Executable Modeling with fUML and Alf in Papyrus: Tooling and Experiments

Sahar Guermazi, Jérémie Tatibouet, Arnaud Cuccuru, Saadia Dhouib and Sébastien Gérard
CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems
P.C. 174, Gif-sur-Yvette, 91191, France
<firstname.lastname>@cea.fr

Ed Seidewitz
Model Driven Solutions
14000 Gulliver's Trail
Bowie MD 20720, USA
ed-s@modeldriven.com

***Abstract*—fUML and Alf are two OMG standards dealing with executable modeling in UML. fUML focuses on semantic aspects, while Alf focuses on syntax. Papyrus (the UML/SysML modeler of the Eclipse foundation) provides tool support for these two standards. The purpose of this article is to provide the community with feedback and lessons learned by the Papyrus team regarding their implementation and usage of these standards, with the perspective of domain-specific uses of the tool. The feedback related to fUML is intended to highlight how tool developers can leverage fUML semantics to develop user and/or domain-specific model execution environments. The feedback related to Alf focuses on key end-user functionality: the combined usage of Alf and UML, with or without profiles.**

***Index Terms*—fUML, Alf, Model Execution, Semantics, Simulation**

I. Introduction

Papyrus[1], the UML/SysML modeler of the Eclipse foundation, provides tool support for executable UML modeling. It includes technologies for the execution and debugging of models, as well as editing facilities to produce executable models more efficiently. The execution part is handled by a plug-in called Moka[2], which relies on an implementation of the OMG standards fUML (Semantics of a Foundational Subset for Executable UML models[3]) and PSCS (Precise Semantics of UML Composite Structures[4]). These specifications formalize the execution semantics of a UML subset. The editing part relies on the OMG standard Alf (Action Language for fUML[5]), with an editor and compiler for that language.

One of the key features of Papyrus is that it can be easily customized to address user and/or domain-specific needs, typically by relying on UML profiles. This feature has been taken into account in the development of the executable modeling tooling, with the idea of using the fUML, PSCS and Alf implementations as a basis. The purpose of this article is to provide the community with feedback and lessons learned by the Papyrus team regarding their implementation and usage of these standards, with the perspective of domain-specific uses of the tool.

Section II focuses on semantic aspects and discusses challenges we faced for the design and implementation of fUML-based execution engines. Section III focuses on syntactic aspects and discusses various use cases and strategies for combining the use of Alf and UML, including its profiles. Section IV briefly discusses how our implementation challenges have been considered in the implementation of other Eclipse-based tooling. Section V concludes the paper and highlights some perspectives of our work.

II. Designing fUML-Based Execution Engines

One of the main goals of Moka is to provide a generic execution environment for Papyrus, in a way that is reusable for any user or domain-specific use of the modeling tool. Moka is based on a straightforward implementation of the fUML execution model (i.e., we basically implemented the interpreter described in this specification) so that any domain-specific flavor of Papyrus/Moka mostly comes down to a specialization of the fUML execution model for that domain. Since this model is object oriented, a specialization relies on typical mechanisms such as class inheritance and polymorphism.

The numerous experiments made by the authors in this area have highlighted four major concerns regarding the design of fUML-based execution engines: Extensibility, control and observability, time support, and connectivity with external tools. They are discussed in the following sub-sections, as well as the fUML-specific solutions used or developed by the authors to address them. Concrete use cases are mentioned for illustration purposes and potential limitations are highlighted. It is important to note that our use of Moka is mostly simulation oriented. Other uses (e.g., code generation and deployment) would probably require alternative solutions or implementation strategies [11], [12].

A. Extensibility

Problem statement: A DSL implemented as a UML profile may require abstract syntax elements (with their own execution semantics) that are out of the scope of the fUML subset (e.g., composite structures, state machines). It may also introduce stereotypes specializing fUML syntax and semantics. These new semantics must be taken into account by a fUML-based execution tool.

[1] https://eclipse.org/papyrus/
[2] https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
[3] http://www.omg.org/spec/FUML/
[4] http://www.omg.org/spec/PSCS/
[5] http://www.omg.org/spec/ALF/

Key fUML aspects: The fUML execution model is designed following the Visitor design pattern. Each (executable) element of the fUML abstract syntax is associated with a semantic visitor that describes its execution semantics. A factory (the so-called *ExecutionFactory*) is responsible for instantiating the appropriate visitors when an execution is started. Another factory (the so-called *Locus*) is responsible for instantiating objects, which represent instances of Classes specified in the model.

Proposed solution: According to this architecture, a natural way of building a domain-specific extension of the execution model is to introduce additional semantic visitors (either new visitors or extensions of existing ones), as well as extensions of the two factories. These visitors should capture the semantics of non-fUML abstract syntax elements, and/or account for the application of stereotypes.

Experiments: This extension strategy was successfully applied for specifying and implementing the execution model of OMG's PSCS, a normative extension of fUML dealing with the semantics of UML composite structures (and to which the authors have strongly contributed), including informative annexes on the semantics of a subset of the MARTE and SysML profiles. Following the same strategy, a systematic approach for specifying the execution semantics of UML profiles has been proposed in [9], as an increment to preliminary proposals developed in [2].

Limitations: The limitations are mainly related to the extension capabilities of the semantic model. The first limitation is introduced by the inhomogeneity of the usage of the factory pattern. When not used, this pattern implies that a visitor may have to embed redundant code. A typical example can be extracted from PSCS semantic model. The visitor CS_*AddStructuralFeatureValueAction* (PSCS specification, subclause 8.2.2.2.1) has to duplicate the code of the *doAction* operation inherited from a fUML visitor. This was due to the instantiation of links that was not handled via the factory pattern. A similar observation can be made in the case of Reference instantiation.

The second limitation is related to the capacity of the interpreter to work in presence of multiple profile applications. This usually implies that model elements have multiple stereotypes applied, each one with a particular semantics. While it is feasible to capture their semantics within semantic model extensions, it is not, for the moment, possible to combine different visitors during runtime in order to interpret a single model element.

B. Control and Observability

Problem statement: Modern execution and simulation tools should provide users with facilities to control (start/stop, suspend/resume, step by step, etc.) and observe (diagram animation, tracing) executions, for debugging purposes or simply to ease understanding of modeled systems.

Key fUML aspects: These tooling concerns are out of the scope of the fUML specification, since the execution model is fundamentally meant to enforce partial execution orders, and not to provide any strict recommendations on how this should be implemented. These execution orders are constrained by token propagation rules (semantics of Activities), where tokens flow (potentially concurrently) as long as they can through activity nodes and edges. The resulting causality must be respected by any compliant execution tool.

Proposed solution: In order to provide control and observation facilities, a fUML-based execution engine needs to reroute the usual token propagation flow through control and observation entities, in order to let them operate appropriately on the execution. This should be done in a way that preserves the original causality. The rerouting can be managed by specific semantic visitors, introduced using the extension mechanisms described in section II.A.

Experiments: This generic principle has been used to establish a connection between the fUML execution engine of Moka and the Eclipse Debug framework. For example, the semantics of activity nodes (captured by visitor *ActivityNodeActivation*) has been overloaded so that, when they receive offered tokens (operation *receiveOffer*), the control is rerouted towards an external control entity. The rerouting is done before the execution of the node is actually fired, as per the normative fUML semantics. This control entity is responsible for the management of debugging events (e.g., the user has stopped or suspended the execution, a breakpoint has been encountered) as well as the animation of nodes on diagrams.

In another experiment, the same delegation mechanism has been used to produce execution traces, by rerouting through a tracing entity. These traces have been used to check the correctness of fUML models with respect to higher-level models of reference scenarios [1]. We are currently refining this work in order to connect with the Papyrus model-based tracing framework. This framework allows adding, deleting and displaying static trace points on UML model elements. Trace points are used by Moka to identify the places where the execution flow has to be routed to the tracing entity. Execution traces are then generated in the CTF format[6], enabling visualization and analysis in the Eclipse Trace Compass Tool[7].

Limitations: The approach implies a pollution of the semantics with statements dedicated to the control derivation. In addition, it currently lacks a systematic methodology for identifying the points where control needs to be extracted, so that a good deal of expertise on the semantic model is required to make these identifications.

C. Time Support

Problem statement: As suggested by the various experiments mentioned in the previous sections, fUML can be used as a basis for a simple simulation process (model, execute, observe, and refine). It can be sufficient in the case where simulation objectives only concern logical correctness of the system. However, system designers usually have to account also for extra-functional aspects, which need to be reflected during experiments. Time is a common example of an extra-functional property that needs to be taken into account.

[6] http://www.efficios.com/ctf

[7] https://projects.eclipse.org/projects/tools.tracecompass

Key fUML aspects: As explained in the fUML specification (subclause 2.4), *"The execution model is agnostic about the semantics of time. This allows for a wide variety of time models to be supported, including discrete time (such as synchronous time models) and continuous (dense) time. Furthermore, it does not make any assumptions about the sources of time information and the related mechanisms, allowing both centralized and distributed time models"*. Implementations are thereby responsible to provide timing mechanisms if needed.

Proposed solution: In widespread simulation tools and frameworks such as Ptolemy II and SystemC, time is usually managed by an explicit control entity (scheduler-like), which appropriately schedules the execution of the various model elements in order to reflect the timing aspects. For the reasons mentioned above, the fUML execution model does not include such a control entity. Nevertheless, the delegation mechanism described in section II.B can be used to reroute the usual fUML execution flow towards a scheduler. The extension mechanisms described in section II.A can also be used to introduce new visitors, responsible for interpreting time-related information specified in a model, in connection with the scheduler.

Experiments: In [10], a discrete event (DE) scheduler for fUML models has been designed, along with a usage methodology to perform DE simulation with fUML models. In ongoing work, these mechanisms have been put into practice in an extension of the fUML execution model for the simulation of timed BPMN models. The extensions account for the duration of tasks as well as the occurrence of timed events. Once fired, tasks (which are modeled as Actions in an Activity) produce their output tokens only when allowed by the DE scheduler, after their simulated duration has elapsed. Similarly, timed events are triggered only when their simulated occurrence date is reached. This prototype was demonstrated at Eclipse Conference France 2015[8].

Limitations: Drawbacks are essentially the same as for the control delegation approach, since we rely on this mechanism. In addition, we only experimented on the integration of the discrete event time model. Other experiments are required to determine if the approach is valid for other time models.

D. Connectivity with external tools

Problem statement: Complex systems are usually composed of different parts that all relate to different kinds of engineering disciplines (mechanical, electrical, computer science). The consequence is that the various parts of the system are usually built using different modeling tools and modeling languages, with their own simulation environments. A co-simulation approach is usually required to assess the correctness of the overall system model. A fUML-based simulation tool must support this use case.

Key fUML aspects: The fUML subset includes Classes and opaque behaviors. The semantics of Classes are defined by the Object visitor. An Object represents a Class instance at runtime, and it is responsible for handling operation calls and signal receptions. The semantics of *OpaqueBehavior* are captured by the *OpaqueBehaviorExecution* visitor, which is abstract. Each *OpaqueBehavior* needs to be hooked to a concrete *OpaqueBehaviorExecution* implementing its own semantics. Specialized Object and *OpaqueBehaviorExecution* implementations can be used to communicate with external tools.

Proposed solution: The opening of Moka and fUML to external tools and libraries entirely relies on the principal of model signatures. It consists in capturing (using classes and opaque behaviors) the interfaces provided by external tools and libraries within fUML models. Moka then provides extension points to associate these signatures with actual visitor implementations. At runtime, when a call to an opaque behavior is made, the semantic visitor implementing its behavior drives the execution flow to logic that enables the connection to a specific tool or library. Similar logic is set up when operations are called on an instance of a Class representing the interface of an external tool.

Experiments: This solution has been put into practice for implementing the primitive behaviors defined in the normative foundational model library. The same principle has been used to establish connection with external graphical rendering tools[9]. We are currently reusing the same approach in connection with FMI (Functional Mockup Interface), a standard dedicated to co-simulation[10]. The idea is to enable a co-simulation between heterogeneous models within Moka. The global system to be simulated is organized as a set of FMUs (Functional Mockup Units, which represent simulation components) that are interconnected. The evolution of the system is governed at runtime by a so-called "master algorithm" expressed in Alf. When accessing an FMU, the master calls operations. These calls result in interaction with the FMU using the FMI API.

Limitations: The principle for connecting with external tools or libraries is well established. Nevertheless its coupling with the FMI standard for co-simulation is still a work in progress. Limitations may come to light as we go further in the implementation.

In this section of the paper, we have discussed how the fUML semantic model (and its implementation in Moka) could be reused for the development of user and/or domain-specific execution engines. The next section focuses on syntactic aspects. It discusses the combined usage of Alf and UML, and the related implementation challenges we are facing.

III. Combining Alf and UML (and its Profiles…)

In order to become actually executable, a fUML model may need to be very detailed. Unfortunately, activity diagrams (the standard graphical notation for UML Activities) quickly become unreadable as the number of nodes and edges increases. Alf provides a concise textual syntax to address this problem, which can be compiled into equivalent executable fUML models. Examples (developed using Papyrus tool support for Alf) are described in [8].

[8] https://youtu.be/ddogjaCtEbE

[9] https://youtu.be/SdDPl4HQ1n4
[10] https://www.fmi-standard.org/

While Alf brings a lot of flexibility to specify executable models, it was not intended to replace the diagrammatic syntax associated with UML. Indeed, diagrams are perfectly usable to represent the structural description of a system or high-level behaviors. What final users really expect is to be able to combine both notations (i.e. textual and diagrammatic). The choice of one or the other is driven by the level of detail required by the model or model element being specified. The combined usage of Alf and UML graphical notation has already been experienced by the Papyrus team. The following subsections describe the various use cases we found.

A. Combining at Unit Level

Problem statement: Graphical notations are useful for capturing the high-level aspects of a system, such as architectures of Classes, decompositions of Classes into parts, or signatures of Operations. Textual notations are useful to go deeper into the details of these elements, such as for the implementation of Operations. It shall be possible to combine the two kinds of notation in order to benefit from both worlds.

Key Alf aspects: Alf and UML can be combined at the "unit level". Alf units are equivalent to text files containing code. A unit typically contains a namespace definition, which can textually specify a Package, Class, Datatype, Enumeration, Signal, Association or Activity. From the user standpoint, this means it is possible to use Alf to fully describe such elements, whether they appear on diagrams or not. The advantage of using Alf at this level is that it offers a compact view of nested namespaces, which can be easily updated without having to go through several diagrams.

Proposed solution: The combination principle is to serialize Alf units as textual representations for UML namespaces. This is done by attaching stereotyped comments to namespaces (using the *TextualRepresentation* stereotype introduced in the Alf specification). Comments simply contain the Alf text body.

Experiments: We used Alf at the unit level in the test-suite model of OMG's PSCS, to specify almost all the Activities it contains, while keeping (composite) Class definitions in regular UML parts. The result is an executable model comprising a set of test cases, including assertions describing how the model should behave according to PSCS semantics. A tool can demonstrate conformance with the specification if no assertion is violated when executing the test suite. The test suite contains hundreds of Activities, so that, without Alf, it could not have been produced within the time frame of the joint submission preparation (~ 2 years).

Limitations: The main limitation of Alf for coupling at the unit level is related to its scope, which is limited to fUML. As highlighted in the work on PSCS, the subset of UML syntax covered by PSCS is larger than the fUML one. Consequently the design of the test suite implied the use of elements (e.g. ports) that are not supported in the context of Alf (only aligned with fUML). For example, the sending of a signal through a specific Port cannot be specified in Alf. Therefore, some of the test cases had to be refined manually after compilation of the Alf specification. Similar problems will certainly arise during the specification of the test suite for PSSM (Precise Semantics of UML State Machines). This illustrates the significance of aligning Alf simultaneously with the other parts of UML that have or will have their semantics formalized. Alf therefore needs to evolve beyond fUML. This evolution is being planned by the Executable UML Working Group within OMG.

B. Combining at Expression and Statement Level

Problem statement: In the context of a UML model, a typed expression language can be used any place an expression is required, such as in default values for properties of a classifier, lower and upper bounds of a multiplicity, or specification of constraints. UML tools like BridgePoint and RSA include this kind of functionalities, with the Object Action Language (OAL)[11] and the Unified Action Language (UAL)[12], respectively. Similarly, these tools also allow the specification of textual behavior statements locally, such as for the effects of a state machine transition or the method of a Class operation. These facilities greatly simplify executable modeling. Similar facilities should be supported by Alf-based tools.

Key Alf aspects: Alf syntax, as given in the OMG specification, is specifically designed to allow integration at the expression and statement level (as discussed at the beginning of Clauses 8 and 9, respectively, of the specification document).

Proposed solution: While the Alf specification was designed with this use case in mind, we have been facing issues due to our monolithic Xtext implementation of the grammar. In this implementation, Expression and Statement rules are not root rules, and type inference, validation, and compiling mechanisms are implemented specifically for this architecture of rules. Using expressions and statements in a context that is not planned by the grammar implementation requires an extension of these mechanisms, which is a complex task. The solution that we are currently prototyping consists in reducing the problem to an Alf/UML combination at unit level (as discussed in section III.A), with additional tool API for the derivation of the validation context.

Experiments: Regarding expressions, we are currently experimenting with the local editing of guards on transitions and default values of properties. We rely on the fact that UML OpaqueExpressions (i.e., a UML facility to embed textual expressions in a model) can have an associated behavior (this behavior specifies the result that should be obtained by evaluating the opaque expression, no matter what language it is specified in). From the tooling standpoint, the idea is to embed Alf-based editors directly into UML diagrams (with a content limited to the expression being edited), and serialize the Alf expression as an Alf unit for the Activity specifying the *OpaqueExpression*. Once correctly serialized (which typically requires the production of an extra return statement), and the validation context correctly determined using the tool API, the Alf unit can be validated as per usual Alf validation rules.

Regarding statements, we are experimenting with the local editing of effects on transitions and entry/do/exit behaviors on states. The solution is more straightforward, and it is directly applicable in any place where a Behavior (and therefore an

[11] http://www.ooatool.com/docs/OAL08.pdf
[12] https://www.ibm.com/developerworks/community/ual.pdf

Activity) can be specified. The embedded editor only shows the edited statements, while directly dealing with the serialization in the Alf unit corresponding to the edited Activity.

Limitations: This solution is a kind of workaround. A more permanent approach might be to refactor the Xtext implementation of the grammar into something more modular. Further investigations are however required to determine the feasibility of this modular approach (realizations made in the Xtext community around Xbase[13] suggest that it would be feasible), as well as its development cost.

C. Combining with profiles

Problem statement: Profiles can be used to implement DSLs on a UML basis. As shown in the examples of section II.A, these UML-based DSLs might be executable, so that the question of the use of Alf in the context of profiled UML models naturally arises.

Key Alf aspects: This use of Alf is considered in the OMG specification. The support in terms of stereotype application is however relatively limited. Alf stereotypes can only have attributes that are typed with primitive types or one attribute typed by a meta-class. In addition, stereotype attributes that are typed by a primitive type can only have a single value.

Proposed solutions: The proposed solution consists in extending the syntax, validation and compiling rules of Alf, consistently with the abstract syntax and the semantics of the considered UML profile.

Experiments: In [5] we have studied a refactoring of the Value Specification Language (VSL) as an extension of Alf. VSL has been standardized in the context of the UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)[14]. The main rationale for VSL was to provide users from the real-time domain with a simple textual syntax for specifying the values of non-functional properties of their system models. In VSL, rules for producing typed expressions mainly address aspects that are not specific to the real-time domain. They are related to general-purpose concerns that are also considered by Alf. Our proposal consisted in properly defining VSL as an extension of Alf, focusing on the aspects of VSL that are valuable for the real-time domain, and thereby leveraging the Alf specification and implementation. In a proof-of-concept implementation, we demonstrated how Alf syntax and validation rules could be extended for that purpose. The experiments were limited to the Expression subset of Alf.

Limitations: While the feasibility of the approach has been demonstrated, its applicability is limited by our current Xtext implementation of the Alf language and the fact that it is monolithic (as discussed in section III.B). The parts that would need to be extended or specialized for a particular domain cannot be easily extracted. More generally, Alf is a complex language, so that any extension (from both syntactic and semantic standpoints) should be considered with great care and may require a lot of development effort. It is not clear today if such extensions of Alf should be encouraged, and if evolution in the specification or implementation could make such extensions easier and safer.

[13] https://wiki.eclipse.org/Xbase
[14] http://www.omg.org/spec/MARTE/

IV. Related Work

We have built a modeling and simulation tool based on the OMG standards fUML, PSCS and Alf. During the definition of this tool, we identified challenges (extensibility, control, time support and connectivity with external tools) related to the use of the fUML standard for simulation, and we proposed solutions for them. In the literature, we have selected two tools that, by construction, should address challenges similar or very close to those we identified: Moliz[15] and Gemoc Studio[16]. These tools have been selected either for their strong link with fUML (Moliz) or their Eclipse-based implementation (both tools). It would have been interesting to consider the Cameo Simulation Toolkit[17] (fUML based) as well. However, since this is a commercial tool, we could not have access to the implementation and make relevant comparisons. We discuss in the following sections how these tools address the challenges we identified.

A. Extensibility

Domain specific languages are expressed with UML using profiles. To capture the semantics of these languages, we proposed a systematic approach to extend the fUML semantic model with new visitors. Moliz and Gemoc do not consider profiles. However they both address the definition of the semantics of MOF-based domain specific languages. Moliz uses fUML to define the semantics [7] while Gemoc uses Kermeta [4].

B. Control and observability

During simulation, we needed to be able to observe the execution to make it easier to understand a model. To do so, we proposed delegating, at specific points, the control of the execution flow to a particular entity (e.g. a debugger) to enable the control of the execution.

Moliz and Gemoc propose a similar approach. They both have a connection with a debugging environment based on the control-delegation principle. Furthermore, both tools use this approach to connect with trace generators (specific to fUML in the case of Moliz [6] and specific to each DSML in the case of Gemoc [4]). These two tools share an implementation approach based on aspect-oriented programming, which could be interesting for addressing the limitation we have identified (i.e., identification of control delegation point, as discussed in section II.B).

C. Time support

Time is a key aspect in simulation that is unfortunately not directly supported in fUML. To introduce time support, we relied on the control-delegation approach, with a control entity responsible for the evolution of time and the scheduling of the execution.

[15] http://www.modelexecution.org/
[16] http://gemoc.org/studio/
[17] http://www.nomagic.com/products/cameo-simulation-toolkit.html

Moliz does not directly provide a solution to capture time semantics during model execution. However it proposes a framework (i.e., a library) [3] for performance analysis that enables the integration of time representation in execution traces. Nevertheless, the approach seems to be also limited to discrete-event time models.

Gemoc enables time support in simulation thanks to clocks defined in CCSL (Clock Constraint Specification Language) [4]. The approach is interesting in the sense that it can combine various time models. However, using it in the context of fUML is not straightforward, since the approach makes strong assumptions about the way the syntax and semantics of a language are defined.

D. Connectivity with external tools

One important feature of Moka is to be open to external tools. This possibility is made possible by extending specific visitors of the fUML semantic model, whose implementation enables programmatically the connection with external tool. As far as we know, neither Moliz nor Gemoc provide comparable features.

V. CONCLUSIONS AND FUTURE WORKS

The Papyrus team is developing an open-source modeling and simulation framework based on standards. The goal of this platform is to support modeling and simulation of models described either with general-purpose languages or domain-specific languages.

The developed platform is composed of two parts. The first one enables the specification of executable models using Alf. The second one enables the execution of these models based on semantics defined by fUML and PSCS.

This paper was organized around these two aspects (cf. section II and III). It reports our use of the standards Alf and fUML as well as the limitations we encountered when using them. We identified two kinds of limitation: those that are related to the tooling of a standard and those that are related to the standard itself.

In the context of fUML, the limitation related to connectivity with other tools is really a tool issue. However those concerning extensibility, control delegation and time support are issues of the standard. The one related to extensibility seems to require a small refinement of the semantic model. Nevertheless this is not the case for the one related to control delegation. Indeed the model of computation of fUML is deeply coupled with the semantics definition and it will surely be painful work to extract it. In the context of Alf, the limitations attributed to the standard are only related to the support of profiles. The other limitations are really introduced by the initial version of the tooling that was implemented. For instance, to nicely integrate Alf at the expression level with UML some part of the tooling will need to improved.

The issues that are related to the standards are going to be addressed by the Executable UML Working Group in the near future. Technological improvements resulting from these refinements of the standards will be integrated in the further development of our tooling.

REFERENCES

[1] M. Arnaud, B. Bannour, A. Cuccuru, C. Gaston, S. Gerard, and A. Lapitre. Timed symbolic testing framework for executable models using high-level scenarios. In *Complex Systems Design & Management (CSDM),* pages 269-282, 2015.

[2] A. Benyahia, A. Cuccuru, S. Taha, F. Terrier, F. Boulanger, and S. Gérard. Extending the standard execution model of UML for real-time systems. In *Distributed, Parallel and Biologically Inspired Systems (DIPES), pages* 43-54, *2010.*

[3] L. Berardinelli, P. Langer, and T. Mayerhofer. Combining fUML and profiles for non-functional analysis based on model execution traces. In *Quality of Software Architectures (*QoSA), pages 79-88, 2013.

[4] B. Combemale, J. Deantoni, O. Barais, A. Blouin, E. Bousse, C. Brun, T. Degueule, and D. Vojtisek. A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio. 2015. Unpublished.

[5] A. Cuccuru, S. Gérard, and F. Terrier. Defining Marte's VSL as an extension of Alf. In *MoDELS'11,* pages 699-713, 2011.

[6] T. Mayerhofer, P. Langer, and G. Kappel. A runtime model for fUML. In *Models@Run.Time*, pages 53-58, 2012.

[7] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. xMOF: Executable DSMLs based on fUML. In *SLE'13*, pages 56-75, 2013.

[8] E. Seidewitz and A. Cuccuru. Agile programming with executable models: An open-source, standards-based Eclipse environment. In *Systems, Programming, and Applications: Software for Humanity*, SPLASH '14, pages 39-40, 2014.

[9] J. Tatibouet, A. Cuccuru, S. Gérard, and F. Terrier. Formalizing execution semantics of UML profiles with fUML models. In *MODELS 2014,* pages 133-148, 2014.

[10] J. Tatibouet, A. Cuccuru, S. Gérard, and F. Terrier. Towards a systematic, tool-independent methodology for defining the execution semantics of UML profiles with fUML. In *MODELSWARD 2014,* pages 182–192, 2014.

[11] G. Dévai, G.F. Kovács, and Á. An. Textual, executable, translatable UML. In OCL 2014, pages 3–12, 2014.

[12] Z. Micskei, R-A. Konnerth, B. Horvath, O. Semerath, A. Voros, and D. Varro. On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf. In OSS4MDE 2014, pages 31–41, 2014.