Event Processing in Code Generation and Execution Framework of UML State Machines

Anna Derezińska, Romuald Pilitowski

Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19
00-665 Warsaw, Poland
A.Derezinska @ii.pw.edu.pl

Abstract. While generating code not only from UML class diagrams but also from state machines, we have to cope with inconsistencies and semantic variation points of the UML specification. The Framework for eXecutable UML (FXU) transforms UML models into programming code and supports execution of the resulting application according to the behavioral model. The FXU is the first framework supporting all elements of UML 2.x state machines in code generation and execution for C# code. In this paper we focus on the event processing and determination of transition priorities resolved within the framework. Two selected problems of not-distinct event priorities occurring in orthogonal states together with proposed solutions are also presented.

Keywords: UML code generation, state machines, statecharts, inconsistencies and semantic variation points

1 Introduction

Automatic transformation and execution of UML models can be very helpful in the rapid development of robust applications. Execution of UML models can be achieved twofold. Firstly, some behavioral models can be directly simulated by a CASE tool. Secondly, a model may be translated into a code in a programming language. This code can be further compiled and executed. In this case, semantic of the state machine behavior is either fully implemented by the generated code [1] or hidden in a library or virtual machine [2,3,4]. The Framework for eXecutable UML (in short FXU) represents this second approach i.e., model transformation and building an application with support of a run-time library [5,6]. Before the compilation the code can also be modified by a programmer to precise elements which are hard to express in the UML.

A part of a model that is executed covers often UML classes and their state machines. As opposed to most of the tools of the similar functionality, the FXU takes into account the entire range of UML state machine features. It was the first tool supporting code generation from complete UML 2.0 state machines to C# code.

Creating the framework, it was indispensable to answer some questions that are left unspecified in the UML-OMG documents [7]. We had to decide among semantic variations and cope with inconsistencies of a state machine behavior. For instance, in management of transitions we can make a randomized choice or choose some

The original publication is: A. Derezińska, R. Pilitowski, Event processing in code generation and execution framework of UML state machines, in L. Madeyski, M. Ochodek, D. Weiss, J. Zendulka (Eds.) Software Engineering in Progress, Nakom, Poznań, Poland 2007, pp.80-92

transitions according to given rules. This paper deals with a problem of determination of the maximal set of non-conflicting transitions in state machines and the extended relation for transition priorities. Solutions of two selected problems of not-distinct event priorities occurring in orthogonal states are also presented.

The rest of the paper is organized as follows. Section 2 summarizes briefly basic features of the FXU framework. Next, the background and related issues are presented. Section 4 describes selected problems of event processing and their solutions implemented in FXU. Final remarks conclude the paper.

2 Code Generation and Execution Framework FXU - Overview

The Framework for eXecutable UML (FXU) is a tool for transforming UML models into executable applications. Next subsections give basic information about the FXU functionality and transformation process. More details about the system architecture and algorithms can be found in [5,6].

2.1 FXU system

The FXU system consists of two components FXU Generator and FXU Runtime Library.

The FXU Generator generates code from UML models, taking into account information from class diagrams and state machine diagrams. The target language is C#. Code generation of classes is straightforward, similar to many other solutions. Elements of UML class model are directly translated into its equivalent object-oriented concepts of the programming language. Editing a provided template of the generated code, we can adjust the generation outcome, if required.

The main advantage of the FXU Generator is its ability to generate code for all elements of state machines defined in the specification of UML 2.0 (but also current version 2.1.1) [7]. It supports all kinds of states such as simple states, composite (including orthogonal) states, entry-, do-, exit- activities and submachine states. Also all possible pseudostates are taken into account, including: initial pseudostate, deep and shallow history, join, fork, junction, choice, entry and exit point and, terminate. Transitions can be external, local or internal. They can have associated guards, actions and events. Finally, the generator processes call events, time events, completion events, change events, signals, deferred evens and priorities of event dispatching.

State machines, describing behavior of classes, are not directly translatable into programming language concepts, as classes can be. Therefore, each element of a state machine is translated into its corresponding class from the FXU Runtime Library.

The FXU Library emulates behavior of UML state machines. It consists of classes implementing all concepts common to all state machines such as a state, transition, fork pseudostate, etc. The FXU Library is responsible for event processing during execution of state machines. It services pooling and transmitting of events. The runtime library supports concurrent execution of orthogonal regions and of many state machines. The concurrent behavior is implemented with multithreading.

2.2 Transformation process

FXU transforms UML models into executable applications in C#. The transformation process can be performed in the following steps (Fig. 1). First, a UML 2.x model is created using a CASE tool. The model is exported and saved as an XML Metadata Interchange (XMI) file. We use an XMI variant supported by Eclipse plug-ins called UML2 format [8]. As an exemplary modeling tool the IBM Rational Software Architect [9] can be used.

Next, the model is transformed by the FXU generator. It imports the XMI file using the UML2 library and generates corresponding C# code. The generator analyses only classes and their state machines. Other parts of the model are ignored. For each class and its state machine, if available, the corresponding source code in C# is generated.

In the last step, the C# code is compiled by a compiler and linked against the FXU Runtime Library. The final application is build on the basis of three components: the generated source code, an additional, directly written code and classes from the FXU library. The FXU Runtime Library is required to execute state machines.

Finally, the input UML model with class and state diagrams can be executed.

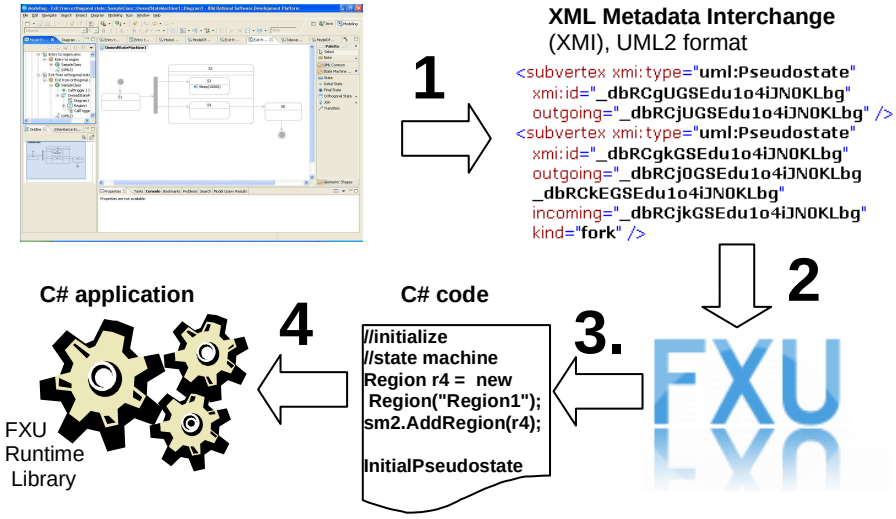

Fig. 1. Transformation process.

3 Background

While transforming state machines into the code, many interpretation problems have to be solved. The UML specification [7] leaves certain semantic options open allowing "semantic variation points". There are several such intentionally unspecified

aspects about the UML 2.x state machines. For example, the order in which events are removed from the event pool. Users can create a UML semantic variant that still compiles with the standard. Other problems are inconsistency issues - aspects of state machines unintentionally not defined by the specification.

According to the UML specification, the UML state machine formalism is an object-based variant of Harel statecharts [10]. There are different projects in developing its formal semantics that could benefit in applying formal verification methods and tools [11-17].

In [12] the extended template semantics was used to define the semantics of UML 2.0 state machines. However this approach retains the semantics variation points that are documented in the standard. Template semantics was provided for state machines implemented in three UML CASE tools. Those semantics were based partially on the informal sources and could not be proved to accurately represent the considered solutions. Another attempt is a structural operational semantics for UML statecharts proposed in [13].

Many ambiguities of UML 2.0 behavioral state machines, identified during an attempt to define the formal semantics, were discussed in [14]. The authors suggested that the concepts of history, priority, and entry/exit points have to be reconsidered. In some cases, several restrictions that might avoid incompleteness (i.e., not clear statements) or inconsistencies (i.e., contradictions within the specification) were proposed. Among others, the authors pointed out on the inconsistency in the definition of priority of joined transitions. A simplification is shown that clarifies the priories in such a case when the arcs from states on different levels come to a join pseudostate. However this case does not cover the general problem of priorities considered in Sec. 4.

There are above ten CASE tools supporting code generation from UML state machines. As showed in comparisons [6,16], many of them consider only a subset of elements presented in the UML 2.x specification. Advanced concepts, like choice pseudostates and deferred events are usually not taken into account.

Transformation capabilities for the most complete state machine model are offered by the Rhapsody tool [4] of Telelogic (formerly I-Logix). It can generate code for C, C++, Ada and Java, but not for C#. Even though ideas of execution semantics of Rhapsody influenced the current version of the UML specification [10], there are syntactic and semantic differences between them [14]. Even bigger differences could be among other solutions.

Executable UML models give an opportunity to test a model before making decisions about implementation technologies. The most progress in this area was achieved by xUML approach [2,18]. However such tools, as for example iUML, use different subsets of UML models and there is no guarantee that two interchanged models will execute the same way. There are attempts to specify a common subset of UML aimed on execution [19].

The approach described in [20] tends to incorporate different semantic variants into one meta-model. A final goal is to give a modeler an opportunity to choose among different policies implementing these variants. The authors noticed that the UML priorities do not determine selection among conflicting transitions. In the proposed approach, choosing of an arbitrary transition or usage of a randomized choice should be added to the meta-model.

Building an executable application, we have to cope with semantic problems and also choose among different implementation possibilities. However, at a modeling level we can usually accept more non-determinism than in the implementation. The FXU framework tries to keep exactly to the UML specification of state machines. For example, event triggers are not permitted after a pseudo-state (e.g. after a junction), as defined in the UML. Therefore, they will be ignored within the FXU framework even if they were allowed not only in classical statecharts [15] but also and in a modeling tool [9]. On the other hand, one implementation of some semantic variants had to be chosen and some inconsistency problems resolved. The next section explains such decisions in regard to event processing, especially transition firing and entering/exiting orthogonal states.

4 Event Processing

A state machine is described by a graph, the nodes of which are states and pseudostates. The nodes are connected by transition arcs. Traversing of transitions is triggered by dispatching of events. Behavior of a state machine is performed by switching from one to another configuration of currently active states. The FXU library implements event processing which can be viewed as one of possible behaviors satisfying the UML specification requirements. Descriptions of algorithms concerning execution of a state machine, as well as entrance and exit from a state, can be found in [5,6]. Here, we briefly recalled the general event processing schemata and discussed several inconsistency problems resolved in the FXU framework.

All processes can generate events, for example activities in states, transitions, and elements from other parts of the model (external to state machines). Events can be either broadcasted to all state machines or can be sent directly to one or more selected state machines.

Each state machine, generated from a UML model by the FXU Generator, has its event pool storing incoming events. In the UML, the order in which events are added to or removed from the pool is not defined. FXU implements an event pool as an extended First-In First-Out queue. The event queue realizes a producer-consumer algorithm. All events are controlled by FIFO policy, only completion events have the highest priority.

In UML state machines, event occurrence processing is based on the *run-to-completion* assumption. An event occurrence can only be taken from a pool and dispatched if the processing of the previous occurrence is fully completed. In FXU events are processed in the stable states of a state machine; it means in a situation, when the activities associated with the transition are completed. One event of the highest priority is removed from the queue at a time.

After selecting an event a set of currently *enabled transitions* is computed. A transition is enabled when three conditions are satisfied:

- it is triggered by an event currently selected from the event pool,
- the source state of the transition is active,
- the guard condition of the transition evaluates to true.

An event that enables no transitions and is not a deferred event will be removed from the queue and ignored. A deferred event that currently does not fire any transition will be returned to the event pool, and a next event will be selected from the queue.

In some cases, more than one enabled transition can exist. Moreover, not all enabled transitions can be fired in the same step. For example, the enabled transitions can have a common source state. Therefore a *maximal set of non-conflicting transitions M* is calculated (Sec. 4.1). The selection of which transitions will fire is based in part on priorities. Priories of transitions are calculated according to the rules given in the UML specification and one interpretation implemented in FXU (Sec. 4.2).

Finally, the selected non-conflicting transitions from the set *M* are fired concurrently (see Sec. 4.3). After all transitions had been traversed, the next event can be removed from the event pool and the entire procedure repeats.

Details concerning transitions selection and firing are given in the subsequent sections. We describe the chosen interpretations and their consequences. Further, solutions of two problems are discussed. The problems regard to interpretation of event processing in the context of orthogonal states. Those UML variation points had to be resolved within FXU framework. Resolving of other problems, including entering a history pseudostate, is beyond the scope of this paper [6].

4.1 Selection of Transitions to Be Fired

The UML specification defines *the maximal set of non-conflicting transitions* (in short set *M*) as a maximal subset in the set of all enabled transitions *E* that satisfies the following conditions:

- All transitions in set *M* are enabled,
- There are no conflicting transitions within set *M*. Two transitions are said to conflict if the intersection of the set of states they exit is non-empty.
- There is no transition outside the set (in *E-M*) that has higher priority than a transition in set *M*. (That is, enabled transitions with highest priorities are in the set while conflicting transitions with lower priorities are left out).

Let us consider a very simple example. A state *s* has two outgoing transitions *t1* and *t2* triggered by the same event, but with different guards. The event was selected from the event pool and both guard conditions are true. None of other transitions is triggered by this event. State *s* is an active state. Therefore the set of all enabled transitions (*E*) consists of two transitions *t1* and *t2*. These transitions are conflicting because they have a common source state *s*. In this case, we can distinguish two possible maximal sets of non-conflicting transitions (*M*). One set consists of one transition {*t1*}, and the second set consists also of one transition {*t2*}.

In general, enabled transitions can originate from different states, there can be any number of sets *M*, and a set *M* can comprise more than one transition.

The problem we face is selection of one maximal set of non-conflicting transitions. It corresponds to selection of one execution path among many possibilities allowed by the specification. Authors that deal with semantics of state machines usual leave this

question open. However an execution engine needs to choose one of these sets. In our opinion this selection should be specified in a justified and predictable way.

Therefore, our goal was to obtain one unique maximal set of non-conflicting transitions *M* satisfying above definition. It was determined using the following algorithm and the extended firing priorities (Def. 2 from Sec. 4.2).

Algorithm of calculating a maximal set of non-conflicting transitions

```
 1 PROCEDURE
 2     StateMachine::ComputeTransitionsToFire(Event: ev){
 3   Let E is an empty set of transitions
 4   FOREACH state s of state machine sm DO
 5         IF (s is active state)  {
 6            FOREACH transition t outgoing from s DO
 7                IF (guard of t is satisfied and
 8                      event ev triggers t)
 9                  Add t to set E
10         }
11   FOREACH transition t ∈ E DO
12         Calculate priority of t
13   Let p be the highest priority of t ∈ E
14   Let M is an empty set of transitions
15   FOREACH transition t ∈ E DO
16         IF (priority of t == p)    Add t to set M
17   RETURN M
18 }
```

The algorithm of calculating the maximal set of non-conflicting transitions selects transitions to be fired. The algorithm is evaluated separately for any state machine. A listing given above describes the algorithm for a state machine *sm* and an event *ev* currently selected from the event pool of *sm*. First, a set *E* of all enabled transitions is calculated. Therefore for each active state is checked whether it has outgoing transitions enabled by *ev* and with guard conditions that evaluate to true (lines 03-10).

Selection of transitions is performed according to priorities. They are calculated for each transition (lines 11-12). Determining of priorities is a crucial part of the algorithm; it will be presented in the next section.

The maximal set of non-conflicting transitions comprises all transitions from *E* which have the highest priority (lines 13-17). There can be more than one such transition.

It should be noted that if any two enabled transitions have the same priority they do not conflict. This condition results directly from the definition of extended priorities (Sec. 4.2). Satisfying this condition guaranties that the resulting set *M* includes non-conflicting transitions, as it consists of transitions having the same priority.

4.2 Determination of Firing Priorities

The set of transitions that can be fired is calculated using priorities of transitions. These priorities resolve some transition conflicts, but not all of them. The priorities of

conflicting transitions are based on their relative position in the state hierarchy. According to the UML specification, "a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states". However, using this definition (Def. 1) we cannot calculate one maximal set of non-conflicting transitions.

Def. 1. Firing priorities
If *t1* is a transition whose source state is *s1*, and *t2* has source *s2*, then:

1. If *s1* is a direct or transitively nested substate of *s2*, then *t1* has higher priority than t2.
2. If *s1* and *s2* are not in the same state configuration, then there is no priority difference between *t1* and *t2*.

Interpretation In order to have selected only one maximal set of non-conflicting states we have to extend the concept of firing priorities. In the extended definition we use features of state machine containment hierarchy. The *least common ancestor* of two states *s = LCA(s1,s2)* is an orthogonal state or region such that *s1* and *s2* are directly or indirectly nested substates of *s* and *s* is the mostly nested element of the state machine satisfying this feature.

Def. 2. Extended firing priorities
If *t1* is a transition whose source state is *s1*, and *t2* has source *s2*, then:

1. If *s1* is a direct or transitively nested substate of *s2*, then *t1* has higher priority than *t2*.
2. If *s1* and *s2* are not in the same state configuration, and *t1* and *t2* can be fired concurrently (do not conflict) then transitions *t1* and *t2* have the same priority (Example 1- Fig. 2).
3. If *s1* and *s2* are not in the same state configuration and *t1* and *t2* cannot be fired concurrently and traversing of *t1* implies exiting more states than traversing *t2* then *t1* has higher priority than *t2* (Example 2 - Fig. 3).
4. If *s1* and *s2* are not in the same state configuration and *t1* and *t2* cannot be fired concurrently and traversing of *t1* implies exiting exactly the same number of states as traversing *t2* than a state hierarchy is considered. Transition *t1* from *s1* is selected if region of *s1* precedes region of *s2* on the list of regions nested in *LCA(s1,s2)* - the least common ancestor of *s1* and *s2*.
5. If condition of the previous case is satisfied but *s1* has more than one enabled transition than this transition *t* is selected that comes before others in the list of transitions outgoing from the state *s1*.

Implications Determination of firing priorities implies the selection of transitions to be fired. Resolutions based on Cases 2 and 3 of the above extended definition are illustrated by two examples (Fig. 2 and 3).

Let us assume that in the state machine from Fig. 2 two transitions are enabled: *t1* from *s3* to *s4* and *t2* from *s6* to *s7*. According to Case 2 in Def. 2 they have the same priority (also no priority difference was stated by Def. 1 Case 2). In the implementation both transitions can be run simultaneously.

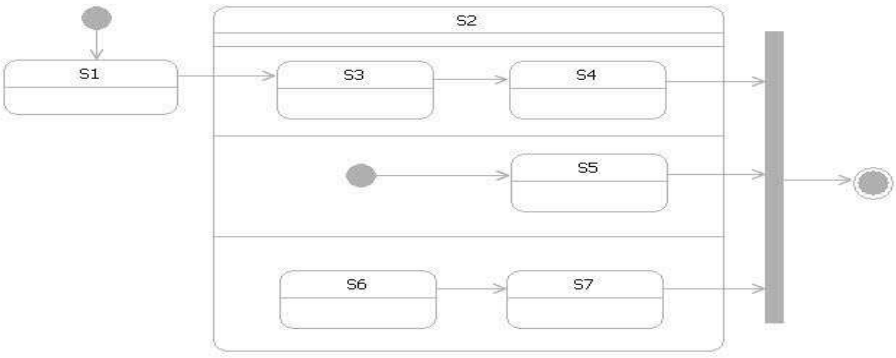

Fig. 2. Example 1 - enabled transitions s3->s4 and s6->s7 have the same priority (Def. 2 Case 2); - firing transition s1->s3 does not imply entering states s6 or s7 (interpretation Sec. 4.4).

Second example is shown in Fig. 3. Let two transitions are enabled: *t1* from *s3* to *s4* and *t2* from *s5* to *s6*. Firing of *t1* would cause exiting of only one state *s3*. While firing *t2* implies exiting also of the enclosing state *s2*. Using the interpretation given in Case 3 of Def. 2 transition *t2* has the higher priority than *t1*. Transition *t1 (s5->s6)* will be selected to fire because it exits more states than *t1*.

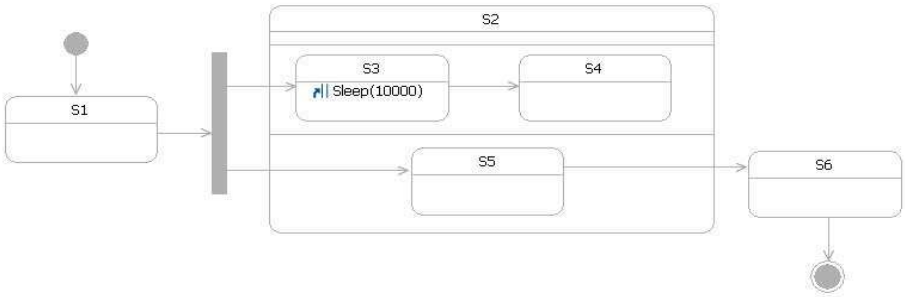

Fig. 3. Example 2 - enabled transition s5->s6 has the higher priority than transition s3->s4 (Def. 2 Case 3); - firing transition s5->s6 implies exiting states s5, s2 and s3 (Sec. 4.5).

Two last Cases (4 and 5) of Def. 2 refer to the internal structure of a state machine. They allow implementing uniquely the execution of a state machine. However they select only one of many possible behaviors (which can still satisfy the UML specification requirements). This priority determination is implementation-dependent and assumes that the sets of considered regions and transitions are ordered.

In the implementation of FXU we assume that all sets of nested regions of a state are organized in an ordered list. Similarly a set of the transitions outgoing from a state is created as an ordered list of transitions. A given UML model is provided to the FXU environment in the form of an XMI file. The order of elements in both lists is determined by the content of this file.

This solution is unfortunately an arbitrary one, as any decision based on the implementation-level information. It should be noticed that in practice, these

questionable situations occurred not very often, especially when models of state machines were carefully created. However, using the extended priorities we could obtain a decisive algorithm and, therefore, an unambiguous behavior. It chooses one possible path of execution. Another alternative, that is choosing one maximal set by random, would cause an undefined behavior. We would consider it as an erratic non-determinism. There is also so far no better proposition given, how to solve this problem.

On the other hand, the nondeterministic behavior is still present in an application created for a corresponding UML model. It is limited only to situations that are intentionally modeled by the parallel mechanisms of UML, e.g., in orthogonal regions (see the next Section).

4.3 The Order of Firing of Transitions

The maximal set of non-conflicting transitions (*M*) can contain more than one item. In this case many transitions should be fired. The UML specification does not define the order in which selected transitions should be fired. However this order can influence the behavior of a state machine.

Interpretation Transitions from a given set of non-conflicting transitions *M* should be fired concurrently because no transition should be discriminated.

Implications Concurrency within the model, among state machines and regions in an orthogonal state, was implemented with multithreading. Therefore, the transitions from set *M* selected to be fired are run in concurrent threads. Concurrent execution of transitions implies risk of incorrect access to the common data. We assume that a complete state machine model should take into account all possible orders of the transition in set *M*. If any synchronization is necessary it should be either included in the UML model or in the code generated from the model. In the later case the designer can extend its code using synchronization mechanisms supported in the .NET environment. However this solution is not recommended if a verification of a UML model using other approaches is performed e.g., model checking of state machines [3].

4.4 A Problem of Entering Regions Enclosed in an Orthogonal State

The next problem refers to entering a region that has neither an initial pseudostate nor any substate explicitly indicated by a transition incoming to the region. The UML specification states that "a transition to the enclosing state represents a transition to the initial pseudostate in each region". If a transition terminates on a substate enclosed in a region, entering the region begins in this substate (so called *explicit* entry). If such a substate of a region is not pointed to, the region execution begins in the initial pseudostate of this region (*implicit* entry).

Let us consider the following situation (Fig 2). An explicit entry transition terminates on one or many regions of an orthogonal state, here transition from *s1* to

s3. Then the explicit pointed substates are visited. Other regions enclosed in this orthogonal state (*s2*) start the execution in their initial pseudostates. However, the situation is undefined when a region enclosed in this orthogonal state has no initial state and a starting state was not explicitly determined. In the example the bottom region has no initial state.

Interpretation Regions with no initial states will be not entered at all and will be treated as completed.

Implications This solution can have serious consequences. If none of the states (*s6* and *s7*) of the given region is entered, then no outgoing transition will be fired. Also a transition outgoing from the region and targeting a pseudostate join will be not executed**.**

The state machine behavior starts with entering state *s1* and following to state *s3*. Before *s3* is entered, state *s2* enclosing *s3* will be entered. According to above interpretation the execution of regions of state *s2* is performed as follows:

- The region enclosing states *s3* and *s4* starts in state *s3* because of explicit entry from *s1* to *s3*.
- The second region, enclosing initial pseudostate and state *s5*, starts its execution in the initial pseudostate (implicit entry).
- The bottom region, enclosing states *s6* and *s7*, will be not entered.

State *s7* will be not visited and transition from *s7* to the join pseudostate will be not enabled. Therefore the compound transition outgoing from states *s4, s5, s7* through the join pseudostate and targeting the final state could be not fired.

We are not excluding this situation in a model. But it can be considered as an ill-formed model because we can not leave state *s2*.

4.5 A Problem of Exiting Regions Enclosed in an Orthogonal State

Another problem refer to firing a transition originating from a substate of an orthogonal state and targeting a state not enclosed in this orthogonal state.

Exiting many substates nested in different regions of an orthogonal state, we can use a pseudostate join, similarly as in the example in Fig. 2. However the UML specification does not exclude exiting those regions in another way.

Interpretation It is not forbidden to have a transition from a substate enclosed in an orthogonal state targeting a state outside this orthogonal state but not any join pseudostate.

Implications According to the specification, in any region at least one active node can exists at the same time. Therefore firing a transition of a kind mentioned above causes exiting all other regions of the orthogonal state and exiting this orthogonal state as well.

We illustrate this situation with an example (Fig. 3). After performing a transition from the initial pseudostate to state *s1*, a compound transition is fired coming through the fork state and entering states *s3*, *s5*, and the enclosing state *s2*.

State *s5* terminates its activity and generates a completion event that enables transition from *s5* to *s6*. This transition can be fired after terminating *entry* behavior in state *s3*. Entry actions, ones started, are always executed to completion prior to any internal behavior or transitions performed within the state.

After completion of *entry* action in state *s3* two transitions are enabled: from *s3* to *s4* and from *s5* to *s6*. According to above interpretation and according to the extended firing priorities (Def. 2 Case 3) transition from *s5* to *s6* will be fired. It results in exiting the source state *s5*, but also state *s3* and the enclosing state *s2*.

4 Final Remarks

In this paper we addressed some issues of event processing during state machine execution and solutions implemented in the FXU framework.

We presented an algorithm of assigning such priorities to the transitions that clear any inconsistency issues. Therefore this method enables selection of transitions in any state configuration. This constitutes the core of the state machine behavior. Determination of priorities depends in some special cases on implementation of a model but it does not violate the current UML specification and can be controlled by a designer.

In general, different approaches could be applied while facing some unclarities. One approach is accepting any model that follows the UML rules, and deciding its unique behavior if necessary. We showed this solution for the entering and exiting of special orthogonal states. Another one is putting more rigorous correctness conditions on state machines. For example, in [14] is suggested that "Many unclarities can be avoided by forbidding transitions crossing state borders. Instead we always use entry/exit points". In any case, it is recommended to comply with good design practices especially while modeling states with many regions.

Several attempts were initiated to define a precise formal semantics of the UML state machines [11-17]. Undoubtedly one, commonly accepted interpretation would be beneficial not only for model verification and exchange, but also for building executable applications from models. However it might not fully eliminate a necessity of more implementation-specific assumptions due to different semantic variants or non-determinism.

The FXU approach was verified on several dozen UML models. We intend to use it also for enhancing state machine comprehension and modeling skills.

Acknowledgment This work was supported by the Dean of the Department of Electronics and Information Technology, Warsaw University of Technology under grant no 503/G/1032/4000/000

References

1. Niaz, I.A., Tanaka, J.: Mapping UML Statecharts into Java code. In: Proc. of the IASTED Int. Conf. Software Engineering, pp. 111--116 (2004)
2. Mellor, S. J., Balcer, M. J.: Executable UML a Fundation for Model-Driven Architecture, Addison-Wesley (2002)
3. Knapp, A., Merz, S., Rauh, C.: Model Checking Timed UML State Machines and Collaborations. In: 7th Int. Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, pp. 395--414. (2002)
4. Rhapsody, http://www.ilogix.com/ (2007)
5. Pilitowski, R., Derezińska, A.: Code Generation and Execution Framework for UML 2.0 Classes and State Machines, In: Elleithy, K., (eds.): Advances and Innovations in Systems, Computing Sciences and Software Engineering, Springer (2007)
6. Pilitowski, R.: Generation of C# code from UML 2.0 class and state machine diagrams (in Polish), Master thesis, Inst. of Comp. Science, Warsaw Univ. of Technology, Poland (2006)
7. Unified Modelling Language v. 2.1.1, formal/2007-02-03, http://www.uml.org
8. Eclipse - an open development platform: http://www.eclipse.org (2007)
9. IBM Rational Software Architect: http://www-306.ibm.com/software/rational (2007)
10. Harel, D., Kugler, H.: The Rhapsody Semantics of Statecharts (or On the Executable Core of the UML) (preliminary version). In: SoftSpez Final Report, LNCS, vol. 3147, pp. 325--354. Springer, Heidelberg (2004)
11. Lilius, J., Paltor I.P.: Formalising UML State Machines for Model Checking. In: France, R., Rumpe, B. (eds.) UML'99, LNCS, vol. 1723, pp. 430--445. Springer, Heidelberg (1999)
12. Taleghani, A., Atlee, J.M.: Semantic Variations Among UML StateMachines. In: Nierstrasz et al. (eds.) MoDELS 2006, LNCS, vol. 4199, pp. 245--259. Springer, Berlin Heidelberg (2006)
13. Beck, M.: A Structured Operational Semantics for UML Statecharts. Software and System Modeling, vol 1(2), pp. 130--141 (2002)
14. Fecher, H., Schönborn, J., Kyas, M., Roever, W.P.: 29 New Unclarities in the Semantics of UML 2.0 State Machines. In: Lau, K-K., Banach, R. (eds) ICFEM 2005, LNCS, vol. 3785, pp. 52--65. Springer, Heidelberg (2005)
15. Crane, M., Dingel, J.: UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal. In: MoDELS/UML 2005, LNCS, vol. 3713, pp. 97-112. Springer, Heidelberg (2005)
16. Crane, M., Dingel, J.: On the Semantics of UML State Machines: Categorization and Comparison., Technical Report 2005-501. School of Computing, Queens University of Kingston, Ontario, Canada (2005)
17. References: Semantics of UML State Machines (Statechart Diagrams) STL Queen's University http://www.cs.queensu.ca/home/stl/internal/uml2/bibtex/ref_umlstatemachines.html
18. Carter, K.: iUMLite - xUML modeling tool http://www.kc.com
19. Mellor, S.J.: Embedded systems in UML, http://www.omg.org/news/whitepapers (2007)
20. Chauvel, F., Jezequel, J-M.: Code Generation from UML Models with Semantic Variation Points. In: MoDELS/UML 2005, LNCS, vol. 3713, pp. 97--112. Springer, Heidelberg (2005)