A Generic Solution for Weaving Business Code into Executable Models

Olivier Le Goaer, Eric Cariou, Léa Brunschwig, Franck Barbier

A generic solution for weaving business code into executable models

Eric Cariou
Olivier Le Goaer
Léa Brunschwig
Franck Barbier
Univ Pau & Pays Adour
Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour, EA3000
PAU, France
firstname.lastname@univ-pau.fr

ABSTRACT

The separation of concerns is a fundamental principle that allows to build a software with separate parts, thereby improving their maintainability and evolutivity. Executable models are good potential representatives of this principle since they capture the behavior of a software-intensive system, that is, when, why and how calling business operations, while the latter are specified apart. EMF is the de facto framework used to create an executable DSL (xDSL) but a solution to weave business operations into it is still missing. This is compounded by the fact that such business operations can be tied to specific technological platforms that stand outside the EMF world (e.g. Android SDK). To that purpose, in this paper we describe a solution for managing business operations both at design-time (creation of executable models with EMF) and at run-time (operation calls from the deployed execution engine). This solution is generic enough to be integrated into any Java-based environment and for any xDSL.

KEYWORDS

Executable DSL, Xmodeling, operational semantics, CASE tool, EMF

1 INTRODUCTION

Executable modeling (Xmodeling) is receiving increasing attention, and hence creation of Executable Domain Specific Languages (xDSL) is centerstage. Executable models can be used for simulation purpose with mock data and operations, but they may also be deployed onto real devices. In that case, the formalism of the DSL captures the behavior of a system, while an embedded execution engine is responsible for the proper call of business-level operations (APIs) provided by the targeted platform.

For example, each elevator has its own firmware responsible for opening and closing doors, winding/unwinding the cable to reach a given floor. The actions of the elevator may be triggered according to a set of states and transitions preferably modeled through a finite state machine formalism. As another example, a travel booking system running on a server inserts customers information into a database or call Web services provided by air transport companies. The behavior of such a system specifies when these business routines have to be executed and under what conditions. The calls to the various Web services may be orchestrated through a BPEL or BPMN formalism.

However, a problem arises when using and creating an executable DSL (xDSL) for using such executable models: the data flow management. Writing a piece of code in a classic way consists in weaving a control flow with a data flow. Let's consider these basic lines of Android-inspired Java code:

```
1 ...
  Cursor sms = smsManager.getAllSMS();
3 String json = cloudManager.cursor2JSON(sms);
  cloudManager.save(json);
5 ...
```

The first line of code retrieves all the SMS stored into the smartphone. Then, this set of SMS is passed as parameter to a second method converting them into a neutral JSON format. Finally, in the last line of code, this JSON contents is saved in a Cloud. The control flow consists here in the definition of the sequence of the method calls and on which object they are called (here `smsManager` or `cloudManager`). The data flow consists in passing parameters to the methods and getting the resulting objects that could be further parameters of other methods. Here for instance, the object named "sms" is obtained by the first method call and passed as parameter to the second method. Of course, depending on the business methods, the number and type of parameters can vary as needed.

As an example of xDSL for applying our solution for managing business operations, we have defined a simple DSL using EMF: PDL for Process Definition Language. A process is composed of a set of ordered activities. The idea is that business operations can be associated with each activity. For our Android example, this will lead to define a PDL model with a sequence of three activities, one for each line of code. From now on, the control flow is reified in the form of a PDL model and the software engineer is only in charge of implementing the business operations. This is the execution engine of PDL that will carry out the complete execution of the model: once the business operation of an activity is finished, it goes to the next activity and executes its business operation and so on till the end of the process.

However, an important problem remains: how to define the data flow? How does one set for an activity which is the operation to be called, on which Java business object and with which parameters? How does one store the result of an operation to reuse it afterward as a parameter of the operation of another activity? This problem is not specific to our PDL language but is a recurrent problem when defining xDSL where executable elements are associated with business-level operations. In this paper, we propose a generic

solution for managing a data flow during the definition of any xDSL. It is composed of a set of meta-classes that can be added in any Ecore metamodel to define a data flow and a set of EMF Java classes that are used to define the execution engine thereof and that will manage automatically the execution of business operations. This solution is delivered as an Eclipse/EMF plugin called Xmodeling Studio [1].

We took Android as an example because development of an Android application requires the Android SDK and the Android Studio IDE, and it can certainly not be achieved in Eclipse/EMF. In this paper, we describe the integration of the execution engine of a xDSL defined with Xmodeling Studio and how to use its executable models in any Java-based development using any IDE. We are convinced that being able to push xDSL and models into mainstream software development (independently of Eclipse/EMF) can lead to a wider adoption of MDE.

The rest of this paper is organized as follows. In Section 2 we describe how a language engineer implements a xDSL, its meta-model and execution engine, using Xmodeling Studio. In Section 3, we explain how a software engineer uses the xDSL defined in any Java development. Before concluding, we discuss related work in Section 4.

2 DEFINITION OF A XDSL

In this Section, we explain how a language engineer can create a xDSL using Xmodeling Studio with associated business operations. As an example, we first define PDL which Ecore meta-model is depicted on Figure 1. Roughly, a process is composed of a set of activities, has a first and a last activity, and all these activities are ordered through their `next/previous` references. The reference `currentActivity` in `Process` enables to know which is the current active activity when the model is interpreted by the engine. The meta-classes `Xmod_Action` and `Xmod_Operation` (in green) are not part of the PDL definition. They are meta-classes automatically added by Xmodeling Studio for managing business operations as we explain in the following.

2.1 Meta-classes

The main idea is at run-time to tag the Java objects that will be either parameters or return values of business methods and also the objects on which the methods are called. The pairs tag/object are then put in a Java map (`java.util.Map`) that will be used by the execution engine to retrieve the objects, as we will see in the following.

A business operation (i.e. a plain Java method in the code) is defined with the meta-class `Xmod_Operation`. It has a method name, a set of parameters, a possible return value and the object on which it will be executed. This information (excepting the method name) corresponds to the tags of the Java objects in the map.

The meta-class `Activity` from PDL metamodel specializes `Xmod_Action`. This is how the business operations are added on meta-elements. We have leveraged the semantics of UML state machines for associating business operations as for an UML state: on entry, on exit and the do action. For almost all xDSL, the execution semantics consists in activating elements and deactivating other ones: from a state to another one when following a transition, from an activity to the next one when the current activity is finished... So, these notions of entering and exiting an element can be generalized to any xDSL. The three operations are optional. However, if one does not want to follow our pattern, the `operations` link between `Activity` and `Xmod_Operation` enables to add operations and to execute them as desired within the execution engine.

The link between our generic meta-classes and those of an Ecore metamodel can be made in two ways with our Eclipse plugin. If you have an existing metamodel, you have to annotate some meta-classes: by "Xmod_main" for the root element of the Ecore metamodel (that will help in the code generation as explained further) and by "Xmod_action" for elements that need to specialize `Xmod_Action`. Then, you launch a metamodel-to-metamodel transformation that will add our generic meta-classes at the right place in your metamodel. If you want to build a brand new xDSL, you create an "Empty Xmodeling project". This will create an EMF project with an Ecore metamodel containing our generic meta-classes. In both cases, in the "src-gen/" directory of the EMF project, the EMF code of our meta-classes will be placed including the built-in code for automatically executing business operations.

2.2 Execution engine

We provide the implementation of the EMF Java classes with the code that will automatically execute business operations. The main part of this implementation is the `execute` method of `Xmod_Operation`. Based on the tags, it first retrieves in the map the objects that will be the parameters of the method and also the object on which the method has to be called. Based on these objects and the method name, a dynamic call of the method is made trough the reflection mechanism of Java. If the method returns a value, it is put in the map with the right tag.

An utility class is also automatically generated for managing the map and the models. It is put for our example in the EMF generated `PDL.util` package and is called "PDLXmodUtil". This class contains static Java methods for:

- Loading a model and getting its root element, based on the "Xmod_main" annotation put on one meta-class of the meta-model. In our example, this annotation has been put on the `Process` meta-class, so the generated method will have the following signature:
  `Process loadProcess(String fileName)`
- Saving a model through a root element, here again based on the "Xmod_main" annotation
- Getting and setting the object map
- Loading and saving the in-memory object map through an XML file

During the execution, saving the model and the map allows to save the complete current state of this application: both its behavioral state and the current contents of the business objects. It can be used to build an execution trace that can be afterwards analysed or to reload and restart the application at a given point in time. However, depending on the size and number of the Java objects in the map and on the size of the model, such loading and saving can be heavy-load operations and then should be used sparingly.

[1] The tool is available at this address: http://www.pauware.com/xmodeling

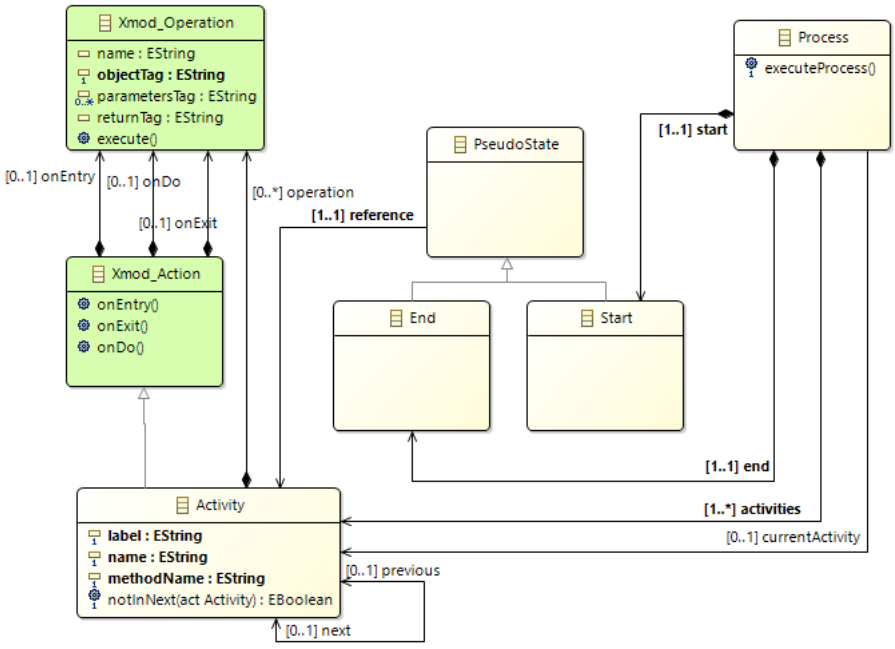


Figure 1: Ecore metamodel of PDL

Thanks to this generic code, the implementation of the execution engine for PDL is straightforward. It consists mainly in implementing the `executeProcess` method of `Process` within the EMF generated code:

```
public void executeProcess() {
  // get the first activity of the process
  Activity act = this.getStart().getReference();
  do {
    // update the current activity
    this.setCurrentActivity(act);
    // execute the operations of the activity if
    // defined by calling our implemented methods of
    // Xmod_Action that Activity is specializing
    act.onEntry();
    act.onDo();
    act.onExit();
    // go to the next activity
    act = act.getNext();
    // end the loop if there is no further activity
  } while (act != null);
}
```

3 IMPLEMENTATION OF A SOFTWARE WITH A XDSL

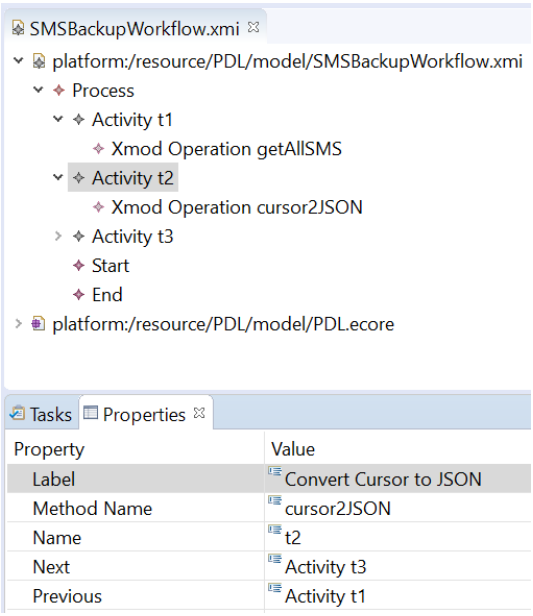
3.1 Business code and executable model

Once the language engineer has released its xDSL, any software engineer can used it to create a software. He/her has to provide an implementation in Java for the business operations (`getAllSMS`, `cursor2JSON` and `save` for our Android example). This development can be made in any Java environment. Then, using EMF, he/she defines the executable PDL model. If using a textual syntax created with Xtext[2], it will give the model of the Figure 2.b for our Android example. It defines a sequence of three tasks each one associated with one business operation (on entry for the two first and on do for the last one). The "on" enables to express on which object (its tag) the operation will be called, "result" the tag of the returned object of the operation. We can notice the tag "allSMSContent" is the result of the operation of the first activity and is passed as parameter for the operation of the second one. In the same way, the result of the second activity tagged "json" is passed as the parameter of the operation of the last activity. This way, the data flow among activities is automatically managed and our generic implemented EMF Java classes will manage the Java objects at run-time through their tags.

For the software engineer building a business application, once its PDL model defined, he/she has to write a code that will initialize the contents of the map and then launch the process execution. For instance, for our small example, it will be something like this:

```
// create the initial contents of the map with business
// objects on which methods will be called
HashMap<String, Object> map = new HashMap<>();
SMSManager smsManager = new SMSManager(...);
CloudManager cloudManager = new CloudManager(...);
map.put("sms", smsManager);
map.put("cloud", cloudManager);
// load the contents of the PDL model through our
// generated utilitary class
Process proc;
proc = PDLXmodUtil.loadProcess("SMSBackupWorkflow.xmi");
// set the map through our generated utilitary class
PDLXmodUtil.setMap(map);
```

[2]This concrete syntax has not yet been implemented for PDL. Currently, the PDL model is created using the EMF generic XMI file editor as presented on Figure 2.a.



(a) abstract syntax

```
process {
   t1 {
      label "Get all SMS"
      call as entry getAllSMS on sms result allSMSContent
   }

   t2 {
      label "Convert Cursor to JSON"
      call as entry cursor2JSON(allSMSContent) on cloud result json
   } next of t1

   t3 {
      label "Backup in Cloud"
      call as do save(json) on cloud
   } next of t2
}
```

(b) textual concrete syntax

Figure 2: PDL model for the Android application example

```
// execute the process: the operation of activities will
// be automatically called by our generic meta-classes
// and the data flow is managed by the tags in the map
proc.executeProcess();
```

3.2 Deployment

Unlike EMF standalone deployment which is done typically exporting as RCP Application, we need a custom, minimal deployment. Indeed, the interpreter has no graphical user interface: it is rather an autonomous executable JAR, able to take XMI files as inputs; provided they conform to the metamodel of the given xDSL. The core libraries coming from EMF that must be isolated from the rest are:

- `org.eclipse.emf.common.jar`
- `org.eclipse.emf.ecore.xmi.jar`
- `org.eclipse.emf.ecore.jar`
- `org.eclipse.ocl.pivot.jar`

In addition, the package of the ANTLR Parser generated by Xtext can be embedded so that a concrete textual syntax is also supported as input. The lightweight, low-dependency and low-memory footprint criterion must drive the deployment phase so that the interpreter is truly embeddable in tiny devices like Android smartphones for example (the four above packages have a total size under 5 MB and even under 2 MB without the OCL run-time verification package).

As a proof of concept, we have successfully implemented a simple Android application with Android Studio using PDL [3]. It basically requires to add in the project the EMF JAR files listed just above, the JAR generated from the EMF classes for PDL including the code of the PDL execution engine and of our generic code for managing the data flow. The *SMSBackupWorkflow.xmi* file has also to be put in the project.

[3]Sources and technical details are available on http://www.pauware.com/xmodeling/android-example.html

3.3 Discussion

At a glance, this way of programming is more complex and less convenient than the classic way but this is due to the basic application example and the choice of a minimalist metamodel as a xDSL example. Our PDL simply executes one sequence of activities and our Java example is a sequence of lines of code. Consequently, in both cases, the behavior is reifed under the concept of sequence. Then, the xDSL does not offer a more high-level and abstract way to define the behavioral part of an application. But let's imagine an extension of PDL with parallel sequences and forks or joins among them. In this case, writing by hand in Java an application following this behavior (such as a method that must wait the end of several parallel methods to be executed) will be a headache whereas it will be straightforward and much quicker using the extended PDL formalism as simply requiring to define a model. This highlights the interest of using xDSL for defining the behavior of an application. Another benefit of such an approach is that changing the behavior of the application is simply made by modifying the model and does not require to change any line of Java code (except business operations of course).

Using a map requiring to put in it all the objects used as parameters, return values and target objects of the business methods seems to be fastidious. But if you are using coarse-grained business actions, once the initialization of the business objects on which the operations will be called is done and that they have been put in the

map, the data flow is automatically managed: results of activities will mainly be the entries of other ones as defined in the high-level executable model. It will not be necessary to put many other objects in the map.

4 RELATED WORK

Several recent research initiatives have been launched to help in building executable DSLs and associated tools. The GEMOC Studio [3] enables to define such DSLs with a lot of tooling features for model edition, simulation, debugging, code generation, definition of execution engine…The GEMOC initiative has for goal to integrate different features and works around implementation at large of executable DSL. Melange [12] enables also to integrate several DSLs together and to define an execution engine in Kermeta 3[4]. [5] and [4] provide generic approaches to add trace and debugging features to any executable DSL. xMOF [15] allows to defined for MOF-based DSL an execution semantics with fUML [18]. [20] is also using fUML for defining an execution semantics but for UML profiles. Kermeta is an interesting language for developing a xDSL: high-level executable code is added by aspect mechanisms to meta-classes, making a clear separation between the structural meta-model and the code manipulating the model. Then, Kermeta can generate the equivalent Java code, that is, the running execution engine. All these works offer interesting and powerful features which can be inspiring for our Xmodeling CASE tool but they do not explicitly focus on the business part of the software application to build as we do.

Of course business code can be integrated during the software development and even legacy code when using these tools. For instance, [11] merges existing DSLs using Melange for specifying business and behavorial parts of an IoT (Internet of Thing) application. But as the development of the final software is mainly based on Eclipse/EMF, it can make difficult to reuse some existing frameworks and can even make some developments almost impossible as for Android applications.

[16] uses a middleware to make interacting a model execution engine (called a virtual machine) for a xDSL with business services. The description of the business operations is made through dedicated script controls whereas in our approach, there are defined directly in the executable model and associated with executable elements. Their approach deals also with advanced features such as runtime adaptability based on context and available resources. With Xmodeling Studio, we focus only on the execution engine implementation and its link with business operations in order to make it the most simple to integrate in any Java-based software development.

fUML is used in previous citations as a language for defining the semantics of a modeling language, i.e. at the metamodeling level, but it has been released originally for defining the equivalent of a programming code in UML models. One can then design for an application the class diagrams, state machines, sequences diagrams and any required diagrams and then add the contents of the class methods in fUML either through a kind of activity diagram or using a pseudo-code syntax thanks to ALF [17]. For instance, [13] generates Java or C++ code from fUML specifications. The software engineer can then add the business part of the application in the UML models. It enables a 100% model development of a software application which is somehow the holy grail of MDD. However in this case, it is not possible to reuse existing code developed in a standard programming language: everything has to be translated under the form of UML diagrams and fUML code which is not an easy task and is indisputably a brake on the use of these full-modeling techniques.

At the complete opposite of these full-modeling approaches but still in making a clear separation between the behavioral and the business parts, there is a tool such as PauWare[5] [2]. PauWare is both a Java library for "programming" UML state machines and an execution engine for them. Developing with PauWare consists in instantiating Java library classes for creating the states and transitions of the state machine and to associate with them business actions coded through Java methods. With PauWare, everything is directly defined in plain Java and using it simply required to add a JAR file in the Java project of your favorite or required IDE. The UML state machine is not defined with a modeling tool but is reified under a set of instances of Java classes, that is, at a low level.

In this paper, we propose an intermediate and pragmatic approach where the behavioral part of an application is defined with an executable model and the business part is defined by the software engineer in plain Java. This allows him/her to reuse directly existing code and to use the IDE he/she prefers or the one that is required such as Android Studio when developing Android mobile applications. We propose tools and techniques to glue these behavioral and business parts together through automatic code generation, for any executable DSL.

5 CONCLUSION

In this paper, we have presented a solution for weaving business operations into executable elements of any xDSL. Inspired by the semantics of UML state machines, business operations are associated with an executable element on entry (when it is actived), on exit (when it is deactivated) and as a do action (when running). We provide generic meta-classes for defining business operations that can be automatically added to any meta-model. We also provide boilerplate EMF Java classes for these meta-classes that contain a generic code carrying out the automatic execution of the business operations through the Java reflection mechanism and helping in the implementation of the execution engine of a xDSL. At run-time, the Java objects (business objects, method parameters or returned values) are put in a map. Our generic code will access to this map to get or set the required objects for an operation execution. This enables to manage the data flow among operations while the control flow is reified within the executable model. This solution is available trough an Eclipse/EMF plugin called Xmodeling Studio.

The originality of this solution is that it is mixing modeling and programming for an easier integration of MDE into everyday software developments within the Java ecosystem. Whilst defining a xDSL for simulation does not required business code, for a concrete software development where using existing API or legacy code is unavoidable, the link between these business parts and the executable model must be made. The xDSL created with Xmodeling

[4] http://www.kermeta.org

[5] http://www.pauware.com

Studio, more precisely their execution engines, can be deployed independently of the targeted software or platforms: an Android application, a Web server, a desktop application, an Arduino program... This is due to the fact the executable model may reify any kind of application behavior, independently of the business operations required. They will be called by our generic EMF Java classes.

As perspectives, we will continue the development of Xmodeling Studio with new features. As well-established, an executable model contains static and dynamic elements [6–10, 14]. Static elements define the contents of the model (for PDL, the sequence of activities) whereas dynamic ones are used to manage the execution of the model (here the `currentActivity` reference of `Process` allowing to know which is the current activity under execution). The static elements of the model must not been changed during the execution: if you change the activities of a PDL model, you are changing the behavior of your software. In some cases, it can be considered for run-time adaptation [8, 19], but in most cases, it will be an unexpected problem. Hence, we plan to control the code written by the language engineer during the implementation of the execution engine of his/her xDSL in order to avoid at run-time the modification of the static elements of the executable models.

ACKNOWLEDGMENTS

The presented work is part of the MegaM@RT2 project (Megamodeling at Runtime – Scalable Model-based Framework for Continuous Development and Runtime Validation of Complex Systems) [1] which has received funding from the Electronic Component Systems for European Leadership Joint Undertaking (ECSEL-JU) under grant agreement No. 737494. This project receives support from the European Union's Horizon 2020 research and innovation program and from Sweden, Spain, Italy, Finland & Czech Republic.

REFERENCES

[1] Wasif Afzal, Hugo Bruneliere, Davide Di Ruscio, Andrey Sadovykh, Silvia Mazzini, Eric Cariou, Dragos Truscan, Jordi Cabot, Daniel Field, Luigi Pomante, and Pavel Smrz. 2017. The MegaMRt2 ECSEL Project – MegaModelling at Runtime – Scalable Model-based Framework for Continuous Development and Runtime Validation of Complex Systems. In *European Projects in Digital Systems Design (EPDSD), Euromicro DSD/SEAA 2017*.

[2] Franck Barbier. 2016. *Reactive Internet Programming – State Chart XML in Action*. the Association for Computing Machinery and Morgan & Claypool. `http://www.pauware.com`.

[3] 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 2016)*, ACM (Ed.). https://hal.inria.fr/hal-01355391

[4] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient Debugging for Executable DSLs. *Journal of Systems and Software* (2018). http://www.sciencedirect.com/science/article/pii/S0164121217302765

[5] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. 2015. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *11th European Conference on Modelling Foundations and Applications (ECMFA 2015) (LNCS)*, Vol. 9153. Springer, 45–61.

[6] Erwan Breton and Jean Bézivin. 2001. Towards an understanding of model executability. In *Proceedings of the international conference on Formal Ontology in Information Systems (FOIS '01)*. ACM.

[7] Eric Cariou, Cyril Ballagny, Alexandre Feugas, and Franck Barbier. 2011. Contracts for Model Execution Verification. In *Seventh European Conference on Modelling Foundations and Applications (ECMFA 2011) (LNCS)*, Vol. 6698. Springer, 3–18.

[8] Eric Cariou, Olivier Le Goaer, Franck Barbier, and Samson Pierre. 2013. Characterization of Adaptable Interpreted-DSML. In *9th European Conference on Modelling Foundations and Applications (ECMFA 2013) (LNCS)*, Vol. 7949. Springer, 37–53.

[9] Peter J. Clarke, Yali Wu, Andrew A. Allen, Frank Hernandez, Mark Allison, and Robert France. 2013. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, Chapter 9: Towards Dynamic Semantics for Synthesizing Interpreted DSMLs.

[10] Benoit Combemale, Xavier Crégut, and Marc Pantel. 2012. A Design Pattern to Build Executable DSMLs and associated V&V tools. In *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*. IEEE.

[11] Thomas Degueule, Benoit Combemale, Arnaud Blouin, and Olivier Barais. 2015. Reusing Legacy DSLs with Melange. 15th Workshop on Domain-Specific Modeling.

[12] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*. ACM, New York, NY, USA, 25–36. https://hal.inria.fr/hal-01197038/document

[13] Codruț-Lucian Lazăr, Ioan Lazăr, Bazil Pârv, Simona Motogna, and István-Gergely Czibula. 2010. Tool Support for fUML Models. *International Journal of Computers Communications & Control* 5, 5 (2010).

[14] Grzegorz Lehmann, Marco Blumendorf, Frank Trollmann, and Sahin Albayrak. 2010. Meta-Modeling Runtime Models. In *Models@run.time Workshop at MoDELS 2010 (LNCS)*, Vol. 6627. Springer.

[15] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. 2013. xMOF: Executable DSMLs Based on fUML. In *The 2013 International Conference on Software Language Engineering (SLE '13) (LNCS)*, Springer (Ed.), Vol. 8225. Springer, 56–75.

[16] Karl A. Morris, Mark Allison, Fábio M. Costa, Jinpeng Wei, and Peter J. Clarke. 2015. An adaptive middleware design to support the dynamic interpretation of domain-specific models. *Information and Software Technology* 62 (2015), 21–41.

[17] OMG. 2017. Action Language for Foundational UML (ALF), version 1.1. `http://www.omg.org/spec/ALF/1.1/`.

[18] OMG. 2017. Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.3. `http://www.omg.org/spec/FUML/1.3/`.

[19] Samson Pierre, Eric Cariou, Olivier Le Goaer, and Franck Barbier. 2014. A Family-based Framework for i-DSML Adaptation. In *9th European Conference on Modelling Foundations and Applications (ECMFA 2014) (LNCS)*, Vol. 8569. Springer, 164–179.

[20] Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. 2014. Formalizing Execution Semantics of UML Profiles with fUML Models. In *17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014) (LNCS)*, Vol. 8767. Springer, 133–148.