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 DSML 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, behav-
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 specialization 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 se-
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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DSML core language model and constraints</td>
<td>UML metamodel extension &amp; UML profile, OCL constraints</td>
<td>UML metamodel extension, UML profile, OCL constraints</td>
</tr>
<tr>
<td>DSML concrete syntax</td>
<td>New diagrammatic elements &amp; UML stereotype, textual notation</td>
<td>New diagrammatic elements, UML stereotypes</td>
</tr>
<tr>
<td>DSML behavior</td>
<td>Textual descriptions, example models</td>
<td>Textual descriptions, example models</td>
</tr>
<tr>
<td>DSML platform integration</td>
<td>EGL generator templates; direct transformation; Java code</td>
<td>API-based generator; indirect transformation; BPEL, WSDL, WS-SecurityPolicy specifications</td>
</tr>
</tbody>
</table>

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; 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
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 de-
fine extra graphical specifications for the integration
profile SOF::Services+SecurityAudit.

![Figure 3. Stereotype and icon definitions for the concrete syntax.](image)

### 4.3 Behavior Specification

For the integration profile (see Section 4.1), we spec-
ify a UML state machine to define the composed be-
havior 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 integra-
tion profile SOF::Services+SecurityAudit merges
these constraints as well as the language mod-
els (see Section 4.1). We define that for the
SOF::Services+SecurityAudit profile, first, the con-
straints 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).

![Figure 4. Integrated DSML behavior specification as a UML state machine.](image)

### 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 gener-
ator of DSML B to issue a Web Service call every time
a secure object flow is instantiated (see Figure 5').
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.](image)

### 5 Related Work

In [14], we present an approach for developing DSLs systemat-
ically. 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. How-
ever, the reflection on DSL design decisions in [7] re-
lates 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 con-
crete graphical syntaxes).

The composition of DSML models (i.e., of meta-
models) has been frequently addressed. Emerson and
Sztipanovits [16] review current metamodel compo-
sition methods (e.g., merge, refinement, interfacing).
This research strand is limited to the phase of integrat-
ing 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


343


