Code Generation from UML Models with Semantic Variation Points*

Franck Chauvel[1] and Jean-Marc Jézéquel[2]

[1] VALORIA
[2] INRIA & Université de Rennes 1

Abstract. UML semantic variation points provide intentional degrees of freedom for the interpretation of the metamodel semantics. The interest of semantic variation points is that UML now becomes a family of languages sharing lot of commonalities and some variabilities that one can customize for a given application domain. In this paper, we propose to reify the various semantic variation points of UML 2.0 statecharts into models of their own to avoid hardcoding the semantic choices in the tools. We do the same for various implementation choices. Then, along the line of the OMG's Model Driven Architecture, these semantic and implementation models are processed along with a source UML model (that can be seen as a PIM) to provide a target UML model (a PSM) where all semantic and implementation choice are made explicit. This target model can in turn serve as a basis for a consistent use of code generation, simulation, model-checking or test generation tools.

1 Introduction

UML (Unified Modeling Language) has been widely critized in the past for its fuziness, making it difficult to build code generators, simulation, model-checking or test generation tools working in a consistent manner. Many tool vendors are nevertheless producing useful tools, some of them even have reach a certain level of industrial acceptance. The interest of having a *unified* modeling language is however questionable if the meaning of a UML model depends on which tool is used for any given purpose. With the advent of UML 2.0 [14] though, many of previous version UML fuziness issues have been solved, and some of the rest have been encapsulated into the notion of *semantic variation points*.

A semantic variation point is a point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics. For instance, we find on page 40 of [14] *The precise lifecycle semantics of aggregation is a semantic variation point.* The interest of semantic variation points is that UML now becomes a family of languages sharing lot of commonalities and some variabilities that one can customize for a

* This work has been partially supported by the Amadeus project of Région Bretagne and by the Artist2 Network of Excellence on Embedded Systems Design (IST-004527).

L. Briand and C. Williams (Eds.): MoDELS 2005, LNCS 3713, pp. 54–68, 2005.

given application domain. This makes a lot of sense, because for instance the type of behavior one would expect from the statecharts of books in a library business application has some differences with the statecharts of a CD player in a real-time system. Furthermore the code one wants to see generated definitively does not look the same.

Similarly to working with product lines [18], the challenge of the tool builders is then obviously to capitalize on commonalities while making it possible to customize their tools with respect to the choosen variants. We propose to reify these semantic variation points as well as possible implementation choices into models of their own. Then along the line of the OMG's Model Driven Architecture [15], these models are processed along with a source UML model (that can be seen as a PIM) to provide a target UML model (a PSM) where all semantic and implementation choice are made explicit. This target model can in turn serve as a basis for a consistent use of code generation, simulation, model-checking or test generation tools. In this paper we concentrate on behavioral aspects described through UML 2.0 statecharts. Section 2 introduces the running example of a CD player modeled as a statechart at a PIM level. It then discusses semantic variation points for UML statecharts and proposes a model M_s for them. Section 3 discusses several implementation techniques for UML statecharts and also proposes a model M_i for them. Section 4 describes how a PSM can be automatically obtained from these three models, (the PIM, M_s and M_i) through model transformations. Section 5 discusses related works, and Section 6 concludes and present some perspectives to this work.

2 Semantic Variation Points for UML 2.0 Statecharts

Let's consider a simple CD player supporting three main functionalities: one can open the player and play a CD, as well as suspend and resume the playing. Furthermore, if the playing is suspended (in pause) for more than 10 minutes, the player automatically stops. This is modeled with a simple statechart [6,8] as illustrated in Figure 1.

Before dealing with semantic variation points let's have a look at the UML Statecharts meta-model which is shown on figure 2. UML 2.0 Statecharts define a set of concepts that can be used to define finite state-transitions systems.

Numerous semantics have already been developed to precisely define the meaning of statecharts notations (see Von der Beek's impressive catalog [3]). Along this line, UML 2.0 defines yet another semantics for statecharts, or more precisely a family of semantics since it lets a number of issues open. These semantic variation points mainly concern 3 aspects: time management (synchronous vs. asynchronous), the event selection policy, and the transition selection policy.

2.1 Time Management

With respect to the statechart progression, time can be either synchronous or asynchronous.

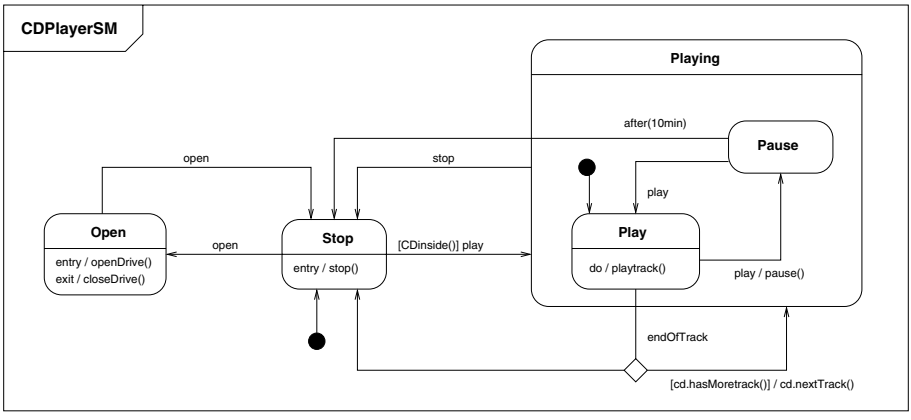

Fig. 1. Behavior of a CD player

Under the asynchronous hypothesis, time is discrete. On each step, the statechart processes events that have occurred between the current and the previous step. The statechart thus needs to store incoming events into some sort of collection. Depending on the policy choosen for event processsing (see below) this collection might be a queue, a bag or a stack or even something more exotic.

Under the synchronous hypothesis, time is continuous. As soon as an event occurs, it is processed in zero time. So, there is no more need of any data structure to store events.

2.2 Event Management

Different kinds of events can be considered. Events can be internal/external or discrete/continuous.

Events are called external if they are produced by an object different from the target object. Let's consider two objects O_1 and O_2 where O_2 reacts when event e_1 occurs. In the context of O_2, e_1 is an external event because it was produced by object O_1. If it were produced by O_2 itself, it would have been viewed as an internal event.

Events are either discrete if they trigger only one transition during their life cycle, or continuous otherwise. In UML 2.0, we also have deferred events: a state may specify a set of event types that may be deferred in that state. If an event occurs in a state where it cannot trigger any transitions, then it should not be discarded if its type matches one of the types in the deferred event set of that state. Instead, it should remain in the event pool while another non-deferred event is dispatched instead.

Using deferred events can lead to conflict: for instance when a substate defers an event while the composite state consumes it, or vice versa. In case of a composite orthogonal state, substates of orthogonal regions may also introduce

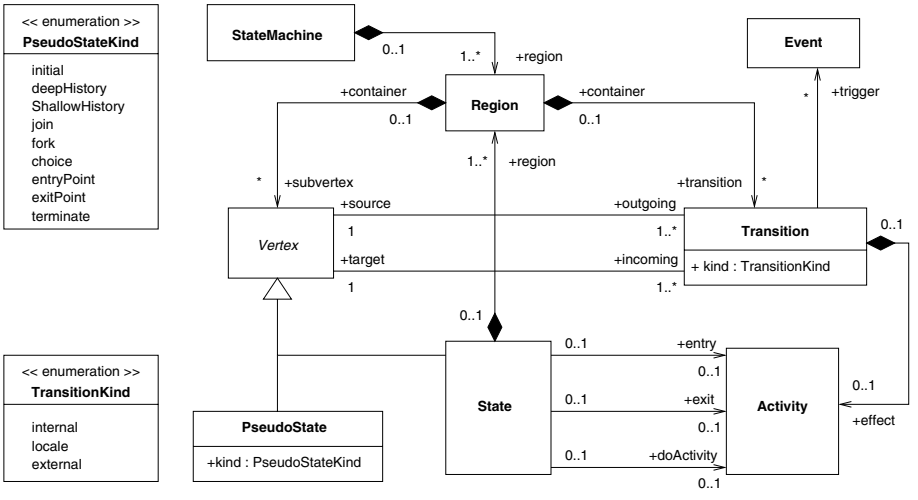

Fig. 2. Excerpt of the UML 2.0 Statecharts Meta-Model

deferral conflicts. To solve this kind of conflict, UML 2.0 consider that nested states override composite state and a consumer state overrides a deferring state when conflict appears between two orthogonal regions.

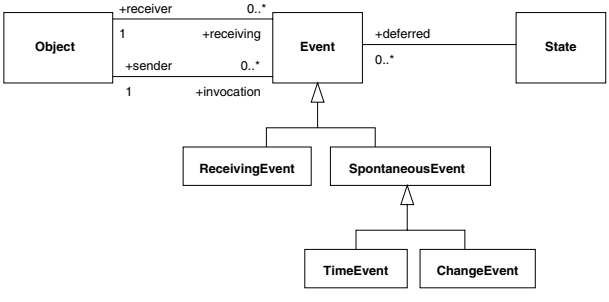

Fig. 3. Events in UML 2.0

One other variation point in UML is the way to select an event in the event pool. It is explicitly listed as a semantic variation point in the UML 2.0. In fact, there are many ways to do this. The structure can be a queue and so events are selected by incoming order. It can also be a stack if the most recent event is selected. We can also use any priority systems or a mail box system to define more powerful selection policies.

2.3 Transition Management

Using a composite state (such as the "playing" state in our CD player) can lead to conflicts among transitions. To solve this, the UML defines a transition priority system based on source states, with transitions originating from deeper states having higher priority. For example, if s_2 is a substate of s_1 then transitions originating from s_2 have higher priority than transitions originating from s_1.

This kind of priority system does not solve every conflicts. For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then only one transition should be fired. Only one transition can be fired simultaneously except for concurrent state's regions. So we need a way to decide between two conflicting transitions. There are two main ways to solve this kind of conflicts: we can either always choose the same arbitrary transition or use a randomized choice (for fairness purposes for example).

2.4 Modeling Statechart Semantic Variation Points

To explicitly express these semantic variation points, we need a model describing the various event selection policies and transition policies. This model can be seen as a reification of the part of the semantics which is subject to variability, with the variability itself modeled using standard OO features such as inheritance and delegation (see Figure 6).

Harel [8] describes the operational semantics of statecharts based on the description of a *run-to-completion* step as it shown in figure 4.

The way this procedure is called depends on whether the time model is synchronous or asynchronous. Under the asynchronous hypothesis, time is discrete and so the *step* procedure must be triggered by a third party mechanism like a clock for example. Under the synchronous hypothesis, time is continuous and this procedure must be encapsulated into an infinite loop to process events as soon as they occur.

```
procedure step()
begin
   eventSet := eventPool.select();
   anEvent := eventSet.choice();
   transitionSet := getFirableTransition(anEvent).select();
   aTransition := transitionSet.choice();
   aTransition.fire();
end.
```

Fig. 4. The run-to-completion procedure

With respect to the semantic variation points described above, this *run-to-completion* procedure looks like a GoF's *Template Method* [4], that is the skeleton

of an algorithm in an operation, deferring some steps to subclasses. The steps we want to be able to redefine here are the following:

1. We apply some priority scheme in order to determine which event we want to process (cf. operation "*eventPool.select()*" on figure 4).
2. Since this priority scheme might return more than one event (events of the same priority), we then need to choose the one we actually process (cf. operation "*eventPool.choice()*").
3. With this event, we now can select the set of firable transitions (cf. operation "*getFirableTransition(anEvent)*").
4. On this transition set, we apply some other priority scheme to first resolve simple cases of non-determinism (cf. operation "*transitionSet.select()*"), and then if this is not enough to get only one transition, we need to decide between selected transitions.
5. finally, fire the transition.

All the semantic variation points are then encapsulated in the operations *select* and *choice* called on event sets and transition sets. So to model statecharts semantics we need to add some behavior behind these operations. Quite straightforwardly, we can use the *Strategy* pattern [4] twice to define both an event management policy and a transition management policy. Each one is described with both a selection policy and a conflict resolution policy (See figure 6).

Event and transition management can be explicitly described with an action language such as the Action Semantics. In figure 5, we use the Kermeta Language [12] to describe the semantic of our "select" operation. Kermeta is an object-oriented meta-language and so is well suited to define semantics into meta-models. So, to define a new event selection policy for example, one just needs to extends the "event selection Policy" and to redefine the *select()* operation (See figure 6). For instance, we can define a new event selection policy where TimeOut events have an higher priority than other events as described in the example below.

```
class MyEventSelection inherits EventSelectionPolicy
{
  method select() : OrderedSet<Event> is
  do
    result := eventPool.select{e | TimeEvent.isInstance(e)}.first()
  end
}
```

Fig. 5. A event selection policy defined with *KerMeta*

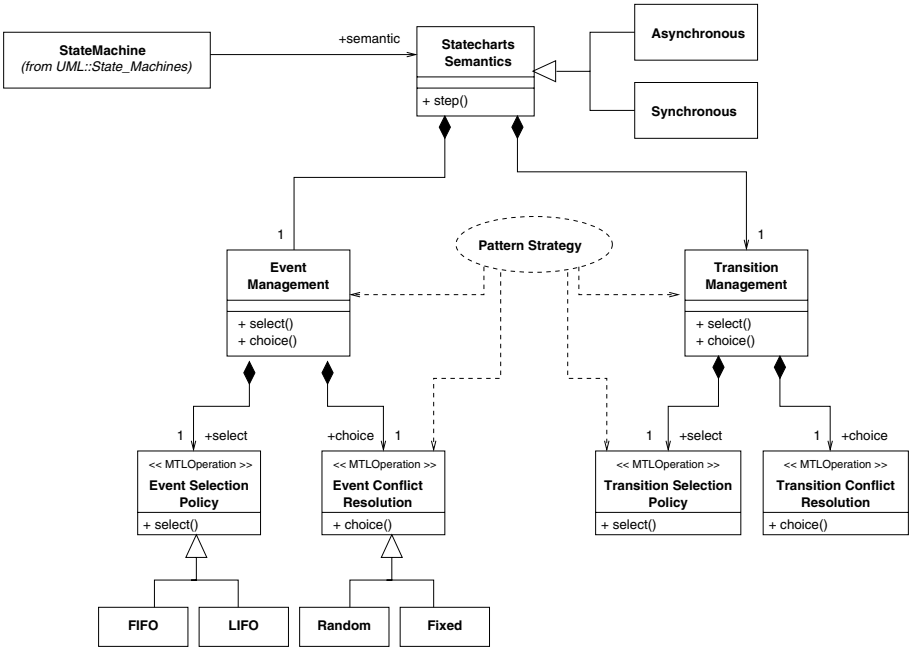

Fig. 6. Model of the UML Statecharts Semantic Variation Points

3 Implementing UML Statecharts

Even if we would have settled on a single possible semantics for statecharts, there can still be many ways to implement them. There are indeed many trade-off to make to handle non-functional issues such as execution time, memory footprint, flexibility, maintenability, possibility of dynamic upgrade and so on. For example, if we need a compact and efficient implementation, we might want to use enumerated values representing states and events. If we rather want a more flexible solution, we might prefer to resort to the State Pattern and/or the Command Pattern [4]. In this section, we propose to model these implementation choices in the same spirit as for the modeling of semantic variation points.

3.1 Enumeration Vs. Reification

For each of the statechart notions, such as states, events, or transitions, we typically face the choice of either hard code it (into static tables or switch blocs) for maximum efficiency, or reify it for maximum flexibility (using the State Pattern, the Command Pattern, and reifying transitions).

The easiest way to manage states is to represent them with an enumeration. In our example, this type would be "open, stop, playing Play, playing Pause". Note that this solution is not however very well suited for hierarchical statecharts, because it requires to first flatten the state hierarchy. Another solution

is to reify the possible states into a specific class hierarchy through the application of the state pattern [4]. See the right side of Figure 7 for an illustration of the application of the state pattern to our CD player statechart. This solution would even allow us to dynamically add new states, which could be very useful to modify a system behavior without stopping it.

Using an enumeration to manage events requires to put the statechart progression mechanism into a specific method called for instance "*processEvent(e : event)*". Its role is to select the right transition using two "switch" statements. Alternatively we can reify events using the Command Pattern [4]. Then, the progression mechanism is distributed into event classes through object-oriented method dispatch. If the states have not been reified, we still need to select the right state with a "switch" statement.

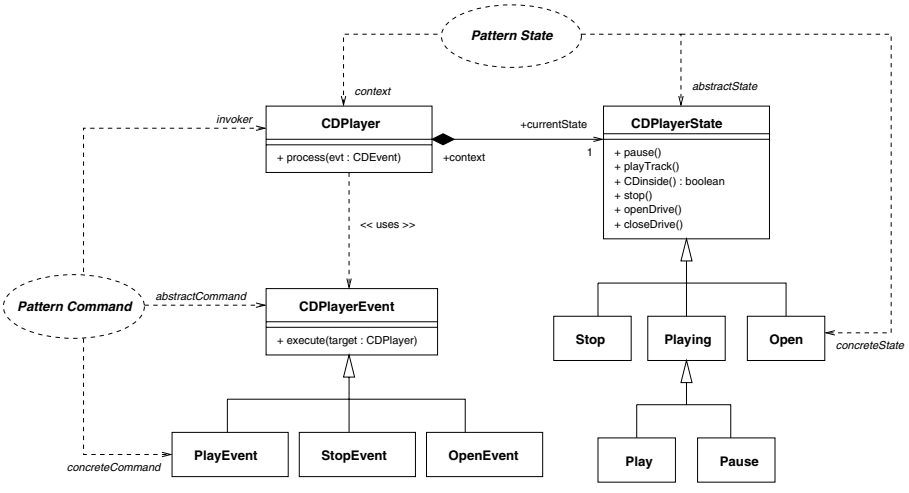

Fig. 7. Implementing CD player using states and events reification

It is also possible to reify both states and events as illustrated in Figure 7. Then, the progression mechanism is distributed into the event classes and the state classes. In fact, here, we use a double dispatch to select the right behavior according to the event and the current state.

We might go as far as also reifying the transition concept. There are some patterns indeed reifying most of the statechart notions (like Tomura's statecharts pattern [16]), including guard condition, actions, etc.

3.2 Statechart Progession

Beyond states, events and transitions, we also have to care for variations about the implementation of the statechart "engine", i.e. the method that makes it progress by selecting which transition must be triggered according to events

and to the current state. The basic choice here is whether the engine is shared accross multiple statecharts or whether it is statechart specific. In the former case, a single statechart can be considered as a passive reactive object with the event dispatching being performed from outside. In the later case we can resort to the Active Object pattern and encapsulate all the internal mechanisms behind a proxy object, as illustrated in Figure8.

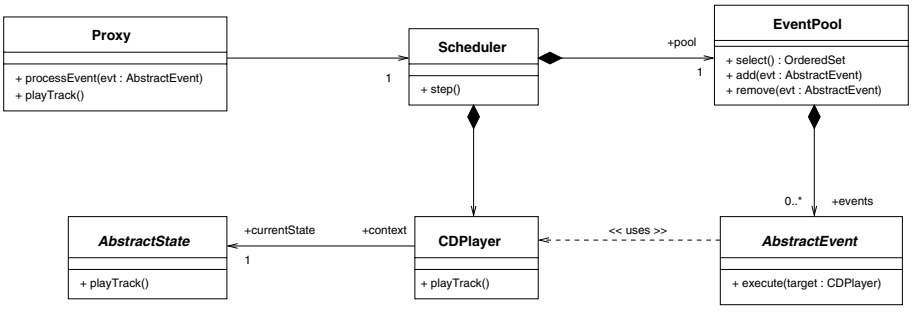

Fig. 8. Active-Object Pattern applied on CD Player

Applying this design patterns allow us to reify the statecharts progression mechanism by defining a scheduler object manipulating object events for example. Note that for this particular implementation choice, it becomes particularly straightforward to attach the semantic variations we described in section 2: the event selection policy would just have to be inserted as the body of the operation select of the EventPool class.

3.3 Modeling the Implementation Choices

In previous sections, we have defined various semantics and implementation choices but we still need to link these choices to our initial statemachine. To help doing that, the UML provides an extension mechanism called a "profile". A profile can be seen as a lightweight extension to the meta-model which adds extra-information to meta-classes. To do this, a profile can contains "stereotype" and "tagged values". Stereotypes are a way to represent boolean information like "is an interface" whereas "tagged values" can be parameterizad by values.

The profile we provide is a way to specify required choices on a statemachine. An example is given in the figure 9. Here, the CDPlayer statemachine would use a FIFO policy for handling events for example.

eventSelection this tagged value allows the specification of the required event selection policy. It correspond to the *select()* operation in the procedure *step()* in figure 4. The value is a string which identifies the event selection policy in the semantic model.

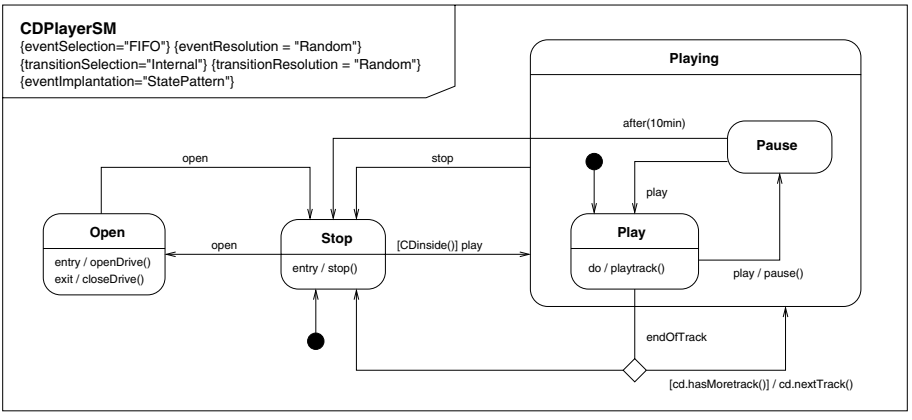

Fig. 9. The CDPlayerSM with some stereotypes specifying semantic

eventResolution this tagged value is used to specify a way to resolve conflict between events. It corresponds to the *choice()* operation in the procedure *step()* in Figure 4. The value is a string which identifies the concrete conflictResolution policy in the semantic model.

transitionSelection this tagged value is used to specify the required transition selection. It corresponds to the *choice()* operation in the procedure *step()* in Figure 4. The value is a string which identifies the concrete transition selction policy in the semantic model.

transitionResolution this tagged value is used to specify a way to resolve conflict between firable transitions. It corresponds to the *choice()* operation in the procedure *step()* in Figure 4. The value is a string which identifies the concrete conflictResolution policy in the semantic model.

eventImplantation this tagged value represent the technic used to specify implantation of event. This is an enumerate value, which can be "enumerate" or "reify" to used a command pattern.

stateImplantation this tagged value represent the technic used to specify implantation of state. This is an enumerate value, which can be "enumerate" or "reify" to used a state pattern.

4 Processing Semantics and Implementation Variants Through Model Transformations

We can now combine the description of a statecharts, its semantics choices and implementation choices in a consistent manner for various software engineering activities such as automatic code generation, simulation, model-checking and test generation.

4.1 Implementing Code Generation as a Model Transformation

For that we need a model transformation language and engine able to process these 3 models as input, and produce either an implementation model or a validation model. In the following, we describe how we used MTL [13], an imperative object oriented language based on KerMeta for that purpose. Our model transformation can be divided in three main steps (See figure 10).

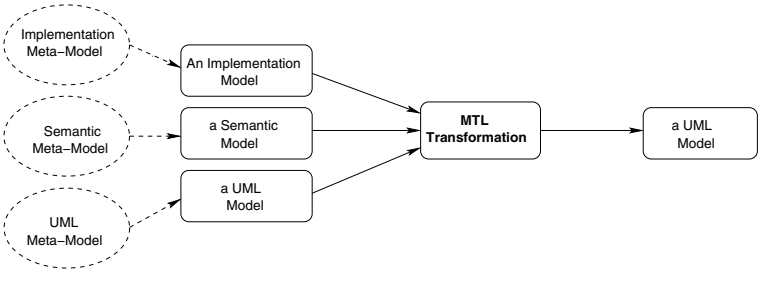

Fig. 10. Processing Statecharts, Semantics and Implementation models to provide a PSM model

Firstly, we search input models for semantic and implementation choices. An abstract factory is then used to dynamically select and configure the needed transformations, most of which are actually quite simple pattern applications [5].

In a second step, we apply the selected patterns. Defining a general way to apply design patterns is a non-trivial issue, but we face here only a subset of this problem: we just need to apply (or not apply) 3 patterns in a specific order, which slightly reduce the combinatory aspect of the problem. So we start with either a direct implementation or with the active-object pattern to define a common structure to implement statecharts mechanisms. Then, depending on the previous choice, we can apply (or not apply) the state and command patterns in an orthogonal way. (See figure 11 for the result of the application of the 3 patterns in a row).

Anyway, we have obtained a detailed model where the statecharts progression mechanism has been fully reified. So, we can easily attach the statecharts semantics by filling the corresponding methods. For example, the semantics specified by the user for the event selection policy would go into the *select* method of the class *Pool* (See figure 11). To do that, we used the MTL language at the meta-model level to describe semantic choices in the input semantic model. As for KerMeta on which it is based, MTL can also be used at the model-level as a kind of simple action language for UML, making it easy to translate the description of the semantics into its implementation.

Finally, our model is refined to the point where each statecharts concept has been mapped onto structural OO notions like classes or operations. An example of behavior of this output model is presented on the figure 12. It can

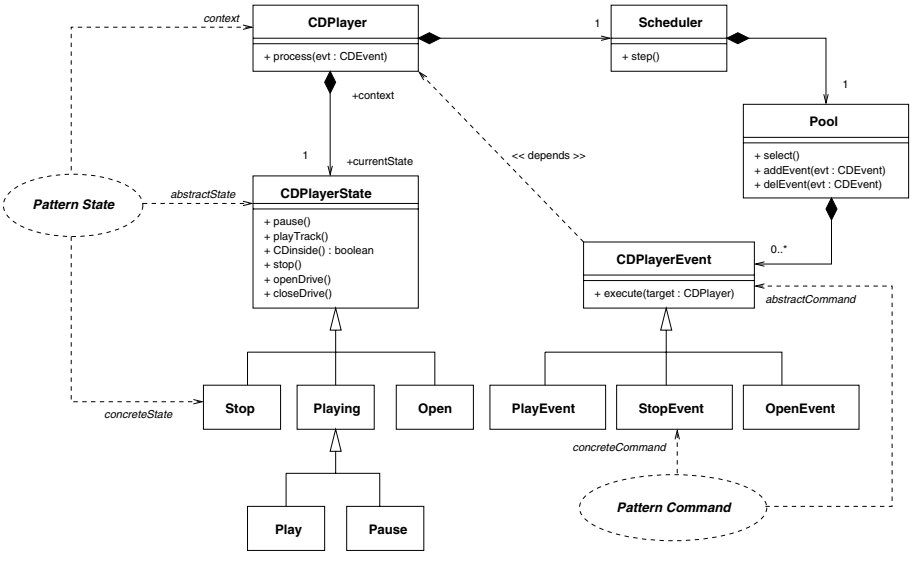

Fig. 11. Output model of the transformation

then be directly translated into executable code using any off-the-self UML code generator, including the MTL one. Indeed, using MTL to describe operation bodies allows us to re-use the MTL Java code generator to generate for free an executable Java code corresponding to our input model.

4.2 Handling UML Variability into UMLAUT NG

UMLAUT NG [9] is an object-oriented framework dedicated to model transformations in a MOF based context. It provides both a library of model transformations specific to UML models (e.g.; UML2RDBMS which translate a UML model to a relational model) as well as composition operators. UMLAUT NG was designed as an open tool working with several flavors of XMI, in order to be easily connected to various CASE tools.

UMLAUT NG also provides a way to connect MDD to formal technics initially developed for SDL, Lotos and others. UMLAUT NG supports model transformations for transforming UML models into labelled transition systems (LTS), to be used with the CADP tool box which provides tools for model checking, simulation, test synthesis and vizualization of the state spaces.

The integration into UMLAUT NG of our approach at reifying statecharts semantic variation points makes it possible to uncouple all of these tools from a specific choice of the statechart semantics. It can be seen as an easy way to specialize a complex tool chain towards a specific domain (e.g.; small embedded devices) where a particular interpretation of the statechart semantics is preferred.

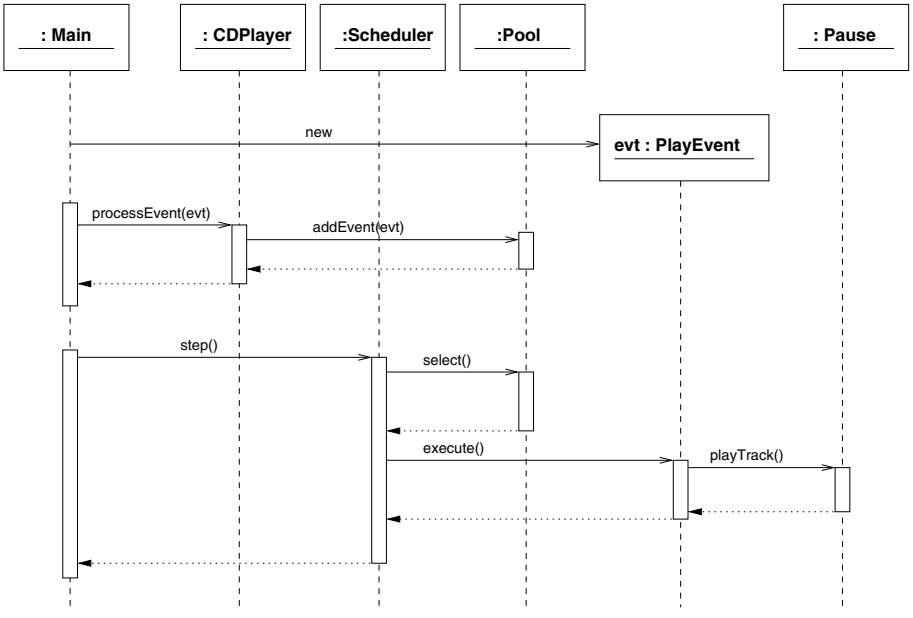

Fig. 12. Behavior of the output model

5 Related Works

The work of [3] has been one of the starting point of our work. Indeed, many papers try to define a formal semantics for Statecharts and especially for UML-Statecharts. Among this works, M.Von der Beeck [17] proposes a structured operational semantics for UML Statecharts. Borger provide another semantics based on abstract StatesMachines [1]. All these works contribute to give a formal ground to UML at the price of choosing a particular semantics, which might be adequate for a particular application domain, but not that much for others, which is the basic reason why the UML provides semantic variation points. In this paper we specifically address this semantic variation point issue.

Building on formal semantics, many tools are able to simulate UML models and specially statecharts [7]. Another example is *iUML* of Kennedy Carter [2] which includes a modeler and a simulator based on the ASL language. Most of these tools are based on more or less formalized semantics and do not take into account semantic variation points.

Another way to execute UML models is to generate executable code directly from models. In an MDA perspective, many models transformations are required to get code from high level models. This process starts at the highest level with the platform independent model (PIM) and continues until a Plateform Specific Model (PSM) is generated. We believe that actual code generation should be used only when a model is low level enough to be directly translated to C++ or Java. However most of dedicated code generation tools provide code generation

directly from e.g. statecharts. For instance Tanaka [11,10] proposes an UML to Java code generation from statecharts diagrams (based on the state pattern with events reified as method calls), or Rhapsody, a UML Case tool, proposes a code generator where events and states are selected using a switch statement (which is a relevant choice for its commercial target which is the real-time domain). These code generators tends to hard code semantic and implementation choices, making them difficult or even irrelevant to use outside of their sometimes very narrow domain. On this aspect, the main contribution of our work consists in providing a way to uncouple semantics issues and implementation choices from code generation, and let them open-ended. Users can always add a particular semantics and choose a particular implementation technics by extending the existing framework.

6 Conclusion and Future Works

The interest of semantic variation points in UML is that it now becomes a family of languages sharing lot of commonalities and some variabilities that one can customize for a given application domain. In this paper, we have proposed to reify the various semantic variation points of UML 2.0 statecharts into models of their own to avoid hardcoding the semantic choices in the tools. We did the same for various implementation choices. Through model transformations, these semantic and implementation models are then processed along with a source UML model to provide a target UML model where all semantic and implementation choice are made explicit. We have shown how this target model can in turn serve as a basis for a consistent use of code generation, simulation, model-checking or test generation tools.

This process has been implemented within our UMLAUT framework for model transformations, along with others tools such as statecharts generation from sequences diagrams or sequences diagram generation from textual requirements. Even if a a complete chain of model transformations from requirements to executable code is not a realistic approach, UMLAUT aims at providing building blocks that can be customized for a specific model driven design and validation process.

In the future, we plan to use the same approach to reify other semantic variation points in the UML2.0 metamodel. In the UML2.0 component model for example, we can find some open issues like the semantics of method dispatch, interfaces conformity or the support of QoS attributes. It could be interesting to describe these semantic choices as we did for statecharts and to merge them with a component model to get a PSM model.

References

1. Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. On formalizing UML state machines using ASM. *Information & Software Technology*, 46(5):287–292, 2004.

2. Kennedy Carter. iUMLite tool suite and ASL language. from Kennedy Carter's website (`http://www.kc.com`).
3. Michael Von der Beeck. A comparison of statecharts variants. In L. De Roever and J. Vytopil, editors, *In Formal technics in Real-Time and Fault-tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, New-York, 1994. Springer Verlag.
4. Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
5. Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In *Proceedings of UML 2000*, volume 1939 of *LNCS*, pages 482–496. Springer Verlag, 2000.
6. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
7. David Harel and Eran Gery. Executable object modeling with statecharts. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 246–257. IEEE Computer Society, 1996.
8. Harel, David and Naamad, Amnon. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, october 1996.
9. Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *Proc. Automated Software Engineering, ASE'99, Florida*, October 1999.
10. Jauhar, Ali and Tanaka, Jiro. Implementation of the Dynamic Behavior of Object Oriented System. In *Third World Conference on Integrated Design and Process Technology (IDPT'98)*, volume 4, Berlin, Germany, July 1998.
11. Jauhar, Ali and Tanaka, Jiro. Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams. *Journal of Computer Science & Information Management (JCSIM)*, 2(1):24–36, 2001.
12. Franck Fleurey Pierre-Alain Muller and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of UML MoDELs 2005, Jamaica*, LNCS. Springer Verlag, 2005. to be published.
13. Damien Pollet, Didier Vojtisek, and Jean-Marc Jézéquel. OCL as a core UML transformation language. WITUML 2002 Position paper, Malaga, Spain, jun 2002. http://ctp.di.fct.unl.pt/ ja/wituml02.htm.
14. UML Revision Task Force RTF. UML draft version 2.0 specification, April 2003.
15. Soley, Richard and OMG Staff Group. Model Driven Architecture. White papers, Object Management Group, Novembre 2000.
16. Toyoaki Tomura and Satoshi Kanai. Developing simulation models of open distributed control system by using object-oriented structural and behavioral patterns. In *ISORC*, pages 428–437, 2001.
17. Michael von der Beeck. A structured operational semantics for UML-statecharts. *Software and System Modeling*, 1(2):130–141, 2002.
18. Twefik Ziadi. *Manipulation de lignes de produits en UML*. PhD thesis, Universit de Rennes 1, 2004.