Interpretation Problems in Code Generation from UML State Machines - a Comparative Study

Anna Derezińska, Marian Szczykulski

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

Abstract. A practical utilisation of a model-driven approach to an information system development is hampered by inconsistencies causing interpretation problems. This paper focuses on the state machine that is a common means for modelling behaviour. A transformation of classes with their state machines into a code assists in the efficient development of reliable applications. A set of interpretation problems of state machines was revisited in accordance with the UML specification and examined on model examples transformed to an executable code. The paper compares the implementation of the problems regarding to two tools that support the transformation and takes into account the most comprehensive set of the UML behavioural state machine concepts. The tools are the IBM Rational Rhapsody, which transforms state machines to C, C++, Java, Ada and the Framework for eXecutable UML (FXU) dealing with the C# code. The basic information about the FXU tool is also given.

Keywords: UML state machines, statecharts, semantic variation points, UML model to code transformation

1 Introduction

The Unified Modelling Language (UML) is a notation commonly used in different phases of an information system development. UML models can be employed to describe static and dynamic aspects of a system from different viewpoints. They constitute a basis for Model Driven Engineering (MDE) [1], where model transformations are used for system development and verification. Building applications based not only on structural models, like class diagrams, but also on behavioural models is considered to be an issue of a sheer practical importance. The benefits of a model-driven approach include detecting errors of the whole system architecture and its components at earlier stages of the system development and the efficient creating of a robust application.

Interpretation problems of design models are serious obstacles in model-driven approaches. There are two main sources of interpretation problems: specification unclarities and semantic variation points. The UML dynamic semantics [2] is specified in a natural language and therefore lacks precision. Moreover, the OMG standard does not specify a number of details about the execution semantics of UML

Published in: T. Kwater (ed.) Computing in Science and Technology 2011, Monographs in Applied Informatics, Dep. of Applied Informatics Faculty of Applied Informatics and Mathematics, Warsaw University of Life Sciences. 2012, pp. 36-50.

models, including the state machines. A semantic variation point is an intentionally unspecified area, where a domain-specific refinement of the UML specification is assumed. On the whole, variation points allow different interpretations according to requirements of a modeler. Despite of this, a clarification might be necessary for some of variation points if we would like to interpret a state machine behaviour.

The UML specification [2] gives syntax and well-formedness rules (i.e. static semantics) described by meta-models and OCL constraints. New versions of UML specification improve the definitions but some inconsistencies still can be found. Specification flaws can be regarded as incompleteness, inconsistency, ambiguity or equivocality [3]. However, at the same time a certain problem can fall in different categories. Generally, this is a source of difficulties with the verification of system models and model to code transformations.

Statecharts were introduced by Harel in [4]. Incorporated into UML they become a de-facto standard for behavioural modelling in the object-oriented analysis and design. Much research on UML semantic was carried out over last years [5-13]. As far as state machines are concerned, most of papers focused on the precise description of its subset. In this paper we are not going to deal with any new semantics of UML state machines. Our aim is to review selected practical interpretation problems with state machines in the context of the current UML specification on the one hand, and its possible tool implementations on the other hand. The discussed problems refer to those considered in the literature [3, 14].

We examine a transformation of a state machine into code, which is implemented in two tools that cover the most complete set of elements of the UML state machine. The first tool is the IBM Rational Rhapsody [15,5] (formerly Telelogic and I-Logix). It supports code generation of state machines for C, C++, Java, and Ada. The second tool is the Framework for eXecutable UML (FXU) [16-18]. It was the first framework supporting all elements of UML 2.x behavioural state machines in code generation and execution for the C# code.

The reminder of this paper is organized as follows. The next two sections are devoted to the background of the work including a brief summary of the basic concepts of UML state machines (Section 2), and related work (Section 3). Different problems of state machines with their possible interpretations and solutions implemented in the tools are discussed in Section 4. Section 5 describes application of the MDE ideas in the code generation and execution framework of UML classes and their state machines - FXU. Final remarks conclude the paper in Section 6.

2 Basic Concepts of UML State Machines

This section recalls selected concepts of UML behavioural state machines defined in the specification [2].

A behavioural state machine is a graph-based notation used for description of a system behaviour, e.g. a class. The basic nodes of the graph are *states*, which can be simple states, composite ones or submachines. Graph nodes are connected by directed *transition* arcs. A traversal of the graph models the desired behaviour. During the traversal a series of activities associated with transitions or states can be executed.

A composite state either contains one *region* or is decomposed into two or more *orthogonal regions*. Orthogonal regions reflect a possible concurrent behaviour within a state. Each region can contain some substates that can, in turn, be simple or composite states, and so on.

In a hierarchical state machine more than one state can be active at the same time. All composite states directly or indirectly encompassing an active simple state are also active. If any of these composite states is orthogonal, all its regions are active. At most one direct substate is active in each region. A set of all currently active states is called an *active state configuration.*

Apart from above mentioned types of states, a graph node can be a final state or a *pseudostate*. There are ten types of pseudostates, initial pseudostate, shallow and deep history, entry and exit point, junction, choice, terminate, fork and join.

A simple transition connects two nodes of a graph. Considering behaviour as a transition execution, a concept of a *compound transition* is used. A compound transition represents a path consisting of possibly many simple transitions, originating and targeting a set of states. Internal nodes of a compound transition can be various pseudostates, according to their well-formedness rules.

A *junction* pseudostate serves as a transition merge point or a split point in compound transitions. In the latter case outgoing transitions of a split junction are labeled with guard conditions, forming a *static conditional branch.* Whereas a *dynamic conditional branch* is constituted by another pseudostate - a *choice*.

Transitions introducing concurrent behaviour are split into orthogonal regions of a composite state, or merged from such regions, using *fork* and *join* pseudostates.

History pseudostates are used to memorize recently active substates during reentrance to a composite state.

An *entry point* pseudostate is an entry point of a state machine or composite state, and an *exit point* pseudostate an exit point, accordingly.

3 Related Work

Semantics of UML state machines was studied using different approaches [5-13]. Many solutions gave detailed descriptions for a subset of the state machine concepts, and therefore could not support code generation of complete state machines. They differ also in the approaches to different semantic variation points and specification inconsistencies, for example selecting a unique, required solution, or creating a possibility for many interpretations.

The Rhapsody semantics of statecharts was described in [5]. It explains system reaction to events and realization of compound transitions, but not all concepts were fully specified, as e.g. history and orthogonal states.

Advanced or ambiguous concepts are often omitted in the description of solutions for code generation [19].

One way of dealing with semantic variation points is proposed in [20]. The authors intend to build models that specify different variants and combine them with the statechart metamodel. Further, different policies could be implemented for these variants. The important variants of time management, event selection and transition selection are shown.

In [13] the semantics of UML 2.0 state machines was defined using the extended template semantics. This approach retains the semantic variation points that are documented in the OMG specifiction. Further, it was used in the tool development that supports building a parametric code generator for Java code [21].

Crane and Dingel [14] discussed several problems of modelling constructs and well-formedness constraints of statecharts. They compared three kinds of statecharts: classical Harel statecharts implemented in Statemate, state machines as described in the UML 2.0 specification, and another variant of object-oriented statecharts implemented in Rhapsody [15]. They showed that, although all solutions are very similar, there are differences influencing creation of a model and interpretation of its behaviour.

Many ambiguities of UML behavioural state machines were discussed in [3]. The authors suggested that the concepts of history, priority, and entry/exit points have to be reconsidered. Some of these problems are addressed in this paper.

Some tools support code generation from UML state machines. They usually consider only a subset of state machine elements, whereas such advanced concepts as orthogonal regions, deep and shallow history pseudostates, deferred events or internal transitions are omitted. An exception is the IBM Rational Rhapsody tool [15] (previously Telelogic and I-Logic) that takes into account the entire state machine during transformations aimed at different programming languages: C/C++, Java, Ada. In 2010 a support of C# was also incorporated, but the code is generated only from the structural model and not from state machines.

The Framework for Executable UML (FXU), introduced in [16], performs transformation from UML class and entire state machine models into a corresponding C# code. Recently Sparx Enterprise Architect [22] was extended with code generator from state machine diagrams to various programming languages, including C#. In this case, a subset of state machine concepts is transformed into predefined code extracts that are placed directly into a considered class.

Tools building executable UML models [23,24] use also different subsets of UML and the interchanged models may not be executed in the same way. The OMG specification of Semantics of a Foundation Subset for Executable UML Models (FUML) was prepared in [25]. It defines a basic virtual machine for UML, but it considers only selected, most frequently used UML elements.

Different interpretation variants can be also treated as errors and used in the mutation testing [26].

4 Interpretation of Selected Problems and their Solutions in the Code Generation Tools for UML State Machines

Selected problems of state machines were considered not only theoretically but also appropriate benchmark models were built, transformed into the code, and tested at the application runtime. The following subsections describe these problems using the same scenario. First, a basic UML concept is briefly recalled. Then, a problem is described with the appropriate bibliographical references and is illustrated by an exemplary model(s). Next, possible interpretations based on the UML specification are presented. Then, a solution applied in the FXU tool is discussed. On top of that, the idea is related to the model behaviour given in the Rhapsody generator. For the approaches defined in the tools, we consider whether it is consistent with the specification and which interpretation is supported.

In the discussion we refer to the latest version of the UML specification [2]. It should be pointed out, that some former problems have already been resolved. For example, triggers associated with transitions outgoing a join pseudostate had an inconsistent interpretation; see also an example in [14]. In earlier releases of FXU different interpretations were implemented, but currently, since publishing of the UML version v2.2 (2008), such triggers are not allowed any more. Therefore the interpretation and the transformation can be now unambiguous.

4.1 Choice and Junction – Dynamic and Static Branches

Choice and *junction* are two pseudostates that can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions [2]. In the case of a junction pseudostate, all guards are evaluated in the source active state configuration before the compound transition that includes the junction is fired. By contrast, a choice pseudostate results, when reached, in the dynamic evaluation of the guards of its outgoing transitions.

The realization of these pseudostates will be compared using models (Fig. 1. and Fig. 2) based on the examples given in [14, Fig. 1].

Problem. A difference between choice and junction should be visible when the results of guard evaluation depend on an action of a transition incoming to the merge point. We compare a transition from state *A* when it is triggered by operation call *Move* (Fig. 1. and Fig. 2).

Interpretation. In this case there is one solution consistent with the UML specification. The next active configuration should be <ChoiceState, C> in Fig.1, and < ChoiceState, B> in Fig. 2, respectively.

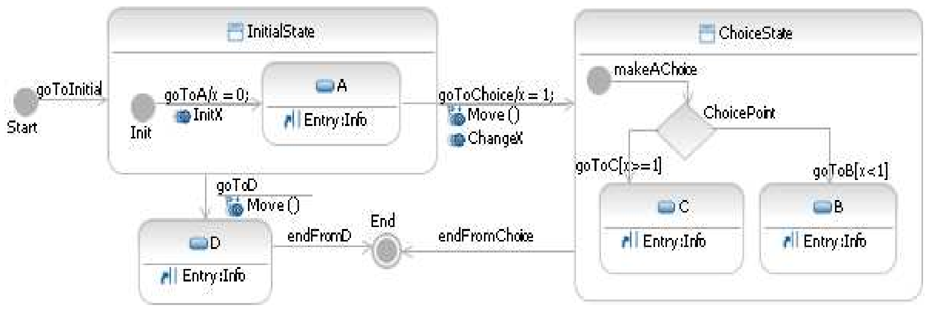

Fig. 1. State machine diagram example – choice pseudostate.

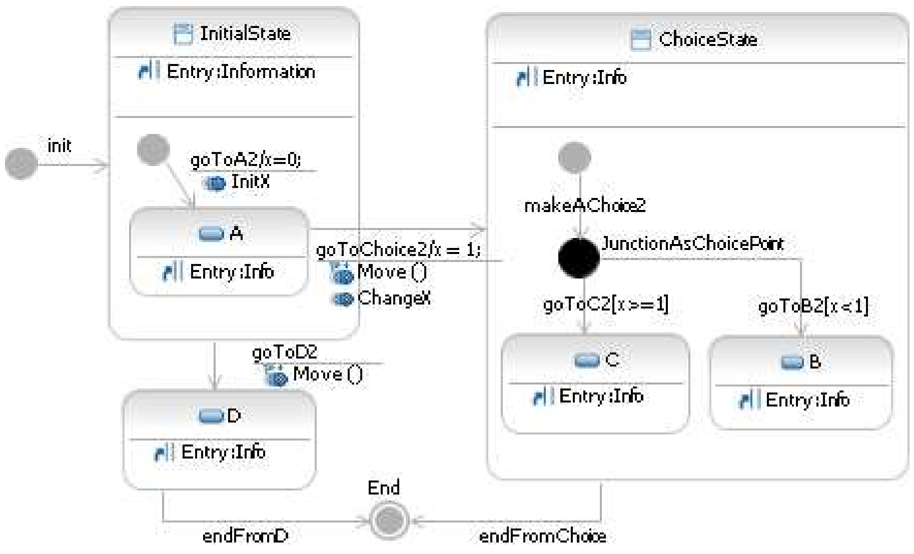

Fig. 2. State machine diagram example – junction pseudostate.

FXU solution. In the FXU tool, dynamic and static branches are implemented. In both cases the generated code and the program behaviour is equivalent to the given interpretation.

Rhapsody solution. In Rhapsody, static and dynamic branches are not distinguished. All guards are calculated after assignment of value 1 to variable *x* at the transition from *InitialState*. As for the choice example (Fig. 1), the selected state will be *C*, which satisfies the UML specification. In the second example (Fig. 2), an output from the junction pseudostate leads also to state *C*, which is not consistent with the specification.

4.2 History Pseudostate

Entering a *history* pseudostate corresponds to entering the most recently active substate of the composite state encompassing the history. In the case of a *deep history*, it is applied recursively to all levels in the active state configuration below this substate. When substates are only simple states (without further decomposition) deep history is equivalent to the shallow one.

If a composite state enclosing a history pseudostate is entered via history for the first time, or the most recently active substate is the final state, the *default history state* is entered. If the active substate determined by history is a composite state, then it proceeds with its default entry.

The history concept is a source of many interpretation and implementation dilemmas and that is why it is often not fully covered in semantic descriptions and model to code transformations. A special case of a history pseudostate used in orthogonal states was presented together with possible solutions by one of the authors in [27]. Three other problems of deep history will be discussed below and illustrated by a model (Fig. 3) based on [3, Fig.2]. A discussion about behaviour of a state machine in the case of a shallow history is similar and is omitted.

Problem A. The first problem concerns an entry to a deep history pseudostate from a state that is situated in the same region as the mentioned pseudostate. The following sequence of transitions can be considered: *goToState1, goToState6, goToState1, goToState5, goToState3, goToHistoryIn2* (Fig. 3). The last transition of this sequence enters pseudostate *DeepHistoryIn2*. Therefore the "most recently active substate" should be entered. However, this statement can be interpreted in different ways. All three interpretations shown below are consistent with the UML specification.

Interpretation A.1. The activated configuration is the one that was active before the exit from the region including the deep history pseudostate. In the exemplary sequence, the last exit from the region was the exit from *State6.* Hence the last active configuration is <*State2, State4, State6*> and it will be activated after entering *DeepHistoryIn2*.

Interpretation A.2. For a region with a history pseudostate, a recently active state is chosen before exiting this region. If the selected state is a composite one, a recently active state before exiting the main region of the state is chosen recursively. Therefore, an active configuration determined for a given sequence is **<***State2, State4, State5*>.

Interpretation A.3. A selected configuration is the recent active configuration before entering the deep history pseudostate. In this case, it is configuration <*State2, State3*>.

FXU solution. The implementation in the FXU tool follows the third interpretation. It is consistent with the way of updating history of states. The history of all states belonging to a state configuration is updated when the configuration is activated. However, the suggestion in [3] leads to the second interpretation. In this case the history of a state is updated during exiting the state, but the updating is limited only to that state and not to the whole configuration.

Rhapsody solution. By contrast to the above solutions, a deep history pseudostate in Rhapsody was implemented according to the first interpretation. This is because the history of a composite state is stored at the exit of this state.

Problem B. The second problem of history is the selection of a default state in a composite state when the recently active state was a final state of the composite state. The situation will be illustrated by a sequence of transitions *goToState1,* *goToState5*, *goToEnd4*, *goToState1*, *goToDeepHistIn2From1* (Fig. 3). If the recent state was the final one, a default state should be the state targeted by the transition outgoing from the history pseudostate, as stated in the UML specification. However, the selected default state can be a composite state, like here *State4*, without an explicit pointing to any included substate.

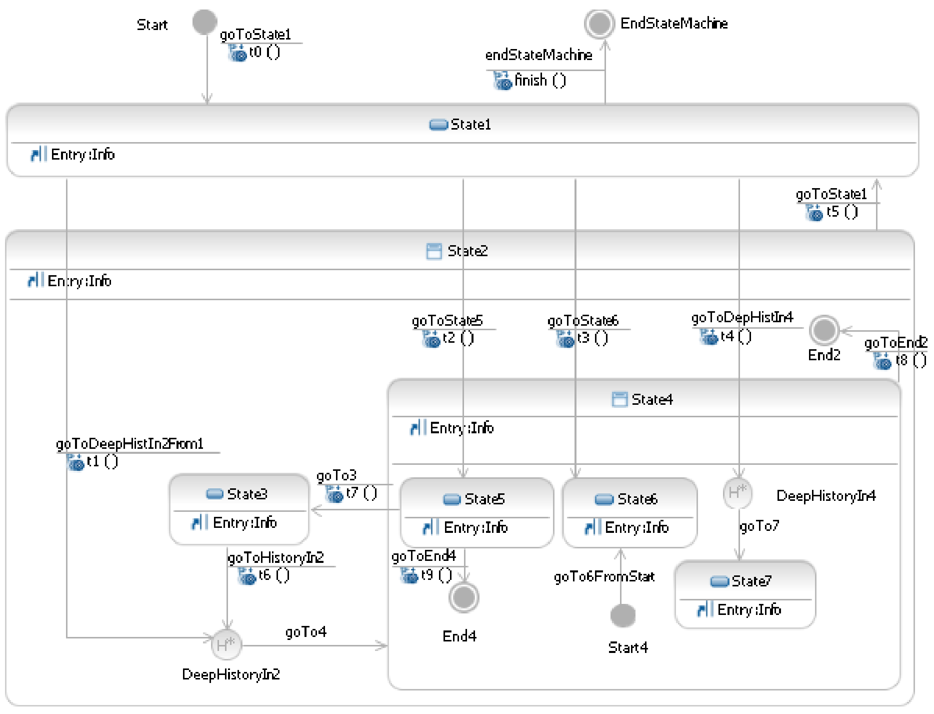

Fig. 3. State machine diagram example – deep history

Interpretation B.1. A state accessible from the initial pseudostate is selected as a default state. In this case, it is *State6.* As a result, the activated configuration is <*State2*, *State4*, *State6*>.

Interpretation B.2. Another solution is the selection of a substate that is pointed by a deep history pseudostate placed in the composite state. The approach would be performed recursively, if the selected state is also a composite one. In this case, the selected state is *State7*. Consequently, the discussed sequence leads to the active configuration <*State2*, *State4*, *State7*>.

FXU solution. A history pseudostate was implemented in the same way as in the second interpretation. This solution was also suggested in [3].

Rhapsody solution. Implementation in Rhapsody does not follow any of the above interpretations B1 or B2. The exemplary sequence results in the final state *End4* included in the composite state *State4,* and this solution is not consistent with the UML specification.

Problem C. The third problem of history concerns removing history information when a final state is entered. After reaching the final state of the composite state encompassing the history, information about visited substates should be removed. It can be illustrated by a transition sequence *goToState1, goToState5, goToEnd2, goToState1, goToDeepHistIn4.* Entering a history pseudostate when previously a final state of the composite state was visited results in selecting a default state targeted from the history pseudostate. It means that information about the history is cleared up. The question is whether this clearing should be propagated to all substates of the composite state including the history pseudostate. The problem is not directly addressed by the UML specification, but the two following, different interpretations are consistent with the specification.

Interpretation C.1. The entire information about history is removed for all regions included in the region in which the reached final state is placed. For the given transition sequence, the activated configuration is <*State2, State4, State7*>, because *State7* is directly targeted by a transition outgoing history pseudostate *DeepHistIn4*.

Interpretation C.2. Only the history of the region encompassing the final state is removed. In this case the history of *State2* including the final state *End2* is removed, and the history of *State4* included in *State2* is not. For this reason, the active configuration is <*State2, State4, State5*>.

FXU solution. A solution used in FXU is the same as the first interpretation. Similarly, [3] proposes the same solution. Moreover, the first interpretation seems to be more intuitive for a model developer, although both solutions can be accepted within the UML specification.

Rhapsody solution. The semantic of history entering after a final state is different in Rhapsody, because information about history is not cleared up. For the discussed sequence, the final state *End2* in the composite state *State2* will be obtained, which is not consistent with the UML specification.

4.3 Priority of Transitions

In a state machine, more than one transition can exit the same state and be enabled to fire at the same time. In a single run-to-completion step only one of such *conflicting* transitions will be chosen to fire, unless there are transitions in mutually orthogonal regions. The selection among conflicting transitions is based on an implicit priority. These priorities resolve some transition conflicts, but not all of them, especially in the context of composite states. The priorities of conflicting transitions are based on their relative position in the state hierarchy "a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states". A solution proposed in [28] and implemented in FXU resolves some problems of transition selection, but extends the priority definition. This helps to solve two problems of event priorities occurring in orthogonal states.

Another problem discussed in this Section, refers to priority of conflicting, compound transitions comprising enabled join pseudostates. The priority of such *joined* transitions is based on priority of the transition with "the most transitively nested source state". The problem is illustrated by the model (Fig.4) based on the example discussed in [3, Fig. 3]

Problem. There are two join pseudostates *JoinT1* and *JoinT2* in a model (Fig. 4). After receiving *makeFork* event, two substates *State1_3* and *State2_3* of the composite state *Orthogonal* belong to the active configuration. Both join pseudostates are ready to be fired. The question is which one should occur first. The decision depends on priorities of two compound transitions: <*goToJoinT1From1_3*, *goToJoinT1From2*, *goToT1*> and <*goToJoinT2From1_1*, *goToJoinT2From2_1*, *goToT2*>. The following two interpretations give contradictory results.

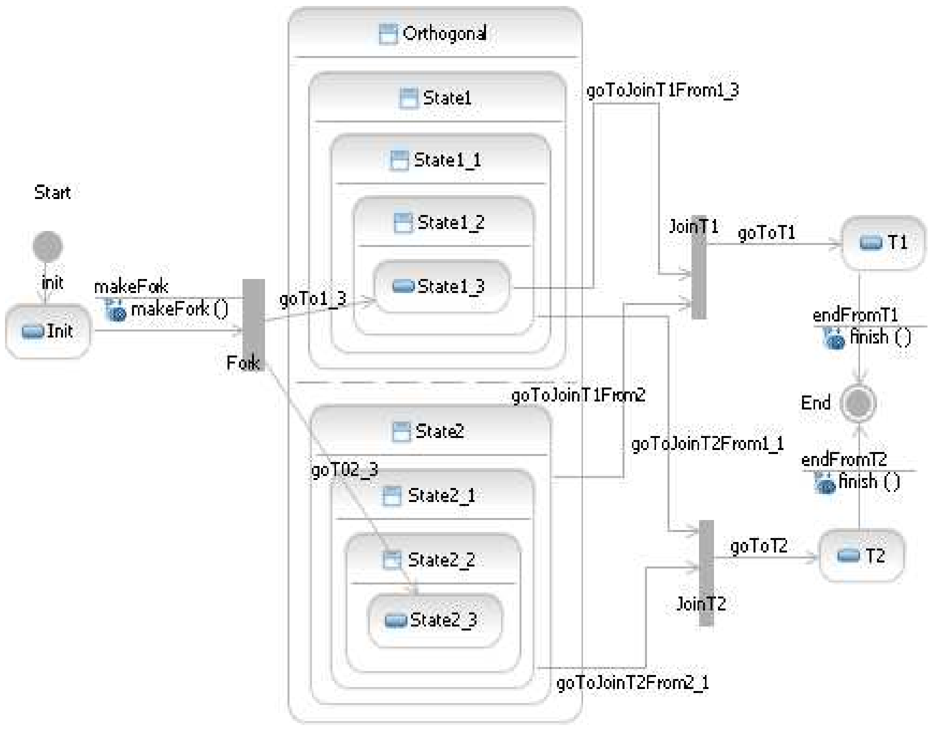

Fig. 4. State machine diagram example – priorities of joined transitions

Interpretation 1. With reference to the definition of priorities of joined transitions, given in the UML specification, a transition associated with *JoinT1* pseudostate has higher priority. Priority of a compound, joined transition is determined according to the priority of a simple transition outgoing the most nested source substate. In the case of *JoinT1* transition, the most nested state is state *State1_3*. As for *JoinT2* transition, the most nested input state is *State2_1*. Comparing the depth of nesting of those states, *State1_3* has higher priority.

Interpretation 2. According to the transition selection algorithm, the transition having *JoinT2* is chosen as one with the higher priority. Priority of a compound transition is evaluated by examining all simple transitions starting from the most nested states. When a transition that can be fired is found, the searching is finished. In this example, two states *goToJoinT2From1_1* and *goToJoinT2From2_1,* which are sufficient for firing *JoinT2,* are more nested than state *goToJoinT1From2.*

FXU solution. In the FXU environment, priorities of state machine transitions, in general, are determined based on the definition. Therefore in the considered example transition *JoinT1* would be activated, as is indicated in the first interpretation. In this case, the use of the basic definition gives an unambiguous result. In fact, the definition of priorities implemented in FXU extends one of the specification in order to give unique resolution in all situations. The extended definition is discussed in [28]. The authors of [3] proposed to use the solution given in the second interpretation.

Rhapsody solution. Priorities of composite transitions are calculated according to the priority of the transition outgoing the most nested state, as in the first interpretation. In this example the result is equivalent to that of FXU.

4.4 Entry and Exit Points

An *Entry Point* indicates a merging point of a transition from a state to a statechart submachine. Alternatively, an *Exit Point* serves as a gate point in a transition from a statechart submachine to a state. A problem concerning entry and exit points is illustrated by an example (Fig. 5) based on the discussion in [3 p.3.3].

Problem. State *Composite* includes a state machine (Fig. 5). It is accessible via EntryPoint and ExitPoint. There are several simple transitions: transition *t1* from state *Initial* to *EntryPoint* and *t2* from *EntryPoint* to *State1*. Similarly, transition *t3* leads from *State1* to *ExitPoint,* and *t4* from *ExitPoint* to *FinalState.* The question is, whether transitions <*t1, t2*> and <*t3, t4*> can be treated as composite transitions.

Interpretation 1. Regarding the UML specification, an active configuration is transformed to another one in a run-to-completion step by triggering a compound transition. From the definition, an active configuration cannot comprise a pseudostate. Entry- and exit-points are not proper states. Hence <*t1, t2*> and <*t3, t4*> are composite transitions. If so, the order of execution of actions should be determined in accordance to the semantics of a composite transition. Consequently, while traversing transition <*t1*, *t2*>, the order is as follows: *EffectT1*, *EffectT2*, *Entry:Info*. Entry action of state *Composite* will be executed after the action of transition *t2*. In the same way, for transition <*t3*, *t4*> the order will be *Exit:Info*, *EffectT3*, *EffectT4*.

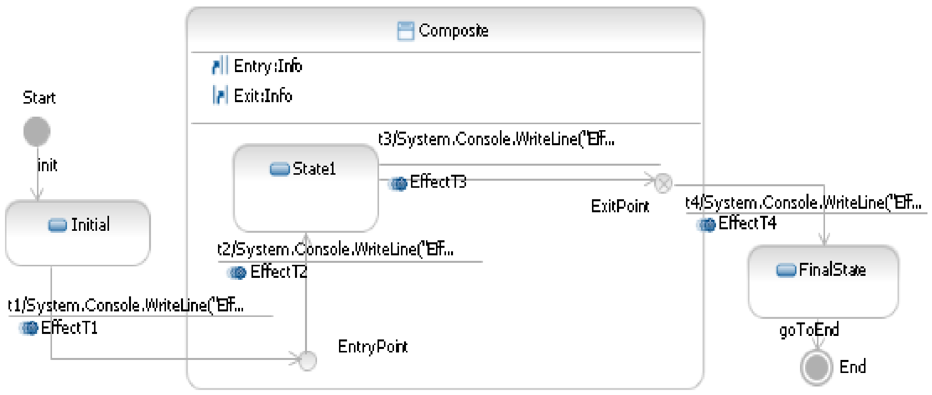
Composite
Entry:Info
Exit:Info
Start
init
Initial
State1
t3/System.Console.WriteLine("Eff...
EffectT3
ExitPoint
t4/System.Console.WriteLine("Eff...
EffectT4
FinalState
t2/System.Console.WriteLine("Eff...
EffectT2
t1/System.Console.WriteLine("Eff...
EffectT1
EntryPoint
goToEnd
End

Fig. 5. State machine diagram example – entry and exit points

Interpretation 2. Another interpretation is based on the semantics of an entry-/exit point. The rule of entering an entry point determines that if a transition enters a composite state through an entry point, then the entry behaviour is executed before the action associated with the internal transition outgoing from the entry point. In this case the order of actions would be different than in the first interpretation, namely *EffectT1*, *Entry:Info EffectT2* for composite transition <*t1*, *t2*>, and *EffectT3*, *Exit:Info*, *EffectT4* for transition <*t3*, *t4*>.

FXU solution. It was assumed that the semantics of entry/exit points will be preserved. Because of this, the order of actions follows the second interpretation. The problem will not occur when usage of transitions from/to entry/exit points is restricted. In a composite state, such transitions can only be used to link a state border with an initial or final pseudstate, as suggested in [3].

Rhapsody solution. Entry and exit points can only be used to enter/exit another state machine. They are not allowed in a composite state. Therefore an example given in Fig. 5 cannot be realized in the Rhapsody tool and in this way the problem is omitted.

5 Model-Driven Approach Supported by FXU

The Framework for eXecutable UML (FXU) generates C# code from UML classes and behavioural state machine models [16-18]. Its advantage is that it takes into account the entire set of concepts of state machines defined in the UML specification.

The first part of the FXU tool is a code generator. It reads UML models and transforms class and state machine sub-models into their C# implementation. The code includes mainly the structure of classes and state machines, as shown in an extract generated for the model given in Fig. 1. After creating all elements, like regions, vertices, transitions, in all state machines, the machines are initialized.

```
InitialPseudostate v2 = new InitialPseudostate(”Start”);
State v2 = new State (“InitialState”);
Transition t1 = new Transition (v2, v3);
```

The machine behavior is determined in the FXU Runtime Library. It provides an implementation of particular elements of UML state machines, as in the above example for the *InitialPseudostate*, *State*, *Transition*, etc. Therefore, the details of a state machine interpretation depend on the library used during the execution of an application after the generation process.

Another independent part of FXU is a tracer, which represents the state machine execution in the FXU environment in terms of model elements. This tool helps with verification of state machine behaviour and assists in error fixing.

The whole software development process using FXU can be briefly summarized in the following steps. Initially, a UML model is created in one of the CASE tools. Next, the model is exported and saved as a XML file, in fact, in the uml format of Eclipse (uml extension), which is also in accordance with the XML Metadata Interchange (XMI) specification. During experiments we used the IBM Rational Software Architect [29] to build models. The second step covers the whole code generation process from the UML models. It is done by the FXU generator. Then, a Visual Studio project of C# can be created using the FXU ApplicationWizzard. The library components for the FXU runtime are also added. Moreover, a supplementary implementation can be done directly in the program if necessary, and the application can be launched.

A new version of the tool has recently been developed. It takes into account various variants of state machine implementation, but is still consistent with the specification. A variant selection is decided by a developer at the code generation time. The variants attend to a policy of event queuing, a time and change event handling, a concurrent realization of cooperating state machines of different classes and of substates in orthogonal regions. A detailed description of the alternatives is beyond the scope of this paper.

The FXU-supported model-driven development was verified on many models covering different modelling constructs [18]. The most recent case study was related to a social network of mobile users. It simulated a presence server for the status service of a social network model. The system, its model description as well as design guidelines based on the experience of the system development are presented in [30].

6 Conclusions

Consequences of selected state machine interpretations on the code generated in an automated model transformation are discussed in this paper. A developer building a model and then transforming it to a running application should be conscious of these problems. In some cases, many interpretations consistent with the UML specification exist but it can not be decided which one is the best solution. This can depend on the context and the application domain. If all interpretations are justified, an opportunity to select one of the variants could be beneficial. Such an approach has been partially incorporated into FXU.

For example, it would be reasonable to implement various interpretations of the history concept as an option for the code generation. On the other hand, a good design practice would be to avoid using history concepts in an ambiguous way. Similarly for entry/exit points, using a restricted modelling policy can eliminate various interpretation problems.

It should be noted, that in case of many tools the implemented variants of UML model transformations are not clearly defined. However, it is important to know which solution is supported by the tool, especially if it is not consistent with the specification. Hence it would be profitable to develop a set of benchmark models, such as used in this paper, which would help in recognizing the applied solution.

An alternative to the code generation is an interpretation technique. The recent case studies shown that interpreting UML state machine, although much slower, can give acceptable results in the context of network and system management [31]. In this approach, various interpretation problems of the UML specification also need to be decided, similarly as in the code generation.

References

1 France, R., Rumpe, B.: Model-driven Development of complex software: A research roadmap. Future of Software Engineering at Proc. Int. Conference on Software Engineering ICSE'07, pp. 37-54. IEEE Soc. (2007)

2. Unified Modelling Language Superstructure v. 2.4.1, OMG Document formal/2011-08-06, (2011), http://www.uml.org/spec/UML

3 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)

4 Harell, D., Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, vol. 8, pp. 231-274 (1987)

5. 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)

6. Yeung, W. L., Leung, K.R.P.H., Wang, Ji, Dong Wei: Modelling and Model Checking Suspendible Business Processes via Statechart Diagrams and CSP. Science of Computer Programming 65, 14--29 (2007)

7. 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)

8. Jin Y., Esser R., Janneck J. W.: A Method for Describing the Syntax and Semantics of UML Statecharts, Software and System Modelling, vol. 3 no 2, 150--163. Springer (2004)

9. Jurjens, J.: A UML Statecharts Semantics with Message-Passing. Proc. ACM Symp. on Applied Computing (SAC'02) pp. 1009--1013. (2002)

10. Fecher, H., Schönborn, J.: UML 2.0 state machines: Complete formal semantics via core state machines. In: FMICS and PDMC 2006. LNCS vol. 4346, pp. 244--260. Springer, Hildelberg (2007)
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. Lano, K., Clark, D.: Direct Semantics of Extended State machines. Journal of Object Technology, vol. 6, no. 9, pp. 35-51. (2007)
13. 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)
14. 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)
15. IBM Rational Rhapsody, http://www-01.ibm.com/software/awdtools/rhapsody/
16. Pilitowski, R., Derezinska, A.: Code Generation and Execution Framework for UML 2.0 Classes and State Machines. In: Sobh, T. (eds.): Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, pp. 421--427. Springer (2007)
17. Framework for eXecutable UML (FXU), http://galera.ii.pw.edu.pl/~adr/FXU/fxu.html
18. Derezińska, A., Pilitowski, R.: Realization of UML Class and State Machine Models in the C# Code Generation and Execution Framework. Informatica, vol. 33, no 4, pp.431-440 (2009)
19. Niaz, I.A., Tanaka, J.: Mapping UML Statecharts into Java code. In: Proc. of the IASTED Int. Conf. Software Engineering, pp. 111--116 (2004)
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)
21. Prout, A., Atlee, J.M., Day, N.A., Shaker, P.: Semantically Configurable Code Generation. In: Czarnecki K. et al. (eds.) MoDELS 2008. LNCS, vol. 5301, pp. 705--720. Springer, Heidelberg (2008)
22. http: Sparx Systems. Enterprise Architect, http://www.sparxsystems.com.au/ products/ea/index.html
23. Mellor, S. J., Balcer, M. J.: Executable UML a Foundation for Model-Driven Architecture. Addison-Wesley (2002)
24. Carter, K.: iUMLite - xUML modelling tool, http://www.kc.com
25. Semantics of a foundation subset for executable UML models (FUML), ptc/2009-10-05 (2009), http://www.omg.org/ spec//FUML/
26. Trakhtenbrot, M.: Implementation-Oriented Mutation Testing of Statechart Models, Proc. of Mutation Workshop at 3rd Int. Conf. On Software Testing, Verification, and Validation, pp. 120—125. (2010)
27. Derezińska, A., Pilitowski, R.: Interpretation of History Pseudostates in Orthogonal States of UML State Machines. In: Y.i A. Feldman, D. Kraft, T. Kuflik (Eds.) NGITS 2009. LNCS 5831, pp. 26-37. Springer-Verlag Berlin Heidelberg (2009)
28. Derezinska, A., Pilitowski, R.: Event Processing in Code Generation and Execution Framework of UML State Machines. In: Madeyski, L., Ochodek, M., Weiss, D., Zendulka, J. (eds.) Software Engineering in Progress. pp. 80-92. Nakom, Poznań (2007)
29. IBM Rational Software Architect, http://www-01.ibm.com/software/awdtools/swarchitect/
30. Derezińska, A., Szczykulski, M.: Towards C# Application Development using UML State Machines – a Case Study. In: T. Sobh, K. Elleithy (Eds.) Emerging Trends in Computing, Informatics, System Sciences, and Engineering. LNEE vol. 151, Springer (2012) (to appear)
31. Höfig, E.: Interpretation of Behaviour Models at RunTime- Performance Benchmark and Case Studies. PhD Thesis, Tech. Univ. Berlin (2011)