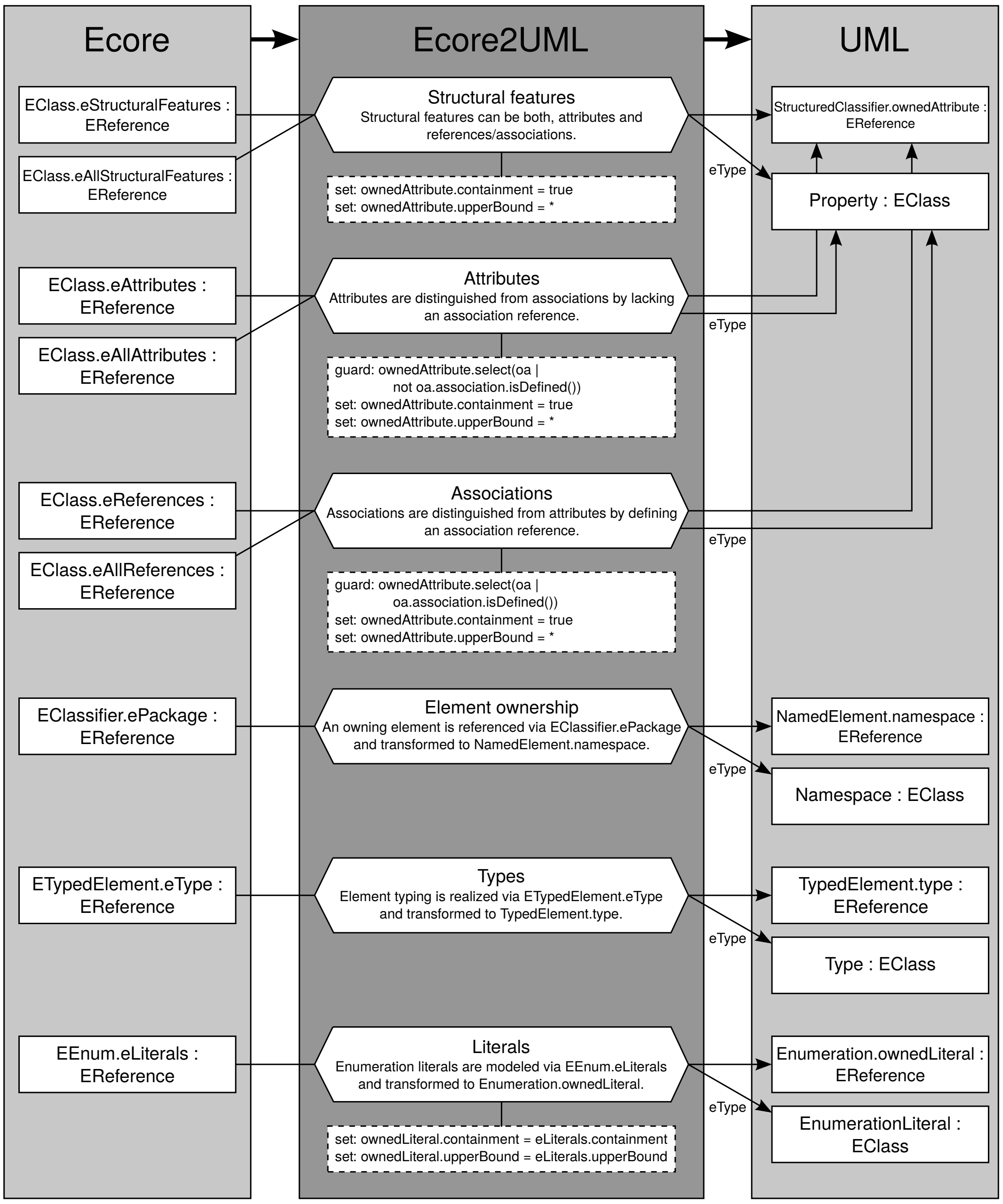

Ecore
Ecore2UML
UML
EClass.eStructuralFeatures : EReference
EClass.eAllStructuralFeatures : EReference
Structural features
Structural features can be both, attributes and references/associations.
set: ownedAttribute.containment = true
set: ownedAttribute.upperBound = *
StructuredClassifier.ownedAttribute : EReference
eType
Property : EClass
EClass.eAttributes : EReference
EClass.eAllAttributes : EReference
Attributes
Attributes are distinguished from associations by lacking an association reference.
eType
guard: ownedAttribute.select(oa | not oa.association.isDefined())
set: ownedAttribute.containment = true
set: ownedAttribute.upperBound = *
EClass.eReferences : EReference
EClass.eAllReferences : EReference
Associations
Associations are distinguished from attributes by defining an association reference.
eType
guard: ownedAttribute.select(oa | oa.association.isDefined())
set: ownedAttribute.containment = true
set: ownedAttribute.upperBound = *
EClassifier.ePackage : EReference
Element ownership
An owning element is referenced via EClassifier.ePackage and transformed to NamedElement.namespace.
NamedElement.namespace : EReference
eType
Namespace : EClass
ETypedElement.eType : EReference
Types
Element typing is realized via ETypedElement.eType and transformed to TypedElement.type.
TypedElement.type : EReference
eType
Type : EClass
EEnum.eLiterals : EReference
Literals
Enumeration literals are modeled via EEnum.eLiterals and transformed to Enumeration.ownedLiteral.
Enumeration.ownedLiteral : EReference
eType
EnumerationLiteral : EClass
set: ownedLiteral.containment = eLiterals.containment
set: ownedLiteral.upperBound = eLiterals.upperBound

Figure B.2: Mapping diagram of the Ecore2UML metamodel transformation operations

Please note that, in general, we focus only on those software artifacts which are essential for the successful execution of the case study. By doing so, we filter out irrelevant details, such as, potential further Ecore2UML transformation rules or analyzing additional EGL templates which are included in the `com.googlecode.pongo` project (e.g., for drawing charts). Disruptive factors for the measurement of metrics are factored out, as well. For example, the metric (2.) *number of called helpers* count only helper operations called at least once during M2T transformation, even though additional operations exist in the accompanying EOL files (but are insignificant for the execution of our case study). By manually analyzing the M2T transformation definitions in the `com.googlecode.pongo` project, we concluded that six transformation rules are required to represent the Ecore2UML metamodel evolution sufficiently to rewrite all relevant parts of the EGL templates (see also Figure B.2). The transformations are employed to deal with syntactical heterogeneity between the Ecore and UML metamodels, in particular, differences according to the source-target-concept cardinality (1:1 and n:1) as well as naming, context, multiplicity, and containment differences of the same metamodeling concept (`EReference`) [177]. The mapping diagram (see Figure B.2) summarizes the six transformation operations as follows:

- *Structural features*: The first three rules transform six different elements of type `EReference` in the Ecore metamodel into one and the same `UML::StructuredClassifier.ownedAttribute` element. In Ecore, `EClass.e(All)StructuralFeatures` references attributes and references, while `EClass.e(All)Attributes` and `EClass.e(All)References` allow access to the individual types. In contrast, in UML, attributes and associations are represented by the same `ownedAttribute Property` of a `StructuredClassifier` (they can be distinguished by querying additional properties; see below). Please bear in mind that the Ecore2UML M2M transformation operates on the M2 layer and that the Ecore as well as the UML metamodel are defined (reflective) via Ecore (see Section 2.2). Hence, the same metamodeling infrastructure is employed for both metamodels (e.g., `UML::StructuredClassifier.ownedAttribute` is also of type `Ecore::EReference`). The transformation rule for structural features (as well as for attributes and associations; see below) defines that the `eType` property of `UML::StructuredClassifier.ownedAttribute` references the `EClass UML::Property`. Additionally, `UML::StructuredClassifier.ownedAttribute` is defined as a containment reference and the upper bound is unlimited.
- *Attributes*: The same transformation rules as for *structural features* apply, with the exception that attributes are distinguished from associations by querying the `association` property of `UML::StructuredClassifier.ownedAttribute` elements. For matching an attribute, the `association` reference must not be defined.
- *Associations*: The same transformation rules as for *structural features* apply, with the exception that associations are distinguished from attributes by

querying the `association` property of `UML::StructuredClassifier.ownedAttribute` elements. For matching an association, the `association` reference must be defined.

- *Element ownership*: The equivalent modeling construct for the `EReference Ecore::EClassifier.ePackage` was identified as the `EReference UML::TypedElement.type`. Both `EReferences` point to the corresponding owning element (`Ecore::EPackage`, `UML::Namespace`). The `eType` property references the `EClass UML::Namespace`.

- *Types*: The `EReference Ecore::ETypedElement.eType` is transformed to the `EReference UML::TypedElement.type`. The `eType` property references the `EClass UML::Type`.

- *Literals*: The Ecore `EReference EEnum.eLiterals` is transformed in its UML equivalent `Enumeration.ownedLiteral`. The `eType` property references the `EClass UML::EnumerationLiteral`; containment and upper bound properties are preserved.

We designed the Ecore2UML M2M transformation in a way that all model elements resemble the original Ecore-specified UML metamodel. For this purpose, we define a factory in the corresponding ETL file (as a `pre` named block of Epsilon statements executed before the transformation operations) for creating non-equivalent UML elements not generated by matching transformation rules (see project `at.ac.wu.nm.dsml.eval.hotm2t`). Thereby, we constructed a smallest possible subset of the UML metamodel with which the case study can be performed, as well. Thus, for the case study, the UML metamodel can be fully replaced by its subset metamodel (named *UMLExcerpt*). We use the UMLExcerpt metamodel solely to improve readability and navigability aspects. For instance, the UMLExcerpt metamodel consists of only 32 `EClasses` compared to 242 `EClasses` contained in the whole Ecore-based UML metamodel implementation (a reduction by ∼87%). Please note that the rewritten M2T transformation templates are equally applicable on model instances of the whole UML metamodel and on its subset (UMLExcerpt) without any modifications.

Our approach uses traces collected from the execution of the Ecore2UML M2M transformation for the generation of rewriting rules for the model representations of EGL templates (we omit the source code of the extended and improved EGL-based generation of ETL rewrite rules for brevity; see project `at.ac.wu.nm.dsml.eval.hotm2t`). The generated rewrite rules target syntactical heterogeneity in EGL models. Structural differences are tackled via 1:1 source-target-concept cardinality adaptations of the same metamodeling concepts (references). In particular, the transformations correct naming and constraint differences of model references resulting from the diverse Ecore and UML metamodel definitions (implicitly changing the context of the references, as well) [177]. An excerpt from the ETL file covering the rewriting rules created from the six transformation operations is shown in Listing B.3. All of our Ecore2UML M2M transformations target `EReferences` exclusively (see Figure B.2). Therefore, in the Pongo EGL template models, the `property` attribute of matching `PropertyCallExpression` elements must be adapted correspondingly (see lines 6–14 in Listing B.3).

Listing B.3: Generated rewrite rules from the Ecore2UML transformation

```
 1 rule PropertyCallExpression
 2   transform s : egl_in!PropertyCallExpression
 3   to t : egl_out!PropertyCallExpression
 4   extends FeatureCallExpression {
 5     switch (s.property) {
 6       case "eReferences": t.property = "ownedAttribute.select(oa | oa.association.
              isDefined())";
 7       case "eAllReferences": t.property = "ownedAttribute.select(oa | oa.association.
              isDefined())";
 8       case "eAttributes": t.property = "ownedAttribute.select(oa | not oa.association.
              isDefined())";
 9       case "eAllAttributes": t.property = "ownedAttribute.select(oa | not oa.association
              .isDefined())";
10       case "eStructuralFeatures": t.property = "ownedAttribute";
11       case "eAllStructuralFeatures": t.property = "ownedAttribute";
12       case "ePackage": t.property = "namespace";
13       case "eType": t.property = "type";
14       case "eLiterals": t.property = "ownedLiteral";
15       default: t.property = s.property;
16     }
17     t.extended = s.extended;
18 }
```

With the generated rewrite rules from Listing B.3, we can now adapt the model representations of the Pongo EGL templates created with our approach to work with UML models as well (code/model round-tripping functionality is provided by the `org.eclipse.epsilon.egl.dom*` projects). To test the rewritten EGL templates, we converted the Ecore-based domain model of the blogging system to a UML class diagram via the functionality provided in the Sample Ecore Model Editor of EMF.[17] By generating EGL template code from the rewritten model representations and by executing the corresponding M2T transformations over the UML instance model (representing the blogging system domain), we encountered two issues: 1) Methods specific to the Ecore (metamodel) implementation are not present when working with the UML metamodel and 2) our approach does not target EGX and EOL files which accompany the M2T transformation definitions in our case study (i.e. structural mismatches in EGX/EOL files resulting from the metamodel evolution cannot be tackled via automatic rewrites). We handled these issues by improving and extending our approach and by executing the evolutionary scenario a second time (UML-Pv2).

[17]We omit showing the resulting UML class model, as its diagram representation is almost identical to the Ecore model presented in Figure B.1 (except that all `EString` types would be replaced by `String` types).

Second Iteration (UML-Pv2)

Regarding the first problem encountered during the initial iteration (UML-Pv1), in particular, two attributes are queried (`isMany` and `isContainment`) which are specific to the Ecore metamodel implementation (and, thus, are not present in the UML metamodel implementation).[18] Attribute `isMany` indicates whether more than one value may appear in a valid instance and attribute `isContainment` indicates whether a reference represents by-value content. In order to deal with this discrepancy, we define temporary *proxy* elements in the Ecore metamodel which are transformed to proxy elements in the UML metamodel by additional transformation rules. These proxy elements are deleted from the metamodels right after the corresponding transformation operation was performed and the associated transformation data was stored in the trace model. Hence, the proxy elements are not persisted and used solely for the purpose of generating rewrite rules to handle attributes specific to the Ecore implementation.

As both Ecore-specific attributes (`isMany` and `isContainment`) can equally be expressed as Epsilon statements, we define two EOL helper operations (`isMany()` and `isContainment()`) to be called instead (see Listing B.4). With the proxy elements and corresponding transformation traces, we can generate rules rewriting the Ecore-specific attribute queries to UML-specific helper operation calls. For this purpose, we define two additional Ecore2UML M2M transformation operations (see Figure B.3). They transform the proxy `EAttribute` elements `Ecore::isMany` and `Ecore::isContainment` into the proxy `EOperation` elements `UML::isMany()` and `UML::isContainment()`. Thus, the transformations tackle syntactical heterogeneity of the introduced proxy elements via 1:1 source-target-concept cardinality adaptations of naming and structural differences between different metamodeling concepts (`EAttribute` and `EOperation`) [177]. The traces collected from the execution of the two transformation operations are used to generate the additional rewriting expressions as shown in Listing B.5. In the actual implementation of our approach, these two lines are included in the ETL file responsible for defining the rewrite rules (the insertion place corresponds to line 12 in Listing B.3).

Listing B.4: Epsilon helper operations for Ecore-specific attributes

```
@cached
operation Any isMany() {
  return (self.upper > 1 or self.upper == -1);
}

@cached
operation Property isContainment() {
  return (self.aggregation.name == "composite");
}
```

[18]The Ecore implementation provides also corresponding getter methods for both attributes via the `ETypedElement` interface (`isMany()`) and the `EReference` interface (`isContainment()`).

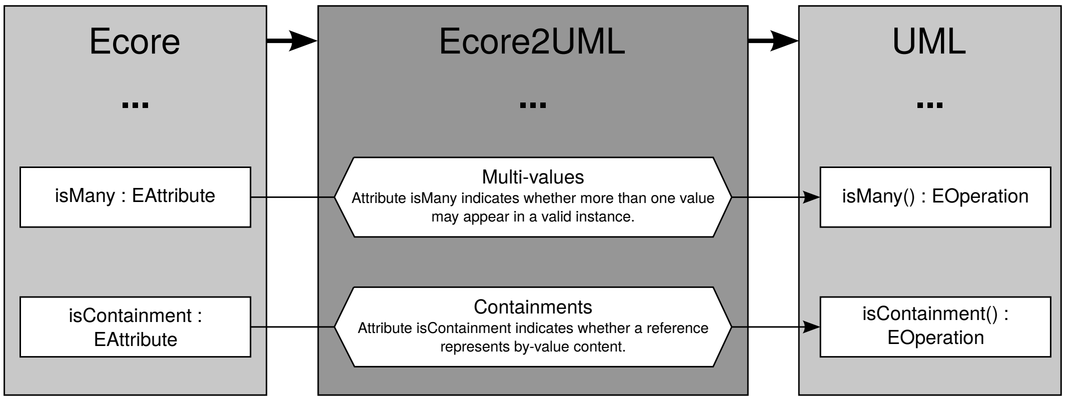

Ecore
...
isMany : EAttribute
isContainment : EAttribute
Ecore2UML
...
Multi-values
Attribute isMany indicates whether more than one value may appear in a valid instance.
Containments
Attribute isContainment indicates whether a reference represents by-value content.
UML
...
isMany() : EOperation
isContainment() : EOperation

Figure B.3: Mapping diagram of the Ecore2UML metamodel transformation operations for proxy model elements

Listing B.5: Additionally generated rewrite expressions from the Ecore2UML proxy elements transformations

```
    case "isContainment": t.property = "isContainment()";
    case "isMany": t.property = "isMany()";
```

Regarding the second issue which emerged from the first iteration (i.e. our approach does not target EGX and EOL files which accompany the M2T transformation definitions in the case study), our current approach works with EGL templates only. Therefore, refactorings in EGX and EOL files have to be performed manually (see also Section B.2.3).

After we dealt with the Ecore-specific problems, we re-executed the Ecore2UML M2M transformation, received a new set of rewrite rules, applied these rewrite rules to the Pongo EGL models, transformed the models back to their initial EGL-based template form, and performed the M2T transformation for the UML-based blogging system once again. This time, the M2T transformation finished without any errors. We compared the Java classes generated from the UML class model with the original Java classes generated from the Ecore model (by using `diff`). As the output artifacts of both M2T transformations were identical, the refactorings had been successful. In the following analysis section, we report on the quantitative data obtained from employing our approach (for each of the two iterations separately: with and without proxy elements; i.e. UML-Pv1 and UML-Pv2) as well as on qualitative aspects not discussed so far.

B.2.3 Analysis

In order to provide a comparative analysis of the evolution of the Pongo project to be UML compliant, we collected the following quantitative data (see Section B.1.4): (1.) Number of manually/automatically rewritten concepts and (2.) number of unique concepts for the refactorings performed when employing our approach—once neglecting Ecore-specific methods (*UML-Pv1*), once with our solution of proxy elements (*UML-Pv2*). We compare these data to the (manual) refactorings required without employing our approach (*UML-NoP*). Furthermore, we look at differences between the evolved Pongo M2T transformations and the original definitions (*Ecore-Orig*) with respect to the number of called helpers, number of calls to helpers, and the transformation size (metrics 3.–5.; see Section B.1.4). In addition, we report on the frequency of EGL model elements as well as on average execution times of M2T transformations, analyze manually performed refactorings in EGX and EOL files, and discuss how to deal with diverging modeling concepts between the Ecore and UML metamodels.

Table B.2 shows the number of manually/automatically rewritten concept occurrences in the Pongo EGL templates/models without as well as with employing our approach (distinguished by using proxy elements or not). As our current approach does not support other Epsilon dialects than EGL, necessary refactorings in EGX and EOL files had to be performed manually for each of the three configurations. As these figures are identical for every presented variant, they do not add any relevant information for the comparison and, thus, are omitted. In total, 60 rewriting operations had to be performed to refactor the M2T transformations to work with UML class models. Table B.2 shows that we could automate all EGL-based rewriting operations in the UML-Pv2 iteration. As the Ecore-specific attributes (`isMany` and `isContainment`) are not handled in the UML-Pv1 iteration, only ~68% of the necessary rewriting operations could be automatically refactored via the preliminary implementation of our approach.

Table B.2: Number of manually/automatically rewritten concepts and unique concepts with/without our approach

Metric	UML-NoP	UML-Pv1	UML-Pv2
1. Number of manually rewritten concepts	60/60 (100%)	19/60 (32%)	0/60 (0%)
1. Number of automatically rewritten concepts	0/60 (0%)	41/60 (68%)	60/60 (100%)
2. Unique concepts	10	10 (8)	10

The total of 60 rewriting operations are based on ten unique concepts (see Table B.3). On average (arithmetic mean), every refactored concept occurs six times in the corresponding EGL templates. For the UML-Pv1 iteration, eight of these unique concepts are automatically rewritten (see Table B.2)—the other two being `isMany` and `isContainment`. These two Ecore-specific attributes are among the top three frequently found concepts (only `eType` occurred more frequently).

Table B.4 shows the number of called helpers, the number of calls to helpers, and the transformation sizes (SLOC) divided into EGX, EGL, and EOL files for the original Ecore- and the evolved UML-based M2T transformations. The increase of two helper operations (to twelve) as well as in the number of calls to helpers (305;

Table B.3: Rewrite frequency of Ecore-based model concepts

Ecore concept	Rewrite frequency
`eType`	22
`isMany`	12
`isContainment`	7
`eReferences`	5
`ePackage`	4
`eAllReferences`	3
`eAttributes`	3
`eStructuralFeatures`	2
`eAllAttributes`	1
`eLiterals`	1
$\sum$	60

metrics 3. and 4.) for all UML-based refactorings has its cause in the additional EOL helper operations `isMany()` and `isContainment()` required to reproduce the Ecore-specific implementation.

Table B.4: Number of called/calls to helpers and transformation sizes for original Ecore- and evolved UML-based M2T transformations

Metric	Ecore-Orig	UML-NoP	UML-Pv1	UML-Pv2
3. Number of called helpers	10	12	12	12
4. Number of calls to helpers	289	305 (352)	305 (352)	305 (352)
5. Transformation size (SLOC)				
EGX	60	60	60	60
EGL	237	237	272	272
EOL	153	173	173	173
$\sum$	450	470	505	505

To optimize the execution of operations, Epsilon supports caching the results of parameter-less operations using the `@cached` annotation [70]. As we define the two helper operations as cached (see lines 1 and 6 in Listing B.4), they are only executed once for each distinct object and subsequent calls on the same target return the cached result. Therefore, the bodies of the two operations are called eleven (`isMany()`) and five times (`isContainment()`), respectively, compared to the 48 and 15 times, they would be called when the operations are not declared cached (summing up to a total number of 352 calls to helpers; see Table B.4). In contrast, queries to the Ecore attributes `isMany` and `isContainment` do not add to the number of calls to helpers of the Ecore-Orig variant.

Regarding the transformation sizes, all EGX files have the same amount of SLOC (see Table B.4). EGL parts are optimized in the refactorings of UML-NoP (please note that 237 SLOC for both, the Ecore-Orig and the UML-NoP configurations is a coincidence). The higher amount of SLOC of EGL files for the UML-Pv1 and UML-Pv2 iterations (272) are due to the output formatting of M2T transformations of our implementation (provided by the code/model round-tripping functionality). The increase of SLOC of EOL files in all three evolved versions (173) are caused by refactorings necessary to work with the UML metamodel (e.g., the definition of the additional two helper operations). All four M2T transformation definitions generate the identical set of Java files consisting of a total of 282 SLOC.

As our approach provides code/model round-tripping functionality for EGL, we can measure M2T transformation specifications also in terms of model elements. Table B.5 shows the frequency of model elements of the UML-Pv2 Pongo M2T transformations (more information on specific model elements can be found in [53]). Our rewriting operations target model element `PropertyCallExpressions` exclusively (see Section B.2.2). In the case study, our approach needs to automatically rewrite 60 (see Table B.2) of the total 153 `PropertyCallExpressions` (∼39%) to render the EGL templates UML compliant. The remaining `PropertyCallExpressions` can be reused as-is (e.g., the attribute `name` is contained in both, `Ecore::ENamedElement` and `UML::NamedElement`).

Table B.5: Frequency of EGL model elements of the UML-Pv2 M2T transformations

Model element	Frequency
DomElement	2868
Expression	2209
FeatureCallExpression	863
FLOMethodCallExpression	14
MethodCallExpression	696
PropertyCallExpression	153
LiteralExpression	500
PrimitiveExpression	500
NameExpression	804
OperatorExpression	13
BinaryOperatorExpression	6
UnaryOperatorExpression	7
VariableDeclarationExpression	29
Import	6
Program	4
Statement	649
ExpressionStatement	628
ForStatement	15
IfStatement	6

Table B.6 provides an overview of the average execution times (in milliseconds; ms) of the different transformations performed during the case study.[19] We em-

[19]Execution times are measured on the following hardware and with the following software specifications: Intel Core i5-3320M CPU 2.6 GHz, 12 GB RAM, 64-bit Ubuntu 13.04, Eclipse 4.2.

ployed the Epsilon profiling mechanism [72, 116] as well as Java's `System.nanoTime()` method for measuring the time needed to execute a particular transformation. We report the arithmetic mean of executing every transformation ten times.[20]

Table B.6: Average execution times of the different transformation phases

Transformation	Average execution time (in ms)
Pongo EGL templates to EGL models	1266±45
Rewriting EGL models	4263±42
Rewritten EGL models to EGL templates	1051±23
Ecore/UML model to Java code	
EMFatic-based model	1974±43
Ecore-based model	1861±54
UML-based model	3058±69
UMLExcerpt-based model	3022±46

The actual rewriting of the EGL models (i.e. executing the generated ETL rules) requires 2520±32ms; that is ∼59% of the total execution time of 4263±42ms. The remaining time is required by loading needed models, performing Ecore2UML M2M transformations (comparing, merging, transforming elements) etc. The time required for generating the ETL rewrite rules from the recorded trace data is negligible (28±2ms; ∼0.7% of the total 4263±42ms).

For the Java code generation from Ecore/UML-based models (defining the blogging system), the Ecore-Orig variant works with either an EMFatic specification (which has to be transformed into an Ecore model first) or with Ecore models directly. Hence, by defining models in the EMFatic textual syntax the average execution time of generating Java code slightly increases to 1974±43ms compared to using Ecore models directly (1861±54ms; see Table B.6). Regardless of basing UML instance models on the whole UML metamodel or only on a small subset of it (UMLExcerpt), the average execution time of generating Java classes increases to nearly the same amount of 3058±69ms and 3022±46ms, respectively (an increase of ∼53–64% compared to using EMFatic/Ecore models).

We could identify the relevant factor responsible for the execution time differences being the respective metamodel registering statements in the orchestrating Java class of the M2T transformation (file `PongoGenerator.java` in project `com.googlecode.pongo.uml`). Registering the metamodels requires 557±11ms (Ecore) and 1603±43ms (UML/UMLExcerpt), respectively. The difference of 1046ms in the average numbers almost matches the average time gap of 1161–1197ms when comparing the whole execution times of the Ecore and the UML/UMLExcerpt model to Java code transformations. The remaining execution times of the model to Java code transformations without the phase of registering the corresponding metamodels account for 1305±57ms (Ecore) and 1455±69ms (UML/UMLExcerpt). Please note that the increase of up to ∼64% when comparing the execution times of the Ecore-

[20]The value after ± shows the standard deviation (σ) from the arithmetic mean ($\bar{x}$).

with the UML-based M2T transformations is not caused by the application of our approach (i.e. the rewritten EGL templates), but solely by the different times needed to register the corresponding metamodels.

In total, the average execution time of the complete case study sums up to 9638ms (UML-based instance model) and 9602ms (UMLExcerpt-based instance model), respectively.

As mentioned earlier, our current approach does not support the automatic adaptation of EOL and EGX files, although, for the case study, Epsilon code written in both dialects needed refactorings. The required EGX-induced changes were marginal (e.g., model element type changes, such as, from `Ecore::EClass` to `UML::Class`) and are, in principle—with slight modifications—covered through our approach and the already generated rewrite rules (e.g., structural features and types; see Figure B.2). Adaptations to existing EOL helper operations primarily targeted `PropertyCallExpressions` and model element types, as well. In addition to the two new helper operations, comparatively larger changes had to be performed for three operations. These changes proved necessary to handle UML-specific data types (i.e. `PrimitiveTypes`). As in our case study we only deal with `PrimitiveTypes` of type `String` (see Listing B.1 and Figure B.1), we commented out the respective source code lines handling the remaining types to keep the adaptations to a minimum.

During the conversion of the Ecore-based domain model of the blogging system into a UML class diagram via the functionality provided in the Sample Ecore Model Editor of EMF, `eAnnotations` were not preserved. As these annotations contain relevant information (e.g., `@db`, `@searchable`; see Listing B.1), we manually re-included them into the UML class model in a different way. In order to refactor the domain model fully UML compliant, we mapped `Ecore::EAnnotations` to `UML::Comments`, although the EMF-based implementation of the UML metamodel root `Element` inherits from `Ecore::EModelElement` which contains the reference `eAnnotations` responsible for annotating model elements. Therefore, in principle, all UML models can also be annotated by using the inherited `EReference Ecore::EModelElement.eAnnotations` having `Ecore::EAnnotation` as its `eType` attribute. This change, induced by mapping Ecore to UML class models, was also reflected in the EOL files (adapting `Ecore::EModelElement.eAnnotations` to `UML::Element.ownedComment`) as well as the diverging inheritance relationship concepts between `Ecore::EClass.eSuperTypes` and `UML::Class.generalization`.

B.2.4 Interpretation

In this section, we interpret the data collected and discuss our observations made during the case study in order to answer the two guiding research questions defined in Section B.1.1.

Regarding Research Question 1 (*To which extent can our current approach be used in a real-life MDD-based software development project?*), we could successfully facilitate the reuse of M2T transformation definitions in an evolving metamodel scenario. However, it was required to extend our approach (e.g., to handle metamodel implementation specific characteristics) and—although the rewriting of the target Epsilon dialect (EGL) could be fully supported (see Table B.2)—the case study results showed that further improvements would be beneficial (e.g., support for other Epsilon dialects).

We interpret the execution times of the individual transformations and the overall process as satisfactory (see Table B.6). After all, the rewriting of M2T transformations and their execution are not time-critical as an evolutionary scenario is not considered to be performed on a daily basis. The implementation of our approach could handle models containing a combined number of almost 3000 elements (see Table B.5) in a moderate time frame.[21] A first experiment on basing the rewrite rules on ATL [156, 157] instead of ETL indicates the potential of faster execution times. Changing the M2M transformation language for rewriting EGL models (e.g., to ATL) can be implemented with a few SLOC by adapting the EGL-based rewrite rule generator. Please note that these aspects affect only the technical implementation (i.e. the prototype), but not the general concepts of our rewriting approach.

Extensions to our approach were developed to support implementation-specific properties of the Ecore metamodel (i.e. proxy elements). An alternative implementation for the additional helper operations would have been to encode the functionality of the helpers directly into the rewrite rules. For this case study, the alternative implementation would have also been a possible option because of the short Epsilon statements spanning only one line (see Listing B.4). The benefit of this alternative implementation being fewer number of called helpers, fewer number of calls to helpers, and a reduction in the transformation size (metrics 3.–5.). However, the internal caching mechanism of Epsilon (`@cached` annotation for operations) could then not be utilized and may result in a potential increase of the execution time of the UML model to Java code transformation. Furthermore, as the Epsilon statements query properties of model elements, their instance variable names used in the EGL templates need to be known at the time of rewriting. The variable names would also need to be homogenized (e.g., by defining naming conventions upfront) which is not realistic for an evolutionary scenario as the M2T transformations already exist. Moreover, the encapsulation of generically applicable functionality into explicitly defined helper operations facilitate their reuse in other scenarios and prevents the replication of statements providing one and the same functionality (which would then also be scattered across the code base). In addition, more complex statements (spanning over multiple lines)—although technically possible—risks decreasing the comprehensibility of the EGL code.

We also improved the EGL-based operations for generating rewrite rules to be more generically applicable (e.g., to facilitate the definition of new rewrite rules) and enabled the support for Epsilon's profiling mechanism. Regarding code/model round-tripping, the methods responsible for the individual transformations (code to model and model to code) were optimized and can now be individually executed (the

[21] In fact, we successfully performed an experimental rewriting of an EGL model containing ~15.000 elements.

execution time is recorded, as well). The creation of EGL template code from its model representation was enhanced, however, further improvements to preserve the layout during round-tripping might have the potential to reduce the SLOC of the output EGL code (see Table B.4).

Regarding Research Question 2 (*What are the observable and quantifiable advantages of applying our approach?*), we can summarize the measured benefits of applying our approach as follows. We were able to fully automate the evolution of EGL templates to support UML models in the UML-Pv2 iteration (metric 1.). In contrast, without proxy elements, ∼32% of EGL concepts needed manual adaptations (see Table B.2). The rewriting operations are based on a moderate number of unique concepts (ten; metric 2.). Thus, for this case study, employing other approaches not relying on transformation traces (e.g., tool-specific search-and-replace functionality) may be a potential option, as well. However, then, concepts used in M2T transformations would need to be matched manually with their original and evolved metamodeling concepts to establish the necessary rewriting operations—a tedious and error-prone task. To execute the rewriting operations, all concept occurrences would need to be identified. Although supported by tools (e.g., search-and-replace functionality of text editors), executing all necessary rewrite operations in one bulk operation may not be available, rendering the rewriting process as a sequence of multiple steps each requiring individual execution. Thus, even for scenarios with only a few unique concepts, our rewriting approach may be a time-saving option as it facilitates the reuse of rewriting operations and can easily be adapted to changed evolution requirements. Furthermore, the evolutionary process is explicitly documented by defining M2M transformations and by generating rewrite rules for the corresponding M2T transformations.

The application of our approach caused a moderate increase of additional control metrics. The number of called helpers increased by 20% (metric 3.). We could not find an appropriate solution to reduce this number (see above). The number of calls to helper operations (metric 4.) slightly increased by nearly 6% due to the benefits gained by the internal caching mechanism of the Epsilon language. Without cached operations (as may be the case when using other model management languages and technologies), the increase of the number of calls to helpers would be almost 22%. However, the difference in the number of calls to helpers between the original Ecore-based and all UML-based variants is only caused by the employed measurement method (as queries to Ecore internal attributes are not counted). Furthermore, regarding execution times, calling the helper operations and executing the short Epsilon statements included in their bodies is not computation expensive. The two helpers do not have a significant effect on the overall execution time, especially, as the EGL code/model round-tripping and the rewriting of EGL models as well as registering the UML/UMLExcerpt metamodel during the UML model to Java code transformation require significantly more time (see Table B.4). For instance, the EGL code/model round-tripping including the rewriting of EGL models account for more than 68% of the overall execution time and registering the UML/UMLExcerpt metamodel accounts for more than 52% of the whole execution time of the UML model to Java code transformation step.

The increase in the SLOC of the M2T transformations (∼4% for UML-NoP, ∼12% for UML-Pv1 and UML-Pv2; metric 5.) is partly due to the additional helper operations included in the EOL files and partly caused by the formatting instructions implemented in the model-to-code part of the code/model round-tripping for EGL. We already argued why we opted for introducing additional helpers (see above), thus, the SLOC of EOL files cannot be decreased. On the contrary, a layout-preserving round-tripping functionality for EGL may be beneficial for reducing the transformation size (see above).

B.3 Limitations and Threats

The particular definition of an evolutionary scenario in combination with the requirements for the software artifacts the case study is based on (open-source, publicly available, non-trivial M2T transformation definitions; see Section B.1.2) narrowed the potential candidate cases. The range of suitable, realistic cases to choose from was small and we had to select a case based on availability (which, in practice, is not exceptional for performing exploratory case studies [5]).

As one of the authors of the evaluated approach performed the case study, control was increased. Although required by the design of our study (see Section B.1.4), this might have introduced personal bias.

By complying to technological requirements, we draw conclusions particularly targeting Epsilon (e.g., cached helper operations) and EMF (e.g., Ecore-based models). Although we discuss implications independent of the implementation (see, e.g., Sections B.2.3 and B.2.4), the generalization of results may be limited. For example, we have a clear understanding on how to extend the implementation of our approach to handle other Epsilon dialects, as well (e.g., EGX, EOL), but currently we have no empirical evidence of its application to non-EGL-based languages.

In our case study, we measured certain metrics in order to evaluate specific advantages of our approach, thereby neglecting other quality aspects (e.g., usability, stability etc.). As our metrics are adopted from related work (see, e.g., [67, 164, 165, 166, 169]), we relied on the authors' demonstration that the metrics are acceptable for their intended use (in the context of model transformations). It was not the goal of the case study to validate the adopted metrics as a representative quantitative measurement for a given attribute (for a discussion on validating software metrics, see, e.g., [80]).

Our case study collects qualitative and quantitative data for a MDD-based scenario of evolving metamodels (i.e. Ecore to UML). Although the EMF-based implementation of these two metamodels differ, they share similar concepts (e.g., classes, attributes, generalizations). This overlap influenced the required refactorings and in a replication case study (to validate our results) the extent of metamodel concept heterogeneity could be increased.

B.4 Summary of Results

The case study was performed in order to evaluate our approach of higher-order rewriting of M2T templates developed in this thesis project. The design of the case study was guided by two research questions: A qualitative assessment of the feasibility of our approach in a real-life MDD-based scenario (Research Question 1) and measurements to quantify the benefits of the application (Research Question 2). To answer Research Question 2, adequate metrics have been identified by adopting the GQM method.

A suitable case was selected based on availability of existing projects to meet the requirements for the case study (open-source, publicly available, non-trivial M2T transformation definitions). In particular, the study was based on the Pongo project, a Java POJO generator for MongoDB, and an evolutionary scenario to migrate Pongo to be able to work with UML class models in addition to already supported Ecore models.

By analyzing and interpreting the results obtained from executing the case study, we collected evidence that our current approach can be employed in a real-life project and that expected advantages materialize and can be observed, measured, and quantified. In particular, we could automatically rewrite all EGL-based concepts for their reuse in evolved M2T transformations of the Pongo project. However, to eliminate the need for manual refactorings in EGL generator templates completely, we had to handle attributes characteristic to the Ecore implementation of EMF. We came up with a solution of defining temporary proxy elements in the Ecore metamodel which are transformed into temporary proxy elements in the UML metamodel.

We could observe that the application of our rewriting approach did not significantly influence defined control metrics. From this perspective, we can conclude that the benefits of applying our approach from the viewpoint of a DSML engineer (i.e. fully reusing generator templates by rewriting EGL concepts) implies no substantial negative aspects. Hence, with regard to the studied case, the achievable reuse of M2T transformation definitions is satisfactory.

While performing the case study, we evaluated our rewriting approach and demonstrated the feasibility of our prototype. During the execution, we identified extension points (e.g., to handle proxy elements) and potential improvements (e.g., with regard to the generation of rewrite rules, to code/model round-tripping functionality) of our prototypical implementation. Inline with our design-science-research approach, the extensions and improvements resulted in a revised design of our software artifact (i.e. our prototype) which was released as a new version (publicly available at [48]).

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.