Reusable Textual Styles for Domain-Specific Modeling Languages*

Patrick Neubauer[1 [0000−0002−9811−4772]], Robert Bill[2 [0000−0001−9914−3287]], Dimitris Kolovos[1 [0000−0002−1724−6563]], Richard Paige[3 [0000−0002−1978−9852]], and Manuel Wimmer[4 [0000−0002−1124−7098]]

[1] University of York, Deramore Lane, York YO10 5GH, UK
{forename.lastname}@york.ac.uk
[2] Technische Universität Wien, Karlsplatz 13, 1040 Vienna, Austria
bill@big.tuwien.ac.at
[3] McMaster University, 1280 Main Street West, Hamilton, Ontario L8S 4L8, Canada
paigeri@mcmaster.ca
[4] Johannes Kepler Universität, Altenberger Straße 69, 4040 Linz, Austria
manuel.wimmer@jku.at

Abstract. Domain-specific languages enable concise and precise formalization of domain concepts and promote direct employment by domain experts. Therefore, syntactic constructs are introduced to empower users to associate concepts and relationships with visual textual symbols. Model-based language engineering facilitates the description of concepts and relationships in an abstract manner. However, concrete representations are commonly attached to abstract domain representations, such as annotations in metamodels, or directly encoded into language grammar and thus introduce redundancy between metamodel elements and grammar elements. In this work we propose an approach that enables autonomous development and maintenance of domain concepts and textual language notations in a distinctive and metamodel-agnostic manner by employing *style models* containing grammar rule templates and injection-based property selection. We provide an implementation and showcase the proposed notation-specification language in a comparison with state of the art practices during the creation of notations for an executable domain-specific modeling language based on the Eclipse Modeling Framework and Xtext.

Keywords: Domain-Specific Language · Model-Driven Engineering · Language Engineering · Concrete Syntax · Notation · Domain-Specific Modeling

1 Introduction

The engineering of a domain-specific language (DSL) is usually initiated by the construction of an artifact that captures concepts and relationships inherent to the domain being represented. Typical artifact types include variations of grammars and

* This work is supported by the European Commission via the CROSSMINER H2020 project grant number 732223, the Austrian agency for international mobility and cooperation in education, science and research (OeAD), grand number ICM-2016-04969, and the Christian Doppler Forschungsgesellschaft (CDG).

metamodels—each inherently of different nature [15]. In general, grammars are employed to describe domain concepts and their textual representation utilizing production rules and terminal rules, respectively. Contrarily, metamodels are used to capture concepts and relationships of a domain but not their syntactic constructs. Although state of the art language workbenches, such as Xtext [5], provide means to generate grammars from metamodels and vice-versa, they provide a single (default) notation, i.e. either graphical, textual, or a combination thereof, that has to fit the needs of all types of domain experts or requires dedicated language engineering skills for adaptation and extension. The construction of a bridge between metamodel and grammar, and in particular from metamodel to grammar, is commonly approached by introducing annotations in metamodels or metamodel-to-grammar transformations [2]. However, construction and maintenance of such bridges is inherently complex and error-prone due to fundamental differences between metamodels and grammars. Moreover, such bridges are often metamodel-dependent and are thus not universally applicable to arbitrary domains.

In this work, we present the Ecore Concrete Syntax Specification (Ecss) framework[5]—a novel textual notation description language and toolkit that enables the definition of both metamodel-dependent and metamodel-agnostic representations for Ecore-based languages—and employ it for automating the generation of textual modeling languages with supporting editors and tools from Ecore metamodels. In summary, Ecss facilitates (i) the creation, extension, and reuse of textual notations (subsequently also referred to as "style models" and "Ecss models") and (ii) the generation of grammar and executable implementation of domain-specific modeling languages (DSMLs) from pairs that consist of domain metamodel and style model. We showcase the implementation of our approach in a comparison with the state of the art in model-driven language engineering the results in the manifestation of a DSML.

Roadmap. The remaining sections of this paper present (i) a brief overview of methodologies and techniques upon which this work is build, (ii) a motivating example, (iii) the conceptual and technical characteristics of our approach alongside state-of-the-art solutions and in particular Xtext and the Eclipse Modeling Framework (EMF), (iv) a selection of related work, and finally (v) a conclusion and outline of future work.

2 Background

Model-Driven Engineering and Domain-Specific Languages. In this work we specifically focus on the construction and maintenance of Domain-Specific Modeling Languages (DSML), i.e. the employment of Model-Driven Engineering (MDE) in the context of Domain-Specific Languages (DSLs) [7], and in particular by constructing our approach on top of the Xtext language workbench that is built on the EMF [5, 25]. More specifically, EMF is the quasi-reference implementation of the Essential Meta-Object Facility (EMOF) standard [20] and provides a closed and strict metamodeling architecture, which defines the model on the uppermost layer to conform to itself as well as the correspondence of every model element with a model element of the layer above, respectively. EMOF, as well as the Extended Backus-Naur Form (EBNF) [27], represent

[5] The source code of Ecss is published at https://github.com/patrickneubauer/ECSS.

DSLs to define languages in form of metamodels and context-free grammars (CFGs), i.e. also referred to as "text-based concrete syntaxes" and "notations", respectively. In the EMF, an Ecore model—also referred to as EMF-based "metamodel" or "abstract syntax"—corresponds to the M2-layer in EMOF and acts as an abstract representation for concepts, properties, and relationships that are embodied by a real-world system. Further, the M1-layer in EMOF represents instances that specify actual values for concepts, properties, and relationships as defined in their corresponding M2-layer Ecore model.

Language Engineering and Workbenches. Language workbenches [6], such as Xtext, are tools that provide a range of features, such as dedicated editors, model transformations and validations, for DSML specifications. In general, Xtext employs the ANTLR parser generator [23] for the production of implementation artifacts, such as lexers and parsers, and offers two different DSML construction-mechanisms, i.e. typically selected as a result of an engineer's familiarity with the technical spaces[6] of grammarware and modelware [26]. On one hand, grammarware engineers, which are more familiar with traditional CFGs, may construct CFGs and employ the Xtext mechanism for deriving EMF-based metamodels. On the other hand, modelware engineers, which are most familiar with MDE-based technologies, may develop EMF-based metamodels and derive CFGs by employing metamodel-to-grammar transformations. Although Xtext supports both, the main focus is to provide grammars at the front-end and metamodels at the back-end to facilitate tool interoperability [22, 28].

3 Motivating Example

Within this section, we present a typical language engineering use case [16, 14] that involves the construction of a DSML by employing Xtext and the EMF and in particular a metamodel for capturing the concepts and relationships of a language for space transportation services. Moreover, this metamodel formulates the foundation upon which state-of-the-art practices, such as model-to-text transformations and grammar adaptation, as well as our approach may create, modify, and apply notations.

Language Structure. The metamodel of our exemplary language (cf. Figure 1) instantiates the core components of the Ecore metamodeling language, such as (abstract) classes, attributes, (containment) references, and enumerations. More specifically, the following concepts and relationships are defined: a *SpaceTransportationService* can own launch sites, spacecrafts, and engine types; a *Spacecraft* is defined by name, relaunch-cost, stages, manufacturer, country of origin, physical properties, functions, such as be an orbital launcher or intercontinental transport vehicle, and launch sites from where it can start; a *Stage* is defined by name, such as booster or spaceship, an engine type, and physical properties; a *PhysicalProperty* is defined by a type, such as length, volume or mass, unit, and value; and a *LaunchSite* is defined by name, location, operator, number of launchpads, operational status, and physical properties.

[6] *Technical space* refers to a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities often associated to a given user community with shared know-how, educational support, common literature, and scientific venues [17].

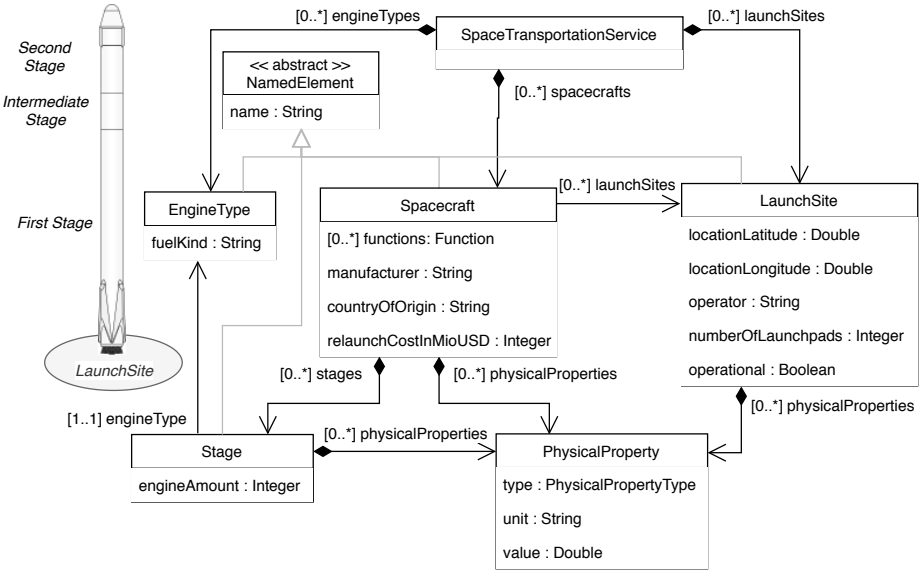

Fig. 1. Space Transportation Service language metamodel.

Notational Requirements. The requirements on the textual notation of the language to be constructed include (*i*) indentation-based layout, i.e. also referred to as offside-rules [18, 1] and represents the determination of code block-structure by means of indentation and layout based on the concept of layout-sensitive languages [18, 1], such as Python, Haskell, CoffeeScript, and YAML Ain't Markup Language (YAML) [3], and (*ii*) comma-separated arbitrary order of declaration, i.e. flexible or unordered sequence of instantiation. Listing 1.1 presents an excerpt of a space transportation service (cf. metamodel in Figure 1) that is built on both indentation-based layout and comma-separated arbitrary order of declaration. The former allows the use of hidden tokens, such as whitespace, as separators instead of visible tokens, such as curly brackets. The latter enables the use of sequences, e.g. location coordinates at the end of a launch site definition, that differentiate from those defined in the metamodel, e.g. location coordinates at the beginning of a launch site definition.

```
SpaceTransportationService:
  launchSites:
    name: KennedySpaceCenter,
    operator: NASA,
    operational: true,
    numberOfLaunchpads: 3,
    locationLatitude: 28.524058,
    locationLongitude: 80.65085
```

Listing 1.1. Instance of a space transportation service (excerpt).

4 Approach

Within this section our contribution is outlined alongside its application to the previously introduced example. First, the state-of-the-art model-first and grammar-first approach is showcased. Secondly, design principles, structural components, and selection modes of the style specification language are introduced. Thirdly, the style specification language is employed to model a DSML that fulfils the requirements presented in Section 3. Finally, the mechanism that generates executable DSML implementations from tuples that consist of style model and domain-specific metamodel is presented.

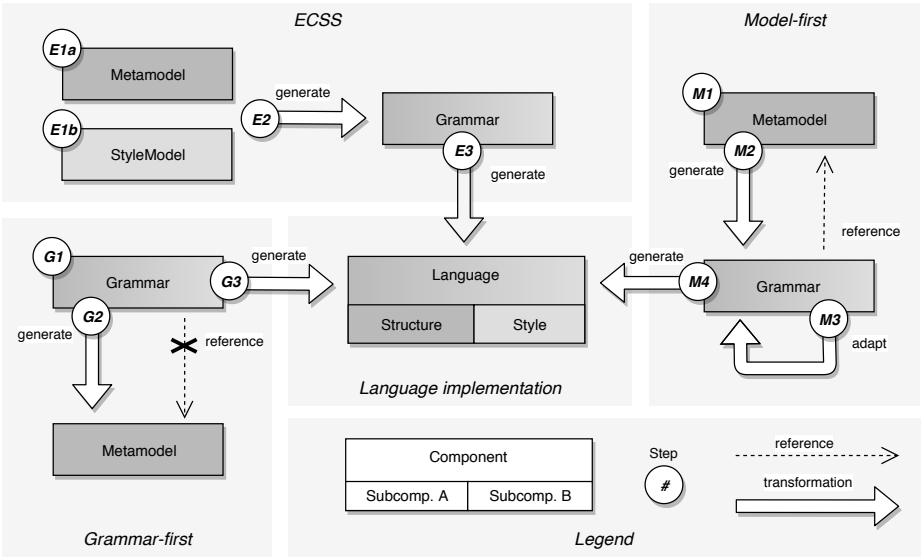

Fig. 2. Overview of DSML creation steps within ECSS, model-first, and grammar-first approach.

4.1 Model-first approach

The model-first approach [21] of constructing a DSML (cf. top-right gray area of Figure 2) is typically applied by developers most familiar with MDE and composed of the steps $M1$, i.e. construction of a domain-specific metamodel (cf. Figure 1), $M2$, i.e. application of a generic metamodel-to-grammar transformation (cf. Listing 1.2), $M3$, i.e. adaptation of generated grammar to fulfill the notational requirements stated above, and $M4$, i.e. generation of language implementation (cf. center gray area of Figure 2).

```
1 Stage returns Service::Stage
2  'Stage' name=ID '{'
3    'engineAmount' engineAmount=EInt
4    'engineType' engineType=[EngineType]
5    ('physicalProperties' '{'
6      physicalProperties+=PhysicalProperty
7      ( "," physicalProperties+=PhysicalProperty)*
8    '}' )?
9  '}';
```

Listing 1.2. Result of step $M2$—generated domain-specific grammar (excerpt).

To yield a DSML that supports indentation-based layout that prescribes that all non-whitespace tokens of a structure must be further to the right than the token that starts the structure, as well as flexible order of specification, the following adaptations on the generated grammar are performed. First, synthetic tokens, i.e. offering the specification of whitespace-semantics employing synthetic terminal rules, for the beginning and the end of a line as well as new lines are introduced (cf. lines 1-3 in Listing 1.3). Next, production rules are adapted to use the specified synthetic tokens (cf. lines 8-13 in Listing 1.3). Secondly, to support arbitrary order of declaration alongside whitespace-semantics, all possible occurring sequences need to be depicted by the grammar, which is accomplished by intermediating rule assignments with a vertical line, i.e. indicating a logical or, and enclosing them with brackets ending with a star-character, i.e. indicating zero or multiple occurrences. However, the combination of flexible sequences and whitespace-semantics requires to state all possible occurring sequences explicitly and thus causes the size of production rules to multiply by the number of structural features occurring in the containing classes.

```
1  terminal BEGIN: 'synthetic:BEGIN';
2  terminal NEWLINE: 'synthetic:NEWLINE';
3  terminal END: 'synthetic:END';
4
5  Stage returns Stage: SINGLESPACE
6   'name' ':' name=EString
7      (NEWLINE 'engineAmount' ':' engineAmount=EInt) &
8      (NEWLINE 'engineType' ':' engineType=[EngineType|EString])
9      (NEWLINE 'physicalProperties' ':' BEGIN
10       physicalProperties+=PhysicalProperty
11       (NEWLINE physicalProperties+=PhysicalProperty)*
12     END)?;
```

Listing 1.3. Result of step $M3$—adapted domain-specific grammar (excerpt).

4.2 Grammar-first approach

The grammar-first approach [21] (cf. bottom-left gray area of Figure 2), i.e. usually applied by developers most acquainted with grammar-based language engineering, of constructing a DSML is composed of the construction of a domain-specific grammar (cf. step $G1$), the application of a generic grammar-to-metamodel transformation (cf. step $G2$), i.e. also referred to as metamodel-derivation, and (iii) the generation of the language implementation (cf. center gray area of Figure 2). Although step $G2$ may be performed as a background process in Xtext, i.e. alongside step $G3$, and thus possibly to the unawareness of language developers, it a process that is required to yielding executable DSML implementations.

Our approach is primarily intended to be applied within the context of model-first language engineering (cf. Section 4.1) due to the facilitation of metamodels for capturing domain-specific structural semantics and constraints. However, it is also applicable for use cases in which metamodels are derived from domain-specific grammar and employed alongside style models to generate modernized DSML implementations.

4.3 Style Specification Language

Design principles. The aim of the Ecore Concrete Syntax Specification (Ecss) language to capture common styles of textual syntaxes (e.g. YAML-like, JSON-like, XML-like) and to support the automated generation of Xtext grammars for such syntaxes[7]. Ecss is inspired by CSS, which allows to style HTML code and offers straightforward composition, and thus aims for similar composability and parameterizability. For example, developers should be able to reuse, extend, and adapt existing language notations through composition with other language notations.

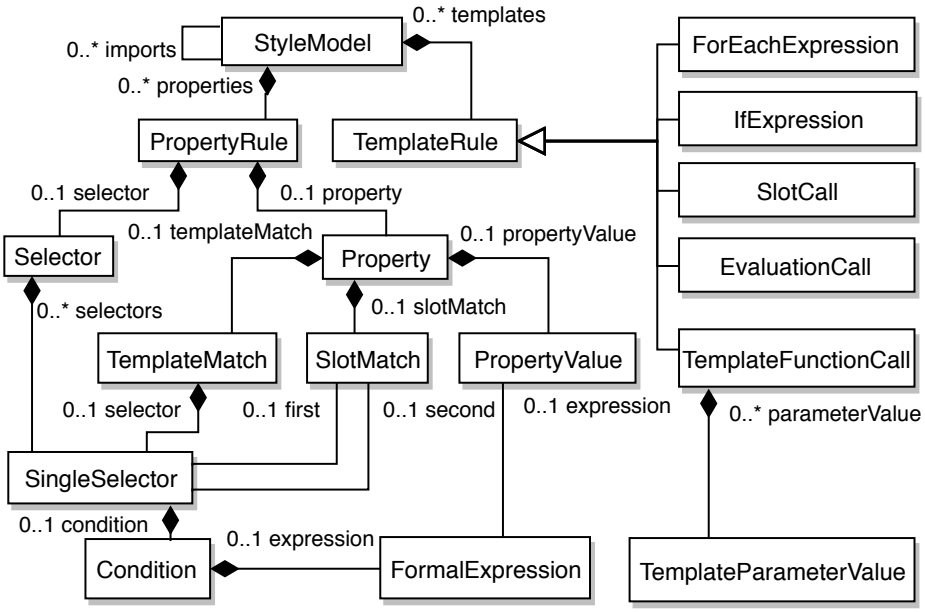

Fig. 3. Overview of Ecss language core components (simplified metamodel).

Structural Components. The core component of the Ecss language (cf. Figure 3) is a *StyleModel* that may extend other instances of style models through imports and contains a set of rules defining properties and templates. *PropertyRules* may be composed of a selector and property. *Selectors* may be composed of *SingleSelector* instances that may contain a *Condition*, i.e. defined by a *FormalExpression* such as an OCL expression. Each single selector selects a particular rule application instance. Selectors may define a *name* and thereby select rule applications with matching names. More specifically, the name for a rule that has a single feature as parameter refers to the name of the feature, i.e. the name of the class in case of a single class. Additionally, a selector may select subclasses by specifying the name of the rule that is named by its superclasses. Similarly to HTML, a sequence of single selectors choose rule applications based on their hierarchy. In detail, a rule application is chosen for a sequence of selectors if (i) it matches the last single selector itself and (ii) a direct or indirect parent matches the

[7] An initial catalogue of reusable Ecss styles is available online at http://bit.ly/ecss-styles.

remaining selectors. In addition to basic names, selectors may also specify that all rule applications are admissible and may restrict the applications using OCL expressions.

Instances of *Property* may be composed of a *TemplateMatch*, a *SlotMatch*, and a *PropertyValue*. A template match contains a *name* and associates a priority for using the template *name* for the rule application under consideration. A slot match contains a pair (*attributename*, *slotname*), specifying the priority that an attribute is matched to a slot. A property value simply associates a property of the rule application to a certain value, which may be a constant or an OCL expression evaluation result. The application of all properties may further be restricted by OCL expressions.

TemplateRules manifest as expressions, such as *ForEachExpression* and *IfExpression*, or calls, such as *EvaluationCall* and *TemplateFunctionCall*, and may contain static executed code parts and static outputs. A *ForEachExpression* triggers the generation of code for each object in a parameter of the rule application. A *TemplateFunctionCall* provides an opportunity to create a subsequent rule (subrule) application and may own instances of *TemplateParameterValue* that originate from rule application parameters, values calculated from property rules, and values of a slot, i.e. defined by a multiplicity determining the minimum and maximum amount of subsequent features (subfeatures) that are distributed in this slot. An *EvaluationCall* produces output based on a property value. A *TemplateFunctionCall* produces output either as part of the parent rule, i.e. acting as the container, or as a reference identifier, i.e. triggering content generation outside the parent rule.

Syntactic Components. The primary syntactic components used within the context of an Ecss model include pronounceable keywords such as *import* , *template*, *rule*, *for*, and *if*, as well as template-activating character sequences such as *[%=...%]* for value insertion through variable access or function calls, *[%...%]* for (local) evaluation call or value call, and *::ruleName()* and *ruleName()* for Java calls.

4.4 Style modeling

The requirements imposed on the language introduced in Section 3 are fulfilled by supplying the Ecss grammar creator with the Ecore metamodel and Ecss model created in step $E1a$ and $E1b$ of Figure 2, respectively. In more detail, the style model *ws-aware.ecss* and *arbitrary-order.ecss* is created to fulfill the first and second requirement, respectively. The rule *whitespaceClassRule* in Listing 1.4 defines that the features of a class are indented, i.e. with respect to the class itself. More specifically, line 6 produces the grammar rule header. Next, if the class being processed has no associated attribute, *classname* followed by an empty space and colon is produced as initial content of a class (cf. lines 7-8). Alternatively, if a class owns a set of attributes, a rule call to the rule group *nameDistRules* is performed (cf. lines 9-11), i.e. selecting a subrule. Next, additional indentation is created for feature definitions that are returned by rule call *attributeDistRules*. The property rule *NamedElement+* is metamodel-dependent and specifies that the *classname* of classes that extend the class *NamedElement* have to appear in uppercase characters (cf. lines 14-15). The global property rule, i.e. indicated by a star-selector, is metamodel-agnostic and specifies that the *classname* of (any matching) class is defined by its name (cf. line 17) or, in other words, the name of a class in its

metamodel. Note that due to the (higher) priority of *2.0* that is defined by property rule *NamedElement+*, the global property rule with (lower) priority of *1.0* is only matched by classes that do *not* extend the class *NamedElement*.

The rule *arbitraryAttributeDistr* in Listing 1.5 defines a style that fulfills the second notational requirement, i.e. comma-separated arbitrary order of declaration, that is not feasible by the use of (simple) unordered groups and causes a quadratic increase in the size of a grammar rule. In more detail, lines 7-9 define the initially occurring feature as arbitrary, i.e. any feature from the set of features of a class may occur first. Next, lines 10-11 encapsulate Java code that computes a list of remaining class features that is subsequently being iterated for the production of individual feature occurrences that are prefixed by comma separators (cf. lines 13-15).

```
1  import "default.ecss";
2
3  templateGen classGenTemplate extends classTemplate;
4
5  rule whitespaceClassRule :classGenTemplate :: classRules:
6      class.name " returns " class.name ":" "{" class.name "}"
7      [% if (slot_name.getValues().isEmpty()) {%]
8          " '" classname "' " " ':' "
9      [% } else {%]
10         nameDistRules(~name[0 .. 1])
11     [% }%]
12     ::BEGIN() " (" attributeDistRules(~other[ 0 .. 99]) ")" ::END()  ';' ;
13
14 NamedElement+
15   { classname: ocl "rule.class.name.toUpperCase()" priority(2.0); }
16
17 * { classname: ocl "rule.class.name" priority(1.0); slot(name,name): 2.0;}
```

Listing 1.4. Ecss model for indentation-based layout (excerpt of ws-aware.ecss).

```
1  import "wsaware.ecss";
2
3  template attributeTemplate: uk.ac.york.cs.ecss.newproc.AttributeXtendRule;
4  templateGen attributeGenTemplate extends attributeTemplate
5
6  rule arbitraryAttributeDistr: attributeGenTemplate :: attributeDistRules:
7      "("
8      for esf: features join ") | (" {
9          attributeRule(esf)
10         [% List<EStructuralFeature> smallerFeatures = new ArrayList(
               features); %]
11         [% smallerFeatures.remove(esf); boolean first = true; %]
12         "(("
13         for EStructuralFeature sub: smallerFeatures join ") & (" {
14             "','" attributeRule(sub)
15         } "))"
16     } ")"
17 ;
```

Listing 1.5. Ecss model for arbitrary order of declaration (excerpt of arbitrary-order.ecss).

4.5 Generation of Grammar and Language implementation

In this section, the process of generating grammars (cf. step $E2$ in Figure 2) that is followed by the final step (cf. $E3$), i.e. the generation of executable DSML implementations as illustrated in Figure 4, is described in detail. The Ecss DSML grammar creator generates grammars based on models by substituting template parameters in style models with actual values and suitable subsequent values.

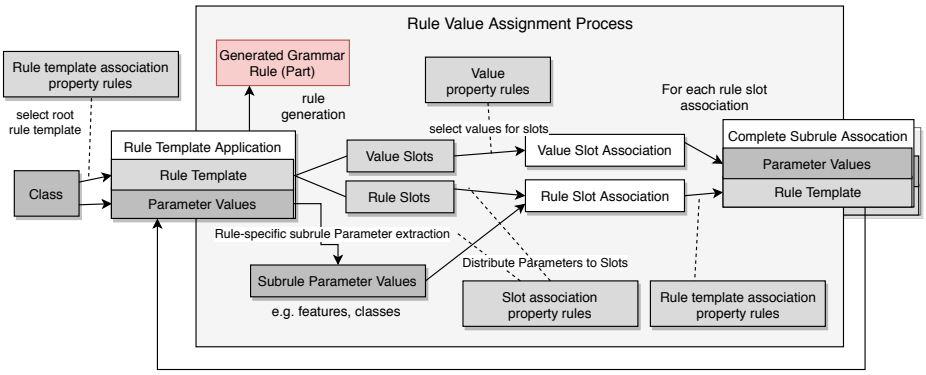

Fig. 4. Overview of DSML grammar generation workflow.

Ecss is built on template rules that act as code generating classes that produce code based on values assigned to class fields. Class fields can manifest as (i) directly assigned fields, i.e. field values determined by constructor parameters, (ii) styled fields, i.e. field values determined by property selectors, and (iii) slot fields, i.e. field values derived from directly assigned fields that are distributed in slots based on priorities computed from the set of available associations between slot and value. Moreover, directly assigned fields may refer to model elements in the input metamodel; the value of styled fields is the result of determining a rule template from the set of available priorities in associations between slot and rule template. The root rule application is selected using template property rules with their root class as a single parameter. In case no root class has been specified, it is automatically determined by selecting the class that is contained by the lowest number of other classes and contains the highest number of other classes. Next, the variable associations of the rule application are established depending on the rule application class and subsequent parameters may be derived from parameters of the rule. Then, property rules are used to assign property values to the set of properties in the rule. Next, the priority of slot parameters is calculated using slot value property rules and specific subsequent parameters that are distributed among rule slot priorities and slot multiplicities. Next, the output generating function of the rule application, i.e. defined by values computed in the previous step, is executed and dynamically calls subrules. In detail, most suitable subrule templates, i.e. based on property rules and parameter types, are selected and similarly processed.

Priority Computation. Assuming that the class *Stage* in our motivating example represents the root class and is converted by employing the rule *whitespaceClassRule* (cf.

Listing 1.4). As a result of the definition of *classGenTemplate*, the variable *class* is set to the class parameters and all class features are distributed to slots as follows. First, values for non-slot fields, i.e. *classname* in this case, are computed. Although *Stage* is a subclass of *NamedElement* and therefore both associations for *classname* are possible, the association defined by the *NamedElement* property rule is chosen due to its higher priority value. Secondly, features are assigned to slots based on defined priority values, i.e. 1.0 if undefined. In our example, the rule *slot(name,name)* defines a priority of 2.0 and thus causes the attribute *name* and remaining attributes to be associated with the slot *name* and the slot *other*, respectively. Next, the generation process in our example continues with a call to the rule group *nameDistRules*, i.e. with feature *name* as parameter, for the non-empty slot *name*. Finally, the rule *arbitraryDistrRule* is prioritised over the rule *defaultAttributeDistr*, i.e. imported from *default.ecss*, due to its priority value being higher by 0.5. Thus, establishing *attributeDistRules([engineAmount, engineType, physicalProperties])* as a new rule call that is similarly executed.

5 Related Work

In this section we present existing literature on the specification of visual textual representations and differentiate them with our approach.

In [12], concrete textual representations are defined using model annotations specified in terms of a dedicated DSL. This approach represents an effort towards reducing redundancy between the specification of metamodels and grammars, e.g., introduced by the duplicated definition of element-multiplicity, by employing a sequence of transformations on Textual Concrete Syntax (TCS) models and metamodels. Compared to our approach, the definition of a DSL is achieved utilizing TCS models that are similarly employed alongside metamodels for the generation of textual grammar. However, as a result of TCS models specifying associations to individual metamodel elements, this approach does not enable the definition of domain metamodel-agnostic styles and thus is limited by its application of styles to particular metamodels.

Further, [8] presents a classification of existing concrete textual syntax mapping approaches and identifies a set of issues associated with incremental parsing, model updating, and partial and federated views. This classification distinguishes between (i) manual development or auto-generation of metamodels from existing language grammars, (ii) manual development of grammars based on existing metamodels, and (iii) manual development of mappings between existing metamodels and grammars. Accordingly, our approach fits both classification (i) and (ii) and thus can neglect (iii).

TCSSL [19] represents an approach to establish bidirectional mappings between abstract syntax trees and concrete syntax trees by defining EBNF-like rules, which differ from EBNF rules by having sub-rules that are triggered based on the inheritance hierarchy depicted in abstract syntax trees. Compared to our approach, TCSSL also allows to define multiple different mappings based on the same metamodel to provide different concrete representations of the same abstract concepts, i.e. fitting the needs of different stakeholders, it does not allow to define concrete representations applicable to different metamodels. Moreover, TCSSL requires to language engineers to manually specify mapping rules for each metamodel element as well as concrete

representation thereof, instead of employing structure-agnostic style models on arbitrary domain metamodels. Further, multiple pass-analysis checks need to be manually implemented to address challenges, such as type checking and reference resolution mechanisms, that are raised during the compiler construction process.

The Textual Editing Framework (TEF) [24] presents an approach to embed generated EMF-based textual model editors into graphical editors created with GMF and tree-based editors generated with EMF. Compared to our approach, TEF also offers the capability to create style specifications for domain metamodels. However, TEF requires language engineers to manually implement complete style definitions for elements in the domain metamodel that are intended to be instantiated within the context of the embedded textual editor.

Moreover, an approach that offers library-based syntactic extensibility based on the Spoofax language workbench has been presented in [13] and offers support for the host languages Java, Haskell, and Prolog. In comparison, grammar-dependent transformations weave and unweave "syntactic sugar" into and out of host language notations, respectively, instead of decoupling style information from abstract domain-specific concepts. Consequently, requiring construction and maintenance of domain metamodel-dependent bi-directional transformations to enable grammar backward-compatibility instead of liberating language engineers from creation and maintenance of such complex transformations.

EMFText [9, 10] presents an approach for the definition of textual representations and the generation of editors from Ecore-based metamodels. Compared to our approach, concrete syntax rules in EMFText are defined based on concrete metaclasses or metaclass attributes instead of, additionally, enabling the definition of metamodel-agnostic notations, i.e. based on types of a metamodeling language such as Ecore.

A dedicated DSL for the construction of (complex) bridges between grammars and metamodels is presented in [11, 4]. In general, their work differs to our approach by promoting the facilitation of bridge-specifications between dedicated metamodels and grammars instead of metamodel-agnostic notations. Thus, effective construction of valid bridges between domain metamodels and grammars requires language engineers to be fully aware of actions and tool specifications available on both sides of a bridge.

6 Conclusion and Future Work

In this work, we proposed an approach for the definition and maintenance of textual notations distinctively and autonomously and implemented the approach through a template language and toolkit for textual style definitions based on EMF and Xtext. Moreover, we showcased the application of our implementation, i.e. involving the creation of domain metamodel-agnostic notational definitions, in a comparison with the model-first approach, i.e. involving the adaptation of (generated) grammar, and indicate usability based on conciseness and expressiveness. Future work includes (i) the application of style models to the set of open-source DSML projects available on open-source software providers, such as Github, to extend findings in regard to the expressiveness and conciseness, (ii) the iterative extension of our notational-specification

language in regards to the findings in (*i*), and (*iii*) the conduction of an empirical user study expanding our findings in regards to the usability of the proposed language.

References

1. Adams, M.D.: Principled parsing for indentation-sensitive languages: revisiting landin's offside rule. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 511–522 (2013)
2. Alanen, M., Porres, I.: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Tech. rep., Turku Centre for Computer Science (2003)
3. Ben-Kiki, O., Evans, C., Ingerson, B.: YAML Ain't Markup Language (YAML™) Version 1.1 (2009)
4. Brucker, A.D., Cabot, J., Daniel, G., Gogolla, M., Herrera, A.S., Hilken, F., Tuong, F., Willink, E.D., Wolff, B.: Recent developments in OCL and textual modelling. In: Proceedings of the 16th International Workshop on OCL and Textual Modelling co-located with 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016. pp. 157–165 (2016)
5. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA. pp. 307–309 (2010)
6. Fowler, M.: Language workbenches: The killer-app for domain specific languages (2005), http://www.martinfowler.com/articles/languageWorkbench.html
7. Fowler, M.: Domain-specific languages. Pearson Education (2010)
8. Goldschmidt, T., Becker, S., Uhl, A.: Classification of concrete textual syntax mapping approaches. In: Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings. pp. 169–184 (2008)
9. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings. pp. 114–129 (2009)
10. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Model-based language engineering with emftext. In: Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers. pp. 322–345 (2011)
11. Herrera, A.S., Willink, E.D., Paige, R.F.: A domain specific transformation language to bridge concrete and abstract syntax. In: Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings. pp. 3–18 (2016)
12. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. pp. 249–254 (2006)
13. Kats, L.C.L., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and ides. In: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA. pp. 444–463 (2010)

14. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages using Metamodels. Pearson Education (2008)
15. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. **14**(3), 331–380 (2005)
16. Kühne, T.: What is a model? In: Language Engineering for Model-Driven Software Development (2004)
17. Kurtev, I., Aksit, M., Bézivin, J.: Technical Spaces: An Initial Appraisal. In: Proc. of CoopIS (2002)
18. Landin, P.J.: The next 700 programming languages. Commun. ACM **9**(3), 157–166 (1966)
19. Muller, P.A., Fondement, F., Baudry, B.: Concrete Syntax Definition For Modeling Languages. Ph.D. thesis, École Polytechnique Fédérale De Lausanne (2007)
20. Object Management Group (OMG): Meta Object Facility (MOF), Version 2.5.1 (2016), http://www.omg.org/spec/MOF/2.5.1/
21. Paige, R.F., Kolovos, D.S., Polack, F.A.C.: Metamodelling for grammarware researchers. In: Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers. pp. 64–82 (2012)
22. Paige, R.F., Kolovos, D.S., Polack, F.A.C.: A tutorial on metamodelling for grammar researchers. Sci. Comput. Program. **96**, 396–416 (2014)
23. Parr, T.: The definitive ANTLR 4 reference. Pragmatic Bookshelf (2013)
24. Scheidgen, M.: Textual modelling embedded into graphical modelling. In: Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings. pp. 153–168 (2008)
25. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)
26. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers. pp. 159–168 (2005)
27. Wirth, N.: Extended Backus-Naur form (EBNF). ISO/IEC **14977**, 2996 (1996)
28. Zschaler, S., Kolovos, D.S., Drivalos, N., Paige, R.F., Rashid, A.: Domain-specific metamodelling languages for software language engineering. In: Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers. pp. 334–353 (2009)