Challenges and Opportunities of Modularizing Textual Domain-Specific Languages

Christoph Rieger, Martin Westerkamp and Herbert Kuchen
ERCIS, University of Münster, Münster, Germany

Keywords: Domain-Specific Language, Modularization, Xtext, Language Composition.

Abstract: Over time, domain-specific languages (DSL) tend to grow beyond the initial scope in order to provide new features. In addition, many fundamental language concepts are reimplemented over and over again. This raises questions regarding opportunities of DSL modularization for improving software quality and fostering language reuse – similar to challenges traditional programming languages face but further complicated by the surrounding editing infrastructure and model transformations. Mature frameworks for developing textual DSLs such as Xtext provide a wealth of features but have only recently considered support for language composition. We therefore perform a case study on a large-scale DSL for model-driven development of mobile applications called MD^2, and review the current state of DSL composition techniques. Subsequently, challenges and advantages of modularizing MD^2 are discussed and generalized recommendations are provided.

1 INTRODUCTION

Domain-specific languages (DSL) have emerged for various purposes (Mernik et al., 2005). Despite their unique capabilities, shared fundamental concepts are rarely reused but are re-implemented for every new language. In the context of Model-Driven Software Development (MDSD) which tries to automate the process of software creation, this causes a frequent re-implementation of similar functionality. In recent years, modularization of DSLs has become a topic of increasing interest in academia due to the expected improvements on software quality (Pescador et al., 2015; Cazzola and Vacchi, 2016). Tightly coupled to the concepts of language evolution, the composition of languages introduces a variety of opportunities regarding maintainability, reusability, and extensibility (Cazzola and Poletti, 2010). For example, changes to language features can be performed in isolation, requiring one to update and rebuild only parts of the language (Vacchi et al., 2014).

Whereas a variety of DSL development frameworks have evolved in the past, support for language composition varies in practice (Erdweg et al., 2015). On the one hand, development frameworks specifically tailored to language modularization often emerged from niche projects and lack features such as sophisticated Integrated Development Environment (IDE) support (Vacchi et al., 2014; Ekman and Hedin, 2007). On the other hand, language workbenches used in practice often provide a broad set of features for developers and modellers but consider DSL composition as a negligible feature.

For example, the mature Xtext framework is widely used for developing textual DSLs and provides seamless IDE integration. However, modularization capabilities such as grammar inheritance are still limited. One objective of this work is to analyse capabilities concerning language modularization in Xtext and the resulting implications for DSL developers.

The analysis is based on a prototypical implementation of a modularized DSL. With Model-Driven Mobile Development (MD^2), Heitkötter and Majchrzak (2013) presented a cross-platform development approach to create data-driven business apps for smartphones and tablets. MD^2 generates native apps for multiple platforms, providing a native look and feel, access to device sensors, and a high level of abstraction for modellers. Whereas the textual DSL facilitates modularization by utilizing the Model-View-Controller (MVC) pattern (Ernsting et al., 2016) for its models, the framework itself is not aligned with this structure but implemented in a monolithic project. In order to improve maintainability and extensibility, an enhanced framework architecture is proposed for MD^2 with respect to its DSL design and tooling.

The contributions of this work are threefold. Firstly, the paper reviews language modularization

Rieger, C., Westerkamp, M. and Kuchen, H.
Challenges and Opportunities of Modularizing Textual Domain-Specific Languages.
DOI: 10.5220/0006601903870395
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 387-395
ISBN: 978-989-758-283-7

approaches in the field of external domain-specific languages. Secondly, modularization capabilities and limitations of the Xtext framework are further investigated, using a case study of the large-scale MD^2 DSL. Thirdly, we generalize the observed benefits and drawbacks and propose recommendations for modularizing existing DSLs. The structure of this paper follows these contributions. Based on related work presented in Section 2 and general modularization concepts in Section 3, Section 4 provides the case study. The findings are then generalized and discussed in Section 5 before concluding in Section 6.

2 RELATED WORK

Introducing formal models as a higher level of abstraction permits domain experts to express their requirements using semantics close to the notation known within the domain, usually using either a textual or graphical syntax (Völter, 2013).

External DSLs are independently developed languages and separate to any host-language, therefore often providing a custom syntax specifically crafted according to domain experts' requirements (Fowler, 2005). Since they are independent, appropriate tools such as linkers, parsers, compilers, or interpreters need to be provided (Völter, 2013). *Internal DSLs* are encapsulated into a General Purpose Language (GPL) and consequently use the same syntax. However, they utilize only a subset of its features to create domain abstractions (Fowler, 2005). *Language Workbenches* offer a custom IDE, specifically designed to the development and usage of DSLs. The IDE becomes an integral part of model-processing, blurring the lines between programming and modelling (Fowler, 2005).

In contrast to GPLs, DSLs are designed in line with a (potentially changing) domain scope and software systems need to cope with changes in their environment (Cazzola and Poletti, 2010). Moreover, DSLs are often iteratively developed, e.g., when deficiencies in the language design are exposed or a DSL is intentionally designed to cover only the most important domain concepts in the beginning, with further components to be developed in the future (Völter, 2013). Additional complexities arise when downward compatibility to previous versions should be provided through techniques such as deprecation markers (Völter, 2013).

Consequently, the attempt to enhance maintainability during the ongoing evolution of a language is a major driver for DSL modularization. Internal DSLs use the extensibility of a host language, e.g., macros in Lisp or metaprogramming in C++ . They, however, lack DSL-specific tooling support as well as a customizable syntax (Fowler, 2005). Therefore, we focus on external DSLs as well as Language Workbenches in the following.

In recent years, the implementation of modular external DSLs has become a subject of increasing research interest (Ekman and Hedin, 2007; Krahn et al., 2010; Cazzola and Vacchi, 2016). Subsequently, a variety of development frameworks have been presented that support the creation of necessary tooling such as parser generation, implementing generators, and supplying IDE integration (Erdweg et al., 2015). With *Neverlang*, Cazzola and Poletti have introduced a language development framework that specifically focuses on creating reusable DSLs (Cazzola and Poletti, 2010). In Neverlang, modularization is organised along two dimensions. Language features are defined individually in modules, containing the syntax definition in Backus-Naur-Form (BNF) notation and an arbitrary amount of so-called roles describing the semantics (Vacchi et al., 2014). Whereas Neverlang provides sophisticated modularization techniques such as traits (Cazzola and Vacchi, 2016), it does not provide IDE integration for generated DSLs, hampering its adoption by domain experts.

A DSL development framework that supports the entire MDSD process including language definition, generator implementation, and IDE integration, is *MontiCore*. The framework generates Eclipse plugins for DSLs which support syntax highlighting, foldable code regions, and error messages. Flexible language composition is enabled through interfaces and multiple inheritance. MontiCore includes external fragments at runtime so that modellers are capable of utilizing arbitrary DSLs when modelling. While a variety of template-engines are available for MontiCore, the framework provides a native engine that facilitates agile development by introducing tags within the source code which are implementable in later iterations (Krahn et al., 2010). Although the framework is under active development, no supporting community has established yet, leaving its future maintenance uncertain.

Being part of the Eclipse Modeling Project, the *Xtext*[1] framework's evolution and support is backed by a comparably large community. It provides sophisticated tooling support such as IDE integration but is not specifically tailored towards language composition (cf. Section 4). For code generation, Xtext advocates Xtend[2], a Java-like programming language that additionally supplies developers with template expressions to ease the implementation of generators.

[1]Xtext – http://www.eclipse.org/Xtext/documentation/

[2]Xtend – https://www.eclipse.org/xtend/

3 DSL MODULARIZATION CONCEPTS

A large amount of design patterns for domain-specific languages have been presented in literature (Spinellis, 2001; Krahn et al., 2010). As this work focuses on modularizing external DSLs, six applicable modularization techniques are visualized in Figure 1.

Language extension allows new features to be added to an existing language (Spinellis, 2001). The novel DSL inherits from the base language, including its semantics and syntax. Being closely related to object-oriented forms of inheritance, language extension is generally not limited to single inheritance, but also allows obtaining features from multiple DSLs. However, due to the threat of possible conflicts, mainly single inheritance is used in practice (Ducasse et al., 2006).

Mixins represent a special form of language extension. Unlike multiple inheritance, mixins are not tied to a particular super class in the type hierarchy. Instead, a mixin provides a self-contained increment of functionality and specifies its dependencies (to classes or other mixins). Concepts defined in a mixin can therefore be used by multiple classes, thus enabling a more flexible class composition. During language compilation, the type hierarchy is linearised such that all language dependencies are resolved using single inheritance (Bracha and Cook, 1990).

Language specialization represents the counterpart to language extension. Rather than extending a DSL, unnecessary parts of a language are removed, creating a novel DSL. It thereby comprises a subset of the former language elements (Spinellis, 2001).

Pipeline Pattern. In contrast to language extension and specialization, the language modules in this pattern are on the same hierarchical level. Each language handles a set of language elements and passes the rest to the next one, so that a language's output is the input for the next language in the pipeline. Pipelining thus encourages the separation of tasks and discourages the use of too feature-rich languages (Mernik et al., 2005).

Trait. Only recently, the use of traits has been proposed for DSL composition (Cazzola and Vacchi, 2016). Similar to an interface with method implementations, a trait is a collection of methods and attributes. Nonetheless, a trait is stateless by definition and does not enforce an order of composition, enabling a more flexible reuse across classes compared to inheritance. In language composition, traits provide or extend language constructs, and conflicts are explicitly disambiguated by the target language. In contrast to mixins, a trait only provides reusable functionality without affecting the type hierarchy and respective semantic implications (Ducasse et al., 2006).

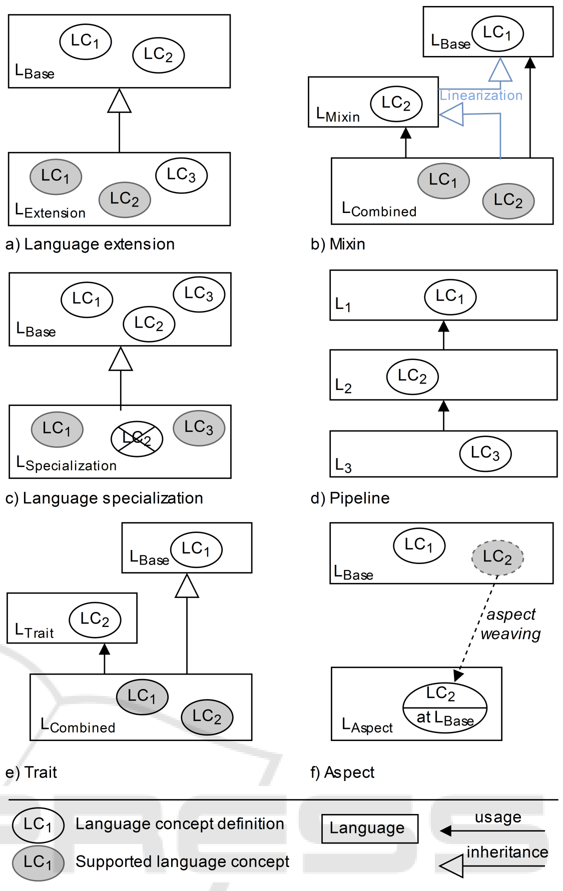

Figure 1: Language modularization concepts.

Aspect. Similar to traits, aspects provide composable units of functionality which are independent of their order and enable a flexible architecture design. However, in contrast to traits, aspects are not invoked within the target class, but an aspect itself declares *pointcuts*, i.e., the set of events during program execution for which the aspect's action should be executed (Wand et al., 2004). This composition mechanism requires a language processor, the so-called *aspect weaver*, which is used to resolve the composition of components and aspects. With respect to language composition, there are two general approaches to implement aspects. Firstly, aspects can be defined in correspondence with generated GPL code as long as an aspect weaver for the target language exists. Moreover, the developer needs detailed knowledge about the generator to define pointcuts. Secondly, defining aspects on a grammar-level provides the possibility to declare pointcuts according to a DSL's structure (Wu et al., 2005). Yet, such an approach requires a transformation engine which operates as aspect weaver.

4 MODULARIZATION IN Xtext

In this section, we present a case study on a large DSL called MD^2 in order to demonstrate the practical implications of current modularization capabilities of DSLs created using the language workbench Xtext.

4.1 Modularization Concepts in Xtext

Whereas a large variety of modularization concepts exists for DSLs in general (cf. Section 3), Xtext's support for language extension is limited. As depicted in Listing 1, grammars can extend other grammars using the `with` keyword (line 1). The inheriting grammar may extend existing language concepts, replace rule definitions, or introduce new ones. However, Xtext only supports single inheritance.

```
1  grammar de.md2.View with de.md2.Md2Basics
2  import "http://md2.de/Model" as model
3
4  OptionInput: 'Option' name=ID
5           widgetInfo
6           'options' values = [model::Enum]
7  fragment widgetInfo:
8           'label' labelText = STRING
9           'tooltip' tooltipText = STRING
```

Listing 1: Language inheritance and grammar mixins.

In addition, Xtext provides a feature called *grammar mixins*. In contrast to the modularization concept with the same name presented in the previous section, any metamodel can be used as Xtext mixin using the `import` keyword (line 2). When importing a metamodel, elements may then be *referenced* across models. However, Xtext does not actually employ the referenced grammar, but its metamodel. Therefore, it is not possible to directly access rules from these referenced grammars. Consequently, its syntax is not available within the importing DSL to *define* new elements of the imported class in the including language. For example, a modeller adding an `OptionInput` element can reference a list of values provided in a distinct model file (according to line 6) but cannot create an Enum object directly in the View model.

Finally, the concept of *fragments* is another instrument for enhanced reusability which has been added recently[3]. Instead of repetitively declaring a similar syntax in multiple parser rules, common parts can be extracted into a fragment and integrated by multiple rules. For example, the fragment `widgetInfo` (line 7) specifies the syntax of two attributes and is included in the `OptionInput` rule (line 5). Although this concept introduces multiple inheritance on the level of model elements, it is only available within a single grammar and cannot be used across languages.

[3]http://zarnekow.blogspot.de/2015/10/the-xtext-grammar-learned-new-tricks.html

4.2 Case Study on MD^2

Cross-platform development frameworks aim to mitigate the redundancy of developing applications for multiple platforms. However, hybrid apps based on web technologies lack a native look & feel and do not provide an additional level of abstraction beyond a common Application Programming Interface (API), e.g., regarding platform-dependent design guidelines (Heitkötter et al., 2013). The Model-Driven Mobile Development (MD^2) framework abstracts from the low-level implementation of business apps – i.e. form-based, data-driven apps interacting with back-end systems (Majchrzak et al., 2015) – and allows for modelling the desired result in a platform-agnostic fashion (Heitkötter et al., 2013). From this model, the framework currently generates native source code for the Android and iOS platform as well as a Java-based back-end application.

Models designed using the MD^2 DSL follow the Model-View-Controller (MVC) pattern, extended by an additional workflow layer (Ernsting et al., 2016). Workflows can trigger workflow elements defined in the controller, while conversely the controller can fire a workflow's events as illustrated in Figure 2.

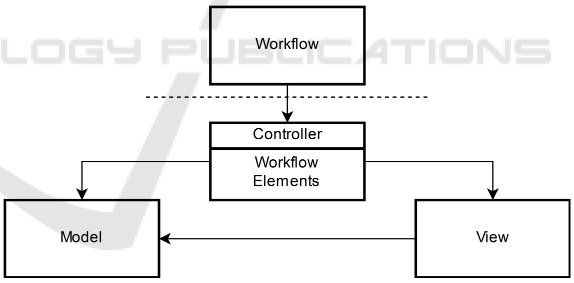

Figure 2: Architecture of MD^2 models.

In contrast to the subdivided design of MD^2 models, the DSL itself is defined in a single Xtext grammar, roughly grouped into Model, View, Controller, and Workflow components which reference each other. However, some cases exist in which this hierarchy is bypassed. For example, the *AutoGenerator* feature is used to automatically derive a generic view representation from a given data structure. This feedback between view layer and content provider (located in the controller component) causes a circular dependency which needs to be considered while restructuring the language.

The purpose of MD^2's modularization is therefore twofold. First and foremost, it should result in a framework that provides enhanced maintainability

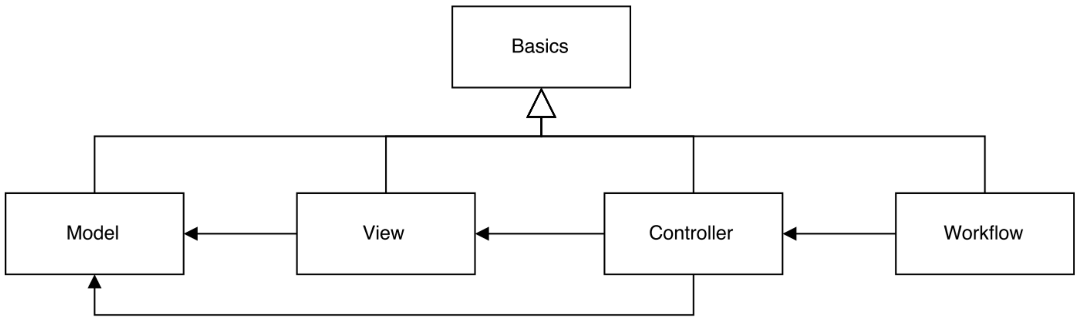

Figure 3: Proposed module structure for MD^2 models.

and increased software quality. Resulting from a series of modifications and extensions, the DSL was not well structured and interrelations were hard to grasp for developers due to the sheer size of the language. Before modularizing, the Xtext file comprised 924 lines of code organized in 188 grammar rule definitions. This makes MD^2 one of the five largest Xtext-based DSLs (in lines of code and file size) published on Github. However, the initial structure contradicted the MVC approach enforced in MD^2 models, and DSL evolution was seriously hampered by this complexity. For example, evaluations of the framework have unveiled a lack of abstraction in the DSL's View layer (Vaupel et al., 2014) which could be tackled more efficiently when dependencies to other components of MD^2 are clear.

Second, MD^2 is tailored to the domain of data-driven business apps (Heitkötter et al., 2013). Future users may wish to create DSLs tailored to their subdomains or even organisations. This can be achieved by extending or specializing reusable modules. In addition, the creation of multiple (technical) sub-DSLs can remedy current deficiencies and foster a more agile and targeted evolution of language features.

To achieve these aims, a clear separation of concerns is mandated on DSL level and further components dealing with code generation to trace concepts along the processing chain, indicate dependencies, and clarify interrelations. In order to focus this work on the modularization of the DSL itself, the existing generators were adapted to the new project structure without formally subdividing them into submodules.

4.3 Modularizing MD^2

Complying with the structure of the MD^2-DSL and corresponding reference architecture (Ernsting et al., 2016), the MVC+Workflow pattern is adopted to decompose the monolithic MD^2 grammar into four sub-DSLs. In addition, a fifth *Basics* DSL is introduced as common infrastructure.

4.3.1 Inheritance-based Modularization

Each module is a domain-specific language on its own but does not exist in isolation as visualized in Figure 3. The *Model* module only relies on *Basics* and provides the MD^2 type system, rules for modelling custom entity structures, and typed parameter definitions to be used by other modules. While the *View* module depends on Model in order to reference data types, the *Controller* relies on both Model and View such that the modeller can define custom actions linking data objects with the desired representation. The *Workflow* module only depends on Controller, since it builds workflow paths from low-level process steps.

Sophisticated modularization techniques such as multiple inheritance are not available in Xtext. However, it permits reusing grammars to a certain extent through single inheritance (see Section 4.1). In theory, unidirectional dependencies between sub-DSLs could be recompiled into a chain of DSLs only using single inheritance, similar to the concept of mixins (see Section 3). Nevertheless, automation would be required to build a (temporary) inheritance structure from the dependency graph each time any of the modules changes, and adapt the IDE and generator code accordingly. We therefore decided to avoid this approach and interrelate multiple DSLs differently.

4.3.2 Interface-based Modularization

In contrast to single inheritance, more than one DSL can be imported in Xtext. As explained in Section 4.1, the flexibility of importing multiple sub-DSLs comes at the cost of modelling components in multiple files according to the module in which the rules are located, and reference the respective objects.

To achieve a coherent structure across multiple DSLs, Voelter and Solomatov (2010) suggest the utilization of interfaces. The modularized MD^2 DSL creatively combines standard Xtext features to create such extensible interfaces as depicted in Listings

```
1  grammar de.md2.Basics
2
3  MD2Model:
4        package = PackageDefinition &
5        model = LanguageElement?;
6  LanguageElement:
7        {LanguageElement};
8  PackageDefinition:
9        'package' pkgName = QUALIFIED_NAME;
```

Listing 2: Basics grammar defining interfaces.

```
1  grammar de.md2.Model with de.md2.Basics
2  import "http://md2.de/Basics" as basics
3
4  MD2Model returns basics::MD2Model:
5        super
6        model = Model;
7  Model returns basics::LanguageElement:
8        {Model} modelElements += ModelElement+;
```

Listing 3: Model grammar implementing interfaces.

2 and 3. Firstly, the concept of *unassigned rule calls* (line 7 in Listing 2) forces the instantiation of rules. As no further attributes are specified, the rule is effectively transformed into an empty interface. Secondly, the `returns` keyword influences the meta model by explicitly merging the inferred class with the given type. A common use case for this feature is the definition of expressions such that the actual subtype is transparent to referencing elements.

For example, the `Model` rule (line 7 in Listing 3) creates valid *Basics::LanguageElement* objects, effectively implementing the imported interface. Thirdly, the `super` keyword (line 5 in Listing 3) provides all contents of the inherited rule to the implementing rule. Therefore, *Basic::MD2Model*'s attributes are available in the corresponding rule of the *Model* grammar and can be overwritten. Together, the concept of interfaces is emulated in Xtext not only within a single language but also across DSLs.

4.3.3 Modularizing Bidirectional Dependencies

Beyond the presented linear dependency structure, intertwined components can also be extracted into separate sub-DSLs. As mentioned before, the *AutoGenerator* feature is such an example that provides a view element but relies both on the Model and Controller module. During the language generation process, Xtext is not able to resolve bidirectional language references. Extracting the problematic feature is possible by introducing a separate module that implements a new *LanguageElement* subtype complying with the aforementioned interface principles. The new structure as visualized in Figure 4 inevitably creates circular dependencies between those DSLs. However, the metamodel of each DSL can be built independently

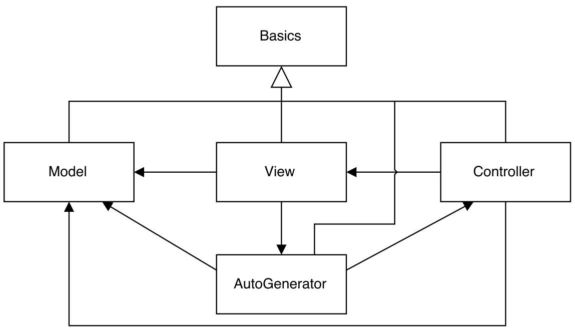

Figure 4: Resolving bidirectional dependencies.

by avoiding bidirectional dependencies. Using cross-language references via `imports`, other grammars can access the respective target normally and the resulting cycle is resolved soundly by Xtext.

4.3.4 Domain Extension

Domain extension is a second type of extension to the presented modular structure. The modularization of MD^2, as described to this point, mainly provides benefits from a DSL developer's perspective. The separated modules retain clear responsibilities and dependencies, easing future development of the framework. Nevertheless, it could be extended to specialized domains or applied to specific organizations which bring along new requirements by adapting MD^2 to the target domain, thus achieving a variable scope of the language (Völter and Solomatov, 2010). Instead of creating new modules that inherit from the Basics grammar, existing modules can be reused through inheritance. For example, a newly introduced *ViewAddon* module inherits from the View grammar to add a new type of view element as depicted in Figure 5.

The add-on's grammar needs to implement the common root of MD^2, i.e. *MD2Model*. Note that in this case, the `super` call (line 6 in Listing 4) does not refer to the rule in Basics but to the inherited definition within the View module. In order to extend a specific rule, e.g., adding a specific view element, a

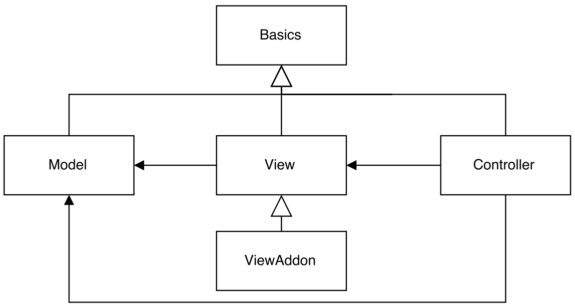

Figure 5: Domain extension.

```
1 grammar de.md2.ViewAddon with de.md2.View
2 import "http://md2.de/View" as view
3 import "http://md2.de/Basics" as basics
4
5 MD2Model returns basics::MD2Model:
6       super;
7 ContentElement returns view::ContentElement:
8       super | AddedButton;
9 AddedButton: ...
```

Listing 4: ViewAddon grammar.

new rule alternative can be provided for the definition of a *ContentElement* (lines 7-8). Nevertheless, the original implementation is maintained by delegating other model inputs to the super language implementation. Conversely, rules may limit which super language rules can be referenced, or place inherited elements arbitrarily in the derived language's structure without affecting the inherited grammar.

4.4 Advantages and Disadvantages of Modularization

In this section, the results of the modularization are discussed with regard to the implications for MD^2 modellers and DSL developers.

Modelling Experience. DSL modularization should minimize changes to the language's scope and its usage for modellers. It can be observed that the scope of the modularized MD^2 DSL has not changed and all language concepts could be transferred. Because the language syntax was not modified, three structural metrics were chosen from a plethora of grammar-related metrics in order to asses the overhead of the modularization (Črepinšek et al., 2010). Table 1 compares the different DSL sizes using the number of grammar rules, non-comment and non-blank lines of code (NCLOC), and the amount of (non-comment) characters in each DSL grammar. As can be seen, the modularization process incurs a slight overhead of about 5% lines of code due to the declaration of imports and the specification of interfaces. However, complexity is greatly reduced compared to the original DSL specification with 2087 lines (including comments). Admittedly, the resulting six DSLs represent only a first high-level separation of concerns. However, the effects of the modularization will amplify when complex constructs of the large *Controller* and *View* modules are further broken down.

However, as drawback of the extension approach, new model files (and file types) are required for each type of extension such as *AutoGenerator* model elements. This restriction is acceptable for large modules (such as enforcing MVC separation on file level) but gets inconvenient in case of multiple small additions.

Table 1: DSL comparison metrics.

(Sub-)DSL	Rules	NCLOC	Characters
Original MD^2	188	924	30.208
Basics	26	93	1.893
Model	20	97	3.498
View	62	301	8.319
Controller	85	421	16.198
Workflow	8	31	1.050
AutoGenerator	5	27	955
Modular MD^2	206	970	31.913
Difference	+9.6 %	+5.0 %	+5.6 %

Language Development. With regard to DSL development, several advantages can be observed. Firstly, splitting the large-scale DSL makes the framework more maintainable. Both MD^2 and Xtext evolve over time, therefore some parts of the implementation became outdated but could not be replaced because of the unknown implications of such actions. Also, the underlying functionalities for validation, preprocessing, and source code generation are untangled. Therefore, deprecated and irrelevant code can be better spotted and safely deleted without fearing side-effects on other parts of the framework.

Secondly, the separation of concerns will likely improve development speed and quality of sub-DSLs. Instead of managing the whole language scope, particular sub-DSLs can evolve separately. This does not only apply to language features but also includes tool support for validation, formatting, code completion, etc. Also, developing generators does not require comprehensive knowledge about MD^2 anymore but can focus on the transformation of specific domain concepts to the respective target platform.

Finally, the modularization prepares the language for future developments. New sub-DSLs can introduce new or extend existing language concepts. Also, language reuse in different DSLs is possible. However, the complete replacement of a language with an existing DSL in the same domain is currently not easily possible because of the interface-based language structure. Yet, considering the complex interrelations within generated code it is questionable whether a replacement is actually desired.

Modularizing MD^2 also introduces some drawbacks for language developers: The grammar of each sub-DSL is simplified at the cost of fragmentation in the overall framework structure. In particular, each new DSL in Xtext is based on five Eclipse projects for grammar definition, editor integration, and unit tests. The current module structure of six DSLs already results in a set of 30 projects and will quickly rise if new extensions and add-ons are introduced. Managing the configuration – including dependencies, DSL versions, and overall language bundling – needs to be automated using build tools such as Gradle[4].

From a conceptual perspective, a balance between core language features and extensions needs to be found. Language specializations may be tailored to the specific environment but such adjustments ideally do not require any change to the core language to avoid compromising on the reusability of the host language. Furthermore, the composition of domain extensions is based on grammar inheritance and therefore prone to the same problems of manually maintaining inheritance chains. As a consequence, introducing a multitude of minor DSL extensions is currently not advisable.

5 DISCUSSION

Beyond MD^2-related advantages and disadvantages derived from the case study, the generalized reflections on the results are presented as recommendations for the modularization of DSLs in a broader context.

Recommendation 1. *Inheritance-based modularization should be limited to closely related language extensions and language specialization.*

Language workbenches often do not support powerful language composition techniques. For example in Xtext, the limitation to *single-inheritance* and *language imports* as main approaches to modularization is not flexible enough to cope with the extensive (de-)composition of real-world DSLs. Because inheritance introduces tight coupling between two languages, this should be used sparingly. Common base languages and language specialization, for instance regarding sub-domains or company-/project-specific adaptations, are well-suited use cases for grammar inheritance. On the other hand, inheritance only allows for the addition or modification of grammar rules but cannot remove existing language concepts. Also, long inheritance chains of otherwise unrelated sub-languages provide maintainability benefits compared to one large-scale definition.

Recommendation 2. *Interface-based modularization should be used for a flexible combination of independent languages as large-scale layers of the DSL.*

A multi-layer structure of the resulting DSL can be achieved by emulating interfaces between different languages as described in Section 4.3.2. With the concept of language imports, references between elements from different DSLs cannot only be performed for hierarchical relationships but can also be used in settings with bidirectional or circular dependencies. Although it was shown how issues can be overcome using existing features, the workarounds are based on implicit conventions that need to be shared by all involved DSL developers.

Recommendation 3. *Language reuse works best if a common root language and infrastructure exists.*

Unfortunately, there is no simple way to “mix & match” arbitrary DSL specifications in Xtext. Reusing a distinct set of rules in different languages is complicated for reasons detailed in Section 4.1. In general, a common infrastructure is required to effectively integrate language concepts across multiple languages. A set of fundamental interfaces shared by all related sub-DSLs eases the integration process also in development frameworks which are not targeted to language modularization. However, the dependency on such a base language limits wide-spread language reuse as no standard set of primitives exists that can be applied generically to a broad set of languages.

Recommendation 4. *The granularity of modules should match the designed DSL’s structure.*

Due to the limited flexibility of referencing constructs by importing the target metamodel, the module structure ideally matches the structure of the resulting DSL. For example, the effect of requiring separate model files for language add-ons is potentially acceptable for larger chunks of functionality that form intrinsic sub-units of the designed DSL. Otherwise, numerous small language extensions require content to be modelled in many different files, potentially causing confusion for the modeller.

Obviously, these pieces of advice are based on the current features of the Xtext framework. If better concepts for modularization are introduced, these recommendations might change over time. Possible options include native interfaces that can be implemented by any grammar rule (Krahn et al., 2010) or techniques such as aspects and traits (general multi-inheritance has its own shortcomings) for embedding individual language concepts in different DSLs such that the resulting integration is transparent to the user. However, currently no development efforts beyond the *fragment* concept for non-extensible and DSL-internal interfaces are known.

6 CONCLUSION AND OUTLOOK

Until now, basic language constructs in DSLs need to be implemented from scratch again and again, thus

[4]Gradle Build Tool – https://gradle.org/

limiting the degree of true domain-specificity. Consequently, interoperability, maintainability, and reuse of DSLs do not reach their full potential. In this work, modularization techniques for language (de-) composition using a state-of-the-art framework for DSL development called Xtext were investigated. A case study on the large Xtext-based DSL MD^2 for modelling business apps is presented in order to achieve a high degree of modularity using available modularization techniques.

Several challenges limit the flexible applicability of language modules. Most importantly, the constraints of single-inheritance and Xtext's inability to embed external grammars negatively affect the possibilities of modular languages and the resulting modelling experience for users. Beyond the particular use case, general opportunities of DSL modularization in Xtext include the reduction of legacy code and improved maintainability of the current code base. The applied practices are distilled into four recommendations for exploiting the current features.

This work reveals future research need concerning suitable modularization techniques for textual DSLs which necessitate more flexibility regarding the reuse of small-scale languages and the extraction of – often technical and not domain-specific – concepts into sub-DSLs. Also, fully modularizing the model-driven process including editing component, model processor, and code generators constitutes future work.

REFERENCES

Bracha, G. and Cook, W. (1990). Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311.

Cazzola, W. and Poletti, D. (2010). DSL Evolution through Composition. In *7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 1–6.

Cazzola, W. and Vacchi, E. (2016). Language components for modular DSLs using traits. *Computer Languages, Systems & Structures*, 45:16–34.

Črepinšek, M., Kosar, T., Mernik, M., Cervelle, J., Forax, R., and Roussel, G. (2010). On automata and language based grammar metrics. *Computer Science and Information Systems*, 7(2):309–330.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388.

Ekman, T. and Hedin, G. (2007). The JastAdd system — modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26.

Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., and van der Woning, J. (2015). Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44, Part A:24–47.

Ernsting, J., Rieger, C., Wrede, F., and Majchrzak, T. A. (2016). Refining a Reference Architecture for Model-Driven Business Apps. In *Intl. Conference on Web Information Systems and Technologies*, pages 307–316.

Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages? http://martinfowler.com/articles/languageWorkbench.html.

Heitkötter, H., Majchrzak, T. A., and Kuchen, H. (2013). Cross-platform model-driven development of mobile applications with MD2. In *28th Annual ACM Symposium on Applied Computing*, pages 526–533.

Krahn, H., Rumpe, B., and Völkel, S. (2010). MontiCore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372.

Majchrzak, T. A., Ernsting, J., and Kuchen, H. (2015). Achieving business practicability of model-driven cross-platform apps. *OJIS*, 2(2):3–14.

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

Pescador, A., Garmendia, A., Guerra, E., Sanchez Cuadrado, J., and de Lara, J. (2015). Pattern-based development of Domain-Specific Modelling Languages. In *International Conference on Model Driven Engineering Languages and Systems*, pages 166–175.

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99.

Vacchi, E., Olivares, D. M., Shaqiri, A., and Cazzola, W. (2014). Neverlang 2: A Framework for Modular Language Implementation. In *13th International Conference on Modularity*, pages 29–32. ACM.

Vaupel, S., Taentzer, G., Harries, J. P., Guckert, M., Stroh, R., Gerlach, R., and Guckert, M. (2014). Model-Driven Development of Mobile Applications Allowing Role-Driven Variants. *MODELS*, 45(355):1–17.

Völter, M. (2013). *DSL Engineering*. CreateSpace Independent Publishing Platform.

Völter, M. and Solomatov, K. (2010). Language modularization and composition with projectional language workbenches illustrated with MPS. *3rd Intl. Conference on Software Language Engineering, LNCS*.

Wand, M., Kiczales, G., and Dutchyn, C. (2004). A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910.

Wu, H., Gray, J., Roychoudhury, S., and Mernik, M. (2005). Weaving a debugging aspect into domain-specific language grammars. In *Proceedings of the 2005 ACM Symposium on Applied computing*, pages 1370–1374.