Energized State Charts with *PauWare*

Franck Barbier
Univ. of Pau, France
BP 1155
64013 Pau CEDEX
Franck.Barbier@FranckBarbier.com

Olivier Le Goaer
Univ. of Pau, France
BP 1155
64013 Pau CEDEX
olivier.legoaer@univ-pau.fr

Eric Cariou
Univ. of Pau, France
BP 1155
64013 Pau CEDEX
Eric.Cariou@univ-pau.fr

ABSTRACT
Persuading software engineers to systematically use on a large scale, a modeling language like SCXML greatly depends upon suited tools. At the very end, only financial concerns prevail: productivity increases due to modeling. Otherwise, modeling stops. This paper comments on *PauWare*, a Java technology that aims at ameliorating the daily practice of State Chart modeling. Beyond design, *PauWare* is based on models@runtime to continuously benefit from models when applications are in production.

Author Keywords
Model-Driven Development; State Charts; Executability.

ACM Classification Keywords
D. Software; D.2 SOFTWARE ENGINEERING; D.2.2 Design Tools and Techniques.

General Terms
Design.

INTRODUCTION
Since the takeoff and development of *Model-Driven Development* (MDD) in the spirit of the *Unified Modeling Language* (UML), modeling take-up remains fairly low. From experience in software industry, mental blocks persist. Developers rather prefer coding than modeling. Being graphical and/or textual, the situation is the same for all kinds of models; accordingly, modeling techniques are still often considered as supports for only producing software documentation.

The reason is "abstraction". Even though abstraction allows sound design principles like "separation of concerns", "incrementality" or "early design error detection", it is also "far from the processor". Latest software tuning is often incompatible with "idealistic worlds" in models. Over time, models and code diverge, leading developers to throw models overboard as soon as possible.

As a modeling language, SCXML spreading may stumble over these well-known "hurdles". The quality of surrounding well-integrated tools (editors, checkers, simulators, code generators…) plays then a crucial role for the success of a modeling language. For example, *Eclipse Modeling Framework* (EMF) [1] has made UML manageable in XML (declarative aspects) and Java (imperative statements as model transformations). In another style, Yakindu (statecharts.org) for State Charts allows model simulation and code generation. Both tools are actual proofs about moving models one step beyond: models benefit from being executable (or "interpretable"). Nonetheless, this idea is not new. In [2] or [3], the intention to offer an executable UML or a definitive virtual machine for the overall UML does not result in something tangible at this time.

This paper presents the *PauWare engine* Java library (PauWare.com) to design ordinary software applications from executable State Charts. From the origin, this library obeys to the execution semantics of UML (with safe homemade corrections), which is, in our opinion, very close to that of SCXML. Regarding theoretical concerns, *PauWare engine* is a research prototype mainly used for carrying out experiences in software adaptation [4]. Otherwise, the two key industrial realizations from *PauWare engine* are the implementation of a service mediator in the ReMiCS project (remics.eu) and a model debugger in the BLU AGE® MDD tool suite (bluage.com).

This paper discusses long experience and practice in State Chart modeling with concise consideration on associated tools, industrial usages, feelings and feedbacks as well on the high necessity of models with greater attractiveness and power of conviction.

REVISITING MDD
Over years, despite a certain know-how engraved in *PauWare engine*, it is still difficult to convince people to switch from prehistoric coding practices to relevant standards like SCXML. Open proven *Computer-Aided Software Engineering* (CASE) tools are important to guarantee progresses. In this context, code generation from SCXML models to *PauWare engine* API continues to raise a squaring-the-circle problem: the gaining of SCXML models is above all an often sizeable modeling effort, especially when requirements are numerous and complex, leading to labyrinthine State Charts. In other words, CASE tools cannot be substituted for human intelligence.

EICS'15 SCXML workshop, June 23, 2015, Duisburg, Germany.

State Chart execution with *PauWare engine* or direct interpretation with tools like Commons SCXML (commons.apache.org/scxml) supposes the prior nontrivial design of complete and ready-to-use SCXML models. Similar to code writing, modeling is error-prone with limited possibilities of testing intermediate designs.

To address these issues, the key idea is to give more latitude to software engineers in seamlessly navigating between models and code. Namely, "hiding" modeling activities can be a sound design principle. Concretely, once *PauWare engine* API under control, software engineers can express State Charts in Java with a very reduced set of classes/interfaces that easily and straightforwardly manage compound/leaf states, state machines and any kind of structuring: state nesting, state exclusiveness, state orthogonality, transitions, guards and actions. Other constructs of *PauWare engine* API are linked functions ("fires" and "run_to_completion" essentially).

In fact, there is no great distinction about dealing with SCXML or *PauWare engine*. Code generation may produce *PauWare engine* API code from SCXML source. SCXML models may also be derived from *PauWare engine* API code.

For example, here is a SCXML source sample extracted from the reference *Barbados Crisis Management System* case study (franckbarbier.com/PauWare/BCMS). States are in blue while events are in red:

```
<state id="Route_for_fire_trucks_development"
initial="Route_for_fire_trucks_to_be_proposed">

<final id="End_of_route_for_fire_trucks_development"/>

<state id="Route_for_fire_trucks_approved"/>

<state id="Route_for_fire_trucks_to_be_proposed">

<transition event="route_for_fire_trucks"
target="Route_for_fire_trucks_fixed"/>

</state>

<state id="Route_for_fire_trucks_fixed">

<transition event="FSC_agrees_about_fire_truck_route"
cond="!In('Route_for_police_vehicles_approved')"
target="End_of_route_for_fire_trucks_development"/>

<transition event="FSC_agrees_about_fire_truck_route" cond="!
In('Route_for_police_vehicles_approved')"
target="Route_for_fire_trucks_approved"/>

<transition event="FSC_disagrees_about_fire_truck_route"
target="Route_for_fire_trucks_to_be_proposed"/>

</state>

</state>
```

The corresponding *PauWare engine* code is as follows:

```
state_machine.fires(route_for_fire_trucks,
Route_for_fire_trucks_to_be_proposed, Route_for_fire_trucks_fixed);

state_machine.fires(FSC_disagrees_about_fire_truck_route,
Route_for_fire_trucks_fixed, Route_for_fire_trucks_to_be_proposed);

state_machine.fires(FSC_agrees_about_fire_truck_route,
Route_for_fire_trucks_fixed, End_of_route_for_fire_trucks_development,
this, "in_Route_for_police_vehicles_approved");

state_machine.fires(FSC_agrees_about_fire_truck_route,
Route_for_fire_trucks_fixed, Route_for_fire_trucks_approved, this,
"not_in_Route_for_police_vehicles_approved");
```

In this Java code, transitions are simply connected to source and target states. Events are later processed as follows:

```
public void route_for_fire_trucks() throws Statechart_exception {

state_machine.run_to_completion(route_for_fire_trucks);

} // Etc. Other events here…
```

As for SCXML conditions:

```
public boolean in_Route_for_police_vehicles_approved() throws
Statechart_exception {

return
state_machine.in_state(Route_for_police_vehicles_approved.name());

}
```

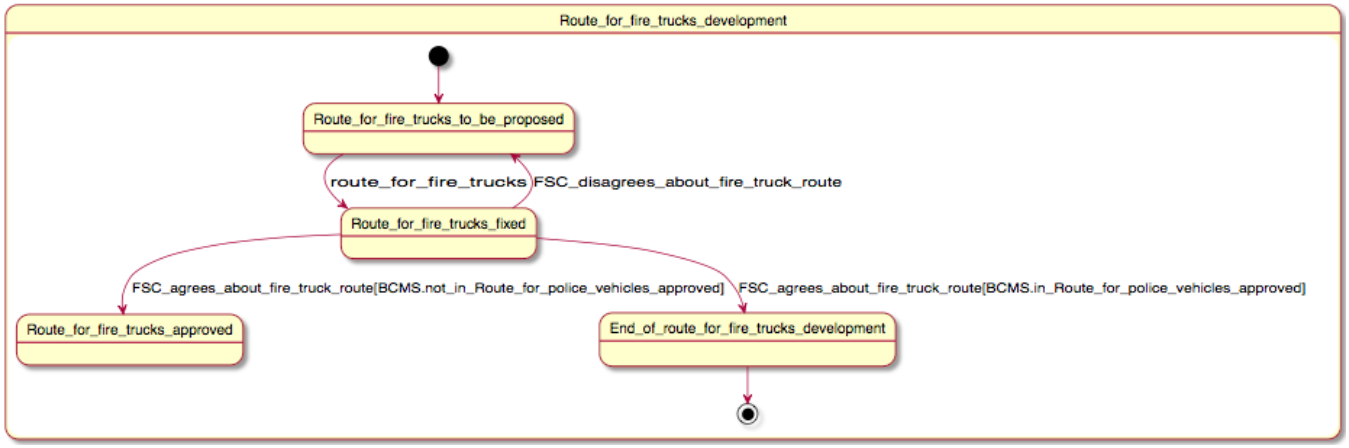

Figure 1. *PauWare view* look & feel (extract from *Barbados Crisis Management System*).

Testing through simulation at design time in particular relies on a third-party tool: *PauWare view*. *PauWare view* is an addon for *PauWare engine*. *PauWare view* dynamically generates one or more instances of State Charts in SVG

format by taking advantage of the PlantUML Java library (plantuml.sourceforge.net). *PauWare view* displays and simulates instances of State Charts in Web browsers in an asynchronous way (Figure 1). Any *PauWare engine* application communicates through Web sockets the discretized status of some or all of its running state machines. This logically results from the processing of event occurrences in run-to-completion cycles. Since applications have their own event processing frequency (for instance, a highly interactive application may be "bombed" by event occurrences), *PauWare view* acts as a buffer for displaying these occurrences in a human readable manner (refreshes are adjustable between 1 sec. and 5 sec.).

MODELS@RUNTIME

Even though *PauWare view* can be rightly viewed as a model testing tool at design time, its main purpose is run-time observation, even control in case of adaptation. The animation of State Charts by means of *PauWare view* in Web browsers is more than the simulation of models in the sense that these models are abstract software artifacts. Here, "abstract" precisely means that models mimic the grand characteristics of the final software, but all low-level details are not yet presented.

Instead, *PauWare view* is plugged in the application **in production** with, often, end-users being the source of event occurrences through GUIs. Running state machines may possibly be embedded in devices with system-oriented events (*e.g.*, battery events in an Android application [5]) or they can power *Enterprise JavaBeans* (EJBs) in large-scale SOA applications.

Keeping or not *PauWare view* at run time is a question of application administration in the spirit of the Java console. The latter aims at tracing, even controlling, operating applications. In all cases, cutting *PauWare view* off from *PauWare engine* is no effort. Performance issues for example may justify such a cutting even though *PauWare view* may run on other machines thanks to Web sockets.

Models@runtime [6] is the major source of inspiration for *PauWare* technology. No significant distinction is made between coding and modeling. Modeling is just disciplined coding to create higher intelligibility in the code by means of persisting models. Consequence is higher software quality, but nothing new under the sun: these are just software engineering entrails, *i.e.*, maintenability, reusability and reliability naturally increase.

WEAKNESSES

- With the exception of Java, there is no devoted mechanism in *PauWare engine* to write the bodies of actions launched in reaction to events or as entry/exit actions of states. The same applies for guards that are embodied by Boolean Java methods (see above). SCXML has a rich and relevant language-neutral approach with ECMAScript or, instead, by offering varied supports for different programming languages. Actions in *PauWare engine* stress data transformations in avoiding any control flow, which, in essence, is under the aegis of State Charts.
- PlantUML has drawing restrictions in the sense that it is not able to manage arrows (*i.e.*, transitions) that cross, from inside or outside, container states. *PauWare engine* does not have this embarrassing limitation, which confines *PauWare view* to specific forms of State Charts only. As an illustration, Figure 2 shows a model, which cannot be simulated at design time (and, mechanically, controlled at run time).

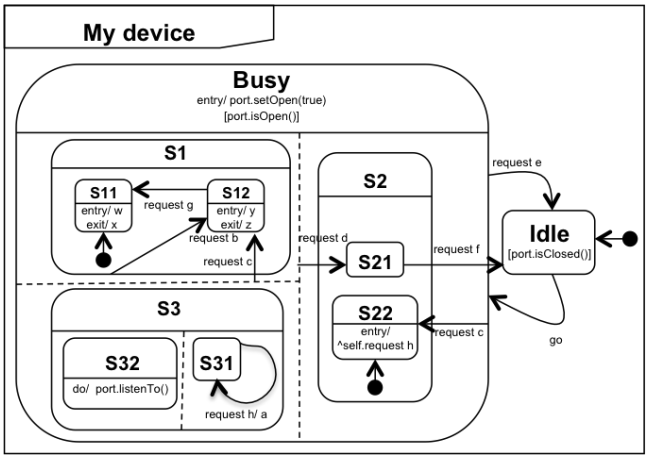

Figure 2. State Chart with numerous factorized transitions from/to superstates to/from substates.

The model in Figure 2 is simply and directly expressible in SCXML apart from proprietary UML constructs: "do/" UML notation for activities and state invariants between brackets (both are supported by *PauWare engine*). At run time, *PauWare engine* seamlessly executes the model in Figure 2, but, again, no behavior visualization is possible through *PauWare view*.

This problem can be bypassed with alternative PlantUML-compliant models having the same business semantics, but such models tend to accidentally become more complicated. Beyond, such an approach is dubious because tools serve modeling. It would simply be erroneous to envisage anything else.

- *PauWare* prompts software engineers to become model supporters with a kind of "extreme modeling" style. Indeed, "Write little matter-Compile-Test" is the rule in extreme programming: in short, tests drive development. However, such an approach is not unanimously known as a proven productive software development method when several stakeholders

are involved. A debatable question is the fact that MDD is recognized (or not?) as disruptive with respect to "ordinary" software development practices. Breaking requirements and specifications into modular pieces is normally favored by modeling. State Charts have intrinsic characteristics for being these pieces. This debate is outside the scope of this paper, but it is interesting to point out that State Chart expression is systematically preceded by an upstream significant modeling activity that is not readily aligned with *PauWare* design style.

STRENGTHS

- Distribution through Web sockets allows the remote run-time observation, even control (or self-control in case of self-adaptation [4]) of *PauWare engine* applications everywhere. Fruitful experiences relate to the Java Embedded technology. Running state machines are embeddable as a *System on Chip* (SoC) using, for instance, the Raspberry PI hardware. State Chart behavior visualization then becomes extremely informative for electronic/software engineers who have experience in only having "physical" perceptions of the SoC's behavior in a given real-word context. With reasonable effort, hardware-oriented events can be "mounted" on models animated in Web browsers.

- Without escape routes, crowded State Charts (due to challenging requirements) are both natural and difficult to read (to understand accordingly). *PauWare view* efficiently addresses combinatory issues. There is a kind of roundtrip engineering between code and (visualized) State Charts that are two distinguished viewpoints of the same thing. Typically, the suppression of useless model complexity often leads to code compaction/rationalization. Practice shows that the divergence between code and models in "traditional" MDD does not occur here.

- As already mentioned, models@runtime constitute an underlying appropriate support to keep control on running applications. For example, "runtime mutation" is recognized useful in [7] for debugging State Charts. Usually, code arises from specifications. Here, State Charts may derive from Java code and vice-versa. There is no effective upstream/downstream dependency between the two. *PauWare view* for example is able to take snapshots of (at rest or active) State Charts for software documentation production: a kind of "upside down software engineering".

CONCLUSION

In our opinion, compared to UML, SCXML succeeded in only keeping the true substance of the original Harel's Statecharts. In this paper, we defend the idea that a bi-layer approach is wrong. We mean: the classical MDD cycle in which code comes into being from models and code is, **later on**, enhanced with implementation details (*i.e.*, platform-specific information) that, in essence, do not belong to models because of their abstract nature, is a strong factor of MDD weakening and consequential rejection. A renewed MDD is possible if and only if models and code share a better articulation as offered by *PauWare*.

In this scope, the evolution of Commons SCXML is quite sound with the principle of an "expression language engine" in charge of parsing and evaluating imperative statements (typically, action bodies between the <*script*> and <*/script*> tags). The possibility of using Groovy for instance as this expression language allows the controlled mixing of code and models as done in EMF, Commons SCXML, *PauWare* and, probably, forthcoming modeling environments.

ACKNOWLEDGMENTS

PauWare has been partly funded by the European Commission. All authors gratefully acknowledge the grant from the European Commission through the ReMiCS project (remics.eu), contract number 257793, within the 7th Framework Program.

REFERENCES

1. Steinberg, D., Budinsky, F., Paternostro M. and Merks, E. *EMF - Eclipse Modeling Framework, Second Edition*. Addison-Wesley, 2008.
2. Mellor, S. and Balcer, S. *Executable UML – A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.
3. Riehle, D., Fraleigh, S., Bucka-Lassen, D. and Omorogbe, N. The Architecture of a UML Virtual Machine. *Proc. 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press (2001), 327-341.
4. Barbier, F., Cariou, E., Le Goaer, O. and Pierre, S. Software adaptation: classification and case study with State Chart XML. *IEEE Software*, in press (2015).
5. Le Goaer, O., Barbier, F., Cariou, E. and Pierre, S. Android Executable Modeling: Beyond Android Programming. *Proc. 2014 International Workshop on Mobile Applications* (2014).
6. Blair, G., Bencomo, N. and France, R. Models@run.time. *IEEE Computer 42*, 10 (2009).
7. Junger, D. Transforming a State Chart at Runtime. *Proc. Engineering Interactive Systems with SCXML Workshop* (2014).