Executable Modeling for Reactive Programming

Franck Barbier[1] **& Eric Cariou**[1]

[1] Univ. of Pau - av. de l'université - 64000 Pau - France

Franck.Barbier@FranckBarbier.com

Abstract. After thirty years, it is reasonably time to critically look at Model Driven Software Development (MDSD). Who may nowadays claim that MDSD has been massively adopted by the software industry? Who may show numbers demonstrating that MDSD allowed/allows massive cost savings in daily software development, but, above all, software maintenance? This paper aims at investigating executable modeling as a balanced articulation between programming and modeling. Models at run-time embody the promising generation of executable models, provided that their usages are thought and intended to cost-effective software development. To envisage this not-yet-come success, this paper emphasizes expectations from the software industry about "reactive programming". Practically, executable modeling standards like the SCXML W3C standard or the BPMN OMG standard are relevant supports for reactive programming, but success conditions still need to be defined.

Keywords: Model Driven Software Development, Executable Modeling, Models at Run-Time, Reactive Programming.

1 Introduction

Since the rise of MDSD, "executable modeling" refers to the possibility of interpreting models via an "execution engine" [1] [2] [3]. Roughly speaking, from industrial experience, models as "abstractions" are not as readable (and thus understandable) as it appears. In other words, "abstraction" pushed forward as a panacea in the literature [4] rather means "approximation". So, as incomplete representations of systems, models hide second-class details (the good), but fail in providing fully controllable software assets from requirements engineering to programming/testing (the bad).

Executable models essentially are more intuitive models that express structures and behaviors of systems with greater clarity. In a nutshell, at design time, execution of models is simulation of behaviors. As for structures, simulation informs us

about the evolution, for example, of object links over time. So what? Executable modeling is just programming (*e.g.*, Java and its *Java Virtual Machine* as "execution engine"). Or, from another perspective, programming is a subtype of modeling when programs (as models) include implementation details. This seeming overlapping raises a worrying question: has MDSD reinvented "traditional" software development with no-value complication, no more? Indeed, it is extremely important to prove that MDSD provided/provides added value over its competing software development "methods".

1.1 Back to Jurassic programming?

By copying and adapting Winston Churchill's maxim, we may write: "Indeed it has been said that *MDSD* is the worst form of *Software Development* except for all those other forms that have been tried from time to time…" Translation: despite (known) defects, MDSD is a true engineering progress in software development. In fact, programming and (executable) modeling are not so confusing notions in spite of some "theoretical" overlapping (see prior paragraph). To justify this opinion, let us go into further detail through the concise analysis of four executable modeling frameworks:

• The Papyrus UML/SysML integrated modeling environment supports the *Action Language for Foundational UML* (ALF) for model execution. Models are classical UML/SysML models endowed with executable parts written in ALF. In this logic, there is, *a priori*, **no use at all** of any programming language (*i.e.*, ALF acts as a neutral "programming" language). Moreover, in the spirit of model transformation, code is derived from models without any tricky model adjustment when deployment concerns occur about specificities of the targeted platforms (Web, real-time…). This (open-minded) idealistic (even naïve) vision omits the reuse of software libraries that require significant imbrication between models and *Application Programming Interfaces* (APIs). To that extent, a key ALF evangelist recently told us that perspectives of ALF take-up (and thus take-off) are poor due to the misunderstood "sophistication" behind the systematic use of ALF. In short, nobody really wants to use ALF!

• jBPM (standing for *Java Business Process Management*) is an integrated BPMN (*Business Process Model and Notation*) modeling environment that supports Java (or JavaScript) as "action language". BPMN 2.x as executable modeling language plus Java as action language for expressing the content of BPMN (routing) gateways and tasks, makes jBPM a complete platform for executable modeling. Indeed, jBPM is a simplified form of MDSD in the sense that the target platform is *Java Enterprise Edition* (Java EE) only, and more precisely the WildFly application server, a Java EE-compliant deployment middleware. jBPM perfectly illustrates the fact that **programming and executable modeling occur at the same time**. Compared to ALF, there is no intrinsic complication (and investment) associated with the use of Java because, simply, everybody knows Java.

• SCION is the JavaScript support of *State Chart XML* (SCXML). Client-side (browser) or server-side (Node.js platform) Web applications benefit from being designed from Harel's Statecharts for which SCXML provides its own (rational) execution semantics. While SCION allows the graphical or textual (a devoted *XML Schema Definition* or XSD) expression of SCXML models, SCION-CORE acts as execution engine for SCION. The actions associated with state entries, exits and transitions must be written in JavaScript (to interact with the *Document Object Model* or DOM, for instance, in the browser). Again, programming and executable modeling appear as totally complementary **and not alternatives**.

• PauWare is similar to SCION apart from supporting the execution semantics of both SCXML and UML State Machine Diagrams. PauWare is an API and a self-contained Java library only. Developers may organize their code from complex state machines in the style of reactive programming [5].

So, it is time to admit that MDSD is not the way of making software that, as much as possible, dismisses programming activities. The later ones cannot be ignored because models cannot encapsulate all of the APIs and software libraries (in varied programming languages), which are the single access to modern deployment platforms. As an illustration, a BPMN script task is both a model piece and a suite of Java statements within jBPM. It is naturally expected to call any "external" code within such a task.

1.2 The economy of executable modeling

Model production is time-consuming and thus costly. In this logic, MDSD is a software engineering "method" (*i.e.*, "a way of making software") whose goal is, indubitably, fighting against code-only approaches, say, having no method at all, preferring DevOps or stressing agility (test driven software development that is often opposed to MDSD in the literature).

In a nutshell, from an economical viewpoint, one has to demonstrate that the return on investment is better with models. In other words, the time **lost** when constructing models definitely leads to the reduction of software cost prices **in all conditions**. Indeed, significant engineering efforts occur earlier with MDSD: intellectually, models, compared to code, require much thinking, much know-how. Consequently, software payers expect that models as software artifacts accelerate software finalization later on: the only way of "getting our money back".

From experience, is it really observed? In other words, nowadays, considering 1000 (randomly chosen and sizeable) starting software projects in any kind of business sector and any kind of technological context (Enterprise (payroll, shipping, supply chain…), Web, safety-critical…), how many will use MDSD? We would bet 10. A missing link therefore exists in the economy of MDSD…

1.3 Executable modeling in action

As written before, as approximations, models (**Fig. 1**, car sketch) do not conform to (generated) code (**Fig. 1**, real car) while the contrary should be true. Inevitable code adaptation at maintenance time (**Fig. 1**, repairers) breaks the initial mapping created at code generation (a.k.a. derivation) time. This is the main factor of MDSD failure because there is (and will be in the future) **no** MDSD tool capable of managing such mapping **at an industrial scale** when software masses pervade computers and above all clutter up developers' minds.

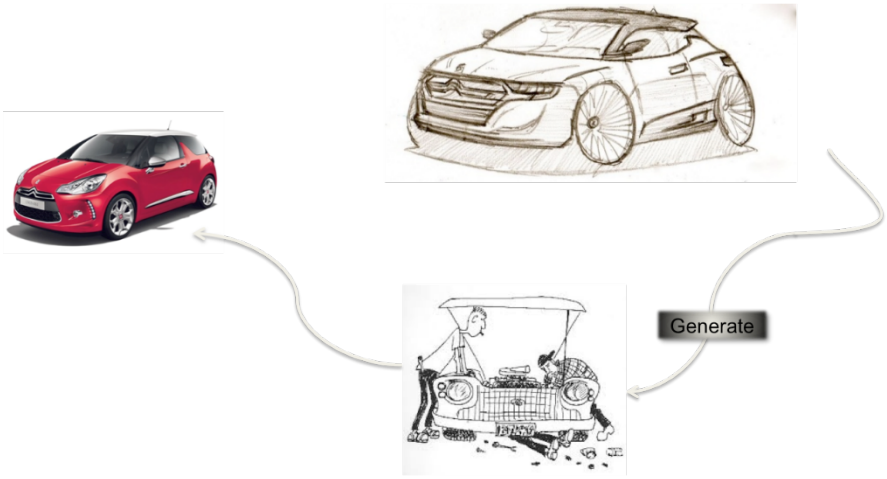

Fig. 1 MDSD recurrent pitfall

In its very deep nature, executable modeling does not impose model transformation. Instead, executable modeling looks for the better composition of code and models. Both are expressed in different languages, but these languages have precise well-defined roles, inner workings and dependencies. Practically, Harel's Statecharts with their high-end expressiveness may be (pre-)coded in JavaScript (SCION) or Java (PauWare). Then, data transformations only require algorithms and/or external calls that are encapsulated in actions triggered in Statecharts models. Models exist and persist at run-time [6]. There is no model transformation provided that model execution engines operate on the top of common programming languages compilers/interpreters.

As an illustration, **Fig. 2**, **Fig. 3** and **Fig. 4** show how to coordinate in PauWare any mobile app. with the Android battery management system.

First, **Fig. 2** shows the connection with the Android API. Building *a priori* models here creates **no value**. Worst still, models may be considered as crippling.

```
class EventSniffer extends android.content.BroadcastReceiver {
  @Override
```

```
    public void onReceive(android.content.Context arg0, an-
droid.content.Intent arg1) {
        if (arg1 != null && arg1.getAction() != null) {
            android.content.Intent intent = new android.content.Intent(arg0,
Android_energy_management_example.class);
            intent.setAction(arg1.getAction());
            arg0.startService(intent);
        }
    }
}
```

Fig. 2 Android battery management (notification subscription)

Next, **Fig. 3** is the realization of a model at run-time. Instead of having cluttering Java *if-then-else* control code, Harel's Statecharts formalism allows a rational code organization that is required for processing Android battery management system events.

```
public class Android_energy_management_example extends an-
droid.app.Service {
    EventSniffer _event_sniffer; // Link to the Java class in Fig. 2
    AbstractStatechart _Nominal_Energy_Level;
    AbstractStatechart _Critical_Energy_Level;
    ... // Other states here...
    AbstractStatechart_monitor
_Android_energy_management_example_state_machine;
    ...
    public void start() throws Statechart_exception {

_Android_example_state_machine.fires(android.content.Intent.ACTION_B
ATTERY_LOW, _Nominal_Energy_Level, _Critical_Energy_Level);

_Android_example_state_machine.fires(android.content.Intent.ACTION_P
OWER_CONNECTED, _Critical_Energy_Level, _Critical_Energy_Level);
        ... // Etc.
```

Fig. 3 Android battery management (notification reaction)

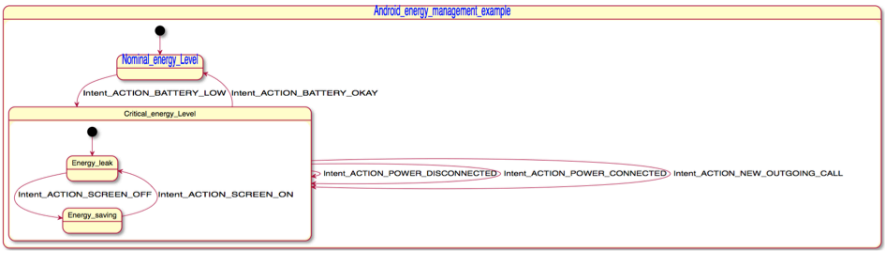

Fig. 4 Android battery management (model at run-time)

From a graphical viewpoint, **Fig. 4** is totally equivalent to the model in **Fig. 3**. This graphical model may automatically be derived from the Java code in **Fig. 3**. Indeed, executable modeling with models at run-time create a true bijection between models and code. In terms of efforts and elapsed phases, models may be created first ("classical" MDSD) **or not** (this example)… While the model in **Fig. 4** is an appropriate tool for test (simulation through graphical animation and execution trace) at design time, it may also be an ideal support for the (remote) administration of an energy-aware Android mobile app. when in production (*i.e.*, in users' hands). In contrast with model transformation, models at run-time are not substituted by code. **They are code**, but they cannot support and express all of the requirements and functionalities. So, the extra code that is not "models" to achieve these requirements and functionalities is similar to that in **Fig. 2**.

As an intermediate summary, we may write that modeling and programming are brothers-in-arms. The vision that uses modeling as means for code "burying" led to the unsuccessful spreading of MDSD in the past thirty years. Instead, modeling succeeds when developers do not distinguish modeling and programming activities: this is executable modeling with direct and smart integration of programming stuff in models. In this context, beyond a human adhesion to MDSD and its inherent tools, there are economical spinoffs as well; these are the basic expectations of a general-purpose software development "method" when facing the recent "event bombing" challenge.

2 Executable modeling for reactive programming

There is a great opportunity for MDSD to bounce back from the idea of "reactive programming" promoted, in particular, by *The Reactive Manifesto* (**Fig. 5**). In this figure, the **entailing** principle is "Message Driven" meaning that contemporary computing views applications' content as complex message exchange and coordination rather than monolithic data transformation.

Messages are point-to-point event communication and processing. As for message data, they are brought by events. Of course, events also are the foundational

abstraction paradigm that makes the idea of "Message Driven" viable at modeling time.

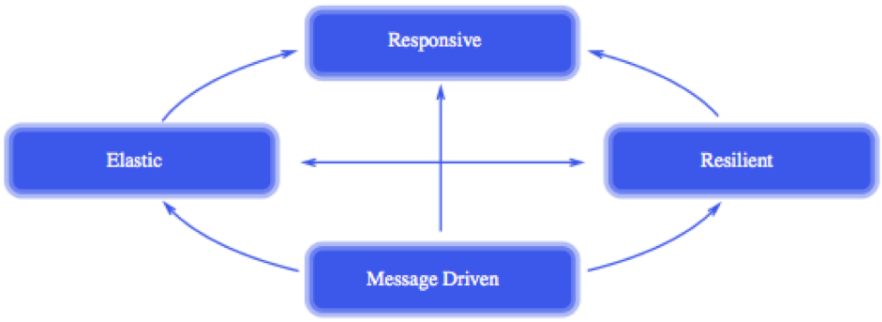

Fig. 5 The Reactive Manifesto four principles

2.1 Reactive programming at a glance

Intuitively, the Internet/Web of Things moves us to some unencountered asynchronicity by which software components in applications have to re-create (*i.e.*, to create in a postponed way, at run-time essentially) unanticipated synchronization through very intensive event communication and processing, *i.e.*, the "event bombing" (**Fig. 6**).

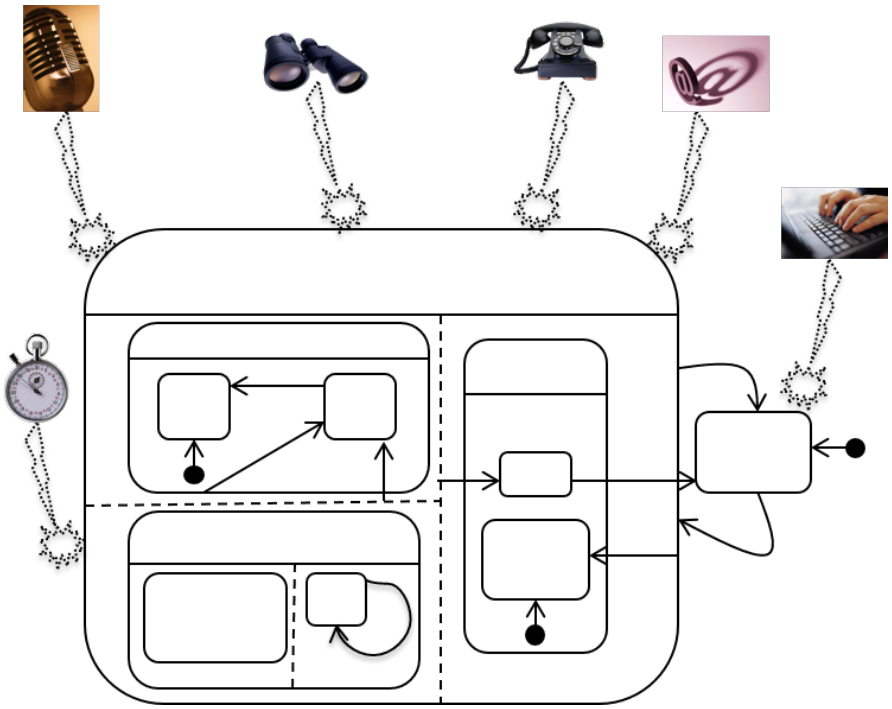

Fig. 6 Event bombing

The key idea in **Fig. 6** is that "Message Driven" middleware is/will be the norm through products, say, the famous Node.js software development framework or any other recognized similar computing infrastructure like ReactiveX. Namely, **Fig. 6** shows that "Message Driven" software development imposes a strict internal organization of the inside of software components. As a complex (modeled) state machine, this organization is then capable of creating components that can serve client requests ("events", "messages", whatever…) on a concurrent and reliable basis.

So, reactive programming refers to computing frameworks that manage event production and distribution (queuing, routing, fault recovery…). Nonetheless, it is also **a programming style** in which event communication and processing benefit from having a direct and smart support. In this scope, behaviors of software components may then be instrumented by concrete models at run-time as illustrated in **Fig. 2**, **Fig. 3** and **Fig. 4**. While the model is these three figures is somehow trivial, **Fig. 6** adds the idea that models may express very complex behaviors imposed by the intrinsic idea of event bombing (great variation of event types, great flow rate of event occurrences…). For example, the core parallelism construct offered by Harel's Statecharts (a.k.a. "state orthogonality") can wisely replace multithreading programming statements. These would surely lead to a nightmare at maintenance

time (remind the initial point of this paper about the MDSD incapacity of offering truly maintainable models).

5 Conclusion

It is never too late to reach maturity, the (accomplishable?) MDSD quest for the forthcoming years. Honesty leads us to assert that even though modeling is "common engineering" for "dynamical" (real-time, safety-critical…) systems, enterprise computing (more than 90% of the existing software) has thrown MDSD overboard. Contemplative modeling, the anti-thesis of executable modeling, was/is the demotivating factor for daily practitioners. By denying this fact, MDSD promoters misled/mislead people behind labyrinthine MDSD: *Model Driven Architecture* (MDA), model transformation and so on.

This papers show that programming and modeling are brothers-in-arms, provided that their relationship is well-defined and based on executable software assets. In the jungle of software development "approaches", fashion highlights agility, DevOps, Kanban, Lean Management… In this universe, MDSD looks like Taylorism in the '30s: an excessively codified approach that limits creativity in general. This results from (contemplative) models as totally nonflexible software matter. Instead, models at run-time are intended to attenuate, even reverse, this fact and consequential feeling.

Acknowledgements

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*) 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. Harel D, Gery E (1997) Executable Object Modeling with Statecharts. IEEE Computer 30(7):31-42.
2. Riehle D, Fraleigh S, Bucka-Lassen D, Omorogbe N (2001) The Architecture of a UML Virtual Machine. Proc. 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications:327-341.

3. Mellor S, Balcer S (2002) Executable UML – A Foundation for Model-Driven Architecture, Addison-Wesley.
4. France R, Rumpe B (2013) The evolution of modeling research challenges. Software and Systems modeling 12(2):223-225.
5. Barbier F (2016) Reactive Internet Programming – State Chart XML in Action, Morgan & Claypool.
6. Blair G, Bencomo N, France R (2009) Models@run.time. IEEE Computer 42(10).