On the Executable Nature of Models

(Position Paper)

Eric Cariou, Olivier Le Goaer, and Franck Barbier
Université de Pau / LIUPPA, PauWare Research Group, BP 1155,
64013 PAU CEDEX, France
Email: {firstname.name}@univ-pau.fr

***Abstract*—Within the model-driven engineering field, the concept of "i-DSML" (interpreted Domain Specific Modeling Language) refers to executable models which are interpreted through an engine. While several works discussed the key ingredients of an i-DSML, few of them answered the original question: What is the class of models that are executable by nature and those that are not? This paper attempts to provide some answers by proposing two discriminating criteria: The possibility of defining execution steps and the reification of the behavior of the running system into the executed model. On this basis, we reconsider some well-known DSML and notice a paradoxical situation with UML diagrams.**

***Keywords*—MDE, model execution, i-DSML, xDSML**

I. Introduction

The Model-Driven Engineering (MDE) aims to bring models as productive artifacts for the software development. Having productive models means that they are directly the base for obtaining the final software system, for instance through the ability to generate code from them. The ultimate challenge within productive models field is to skip the implementation stage and that the models at design-time are fully reifying the whole system at run-time. This can be achieved through model execution. Recent initiatives at the OMG such as fUML[1] or ALF[2] enable to add an executable behavior for instance on UML[3] class diagrams that are basically static structures. In a more general way, the MDE enables to define its very-own modeling language dedicated to a specific purpose within the concept of DSML (Domain Specific Modeling Language). Therefore one can also define DSML for executable models. Such DSML are called i-DSML [5] for interpreted DSML or xDSML [6] for executable DSML.

Model execution has been widely studied and a consensus has been established on the constituent elements of an i-DSML and so, on how carrying out model execution. This answers the question: "How to build an i-DSML?". In this paper, we try to answer a symmetrical question that is: "If we are facing a model, can we determine if this is an executable model, or worded other way, if the DSML used is actually an i-DSML?". We are then looking for characteristics of the executable nature of a model.

In the next section, we recall the constitutive elements of an i-DSML. In section III, we give two criteria allowing to determine if a model is executable. These criteria are then applied on well-known DSML from the OMG.

II. Characterization of model execution

Model execution has been studied by several works, especially [2], [3], [4], [5], [6], [7]. All these works built a consensus on the rationale of model execution and how to design an i-DSML. Figure 1 summarizes our characterization from [4]. An i-DSML is a specific kind of DSML whose metamodel contains two types of elements: Elements called 'static' which describe the steady structure of a model, and elements called 'dynamic' which indicate the global model state at a given execution step. In the case of a state machine for example, the static elements are states and transitions while the dynamic elements are the current active state(s). A precise execution semantics is attached to an i-DSML. It specifies how to make evolving the model at runtime, acting only on the dynamic elements of the model under execution. Reconsidering the state machine example, it specifies how the transitions have to be fired according to both the current active state(s) and an incoming event, leading to modify the current active state(s). This execution semantics is implemented through an execution engine. The engine takes an executable model (conforming to an i-DSML) in input and is in charge of its interpretation, that is, making evolving it, thereby generating a sequence of execution steps (a.k.a an execution trace).

It is worthwhile mentioning that the aforesaid characterization assumes that the current state of a model under execution is stored in the model itself. This is not always the case, since knowledge about the current state can also be internally managed by the engine. As an example, PauWare[4] is an execution engine for UML state machines, written in Java. Nevertheless, the genuine UML specification did not planned to store in a state machine diagram what are the current active state(s). Consequently, the PauWare engine is responsible for this. To our opinion, storing the execution state inside the model rather than inside the engine has the advantage that it provides a self-contained execution trace. Thanks to that, it is possible to perform failure recovery or to apply verification techniques onto the trace.

As a side note, we identified a paradoxical situation about UML. Indeed, the genuine UML specification does not focus enough on the execution of the different types of diagrams it defined, and especially about the behavioral ones that yet are executable *a priori*. Reconsidering the state machine diagram, the UML specification defines a kind of a dynamic part and

[1]Semantics of a Foundational Subset for Executable UML Models (http://www.omg.org/spec/FUML/)

[2]Concrete Syntax for a UML Action Language: Action Language for Foundational UML (http://www.omg.org/spec/ALF/)

[3]Unified Modeling Language (http://www.omg.org/spec/UML/)

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

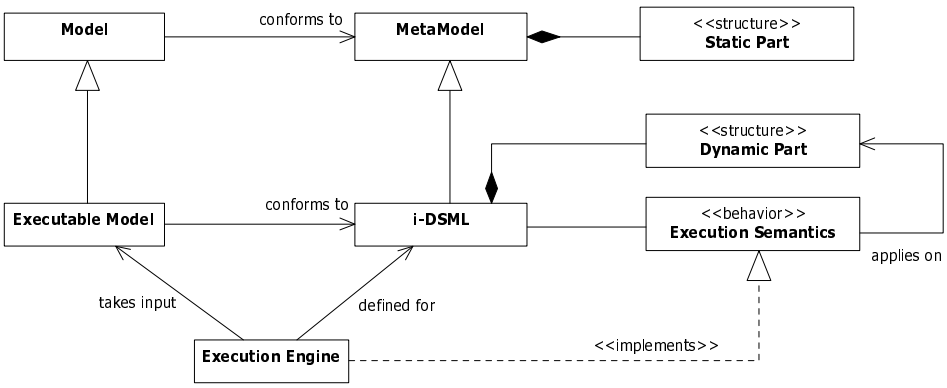

Figure 1. Conceptual framework of model execution

an execution semantics but only in an informal manner, in natural language. The dynamic elements are missing from the UML metamodel (in [3] we proposed an extension in that purpose). Conversely, the famous class diagram, whose execution is counter-intuitive at first sight, is curiously released with dynamic elements thereof: Those of the object diagram. Indeed, despite that the object diagram has not been invented with execution concerns in mind, it may serve to capture a current state of the instances of a class diagram, at a given moment of the future running system. These ideas of state and time are clearly expressed in the OMG's specification of the UML Object diagram[5]: "A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time".

III. Characterization of the executable nature of a model

The aforesaid characterization of model execution does not allow knowing explicitly what are models that can be executed, and those who cannot. This is rather a matter of intuition from the software engineer, sometimes giving rise to debate or controversy. In order to draw a clearer demarcation line, let us describe two criteria that seems enough to characterize the executable nature of a model:

- The behavior of the software system is located into the model;
- It is always possible to determine the execution next step and an initial state.

These two simple criteria tell us if execution semantics can be defined for a given DSML and hence if an engine can be implemented thereafter. In other words, they tell us if a DSML is an i-DSML.

A. Location of the system behavior

Let us assume some kind of software system which uses a model during its execution (at runtime). The question is how to know the role of this model against the overall behavior of the running system. We recall that a system performs "business actions". For example, an elevator system opens and closes doors, winds/unwinds the cable to reach a given floor. As another example, a travel booking system inserts customers data into database or call Web services provided by air transport companies. The behavior of a system determinates when these business routines have to be executed and under what conditions.

In this context, the question is whether the decisions to carry out these actions are impelled by the model itself or if they come from elsewhere, in which case the model is considered as mere input data helping to take the decisions. Reconsidering the examples above, the actions of the elevator may be triggered according to a set of states and transitions modeled through a finite state machine. Similarly, the calls to the various Web services may be orchestrated through a BPEL[6] or BPMN[7] model. In both cases, the behavior of the system is fully defined by the model. Conversely, in the spirit of *models@run.time* [1], the elevator system may leverage during its execution from a model about the extent of wear of the components or about daily uses, etc. Such kind of models ought to be queried by the business actions, but in no case to trigger them. Hence, these are not executable models. In the wake, if a model defines the global behavior of a system, then the software unit taking the latter as input can be referred as an execution engine.

B. Determinism of execution steps

As discussed in Section II, model execution consists in carrying out execution steps; each step making the model evolving from an active state to another one. The model must then intrinsically enable to pinpoint different execution states. However, being able to define the current state of the model is necessary, but not sufficient. Indeed, we highlighted that the object diagram can be seen as the current state of a class diagram although the class diagram is absolutely not executable. Unless it is associated and complemented with behavioral diagrams (sequence, state machines, ...) or fUML specifications, taken in isolation, a class diagram does not enable to determine how making evolving a related object diagram.

Consequently, the second criterion of the executable nature of models is to be able to carrying out execution steps. This

[5]http://doc.omg.org/formal/2000-03-01.pdf, section 3-20, page 278

[6]Web Services Business Process Execution Language (https://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm)

[7]Business Process Model and Notation (http://www.omg.org/spec/BPMN/)

feature can be found for example within activity diagrams, finite state machines or Petri nets. In all these types of models, modeling elements exist in that purpose, typically transitions that can be followed between states, activities or places. In some other cases, the "computing" of the next state can be made without dedicated modeling elements. For instance, for SBVR models[8], it is the execution engine that browses the entire rules set to determine the ones to trigger, as for any declarative language.

There is a step which stands out from others: The one that starts the model execution by placing it in its initial state. The model must then define a starting point. Making an analogy with programming languages, it can be seen as the equivalent of a `main()` operation defining what to do when the program is just launched. For finite state machines, activities diagram or Petri nets, the starting point is defined explicitly as special modeling elements exist for this purpose. Having an ending point is optional as some executions are not expected to end.

Having the capability to make the model evolving from a given running state to another one is a fundamental feature because, without it, it will be impossible to specify an execution semantics. Indeed, the goal of an execution engine is to carrying out the model evolution accordingly to an execution semantics.

C. Some examples of OMG's DSML

Modeling language	System behavior	Current state	Execution steps	Executable?
BPMN	Yes	External	Explicit	Yes
UML use case	Yes	Internal	None	No
UML class diagram	No	Internal	None	No
UML component diagram	No	Internal	None	No
UML state machines	Yes	External	Explicit	Yes
SBVR	Yes	External	Implicit	Yes
UML sequence diagram	Yes	External	Explicit	Yes

Table I. EXECUTABLE NATURE OF SOME DSML FROM THE OMG

Table I applies our criteria on some well-known DSML of the OMG, based on their OMG specifications[1,7,8]. For sake of simplicity, we consider each UML diagram as a DSML. A model is considered as executable if it reifies the behavior of the system and if execution steps can be defined. The current state of a model can either be defined as a full-fledged part of the model or in an external way.

Unsurprisingly, models defining structural software artifacts such as component or class diagrams are not executable. None of them enables to have execution steps. All UML diagrams classified as behavioral diagrams (state machines, sequence diagram, etc.) are unquestionably executable. However the OMG has never planned to add to them their dynamic part. For these models, the management of the current model state will be left to the responsibility of the execution engine. The specification of BPMN models suffers from the same limitation.

Paradoxically, some non-executable models such as class diagrams or use case diagrams have a dynamic part defined by their meta-model. Indeed, as seen before, for class diagrams, the object diagram plays the role of its dynamic part. For use cases, curiously, the UML meta-model defines the concept of use case instance that could be used as a dynamic part. However there is no way to determine how to execute a use case as the notion of execution step is totally missing in use cases. It is not possible to know which use case or in which order use cases must be executed. Moreover, the description of a use case is informal, written in natural language, and does not permit to execute a case on its own.

SVBR models are executable without internal dynamic part and have the particularity to define implicit execution steps. This is due to the declarative nature of this modeling language.

[8]Semantics of Business Vocabulary and Business Rules (http://www.omg.org/spec/SBVR/)

IV. CONCLUSION

In this paper, we first recalled what is an i-DSML from a technical point of view: An i-DSML defines a dynamic part within the meta-model, an execution semantics is specified and is implemented by an execution engine. Then, we studied, in a more "philosophical" way, the executable nature of models. We have identified two criteria allowing determining if a model is executable and consequently if the DSML it is conforming to, is actually an i-DSML: 1) the fact that the behavior of the running system can be reified in the model and 2) that the DSML enables to define execution steps during the model execution. These two features make possible to define an execution semantics and execution engines for this i-DSML. Finally, we applied these criteria of some known DSML of the OMG and showed a completely backward situation: All the executable models may not have their DSML defining a dynamic part while some non-executable models do have a possible dynamic part.

This characterization of the executable nature of a model is presumably not comprehensive but we guess that these two criteria are the most important to take into account. We will study more deeply several other i-DSML to make new criteria emerge.

REFERENCES

[1] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.

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

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

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

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

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

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