On Leveraging Executable Language Engineering for Domain-Specific Transformation Languages

(Position Paper)

Erwan Bousse
TU Wien
Vienna, Austria
bousse@big.tuwien.ac.at

Manuel Wimmer
TU Wien
Vienna, Austria
wimmer@big.tuwien.ac.at

Wieland Schwinger
JKU Linz
Linz, Austria
wieland.schwinger@cis.jku.at

Elisabeth Kapsammer
JKU Linz
Linz, Austria
elisabeth@cis.jku.at

***Abstract*—An increasing number of *domain-specific transformation languages* (DSTLs) are used to define model transformations in specific contexts. This shift led to many approaches to define new DSTLs and associated tools, either through frameworks or complete generative approachs. In parallel, the fields of language engineering and model execution have seen the development of many approaches to efficiently define new *executable domain-specific modeling languages* (xDSMLs), and to provide tools (*e.g.*, editor, debugger) for any newly defined xDSML. In this position paper, we propose to study how the engineering of DSTLs could benefit from state-of-the-art xDSML engineering approaches. We first demonstrate why a DSTL is an xDSML with specific characteristics. We then give a selection of research directions to apply xDSML engineering approaches on DSTLs.**

***Index Terms*—model transformation, domain-specific transformation languages, language engineering, model execution**

I. Introduction

Model transformations are at the core of all model driven engineering activities [15]. They are defined using *model transformation languages*, which provide concepts to express model modifications or code generation. Because model transformations are complex artifacts, an increasing number of *domain-specific transformation languages* (DSTLs) [12] are used to define them. A DSTL can be specific to a technical concern of model transformation (*e.g.*, Epsilon task-specific DSTLs [11]), or to a specific context (*e.g.*, mapping of abstract and concrete syntaxes [9]), or even to a specific set of input and output metamodels (especially in generative approaches, *e.g.*, [10]). The growing usage of DSTLs led to many methods to efficiently define new DSTLs and associated tools, such as frameworks [5], [16] or generative approaches [7], [10]. A good illustration of this trend is the Tool Transformation Contest 2016 (TTC'16), which confronted transformation tools on the implementation of an engine for a dataflow-based DSTL[1].

In parallel, the fields of language engineering and model execution have seen the development of many approaches to efficiently define new *executable domain-specific modeling languages* (xDSMLs) [3], [6], [13], which are DSMLs supported by execution semantics. Such approaches can automatically provide static (*e.g.*, editor) or dynamic (*e.g.*, debugger) tools for any newly defined xDSML. For example, our recent work focused on the generation of dedicated execution trace management facilities of an xDSML [4], and on the usage of such facilities for model omniscient debugging [2].

In this context, we make the following observation: if we consider a model transformation as a model [1], [12], we can consider a DSTL as a specific sort of xDSML. Therefore, all research results in the field of xDSML engineering can be applied to DSTLs, in order to improve or supplement existing DSTL engineering approaches. For instance, a language workbench such as the GEMOC Studio [3] can be used to implement a DSTL and to automatically provide it with a debugger. Furthermore, such research results can presumably be adapted or extended for the specific case of DSTLs, given a study of what characterizes DSTLs as specific xDSMLs. For instance, in the case of DSTLs, the parameters given to the executed model (*i.e.*, the model transformation) commonly always comprise an input model and input/output metamodels.

In this position paper, we propose to study how the engineering of DSTLs could benefit from state-of-the-art xDSML engineering approaches. In Section II, we present why a DSTL is a specific sort of xDSML. In Section III, we present some specific characteristics of DSTLs, and we identify a selection of research directions to apply xDSML engineering approaches on DSTLs. Finally, we conclude in Section IV.

II. From xDSML to DSTL engineering

In this section, we first briefly explain what is an xDSML, we then demonstrate how a DSTL is as a specific sort of xDSML, and finally we present a category of DSTLs that are defined with fixed input and output metamodels.

A. Executable Domain-Specific Modeling Language (xDSML)

An xDSML is a domain-specific modeling language supported by execution semantics [6]. There are two main approaches to define execution semantics: translational (*i.e.*, compilation or code generation) and operational (*i.e.*, interpretation). In this paper we focus on operational semantics, only.

Figure 1 shows an xDSML and its usage to execute a model. The central part of an xDSML is the *abstract syntax*, also called the domain model. It is completed by the *parameter metamodel* that defines the possible parameters for an execution. The *executable model* and the *parameter model* conform to these

[1] https://github.com/bluezio/ttc2016-live

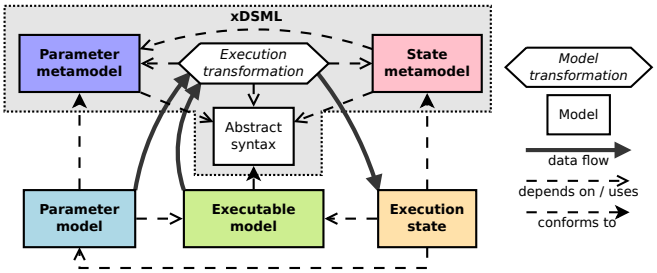

Figure 1. Representation of an xDSML and its usage.

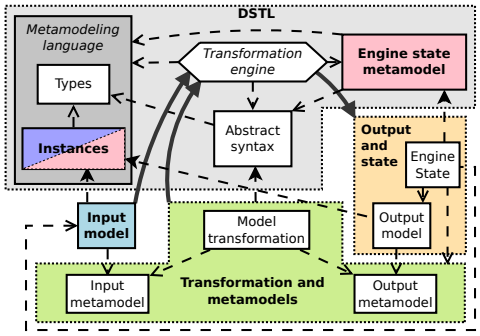

Figure 2. Representation of a DSTL as an xDSML, and its usage for a model-to-model transformation. Some arrows are not shown for readability (*e.g.*, the input and output metamodels conform to *Types*).

metamodels. Next, the operational semantics of the xDSML comprise two parts. First, the *state metamodel* defines what is the dynamic *execution state* of an executable model. Second, the *execution transformation* is a model transformation (often in-place) that defines how the state changes during execution.

B. Domain-Specific Transformation Language (DSTL)

A DSTL is domain-specific language that can be used to express model transformations. If we consider a model transformation as a model [1], [12], we can consider a DSTL as a specific sort of xDSML, and a model transformation as an executable model. In the following, we present the common case of DSTLs that can process models conforming to any given input/output metamodels (*e.g.*, Epsilon DSTLs [11]).

Figure 2 shows a refinement of Figure 1 to represent a DSTL in the case of model-to-model transformations. While it was implicit in Figure 1, we explicitly must show here the considered *metamodeling language* (*e.g.*, EMF Ecore) that is required by a DSTL to generically handle any input or output metamodels and models. It is shown at the top left, and is composed of two parts: *Types* (*e.g.*, EClass, EReference) and *Instances*[2] (*e.g.*, EObject). The *Types* part is used by the abstract syntax, since a model transformation references classes of the considered input and output metamodels. The *Instances* part is used both as the parameter metamodel,

[2]Relationships are presented from the perspective of the DSTL, meaning that the input/output models are considered as generic models conforming to the *Instances* metamodel, *i.e.*, we present the *linguistic* relationships. Yet, if we consider *ontological* relationships, the input/output models do conform to the input/output metamodels, which is only shown here as dependency links.

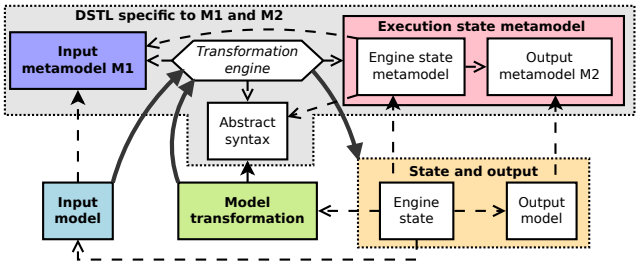

Figure 3. Representation of a DSTL with fixed input and output metamodels M1 and M2, and its usage for a model-to-model transformation.

so that any input model may be taken as a parameter, and for the state metamodel, so that the output model can be constructed. The executable model not only consists of the model transformation, but also of its *input* and *output metamodels*. The execution transformation is commonly called a *transformation engine*, which is responsible for applying the rules of the executed *model transformation*. More precisely, such engine is a transformation that generically reads both the input transformation and the input model (*e.g.*, using `eGet` in EMF) and produces the output model (*e.g.*, using `eSet` and factories in EMF). The engine is arbitrarily complex and completely dependent on the paradigms and features provided by the implemented DSTL (*e.g.*, pattern matching). Finally, at runtime, the execution state comprises both the *engine state* (*e.g.*, current rule being executed), and the *output model* being produced. Note that if the input and output models are distinct, we have an out-place model transformation, and if they are the same single model, we have an in-place model transformation. The latter case is possible since the input and output models conform to the same *Instances* metamodel.

C. DSTL with fixed input and output metamodels

While a DSTL can commonly consider any input and output metamodels, it is also possible to define a DSTL for fixed input and output metamodels. This is especially the case for generative approaches that produce the DSTL specific to a given DSML [10], in order to facilitate the definition of transformations for this DSML. While such choice implies a loss of genericity, it makes possible to provide syntactic constructs very similar to the input and output metamodels (*e.g.*, by deriving a pattern language from the input metamodel), and ease the definition of the engine, among other advantages.

Figure 3 shows a refinement of Figure 1 when considering a DSTL that is specific to an input metamodel M1 and to an output metamodel M2. In this scenario, the DSTL does not have to explicitly rely on a metamodeling language, since M1 is directly used as a parameter metamodel and M2 is part of the execution state metamodel. Consequently, the only parameter given to the engine is an input model conforming to M1.

III. Observations and Research Directions

While being xDSMLs, DSTLs have a number of specific characteristics. For example, when a DSTL supports any input or output metamodels (see Section II-B), the executed model

is heterogeneous and complex since it is composed of both input/output metamodels in addition to the transformation. In other words, the abstract syntax of such DSTL has to import all the complexity of a complete metamodeling language. In addition, the parameter metamodel is a subset of the same metamodeling language, meaning that the parameter can be any kind of arbitrarily complex model. The execution state is likewise rather complex, since it includes the complete output model under construction. Lastly, the transformation engine of a DSTL can also be elaborated to manage diverse concerns (*e.g.*, pattern matching, order of execution of rules) [8], [16].

While there is already a wide range of efficient approaches to engineer DSTLs [7], [10], the aforementioned complexity is a strong motivation to further improve these approaches. Research in language engineering and model execution include many approaches to tackle the complexity of engineering an xDSML. To potentially benefit from such results in the context of DSTLs, we propose a selection of research directions:

a) xDSML design pattern: Understanding a DSTL, especially its transformation engine [14], helps verifying that both a DSTL and conforming transformations are correct. The *xDSML design pattern* [6] was proposed to capitalize good engineering practices for xDSMLs with a strong focus on the separation of concerns, which helps understandability. Such design pattern could be used or adapted for engineering DSTLs. Furthermore, defining xDSMLs in a systematic manner gives the possibility to analyze them with automatic procedures.

b) xDSML generic and generative approaches: Generic and generative approaches have been proposed to support any xDSML defined in a systematic manner (see previous point). For instance, we proposed an approach to generate the domain-specific trace metamodel of an xDSML [4], and a generic omniscient debugging approach for model execution [2]. Such approaches could be applied or adapted for DSTLs.

c) Extend xDSML workbenches for DSTLs: Some existing work aim at capitalizing knowledge and practices for the engineering of transformation languages. For instance, the T-Core framework [16] provides a wide range of primitives to greatly facilitate the definition of transformation engines. Such frameworks or libraries could be used in xDSML language workbenches to help defining operational semantics of DSTLs.

d) Use DSTLs as case studies for model execution: Due to their peculiarities, DSTLs represent interesting cases of xDSMLs. Following the trend of the TTC'16, DSTLs could be used as case studies for model execution approaches.

e) Efficiency of transformation engines: Because the semantics of DSTLs tend to be complex, implementing *efficient* transformation engines is a difficult task. For example, providing a query mechanism requires efficient ways to explore the input model, and to store or cache intermediate results. While there are already common ways to improve efficiency, such as using traceability maps in operational semantics or choosing an efficient target language in translational semantics, specific techniques could be developed or studied.

As a starting point for exploring these directions, we have implemented an open-source toy DSTL called MiniTL[3] using the GEMOC Studio language workbench.

IV. Conclusion

We showed that DSTLs are a specific sort of xDSMLs, and we proposed research directions to study how xDSML engineering can be leveraged to improve or supplement DSTL engineering. We plan to explore these directions as part of the *TETRA Box* research project to improve the testing of model transformations, and to study whether DSTLs are the "killer" case studies for xDSMLs workbenches.

Acknowledgment

This work has been funded by the Austrian Science Fund (FWF): P 28519-N31, by the Christian Doppler Forschungsgesellschaft CDL-Flex and the BMWFW (Austria).

References

[1] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model Transformations? Transformation Models! In *Int. Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2006.

[2] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In *Int. Conference on Software Language Engineering (SLE)*, 2015.

[3] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *Software Language Engineering (SLE)*, 2016.

[4] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *European Conference on Modeling Foundations and Applications (ECMFA)*, 2015.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72:52–70, 2008.

[6] B. Combemale, X. Crégut, and M. Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. *Asia-Pacific Software Engineering Conference (APSEC)*, 2012.

[7] J. S. Cuadrado, E. Guerra, and J. De Lara. Towards the Systematic Construction of Domain-Specific Transformation Languages. In *European Conference on Modelling Foundations and Applications (ECFMA)*, 2014.

[8] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 2006.

[9] A. S.-B. Herrera, E. D. Willink, and R. F. Paige. A Domain Specific Transformation Language to Bridge Concrete and Abstract Syntax. In *Int. Conference on Model Transformation (ICMT)*, 2016.

[10] K. Hölldobler, B. Rumpe, and I. Weisemöller. Systematically deriving domain-specific transformation languages. In *Model Driven Engineering Languages and Systems (MODELS)*, 2015.

[11] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The Epsilon Transformation Language. In *Int. Conference on Model Transformation (ICMT)*, 2008.

[12] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. Explicit Transformation Modeling. In *Int. Workshop on Multi-Paradigm Modeling (MPM)*, 2010.

[13] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. xMOF: Executable DSMLs based on fUML. In *Int. Conference on Software Language Engineering (SLE)*, 2013.

[14] J. T. Saxon, B. Bordbar, and D. H. Akehurst. Opening the Black-Box of Model Transformation. In *European Conference on Modeling Foundations and Applications (ECMFA)*, 2015.

[15] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5), 2003.

[16] E. Syriani, H. Vangheluwe, and B. LaShomb. T-Core: a framework for custom-built model transformation engines. *Software and Systems Modeling (SoSyM)*, 14(3), 2013.

[3] https://github.com/tetrabox/minitl