Towards Component-Based Development of Textual Domain-Specific Languages

Andreas Wortmann

Software Engineering
RWTH Aachen University
Aachen, Germany
http://www.se-rwth.de/

***Abstract*—Software-intensive systems are developed with the help of experts of different domains. This requires reifying their domain expertise in software, which raises the need for Domain-Specific Languages (DSLs) to bridge the gap between the problem space of the experts' experience and software development. Developing suitable DSLs still is prohibitively complex due to the lack of pervasive concepts for DSL reuse. Existing concepts either give rise to a conceptual gap between their abstractions and language definition constituents or are tied to specific technological spaces. To mitigates this, we present a novel conceptual model for the systematic reuse of textual DSLs. This technology-independent model promotes modularity and reusability based on language families that exhibit specific reuse interfaces. To realize these concepts, we conceived an extensible modelling infrastructure that supports the engineering of reusable textual DSLs using the MontiCore language workbench. This enables systematic reuse of textual DSLs for compatible technological spaces from which DSL engineers in many domains can greatly benefit.**

***Index Terms*—Software Language Engineering, Textual Languages, Language Components**

I. Introduction

Society increasingly depends on systems developed by experts from various domains using their own Domain-Specific Languages (DSLs) [1]. DSLs have become innovation drivers in many disciplines, including automotive, avionics, civil engineering, Industry 4.0, robotics, and software engineering itself. This, *e.g.,* led to the engineering of over 120 DSLs for software architectures [2] used in different domains and various technological spaces [3]. All of these need to be developed, maintained, and evolved on their own, which is costly, error-prone, and hinders progress in the multi-domain engineering of modern software-intensive systems.

Research in Software Language Engineering (SLE) [4] investigates the efficient and reliable engineering, maintenance, deployment, use, and evolution of DSLs to support software engineers and domain experts in efficiently developing future systems. Despite attempts to a systematic SLE, many DSLs are engineered ad-hoc, for very specific challenges, and very limited purposes only [5]. Hence, research has produced a multitude of solutions to facilitate creating DSLs. These include on metamodels [6], grammars [7], or abstract data types [8], interpreters [9] or code generators [10], and well-formedness rules defined in metalanguages [8] or programming languages [11]. For these, the SLE community has proposed various reuse techniques, based on experiences from general software reuse (*e.g.,* polymorphic [12] and parametric [13] reuse, composition [7] or variability [14]). Although these techniques address a wide range of scenarios, most support specific parts of DSL definitions (*e.g.,* abstract syntax or code generators) only and are limited to specific technological spaces. This complicates the engineering and customization of real-world DSLs for different usage scenarios, which ultimately hinders systems engineering by domain experts.

To mitigate this, we present the COLD4TXT conceptual model for component-based language development of textual DSLs that implement behavior with code generators (txtDSLs). In this model, *language components* with explicit interfaces of required and provided grammar rules, well-formedness rules, and code generators are the principal elements of reuse. Feature models arrange these components, according to their required and provided extension points, in language families. Thus selecting features governs how the language components are composed. Based on this model, we present a systematic method to describe and resolve the component's variability, as well as their customization.

As the technical realizations of composing grammars, well-formedness rules, and code generators have been presented already [10], [15], this contribution illustrates their conceptual framework consisting of:

1) The COLD4TXT conceptual model for reusable txtDSL components featuring explicit interfaces of required and provided elements.
2) A systematic method for engineering languages based on reusable txtDSL components.
3) A realization of both with the MontiCore language engineering workbench.

With these, reusing language components in different language families can greatly facilitate engineering DSLs.

In the following, Section II motivates our method by example. Afterwards, Section III presents txtDSL language components and Section IV our method to reuse theses for efficient txtDSL engineering. Ultimately, Section V debates observations, Section VI discusses related work, and Section VII concludes.

II. Example

Consider using Architecture Description Languages (ADLs) [2] – DSLs for the specification of software

 ISBN: 978-1-61208-752-8

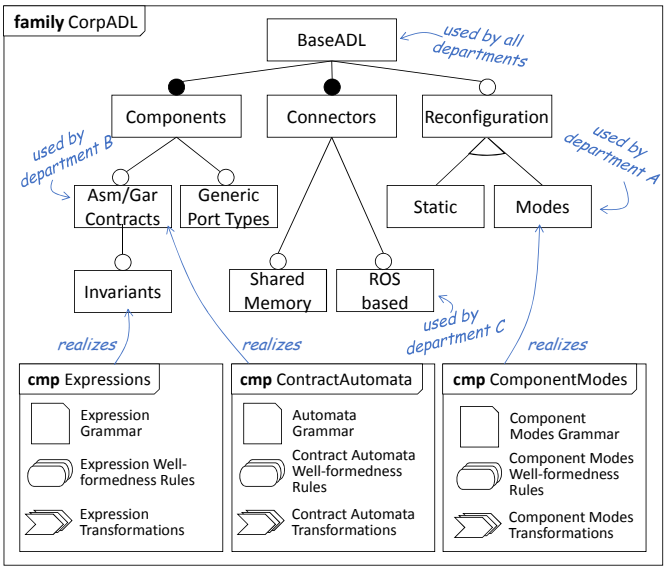

Fig. 1. A language family comprising features of language components that can fulfill the requirements of all three departments.

architectures – for the different departments of a large corporation. In each of these departments, some developers occasionally, maybe once a week, (re-)model parts of a specific software architecture (*e.g.,* of a train, a factory, or a mobile service robot). Instead of learning overly generic ADLs and operating with complex modeling guidelines that describe how to properly model with these, modelers of each department should be able to use their specific terminology and learn only the modeling elements required for their specific application.

Hence, while in general, these ADLs require some notion of components, ports, and connectors, each department has domain-specific requirements for the ADLs to be used:

- Department A (trains) requires components that support dynamic reconfiguration via components modes [16] to enable switching components related to country-specific technology when the train crosses a border.
- Department B (smart factories) demands components with assumption/guarantee contracts [17] that facilitate correct integration of new components when the factory reconfigures.
- Department C (robotics) demands novel connectors that support bridging architecture models with the robot operating system (ROS) [18].

Developing a general ADL that captures all of these concepts is not feasible as it complicates modeling in departments where only some of these modeling elements are required. Alternatively developing three specific ADLs – each with their specific infrastructure (*e.g.,* parsers, model checkers, code generators) – independently is costly and inefficient.

Instead, building suitable language components and combining these as required can significantly reduce the effort of fulfilling the departments' requirements. For our example, consider the language family of Figure 1: this family contains the language features required by the different departments and each feature is implemented by a language component comprising a combination of grammar, well-formedness rules, and code generators. By developing independent language components that implement the different features and by leveraging variability modeling techniques, the configuration of the base ADL for the different departments only requires selecting the appropriate language components and (semi-)automatically integrating these. If no appropriate features are available, developing and integrating novel language components and integrating these into existing language families reduces the effort of building a suitable ADL.

Our method to engineer and reuse language components considers both, planned variability and opportunistic reuse, and supports semi-automated composition of language component constituents in the technological space of the MontiCore [11] language workbench.

III. COLD4TXT Language Components

The conceptual model of COLD is a vision of language reuse that requires concretization. For txtDSLs, we have developed the COLD4TXT variant of COLD which realizes variability, explains how resolving variability affects the language components, how variability and customizability interact, how variability, customizability, the language facets' artifacts relate, and provides modeling techniques to realize this. At its core, COLD4TXT resolves variability and customizability through the additive composition of language components according to their explicitly provided and required extension points.

To enable this, COLD4TXT differs from COLD: In COLD4TXT, *language families* and *language components* replace language concerns and language facets of COLD, respectively: The language concerns of COLD provide both variability and customizability. This entails that they provide the complete customizability of their intrinsic language product line and express this towards the user despite only a small subset of customization options being available in the language product derived from the product line (namely these provided by the features selected for the product). In contrast, customizability should express means for tailoring languages that are not resolved by variability. Therefore, the language component comprising the derived language product provides customizability options instead. Moreover, to enable the proper composition of language components based on a feature selection, the COLD4TXT language components yield interfaces themselves. These interfaces guide and restrict their use in the variation interface's feature model and enable composing two language components (semi-)automatically, with only the implementation of adapters for generator composition requiring manual interaction [10]. To explain the effects of resolving variability and customizability in COLD4TXT, a *language component* consists of a

- one language component interface,
- one customization interface,

 ISBN: 978-1-61208-752-8

- up to one grammar artifact,
- arbitrary many well-formedness rule artifacts, and
- arbitrary many code generator artifacts.

The language component interfaces explicitly provide or require language grammar productions, well-formedness rules, or code generators. Also, they may yield constraints between these (*e.g.,* representing whether an extension point is optional or mandatory, or to express that selecting a provided code generator entails selecting a grammar production as well). The provided extension points for grammar rules identify productions of the contained CFG that are meant for reuse (*e.g.,* expressions of an imperative modeling language, method signatures of a class diagram language, *etc.*). The required extension points for grammar productions explicate productions that demand (optional or mandatory) extension for the contained syntax to be completed.

Specifying required well-formedness rules within the interface either demands complete specifications of the required well-formedness rules behavior (*i.e.,* their implementation) or demands conditions under which an independently provided well-formedness rule is suitable for the required rule (*i.e.,* some form of acceptance tests). The former entails having a specification that is precise enough to become a implementation automatically and the latter testing rarely would be complete. Hence, we decided to consider the set of well-formedness rules of a language component as its extension point. Thus, a language component can provide arbitrary many well-formedness rules that may or may not be used by other components, but it cannot (yet) describe that it requires additional well-formedness rules. Specifying the semantics of required well-formedness rules is subject to ongoing work. For code generators, language components leverage the notions of producer interface and product interface as introduced in [10]. Hence, language components may provide and require extension points that declare exactly one producer interface and one product interface. The customization interfaces of language interfaces comprise parameters of well-formedness rules and generators that are not meant to be resolved through the closed variation of language families but enable open customization instead. Such customization could be the numbers of initial states supported in models of a language component for an automaton DSL or the path a generator should produce artifacts in.

The language interfaces ground their required and provided extension points in the artifacts of their language components as illustrated in Figure 2. Here, the red concepts (solid lines) represent the language components and the yellow concepts (dashed lines) highlight their customization interface parts. The language components are part of language families. Aside from at least one language component, a *language family* contains a variation interface comprising a single feature model and a mapping that relates features to language component interfaces. By transitivity of language interface extension points, this also identifies one language component per feature. The feature model of the variation interface is developed by a

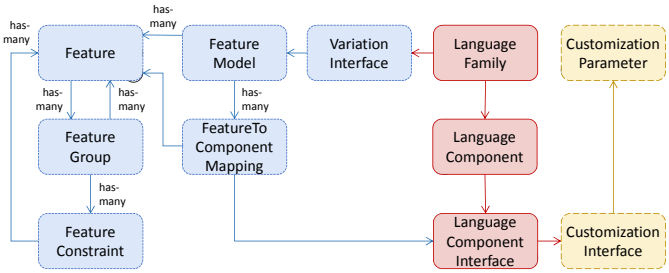

Fig. 2. Conceptual model for txtDSL reuse focusing on language families and their variation interfaces.

```
01 language family CorpADL {                          available language components
02   components
03     MontiArc, ContractAutomata, ComplexPortTypes, Expressions;
04   variation interface root BaseADL {                          ┐
05     mandatory Components {                                    │
06       optional AsmGarContracts { optional Invariants; }       │ variation
07       optional GenericPortTypes;                              │ interface
08     }                                                         │
09     // additional features relations                          ┘
10   }
11   root feature BaseADL uses MontiArc;
12   abstract feature Components;
13   feature AsmGarContracts uses ContractAutomata {
14     binds production Automaton to Components.ArcElement;
15     binds generator Automaton2Java to Components.BehaviorGenerator;
16     binds wfrs NonHierarchical;
17   }      feature definition
18   feature Invariants uses Expressions {
19     binds production Expression to AsmGarContracts.Expr.
20     binds Expression.All;
21     binds generator Expressions2POJO to AsmGarContracts.Guard2Java;
22   }                                  feature to component mappings
23   // additional features definitions
24 }
```

Fig. 3. Textual model of the `CorpADL` language family of Figure 1.

language family designer that intends to derive similar DSLs of joint buildings blocks. As such, the designer models the selection of a specific child feature implementing the extension points of its parent feature and specifies constraints between features in the `Feature2ComponentMappting`.

The language components are composed based on the arrangement of language components in the variation interfaces' feature model. From this, a new language component comprising their (possibly composed) artifacts together with a derived interface are synthesized. If there are required extension points not fulfilled by the selected features, these become part of the new component's interface.

COLD4TXT is realized as a language engineering framework using the MontiCore language workbench. To this end, we have developed modeling languages for language families, language components, feature configurations, and customization configurations as well as a toolchain that supports resolving variability and customizability.

The language family `CorpADL` of our example (*cf.* Figure 1) can be represented as illustrated in Figure 3. This family describes which language components it comprises (ll. 2-3), its variation interface in terms of a feature model (ll. 4-10), and defines its features (ll. 11-24). A feature either is a root feature (at most one), an abstract feature solely for grouping other features (such as the feature `Component`),

 ISBN: 978-1-61208-752-8

```
01 language component ContractAutomata {
02   grammar mc.automata.ca.ContractAutomata;
03
04   provides production ConractAutomatonMain;
05   provides AsmAutomaton for production AssumptionAutomaton;
06   provides GarAutomaton for production GuaranteeAutomaton;
07   requires mandatory Expr for production IGuardExpression;
08   provides generator Automaton2Java for ContractAutomatonMain {
09     producer IAutomatonGen;
10     product IAutomatonPairRealization;
11   }
12   requires generator Guard2Java for IGuardExpression {
13     producer IGuardExpressionGenerator;
14     product IGuardExpression
15   }
16
17   provides wfrs Hierarchical {
18     mc.automata.ca.coco.base.*;
19     mc.automata.ca.coco.hierarchical.*;
20   }
21   provides wfrs NonHierarchical {
22     mc.automata.ca.coco.base.*;
23     mc.automata.ca.coco.NoHierarchy;
24   }
25
26   parameters {
27     int max for mc.automata.cocos.NumHierarchyLevels.initialize;
28     String prefix for IAutomatonGen.generate;
29   }
30 }
31
```

provided grammar productions
provided generator
required grammar production
required generator
provided sets of well-formedness rules
customization parameters

Fig. 4. Model of the `ContractAutomata` component of Figure 1.

```
01 language component Expressions {
02   grammar mc.basic.expressions.Expressions;
03   provides production Expression;
04   provides wfrs All { mc.basic.expressions.*; }
05
06   provides transformation Expressions2POJO for Expression {
07     producer IExpressionJavaGenerator;
08     product IExpression;
09   }
10 }
```

Fig. 5. Model of the `Expression` component of Figure 1.

or is realized by a language component. Each feature of the latter kind defines how the provided extension points of its language component are mapped to the required extension points of its parent feature. For instance, selecting the feature `Invariants` entails that (1) its production `Expression` will be embedded [15] into the extension point `Expr` of the language component `AsmGarContracts` (l. 21); (2) its well-formedness rules provided via the extension point `All` will be reused (l. 22); and (3) its code generator provided via the extension point `Expressions2POJO` will be embedded into the code generator `Guard2Java` of language component `AsmGarContracts` (l. 23). The well-formedness rules of the language family ensure that these mappings are valid w.r.t. the language components illustrated in Figure 4 and Figure 5.

IV. Deriving Languages

Modeling language families with COLD4TXT first demands its instantiation for a specific technological space by providing modules for (1) analysing the compatibility of COLD4TXT models with the referenced technology space artifacts and (2) composing these artifacts according to COLD4TXT specifications as depicted in Figure 6. The former modules, for instance, check whether a well-formedness rule provided by a language component exists or whether a grammar production declared as an extension point indeed is an interface production. The latter modules take composition instructions (the binding mappings) and related artifacts, and compose these accordingly. For MontiCore, these modules are provided. Language engineers than can use this instance of COLD4TXT to engineer language components. Language family developers can reuse these in different contexts through arranging these in the variation interfaces. Language family users select the desired language features matching their requirements and use the COLD4TXT instance to synthesize a suitable language component. If this language component is incomplete w.r.t. its mandatory required extension points or parameters, it cannot be used as a DSL yet. In this case, the language family user has to specify the missing customization configuration, before a fully configured language component and the artifacts for a DSL in the corresponding technological space are derived.

For MontiCore, these artifacts are a synthesized CFG, the union of the selected well-formedness rules, and a code generator composed along its producer and product interfaces. These artifacts can be processed by MontiCore to produce a DSL that is completely independent of language families and language components. Moreover, the (possibly incomplete) language components derived from resolving variability and customizability can be used as parts of other language families again, which facilitates their reuse.

Based on a feature configuration, the COLD4TXT framework composes the language components associated with the selected features pairwise and top-down. The resulting component yields the provided extension points of the parent and child components. For each mandatorily required extension point (*e.g.,* `Expr` of language component `ContractAutomata`), if an implementation is defined by the binding mappings in the variation interface's feature model, then this extension point becomes optional and is copied to the interface of the new component as well. The sets of well-formedness rules from the parent component and the ones from the selected provided extension point of the child component are merged and provided as a new extension point in the new component. For the CFGs, COLD4TXT expects the responsible modules of the specific technology space to produce combined CFGs and adapters between the participating code generators accordingly.

For instance, selecting the features "Asm/Gar Contracts" and "Invariants" depicted in Figure 1 with the variation interface specified in Figure 3 entails combining the language components `ContractAutomata` (Figure 4) and `Expressions` (Figure 5) accordingly. The resulting language component is given in Figure 7. This component uses a synthesized CFG featuring contract automata and expressions (l. 2), the union of selected well-formedness rules, and the composed code generators. Its interface reduces the cardinality of the required grammar extension point `Expr` to optional (l. 7), adds the provided extension point `Expression` (l. 8), as well as the code generator for expressions (ll. 14-17)

 ISBN: 978-1-61208-752-8

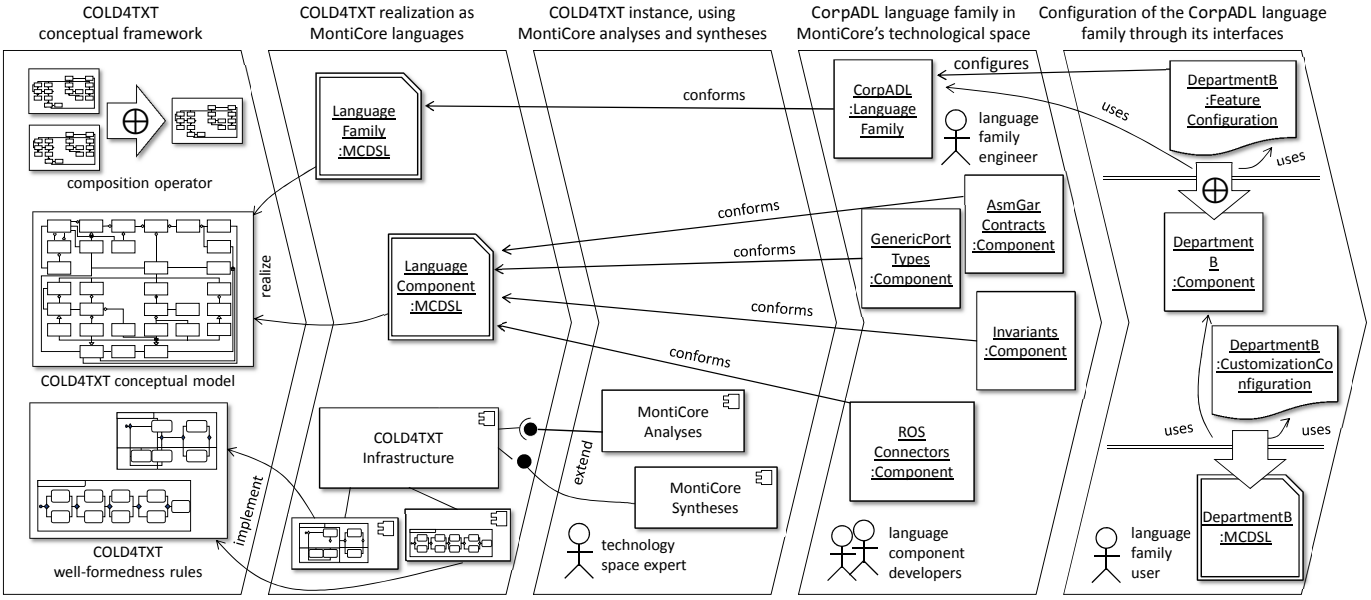

Fig. 6. After tailoring CORE4TXT for a specific technological space, developers can engineer language components to be used by language family developers to facilitate creating DSLs.

```
01 language component ContractAutomataWithExpresssions {
02   grammar mc.automata.ca.ContractAutomataWithExpressions;
03
04   provides production ConractAutomatonMain;                  // synthesized grammar
05   provides AsmAutomaton of production AssumptionAutomaton;
06   provides GarAutomaton of production GuaranteeAutomaton;
07   requires optional Expr for production IGuardExpression;   // became optional
08   provides production Expression;                           // added
09
10   provides transformation Automaton2Java for ContractAutomatonMain {
11     producer IAutomatonGen;
12     product IAutomatonPairRealization;
13   }
14   provides transformation Expressions2POJO for Expression {  // added
15     producer IExpressionJavaGenerator;
16     product IExpression;
17   }
18
19   provides wfrs All {                                       // synthesized set of well-formedness rules
20     mc.automata.ca.coco.base.*;
21     mc.automata.ca.coco.NoHierarchy;
22     mc.basic.expressions.*;
23   }
24
25   parameters {
26     int max for mc.automata.cocos.NumHierarchyLevels.initialize;
27     String prefix for IAutomatonGen.generate;
28   }
29 }
```

Fig. 7. Language component synthesized as result from selecting the features "Asm/Gar Contracts" and "Invariants" of Figure 3.

from the `Expressions` language component of Figure 5, and provides a new set of well-formedness rules (ll. 19-23). As this component does not require further extension, specifying values for its parameters enables MontiCore to derive a complete DSL from it.

V. Discussion

In contrast to the purely conceptual models of DSL reuse [19], [20], COLD4TXT supports capturing all DSL definition constituents at a sufficient level of abstraction to support the precise explanation of the effects of composing these, binding their variability, and resolving their customizability on its own.

The conceptual model of COLD4TXT aims to be independent of technological spaces as long as these enable to (1) identify grammar extension points; (2) compose grammars, sets of well-formedness rules, and code generators without eliminating the extension points in the process; (3) describe code generators and the generated products in terms of their interfaces; (4) identify parameters of well-formedness rules and code generators in an object-oriented fashion. While these are strong assumptions, we currently investigate applying COLD4TXT and its realization within the technological spaces of Neverlang [7] and Xtext [21]. Moreover, it currently only supports embedding in the sense of [10], whereas there are various other composition operators for txtDSLs. Whether and how supporting these is possible, also is ongoing research.

In the future, we aim to extend the notion of language components to feature additional constituents (*e.g.,* model-to-model transformations or editor fragments), support other forms of composition (*e.g.,* coordination or aggregation), and make the constraints of required well-formedness rule extensions more explicit.

VI. Related Work

Research on Language product lines (LPLs) [15], [22], [23] is scattered across different kinds of DSL definition constituents and technological spaces. And while we developed a notion of LPLs for the technological space of MontiCore [15] in particular, there currently is no actionable understanding of the variability of complete txtDSLs (*i.e.,* encompassing all four kinds of constituents). Moreover, (closed) variation rarely is connected with (open) customization to systematically reuse

 ISBN: 978-1-61208-752-8

DSLs in general. There are only a few solutions that consider either txtDSL variation or customization across different kinds of DSL definition constituents. These include a few language workbenches [24], such as Argyle [23], Neverlang [7], or the combination of SDF and FeatureHouse [22].

In Argyle [23], DSLs are constructed from language assets that resemble concerns and comprise syntax, data types, and code generation templates. A feature model arranges assets according to their dependencies, which demands their white-box apriori composition that hinders the reuse of facets. In contrast, COLD4TXT will be based on our exploratory work [15] that makes extension points of concerns explicit and supports the black box composition of their artifacts through the generation of suitable adapters between these.

SDF and FeatureHouse realize variability based on compositional language modules containing grammar rules, typing rules, and evaluation rules [22]. It also focuses on the white-box composition of artifacts and interpretation. Similar partial solutions towards variation or customization of selected kinds of DSL definition constituents are available from a variety of language workbenches. For instance, ableC [25] is an extensible C language that leverages attribute grammars to reuse syntax and semantics, MPS [21] enables reuse of projectional languages with views and model transformations, and Spoofax [8] supports reuse of textual, interpreted languages. All of these concepts for (partial) DSL reuse focus on specific technological spaces. and lack support for integrated reuse across variation and customization.

VII. Conclusion

We have presented the novel COLD4TXT conceptual framework to facilitate reusing textual DSLs through systematic variability and customizability. In COLD4TXT, language families capture txtDSL variability as feature models and realize it via composition of language components according to their interfaces. Composing language components yields new language components that may demand further extension or customization before these can be translated into complete DSLs for specific contexts. Making the interfaces of language components explicit enables reusing these in different language families. This facilitates engineering textual DSLs for different contexts and fosters the application of DSLs.

References

[1] M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Alff, L. Wiart, A. Wortmann, and A. Nordmann, "Using language workbenches and domain-specific languages for safety-critical software development," *Software & Systems Modeling*, pp. 1–24, 2018.

[2] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What Industry Needs from Architectural Languages: A Survey," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 869–891, 2013.

[3] I. Kurtev, J. Bézivin, and M. Aksit, "Technological spaces: An initial appraisal," *CoopIS, DOA*, vol. 2002, 2002.

[4] K. Hölldobler, B. Rumpe, and A. Wortmann, "Software Language Engineering in the Large: Towards Composing and Deriving Languages," *Computer Languages, Systems & Structures*, vol. 54, pp. 386–405, 2018.

[5] J. Whittle, J. Hutchinson, and M. Rouncefield, "The State of Practice in Model-Driven Engineering," *Software, IEEE*, vol. 31, no. 3, pp. 79–85, 2014.

[6] T. Kühne, "Matters of (meta-) modeling," *Software & Systems Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[7] E. Vacchi and W. Cazzola, "Neverlang: A framework for feature-oriented language development," *Computer Languages, Systems & Structures*, vol. 43, pp. 1–40, 2015.

[8] G. H. Wachsmuth, G. D. P. Konat, and E. Visser, "Language Design with the Spoofax Language Workbench," *IEEE Software*, vol. 31, no. 5, pp. 35–43, 2014.

[9] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting efficient and advanced omniscient debugging for xDSMLs," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2015, pp. 137–148.

[10] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Modeling Language Variability with Reusable Language Components," in *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, 9 2018.

[11] K. Hölldobler and B. Rumpe, *MontiCore 5 Language Workbench Edition 2017*, ser. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017. [Online]. Available: http://www.se-rwth.de/phdtheses/MontiCore-5-Language-Workbench-Edition-2017.pdf

[12] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Safe model polymorphism for flexible modeling," *Computer Languages, Systems & Structures*, vol. 49, pp. 176–195, 2017.

[13] J. de Lara and E. Guerra, "Generic Meta-modelling with Concepts, Templates and Mixin Layers," in *Model Driven Engineering Languages and Systems*, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 16–30.

[14] T. Kühn, W. Cazzola, and D. M. Olivares, "Choosy and picky: configuration of language product lines," in *Proceedings of the 19th International Conference on Software Product Line*. ACM, 2015, pp. 71–80.

[15] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic composition of independent language features," *Journal of Systems and Software*, vol. 152, pp. 50–69, 2019.

[16] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[17] M. Broy and K. Stølen, *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2012.

[18] K. Adam, K. Hölldobler, B. Rumpe, and A. Wortmann, "Modeling Robotics Software Architectures with Modular Model Transformations," *Journal of Software Engineering for Robotics (JOSER)*, vol. 8, no. 1, pp. 3–16, 2017.

[19] T. Clark, M. v. d. Brand, B. Combemale, and B. Rumpe, "Conceptual Model of the Globalization for Domain-Specific Languages," in *Globalizing Domain-Specific Languages*, ser. LNCS 9400. Springer, 2015, pp. 7–20.

[20] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, and A. Wortmann, "Concern-oriented language development (COLD): Fostering reuse in language engineering," *Computer Languages, Systems & Structures*, vol. 54, pp. 139–155, 2018.

[21] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *{DSL} Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. [Online]. Available: http://www.dslbook.org

[22] J. Liebig, R. Daniel, and S. Apel, "Feature-oriented language families: a case study," in *VaMoS*, 2013.

[23] C. Huang, A. Osaka, Y. Kamei, and N. Ubayashi, "Automated DSL construction based on software product lines," in *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, 2015, pp. 1–8.

[24] S. Erdweg, T. Van Der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, and Others, "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," *Computer Languages, Systems & Structures*, vol. 44, pp. 24–47, 2015.

[25] T. Kaminski, L. Kramer, T. Carlson, and E. Van Wyk, "Reliable and automatic composition of language extensions to C: the ableC extensible language framework," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 98, 2017.

 ISBN: 978-1-61208-752-8