Towards xMOF: Executable DSMLs Based on fUML [*]

Tanja Mayerhofer
Business Informatics Group, Vienna University of Technology, Austria
mayerhofer@big.tuwien.ac.at

Philip Langer
Business Informatics Group, Vienna University of Technology, Austria
langer@big.tuwien.ac.at

Manuel Wimmer
Software Engineering Group, Universidad de Málaga, Spain
mw@lcc.uma.es

Abstract

When defining a domain-specific modeling language (DSML), the two key components that have to be specified are its syntax and semantics. For specifying a modeling language's abstract syntax, metamodels are the standard means. MOF provides a standardized, well established, and widely accepted metamodeling language enabling the definition of metamodels and the generation of accompanying modeling facilities. However, no such standard means exist for specifying the behavioral semantics of a DSML. This hampers the efficient development of model execution facilities, such as debugging, simulation, and verification. To overcome this limitation, we propose to integrate fUML with MOF to enable the specification of the behavioral semantics for DSMLs in terms of fUML activities. We discuss alternatives how this integration can be achieved and show by-example how to specify the semantics of a DSML using fUML. To reuse existing runtime infrastructures, we further demonstrate the usage of external libraries in fUML-based specifications.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques

Keywords model execution, domain-specific modeling languages, semantics, foundational UML

1. Introduction

The success of model-driven engineering (MDE) depends significantly on the availability of adequate means for defining domain-specific modeling languages (DSMLs). The two key components that constitute a DSML are its *syntax* and *semantics*. For defining the *abstract syntax* of a DSML in terms of a metamodel, the OMG standard MOF not only provides a well-established and commonly accepted metamodeling language, but also fostered the emergence of a variety of tools for deriving *modeling facilities from a metamodel* (semi-)automatically, such as modeling editors, model validators, and generic components for model serialization, comparison, and transformations.

Unfortunately, for defining the *behavioral semantics* of a DSML, no standard way has been established yet. In practice, models are usually executed using a code generator or a model interpreter specified with a general purpose programming language (GPL). Although this enables exploiting the full power of programming languages, the generated code or the model interpreter constitute only an *implementation* of the behavioral semantics rather than an explicit *specification* which violates the main MDE principle "everything is a model" [1]. The consequence is that the emergence of potential techniques building upon an explicit behavioral semantics to derive model debuggers, simulators, and verification tools (semi-)automatically is drastically hindered [2].

To overcome this limitation, we stress the need for a standardized and model-based way of specifying the behavioral semantics of a DSML to facilitate the same benefits as MOF granted for specifying the abstract syntax. Therefore, we propose the usage of fUML[1] as behavioral semantics specification language. fUML is standardized by the OMG and defines the semantics of a key subset of the UML 2.3 metamodel by specifying a virtual machine for executing models compliant to this subset. In particular, we argue to use fUML for extending the DSML's metamodel in terms of fUML activities that describe how a model is executed. Existing research has already investigated such a usage of action languages similar to fUML with the result that such semantics specifications are sufficient for this purpose and comprehensible by language designers [13].

However, to establish fUML as a language for specifying the behavioral semantics of DSMLs and leverage the full potential of having a formal semantics specification, the following challenges have to be addressed. First, the current language for specifying the abstract syntax of a DSML is MOF and

* This research has been partly funded by our industry partner LieberLieber Software GmbH.

DSM'12, October 22, 2012, Tucson, Arizona, USA.

[1] `http://www.omg.org/spec/FUML`

not UML; fUML, however, is a subset of UML. Therefore, fUML has to be first integrated with MOF before it can be used for MOF-based metamodels. Second, it is often not feasible to model everything down to the very last detail using plain fUML. Existing third-party libraries providing, e.g., complex mathematical calculations or control of external resources, may be required. Hence, the usage of external libraries has to be facilitated in the semantics specification of DSMLs. However, it is currently not possible to use external libraries from within the fUML virtual machine.

In this paper, we present ongoing research towards using fUML as a standardized way for specifying the behavioral semantics of DSMLs. Therefore, we discuss important requirements for such an approach, show alternatives how fUML can be integrated with MOF, demonstrate how it can be used to specify the semantics of an example DSML, and present an approach that enables the usage of external libraries in fUML-based specifications without extending the fUML virtual machine or loosing platform-independence.

The remainder of this paper is structured as follows. Section 2 summarizes related work regarding the semantics specification of DSMLs. Requirements that shall be fulfilled by a semantics specification approach are discussed in Section 3. Section 4 introduces the proposed semantics approach for defining executable DSMLs using fUML. In Section 5, we present how the fUML-based semantics specification can be extended by incorporating external libraries and conclude in Section 6 with an outlook on future work.

2. Related Work

The need for executable models stimulated intensive research on how to define the behavioral semantics of modeling languages. Consequently, various approaches have been proposed in the past. In the following, we give a brief overview of these approaches.

Denotational and *translational semantics* approaches map the constructs of a modeling language to constructs of another language already having a formal semantics. This has the advantage that existing tools for executing and analyzing the target language can be used for executing the source language. The drawback, however, is the fact that the semantics of a language is defined by the mapping into the target language leading to an additional level of indirection. The definition of the mapping is a complex task and requires a deep knowledge about the target language. Furthermore, the results of the execution are only available in the target language. Thus, the execution results have to be mapped back to the source language. One example for a translational semantics approach is the work of Chen *et al.* [3] who use the Abstract State Machine formalism as target language. Another example is Rivera *et al.* [12] who use Maude for formalizing the behavioral semantics of DSMLs.

Compared to denotational and translational semantics, the *operational semantics* approach is more light-weight, but sufficient for executing models directly. One way for defining an operational semantics is to introduce executability concerns by defining graph transformation rules operating on metamodel instances as proposed by Engels *et al.* [6]. Another possibility is to follow an object-oriented approach by specifying the behavior of operations defined for the metaclasses of a modeling language using a dedicated action language. A plethora of action languages has been proposed including the application of existing GPLs: Kermeta [11], Smalltalk [5], xCore [4], EOL [7], and the approach proposed by Scheidgen and Fischer [13] to name just a few. Our approach follows the same spirit, but instead of introducing yet another action language, we employ fUML. Thus, we use a standardized and UML 2 complied action language which should act as a stimulus towards the establishment of a common action language for metamodeling.

Recently, Lai and Carpenter [9] also proposed the usage of fUML for specifying the operational semantics of DSMLs. However, they focus on the static verification of fUML models to identify structural flaws such as unused or empty models. The authors neither discuss the possible strategies for using fUML as action language on the metamodeling level, nor consider the dynamic analysis of fUML models. In contrast, the aim of our work is to enable the specification of executable DSMLs by providing a framework that allows to (semi-)automatically generate execution facilities, such as model debugging or testing environments, and the integration with existing execution frameworks using APIs.

3. Requirements

In the following we describe important requirements that shall be fulfilled by semantics specification languages and facilities to utilize the full potential of having a formal semantics specification of a DSML.

Standardization One very important requirement is that the semantics specification shall be based on standardized technologies. This not only enables interoperability and vendor-independence, but the usage of well-established standard modeling technologies enables an eased application of the semantics specification approach, as language designers are already accustomed to apply them.

We will see in Section 4 that because our semantics specification approach is based on MOF and fUML, only technologies standardized by the OMG are used for specifying the abstract syntax as well as the semantics of a DSML. However, currently fUML is neither integrated with MOF, nor is its usage as semantics specification language standardized.

Extensibility When specifying the semantics of a DSML, there might be the need to use external libraries. Specifying for instance complex mathematical calculations by means of an action language such as fUML in the course of the semantics definition of a DSML is just out of scope of the language specification process (if the language is not about complex mathematical calculations). Therefore, a semantics specification language needs to provide means for integrating other

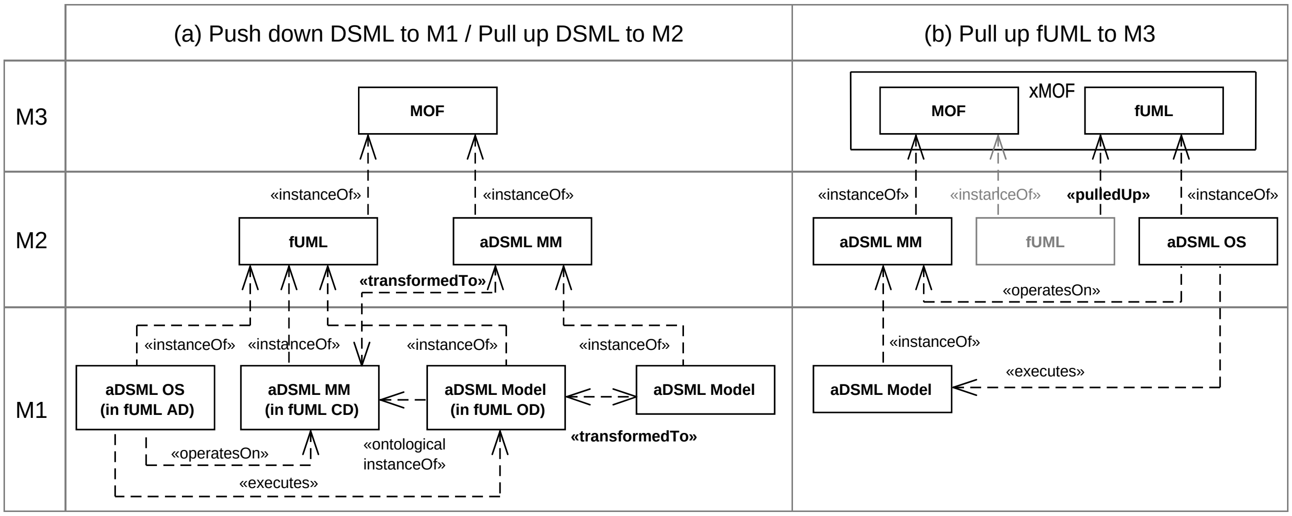

Figure 1. Two strategies for using fUML as semantics specification language

languages in terms of libraries, thus hiding implementation details outside the problem domain of the DSML.
In Section 5, we present a possibility to integrate external libraries in fUML-based semantics specifications.

Reusability As we will see in Section 4, specifying the semantics of a DSML from scratch remains a complex task. Therefore, means for reusing semantics specifications is highly desirable, as this would ease the task of specifying the semantics tremendously. We envision the definition of what we call "kernel semantics" that express reoccuring patterns in behavioral semantics specifications (cf. [2] for a similar approach using so-called "semantic units"). If we for instance consider the various behavioral diagrams provided by the UML, we can identify different patterns of behavioral semantics, such as control flow and data flow used in activities, and triggers and events driving the execution of state machines. Having the formal specifications of such kernel semantics at hand, we could use them to specify the behavioral semantics of a DSML by composing the needed semantics patterns. Another usage scenario would be to provide means for specializing existing semantics specifications. This could be useful when introducing semantic variation points into a language or when using a profile mechanism. However, providing the means for specifying and reusing kernel semantics is subject to future work.

Automation Today, model execution is often realized either by generating code out of models or by implementing a model interpreter using a GPL. In both cases, the actual semantics of the modeling language is only implicitly given. In the case of code generation, the semantics is hidden within the generation templates; when using a model interpreter, the interpreter's implementation actually defines the modeling language's semantics. Thus, using these approaches, model execution facilities, such as debuggers or verifiers, have to be built from scratch for every DSML, which entails high development efforts. Having an explicit formal semantics specification for a DSML can enable us to automatically generate model execution tools, such as debuggers, simulators, verifiers, and testing environments [2]. However, this also belongs to future work.

4. Specifying semantics with fUML

In the operational semantics approach, executability is introduced into the abstract syntax of a DSML using an action language. Following object-orientation, this is done by specifying the bodies of metaclass operations using the chosen action language, defining how models are executed.

4.1 xMOF: Integrating fUML with MOF

We propose an operational semantics approach for defining executable DSMLs based on the new OMG standard fUML. fUML defines a key subset of UML 2.3 and specifies a virtual machine for executing compliant models. For modeling structural aspects of a system, fUML contains a subset of the *Classes::Kernel* package of UML. For modeling behavior, a subset of the packages *CommonBehaviors*, *Actions*, and *Activities* is included in fUML.
However, to establish fUML as a standardized specification language for defining the operational semantics of DSMLs, it has to be integrated with MOF, as MOF is the standardized means for specifying the abstract syntax of DSMLs, and not UML. As fUML uses the UML package *Classes::Kernel* for defining the structural part of a model, which is also merged into MOF for enabling the specification of the abstract syntax of a DSML, the structural part of the fUML metamodel complies with MOF. Therefore, we propose the usage of the behavioral part of fUML for defining the operational semantics of a DSML. However, when considering the MOF metamodeling stack [8], fUML models are—such as UML class and activity diagrams—situated on level M1, whereas the DSML specification is located on level M2. To overcome this level mismatch, we identified the following two strate-

gies, enabling the usage of fUML as semantics specification language for DSMLs, which are depicted in Figure 1.

(a) Push down DSML to M1 / Pull up DSML to M2 The first strategy is to apply a model-to-model transformation to generate a model on level M1 for a specified metamodel on level M2 in case a metamodel is already available for the DSML (*push down DSML to M1*). The generated model denoted as *aDSML MM (in fUML CD)* in column (a) of Figure 1 is created by mapping the elements of the DSML metamodel (*aDSML MM*) compliant to MOF to elements of the fUML metamodel. As fUML uses the UML package *Classes::Kernel* to represent the structural part of a model and this UML package is also used in MOF for specifying metamodels, this transformation works straight-forward. With this transformation it is possible to define fUML activities, specifying the operational semantics of the DSML (*aDSML OS (in fUML AD)*). In order to execute a DSML model (*aDSML Model*) it has to be transformed into a fUML compliant representation of a corresponding object diagram (*aDSML Model (in fUML OD)*) representing ontological instances (cf. [8]) of the fUML classes which define the metaclasses of the DSML (*aDSML MM (in fUML CD)*).
If no metamodel is available in the first place, one may start on the M1 level by purely using fUML and generate a metamodel of a DSML for the level M2 afterwards (*pull up DSML to M2*). This approach has been used in [9].

(b) Pull up fUML to M3 A second strategy is to pull up fUML from the metamodel level M2 to the meta-metamodel level M3 by integrating it with MOF. This approach is depicted in column (b) of Figure 1. Using this approach, the abstract syntax of a model in form of a metamodel denoted as *aDSML MM*, as well as the operational semantics *aDSML OS*, can be specified on the metamodel level M2 by the means of this integrated meta-modeling language. The abstract syntax can be expressed using the modeling concepts provided by the package *Classes::Kernel* which is available in MOF as well as in fUML. The operational semantics can be specified using activities compliant to fUML, enabling the execution of DSML models (*aDSML Model*).

The first strategy, which enables the usage of fUML as semantics specification language by transforming DSML models between MOF and UML utilizing model transformations, has the advantage that present modeling tools already offer the necessary facilities to implement it. However, applying this approach also has a major drawback: on the one hand, the UML modeling environments have to be used for defining the operational semantics, as well as for executing, analyzing, and debugging models, and on the other hand, models are normally defined and manipulated in metamodeling environments. Thus, similar drawbacks arise as mentioned in Section 2 for translational semantics approaches.
Therefore, we advocate the second strategy, i.e., pulling up fUML to the meta-metamodel level M3 by integrating it with

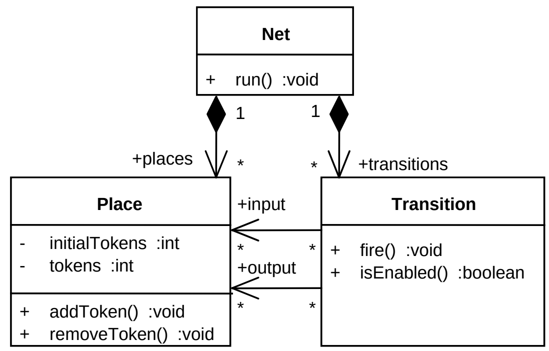

Figure 2. Metamodel of Petri Net DSML

MOF into a framework we call xMOF (eXecutable MOF). xMOF merges MOF and fUML resulting in a metamodeling language capable of specifying the abstract syntax of a DSML using MOF constructs, as well as the operational semantics of a DSML by the means of fUML activities.

4.2 xMOF: An Example

We demonstrate our approach by specifying the operational semantics of Petri Nets. Figure 2 depicts the metamodel of our Petri Net DSML. A `Net` consists of `Places` and `Transitions`. `Places` hold a particular amount of `initialTokens` and `Transitions` reference the `Places` providing `input` and `output`.
For defining the operational semantics of a DSML, we introduced operations which specify how models of the Petri Net DSML are executed. This is specified by defining the behavior of each operation using a fUML activity. Additionally we added the attribute `tokens` to the metaclass `Place` which stores the amount of tokens held by a place at a given point in time during the execution of a Petri Net model.
Figure 3 depicts the fUML activities which specify the behavior of each operation defined for the metaclasses. These activities altogether completely specify the operational semantics of our Petri Net DSML. The `run()` operation of the metaclass `Net` is the main operation controlling the execution of a `Net`. It repeatedly determines a list of enabled `Transitions`, i.e., `Transitions` where the operation `isEnabled()` returns `true`, and calls `fire()` for the first `Transition` in this list. The operation `isEnabled()` returns `true`, if all input `Places` of a `Transition` hold at least one token. This information is represented by the `tokens` attribute of the `Place` representing an input `Place`. More precisely, the operation `isEnabled()` returns `false` for a `Transition` if at least one input `Place` without tokens exist (`tokens=0`); otherwise it returns `true`. The operation `fire()` updates the amount of tokens held by the input and output `Place` accordingly. This is done by decrementing the value of the `tokens` attribute of the `Place` representing an input `Place` by calling `removeToken()`, and incrementing it for output `Places` using `addToken()`. Due to space limitations, these operations are not shown in Figure 3.

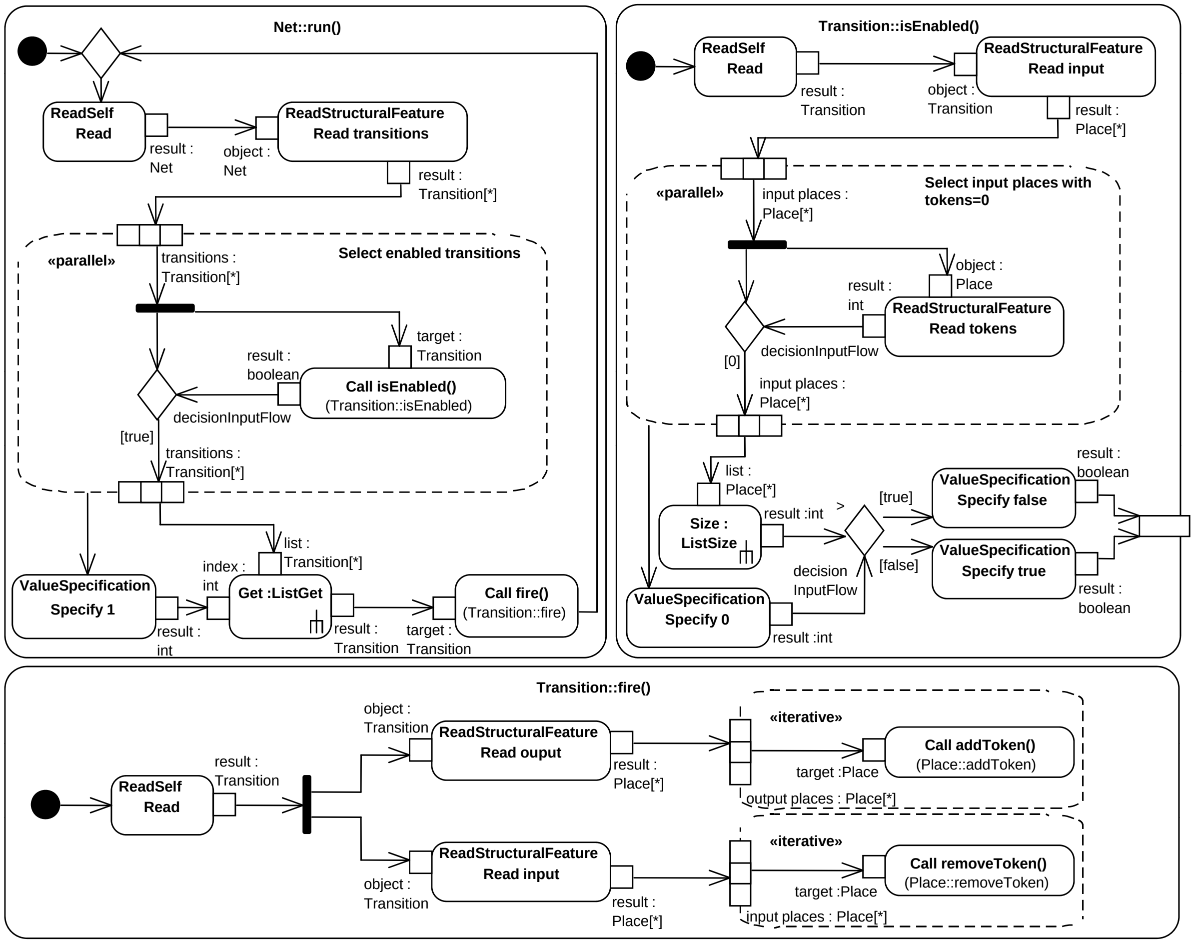

Figure 3. Semantics specification of Petri Net DSML using fUML

5. Extensibility of semantics

When generating code from models to execute them, one may benefit from the full power of the target GPL and, as a result, may utilize powerful libraries or interact with external resources, using their dedicated APIs. Unfortunately, this benefit is usually not provided when weaving the behavior into the abstract syntax of a DSML in terms of an *action language*, such as fUML, because developers may not escape the borders of the action languages' virtual machines.

To overcome this major drawback, we propose an approach for integrating external libraries with the fUML virtual machine. With this approach, we aim at realizing the following requirements: neither the metamodel of fUML nor its virtual machine should be extended, as this would break its conformance to the OMG standard. Moreover, the usage of external libraries should be transparent to the developer when designing the operational semantics of the DSML using fUML. Thus, developers should be able to interact with the components of the external libraries, including ingoing and outgoing data objects of these components, just as with any other component defined natively with fUML.

For realizing the aforementioned requirements, we propose to integrate the required interfaces of the external libraries into the fUML model and employ a dedicated integration layer, which forwards calls of the integrated interfaces to the actual external library at runtime. In the following, we discuss these steps in more detail.

Importing external libraries For importing the interfaces of external libraries, we may apply a reverse engineering framework for extracting a class diagram representation in terms of classes, as well as their fields and operation signatures of the external libraries. Once this class diagram is obtained, we may import it into the fUML model. The bodies of the operations of the integrated classes can be omitted. Instead, an empty fUML activity is added as behavior of each imported operation. These empty activities act as special place holder for the actual functionality of the external library, as described in the following.

Integrating external libraries at runtime Whenever activities representing place holders for operations of imported external libraries are called during the execution of the DSML, a dedicated integration layer forwards the call to the external library. To enable this layer to be notified whenever such a place holder activity is called and to allow for pausing the execution until the external library responds to the forwarded call, we make use of an event mechanism and a command API that we integrated into the standardized fUML virtual machine (cf. [10]). The developed event mechanism notifies listeners about the state of the fUML model execution; for instance, it indicates that a specific activity has been entered. The command API enables controlling the execution precisely in terms of suspending the execution at a certain fUML activity node, performing single execution steps, as well as resuming the execution. It is worth noting that while a model execution is suspended, we may access and modify the runtime model of the execution. Based on this functionality, the integration layer may register itself as listener to the fUML virtual machine and if a place holder activity is entered, it may suspend the execution to forward the invocation to the actual operation of the external library that is represented by the place holder activity and integrate its result back into the runtime model of the fUML execution. The same is done when an integrated class is instantiated in a fUML model. Instantiations, as well as modifications of instance values, are also indicated by the fUML virtual machine using dedicated events. Thus, when an imported class is instantiated (or an existing instance is modified), the integration layer may instantiate (or modify) an existing instance using the actual external library accordingly. To maintain a mapping between the instances in the fUML runtime and the actual instances of the external library, the integration layer also has to keep track of all created instances and their representatives in the fUML runtime.

In our Petri Net DSML example, one could require that if multiple `Transitions` are enabled, one is randomly chosen (instead of the first one) to enable a nondeterministic execution of the Petri Net. Therefore, the value specification action in the activity `Net::run()` that specifies that the operation `fire()` is called for the first `Transition` of the list of enabled `Transitions`, has to be replaced by a call operation action triggering the execution of an operation of an appropriate external library, such as `java.util.Random`.
Using this approach of integrating external libraries into the fUML virtual machine, enables language designers to exploit the full power of third-party libraries in the semantics specification of their DSML.

6. Conclusion and Outlook

In this paper, we presented ongoing work towards the usage of fUML as standardized means for specifying the behavioral semantics of DSMLs. We demonstrated how fUML can be integrated with MOF and how libraries of programming languages can be utilized in semantics specifications.
The next step in our research is the automatic generation of model execution facilities for DSMLs from their xMOF-based behavioral semantics specification. In previous work [10], we created the basis for enabling this. We enhanced the reference implementation of the fUML virtual machine in terms of a dedicated trace model, an event model, and a command API, thus enabling the runtime analysis, observation and control of the execution of fUML models. Using these extensions, we plan to implement an approach for deriving execution facilities for DSMLs in terms of dedicated model debuggers and testing environments.
Regarding the reusability of semantics specifications, we plan to elaborate kernel semantics representing reoccuring patterns in behavioral semantics specifications, such as for example control flow semantics, by surveying the semantics of existing DSMLs. Further, we plan to develop adequate means for formalizing these kernel semantics to provide facilities to reuse the kernel semantics in the behavioral semantics specification of DSMLs.

References

[1] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[2] B. R. Bryant, J. Gray, M. Mernik, P. J. Clarke, R. B. France, and G. Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Computer Science and Information Systems*, 8(2):225–253, 2011.

[3] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic anchoring with model transformations. In *Proc. of ECMDA-FA'05*, pages 115–129, 2005.

[4] T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied Metamodelling: A Foundation for Language Driven Development*. Ceteva, Sheffield, 2004.

[5] S. Ducasse and T. Gîrba. Using Smalltalk as a reflective executable meta-language. In *Proc. of MODELS'06*, pages 604–618, 2006.

[6] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proc. of UML'00*, pages 323–337, 2000.

[7] D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *Proc. of ECMDA-FA'06*, pages 128–142, 2006.

[8] T. Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[9] Q. Lai and A. Carpenter. Defining and verifying behaviour of domain specific language with fUML. In *Workshop Proc. of BM-FA'12 @ ECMFA'12*, pages 1–7, 2012.

[10] T. Mayerhofer, P. Langer, and G. Kappel. A runtime model for fUML. In *Workshop Proc. of MRT'12 @ MODELS'12*, 2012.

[11] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of MODELS'05*, pages 264–278, 2005.

[12] J. E. Rivera, F. Durán, and A. Vallecillo. On the behavioral semantics of real-time domain specific visual languages. In *Workshop Proc. of WRLA'10 @ ETAPS'10*, pages 174–190, 2010.

[13] M. Scheidgen and J. Fischer. Human comprehensible and machine processable specifications of operational semantics. In *Proc. of ECMDA-FA'2007*, pages 157–171, 2007.