Modularity: G: Interoperability and Composition of DSLs with Melange

Thomas Degueule
Inria, France
thomas.degueule@inria.fr

Abstract

Domain-Specific Languages (DSLs) are now developed for a wide variety of domains to address specific concerns in the development of complex systems. However, DSLs and their tooling still suffer from substantial development costs which hamper their successful adoption in the industry. For over a decade, researchers and practitioners have developed language workbenches with the promise to ease the development of DSLs. Despite many advances, there is still little support for advanced scenarios such as language evolution, composition, and interoperability. In this paper, we present a modular approach for assembling DSLs from other ones and seamlessly evolving them, while ensuring the reuse of associated tools through subsequent versions or across similar DSLs. We introduce the theoretical foundations of our approach, its implementation in the Melange language workbench, and summarize its benefits on various case studies.

1. Problem and Motivation

The development of complex 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 Language (DSL) closely linked to the needs of each stakeholder [15]. With DSLs, stakeholders express their models in terms of problem-level abstractions. Associated tools are then used to semi-automatically transform the models into concrete software artifacts. Amongst the different kinds of DSLs (internal, embedded, external), external DSLs offer the best flexibility [12]. However, the definition of an external DSL (syntax and semantics) and its environment (*e.g.* checkers, editors, generators, IDEs) still requires substantial development efforts for, by definition, a limited audience. The Software Language Engineering community thus focuses on the development of tools and methods that ease the development of DSLs.

Although DSLs are by definition tied to a particular domain, one can see the emergence of recurrent paradigms shared by various DSLs used in different areas. Finite-state machine (FSM) languages, for instance, are used in many distinct contexts (*e.g.* language processing, user interfaces, systems and software engineering). While variants of FSM languages share many common concepts, they also exhibit syntactic and semantic variation points imposed by the specificities of the domain [6]. Redefining from scratch a new FSM language for each new domain contrasts with the good practices established in software engineering. Instead, language workbenches should provide language designers with the ability to import existing DSL specifications, customize and compose them to build new ones for other domains.

Another recurring problem in DSL development results from the fact that the tooling defined around a given language is highly coupled with it. Therefore, DSLs are extremely fragile to evolution: when a DSL evolves, its environment (*e.g.*, IDE) and the models created from it must be updated consequently. As DSLs evolve at a rather fast pace, this has severe consequences. The coupling between DSLs and their tooling also prevents the manipulation of models in different modeling environments. It is for instance not possible to share the same model between two FSM modeling environment (*e.g.* used by two different stakeholders), even when they share a lot of commonalities. This lack of flexibility and interoperability makes the work of language users difficult, as they cannot benefit from tools defined for different yet similar DSLs.

In this paper, we identify that this lack of flexibility arises from the theoretical foundations of MDE. To circumvent these limitations, we propose the notion of *language interface* that we use to design composition operators that ease the work of language designers, and to design a dedicated type system that provides more flexibility in model manipulation to DSL users.

The remainder of this paper is organized as follows. Section 2 gives some background notions and an overview of the related work. Section 3 details our approach. Section 4 presents the Melange language workbench and the results we gathered through its application to various case studies.

2. Background and Related Work

DSLs are typically defined through three main concerns: abstract syntax, concrete syntax(es) and semantics. Various approaches may be employed to specify each of them, usually using dedicated meta-languages [34]. The abstract syntax specifies the domain concepts and their relations and is defined by a metamodel or a grammar. This choice often depends on the language designer's background and culture. Examples of meta-languages for specifying the abstract syntax of a DSL include EMOF [1] and SDF [17]. The semantics of a DSL can be defined using axiomatic semantics, denotational semantics, operational semantics, and their variants [26]. Concrete syntaxes are usually specified as a mapping from the abstract syntax to textual or graphical representations, *e.g.* through the definition of a parser or a projectional editor [36]. In this paper, we focus on DSLs whose abstract syntaxes are defined with metamodels and whose semantics are defined in an operational way through the definition of computational steps designed following the interpreter pattern. Computational steps may be defined in different ways, *e.g.* using aspect-oriented modeling [18] or endogenous transformations. In this paper, however, we only focus on the weaving of computational steps in an object-oriented (OO) fashion with the interpreter pattern. In such a case, specifying the operational semantics of a DSL involves the use of an action language to define methods that are statically introduced directly in the corresponding concepts of its abstract syntax [19].

2.1 Modular Development of DSLs

Recent work in the community of Software Language Engineering focused on language workbenches that support the modular design of DSLs, and the possible reuse of such modules [21, 32]. Besides, particular composition operators have been proposed for unifying or extending existing languages [11, 24]. Other techniques have been studied for addressing the challenge of language extension and composition, such as projectional editing [35] or composable meta-languages [37]. Other authors demonstrated the possibility to create language modules using attribute grammars [20, 25, 29]. MontiCore applied modularity concepts for designing new DSLs by extending an existing one, or by composing other DSLs [22]. Finally, some works propose to leverage concepts from the component-based software engineering community to modularly develop DSLs [32, 38]. However, while most of the approaches propose either a diffuse way to reuse language modules, or to reuse as is complete languages, there is still little support for easily assembling language modules with customization facilities (*e.g.* restriction, specialization) in order to finely tune the resulting DSL according to the language designer's requirements.

2.2 Model Transformation Reuse

To ease the use of a DSL by stakeholders, language designers usually develop an associated modeling environment with dedicated editors, simulators, transformations, generators, *etc.*. Typically, these different tools are model transformations that manipulate the models conforming to a given DSL. To state whether a model conforms to a given DSL, MDE relies on the *conformance relation* that stands between a model and its metamodel [2, 3, 14]. The conformance relation plays a crucial role in MDE as it identifies *which* models are valid instances of a given DSL and *how* they should be safely manipulated. A fundamental property of the conformance relation is that a model conforms to one, and only one, metamodel. Therefore, when the metamodel defining the abstract syntax of a DSL evolves, the models no longer conform to it. As a consequence, all the tools and transformations defined around it must be updated. The research community has identified this problem of transformation reuse and proposed techniques to design generic model transformations. Varró and Pataricza introduced *variable entities* in patterns for declarative transformation rules [33]. Later, Cuccuru et al. introduced the notion of semantic variation points in metamodels [7]. Sánchez Cuadrado and García Molina propose a notion of substitutability based on model typing and model type matching [28]. De Lara and Guerra present the *concept* mechanism, along with model templates and mixin layers [8]. However, all these approaches require either to design metamodels and transformations in a special way, or to explicitly write the bindings between similar DSLs. As we shall see in the next section, our approach does not require any additional work from the language designer or user when the languages are similar enough, such as different versions or variants of a DSL.

3. Approach and Uniqueness

We structure our approach around two interconnected contributions: a dedicated type system for language interoperability and an algebra of operators for language composition. These two contributions rely on a common core concept of *language interface*. The operators and relations discussed in this section are summarized in Figure 1.

Language interfaces allow to abstract some of the intrinsic complexity carried in the implementation of languages, by exposing meaningful information (i) concerning an aspect of a language (*e.g.* syntactical constructs) (ii) for a specific purpose (*e.g.* composition, reuse, coordination) (iii) in an appropriate formalism (*e.g.* a metamodel). In this regard, language interfaces can be thought of as a reasoning layer atop 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. Interfaces may be manually crafted (thereby defining a contract) or automatically inferred from an existing language. Using language interfaces, one

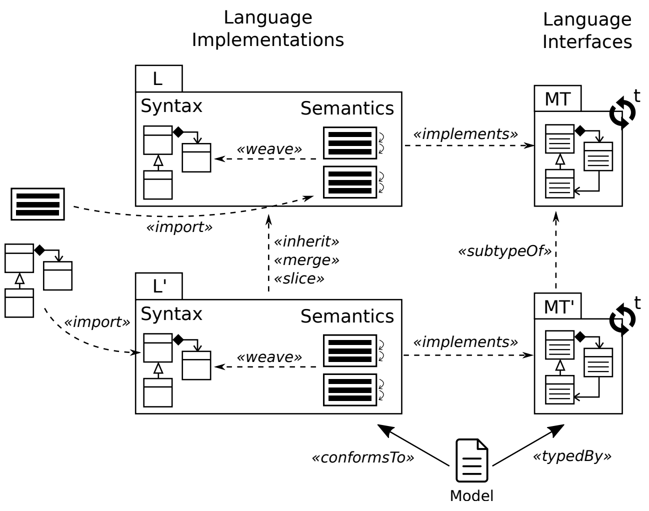


Figure 1. Model Types (MT) as a Typing Layer on top of Language Implementations

can vary or evolve the implementation of a language while preserving its interface.

In this paper, we focus on one kind of language interface: *model types*. Model types are structural interfaces over a language. As such, a model type specifies how a model written in a given language can be manipulated. All the features of a language, including those specific to its semantics (*e.g.* the methods used to initialize a simulation or invoke an interpreter) are exposed and accessible through its unified interface materialized in a model type. Models are linked to model types by a typing relation [30]. Most importantly, model types are linked one another by subtyping relations [16]. These relations are supported by a model-oriented type system that leverages family polymorphism [13] and structural typing to provide *model polymorphism*, *i.e.* the ability to safely manipulate a model through different interfaces. This mechanism can be understood intuitively by analogy with polymorphism in object-oriented languages: models play the role of objects, languages of classes, and model types of interfaces. Model polymorphism enables the definition of generic tools that can manipulate any model matching the interface on which they are defined, regardless of their concrete language. Doing so, modeling environments are much more resilient to evolution, as new versions of a DSL are likely to preserve its interface, or lead to a new interface that subtypes the previous one. Additionally, DSLs designed in different domains that share some commonalities (such as variants of a FSM language) are likely to match the same interface[1]. One can for example design a generic *flatten* transformation on an abstract FSM model type and reuse it for several variants or versions of FSM languages, provided that they implement the appropriate model type.

[1] Structural dissimilarities between two languages (such as different names for the same concept) may be fixed using adaptation mechanisms that are beyond the scope of this paper.

Besides, model types provide a reasoning layer that allows to reason on language implementations. We use this reasoning layer to define an algebra of operators for language assembly and composition [10]. The `import` operators depicted in Figure 1 enable to import fragments of syntax and semantics to form new languages (`L` and `L'`). Model types are used to check the correctness of the assembly, *e.g.* to assess that a given semantics fragment can be structurally applied to a given syntax fragment. New DSLs can then be manipulated as first-class entities to be specialized (`inherit`), composed (`merge`), or restricted (`slice`). Each of these operators takes both syntax and semantics into account and relies on the aforementioned type system to statically ensure the structural correctness of the produced DSLs. The *merge* operator serves as a language unification mechanism [11] and is inspired by the UML PackageMerge relation [27]. The *slice* operator is inspired by model slicing [4] and consists in extracting a subset of an existing language to be imported in a new one. Finally, the *inherits* operator allows to reuse the definition of one or more super-languages into a new language. In addition to the *merge* operator, the *inherits* operator ensures that the resulting language remains compatible with its super-languages, *i.e.* the tools defined on a super-language can always be reused without adaptation on its sub-language. This set of operators enable language designers to reuse as much as possible other DSLs for the creation of new ones with customization facilities. The associated type system checks the structural correctness of the composed languages and provides the ability to safely manipulate models in different environments.

4. Results and Contribution

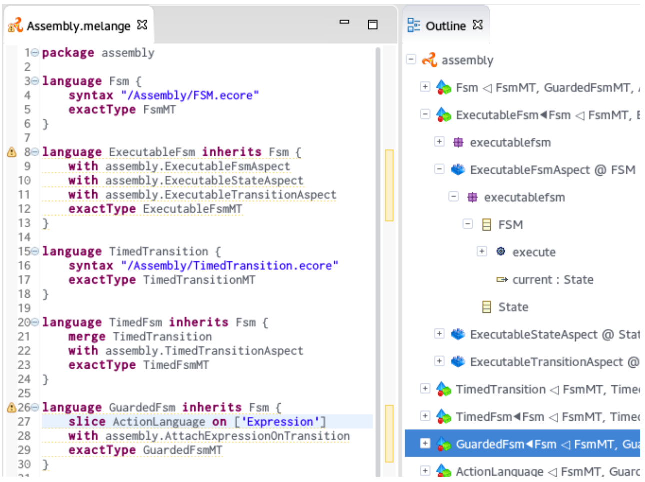
We implemented our approach in the Melange language workbench [10]. Melange consists of a set of Eclipse plug-ins seamlessly integrated with the *de facto* industrial standard Eclipse Modeling Framework [31] and is available under the Eclipse Public License (EPL) on Github [2]. Melange is composed of two main parts:

- A language workbench where language designers can define the syntax and semantics of their own DSLs, use composition operators to assemble legacy DSLs into new ones, implement syntactic and semantic variation points, extend existing DSLs, *etc.*;
- The *MelangeResource*, a mechanism integrated with EMF that provides model polymorphism to *any* EMF-based tool of the Eclipse modeling ecosystem in a *non-intrusive* way.

In the Melange language workbench, language designers use a concise language for designing the syntax and semantics of their DSLs, as illustrated in Figure 2. This language incorporates the composition operators and the model-oriented type system introduced in Section 3. In particular, each DSL

[2] `https://github.com/diverse-project/melange/`

is associated to its structural interface captured in a model type. Based on the subtyping relation automatically inferred between model types, the *MelangeResource* provides model polymorphism and substitutability to the EMF ecosystem. Popular model transformation tools, such as ATL [], QVTo [], or even Java code, can then manipulate the models conforming to the created DSLs in a safe and polymorphic way The *MelangeResource* transparently provides model polymorphism while the transformation tool itself remains agnostic of this mechanism.



```
package assembly

language Fsm {
    syntax "/Assembly/FSM.ecore"
    exactType FsmMT
}

language ExecutableFsm inherits Fsm {
    with assembly.ExecutableFsmAspect
    with assembly.ExecutableStateAspect
    with assembly.ExecutableTransitionAspect
    exactType ExecutableFsmMT
}

language TimedTransition {
    syntax "/Assembly/TimedTransition.ecore"
    exactType TimedTransitionMT
}

language TimedFsm inherits Fsm {
    merge TimedTransition
    with assembly.TimedTransitionAspect
    exactType TimedFsmMT
}

language GuardedFsm inherits Fsm {
    slice ActionLanguage on ['Expression']
    with assembly.AttachExpressionOnTransition
    exactType GuardedFsmMT
}
```

```
assembly
  Fsm ◁ FsmMT, GuardedFsmMT,
  ExecutableFsm◀Fsm ◁ FsmMT, E
    executablefsm
    ExecutableFsmAspect @ FSM
      executablefsm
        FSM
          execute
          current : State
        State
    ExecutableStateAspect @ Stat
    ExecutableTransitionAspect @
  TimedTransition ◁ FsmMT, Time
  TimedFsm◀Fsm ◁ FsmMT, Time
  GuardedFsm◀Fsm ◁ FsmMT, Gua
  ActionLanguage ◁ FsmMT, Guar
```

Figure 2. Modular Definition of a FSM Language in The Melange Language Workbench

Melange was used for the definition of an executable DSL for the design and simulation of Internet of Things (IoT) systems. The resulting IoT language is inspired from both general-purpose executable modeling languages such as fUML and modeling languages dedicated to IoT such as ThingML. Specifically, we designed the IoT language as an assembly of publicly-available DSLs on Github: (i) an IDL language for specifying the structural interface of sensors (ii) Lua for expressing their behavior and (iii) an activity diagram to express concrete scenarios involving different sensors. Taken independently, each of these languages has been defined by different groups of people for specific purposes, unrelated to IoT systems. Combining them in a consistent way, however, leads to a new DSL particularly suited to a new context, *i.e.* the IoT domain. Because most of their syntax and semantics can be reused as is, this drastically reduce the development costs compared to a top-down approach, with no penalty in terms of runtime performance of the resulting language. More information on this case study can be found in [9, 10].

Within the ANR INS project GEMOC [3], we used Melange to define the operational semantics of various DSLs (activity diagrams, timed finite-state machines, Internet of Things languages, *etc.*), and infer the resulting structural interfaces atop which we defined editors and simulators.

Another case study involved the design of a family of statechart languages exposing syntactic variation points (*e.g.* hiearchical, with time constraints) and semantic variation points (*e.g.* run-to-completion or simultaneous events processing policies). While all these languages differ in some ways, they also match the same common interface on top of which we define a set of generic tools for pretty-printing, flattening, and executing statecharts.

Melange was used as part of our submission [5] to the "*model execution*" case study of the *8th Transformation Tool Contest* [4]. This case study consisted in the specification of the operational semantics of a subset of the UML activity diagram language. In this context, Melange was used to extend the original activity diagram metamodel with its operational semantics, infer the corresponding interfaces, and provide substitutability between the original metamodel and the executable one. Our submission earned the "*overall winner prize*" of the model execution case study.

Current investigations of Melange include the definition and management of viewpoints on a large-scale systems engineering language named Capella [5] in collaboration with Thales Group. The goal is to modularly extend the Capella language with task-specific information (*e.g.* for performance or cost analysis) while retaining the compatibility with all the tools of the Capella environment.

Based on the recent collaborations, we are also investigating the extension of Melange to support behavioral semantics expressed in xMOF [23].

References

[1] MOF, 2.0 core final adopted specification, 2004.

[2] C. Atkinson and T. Kühne. Profiles in a strict metamodeling framework. *Science of Comp. Program.*, 44(1):5–22, 2002.

[3] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proc. of ASE'01*, pages 273–280, 2001.

[4] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: Modeling and generating model slicers. *Software and Systems Modeling (SoSyM)*, pages 1–17, 2012.

[5] B. Combemale, J. Deantoni, O. Barais, A. Blouin, E. Bousse, C. Brun, T. Degueule, and D. Vojtisek. A solution to the ttc'15 model execution case using the gemoc studio. In *8th Transformation Tool Contest*. CEUR, 2015.

[6] M. L. Crane and J. Dingel. Uml vs. classical vs. rhapsody statecharts: Not all models are created equal. In *Model Driven Engineering Languages and Systems*, pages 97–112. Springer, 2005.

[7] A. Cuccuru, C. Mraidha, F. Terrier, and S. Gérard. Templatable metamodels for semantic variation points. In *Proc. of ECMDA-FA'07*, pages 68–82, 2007.

[3] `http://gemoc.org/ins/`

[4] `http://www.transformation-tool-contest.eu/2015/`

[5] `https://polarsys.org/capella/`

[8] J. De Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *Proc. of MODELS'10*, pages 16–30, 2010.

[9] T. Degueule, B. Combemale, A. Blouin, and O. Barais. Reusing legacy dsls with melange. In *15th Workshop on Domain-Specific Modeling*, 2015.

[10] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel. Melange: A meta-language for modular and reusable development of dsls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 25–36. ACM, 2015.

[11] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proc. of the Workshop on Language Descriptions, Tools, and Applications*, page 7, 2012.

[12] S. Erdweg, S. Fehrenbach, and K. Ostermann. Evolution of software systems with extensible languages and dsls. *Software, IEEE*, 31(5):68–75, 2014.

[13] E. Ernst. Family polymorphism. In *ECOOP 2001—Object-Oriented Programming*, pages 303–326. Springer, 2001.

[14] J.-M. Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon. *Dagstuhl Reports*, 2004.

[15] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[16] C. Guy, B. Combemale, S. Derrien, J. R. Steel, and J.-M. Jézéquel. On model subtyping. In *Modelling Foundations and Applications*, pages 400–415. Springer, 2012.

[17] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual—. *ACM Sigplan Notices*, 24(11):43–75, 1989.

[18] J.-M. Jézéquel. Model driven design and aspect weaving. *Software & Systems Modeling*, 7(2):209–218, 2008.

[19] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, pages 1–16, 2013.

[20] U. Kastens and W. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994.

[21] L. C. Kats and E. Visser. *The spoofax language workbench: rules for declarative specification of languages and IDEs*, volume 45. ACM, 2010.

[22] H. Krahn, B. Rumpe, and S. Völkel. Monticore: a framework for compositional development of domain specific languages. *JSTT*, 12(5):353–372, 2010.

[23] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. xmof: Executable dsmls based on fuml. In *Software Language Engineering*, pages 56–75. Springer, 2013.

[24] M. Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9):2451 – 2464, 2013.

[25] M. Mernik and V. Zumer. Reusability of formal specifications in programming language description. In *8th Annual Workshop on Software Reuse, WISR8*, pages 1–4, 1997.

[26] P. D. Mosses. The varieties of programming language semantics and their uses. In *Perspectives of System Informatics*, pages 165–190. Springer, 2001.

[27] *Unified Modeling Language 2.0, Infrastructure*. OMG, 2005.

[28] J. Sánchez Cuadrado and J. García Molina. Approaches for model transformation reuse: Factorization and composition. In *Proc. of ICMT'08*, pages 168–182, 2008.

[29] J. a. Saraiva. Component-based programming for higher-order attribute grammars. In *Proc. of GPCE*, pages 268–282, 2002.

[30] J. Steel and J.-M. Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, 2007.

[31] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[32] E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 2015.

[33] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *Proc. of UML'04*, pages 290–304, 2004.

[34] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua, and G. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 95–111. ACM, 2014.

[35] M. Voelter. Language and IDE modularization, extension and composition with MPS. *Generative and Transformational Techniques in Software Engineering*, 2011.

[36] M. Voelter, B. Kolb, and J. Warmer. Projecting a modular future. 2014.

[37] M. Völter and E. Visser. Language extension and composition with language workbenches. In *Proc. of the OOPSLA companion*, pages 301–304. ACM, 2010.

[38] S. Živković and D. Karagiannis. Towards metamodelling-in-the-large: Interface-based composition for modular metamodel development. In *Enterprise, Business-Process and Information Systems Modeling*, pages 413–428. Springer, 2015.