[BEK+18] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann:
Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features.
In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18), pg. 75-82. ACM, Feb. 2018.
www.se-rwth.de/publications

Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features

Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany
<lastname>@se-rwth.de

ABSTRACT

"Software languages are software too", hence their creation, evolution, and maintenance is subject to the same challenges. Managing multiple stand-alone variants of similar DSLs raises the related maintenance and evolution efforts for the languages and their associated tooling (analyses, transformations, editors, *etc.*) to a higher power. Software variability management techniques can help to harness this complexity. Research in software language variability focuses on metamodels and consequently mainly supports managing the variability of abstract syntaxes, omitting concrete syntax variability management. We present an approach to manage controlled syntactic variability of extensible software language product lines through identification of dedicated syntax variation points and specification of variants from independently developed features. This fosters software language reuse and reduces creation, maintenance, and evolution efforts. The approach is realized with the MontiCore language workbench and evaluated through a case study on architecture description languages. It facilitates creating, maintaining, and evolving the concrete and abstract syntax of families of languages and, hence, reduces the effort of software language engineering.

CCS CONCEPTS

• **Software and its engineering** → *Source code generation*; *Domain specific languages*; *Software product lines*;

KEYWORDS

Language Variability, Language Product Lines, Software Language Engineering

ACM Reference Format:
Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann. 2018. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2018).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3168365.3168368

VAMOS 2018, February 7–9, 2018, Madrid, Spain

ACM ISBN 978-1-4503-5398-4/18/02…$15.00
https://doi.org/10.1145/3168365.3168368

1 INTRODUCTION

Model-driven development (MDD) leverages (domain-specific) modeling languages (DSMLs) to reduce the conceptual gap between problem domains and the solution domain of software engineering [8]. Hence, efficient engineering, customization, and reuse of DSMLs has become a prime concern in MDD characterized as software language engineering (SLE) [13]. But "software languages are software too" [7] and as such are subject to the same challenges regarding creation, evolution, and maintenance. Consequently, SLE has produced a multitude of solutions to create languages based on metamodels or grammars, interpreters or generators, well-formedness rules in metalanguages or programming languages. Metamodels encode the abstract syntax (*i.e.,* structure) of languages as classes and their associations without providing means to instantiate models. Grammars also describe the structure of a language, but can support integrated definition of concrete syntax as well [9]. From these, model processing infrastructure to translate textual models into abstract syntax instances can be derived automatically, which greatly facilitates the efficient usage of DSMLs.

Research and industry have contributed a wealth of different DSMLs for different application domains and scenarios. A study [18] on architecture description languages (ADLs), for instance, discovered over 120 different ADLs for various domains. This requires creating, evolving, and maintaining independent languages and tooling for each of these individually. Research in software product lines (SPLs) has produced means to capture and manage variability of similar software in product lines. From these, different products can be derived through selection of features. Leveraging SPLs to DSMLs can facilitate engineering and maintaining product lines of similar languages. Research to DSML SPLs focuses on metamodels [22, 29] or grammars [16]. Both requires to maintain variability of the semantic mapping separately. Moreover, approaches to DSML variability either are restricted to fixed 150% models, where every possible feature must be known a priori [29], or support arbitrary language feature extension, which allows to break the structure imposed by feature diagrams easily.

We present a concept of controlled language variability that facilitates a posteriori extensibility with additional features, considers concrete syntax, and enables (re-)using languages as features without explicitly foreseeing this usage at language design time. It is realized through a family of integrated MontiCore [9] DSMLs and a composition mechanism based on well-defined language extension points. The realization builds upon existing language composition techniques of MontiCore and enables *controlled* language composition to support, *e.g.,* validation on product line level. The individual languages are independent of each other, which enables these to be developed by different language engineers.

The contributions of this paper, hence, are:

- A concept of controlled, extensible language product lines (LPLs) leveraging composition of independent language features that can be composed post hoc without invasion.
- Its realization with the MontiCore language workbench.
- A case study in architecture description languages.

To this end, Section 2 introduces our running example, before Section 3 introduces preliminaries. Afterwards, Section 4 presents our problem space concept for managing DSML variability and Section 5 presents its solution space implementation. Section 6 describes the application of our approach to managing features of an ADL. Section 7 highlights related work and Section 8 discusses observations.

2 EXAMPLE

Consider developing software architectures for different domains. To prevent the efforts of creating, maintaining, and evolving multiple stand-alone ADL variants tailored to the specific domains, language engineering starts with a core ADL with specific extension points and independently developed language components that provide modeling elements required for architectures of the different domains. A feature model associates the different language features (LFs) with extension points of the core ADL and of other features. Based on a feature configuration, the language components are combined such that an integrated ADL is created that allows the different domain experts to use precisely the modeling elements required. This enables a separation of concerns where language engineers develop LFs independent of each other. The arrangement of these features to a feature diagram is performed by a LPL manager. A language product manager is a domain expert who selects all features of a LPL that are relevant to the domain to generate language-processing tooling. A modeler then uses such a tool to implement models that conform to this language.

An example of a compact LPL for ADLs used for cloud systems and embedded systems is depicted in Figure 1. Based on independent language components (depicted right) with explicit extension points provided by different language engineers, the LPL manager defines the feature model governing, which features are available and how these relate (depicted left). Besides a common base feature, the LPL described by the feature model includes features typical to ADLs for embedded systems (such as automated connection of ports based on their types or names or component behavior models) as well features related to scalable and secure cloud systems (such as replicating components and encrypted communication). Each feature contains a language component that may yield further extension points. The relation between two features defines how their language components can be integrated. This decoupling enables reusing the language components in different feature models. Based on a feature configuration defined by the product manager (middle left), a software tool establishes the connections between the selected features' language components. Based on the selection of features, their respective language components are integrated into the extension points of the language components of the respective feature's parent. For instance, the language component of feature `InputOutputAutomata` is integrated into the

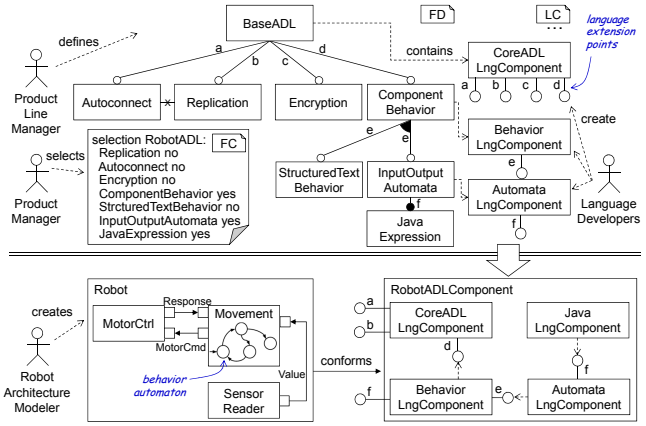

Figure 1: A LPL defined as feature model over language components. Given a feature configuration (top), the variant is transformed into a new language component (bottom).

extension point *e* of the language component contained in the feature `ComponentBehavior`. After integrating all referenced language components of the selected features, a new language component is generated, which can be used by the respective domain experts to model corresponding software architectures using the modeling elements selected through the feature configuration (in this case automata models describing component behavior).

Being able to reuse language components without modification enables to reuse the associated tooling (analyses, transformations) with the generated language component as well. Changes to a language component and its tooling are immediately available in the generated language modules as well. Both reduces the effort in creating, maintaining, and evolving modeling languages. The loose coupling between features and language components also enables to easily integrate new features into the feature diagram – integrating a new feature below `ComponentBehavior`, for instance, does not influence other features and language components. As the resulting language component can yield extension points again, the creation of intermediate products that require further refinement also is supported. Where multiple similar domains are addressed, creating refined domain-specific LPLs enables restricting a large base LPL accordingly.

3 PRELIMINARIES

While our concept for syntactical language variability can be applied to metamodel-based languages (*e.g.,* via abstract metamodel classes) as well, its realization is based upon the MontiCore language workbench [9]. MontiCore employs extended context-free grammars (CFGs) supporting integrated definition of concrete and abstract syntax [9] of DSMLs. From a DSML's CFG, MontiCore generates its abstract syntax tree (AST) classes and parsers that translate textual models into AST instances. To validate well-formedness constraints not expressible with CFGs, MontiCore features compositional *context conditions* (CoCos). Template-based code generators realize the DSMLs' semantics. MontiCore also supports compositional DSML integration via inheritance, embedding, and

aggregation [9]. Inheritance enables DSMLs to extend and override productions of their (possibly multiple) parent DSMLs. From inheriting DSMLs, MontiCore produces refined AST classes that inherit from the AST classes of the overridden productions. MontiCore also features *interface* productions, which enable underspecification in grammars that can be leveraged through inheritance to contribute new productions at well-defined extension points as depicted in Figure 2.

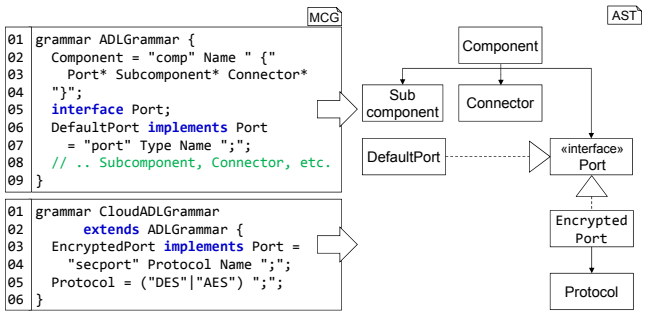

Figure 2: Example: Grammar Inheritance in MontiCore.

Here, the grammar `ADLGrammar` (top left) describes the quintessential elements of an ADL [19], *i.e.,* components that yield interfaces of typed ports and subcomponents that exchange messages through connectors between their ports (ll. 2-4). The production `Port` (l. 5) is an interface production that does not describe concrete or abstract syntax. Instead, it acts as extension point, which can be used in the defining language, such as the production `DefaultPort` (ll. 6-7), which consists of a data type and a name. From this, MontiCore generates five AST classes (depicted top right), out of which `Port` is an interface implemented by the AST class `DefaultPort`.

Interface productions can also be used in inheriting grammars, such as illustrated with grammar `CloudADLGrammar` depicted bottom. The grammar extends `ADLGrammar` (l. 2) and defines another implementation of `Port` that features security properties and omits port data types (ll. 3-4). Accordingly, MontiCore generates the two AST classes `EncryptedPort` and `Protocol`. Through this, `CloudADLGrammar` can reuse all modeling elements of the extended grammar and introduce new ones where foreseen by the developers of `ADLGrammar`.

To leverage interface productions as extension points for DSML features, we specify their use and relations through feature models [12]. We use a textual representation of feature trees, with the usual relations (mandatory, optional) between parent- and sub features and feature groups (alternative, exclusive). Besides this, the possible feature configurations can be restricted via cross-tree constraints (requires, excludes). For visualization purposes, we sometimes provide the graphical representation.

4 MODELING DSML VARIABILITY

Generally, languages are characterized as "the set of sentences" [13] that constitute the language, which also applies to modeling languages. With this definition being hardly accessible to investigation, a common refinement [4] is that DSMLs comprise (1) a concrete syntax (its sentences); (2) a minimal abstract syntax (structuring its sentences); (3) a semantic domain (typically a well-defined mathematical theory); and (4) a semantic mapping (giving meaning to the abstract syntax by mapping it to the semantic domain).

Software languages can be constructed from a variety of different constituents. Abstract syntaxes can be defined through metamodels (*cf.* EMF's Ecore [24], MPS [28]) or grammars (*cf.* Neverlang [25], Xtext [1]). Concrete syntaxes can be implemented through parsing textual models [14] or graphical editors [27]. Semantic mappings can be implemented through interpretation [5] or code generation [1]. Depending on the selected language constituents, various different forms of language composition are possible as well [6]. To manage variability and to explain composition, the definition above needs refinement. In the following, we define *language components* to comprise of (1) a grammar defining its abstract syntax (AS) and concrete syntax (CS) in an integrated fashion, (2) a (possibly empty) set of dedicated interface productions acting as abstract syntax extension points, and (3) a set of well-formedness rules. The dedicated set of abstract syntax extension points is foreseen by the language engineer to enable extending the language with new capabilities. The language itself might provide (default) implementations for its extension points. This, for instance, enables to define a Statechart language with extension points for guard transition expressions that already yields built-in expression but is open for future extension as well. The notion of AS extension points also applies to metamodel-based AS definitions, where these can be realized similarly.

As language components are unaware of being used with features, the composition of LFs and their composition enables reusing independently developed language components in different feature models. This facilitates extending the feature model post-hoc with new features, and prevents uncontrolled composition (*cf.* [25]). The composition operator we use for composing language components prevents invalidation of feature models through adding additional features. This property, called *conservative extension*, holds because it is impossible to remove syntactical language concepts by adding new features. The property holds only for the language's syntax and guarantees tooling stability, but cannot guarantee semantical correctness of analyses. The next section presents a realization of language components, LFs, and the composition operator based on MontiCore.

Our variability concept aims at enabling a well-defined integration of language components through definition of LPLs over features using these components. To this end, we describe language variability as trees of LFs supporting the usual relations [12] between feature diagram elements. Each feature contains a reference to its parent feature, a language component, and a set of mappings. Each of these mappings relates a production of its grammar to an interface production of the grammar contained by its parent feature (relative to a specific feature model). The concept also allows to refine an extension point with another extension point, *i.e.,* interface production. For the top-level feature, the set of mappings is empty. Through the relation to a parent feature, LFs are specific to the feature model they are used in, which also governs the restrictions between LFs (such as being optional, mandatory, exclusive, *etc.*). In particular, this enables reusing language components in different features and with different extension points. We reuse the *requires* relationship between two features to denote that a feature

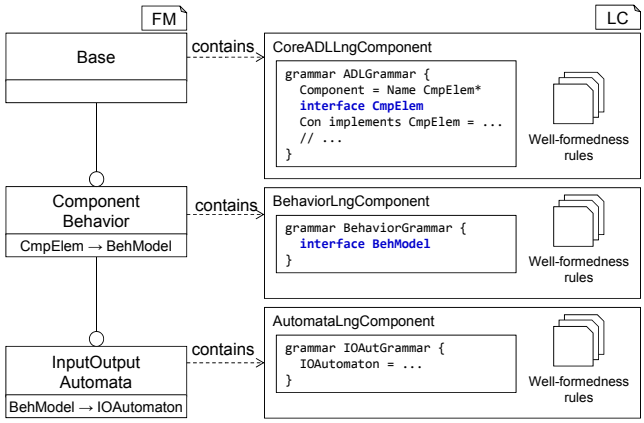

Figure 3: An illustration of LFs based on language components comprising grammars with dedicated extension points.

relies on the presence of another feature, which, *e.g.,* reflects in an inheritance relationship between the grammars of these features.

Individual LFs are not required to implement all extension points of their parent feature, which enables to refine LPLs by fixing the binding of some extension points and leaving other to be bound through features. Moreover, this enables using any suitable language as base language without modifying the language itself. Creating a LPL requires identifying and arranging LFs in a feature model based on their extension points. Language product managers then can derive language products from the LPL according to specific requirements (*e.g.,* to derive a robot ADL). How language components are related to LFs is depicted in Figure 3, which highlights the contents of some of the features depicted in Figure 1.

Here, the `Base` LF contains the `ADLGrammar` grammar, which describes quintessential elements of a component & connector ADL. This includes the extension point `CmpElem`, which is realized as an interface production of `ADLGrammar` and for which the `ADLGrammar` might provide its own (default) implementations. The feature `ComponentBehavior` contains the `BehaviorGrammar` comprising an interface `BehModel` that refines the extension point `CmpElem`. The grammar of the feature `InputOutputAutomata` describe the syntax for input output automata, and the feature maps these to the interface production `BehModel` of its parent feature. Each feature can contribute well-formedness rules operating on its abstract syntax: for instance, the `AutomataLngComponent` might yield a well-formedness ensuring that names of automaton states are unique. Based on the selected feature configuration, the features are composed to ultimately produce an integrated language component that enables the modeler to describe precisely what she needs to express without facing the accidental complexities of superfluous modeling elements. This composition is directed, as features lower in the feature models' hierarchy implement extension points of their parent features.

Language features are composed pairwise one after another, such that ultimately, a single, composed LF remains. The order in which sibling features are composed has no influence on the overall result. All context conditions are contained in the composed feature, and a joined grammar is generated. The composed LF retains all extension points.

To this effect, the composition is monotonically increasing in number of the extension points as features cannot remove extension points of their parents' grammars. The notion of a *mandatory* extension point only reflects in the feature model, but not in a single LF. Moreover, if grammar extension is allowed in the implementation, the approach only permits the grammars of a LPL to inherit from another if this is indicated as a requires relation between the features. As the composition relies on grammar extension, the newly added inheritance relations might interfere with existing ones and create a circular inheritance relation. To overcome this, we check the validity of a feature selection also with regard to potential circular inheritance relationships. The next section presents a realization of language components, language features, and composition based on the MontiCore languages.

5 INTEGRATING DSML SYNTAXES

With MontiCore [9], languages are defined in terms of extended context-free grammars (CFGs) that integrate concrete syntax with abstract syntax and use well-formedness rules implemented in Java, called context conditions (CoCos), to add restrictions not expressible through CFGs. Interface productions describe grammar extension points for which the grammar itself may provide (default) implementations (*cf.* Figure 2). Our solution space variability mechanisms, hence, are based on composing language components provided as MontiCore CFGs and context conditions.

Language components are developed independent of how they are used within a LPL. Nonetheless, language engineers have to foresee potential extension points in terms of interface productions. In our realization of the concept, language components are defined by a MontiCore CFG (MCG) and a list of context conditions. The approach generally permits the MCG of a language component to extend another MCG, unless this grammar is also part of the LPL. MontiCore supports dedicated interface productions that do not have a right-hand side as illustrated in l. 5 of Figure 2. Interfaces can be used in other productions (*cf.* `Component` in ll. 2ff). In another production, which is not necessarily in the same grammar model, interfaces can be implemented by other productions (*cf.* ll. 6f). The resulting structure of the AS is depicted on the right of Figure 2.

A *language component* encapsulates all artifacts of the definition of a single language into one dedicated artifact by holding explicit links to the grammar and a set of links to CoCo classes. Figure 4 depicts an exemplary language component `ADLLC` that references the ADL grammar (see Figure 2) and two CoCos. The *exports* keyword starts a list of explicit extension points via the names of the respective MontiCore interface productions. As the implementation exploits MontiCore interface productions as extension points, it might be that several interface productions are exposed as extension points undesiredly. To overcome this, we require language engineers to explicate all language extension points in the language component.

We separate LFs from language components to foster decoupled development of a language component and its context in a LPL. A LF defines (1) the language component it is based on, (2) the parent feature in the feature tree, and (3) a binding of grammar

```
01 language ADLLC {                                   LC
02   grammar com.ma.ADLGrammar;
03   cocos {
04     com.ma.cocos.CompNameLowerCase,
05     com.ma.cocos.PortNamesUnique
06   }
07   exports { CmpElem }
08 }
```

Figure 4: The language feature ADLLC.

```
01 feature BehaviorLF {                               LF
02  parent ADLLF;
03  language BehaviorLngComponent;
04  bindings { CmpElem -> BehModel; }
05 }
```

```
01 feature ADLLF {                                    LF
02  // no parent
03  language ADLLC;
04  // no bindings
05 }
```

Figure 5: The LFs BehaviorLF and ADLLF.

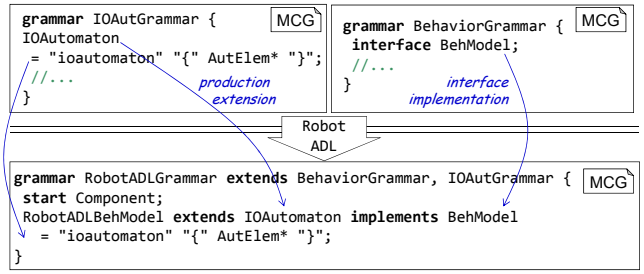

Figure 6: The composition (bottom) of two grammars (top).

productions to extension points of the parent feature. Language features are the building blocks of the feature model of a LPL. Each LPL requires a root feature, which has an empty parent feature and must not have bindings. For all other features, we require the parent to be present. If any bindings are present, the composition operator realizes language embedding (*cf.* Section 2). Otherwise, if no bindings are present, it realizes language aggregation.

Figure 5 depicts two LF models. `BehaviorLF` (left) references `ADLLF` as parent feature (l. 2) and `BehaviorLngComponent` as the implementing language component (l. 3). The binding relates the extension point `CmpElem` of the parent feature with the `BehModel` extension point of the grammar of the `BehaviorLngComponent` (l. 4). `ADLLF` (right) neither yields a parent feature, nor bindings.

The features are arranged in a feature model. We reuse a language and tooling for textual feature diagrams and selected variants (*i.e.,* feature configurations) implemented with MontiCore. With the LPL defined by the LPL manager at hand, a product manager can define a language variant by selecting a set of features. If the selection is valid with regard to the feature model and with regard to the approach's constraints, a new, composed language component is generated. From this, MontiCore generates language processing infrastructure such as a parser and an AST data structure on a push-button basis and iteratively composes two features from the leaves of the feature tree to the root as explained in Section 4. Figure 6 (top) depicts the grammars of the two LFs `BehaviorLF` and `InputOutputAutomata`. The bottom of Figure 6 depicts the grammar that results from the composition of these features, with

```
01 language RobotADLLC {                              LC
02   grammar com.ma.RobotADLGrammar;
03   cocos {
04     com.behavior.SingleBehaviorModel,
05     com.ioaut.SingleInitialState,
06     //...
07   }
08   exports { CmpElem, BehElem}
09 }
```

Figure 7: Language component of the composed LFs.

the binding applied. The name of the generated grammar is derived, with the feature configuration name as prefix before the name of the grammar of the root feature. The resulting grammar uses MontiCore's inheritance mechanism to extend both grammars. For technical reasons, the generated grammar has to reference the start production of the grammar of the root feature, to be used as top-level element for the generated parser. Further, the generated grammar comprises a new, generated production for every binding that has been applied. The left-hand side of the generated production is a derived non-terminal name, that states that the production extends the production of the extension and that it implements the interface production of the extension point. The effect of implementing an interface production has been explained in Section 3, the effect of extending another production is that the production can be applied wherever the extended production can be applied. Additionally, the generated abstract syntax class extends the generated abstract syntax class of the extended production. This has the advantage that all algorithms and tooling that are applicable to the extended AS element can be applied to the new one. The right-hand side of the generated production usually equals the right-hand side of the extended production. An exception is that if the right-hand side of a generated production contains a non-terminal symbol that has been bound as part of mapping. This is replaced with the left-hand side of the production generated from this mapping.

If an extension point is refined, both the extension and the extension point are realized as interface productions. In this case, a new interface (with a derived name) is generated that extends both interfaces. MontiCore generates parsers such that they are capable of parsing, despite this ambiguity in the concrete syntax.

Figure 7 depicts the language component resulting from the compositions of the two language features `InputOutputAutomata` and `BehaviorLF`. The referenced grammar is the grammar depicted at the bottom of Figure 6, which combines the abstract syntaxes and concrete syntaxes of the composed grammars. The context conditions of both features are joined. As the context conditions are checked against certain AS classes (and therefore, also their subclasses), all context conditions can be applied to the generated classes. It also comprises all exported extension points of the individual language components. Generally, the composition of two features f and g, where g is the parent of f, results in a composed LF with the parent feature of g, the composed language component, and the bindings of f.

6 EXTENDED EXAMPLE

This section demonstrates the application in context of an ADL to adapt it towards different domains. Developing and maintaining

BaseADL

MCG
```
01 grammar ADLGrammar {
02  Component = "component" Name "{" CmpElem* "}";
03  interface CmpElem;
04  interface Port extends CmpElem;
05  DefaultPort implements Port = "port" Type Name ";";
06   /* Subcomponent and Connector productions omitted */
07 }
```

LC
```
01 language CoreADLLngComponent {
02  grammar ADLGrammar;
03  cocos {
04   com.adl.cocos.CompNameLowerCase,
05   com.adl.cocos.PortNamesUnique
06  }
07  exports { CmpElem }
08 }
```
for embedding further top-level ADL elements

LF
```
01 feature BaseADL { language CoreADLLngComponent; }
```

Figure 8: Parts of the `BaseADL` language feature.

ComponentBehavior

MCG
```
01 component grammar ComponentBehaviorGrammar {
02  interface BehModel;
03 }
```

LC
```
01 language BehaviorLngComponent {
02  grammar BehaviorGrammar;
03  cocos {
04   com.compbeh.cocos.BehaviorUnique,
05   com.compbeh.cocos.BehaviorNotInComposedComponents
06  }
07  exports { BehModel }
08 }
```

LF
```
01 feature ComponentBehavior {
02  parent BaseADL;
03  language BehaviorLngComponent;
04  bindings { CmpElem -> BehModel }
05 }
```

Figure 9: The `ComponentBehavior` language feature.

InputOutputAutomata

MCG
```
01 component grammar IOAutGrammar {
02  IOAutomaton = "ioautomaton" "{" AutElem* "}";
03  interface AutElem;
04  State implements AutElem =
05                      (["initial"])? "state" Name ";";
06  Transition implements AutElem = "transition" src:Name
07           "[" Guard "]" "{" PortAss* "}" trg:Name ";";
08  interface Guard;
09  interface PortAss;
```

LC
```
01 language AutLngComponent {
02  grammar IOAutGrammar;
03  cocos {
04   com.ioaut.cocos.StateNamesUpperCase,
05   com.ioaut.cocos.UniqueInitialStates
06  }
07  exports { Guard, PortAss }
08 }
```

LF
```
01 feature InputOutputAutomata {
02  parent ComponentBehavior;
03  language AutLngComponent;
04  bindings { BehModel -> IOAutomaton }
05 }
```

Figure 10: The `InputOutputAutomata` language feature.

domain-specific variants of an ADL is challenging [2]. To this effect, we applied the approach presented in this paper to derive domain-specific ADL variants from a domain-agnostic BaseADL. Figure 1 in Section 2 depicts the language feature model representing all possible ADL variants. The root feature `BaseADL` contains the basic elements such as components, ports, and connectors that are common to each variant. The `Autoconnect` feature adds syntax and transformations to realize an automatic connection of ports with either identical names or types. The feature `Encryption` enables to describe secure ports (`SecurePort`) and encrypted connections (`EncryptedConnector`) between them. The `Replication` feature provides elements for modeling systems where components are capable of replicating themselves when needed. This is useful in client-server architectures, for instance, where a client component is replicated on each request. The LPL manager considers component replication to be a threat for autoconnecting ports. Choosing one of the two corresponding features thus excludes the other. The `ComponentBehavior` feature introduces *behavior-blocks* to the ADL. Component behavior models are intended to be modeled in such blocks, only. The subfeatures `StructuredTextBehavior` and `InputOutputAutomata` contain different behavior languages. As it should not be possible to model empty behavior blocks, choosing the `ComponentBehavior` feature requires to choose at least one feature that defines a component behavior language. Automata use expressions on their transitions as guard conditions. For this purpose, the LPL currently only includes `JavaExpressions`, which are therefore marked mandatory.

Consider a product manager who aims at developing static software architectures where atomic components' behavior can be specified via input/output automata. She thus selects the configuration containing the following features `BaseADL`, `ComponentBehavior`, `InputOutputAutomata`, and `JavaExpression`. Parts of the configuration's constituent are depicted in Figures 8-11. The `BaseADL` LF (cf. Figure 8) neither has a parent feature nor defines a binding as it is the LPL's root feature. The feature's grammar defines language elements common to all ADL variants such as components, connector, and ports. The language component further defines two context conditions and exports the interface `CmpElem`. With this, it is possible to extend component definitions with further top level elements through LF composition. The `Subcomponent` and `Connector` productions of the `BaseADL` grammar are omitted. The grammar of LF `ComponentBehavior` (cf. Figure 9) defines a single interface `BehModel` where behavior models for atomic components are intended to be embedded. The feature's language component further comprises two context conditions. The first ensures that each component contains at most one behavior model. The second requires that composed components must not contain behavior models. The LF binds the `BehModel` interface to the `CmpElem` interface of its parent feature's grammar. As a result, it is possible to specify component behavior models as top-level elements in component definitions. However, the syntax of possible component behavior models is still underspecified. For this reason, the `ComponentBehavior` LF is connected to two further features via an or-node (cf. Figure 1). Thus, each valid configuration containing the `ComponentBehavior` feature also contains at least one of the two subfeatures. The product manager chose the `InputOutputAutomata` feature out of these two features. The feature's grammar (cf. Figure 10) enables to model input/output automata for specifying component behavior. Transitions of such automata consist of guards (l. 7) and port assignments (l. 7). The productions' implementation remain underspecified (ll. 8-9) and are exported by the feature's language component. Thus, the exported interfaces must be bound by the LF's sub-features. The language component further consists of two

JavaExpression

MCG

```
01 grammar JavaInADLExprGrammar extends JavaDSL {
02  GuardExpr = Expression;
03  PortAssExpr = Expression;
04 }
```

imported from JavaDSL

LC

```
01 language JavaExprInADLExprLC {
02  grammar JavaGuardExprGrammar;
03  cocos {
04   com.javaexprguard.cocos.PortAssSimpleNameOnLHS,
05   com.javaexprguard.cocos.PortAssCorrectlyTyped,
06   com.javaexprguard.cocos.GuardExprBoolean,
07   com.javaexprguard.cocos.ReferencedPortsExist
08  }
09 }
```

LF

```
01 feature JavaExpression {
02  parent InputOutputAutomata;
03  language JavaExprInADLExprLC;
04  bindings {Guard -> GuardExpr, PortAss -> PortAssExpr}
05 }
```

Figure 11: The `JavaExpression` language feature.

context conditions ensuring each input/output automaton contains exactly one initial state and that state names start with capital letters. The grammar's `IOAutomaton` production is embedded into the `BehModel` production of the LF's parent feature. Further, the `JavaExpression` feature (cf. Figure 11) has to be embedded, as it is marked as mandatory subfeature of `InputOutputAutomata`. Its grammar inherits the productions from a Java grammar (l. 1) and defines two new Productions (ll. 2-3). Using the new productions `GuardExpr` and `PortAssExpr` enables to specify Java expressions (The `Expression` production is part of the inherited Java grammar). The two productions are bound to the `Guard` and `PortAssExpr` interfaces exported by the `InputOutputAutomata` LF. The LF's grammar introduces two new productions and does not simply directly bind the Java `Expression` production to the `Guard` and `PortAssignment` productions to enable separate handling of guards and port assignments via their types. The first two context conditions of the feature's language component, for instance, only restrict the well-formedness of expressions used in port assignments, whereas the third context condition only restricts guard expressions, and the fourth context condition affects guards as well as port assignments. Composing the four features as described in Section 5 leads to the LF depicted in Figure 12 that models the composed language. The grammar is composed of the grammars of the selects LFs by iteratively applying the transformation described in Section 5. The new LF's CoCos are all CoCos of all selected LFs. The new LF exports each interface exported by any selected LF. A valid model of the new language is depicted in Figure 13. The component and port declarations (ll. 1-3) originate from the `ADLGrammar` (cf. Figure 8). The `InputOutputAutomata` LF's grammar (cf. Figure 10) provides the possibility to declare automata, states, and transitions (ll. 5-7) through extending the interface added by the `ComponentBehavior` LF (cf. Figure 9). The expressions `true` and `in = out` used in the transition's guard and port assignment originate from the `JavaExpression` LF (cf. Figure 11).

7 DISCUSSION AND RELATED WORK

Our notion of DSML features is based on unrestricted interfaces, *i.e.,* the interface productions do no prescribe parts of the required abstract syntax or concrete syntax. While this prevents lifting functionality considering these features to the LPL, it allows for great

CompoundADL

MCG

```
01 grammar CompoundADLGrammar extends ADLGrammar,
02         BehaviorGrammar, IOAutGrammar,
03         JavaInADLExprGrammar {
04  start Component;
05  interface CompoundBehModel extends BehModel, CmpElem;
06  CompoundIOAutomaton extends IOAutomaton
07                      implements CompoundBehModel =
08                      "ioautomaton" "{" AutElem* "}";
09  CompoundGuardExpr extends GuardExpr
10                    implements Guard = Expression;
11  CompoundPortAssExpr extends PortAssExpr
12                      implements PortAss = Expression;
13 }
```

LC

```
01 language CompoundLC {
02  grammar CompoundGrammar;
03  cocos {
04   com.adl.cocos.CompNameLowerCase,
05   com.adl.cocos.PortNamesUnique,
06   com.compbeh.cocos.BehaviorUnique,
07   com.compbeh.cocos.BehaviorNotInComposedComponents,
08   /* CoCos of IOAutLC and JavaExprInADLExprLC omitted */
09  }
10  exports { CmpElem, BehModel, Guard, PortAssignment }
11 }
```

LF

```
01 feature CompoundLF { language CompoundLC; }
```

Figure 12: Result of composing the configuration's features.

CompoundADL

```
01 component MyComponent {
02   port Integer in;
03   port Integer out;
04   ioautomaton {
05     initial state s1; state s2;
06     transition s1 [true] {in = out} s1;
07   }
08 }
```

Figure 13: A valid model of the LF depicted in Figure 12.

extension flexibility. The inheritance relation between the feature grammars and the base grammar is established after feature selection. This can liberate feature developers from comprehending the base grammar at all, leading to fully independent DSML features. However, our concept also supports feature grammars aware of the base grammar to enable more specific features. Further, using the *requires* mechanism of feature diagrams, more dependencies between features can be described. Where most existing approaches use either bottom-up or top-down development of LPLs [15], we allow both directions as the feature model and the domain model are only loosely coupled. This supports agile extension of the LPL. The approach does not cover pure presentational variability [3] in the concrete syntax that does not affect the abstract syntax. As stated in [11], a usable language extension framework should have independent language extensions, which shall be automatically composable, and must not yield a corrupted composed compiler. Our approach satisfies these assumptions, because language components are (usually) independent of each other and can be composed using the composition mechanism described above. Our form of language (syntax) composition realizes the concept of conservative extensions known from formal languages. To this effect, all models that are conform to a language defined by a set of selected features, are still valid models of a language that is based on these features and arbitrary other additional selected features. Through a systematic literature review [20] comparing different approaches for LPLs, the authors identified 14 approaches realizing LPLs, where many

different concepts are involved. Some of the approaches support variability in abstract syntax only [10, 23, 29]. Most approaches use variability in metamodels, only few support variability in abstract syntax and concrete syntax on grammars, such as Neverlang [25], LISA [21], and FeatureHouse [17]. Our concept of LFs relates to the language components of Neverlang [25, 26], which contain syntax definitions in form of grammars and corresponding evaluation phases realizing its semantics. It differs in the way extension points are defined, which are realized in Neverlang using placeholders in the grammar, which are resolved via matching names. AiDE [16], built on top of Neverlang, also supports variability management of language components.

8 CONCLUSION

We have presented a concept for syntactic DSML variability that facilitates engineering, maintaining, and evolving product lines of related languages. The concept relies on modularly composable grammars encapsulated in DSML feature models related through a feature diagram model. The composition of the modular DSML features produces an integrated new feature that realizes the property of a conservative extension. With the generated DSML feature, a language workbench, such as MontiCore, can generate a parser and further tooling on a push-button basis. We lay this as the foundation for further research to be capable of covering not only the syntax, but all constituents of DSMLs.

REFERENCES

[1] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.

[2] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017*. Springer International Publishing, 53–70.

[3] María Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within modeling language definitions. *Model Driven Engineering Languages and Systems* (2009), 670–684.

[4] Tony Clark, Mark den Brand, Benoit Combemale, and Bernhard Rumpe. 2015. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*. Springer, 7–20.

[5] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*. Pittsburgh, United States.

[6] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA '12)*. ACM, New York, NY, USA.

[7] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. 2010. Empirical Language Analysis in Software Linguistics. In *SLE*. Springer, 316–326.

[8] Robert France and Bernhard Rumpe. 2007. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE*.

[9] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Voelkel, and Andreas Wortmann. 2015. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Scitepress, Angers, France.

[10] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. 2015. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling* 14, 2 (2015), 905–920.

[11] Ted Kaminski and Eric Van Wyk. 2013. Creating and using domain-specific language features. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*. ACM, 18–21.

[12] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[13] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages using Metamodels*. Pearson Education.

[14] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2008. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*.

[15] Thomas Kühn and Walter Cazzola. 2016. Apples and oranges: comparing top-down and bottom-up language product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 50–59.

[16] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Software Product Line Conference*. ACM, 71–80.

[17] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-oriented Language Families: A Case Study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*. ACM, New York, NY, USA, Article 11, 8 pages. https://doi.org/10.1145/2430502.2430518

[18] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. 2013. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering* (2013).

[19] Nenad Medvidovic and Richard N Taylor. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* (2000).

[20] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging software product lines engineering in the development of external dsls: A systematic literature review. *Computer Languages, Systems & Structures* 46 (2016), 206–235.

[21] Marjan Mernik. 2013. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software* 86, 9 (2013).

[22] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jézéquel. 2009. Weaving variability into domain metamodels. *Model driven engineering languages and systems* (2009), 690–705.

[23] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. 2009. Composing visual syntax for domain specific languages. *Human-Computer Interaction. Novel Interaction Methods and Techniques* (2009), 889–898.

[24] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2. ed.). Addison-Wesley, Boston, MA.

[25] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40.

[26] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. 2014. Automating Variability Model Inference for Component-Based Language Implementations. In *Proceedings of the 18th International Software Product Line Conference*. ACM, 167–176.

[27] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE, 233–238.

[28] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C L Kats, Eelco Visser, and Guido Wachsmuth. 2013. *{DSL} Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[29] Jules White, James H Hill, Jeff Gray, Sumant Tambe, Aniruddha S Gokhale, and Douglas C Schmidt. 2009. Improving domain-specific language reuse with software product line techniques. *IEEE software* 26, 4 (2009).