A software development process based on UML state machines

Eric Cariou*, Léa Brunschwig†, Olivier Le Goaer*, and Franck Barbier*

*LIUPPA, Université de Pau et des Pays de l'Adour, E2S UPPA
B.P. 1155, 64013 PAU CEDEX, France
{Eric.Cariou, Olivier.LeGoaer, Franck.Barbier}@univ-pau.fr
†Universidad Autónoma de Madrid
Madrid, Spain
lea.brunschwig@uam.es

***Abstract*—We propose a model-based software development process based on UML state machines. State machines are executable models and such models offer the advantage to capture the behavior of a system at a high-level of abstraction. Besides, the business parts of the system can be specified and weave onto the executable models, applying a good separation of concerns. While advanced standards such as fUML enable to define the complete contents of an application at the model level, it leads to too much complexity and prevents flexibility in the use of existing code. For these reasons, we propose an intermediate and pragmatic approach where a UML state machine is compiled onto Java code for our lightweight execution engine PauWare. The business parts of the application are then implemented in standard Java.**

***Index Terms*—Model-driven engineering, software development, executable models, code generation, UML state machines, PauWare**

I. INTRODUCTION

In 2000, the Object Management Group (OMG) launched the Model-Driven Architecture (MDA) initiative[1]. Its main goal was to shift the complexity and the focus of software development from code to model: models should be the most complete as possible to define the precise behavior of an application. MDA separates business and application logic from underlying platform technology and through model transformations and code generation, the final code of the application should be automatically obtained. The goal is, if new implementation technologies appear, that the adaptation of the application to these technologies is "simply" made by regenerating the code from the business logic that does not change.

As being an initiative of the OMG, the considered models are based on its modeling standards: UML [1], OCL [2] and the MOF [3] for defining new modeling languages. Some years later, the Eclipse Modeling Framework (EMF) [4] became the de facto modeling environment for the Model-Driven Engineering (MDE). Its Ecore meta-meta-model is the reference implementation of the OMG's MOF and a large variety of tools for editing models, verifying them, transforming them and generating code from them have been developed.

However, the "holy grail" of MDA can only be achieved if models are sufficiently precise to automatically generate the complete running code (in Java for instance) of the designed application. The models must capture the execution behavior of the application. Concretely, the models must contains the equivalent of the generated code within the UML diagrams (or any other kind of models if one defines his own modeling language). For instance, if one defines a bank account class in a UML class diagram with a `void withdraw(float amount)` operation, the signature of the operation in the class is not sufficient: it is required to specify that the call of this operation subtracts the amount passed as parameter from the balance attribute of the class. If the implementation of this operation is straightforward using a programming language, it becomes difficult and even impossible to define it using only standard UML diagrams. To fill that void, the OMG's fUML specification [5] "defines a basic virtual machine for the Unified Modeling Language, and the specific abstractions supported thereon, enabling compliant models to be transformed into various executable forms for verification, integration, and deployment". fUML enables to define the execution semantics of elements of UML diagrams through dedicated activity diagrams but with the concrete textual syntax ALF [6], one can also write "code" in the diagrams. This code is abstract in the sense that it does not rely on any specific programming language. Now, it is possible to model the behavior of our withdraw operation using fUML. Recently, fUML has been extended to define the execution semantics of UML composite structures [7] and UML state machines [8]. All these specifications are notably implemented within Papyrus, an EMF-based modeling environment[2].

The full-model approach of an application implementation offers the expected benefits: independence of implementation technologies, no explicit manual coding phase where bugs and errors can be introduced compared to the specification (the model is by definition the final system), early detection of problems (the model can be simulated before generating the code), ... However, as we pointed out in [9], [10], it also raises

[1]https://www.omg.org/mda/

[2]https://www.eclipse.org/papyrus/

several questions and problems. How to integrate in the model existing code or that must developed with a specific IDE? How to make your software engineers moving from coding with programming languages to design and code with models? Does the full-model approach scale when developing large applications? For these reasons, we propose in this paper an intermediate and pragmatic approach where only a part of the application will be obtained from the models. Such models are executable models that define the behavior of the system. In the paper, our approach is applied to UML state machines. For executing state machines, we provide the PauWare engine implemented in plain Java. Then, business operations implemented apart in Java are weaved onto a PauWare Java state machine and this produces the final and complete application. This enables implementing a part of the code of the system using a regular programming language with any IDE and to rely on existing libraries. Another flexibility point we provide is to be able to define a UML state machine diagram with your favorite UML modeler and then to automatically generate the equivalent code for the PauWare engine.

The rest of the paper is organized as follow. Section II recalls the principles of model execution. Section III defines our model-based software process based on UML state machines and the weaving of external business operations. Before concluding, we discuss related work in section IV.

II. Model Execution

In software engineering, the separation of concerns is a fundamental principle that allows building a software with separate parts, thereby improving their maintainability and evolutivity. Executable models are good potential representatives of this principle since they capture the behavior of a software intensive system, that is, when, why and how calling business operations, while the latter are specified apart. For example, an elevator has its own firmware responsible for opening and closing doors, winding/unwinding the cable to reach a given floor. The actions of the elevator may be triggered according to a set of states and transitions modeled through a UML state machine. As another example, a travel booking system running on a server inserts customers information into a database or call Web services provided by air transport companies. The behavior of such a system specifies when these business routines have to be executed and under what conditions. The calls to the various Web services may be orchestrated through the BPMN formalism [11].

The control flow of an application implemented with regular programming languages can rapidly lead to a spaghetti code where methods are calling each other depending on "if then else" or "switch case" statements. An executable model offer the ability to specify the behavior of the system at a higher level of abstraction with a dedicated formalism and often graphically. Changing or fixing the behavior of the system consists only in changing the executable models.

All the dynamic diagrams of UML (state machine, sequence, activity diagrams,...) are by nature executable even if their execution semantics is not well defined [12]. This has been partially fixed with the recent specifications of fUML [5], PSCP [7] and PSSM [8]. MDE offers also the ability to define executable DSL (Domain Specific Language). A lot of a tools such as GEMOC have been developed for this purpose [13].

Such models are said executable because they have a direct correspondence with the running code of the system. Either the model is interpreted, in the sense that an execution engine reads the model and makes it evolving and calls the associated business operations, either the model is compiled into code. In our software development process, we use the second variant: the target technology of the generated code is Java for our PauWare state machine execution engine. PauWare enables to program in plain Java a UML state machine. Thereby, the associated business part is also implemented in Java.

III. Model-Based Development Process

A. Overview of the process

We propose the following software development process based on UML state machines and the PauWare tools:

1) Define with your favorite UML modeler the state machine of the application. Add the signatures of business operations on states and transitions.
2) Generate the code of the state machine for the PauWare API. At this time, without implementing the business operations, you can simulate the state machine to ensure that the reified behavior is correct. If not, you fix the state machine diagram in the UML modeler and regenerate the code. The operations that can nevertheless be required at this stage are the guards of the transitions (they are implemented with plain Java methods, as it will be shown in the following of the paper).
3) Develop apart in Java the business operations.
4) Weave the business operations onto the state machine. It concretely only requires to define a class and/or instantiating the right objects that embed these operations.
5) Execute your application that is now complete.

This process allows modeling on one side the behavior of the system through a state machine and on the other side implementing apart the business operations in Java. This enables to reuse easily legacy code or existing libraries and avoids a too big complexity in the model as all the algorithmic or technical parts are directly implementing in Java that is the most suitable way to define them.

The rest of the section will describe how to implement state machines in PauWare and weave business operations onto them. As an example, a state machine defining the behavior of a microwave oven is presented. Another simpler example of a stack is given for describing some important details related to the parameters and returned values of the business operations.

B. A microwave oven example

Figure 1 shows the screenshot of a UML state machine specifying the behavior of a microwave oven. The state machine diagram has been designed using Modelio (open source version 3.8). The microwave can be in two main states depending on the state of the door: open or closed. When the

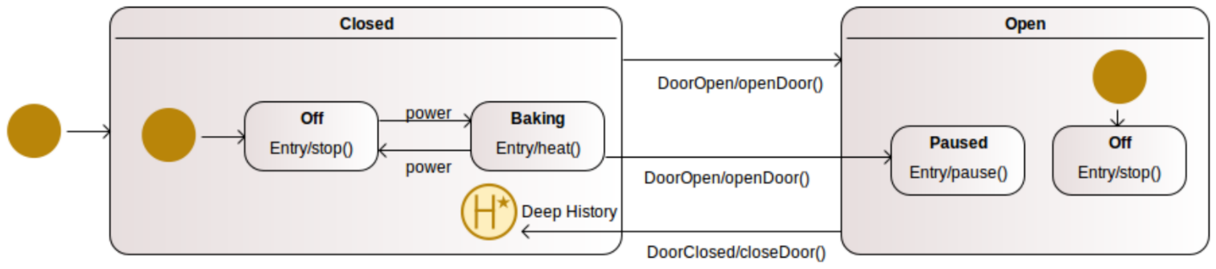

Fig. 1. UML state machine diagram of a microwave oven designed with Modelio

door is closed, the power button allows a cycle from baking to putting the microwave off. When opening the door, if the microwave was baking, it gets in a pause mode. Otherwise, it gets in off mode. Closing the door leads to come back in the previous mode when the door was closed, either baking or being off: this is specified thanks to the history state of the composite state "Closed".

Some business operations are attached to the elements of the state machine: each primitive state defines an entry action ("heat" for the state "Baking" for instance) and transitions can have an associated operation ("closeDoor" for instance for the transition associated with the "DoorClosed" event). In the UML model, these operations have been defined in classes of a UML class diagram (not represented on the figure).

C. Programming models with PauWare

PauWare[3] [14] is a tool enabling to execute UML state machines in plain Java programs. It is composed of two elements:

- an API defining a set of classes for building/programming a UML state machine associated with business operations. The API offers almost all the state machine features of the UML specification [1];
- an execution engine that carries on events, makes evolving the active states and executes the business operations.

PauWare can be used with any Java platform. There exists a version for Java ME and Java EE, both compatible with Java SE, as well as a boilerplate code for Android or the NAO robot. Programing with the PauWare API can be done with any Java IDE as it requires only to import a JAR file. PauWare is a lightweight tool, this file has a size of only 100 kB.

The listing of Figure 2 is the PauWare code of the microwave state machine. This code has been entirely generated by the PauWare code generator but it is exactly similar to the code that one developer would have written by hand to implement the state machine. The code generation has the advantage to generate a code that is readable and maintainable by human developers.

Mainly, the PauWare API defines classes for building the states and the transitions of the state machine. Through this example, the main features of PauWare can be presented:

- line 4, the object representing the state machine is defined;
- lines 7 to 12, the objects representing the 6 states of the state machine are defined;
- line 15, the "business object" is defined (and instantiated line 18). This is the object on which the business operations will be called. The class of the object can be manually changed and several objects of different classes can be used if required;
- line 19, the state "Off" of the composite "Closed" is instantiated. It is marked as the input state of its composite (line 20) and a business operation is associated as an entry action (line 21): on the object `bo`, the "stop" method will be called without parameter (`null` value). The way to associate business operations (or guards) with the state machine elements is always made in this way: a Java object, the name of the method and the array of parameters. The name of the method will be used to make a dynamic call on the object thanks to the reflective mechanisms of Java. Concretely here, the `BusinessObject` class implements a method with the signature `public void stop();`
- lines 29 and 30, the two composite states are created using the "xor" operator between their respective internal states (it is not used in the example but in a similar way, the "and" operator will define parallel regions);
- line 32, a deep history state is added to the "Closed" state;
- line 33, the structure of the state machine is built by composing the two composite states as exclusive;
- line 37, the method `start()` defines the transitions of the state machine with the methods `fires(...)` on a state machine object. Contrary to the states, here, there is no explicit class for defining a transition. More precisely, such a class has been added in the last version of PauWare but the code generator is working for the previous version;
- lines 40 and 41, basic transitions are defined with the minimal set of mandatory elements: the name of the event ("power") followed by the source state and the target state of the transition;
- lines 38, 39 and 42, transitions are defined and associated with a business operation. For instance, on line 38, the transition is associated with the `openDoor()` method on the `bo` object;
- on the state machine diagram of Figure 1, a transition

[3]https://www.pauware.com

```java
public class Microwave_StateMachine {

    // State Machine
    protected AbstractStatechart_monitor Microwave_StateMachine;

    // States
    protected AbstractStatechart Closed;
    protected AbstractStatechart Off_in_Region_in_Closed;
    protected AbstractStatechart Baking_in_Region_in_Closed;
    protected AbstractStatechart Open;
    protected AbstractStatechart Paused_in_Region_in_Open;
    protected AbstractStatechart Off_in_Region_in_Open;

    // Business object associated with the SM
    protected BusinessObject bo;

    private void initialize_SM() throws Statechart_exception {
        bo = new BusinessObject();
        Off_in_Region_in_Closed = new Statechart("Off");
        Off_in_Region_in_Closed.inputState();
        Off_in_Region_in_Closed.set_entryAction(bo, "stop", null, AbstractStatechart.Reentrance);
        Baking_in_Region_in_Closed = new Statechart("Baking");
        Baking_in_Region_in_Closed.set_entryAction(bo, "heat", null, AbstractStatechart.Reentrance);
        Paused_in_Region_in_Open = new Statechart("Paused");
        Paused_in_Region_in_Open.set_entryAction(bo, "pause", null, AbstractStatechart.Reentrance);
        Off_in_Region_in_Open = new Statechart("Off");
        Off_in_Region_in_Open.inputState();
        Off_in_Region_in_Open.set_entryAction(bo, "stop", null, AbstractStatechart.Reentrance);
        Open = (Paused_in_Region_in_Open.xor(Off_in_Region_in_Open)).name("Open");
        Closed = (Off_in_Region_in_Closed.xor(Baking_in_Region_in_Closed)).name("Closed");
        Closed.inputState();
        Closed.deep_history();
        Microwave_StateMachine = new Statechart_monitor((Closed.xor(Open)), "Microwave",
                                                   AbstractStatechart_monitor.Don_t_show_on_system_out);
    }

    public void start() throws Statechart_exception {
        Microwave_StateMachine.fires("DoorOpen", Closed, Open, true, bo, "openDoor");
        Microwave_StateMachine.fires("DoorClosed", Open, Closed, true, bo, "closeDoor");
        Microwave_StateMachine.fires("power", Off_in_Region_in_Closed, Baking_in_Region_in_Closed);
        Microwave_StateMachine.fires("power", Baking_in_Region_in_Closed, Off_in_Region_in_Closed);
        Microwave_StateMachine.fires("DoorOpen", Baking_in_Region_in_Closed, Paused_in_Region_in_Open,
                                                                              true, bo, "openDoor");
        Microwave_StateMachine.start();
    }

    public void stop() throws Statechart_exception {
        Microwave_StateMachine.stop();
    }

    // Constructor
    public Microwave_StateMachine() throws Statechart_exception {
        initialize_SM();
    }

    // Events
    public void DoorOpen() throws Statechart_exception {
        Microwave_StateMachine.run_to_completion("DoorOpen");
    }

    public void DoorClosed() throws Statechart_exception {
        Microwave_StateMachine.run_to_completion("DoorClosed");
    }

    public void power() throws Statechart_exception {
        Microwave_StateMachine.run_to_completion("power");
    }
}
```

Fig. 2. PauWare generated code from the microwave UML state machine diagram

is defined from the state "Open" to the history state of "Closed". In PauWare, there is a small difference of semantics with the UML specification concerning the history states: there is no direct transition targeting an history state but if a transition has for target a composite state that contains an history state, then this transition is implicitly targeting this history state. This is why, in line 39, the transition with the "DoorClosed" event has for target the "Closed" state;
- line 44, the state machine is started.

Once started, the state machine can process events. This is done by calling the `run_to_completion` method with an event name as parameter. If transitions associated with this event are starting from the current active states, then the transitions are triggered, the active states are changed and the associated business operations are executed.

In the generated code, there is a method for each event processing. For instance, line 56, the `DoorOpen()` method process the "DoorOpen" event. Such methods seem to be irrelevant as they only call the `run_to_completion` method but in the following of this section, we will see their interest when events are coming with values.

To finish presenting the example, the following code instantiates and starts the state machine and processes the "power" event:

```
Microwave_StateMachine sm;
sm = new Microwave_StateMachine();
sm.start();
sm.power();
```

Concretely, it executes the following steps:

1) The initial state of the state machine is activated: the "Closed" state.
2) The "Closed" state is a composite: its initial state "Off" is activated.
3) The Off state has an entry action: the `stop()` method is called on the `bo` object.
4) The "power" event is processed: starting from the "Off" active state of "Closed", there is a transition associated with this event: the transition is triggered, "Off" is deactivated and "Baking" is activated.
5) As the "Baking" state owns an entry action, it is executed: the `heat()` method on the `bo` object.

The code of the Java business operations is not presented here, they control the running of the microwave: turning on or off the light or the magnetron,...

An optional thing that is possible to do when executing a state machine, is to use the PauWare viewer. It is an experimental tool that draws automatically in a Web browser the state machine under execution. After executing the previous lines of code, it will draw the Figure 3. It is easy to retrieve the state machine of the initial UML diagram. The current active states are written in blue: here, " Baking" and its composite state "Closed".

D. Pauware code generator

The PauWare code generator is a tool that generates the code of a state machine for the PauWare API from a UML state machine model. As an example, the code of Figure 2 has been generated from the UML state machine of Figure 1. This generator has been implemented for being the most easy to use as possible.

Firstly, it requires no installation as it is available as a Web application. One has just to open the Web page of the generator[4], to upload his UML model and then, the Java PauWare code is generated in the page of the Web browser. The code can be copied/pasted in the sources of a Java project opened with any Java IDE.

Secondly, the code generator has been designed for being independent of the UML modelers. For this purpose, it relies on the XMI file format [15]. XMI is a standard of the OMG for storing models and meta-models in a XML file. The code generator loads an XMI file through the implementation of the UML specification in EMF. Then, the code generator generates the PauWare code by navigating in the UML model. However, the saving of a model under the XMI format is not always well respected by the tools. If some tools do not offer this capability, others are interpreting the XMI format at their own way that leads to difficulties to read the contents of the model. The code generator has been able to successfully load and generate valid PauWare code from UML models designed with Modelio[5], Papyrus[6] and StarUML[7]. Some problems have been encountered with models designed with Entreprise Architect[8]: depending on the UML model contents, the code generation works correctly or not.

E. Weaving of business operations

The example of the microwave is weaving basic business operations onto states and transitions in the sense that these operations have no parameters or returned values. To explain how to manage any kind of operations, here is another example: Figure 4 is a state machine representing the behavior of a stack. There are only two states, depending on the fact that the stack is empty or not. For instance, if one pushes a value when the stack is empty, the stack becomes not empty and it is required to push this value onto the stack. Each transition with the event "pop" or "push" is then associated with business operations modifying the stack contents: `pushAction` for pushing a value onto the stack and `popAction` for removing and getting the value on top of the stack.

The listing of Figure 5 shows a part of the code associated with the stack state machine. For instance, the transitions associated with the "pop" event are defined using a guard: there are implemented with the methods `onlyOne()` and `moreThanOne()`. A guard is a Java method returning a boolean. It is associated with a transition with the same

[4]https://pauware.univ-pau.fr/generator/
[5]https://www.modelio.org/
[6]https://www.eclipse.org/papyrus/
[7]http://staruml.io/
[8]http://sparxsystems.com/

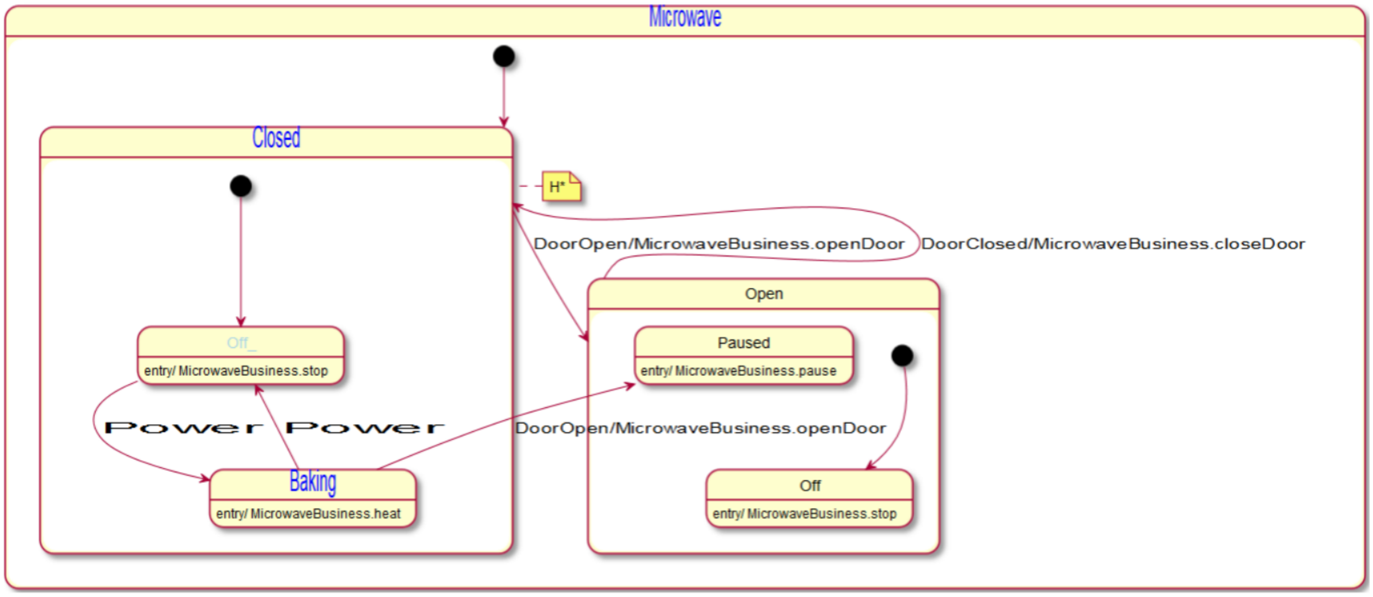
Microwave
Closed
H*
DoorOpen/MicrowaveBusiness.openDoor
DoorClosed/MicrowaveBusiness.closeDoor
Open
Off_
entry/ MicrowaveBusiness.stop
Paused
entry/ MicrowaveBusiness.pause
Power
Power
DoorOpen/MicrowaveBusiness.openDoor
Baking
entry/ MicrowaveBusiness.heat
Off
entry/ MicrowaveBusiness.stop

Fig. 3. The microwave PauWare state machine under execution drawn with the PauWare viewer tool

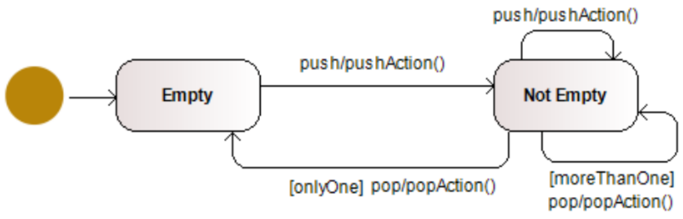

Fig. 4. UML state machine diagram of a microwave oven designed with Modelio

elements as for a business operation: the object on which the guard is called, the name of the guard method and the optional parameters (here there is none). The guards simply checks here the size of the concrete stack object that is an instance of `java.lang.Stack`.

The `pushAction` business operation takes a String as parameter and pushes it onto the stack. This business action is associated with the two transitions for the "push" event as shown on the state machine of Figure 4[9]. In the code, the `pushEvent` method enables to process a "push" event with the call of the `run_to_completion` method on the state machine instance. Before that, surprisingly, it defines the two transitions associated with the "push" event. The code is not presented here, but as for the microwave state machine, these transitions have been already defined when building and starting the state machine. So why here redefining these transitions each time the "push" event is processed? In fact, they are not defined several times. The PauWare engine detects that the same transitions (same source and target states, same event, same operation or guard with same types of parameters

[9]In the Modelio state machine diagram, the parameters and returned values of the methods are not shown but they have been defined as in the Java code.

```
...
private java.util.Stack<String> stack;

public boolean onlyOne() {
    return stack.size() == 1;
}

public boolean moreThanOne() {
    return stack.size() > 1;
}

public void pushAction(String value) {
    stack.push(value);
}

public String popAction() {
    return stack.pop();
}
...
public void pushEvent(String value) {
    sm.fires("push", empty, notEmpty, true,
        bo, "pushAction", new Object[] { value });
    sm.fires("push", notEmpty, notEmpty, true,
        bo, "pushAction", new Object[] { value });
    sm.run_to_completion("push");
}
...
```

Fig. 5. Part of the code associated with the stack state machine

on the same objects) exist already. The goal here is to modify the set of parameters of the `pushAction` business operation: the parameters become an array of Object containing the String "value" parameter of the `pushEvent` method. In this way, we are able to define a transition associated both with an event and business values. These business values can then be passed as parameters of the business operation or the guard[10].

The `popAction` business operation is associated with the two transitions with the "pop" event. Its code is presented in the listing of Figure 5: it simply removes and returns the top of the stack. But which element is getting this returned value? Concretely, there is none. The problem here comes from the inherent nature of executable models as we explained in [10]: they capture the behavior of the system and they control the call of the business operations. Both parts of the application, control and business, are clearly separated. The executable model reifies the control flow but the data flow among business operations is difficult to manage. It is not possible to set that the returned value of `popAction` will be a parameter of another business operation. In [10], we propose a generic solution to manage the data flow when defining an executable DSL. But here, we are using standard UML state machines and our solution requires to modify the meta-model of the DSL. Our solution can anyway be manually implemented in the Java code: the idea is to use a shared object such as a Java Map to store each required data (parameters or returned values) with a name. Concerning our `popAction` operation, the transitions will be associated with another operation, a wrapper, that will call the business operation and put the returned value in the Map. So, this value can be used when another business operation will be called later on during the state machine execution. Here is what it could be:

```java
// the new operation associated with the transitions
public void wrapperPopAction() {
    // call the business operation
    String res = bo.popAction();
    // put its returned value in the shared Map
    mapValues.put("popActionResult", res);
}
```

Finally, concerning the business operations, it is of course also possible to change the parameters of the operations associated with states (our examples do not require to use this feature).

IV. RELATED WORK

Code generation from UML diagrams or state machines expressed in another formalism has been widely studied. Some works are based on formal methods but this prevents from being used by most of the software engineers because of their complexity. The main advantage of our approach does not deal with this code generation but more generally with the ability to build an application based on the weaving of easy-to-define executable models and regular code for the business part of the application. Two main approaches dealing with Java business code can be cited in this domain.

The first one is Yakindu [16]. It is a tool for defining state machines based on the semantics of Harel's statecharts [17] that are the basis of the UML state machines with slight differences. Yakindu offers a graphical and integrated environment for defining state machines, simulating, debugging and testing them. From the point of view of these verification capabilities, Yakindu goes further than our PauWare tools. However, the state machine must be defined with their tool whereas we are the most independent as possible of UML modelers. Yakindu offers also code generation features for several languages, including Java. Business operations and variables can be defined in the statechart model and the Java code will integrate the signatures of the business operations and even the affectation and the update of the business variables. It is possible as for PauWare to weave business code developed apart into the state machine. However, the code generation is made at a lower level than for PauWare. Yakindu does not offer an extended API for programming state machine structures as PauWare. As a consequence, the PauWare code generator produces a code that is more easily readable and maintainable manually by a developer concerning the definition of the state machine. Moreover, the generated code of Yakindu is mixing the business operations with a part of the execution engine. In PauWare, the execution engine code is fully separated from the state machine code.

The second approach is jBPM for Java Business Process Modeling [18]. Business processes are kind of activity diagrams that define the orchestration of business operations. In the OMG's world, such models are defined in BPMN (Business Process Model and Notation) [11] which is one of the specifications implemented by the jBPM toolkit. jBPM is similar to PauWare on several points: the executable models can be integrated and woven with business operations within any kind of Java application. jBPM can be used as PauWare for programming an executable model but an Eclipse-based and a Web-based graphical editors are also available. A Web-based editor offers the advantage of not requiring any software installation where in PauWare we are independent of the UML modelers. jBPM integrates a lot of technical features (persistence, links to existing Java frameworks...) but the main difference with PauWare deals with the kind of executable models that can be used. jBPM is based on process models for defining the orchestration of business operations whereas PauWare is based on state machines and is then more suitable for event-based and reactive applications.

State Chart XML (SC-XML) is a W3C standard [19] for defining state machines under a XML format. If a partial code generator from SC-XML to PauWare has only been implemented, PauWare however integrates the semantics of SC-XML and can be used to execute equivalent models [14], [20]. For Java development, PauWare offers the same functionality as the Javascript SCION engine [21] able to execute SC-XML state machines.

Finally, readers interested in how weaving business operations onto executable DSL can read [10] and its related work section to compare with existing approaches.

V. CONCLUSION

In this paper, we propose a model-based software development process based on UML state machines. State machines are executable models and such models offer the advantage to

[10]If this way of passing the parameters of business operations is at a first sight a bit complex, remember that the code of the `pushEvent` method is automatically generated from the UML model.

capture the behavior of a system at a high-level of abstraction. Our PauWare API and engine enable to program in plain Java a UML state machine, to weave business operations onto its states and transitions and to execute the resulting application. To simplify the implementation of such state machines, we have implemented a code generator that takes as input a UML model defining a state machine and produces the equivalent code for the PauWare API. Then, it is easy to weave existing Java business operations implemented apart onto this generated code. This development process is an intermediate and pragmatic approach between modeling and coding: the executable state machine is defined as a standard UML model because it is the most suitable way to define it but for the business part, integrating the complete business operations contents in the model with fUML for instance will be too complex and restrictive. For this reason, the business part relies on a plain Java implementation to maximize the flexibility and efficiency in the development of the business operations.

As a perspective, the main point is to develop the verification features of the PauWare tools. As shortly described in section III-A, once the code generated from an UML state machine, you can simulate the state machine without the business operations. Once you have weaved the concrete business operations with the state machine code, you get the complete application and you can execute it. As both for simulation and execution stages, the execution engine is the same (it is the PauWare engine). That is important to notice as it implies that the simulated model and the executed model are processed with the same execution semantics. There is no gap between the specification and the code: the same model (under the form of PauWare Java code) is executed in the same manner. Consequently, if your state machine model is valid at design, it will also be valid at runtime. To fulfill the verification capabilities at simulation or running stages, PauWare has been extended to attach a monitor to the running state machine. Concretely, a monitor implements a set of methods that will be called when actions are processed on the state machine: a new state is activated, a business operation is executed, ... This kind of monitor can be used for generating an execution trace, for ensuring that the state machine is running correctly or to plug a model checker to verify the state machine behavior. The main perspective around PauWare is to develop case studies for these verification capabilities.

Another perspective is to be able to make retro-engineering. In the paper, we have presented a process from top to bottom where PauWare code is generated from a UML model. It could also be possible to retrieve the UML state machine from a PauWare program. This is what the PauWare viewer is doing as presented in the paper but an interesting feature will be to generate a UML state machine model readable with a UML modeler. This will also enable making co-evolution between the model and the code.

VI. Acknowledgment

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) [22] 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 Unions Horizon 2020 research and innovation program and from Sweden, Spain, Italy, Finland & Czech Republic.

References

[1] OMG, "Unified Modeling Language (UML) Specification, version 2.5.1," 2017, http://www.omg.org/spec/UML/2.5.1/.
[2] ——, "Object Constraint Language (OCL) Specification, version 2.4," 2014, http://www.omg.org/spec/OCL/2.4/.
[3] ——, "Meta Object Facility (MOF) Specification, version 2.5.1," 2016, https://www.omg.org/spec/MOF/2.5.1.
[4] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, 2008.
[5] OMG, "Semantics of a Foundational Subset for Executable UML Models (fUML) Specification, version 1.4," 2018, http://www.omg.org/spec/FUML/1.4/.
[6] ——, "Action Language for Foundational UML (ALF) Specification, version 1.1," 2017, http://www.omg.org/spec/ALF/1.1/.
[7] ——, "Precise Semantics of UML Composite Structures (PSCS) Specification, version 1.2," 2019, https://www.omg.org/spec/PSCS/1.2/.
[8] ——, "Precise Semantics of UML State Machines (PSSM) Specification, version 1.0," 2019, https://www.omg.org/spec/PSSM/1.0/.
[9] F. Barbier and E. Cariou, "Executable Modeling for Reactive Programming," in *Model-Driven Engineering and Software Development (MODELSWARD 2018)*, ser. CCIS, vol. 991. Springer, 2019.
[10] E. Cariou, O. Le Goaer, L. Brunschwig, and F. Barbier, "A generic solution for weaving business code into executable models," in *The 4th International Workshop on Executable Modeling at MODELS (EXE 2018)*, vol. 2245. CEUR Workshop Proceedings, 2018.
[11] OMG, "Business Process Model and Notation (BPMN) Specification, version 2.0," 2015, https://www.omg.org/spec/BPMN/2.0/.
[12] E. Cariou, O. Le Goaer, and F. Barbier, "On the Executable Nature of Models," in *The 2nd International Workshop on Executable Modeling at MODELS (EXE 2016)*, vol. 1760. CEUR Workshop Proceedings, 2016.
[13] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution Framework of the GEMOC Studio (Tool Demo)," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2016, ACM, Ed., 2016.
[14] F. Barbier, *Reactive Internet Programming – State Chart XML in Action*. the Association for Computing Machinery and Morgan & Claypool, 2016.
[15] OMG, "XML Metadata Interchange (XMI) Specification, version 2.5.1," 2015, https://www.omg.org/spec/XMI/2.5.1/.
[16] Itemis, "YAKINDU Statechart Tools Web site," visited July 2020, https://www.itemis.com/en/yakindu/state-machine/.
[17] D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *Computer*, vol. 30, no. 7, pp. 31–42, 1997.
[18] M. N. De Maio, M. Salatino, and E. Aliverti, *jBPM6 Developer Guide*. Packt Publishing Ltd, 2014, https://www.jbpm.org/.
[19] W3C, "State Chart XML (SCXML): State Machine Notation for Control Abstraction, recommendation 1," 2015, https://www.w3.org/TR/2015/REC-scxml-20150901/.
[20] F. Barbier, O. Le Goaer, and E. Cariou, "Energized State Charts with PauWare," in *2nd Workshop on Engineering Interactive Systems with SCXML at EICS 2015*.
[21] Jacobean Research and Development LLC., "SCION: a suite of software for standard state machine support," visited July 2020, https://scion.scxml.io/.
[22] W. Afzal, H. Bruneliere, D. D. Ruscio, A. Sadovykh, S. Mazzini, E. Cariou, D. Truscan, J. Cabot, A. Gómez, Y. Gorronogoitia, L. Pomante, and P. Smrz, "The MegaMRt2 ECSEL project: MegaModelling at Runtime Scalable model-based framework for continuous development and runtime validation of complex systems," *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)*, no. 61, Elsevier, pp. 86–95, 2018.