Variability Management in Domain-Specific Languages

David Méndez-Acuña

Variability Management in Domain-Specific Languages

David Méndez-Acuña

University of Rennes 1 and INRIA, France
david.mendez-acuna@inria.fr

Abstract. Domain-specific languages (DSLs) have demonstrated their capability to reduce the gap between the problem domain and the technical decisions during the software development process. However, building a DSL is not an easy task because it requires specialized knowledge and skills. Moreover, the challenge becomes even more complex in the context of multi-domain companies where several domains coexist across the business units and, consequently, there is a need of dealing not only with isolated DSLs but also with families of DSLs. To deal with this complexity, the research community has been working on the definition of approaches that use the ideas of Software Product Lines Engineering (SPLE) for building and maintaining families of DSLs. In this paper, we present a PhD thesis that is aimed to contribute to this effort. In particular, we explain the challenges that need to be addressed during the process of going from a family of DSLs to a *software language line*. Then, we briefly discuss the state of the art, and finally we introduce a research plan.

1 Motivation

Domain-specific languages (DSLs) allow domain experts the expression of solutions directly in terms of relevant domain concepts and, for example, use generative mechanisms to transform specifications of DSLs into software artifacts (e.g. code, configuration files or documentation). Thus, abstracting away from the complexity of the rest of the system and the intricacies of its implementation. As a result, the construction of DSLs is becoming a recurrent activity during the development of software intensive systems [11]. However, the engineering of DSLs is a complex task in the context of large companies such as Thales where the users often have different –and sometimes conflicting– requirements on the same DSL. For example, certain user may require a textual concrete syntax while another user prefers a graphical concrete syntax. Worst, for two different users, the meaning (or semantics) of a particular concept of the language may differ. When this occurs, we have a set of different DSLs that share some commonalities and that are distinguished each other by some particularities. In the literature this is known under the term of *families of DSLs* [14].

Figure 1 illustrates this situation by presenting a segment of the family of DSLs for modeling finite state machines deeply studied in [3] and composed of:

Rhapsody [8], Classical statecharts [9], and UML state machines diagrams [7]. In this case, the commonalities among the three DSLs are the basic concepts of states and transitions. In turn, the differences are focused on the concepts of timed transitions and simultaneous events; while Rhapsody and UML include the notion of timed transitions, it is not supported in classical statecharts. On the other hand, classical statecharts support simultaneous events whereas both Rhapsody and UML adopt to the idea of run-to-completion i.e., each event is attended after processing the precedent one. Note that the second difference corresponds to a conflict in the semantics of the events dispatching functionality; all languages support that functionality but its semantics varies.

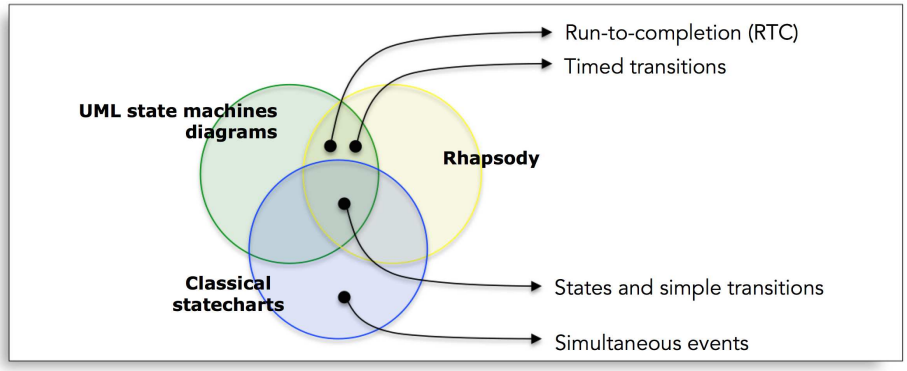

Fig. 1: A family of DSLs for finite state machines

Although the problem of dealing with families of DSLs is becoming more and more recurrent, currently there is little support for their implementation. Usually, each DSL member of the family is developed in isolation and from scratch what is not convenient because the construction of DSLs is a challenging task; to successfully perform such activity, an engineer must own not only quite solid modeling skills but also the technical expertise for conducting the definition of specific artifacts such as grammars, metamodels, compilers, and interpreters, among others. Nevertheless, as remembered by J.M. Favre in [5] **software languages are software too** and, consequently, there is room for application of software engineering techniques that facilitate their construction process (i.e., software languages engineering [12]). In particular, the research community has realized that the ideas behind Software Product Lines Engineering (SPLE) can be used for facilitating the construction of families of DSLs. This paper presents a PhD thesis aimed to contribute in this effort. To do so, we briefly discuss the challenges that should be addressed during the process of going from a family of DSLs to a *software languages line*. Then, we discuss the state of the art, and finally we present a research plan.

2 Problem Statement

We have identified three challenges towards the construction software languages lines.

- **Languages modular design.** The main purpose of an approach for well-engineering families of DSLs is to increase the reuse in the sense that those language segments that are shared between two or more members of the family can be easily shared without being implemented independently and from scratch. To do so, it is necessary to separate those shared segments and encapsulate them in such a way that their specification artifacts can be actually used as part of the specification of the languages that require it. This separation implies a mechanism that allows to specify a language in different modules (a.k.a. language units) that can be later composed together to provide a complete language specification.
- **Multi-dimensional variability modeling.** It is worth noting that differences and particularities among members of a family of DSLs can be found in one or several dimensions of the specification. Indeed, the work presented in [2] provides a classification of the possible types of variability that can be found within families of DSLs. A brief summary of this classification is introduced below; the challenge in this case is to be able to propose an approach that supports these types of variability.
  - *Functional variability:* One of the motivations for implementing families of DSLs is to offer customized languages that provide only the constructs required by certain type of users. The hypothesis is that the user will adopt the language easier if the language only offers the constructs he/she actually needs. If there are additional concepts the complexity of the language (and the tools) needlessly increases and *"the users are forced to rely on abstractions that might not be naturally part of the abstraction level at which they are working"* [17]. Functional variability refers to the capability of selecting the desired language constructs for a particular type of user. Because of the abstract syntax of the language is the base of the specification –i.e., there can not be definitions neither in the concrete syntax nor the semantics for concepts that do not exist in the abstract syntax– the functional variability relies on the activity of selecting the subset of the abstract syntax required for a particular user.
  - *Syntactic variability:* Depending on the context and, again, on the type of user, the use of certain types of concrete syntax may be more appropriate than other one. Consider for example the dichotomy between textual or graphical notations. Empirical studies such as the presented in [15] show that, for a specific case, graphical notations are more appropriate than textual notations whereas other evaluation approaches argue that textual notations have advantages in cases where models become large [4]. More intermediate perspectives (e.g., [10]) state that graphical and

textual notations more that mutually exclusive are complementary. Syntactical variability refers to the capability of supporting different representations for the same language construct. In terms of the specification, the syntactical variability can be viewed as different implementations of the concrete syntax specification unit for a given abstract syntax specification unit.

- *Semantic variability:* Another problem that has gained attention in the literature of software languages engineering is the semantic variation points existing in software languages. A semantic variation point appears where the same construct can have several interpretations. Consider for example the semantics differences that exist between state machines languages explored in [3]. For example, the construct *fork* can be interpreted as a concurrency point where all the output transitions are dispatched simultaneously or simply as a bifurcation point where the output transitions are dispatched sequentially. Semantic variability refers to capability of supporting different interpretations to the same language construct. In terms of the specification, semantic variability can be viewed as different implementations of the semantic specification unit for a given abstract specification unit.

– **Language units composition.** After successfully modeling the variability, the next challenge is to compose a concrete DSL from a configuration of the family. In other words, after selecting the required language units, they have to interact each other for constituting a DSL that actually works. To do so, a mechanism for language unit composition is required. The main requirement on this composition mechanism is that allows to select from a set of language units the ones that should be combined. To do so, an external language that allows to express the composition is needed. In the context of software architecture that should be an architectural description language that allows to select a set of given components and compose them together. Note that this requirement is strongly related with the modularization mechanism.

3 Related work

The idea of families of languages has already been discussed in the software engineering literature. We can find *"integral approaches"* that propose an integral solution to the problem by addressing the three challenges presented above. Besides, there are *"partial approaches"* that deal only with one or two of these challenges (e.g., approaches for components-based DSLs development deal with languages modular design but do not consider variability management). In this section, we briefly discuss the strengths and weaknesses of integral approaches. Partial approaches will discussed in a publication currently in progress.

In the work presented in [16] the modularization problem is treated by using Neverlang [1] i.e., a tool that allows the expression of a software language in separate language units that can be later composed for generating an interpreter. In

turn, the variability management mechanism is addressed by using the Common Variability Language (CVL). The main advantage of that work is that there is a clear mapping between the variability modeling approach with the modularization tool. The specifically the authors present the notion of “slices" as the mechanism for composing certain features of a language in Neverlang, then, providing support for functional and semantical variability. However, in Neverlang the abstract and the concrete syntax of language units are defined in the same artifact (i.e., a BNF-like grammar). As a result, it is not possible to support syntactical variability. Besides, there is not any support for static validation of compatibility between different language units. Hence, compatibility problems can only be detected in composition time by reading the errors produced by the composition tool.

The work presented in [14] introduces the concept of *crosscutting modularization*, that is, the capability of decomposing a language not only into different language units but also decomposing a language feature into several tool features in order to support not only functional variability but also syntactical and semantical variability. Each tool feature represents a dimension of the language specification (e.g., syntax, semantics, constraints, documentation). The weaknesses of this approach are fundamentally the lack of static verification of composability and the lack of support for the definition of constraints in the variability model. Using this approach, the language engineer may produce invalid configurations by selecting, for example, a semantic feature that requires a non-selected abstract syntax feature. To void this, the variability model should be annotated with dependency constraints –what may produce as many constraints as existing features in the model–. We claim that this process can be facilitated by considering the definition of a staged processes where each dimension of the variability is defined in a different variability model and they are presented in order to the user in such a way that each step only includes selectable features.

Finally, work presented in [2] is a formalization of the foundations for managing variability managment in software languages lines. It considers the notions of functional, semantic, and syntactic variability. Besides, the authors present a languages benchmark called MontiCore [13] that supports modularization of modeling languages. Although there is not a technical impediment for implementing a family of DSLs using this variability modeling mechanism in junction with MontiCore, to the best of our knowledge, there is not a concrete implementation of an approach that includes both functionalities at the same time.

4 Proposed approach

Our solution approach includes one solution strategy for each one of the aforementioned challenges. These strategies are summarized below:

– **Components-based DSLs’ development.** We propose to address languages modularization by means of an approach for components-based DSLs’ development where the main concept is the notion of language units that

interact each other by means of languages interfaces. The idea is that dependencies between language units are expressed as software interfaces that offer capabilities for compatibility checking and, in a latter phase, languages composition. Our preliminary results, suggest the need of the definition of different types of languages interfaces that support different scenarios for languages modularization such as languages embedding or languages extension.

- **Multi-dimensional CVL for variability modeling:** To deal with variability modeling within families of DSLs, we propose to enhance the CVL (Common Variability Language) with capabilities for multi-dimensional variability modeling. We choose CVL because it provides support not only for variability models but also for implementation models which facilitate the mapping between language features and language units. Besides, it introduces the notion realization models constitute a mechanism for configuring a DSL by offering multi-staged variability.

- **Language units composition:** In order to perform the composition of several language units we explore two strategies: compilation based composition and interpretation based composition. In the first case, the idea is to compose the language units specifications and produce a complete specification that can be latter used for automatically generating language tooling such as editors or type-checkers. In the second case, the idea is to maintain specification separated and generate the corresponding tooling for each one. After that, the services of each of tooling are orchestrated to offer an infrastructure for the DSL. Each strategy presents advantages and disadvantages. For example, whereas interpretation based composition may impact the performance of the language, it enables variability at runtime.

5 Evaluation & validation plan

Currently, the plan for the evaluation of the approach is based on two case studies. The first one is a family of languages for finite state machines that presents the syntax and semantics variation points that can be found in languages for expressing state machines and that have been largely studied in the literature (e.g., [3,6]). It is worth to mention that this is a real-world case study that can be found in the context of Thales. The second case study is a family of languages for extended feature models that shows the different interpretation that constructs such as feature attributes have received.

The two case studies include the three dimensions of the variability. However, whereas the former requires the implementation of operational semantics the second one requires the implementation of translational semantics. Like this, we validate that our ideas are useful not only for executable DSLs but also for generative approaches that use transformations.

6 Current status and planned time-line

Planned time-line in terms of the expected contributions and current status (red line) is shown in Figure 2.

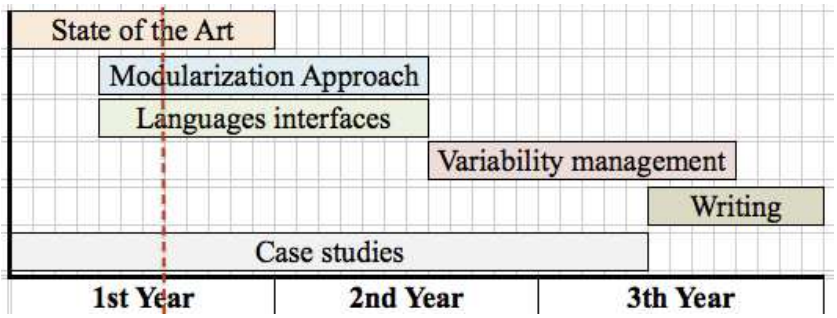

Fig. 2: Planned time-line and current status

Acknowledgments

The research presented in this paper is supported by the European Union within the FP7 Marie Curie Initial Training Network "RELATE" under grant agreement number 264840 and VaryMDE, a collaboration between Thales and INRIA.

References

1. W. Cazzola. Domain-specific languages in few steps: The neverlang approach. In *In Proc. of Intl. Conf. on Software Composition (SC), volume 7306 of LNCS*, pages 162–177. Springer, 2012.
2. M. Cengarle, H. Gr^nniger, and B. Rumpe. Variability within modeling language definitions. In A. Sch¸rr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 670–684. Springer Berlin Heidelberg, 2009.
3. M. Crane and J. Dingel. Uml vs. classical vs. rhapsody statecharts: not all models are created equal. *Software & Systems Modeling*, 6(4):415–435, 2007.
4. H. Eichelberger and K. Schmid. A systematic analysis of textual variability modeling languages. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 12–21, New York, NY, USA, 2013. ACM.
5. J.-M. Favre, D. Gasevic, R. L‰mmel, and E. Pek. Empirical language analysis in software linguistics. In *Software Language Engineering*, volume 6563 of *LNCS*, pages 316–326. Springer, 2011.
6. H. Fecher, J. Sch^nborn, M. Kyas, and W.-P. de Roever. 29 new unclarities in the semantics of uml 2.0 state machines. In K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer Berlin Heidelberg, 2005.

7. O. Group. Uml specification, version 2.0, 2005.
8. D. Harel and H. Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). In H. Ehrig, W. Damm, J. Desel, M. Grofle-Rhode, W. Reif, E. Schnieder, and E. Westk‰mper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2004.
9. D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, Oct. 1996.
10. F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In R. Paige, A. Hartman, and A. Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2009.
11. J.-M. Jézéquel, D. Méndez-Acuña, T. Degueule, B. Combemale, and O. Barais. When Systems Engineering Meets Software Language Engineering. In *CSD&M'14 - Complex Systems Design & Management*, Paris, France, Nov. 2014. Springer.
12. A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008.
13. H. Krahn, B. Rumpe, and S. Vˆlkel. Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010.
14. J. Liebig, R. Daniel, and S. Apel. Feature-oriented language families: A case study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.
15. B. Mora, F. GarcÌa, F. Ruiz, and M. Piattini. Graphical versus textual software measurement modelling: an empirical study. *Software Quality Journal*, 19(1):201–233, 2011.
16. E. Vacchi, W. Cazzola, S. Pillay, and B. Combemale. Variability support in domain-specific language development. In M. Erwig, R. Paige, and E. Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 76–95. Springer International Publishing, 2013.
17. S. Zschaler, P. S·nchez, J. Santos, M. AlfÈrez, A. Rashid, L. Fuentes, A. Moreira, J. Ara˙jo, and U. Kulesza. Vml* ? a family of languages for variability management in software product lines. In M. van den Brand, D. Ga?evi?, and J. Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 82–102. Springer Berlin Heidelberg, 2010.