Shape-Diverse DSLs: Languages without Borders (Vision Paper)

Fabien Coulon
Univ. Toulouse, IRIT, Obeo
France

Thomas Degueule
Centrum Wiskunde & Informatica
Netherlands

Tijs van der Storm
Centrum Wiskunde & Informatica, Univ. of Groningen
Netherlands

Benoit Combemale
Univ. Toulouse, IRIT, Inria
France

Abstract

Domain-Specific Languages (DSLs) manifest themselves in remarkably diverse shapes, ranging from internal DSLs embedded as fluent APIs, to external DSLs with dedicated syntax and tool support. Although different shapes have different pros and cons, combining them for a single language is problematic: language designers usually commit to a particular shape early in the design process, and it is hard to reconsider this choice later. In this *new ideas* paper, we envision a language engineering approach enabling (i) language users to manipulate language constructs in the most appropriate shape according to the task at hand, and (ii) language designers to combine the strengths of different technologies for a single DSL. We report on early experiments and lessons learned building Prism, our prototype approach to this problem. We illustrate its applicability in the engineering of a shape-diverse DSL implemented conjointly in Rascal, EMF, and Java. We hope that our initial contribution will raise the awareness of the community and encourage future research.

CCS Concepts • **Software and its engineering** → **Domain specific languages**;

Keywords domain-specific language, shape-diverse dsl

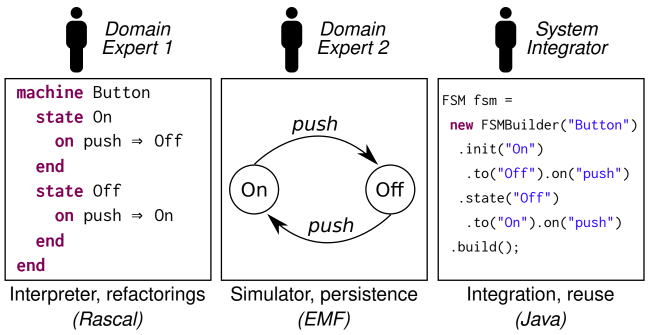
ACM Reference Format:
Fabien Coulon, Thomas Degueule, Tijs van der Storm, and Benoit Combemale. 2018. Shape-Diverse DSLs: Languages without Borders (Vision Paper). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3276604.3276623

1 Introduction & Motivating Example

One of the first steps in designing a new Domain-Specific Language (DSL) is to choose which *language vehicle* (LV) will be used to engineer it. We define a LV as the technological means for implementing a language. This includes language workbenches as well as programming languages and ontology languages, to name a few. The notion of language vehicle is orthogonal to the distinction between technological spaces (e.g., grammarware, modelware [9]); between graphical and textual syntax; between internal, embedded, and external DSLs. For instance, we consider Rascal [8] and Spoofax [6] as two distinct language vehicles within the broader technological space of grammarware and meta-programming; EMF [13] and UML [5] (using Profiles [11]) as two distinct language vehicles within the broader technological space of modelware. LVs usually come with their own meta-languages for expressing the various aspects of a DSL: abstract syntax, concrete syntax, static and execution semantics, tools, etc. As implementation techniques differ radically from one LV to another, this initial design choice commits the development of a DSL in a set direction that can hardly be reconsidered later.

From the language designer's point of view, however, every LV has its own strengths. The ecosystem around EMF excels in the definition of user-friendly editors and persistence frameworks for large models, while the Rascal environment excels in the definition of interpreters and refactoring tools. The benefits of various LVs are also visible from the language users' point of view. While domain experts may prefer to manipulate domain concepts through a dedicated syntax, advanced users (e.g., system integrators) may favor the flexibility of a fluent API in their favorite programming language to manipulate the very same constructs.

Let us consider a simple Finite-State Machine (FSM) language as a motivating example. As depicted in Figure 1, one would like to combine the strengths of multiple LVs to engineer this DSL. Rascal could be used to develop its interpreter, a set of refactoring tools (e.g., state collapsing and minimization), and a textual editor; EMF to develop a graphical animator for debugging FSM models and a persistence layer;


Domain Expert 1
Domain Expert 2
System Integrator
push
On
Off
push
Interpreter, refactorings (Rascal)
Simulator, persistence (EMF)
Integration, reuse (Java)

```
machine Button
  state On
    on push ⇒ Off
  end
  state Off
    on push ⇒ On
  end
end
```

```
FSM fsm =
 new FSMBuilder("Button")
  .init("On")
   .to("Off").on("push")
  .state("Off")
   .to("On").on("push")
 .build();
```

Figure 1. Three incarnations of the same FSM model in three language vehicles: different representations and tools for different users and tasks.

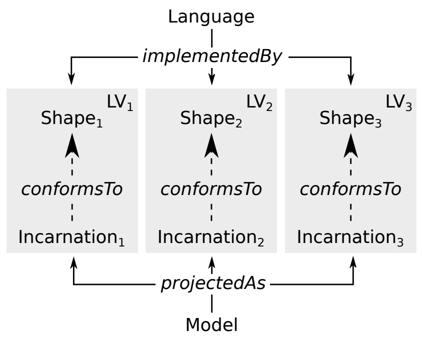


Figure 2. Languages are implemented as shapes in LVs and models are projected as incarnations conforming to the shapes.

Java to offer a fluent API for advanced users who focus on its integration with other system concerns.

Using today's techniques, it is possible to define the same FSM language in these three LVs separately. It is not possible, however, to apply the tools of a given LV on the models or programs created in another LV—for instance, animating a FSM model written in EMF using the Rascal interpreter, or synchronizing a textual FSM model in Rascal with its equivalent incarnation as a Java AST. Achieving this goal requires to *efficiently synchronize the diverse representations of the same model in different LVs*; for instance to let the FSM interpreter written in Rascal update its own representation of an FSM model after each execution step and synchronize it with the representation of the same model in EMF for animation purposes.

In this paper, we envision a language engineering approach enabling (i) language designers to combine tools from multiple LVs to engineer diverse shapes for a single DSL and (ii) language users to manipulate language constructs in the most appropriate shape. We present the notion of shape-diverse DSL in Section 2. We then present our prototype approach, Prism, in Section 3, and discuss our implementation of a shape-diverse FSM language in Section 4. Finally, we discuss open questions and next steps in Section 5.

2 Shape-Diverse DSLs

The cornerstone artifact defining a DSL in any LV is its abstract syntax. The way abstract syntax is expressed differs drastically from one LV to another: GEMOC [4] and Xtext [3] use Ecore metamodels [13], MPS uses *concepts* [16], Rascal [8] uses Algebraic Data Types (ADT), etc. Language embedding techniques, on the other hand, use the constructs of a host language to materialize the constructs of a DSL in the host language itself (e.g., a set of Java classes). Concrete models are then built as instances of the corresponding abstract syntax formalism: Ecore models, ADT values, Java ASTs, etc. The tools defined within a particular LV (an interpreter in Rascal, an editor in EMF) manipulate models in the corresponding formalism (respectively, ADT values and Ecore models). These formalisms radically differ in many ways [7]: object-oriented vs. functional, graphs vs. trees, mutable vs. immutable datatypes, cross-references vs. symbolic names, etc. As LVs are developed by independent groups of people and rely on different underlying theories, it is neither possible nor desirable to establish a common foundation upon which all LVs would agree.

Figure 2 gives an overview of the concept of shape-diverse language and the terminology we use throughout the paper. A shape-diverse language $\mathcal{L}$ (e.g., the FSM language of Figure 1) is a language that is implemented in multiple LVs through multiple shapes $\mathcal{S}_i$. As mentioned earlier, Ecore metamodels, ADT definitions, and Java APIs, along with their associated tooling, are examples of shapes. Similarly, a "conceptual" model m that uses the constructs of $\mathcal{L}$ (e.g., the simple `Button` machine) is projected[1] as an incarnation $\mathcal{I}_i$ conforming to the shape $\mathcal{S}_i$ in a LV: an Ecore model, an ADT value, or a Java AST.

As the same model is incarnated many times, each of its incarnations $\mathcal{I}_i$ must remain synchronized. This synchronization mechanism must ensure three essential properties. First, it must be efficient. This rules out any synchronization mechanism that would require a full traversal or full (de)serialization of the incarnations after every update. Second, it must account for any extra shape-specific information the various LVs have to maintain to function properly, such as layout information in a textual or graphical editor, or runtime state in a simulation environment. The synchronization mechanism must thus isolate the information that relates to the model itself from the information that is specific to a particular shape. Third, the synchronization mechanism must be language-agnostic, meaning it should not have to be re-implemented for every shape-diverse DSL.

[1]The notion of projection here is unrelated to the notion of projectional editing [17] as there is no underlying AST to project from.

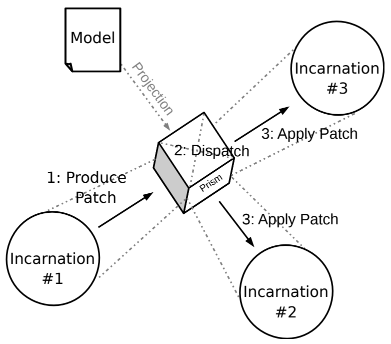


Figure 3. Using Prism to synchronize three incarnations of the same model. Here, a change occurs on Incarnation #1 and the resulting patch is shipped to Incarnation #2 and #3.

3 Synchronizing Incarnations with Prism

Figure 3 depicts our prototype approach to the problem of synchronizing various incarnations of a model, Prism. Prism acts as a communication bus between LVs that remain fully independent. The key idea is that every change occurring on one incarnation is shipped to all other incarnations of the same model in the form of a *patch*. This patch represents the exact set of changes that occurred on one incarnation. It allows synchronizing incarnations online efficiently without requiring serialization or a full traversal of any of the incarnation. Prism keeps track of a matrix that associates every conceptual model to its incarnations in various LVs. When a change occurs on one incarnation, for instance resulting from a user edit or an execution step of an interpreter, the LV hosting this incarnation generates a patch describing the change as a set of CRUD-like operations. In our prototype implementation, the structure of this patch is prescribed by the Rascal ADT shown in Listing 1, largely inspired by *edit scripts* [15]. Essentially, patches consist of a set of operations attached to identities [7] that represent particular objects in the model. To ensure that every LV can apply the operations on the right elements, identities are preserved across LVs and, in our case, they are represented by URIs [2].

Every LV then interprets the patch in its own way to keep its incarnations synchronized. In EMF, for instance, the patch is interpreted as a set of changes that impact a model conforming to an Ecore metamodel, while in Rascal it is interpreted as a set of changes that impact an ADT value.

As mentioned earlier, each LV may want to preserve extra shape-specific information across the patches. A textual editor in Rascal, for instance, needs to keep some of the parsing information to maintain layout whenever patches are applied. So it should be possible to apply the patch while maintaining the extra information specific to a given LV. Intuitively, our approach supposes that all the information that does not directly relate to the constructs of a language is "extra" and therefore should not be part of the patch itself. There might be cases where sharing extra-information from one shape to the other is desirable, for instance to share layout information between two textual editors. We discuss this point further in Section 5.

```
@doc{A patch consists of a sequence of edits}
alias Patch = tuple[Id root, Edits edits];

@doc{Edits are operations attached to object identities}
alias Edits = lrel[Id obj, Edit edit];

data Edit = put(str field, value val)
          | unset(str field)
          | ins(str field, int pos, value val)
          | del(str field, int pos)
          | create(str class)
          | destroy();
```

Listing 1. CRUD-like patch definition in Rascal.

New LVs can be connected to Prism by implementing a simple interface that consists of two operations, namely (i) *produce* which creates a patch materializing the changes on an incarnation and notifies Prism, and (ii) *apply* which receives a patch from Prism and interprets it to update an incarnation, taking into account the specificities of the LV. The way changes are detected in an incarnation and patches are produced is not prescribed by our approach. For instance, our Rascal implementation computes patches from a *diff* operation between two ADT values, while our EMF implementation captures the result of transactions on an Ecore model to produce the patches. The produce and apply operations are implemented once for every LV and do not have to be re-implemented for every language.

A cornerstone artifact in Prism is the dispatch mechanism that routes patches to the appropriate incarnations. When receiving a patch, Prism looks up its internal matrix to determine which other incarnations of the same model should be updated. The patch is then copied and routed accordingly. Our current implementation of the dispatch mechanism is kept simple, and we leave for future work the support of concurrent edits on different incarnations of the same model. This will allow Prism to scale to advanced scenarios that go beyond the scope of this paper, such as collaborative editing.

4 A Shape-Diverse FSM Language

To illustrate Prism, we build a shape-diverse FSM language conjointly in Rascal, EMF, and Java, available on a companion webpage [14]. Figure 4 depicts the implementation of the abstract syntax of this FSM language in the three LVs. The corresponding incarnations are those given in Figure 1.

We use Rascal to define a textual editor and a simple transformation that inserts a new state in a machine. We use EMF to define two graphical editors: a classical tree editor and a domain-specific representation with Sirius. We build the Java API following a simple systematic convention, so as to easily pinpoint which parts of the Java AST have changed (to compute a patch) or need to be updated (to apply a patch).

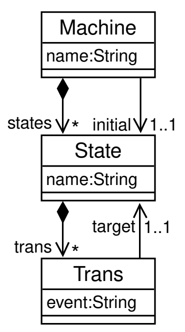
Using Prism, the Rascal and EMF shapes synchronize seamlessly, but we noticed a number of challenges with the

```
data Machine(Id uid) =
 Machine(str name,
  list[State] states,
  Ref[State] initial);

data State(Id uid) =
 State(str name,
  list[Trans] trans);

data Trans(Id uid) =
 Trans(str event,
  Ref[State] target);
```

(a) Rascal ADT



(b) Ecore MM

```
class Fsm {
 Fsm(String name);
 State init(String name);
 State state(String name);
 Fsm end();
}
class State {
 State state(String name);
 Trans tgt(String name);
 Fsm end();
}
class Trans {
 Trans tgt(String name);
 State on(String event);
 Fsm end();
}
```

(c) Java API

Figure 4. Three shapes of an FSM language; the corresponding incarnations are those depicted in Figure 1.

Java API. As the Java API inherits the (domain-agnostic) tooling of Java itself, it lacks the domain knowledge necessary to always generate correct patches. Due to the lack of domain-specific static semantics, a well-formed Java program may indeed produce an ill-formed FSM that cannot be interpreted by the other shapes. Besides, our prototype implementation does not account for complex string manipulation when invoking the API or use of variables. However, we believe that these are purely engineering concerns and that enough effort spent on the Java API shape would provide a flawless experience.

5 Open Questions & Next Steps

We now list in this section some open questions, as well as possible next research steps in this direction.

Closed-world vs. Open-world Implementers of a synchronization mechanism for shape-diverse DSLs may opt for the closed-world or open-world assumption. In the former, one assumes that all LVs are known beforehand, while in the latter new LVs may be connected at any point in time, for instance using our *produce/apply* interface for patches. Although the closed-world assumption eases the definition of a common patch formalism on which all LVs agree, it hampers evolution and adaptability of the communication bus.

Automatic shape generation In our evaluation (Section 4), we handcrafted every shape of the FSM language in Rascal, EMF, and Java. It may, however, be possible to automatically generate shapes of a language, either from a common language definition or from a shape to another. For instance, researchers have studied the generation of fluent APIs from BNF-like grammar definitions [10]. Automatic generation of shapes is not necessary for shape-diverse DSLs, but future research in this area would significantly ease their adoption.

Patch formalism In Prism, we opted for patches in the form of edit scripts [15] and were successfully able to bridge three distinct LVs relying on radically different theories. We cannot conclude however that the information contained in such patches is sufficient for any abstract syntax formalism. In an open world especially, connecting new LVs raises the problem of patch evolution. Also, if extra information (e.g., textual layout) must be shared amongst various LVs, the patch formalism should be adapted accordingly. Patches are nonetheless central to our vision, as most other approaches (e.g., change propagation [12]) assume the existence of an underlying model that is not materialized in our case.

Towards collaborative modeling As mentioned in Section 3, Prism does not account for concurrent edits of different incarnations of the same model. It does not account either for a possible distribution of the shapes and incarnations on the network, or the possibility of conflicts. Nonetheless, we believe that the idea of exchanging patches would be a good fit for advanced scenarios such as collaborative and distributed editing of models by different stakeholders under different shapes. A crucial step towards this direction would be to improve the dispatch mechanism accordingly.

Challenges of internal DSLs We encountered a number of challenges when engineering the Java shape of our DSL (Section 4). These are mainly because domain-specific static semantics is lost when manipulating Java ASTs using domain-agnostic Java tooling. Besides, it may be hard to statically analyze the Java code manipulating models to account for reflexivity, string manipulation, or use of variables. Future work must investigate what the limits imposed by internal DSLs in this context are, especially regarding the absence of domain-specific static semantics.

Towards metamorphic DSLs We view this initial contribution as a first step towards *metamorphic DSLs* [1]. Beyond the ideas presented in this paper, the notion of metamorphic DSL envisions self-adaptable languages that automatically adapt their shapes and the associated IDE according to a particular usage or task. How self-adaptability of languages could be brought to life remains an open question.

In this paper, we have stressed the importance of shape-diverse DSLs and have proposed a first prototype approach, Prism. We hope that our initial contribution will raise the awareness of the community regarding these notions and that the challenges we identify encourage future research.

Acknowledgments

This work is partially supported by the associate team ALE (http://gemoc.org/ale/), the bilateral collaboration Kaulua (http://gemoc.org/kaulua/) between Inria & Obeo, and the EU's Horizon 2020 Project No. 732223 CROSSMINER.

References

[1] Mathieu Acher, Benoit Combemale, and Philippe Collet. 2014. Metamorphic Domain-Specific Languages: A Journey into the Shapes of a Language. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. 243–253.

[2] Tim Berners-Lee, Roy Fielding, and Larry Masinter. 2004. *Uniform resource identifier (URI): Generic syntax.* Technical Report.

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

[4] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution framework of the GEMOC Studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE'16)*. 84–89.

[5] Martin Fowler. 2004. *UML distilled: a brief guide to the standard object modeling language.* Addison-Wesley Professional.

[6] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*. 444–463.

[7] Paul Klint and Tijs van der Storm. 2016. Model Transformation with Immutable Data. In *Proceedings of the 9th International Conference on Theory and Practice of Model Transformations (ICMT'16)*. 19–35.

[8] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. EASY Meta-programming with Rascal. In *Generative and Transformational Techniques in Software Engineering III - International Summer School (GTTSE'09)*. 222–289.

[9] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. 2002. Technological Spaces: An Initial Appraisal. (2002).

[10] Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. 2017. Silverchain: a fluent API generator. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17)*. 199–211.

[11] Bran Selic. 2007. A systematic approach to domain-specific language design using UML. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*. 2–9.

[12] Oszkár Semeráth, Csaba Debreceni, Ákos Horváth, and Dániel Varró. 2016. Change Propagation of View Models by Logic Synthesis using SAT solvers. In *Proceedings of the 5th International Workshop on Bidirectional Transformations (BX'16)*. 40–44.

[13] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework.* Pearson Education.

[14] ALE Team. 2018. Prism. https://github.com/fcoulon/prism/.

[15] Riemer van Rozen and Tijs van der Storm. 2017. Toward live domain-specific languages: From text differencing to adapting models at run time. *Software & Systems Modeling* (Aug 2017).

[16] Markus Voelter. 2014. *Generic tools, specific languages.*

[17] Markus Voelter, Jos Warmer, and Bernd Kolb. 2015. Projecting a modular future. *IEEE Software* 32, 5 (2015), 46–52.