Dethroning Programming Languages as Endorsed Means for Fine-grained UML Behaviour Modelling in Open Source MDE

Federico Ciccozzi

School of Innovation, Design and Engineering – IDT
Mälardalen University, Västerås, Sweden
federico.ciccozzi@mdh.se

Abstract. Models are means for unification and UML was born with the ambition of providing "unified" modelling language and methodology. The myriad of competing proprietary tools, with every tool provider only focusing on its own interests, resulted in the creation of a multitude of similar but still different solutions and "dialects", which clashes with UML's ambition. A glaring example is the appalling number of action languages and code generators defined for UML. With this work we recognise the need of a unified effort towards an open source baseline for getting the best out of UML. More specifically, we contribute by showing how to simplify the transition from the use of programming languages for modelling fine-grained behaviours within models to model-aware action languages in industrial MDE leveraging open source tools. This is achieved by making our solution for the automated *translational execution* of the Action Language for Foundational UML cooperate with existing UML-based code generators that exploit programming languages for defining action code.

Keywords: open source, model-driven engineering, code generation, Papyrus, UML, fUML, ALF, translational execution, C++

1 Introduction

Programming languages have represented the heart of software engineering from the early 70's, when software eventually turned to be too complex to be addressed by handwriting machine code; simplification by *abstraction* was needed, and programming languages seemed to be the answer [1]. Since then, abstraction has been progressively leveraged for managing software complexity; but that came at a cost. In fact, it usually brought about the need of additional development artefacts and phases, complicating, as a matter of fact, the engineering process. In this scenario, *automation* was regarded as a primary instrument for partially alleviating the complexity of such a process. Among the others, Model-Driven Engineering (MDE) emerged as a promising way to (i) mitigate software's complexity by abstraction via models, and (ii) provide means for automating engineering phases typically through model manipulations [2].

MDE promotes automation at almost any development phase, such as early analysis, simulation, code generation, and back-propagation from code to models. Among them, automated code generation represents a profitable feature that, especially from an industrial point of view, an MDE process cannot afford to fail in providing. The reason is that adequate code generation is able to mitigate the accidental complexity introduced by modelling activities [3] through reducing time-to-market as well as overall costs and risks. In addition, when automatically producing 100% of target code (i.e., *full-fledged code generation*) from models, consistency between models and code can be managed more easily, and configurability of code generators introduces the possibility to target different deployment and platform configurations and therefore different target languages from the same source models [4].

One of the crucial characteristics for a modelling language to suffice as input for full-fledged code generation is the ability to provide means for specifying fine-grained behaviours. Most often, this is done by inserting code written in common programming languages (e.g., C++, Java) as behavioural descriptions in the model. On the one hand, this represents a pragmatic way to address the problem in industrial settings since it enables to reuse legacy models with code describing behaviours; nevertheless, this practice can bring more drawbacks than benefits, as explained in the remainder of the paper. On the other hand, the use of model-aware action languages, based on the modelling languages themselves, is considered a preferable way for defining fine-grained behaviours when modelling a software system. This is the case of the Action Language for Foundational UML (ALF)[1], defined by the Object Management Group (OMG) to act as the surface notation for specifying executable behaviors within a wider model primarily represented using the graphical notations of the Unified Modeling Language (UML)[2].

In this paper we describe a “not-so-painful” transition from the use of programming languages within models to the adoption of model-aware action languages in UML-based industrial MDE leveraging open source tools. More specifically, we show how a solution for the automated *translational execution*[3] of ALF could be progressively introduced into already existing UML-based code generators that leverage programming languages for defining action code. In [5] we introduced our first attempt to the translational execution of ALF to show its feasibility; in this contribution we show how a newly implemented, fully functioning, translation solution based on the latest official implementation of the ALF specification, could be employed for our purpose.

The remainder of this paper is organized as follows. In Section 2 we describe motivation and context for the presented contribution. In Section 3 we give a snapshot on the history of action languages from the birth of UML, and in Section 4 we provide our open source solution for introducing translational execution of ALF in existing UML-based MDE code generators; the solution is run on a simplified version of the Self-Orienting Carrier Robot Software System. In Section 5 we discuss about the importance of a non-breaking solution in industrial settings as well as the need for a cooperative effort towards an open source UML baseline. The paper is concluded in Section 6 with the current status of the presented work as well as planned future enhancements.

[1] `http://www.omg.org/spec/ALF/1.0.1/`

[2] `http://www.uml.org/`

[3] According to the ALF specification, *translational execution* is regarded as the translation of ALF into an executable for a non-UML target platform on which it is executed.

2 Modelling behaviours with UML: ALF will be the star

In most UML-based MDE processes aiming at full-fledged code generation, the specification of fine-grained behaviours within UML models is done with action code written in common programming languages. Since, when UML started to get a foothold in industry, no proper action language was available, the use of programming languages for defining actions was almost unavoidable for achieving the generation of 100% code. The introduction of UML2, together with the standardisation of (i) the Foundational Subset For Executable UML Models (fUML), which gives a precise execution semantics to a subset of UML limited to composite structures, classes and activities (application models designed with fUML are executable by definition) [6] and (ii) a textual action language, ALF, to express fine-grained execution behaviors, has made UML a full-fledged implementation quality language [7]. The execution semantics for ALF is given by mapping its concrete syntax to the abstract syntax of fUML. Additionally ALF provides an extended notation that can be used to represent structural descriptions.

Even though pragmatically still leveraged for maximising the reuse of existing models and embedded behaviours, approaches employing programming languages for action code bring about several issues. For instance, how to maintain, or even check, consistency at modelling level when the abstraction gap between modelling and programming languages hinders action code from being aware of surrounding modelling concepts? Consistency is not the only issue. In fact, by using programming languages for defining action code, the developer may infer assumptions on the target platform, which undermine models reusability. To tackle these issues, the use of model-aware action languages like ALF, based on the modelling languages themselves, should be preferred when modelling fine-grained behaviours.

With the formalisation of ALF, we have noticed an increasing industrial interest in gradually moving towards legitimate action languages. It would be naive though to think that this adoption can be painless and swift, since the use of programming languages within models is rooted in UML-based industrial MDE. In fact, pragmatism and attention to costs, core priorities in industrial settings, go hand in hand with maximised reuse of legacy models. Our main goal is to show how, through our solution for the translational execution of ALF, we could support and boost this adoption process by giving the possibility to exploit ALF as a complement to existing MDE processes. More specifically, this would mean that MDE processes generating, e.g., executable C++ from UML with C++ as action code, could reuse legacy models (or parts of them) and at the same time endorse a "cleaner" model-driven approach by designing new models (or parts of them) entirely using UML and ALF.

Our open source solution is also meant to reivingorate the interest of developers and companies in UML-based MDE. At its dawn, UML was mainly seen as an instrument to describe purposes, support analysis, design, and document only, since its semantics was too ambiguous and much weaker than well-established programming languages. Therefrom, the initial strong interest of practitioners in MDE, and specifically in UML, briskly diminished. Thanks to the formalisation of its execution semantincs (fUML) and action language (ALF), UML has now become a powerful implementation quality language [7]. By providing an open source automation in the translational execution of ALF, we want to contribute to smooth the way for the interested practitioners in adopting UML-based MDE.

The modelling environment we leverage is Papyrus, an open-source integrated environment for editing Eclipse Modeling Framework (EMF) [8] models, supporting modelling and validation features for UML, fUML, and ALF on the Eclipse platform. Note that our solution is achieved by model transformations (both model-to-model and model-to-text) [9] defined through open source transformation languages, such as Xtend[4] and Operational QVT[5]. Moreover, by leveraging the implementation of ALF in terms of metamodelling concepts in Papyrus, the translational execution does not perform any parsing activity and operates on the ALF code in terms of its representation as a model.

3 Not all action languages were born equal

Many commercial tools, such as Enterprise Architect [10], IBM Rational Rhapsody [11] and IBM Rational Software Architect [12] (in all its versions) have historically taken advantage of programming languages to define fine-grained behaviours within UML models and generate full-fledged code; this practice brings a set of drawbacks, as mentioned in the previous sections.

When it comes to model-aware action languages, some of those proposed during the years were inspired by the action semantics of UML, but none of them conforms entirely to the formalised execution semantics defined through fUML. Starting from the very beginning, the Shlaer-Mellor Action Language (SMALL) [13] was the first of its kind and was specified to defined a data-flow-based execution similarly to fUML; nevertheless, the language was never actually implemented. In the context of the OOA tool for executable UML, the Action Specification Language (ASL) [14] represented a quite capable action language at the time. The OOA tool provided limited code generation features, which were augmented by its successor, MentorGraphics's BridgePoint [15]. This tool provided a powerful action language, known as the Object Action Language (OAL) [16], which was a proprietary dialect of the predecessor of ALF, the UML Action Language (UAL). UAL was adopted and customised [17] by IBM too, as part of the their Rational Software Architect tool [12].

The Platform Independent Action Language (PAL) was another proprietary action language, which was not based on the formal execution semantics of UML. It was exploited by the PathMATE tool for providing assisted code generation based on models and marking techniques [18]. An extension of PAL with concepts from the Object Constraint Language (OCL)[6] was produced by Motogna et al. [19]. OCL was used also by Jiang et al. [20], who defined the OCL4X action language, where OCL was enriched with meta-actions for changing the system state. A Java-inspired solution was represented by the Action Language for Business Logic (ABL) [21], which aimed at converting actions defined with the action semantics of UML, but included non-UML

[4] `https://eclipse.org/xtend/`
[5] `http://wiki.eclipse.org/QVTo`
[6] `http://www.omg.org/spec/OCL/2.4/`

concepts. +CAL [22] was an action language for distributed real-time embedded systems and based on the action semantics of UML.

Each of the aforementioned action languages, mostly defined to achieve model simulation and code generation, was constructed either through the use of programming languages, or the proprietary customisation of the UML action semantics. To the best of our knowledge, none of the approaches documented in the literature provides a solution for the translational execution of the de-jure standard action language for UML, ALF. Moreover, proprietary action languages were "closed" solutions. Our goal is instead to endorse ALF and its open source implementation provided by Papyrus. Additionally, we build upon it for providing an open source solution to the translational execution of ALF to C++, which can be freely used[7] and customised by researchers and practitioners.

4 Towards a less painful transition to cleaner MDE in industry

In this section we describe how our solution for the translational execution of ALF to C++ can be seamlessly integrated into an already existing UML-based code generator that exploits C++ for defining action code. In order to show the solution, we make use of a simplified version of the Self-Orienting Carrier Robot Software System. The task of this terrestrial robot consists of travelling between checkpoints in a delimited and known environment, and simulating item retrieval and delivery.

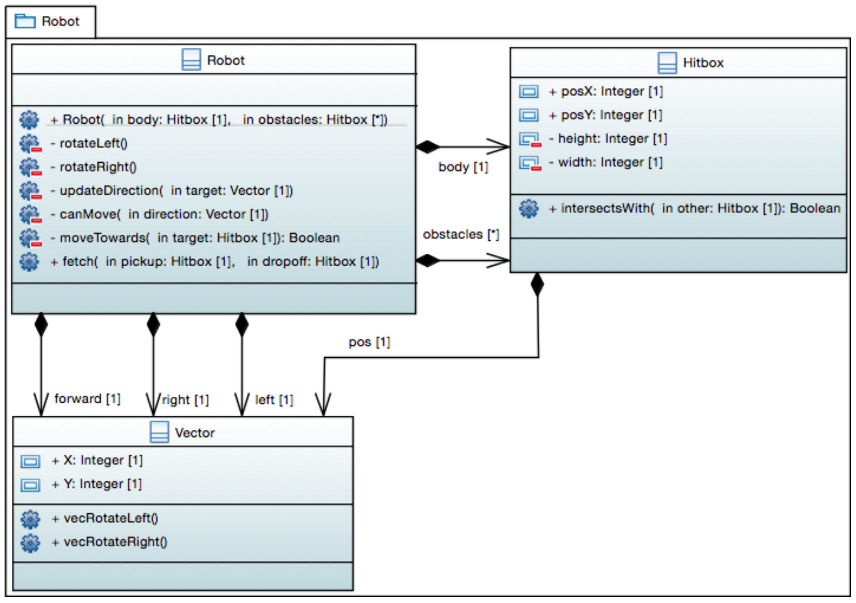
Robot
Robot
+ Robot(in body: Hitbox [1], in obstacles: Hitbox [*])
- rotateLeft()
- rotateRight()
- updateDirection(in target: Vector [1])
- canMove(in direction: Vector [1])
- moveTowards(in target: Hitbox [1]): Boolean
+ fetch(in pickup: Hitbox [1], in dropoff: Hitbox [1])
Hitbox
+ posX: Integer [1]
+ posY: Integer [1]
- height: Integer [1]
- width: Integer [1]
+ intersectsWith(in other: Hitbox [1]): Boolean
body [1]
obstacles [*]
pos [1]
forward [1]
right [1]
left [1]
Vector
+ X: Integer [1]
+ Y: Integer [1]
+ vecRotateLeft()
+ vecRotateRight()

Fig. 1: Robot modelled with UML in Papyrus

[7] The interested reader shall not hesitate in contacting the author to get the latest version of the translational execution implementation.

The application is intended to give the robot the ability to orient itself around obstacles of square shape; obstacles are created in different spots within the environment's delimitations. Similarly, a set of pick-up spots and one drop-off spot are created too. The system has been modelled in Papyrus by means of a UML class diagram for defining the system's structure, as shown in Figure 1. `Robot` is the main class and leverage two other classes, `Hitbox` for identifying sensitive spots (robot's body, pick-up and drop-off spots, obstacles) and `Vector` for moving around in the delimited environment.

```
{
  let res : Boolean = true;
  this.Body.pos.X += dir.X;
  this.Body.pos.Y += dir.Y;
  for(Hitbox obs : this.obstacles)
  {
    if(this.Body.IntersectsWith(obs))
    {
      res = false;
      break;
    }
  }
  this.Body.pos.X -= dir.X;
  this.Body.pos.Y -= dir.Y;
  return res;
}
```

Code 1.1: Fine-grained ALF behaviour for operation `canMove()`

```
{
   int temp = this->X;
  (this->X = ((this->Y * this->Y) * this->Y));
  (this->Y = ((temp * temp) * - temp));
}
```

Code 1.2: Fine-grained C++ behaviour for operation `vecRotateRight()`

Regarding fine-grained behaviours, in terms of bodies of operations owned by classes, both ALF and C++ have been used (separately for individual bodies). As an example, we can see the ALF definition of the `canMove()` operation owned by `Robot` and used to check if the robot can move in a specific direction, shown in Code 1.1, and the C++ definition of the `vecRotateRight()` operation owned by `Vector`, used for turning right and depicted in Code 1.2.

For the structural translation we make use of an existing code generator, which we had previously implemented in Papyrus [23], extended for class diagrams that, from a UML class diagram (including packages), produces the corresponding C++ skeleton code. During the structural translation, the generator checks for each operation if there is a fine-grained behaviour defined in terms of a UML opaque behaviour. If yes, the generator can perform either of the following three actions depending on how the opaque behaviour is defined:

1. Opaque behaviour defined in C++: the generator copies the body, as it is, in the corresponding method implementation in the resulting .cpp file;
2. Opaque behaviour defined in ALF: the generator triggers the model-to-text transformation implementing the translation from ALF to C++ and puts the resulting C++ code in the corresponding method implementation in the .cpp file;

3. Opaque behaviour defined in other languages: the generator does not take any action. This specific case could be handled by triggering an ad-hoc transformation for the specific language (if available) as in action 2, or by performing an action similarly to 1, in case of deployment of different functions (and classes) to different hardware nodes running different target languages.

```cpp
1  #include "Robot.h"
2
3  namespace ALF2CPP
4  {
5    namespace Robot
6    {
7      // .....
8
9      bool Robot::canMove(shared_ptr<ALF2CPP::Robot::Vector> dir)
10     {
11       bool res = true;
12       (this->Body->pos->X += dir->X);
13       (this->Body->pos->Y += dir->Y);
14       for (auto &obs : this->obstacles)
15       {
16         if (this->Body->IntersectsWith(obs))
17         {
18           (res = false);
19           break;
20         }
21       }
22       (this->Body->pos->X -= dir->X);
23       (this->Body->pos->Y -= dir->Y);
24       return res;
25     }
26     // .....
27
28     void Vector::vecRotateRight()
29     {
30       int temp = this->X;
31       (this->X = ((this->Y * this->Y) * this->Y));
32       (this->Y = ((temp * temp) * - temp));
33     }
34 }
```

Code 1.3: Extract of the generated C++ .cpp file

The generated code is C++. A portion of the generated .cpp file is shown in Code 1.3, where we can see the results of the different actions that the generator took when it encountered opaque behaviours defined in different ways. The C++ description of `vecRotateRight()` operation is copied to the output .cpp file (lines 28-33). For the ALF description of `canMove()` operation, it translated it to C++ (lines 9-25). During this translation, the transformation took care of two core aspects: type deduction and garbage collection[8].

For instance, in ALF access to members is done through a dot operator ('.'). This can be translated in C++ into dot, arrow ('→'), or even semicolon ('::') operators depending on the type of objects as well as the memory management mechanism. To correctly generate access to members, the transformation is equipped with type deduction mechanisms. An example is the translation of the ALF expression **`this`**`.Body.pos.X`

[8] In ALF a transparent garbage collection is supposed to take care of automatically managed memory, while in C++ a specific garbage collection mechanism needs to be defined.

(Code 1.1, line 3) that is transformed into **this**->Body->pos->X in C++ (Code 1.3, line 12). The dot operators are translated into arrows since we use *shared pointers* for managing garbage collection in C++, therefore Body and pos, objects of type Hitbox and Vector, are wrapped as **shared_ptr**<Hitbox> and **shared_ptr**<Vector> respectively, and accessed as pointers.

Note that in this example we endorsed plain UML for structural modelling. Anyhow, the proposed solution for translational execution of ALF can be equally adopted when dealing with UML profiles; this has been validated by combining a code generator for the CHESS-ML profile and ALF in [23]. Additionally, we provide a translation of part of the concepts, addressed as ALF units, that are used to textually describe structural portions of a UML model (within the fUML subset). That is to say, the developer is able to even define the structural parts of the model in terms of ALF to get corresponding executable C++ generated entirely from an ALF model.

Since they do not represent the main contribution of this work, details on implementation and evaluation of the translational execution of ALF to C++ will be treated in a separate work. To give an idea on the scalability of the solution, we evaluated it on several models of different sizes, from the smallest containing 130 lines of ALF action code to the biggest containing 109335 lines. The transformation process was always able to produce executable C++ with a generation time under 10 seconds (for the biggest model) on a laptop running a 1,7 GHz Intel Core i7 with 8GB DDR3 RAM.

5 Reflections

Clearly, on the one hand a UML-based modelling approach leveraging a programming language for the definition of fine-grained behaviours makes code generation easier since action code is simply copied, as it is, in the resulting code artefacts. On the other hand, it does not give much control on the correctness of the modelled behaviours nor on how they are translated. Moreover, this practice binds the models to a specific platform (or set of platfoms) already at functional modelling level (e.g., by employing a specific target language or garbage collection mechanism); this may undermine, for instance, reusability of models. By employing an action language like ALF, action code is empowered with full knowledge of the surrounding model elements. This triggers several benefits, among which simplified model-based analysis, model simulation and consistency checking at modelling level, to mention a few. When it comes to reusability of models, since ALF is not bound to any specific target platform, code generators can target different variations (providing some degree of reusability for code generators too) of one language (e.g., different garbage collection mechanisms depending on the user's selection) or different languages, from the same model. While we showed how to gradually move towards fully ALF-compliant modelling by mixing ALF and C++, it should be clear that developers get to enjoy platform-independence if only ALF is used, with no opaque behaviours written in the other languages (such as C++ in our example). Moreover, one of the main motivations for using a traditional programming language in models is to access code from existing external libraries and software components written in that specific language. This feature can be kept even when exploiting fully ALF-compliant behavioural modelling thanks to the possibility to define inline code snippets in any programming language within ALF. These snippets would then be taken into consideration only when translating towards the specific programming language they are written in, and ignored otherwise without jeopardising platform-independence at modelling level.

At this point a question arises: *what about legacy models with action code written using programming languages?* In order for existing UML-based MDE processes to adopt ALF we believe that a non-breaking solution, such as the one we propose, is pivotal. By providing the possibility to leverage legacy models and thereby a way to progressively replace programming languages with ALF for the specification of new fine-grained behaviours, the transit to a cleaner MDE in industry can become less painful.

A more general question would then be: *why open source?* The main goal of UML from its birth was to provide a "unified" modeling language; moreover, models in general have historically been seen as means for unification [4]. Initially, and until a few years ago, most of the well-established modelling environments used in industry were provided by specialised companies under different kinds of license. The rush towards the "most powerful" or the "most usable" UML tool made the various developing parties to fall into the prisoner's dilemma. Each party took its own way and protected its interests by customising, enhancing, creating dialects of a language and a methodology which was thought for unifying interests and solutions, rather than splitting them apart. The result of this self-centered way to exploit and build upon UML ended up in creating a modelling landscape which in many ways contradicts the UML's creed. With this we would like to stress the fact that, in order to get the best out of a unification technology, unified intents and efforts are crucial, and an open source solution, acting as a common baseline for domain-specific customised variations, is unescapable. In addition, a synergic effort between developers, users and researchers from both industry and academia is vital for triggering ideas, sharing problems, finding solutions, and evaluating results.

6 Outlook

In this paper we highlighted the need of a unified effort towards an open source baseline for UML. In particular, we contributed by proposing a way to simplify the transition from the use of programming languages within models to the adoption of model-aware action languages in UML-based industrial MDE. This is achieved by making our solution for the automated translational execution of the ALF to cooperate with existing UML-based code generators that use programming languages for defining action code. The solution leverages open source languages and tools only, and can be freely used and customised. We are currently working on enhancing our solution for providing the generation of Java, different alternatives for garbage collection, and so forth. At the same time we are investigating the possibilities to minimise semantic pollution, typical of translational approaches and due to the idiosyncratic differences between modelling and programming languages, by moving towards compilative and intepretive execution of ALF (and UML). This would represent an even further step towards neat UML-based MDE free from programming languages as part of models.

Acknowledgements

This research is supported by Ericsson AB, ABB Corporate Research, Alten Sweden, and KKS through the SMARTCore project (`http://www.es.mdh.se/projects/377-SMARTCore`). The authors would like to thank Thomas Åkerlund, and Knightec AB, for the important contribution to the implementation of the translational execution.

References

1. M. H. Van Emden and R. A. Kowalski. The Semantics of Predicate Logic As a Programming Language. *J. ACM*, 23:733–742, 1976.
2. D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39:25–31, 2006.
3. C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. *Sosym*, 7(3):345–359, 2008.
4. J. Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.
5. F. Ciccozzi, A. Cicchetti, and M. Sjödin. Towards Translational Execution of Action Language for Foundational UML. In *Procs of SEAA*, pages 153–160, 2013.
6. Jrmie Tatibout, Arnaud Cuccuru, Sbastien Grard, and Franois Terrier. Formalizing Execution Semantics of UML Profiles with fUML Models. In *Procs of MODELS*, pages 133–148. 2014.
7. B. Selic. The less well known uml. In *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2012.
8. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T.J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.
9. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, pages 621–645, 2006.
10. Sparx-Systems. Enterprise Architect. `http://www.sparxsystems.com.au/`, 2012.
11. IBM. Rational Rhapsody. `http://www-01.ibm.com/software/awdtools/rhapsody/`, 2012.
12. IBM. Rational Software Architect. `http://www.ibm.com/developerworks/rational/products/rsa/`, 2013.
13. Project Technology Inc. Shlaer-Mellor Action Language. `http://www.modelint.com/downloads/small.pdf`, 1997.
14. Kennedy Carter Ltd. UML ASL Reference Guide. `http://www.ooatool.com/docs/ASL03.pdf`, 2003.
15. Mentor Graphics. BridgePoint. `http://www.mentor.com/products/sm/bridgepoint`.
16. Mentor Graphics. Object Action Language Reference. `http://www.mentor.com/products/sm/techpubs/object-action-language-reference-manual-38098`, 2013.
17. M. Mohlin. Using the UML Action Language in Rational Software Architect. `http://www.modelint.com/downloads/small.pdf`, 2011.
18. PathFinder Solutions. Platform Independent Action Language (PAL). `http://www.ooatool.com/docs/PAL04.pdf`, 2004.
19. S. Motogna, B. Pârv, I. Lazar, I.G. Czibula, and C.L. Lazar. Extension of an OCL-based Executable UML Components Action Language. *Studia Universitatis Babes-Bolyai, Informatica*, 53(2):15–26, 2008.
20. K. Jiang, L. Zhang, and S. Miyake. OCL4X: An Action Semantics Language for UML Model Execution. In *Procs of COMPSAC*, pages 633–636, 2007.
21. C. Heitz, P. Thiemann, and T. Wlfle. Integration of an Action Language Via UML Action Semantics. In *Trends in Enterprise Application Architecture*, volume 4473, pages 172–186. Springer Berlin Heidelberg, 2007.
22. I. Perseil and L. Pautet. A Concrete Syntax for UML 2.1 Action Semantics Using +CAL. In *Procs of ICECCS*, pages 217–221, 2008.
23. F. Ciccozzi, A. Cicchetti, and M. Sjödin. On the Generation of Full-fledged Code from UML Profiles and ALF for Complex Systems. In *Procs of ITNG*, February 2015.