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 approaches. 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 xDSL with specific characteristics. We then give a selection of research directions to apply xDSML engineering approaches on DSTLs.

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, which confronted transformation tools on the implementation of an engine for a dataflow-based DSTL1.

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 (xDSLs) [3], [6], [13], which are DSLs supported by execution semantics. Such approaches can automatically provide static (e.g., editor) or dynamic (e.g., debugger) tools for any newly defined xDSL. For example, our recent work focused on the generation of dedicated execution trace management facilities of an xDSL [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 xDSL. Therefore, all research results in the field of xDSL 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 xDSLs. 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 xDSL engineering approaches. In Section II, we present why a DSTL is a specific sort of xDSL. In Section III, we present some specific characteristics of DSTLs, and we identify a selection of research directions to apply xDSL engineering approaches on DSTLs. Finally, we conclude in Section IV.

II. FROM XDSL TO DSMTL ENGINEERING

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

A. Executable Domain-Specific Modeling Language (xDSL)

An xDSL 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). We only focus in this paper on operational semantics.

Figure 1 shows an xDSL and its usage to execute a model. The central part of an xDSL is the abstract syntax, also called the domain model. It is completed by the parameter metamodel

1https://github.com/bluezio/ttc2016-live
that defines the possible parameters for an execution. The executable model and the parameter model conform to these 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. We present here the common case of DSTLs that can process models conforming to any given input/output metamodels (e.g., Epsilon DSTLs [11]), and we present after the case of DSTL with fixed input/output metamodels.

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 must here show explicitly 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, 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 DSL [10], in order to facilitate the definition of transformations for this DSL. 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 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.

---

\(^2\)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.
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 have 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 elaborate a 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 Tool Transformation Contest 2016 (TTC’16), DSTLs could be more widely used as case studies for experimenting 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.

IV. CONCLUSION

We showed that DSTLs are a specific sort of xDSMLs, and we proposed some research directions to study how existing xDSML engineering methods can be leveraged to improve or supplement DSTLs engineering approaches. For future work, we plan to investigate these directions as part of the TeTraBox research project, to improve the dynamic verification and validation of model transformations, and to study whether DSTLs are the “killer” case studies for xDSML workbenches.

ACKNOWLEDGMENT

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

REFERENCES


