Towards Language Interfaces for DSLs Integration

Thomas Degueule
INRIA
thomas.degueule@inria.fr

Developing software-intensive systems involves many stakeholders who bring their expertise on specific concerns of the developed system. Model-Driven Engineering (MDE) proposes to address each concern separately with a dedicated Domain-Specific (possibly modeling) Language (DSL) closely tied to the needs of each stakeholder [4]. With DSLs, models are expressed in terms of problem-level abstractions. Associated tools are then used to semi-automatically transform the models into concrete artifacts. However, the definition of a DSL and its tooling (e.g., checkers, editors, generators, model transformations) still requires significant development efforts for, by definition, a limited audience.

DSLs evolve as the concepts in a domain and the expert understanding of the domain evolve. A mere example is the addition, refinement or removal of features from a DSL, with possibly the intent to ensure the compatibility between the subsequent versions. Additionally, the current practice in industry has led to widespread use of small independently developed DSLs leading to challenges related to the sharing of languages and corresponding tools [6]. For example, the core concepts of an action language can be shared by all DSLs that encompass the expression of actions. Finally, while more and more DSLs are developed in various domains, recurrent paradigms are observed (e.g., state-transition, classifiers) with their own syntactic and semantic variation points reflecting the domain specificities (e.g., family of finite-state machines).

Given the DSL development costs, redefining from scratch a new ecosystem of tools for each variant of a DSL is not scalable. Instead, one would like to leverage the commonalities of these languages to enable reuse of existing tools. An underlying challenge is the modular definition of languages, i.e., the possibility to define either self-contained or incomplete language components (in terms of syntax and semantics) that could be recomposed afterwards for the definition of new DSLs. To support modularity, DSLs designers should be able to define proper provided and required interfaces for each language component, together with composition operators.

To improve modularity and abstraction capabilities in software language engineering and support the aforementioned scenarios, we advocate the definition of explicit language interfaces on top of language implementations. Language interfaces allow to abstract some of the intrinsic complexity carried in the implementation of languages, by exposing meaningful information concerning an aspect of a language (e.g., syntactical constructs) and for a specific purpose (e.g., composition, reuse or coordination) in an appropriate formalism. In this regard, language interfaces can be thought of as a reasoning layer on top of language implementations. The definition of language interfaces relies on proper formalisms for expressing different kinds of interfaces and binding relations between language implementations and interfaces. Using language interfaces, one can vary or evolve the implementation of a language while preserving tools and analyses defined over its interface. Language interfaces also facilitate the modular definition of languages by enabling the description of required and provided interfaces of a language (or language component). Syntactical or semantical composition operators can then be defined upon these interfaces. Languages interfaces may be crafted manually or automatically inferred from an implementation.

Model types [5] are an illustration of such kind of interfaces. Model types are interfaces on the abstract syntax of a language (defined by a metamodel). Models are linked to model

types by a typing relation. Most importantly, model types are linked one to another by subtyping relations, providing model polymorphism, i.e., the ability to manipulate a model through different interfaces. Model polymorphism enables the definition of generic tools that can be applied to any model matching the interface on which they are defined, regardless of the concrete implementation of their language. Model types can also be used to filter the information exposed from the abstract syntax of a language. Doing so, they can define language viewpoints by extracting the appropriate information on a system for one specific development task of a stakeholder. Model types are supported by a model-oriented type system that leverages family polymorphism [3] and structural typing to abstract the conformance relation standing between models and metamodels with a typing relation between models and model types.

We incorporated these concepts into Melange [2], a new language for DSLs designers and users. Melange is a language-based, model-oriented programming language in which DSLs designers can manipulate languages definitions with high-level operators (e.g., inheritance, composition, slicing) and express their relations through the definition of metamodels, language interfaces, and transformations. Melange provides DSLs users with an action language where models are first-class typed citizens and embeds a model-oriented type system that natively provides model polymorphism through model typing. We applied Melange on two industrial use cases to maximize the reuse of DSLs ecosystems: managing syntactic and semantic variation points in a family of FSM languages; providing an executable extension of Capella [1], a large-scale system engineering modeling language.

References

1. Capella. https://www.polarsys.org/projects/polarsys.capella (2014)
2. The Melange Language. http://melange-lang.org (2015)
3. Ernst, E.: Family polymorphism. In: ECOOP 2001—Object-Oriented Programming, pp. 303–326. Springer (2001)
4. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering. pp. 37–54. IEEE Computer Society (2007)
5. Steel, J., Jézéquel, J.M.: On model typing. Software & Systems Modeling 6(4), 401–413 (2007)
6. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. Software, IEEE 31(3), 79–85 (2014)