Mid-term report: Approaches for integrating Model-Driven Engineering into industrial practice

Van Cam Pham, Ansgar Radermacher, Sebastien Gerard
CEA-List, Laboratory of Model-Driven Engineering for Embedded Systems (LISE)
Gif-sur-Yvette, France
Email: first-name.lastname@cea.fr

I. Introduction

Internet of Things [1] raises the complexity of embedded systems today rapidly. Model-Driven Engineering (MDE) is recognized as an efficient means to deal with the complexity. MDE [2] promotes abstraction and automation. The latter often relies on chaining transformations from source models at high level abstraction to target models and finally to code. Those two techniques are identified as model to model (M2M) and model to code (M2C) transformations. Among modeling languages, the Unified Modeling Language (UML) is most widely used. UML and its diagrams are much more useful to design such systems than text-based languages. Especially, in embedded system domains such as vehicle controlling, UML State Machines [3] are used as a powerful means to describe the dynamic behavior of such complex systems.

Ideally, a full model-centric approach is preferred by the MDE community due to its advantages [2]. However, in industrial practice, there is significant reticence [4] to adopt it. The reticence is due in part to the perceived gap [5] between diagram-based languages and textual languages. On one hand, programmers prefer to use the more familiar textual programming language. On the other hand, software architects, working at higher levels of abstraction, tend to favor the use of models, and therefore prefer graphical languages for describing the architecture of the system (see Fig. 1).

The survey described in [6], [7] polled stakeholders in companies who use MDE approaches. The survey reveals that UML is the prominent and dominating language. 85% of the respondents used UML for various purposes, especially problem understanding and automation such as code generation (88,2%). It notes that 70% of the respondents primarily work with models, but still require manually-written code to be integrated. Furthermore, 35% of the respondents answered that they spend a lot of time and effort synchronizing model and code. The need of synchronization is critically required by systems today. Besides the synchronization aspect, efficiency of code generated from models is a concern for the respondents.

The code modified by programmers and the model are then inconsistent. Round-trip engineering (RTE) [8] achieves synchronization between related artifacts that may evolve concurrently by incrementally updating each artifact to reflect

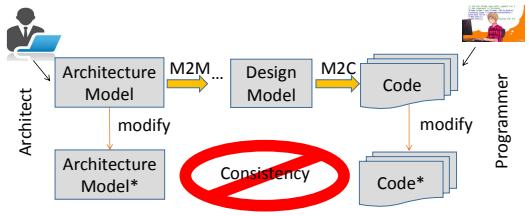

Fig. 1. Concurrent modifications made to different artifacts

editions made in the other artifacts. RTE enables actors (software architect and programmers) to freely move between different representations [9] and stay efficient with their favorite working environment.

Even, if the code is not intended to be modified by programmers, RTE is still helpful in the state-of-the-art MDE practice, e.g. debugging generated code. Some code generators generate fully operational code. The debugging support helps developers trace bugs in the generated code back to precise model elements.

The back-and-forth switching between diagram-based and textual languages, as well as between model and code, has hindered the adoption of MDE in industrial practice. To solve this issue, we believe that the sharp distinction between model and code must be blurred. We feel that a collaborative solution must be offered to different categories of developers (e.g. architects, programmers), who use different development practices.

Collaboration between developers producing different types of artifacts, in different languages, using different tools, raises the issue of artifact synchronization. This is a well-known concern with round-trip engineering [10]. It is related to traditional software engineering disciplines such as forward and reverse engineering [11].

Usually, a direct synchronizing between the architecture model and the code is hard because of the large abstraction gap. The synchronization is therefore decoupled into several synchronization steps of artifacts participating in one of the transformation phases of the chain. Hence, the whole synchronization is achieved if these steps do.

A. Research method

The research is realized by an iterative method depicted in Fig. 2. We started by studying the literature related to MDE and approaches to integrating MDE into industrial practice. We are interested in approaches for developing embedded systems using synchronization of model and generated code, which is critical as previously discussed. Specifically, we found that, (1) current approaches and tools do not support synchronization of concurrent modifications made to model and code. Furthermore, (2) the synchronization is, for UML community, only supported for available concepts of UML class diagrams while the behavior model such as UML State Machine (USM) is not supported. The reason is that there is no trivial mapping between mainstream programming languages such as C++ and Java, and the behavior model.

However, USM plays an important role in modeling and designing reactive, real-time, embedded systems. Hence, the lack of synchronization support for USMs harms enterprises to adopt UML as a development language.

It is common in MDE that code is generated, from either USMs or component models, by using templates. When models or templates change, code needs to be changed accordingly. However, when these artifacts are concurrently modified, for some reasons, making these consistent again are challenging. Some approaches using specialized comments such as *@generated* and *@non generated* to specify which code areas should be untouched or modified, respectively. By this way, when model elements and code change, code areas (user-modified code) marked by *@non generated* are preserved. It means that changes made to models or templates are not propagated to some code areas. This may lead to the dead code problem, in which the user-modified code is not used in the new generated code (see IV-B for more details). Hence, we realize (3) the problem of models, code, and template co-evolution, which is not solved by existing approaches.

Once these related problems are identified, we then focus on defining and resolving the problems by proposing different approaches, whose goal to gain the acceptance of MDE by the enterprises. The development and evaluation of the solutions are then conducted. We iteratively evaluate by going to results of evaluation, compare with other approaches, and check whether the solutions can be improved to get better results.

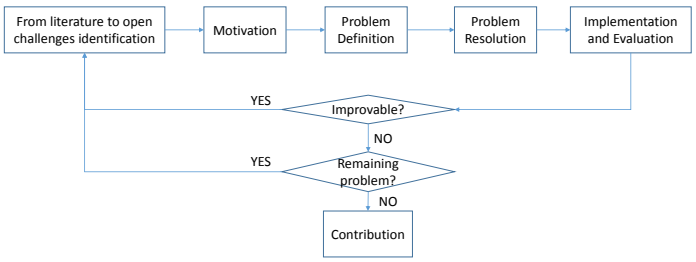

Fig. 2. Research method

B. Major topics

To sum up, the thesis can be divided into two related major parts as followings:

- **Artifact synchronization**: As previously discussed, this is critical for step-by-step integrating MDE into current software development practices and collaborating these with MDE methodologies. Particularly, synchronization of artifacts is realized by means of incremental model-to-model and model-to-text transformations. Architecture models are often at a higher level abstraction than low level implementation code. Mappings from the architecture are usually not one-to-one. This raises the difficulty in mapping code-side elements back to architecture model-side elements. The existing synchronizations (state of the art) mainly focus on static parts while the dynamics are missing.
- **Improving code generation solutions and towards synchronization for UML State Machine**: UML State Machines are widely used for reactive, real-time and embedded systems. However, the synchronization between UML State Machine and generated code is not supported by any existing approaches, even though it is critical. Furthermore, on the one hand, the OMG seeks to raise the usefulness of UML State Machine by providing more modeling concepts supporting software architects for modeling and designing complex systems. On the other hand, the support of existing generation approaches, over the years of research and development, is still limited to simple cases, especially when considering concurrency, pseudo state and event support.
- **Co-evolution of models, code, and generation templates**.

C. Objective

The transformation from architecture models to code directly involves chaining model-to-model and model-to-text transformations. Therefore, to synchronize the architecture models and code, the following objectives need to be met:

- A generic methodology, which synchronizes two different system artifacts (model and code). This method is based on incremental model transformation and synchronization. It will be used in different specific synchronization cases developed in the thesis deployment.
- A specific synchronization between UML State Machine with full features and code. Therefore, one task is needed to provide a complete generation solution, especially considering the concurrency support for real-time systems. From the generation solution, a specific synchronization methodology combined with the generic one as above is also desired.
- Synchronization of and making concurrently modified models, code, and templates consistent again.

The final goal is to provide a common framework with many facilities for collaborating MDE developers and programmers

so effectively that the acceptance of MDE in practice can be gained.

D. Contributions

The contributions of this report are the followings:

- An extensible generic methodology, which synchronize two different system artifacts (model and code).
- A specific methodology for the synchronization of UML State Machine and code.
- Brief presentation of ongoing works around model-code synchronization which are not detailed in this report.

E. Structure of the report

The remaining of this report is structured as followings: Section II presents a generic pattern for artifact synchronization in case of concurrent modifications made to model and code. Section III describes a round-trip engineering from UML State Machine (a subset) to code and back. Section IV presents some ongoing works including an approach for round-trip engineering from USMs with full features and C++ code for reactive systems, an approach for co-evolution of concurrently modified models, code, and templates, and a verification of semantic-conformance of generated code runtime execution. Section V sketches other works which are not detailed in this report.

II. A GENERIC MODEL-CODE SYNCHRONIZATION PATTERN

This section describes our generic artifact synchronization methodology pattern. For the sake of generality, we postulate that the architect and programmer are actors with starkly opposite development practices. This allows the approach to be used even in cases where model and code can both be used for the full implementation of a system, rather than just architectural design for the former, and code implementation for the latter.

A. Definitions

In this section we define the actors who will use our model-code synchronization approach to collaborate during development. Some basic concepts related to the actors and use-cases are also defined in this section.

First, we introduce the concepts of *development artifact* and *baseline artifact*.

Definition II.1 (Development artifact)**.** A development artifact is an artifact, as defined in [12], that can be used for the full implementation of the system.

In our work, we assume that model and code are both development artifacts. A development artifact may be the baseline artifact, defined in this paper as follows:

Definition II.2 (Baseline artifact)**.** A baseline artifact is one which may be edited manually. All other artifacts are produced from the baseline artifact through some process, and only through a process. Manual edition of artifacts other than the baseline artifact is forbidden.

Two primary actors, called *model-driven developer* and *code-driven developer*, are introduced.

Definition II.3 (Model-driven developer)**.** A model-driven developer is an actor in a software development process for whom the baseline artifact is the model.

The code must always be produced from the model automatically by some process that guarantees that the code is consistent with the model. A software architect is a kind of the model-driven developer who edits the model to specify the architecture of the system.

Definition II.4 (Code-driven developer)**.** Code-driven developer is an actor in a software development process for whom the code is the baseline artifact.

A programmer is a specialization of the code-driven developer. Indeed, programmers may modify the code, such as editing method bodies.

There are some use-cases for manual edition of artifacts. The `Edit Artifact` use-case implies that the IDE must have some tool to let the developer manually edit an artifact. The `Edit Model` and `Edit Code` use-cases are specializations of the `Edit Artifact` use-case where the artifact is the model or code.

There are also some use-cases related to the synchronization of artifacts. The `Synchronize Artifact` use-case (1) compares two artifacts, (2) updates each with editions made in the related artifact, and (3) reconciles conflicts when appropriate. The `Synchronize Model` and `Synchronize Code` use-cases are specializations where, respectively, the model or the code are the artifacts being synchronized.

The `Generate Code` use-case is related to forward engineering. It is the production of code in a programming language from a model. The developer can either use `Generate Code (Batch)` or `Generate Code (Incremental)`.

Definition II.5 (Batch code generation [13])**.** Batch code generation is a process of generating code from a model, from scratch. Any existing code is overwritten by the newly generated code.

Incremental code generation is a specialization of incremental model transformation, which is defined in [13] as model transformation that does not generate the whole target model from scratch but only updates the target model by propagating editions made in the source model.

Incremental code generation is defined in this paper as follows:

Definition II.6 (Incremental code generation)**.** In-cremental code generation is the process of taking as input an edited

model, and existing code, and then updating the code by propagating editions in the model to the code.

Finally, the `Reverse Code` use-case is related to reverse engineering. `Reverse Code` is the production of a model, in a modeling language, from code, written in a programming language. The developer can either use `Reverse Code (Batch)` or `Reverse Code (Incremental)`, which are defined in this paper as follows:

Definition II.7 (Batch reverse engineering)**.** Batch reverse engineering is a process of producing a model from code, from scratch. The existing model is overwritten by the newly produced model.

Definition II.8 (Incremental reverse engineering)**.** Incremental reverse engineering is the process of taking as input a edited code, and an existing model, and then updating the model by propagating editions in the code to the model.

In the next section, the use-cases of the IDE are integrated into a process that covers model-code synchronization.

B. Processes to synchronizing model and code

We propose two synchronization strategies for this scenario. The general approach behind our strategies is to represent one artifact in the language of its corresponding other artifact. These two can then be compared. For this, we define a concept of a *synchronization artifact*:

Definition II.9 (Synchronization artifact)**.** An artifact used to synchronize a model and its corresponding code is called a synchronization artifact. It is an image of one of the artifacts, either the model or the code. In this context, an image I of an artifact A is a copy of A obtained by transforming A to I. A and I are semantically equivalent but are specified in different languages.

For example, a synchronization artifact can be code that was generated from the edited model in batch mode. In that case, it is code that represents an image of the edited model (being image requires that the model is able to be reconstructed from the code).

Using the concept of synchronization artifact, two strategies are proposed in this paper: one in which the synchronization artifact is code, and the other in which the synchronization artifact is a model. The developer can choose to either use these two use-cases of the IDE. The choice may be determined by preferred development practices or the availability of suitable tools (e.g. the programmer may prefer to synchronize two artifacts, both represented in the same programming language, because he prefers to work exclusively with code).

Figure 3 shows the first synchronization strategy based on using code as the synchronization artifact. The general steps of the process shown in Figure 3 are described as follows:

Step 1 Both the model and code may be edited concurrently. (To simplify Figure 3, we don't show the Read and Write interactions for this step.) After both artifacts have been edited concurrently, we need to synchronize them.

Step 2 First we create a synchronization artifact from the edited model by generating code in batch mode. This synchronization artifact is code and it is an image of the edited model.

Step 3 The synchronization artifact is synchronized with the edited code. Since the synchronization artifact is code itself, this step is done with the `Synchronize Code` use-case of the IDE.

Step 4 Once synchronization artifact and edited code are synchronized, the former is reversed incrementally to update the edited model.

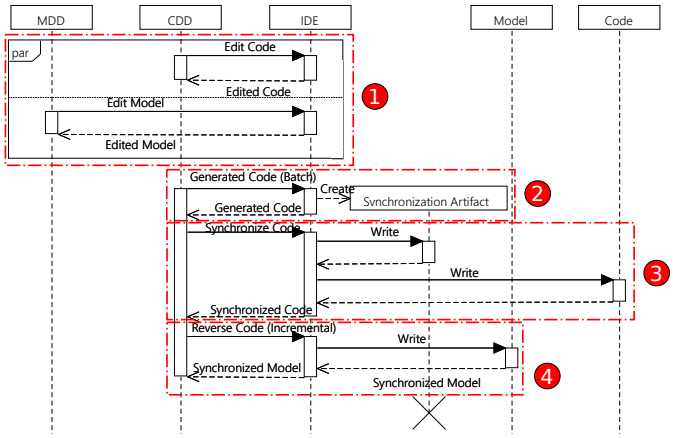

Fig. 3. Synchronization process, in which the model and the code are concurrently edited with code as the synchronization artifact (CDD = Code-Driven Developer, MDD = Model-Driven Developer). The API calls for Model and Code are represented generically as "Read" and "Write".

The second strategy, based on using model as the synchronization artifact, is the opposite of the first strategy. In the second strategy, the synchronization artifact is obtained by reversing the edited code in batch mode. Afterwards the synchronization artifact is synchronized with the edited model. Finally, we generate code incrementally from the synchronization artifact to update the edited code.

We propose two strategies based on the preferences of the developers. They may even use both strategies, successively, as a kind of hybrid strategy. This may be useful when developers want to synchronize parts of the system using one strategy, and other parts using the other strategy.

III. From UML State Machine to Code and Back

This section presents an approach, which allows to round-trip engineer model and code. At first, it sketches UML State Machine (USM) concepts supported by this study. The outline and the detail of the approach are presented afterward.

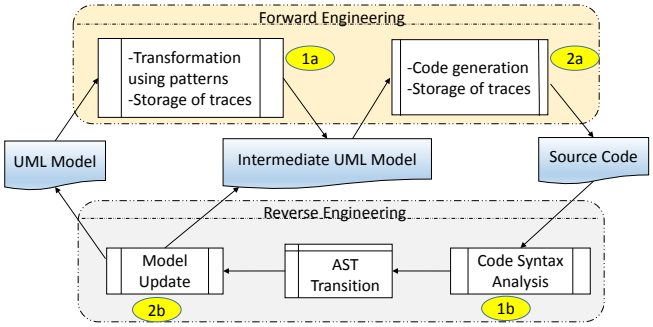

Fig. 4. Outline of state machine and code RTE

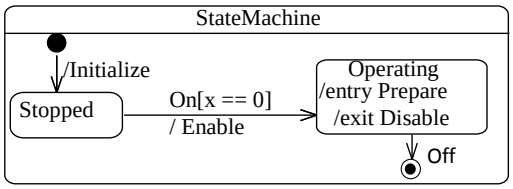

Fig. 5. An example of USM for tracing table

A. Scope

A USM describes the behavior of an active UML class which is called context class. A USM has a number of states and well-defined conditional transitions. A state is either an atomic state or a composite state that is composed of sub-states and has at most one active sub-state at a certain time. Transitions between states can be triggered by external or internal events. An action (effect) can also be activated by the trigger while transitioning from one state to another state. A state can have associated actions such as *entry/exit/doActivity* executed when the state is entered/exited or while it is active, respectively.

B. Approach outline

Our RTE approach is based on the double-dispatch pattern presented in [14] for mapping from USM to a set of classes with embedded code fragments. Fig. 4 shows the outline of our RTE approach consisting of a forward and a backward/reverse (engineering) process. In the forward process, a USM is transformed into UML classes in an intermediate model. The use of the intermediate model facilitates the transformation from the USM to code. Each class of the intermediate model contains attributes, operations and method bodies (block of text) associated with each implemented operation. The transformation utilizes several patterns which will be presented later.

When the source code is modified, a syntactic analysis process belonging to the backward transformation checks whether the modified code conforms to the USM semantics (see Subsection III-D3 for the detail of the analysis). This transformation takes as input the created intermediate model and the USM to update these models sequentially. While the forward process can generate code from hierarchical and concurrent USMs, the backward one only works for hierarchical machines excluding pseudo-states which are *history*, *join*, *fork*, *choice* and *junction*. These features are in future work.

C. From UML state machine to UML classes

This section describes the forward process which utilizes transformation patterns for states, transitions, and events to an intermediate model and code. We start by a simple USM example in Fig. 5. It consists of two states (*Stopped* and *Operating*), two external events (*On* and *Off*), transitions, and an initial and a final pseudo state. Listing 1 shows a code portion generated from the USM following our approach.

Listing 1. A segment of C++ generated code

```
class CompositeState: public State {
protected:
  State* activeSubState;
public:
  bool dispatchEvent(Event* event) {
    bool ret = false;
    if (activeSubState != NULL) {
      ret = activeSubState->dispatchEvent(event);}
    return ret || event->processFrom(this);}}
StateMachine::StateMachine(Client* ctx){
  this->context = ctx;
  stopped = new Stopped(this, ctx);
  operating = new Operating(this, ctx);
  this->setIniDefaultState();
  this->activeSubState->entry();}
void StateMachine::setIniDefaultState(){
  this->context->Initialize();
  this->activeSubState = stopped;}
bool StateMachine::transition(Stopped* state,
      On* event) {
  if(this->context->guard(event)){
    this->activeSubState->exit();
    this->context->Enable(event);
    this->activeSubState = this->operating;
    this->activeSubState->entry();
    return true;}
  return false;}
bool StateMachine::transition(
    Operating* state, Off* event) {
  this->activeSubState->exit();
  //no action defined
  this->activeSubState = NULL;
  return true;}
class Stopped: public State {
private:
  StateMachine* ancestor;
public:
  virtual bool processEvent(On* event) {
    return ancestor->transition(this,event);}
}
class On: class Event {
public:
  processFrom(State* state) {
    state->processEvent(this);}
}
class Operating: public State {
private:
  StateMachine* ancestor;
public:
  void onEntryAction() {
    context->Prepare();}
  void onExitAction() {
    context->Disable();}
```

}

1) Transformation of states: Each state of the USM is transformed into a UML class. A state class inherits from a base class, namely, *State* (the detail of this class is not shown due to space limitation). *State* defines a reference to the context class, a process event operation for each event, state actions (*entry/exit/doActivity*). A bidirectional relationship is established between a state class and the state class associated with the containing state. For example, the USM example, considered as a composite state, has attributes typed by classes associated with its contained states, *Stopped* and *Operating* in Listing 1, lines 12-13. Inversely, attributes named *ancestor* (line 36) and typed by *StateMachine* in the classes *Stopped* and *Operating* are used to associate with the parent state.

A composite state class, which inherits from a base composite class (line 1), has an attribute *activeSubState* (line 3) indicating the active sub-state of the composite state and a *dispatchEvent* operation (line 5), which dispatches incoming events to the appropriate active state.

The *dispatchEvent* method implemented in the base composite state class delegates the incoming event processing to its active sub-state (line 8). If the event is not accepted by the active sub-state, the composite state processes it (line 9).

2) Transformation of events: Each event is transformed into a UML class (see lines 41-45 in Listing 1). An event can be either a *CallEvent*, *SignalEvent* or *TimeEvent* (see the UML specification for definitions of these events). An event class associated with a *CallEvent* inherits from a base event class and contains the parameters in form of attributes typed by the same types as those of the associated operation. The operation must be a member of the context class (a component as described above). For example, a call event *CallEventSend* associated with an operation named *Send*, which has two input parameters typed by *Integer*, is transformed into a class *CallEventSend* having two attributes typed by *Integer*. When a component receives an event, the event object is stored in an event queue.

A signal event occurs if a the component receives a signal through a port. The implementation view of this scenario depends on the mapping of component-based to object-oriented concepts. We choose the mapping done in Papyrus Designer [15]. In this mapping, the signal is transferred to the context class by an operation provided by the interface of the component's port. For example, a signal event containing a data *SignalData* arrives at a port *p* of a component *C*. The transformation derives an interface *SignalDataInterface* existing as the provided interface of *p*. *SignalDataInterface* has only one operation *pushSignalData* whose body will be generated to push a signal event to the event queue of the component.

A *TimeEvent* is considered as an internal event. The source state class of a transition triggered by a *TimeEvent* executes a thread to check the expiration of the event duration as in [16] and puts the time event in the event queue of the context class.

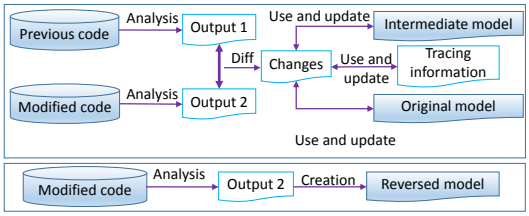

Fig. 6. Overall method for reversing code to state machine

3) Transformation of transitions and actions: Each action is transformed into an operation in the transformed context class. *Entry/Exit/doActivity* actions have no parameters while transition actions and guards accept the triggering event object. *doActivity* is implicitly called in the *State* class and executed in a thread which is interrupted when the state changes.

OnEntryAction and *OnExitAction* - abstract methods in the base state class *State* - are called by the entry and exit methods, respectively. Lines 50-53 in Listing 1 show how these methods are overwritten by the *Operating* class. *Prepare* and *Disable*, implemented in the context class, are called in these methods, respectively.

A transition is transformed into an operation taking as input the source state object and the event object similarly to DD. Transitions transformed from triggerless transition which has no triggering events accept only the source state object as a parameter. For example, the *Enable* action in the example is created in the context class and called by the transition method in lines 19-26. The guard *guard* is implemented as a method in the context class and called in line 21.

D. Reverse engineering from code to USM

This section describes the backward process.

1) Method Overall: The overall method for backward transformation is shown in Fig. 6. The modified code is first analyzed by partly inspecting the code syntax and semantics to guarantee that it is reversible. There are cases in which not all code modifications can be reversed back to the USM. The analysis also produces an output (*output2*) whose format is described later. If the intermediate model or the original USM is absent (the lower part of Fig. 6), a new intermediate model and a new USM are created from the UML model. In the contrary, the previous code taken, for instance, from control versioning systems is also semantically analyzed to have its output (*output1*) (the upper part of Fig. 6). *Output1* and *Output2* are then compared with each other to detect actual semantic changes which are about to be propagated to the original model.

Due to space limitation, we only show how to reconstruct (create) a new USM from the modified code.

2) Illustration example: To give an overview how the backward works, Fig. 7 presents a partition for mapping from the

code segments generated from the example in Fig. 5 to actual USM concepts. Each partition consists of a code segment and the corresponding model element which are mapped in the backward direction. The *Stopped* class in code is mapped to a state.

3) Semantic Analysis: The output of the semantic analysis contains a list of event names, a list of state names, a list of transitions in which each has a source state, a target state, a guard function, an action function and an event represented in so called abstract syntax tree (AST) transition [15]. For example, Fig. 8 presents the EMF [17] representation of transitions in a **C++** AST in which *IStructure* and *IFunctionDeclaration* represent a structure and a function in **C++**, respectively. Each state name is also associated with an ancestor state, an entry action, an exit action, a default sub-state and a final state. The output is taken by analyzing the AST. The analysis process consists of recognizing different patterns. Table I shows the main patterns including state, transition and event.

Algorithm 1 Semantic Analysis

Input: AST of code and a list of state classes stateList
Output: Output of semantic analysis
for s in $stateList$ **do**
  for a in attribute list of s **do**
    if a and s match child parent pattern **then**
      put a and s into a state-to-ancestor map;
    end if
  end for
  for o in method list of s **do**
    if o is $onEntryAction$ || o is $onExitAction$ **then**
      $analyzeEntryExit(o)$;
    else if o is $processEvent$ **then**
      $analyzeProcessEvent(o)$;
    else if o is $setInitDefaultState$ & s is composite **then**
      $analyzeInitDefaultState(s)$;
    else if o is timeout & s is a timedstate **then**
      $analyzeTimeoutMethod(o)$;
      $analyzeProcessEvent(s, o)$;
    end if
  end for
end for

Algorithm 1 shows the algorithm used for analyzing code semantics. Due to space limitation, *analyzeEntryExit*, *analyzeProcessEvent*, *analyzeInitDefaultState*, *analyzeTimeoutMethod* and *analyzeProcessEvent* are not presented but they basically follow the pattern description as above. In the first step of the analysis process, for each state class, it looks for an attribute typed by the state class, the class containing the attribute then becomes the ancestor class of the state class. The third steps checks whether the state class has an entry or exit action by looking for the implementation of the *onEntryAction* or *onExitAction*, respectively, in the state class to recognize the *Entry/Exit* action pattern. Consequently, event processing, initial default state of composite state and time event patterns are detected following the description as above.

4) Construction of USM from analysis output: If an intermediate model is not present, a new intermediate model and a new USM are created by a reverse engineering and transformation from the output of the analysis process. The construction is straightforward. At first, states are created. Secondly, UML transitions are built from the AST transition list. Lastly, action/guard/triggering event of a UML transition is created if the associated AST transition has these.

For example, assuming that we need to adjust the USM example shown in Fig. 5 by adding a guard to the transition from *Operating* to the final state. The adjustment can be done by either modifying the USM model or the generated code. In case of modifying code, the associated transition function in Listing 1 is edited by inserting an *if* statement which calls the guard method implemented in the context class. The change detection algorithm adds the transition function into the updated list since it finds that the source state, the target state and the event name of the transition is not changed. By using mapping information in the mapping table, the original transition in the USM is retrieved. The guard of the original transition is also created.

IV. Ongoing and future work

A. Towards synchronization of UML State Machine and programming languages

In Section III, an approach for round-tripping USMs and code is presented. However, the approach only support a sub-set of USM concepts. Furthermore, the size of binary file compiled from the generated code is larger than existing approaches using SWITCH/IF constructions (15% but not detailed here). This is not suitable to applications to be deployed on resource-constrained systems.

Synchronization of UML State Machine and code is hard, even impossible in case of resource constraints since one-to-one mappings from UML State Machine concepts to code statements are hard to achieve. To illustrate, Fig. 9 **(m1)** and **(c1)** show a USM example and its code generated by our code generator (see Section V-B). The example contains some pseudo states such as exit point, choice, and history. These are supported by only a few tools such as IBM Rhapsody and ours. It is worth noting that it is not trivial to understand the USM control flow in the generated code. Hence, programmers feel hard, even impossible, to modify the flow in the code.

For example, for some reasons, the USM should be evolved to the next version shown in Fig. 9 **(m2)**. The new version removes the exit point, changes the shallow to deep history pseudo state, and adding some states. The code should therefore be modified to **(c2)**. As discussed in Section I, this modification should be able to be realized by either programmers or using a modeling tool with a built-in code generator because of many mentioned reasons for collaboration. However, on the one hand, modifying **(c1)** to **(c2)** is non-trivial and includes much effort for testing. On the other hand, the modeling tool is usually expensive and not suitable for small enterprises (that's why many surveys on the use of MDE in practice only focus on large companies even though many advantages of MDE such as productivity and maintainability are suitable to small companies).

Our goal is to provide a mechanism from which software

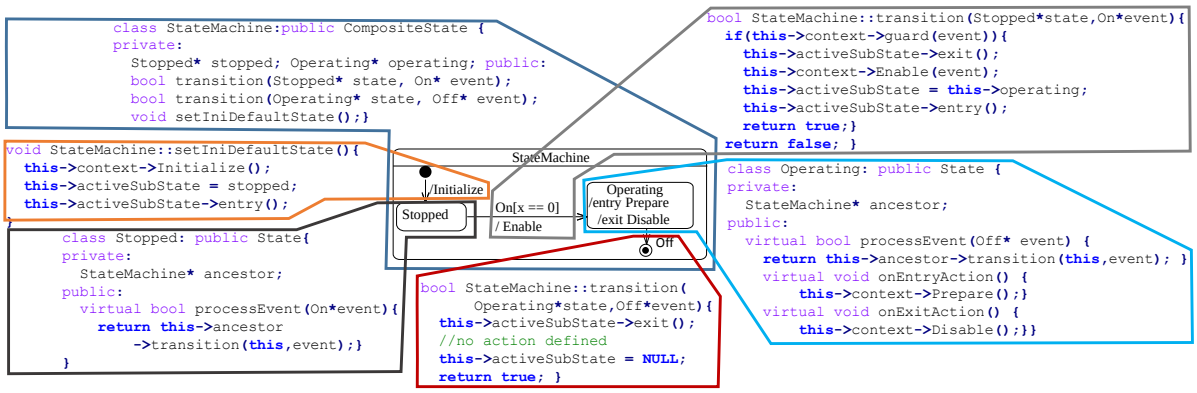

Fig. 7. USM element-code segment mapping partition

TABLE I
PATTERN RECOGNITION FOR REVERSE ENGINEERING GENERATED CODE

Pattern	Description
State	A state class inherits from the base state class or the composite base state class. For each state class, there must exist exactly one attribute typed by the state class inside another state class. The latter becomes the ancestor of the state class.
Composite state	A composite state class (CSC) inherits from the base composite state. For each sub-state the CSC has an attribute typed by the associated sub-state class. The CSC also implements a method named *setInitDefaultState* to set its default state. The CSC has a constructor is used for initializing all of its sub-state attributes at initializing time.
Entry action	If a state has an entry action, its associated state class implements *onEntryAction* that calls the corresponding action method implemented in the context class. Activity and exit patterns are recognized in the same way.
Event processing	If a state has an outgoing transition triggered by an event, the class associated with the state implements the *processEvent* method having only one parameter typed by the event class transformed from the event. The body calls the corresponding transition method of the ancestor class.
CallEvent	A call event class inherits from the base event class. The associated operation is found if the types of attributes of the event class match with the types of parameters of one method in the context class. A signal event is treated as a *CallEvent* as previously described.
TimeEvent	A transition is triggered by a *TimeEvent* if the state class associated with its source state implements the timed interface. The duration of the time event is detected in the transition method whose name is formulated as *"transition" + duration*.
Transition	Transition methods are implemented in the ancestor class, which is the class associated with the composite state owning the source state of the transition. The first parameter of the methods is the class representing the source state. The second parameter is the triggering event. Methods associated with *triggerless* transitions do not have a second parameter. The body of external and internal transition methods contains ordered statements including exiting the source state, executing transition action (effect), changing the active state to the target or null if the target is the final state, and entering the changed active state by calling entry. The body can have an if statement to check the guard of the transition.
Effect/ guard	Transition actions and guards are implemented in the context class.

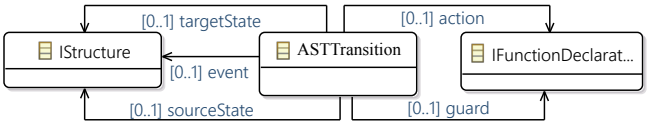

Fig. 8. Transitions output from the analysis

engineering stakeholders can profit in practice, e.g. software architects work with diagram-based languages while programmers are productive with text-based languages. To do it, we introduce newly additional constructs to existing languages. The former are conformant to the latter. Specifically, we use C++ preprocessors to seamlessly connect USM features to the code level. The idea is inspired by ArchJava presented in [18], in which the authors proposed an approach for co-evolution of architecture and implementation by providing concepts such as components and ports to Java.

```
class System {
  STATE_MACHINE(Machine) {
    INITIAL_STATE(S1, NULL) {
      INITIAL(Initial1);
      STATE(S11) {
        STATE(S111) {};
      };
      CHOICE(c1);
      EXIT_POINT(ex1);
    };
    STATE(S2) {
      SHALLOW_HISTORY(h1);
      FINAL_STATE(S21);
    };
    FINAL_STATE(S3);
    //Event table definitions
    //Transition table
    TRANSITION_TABLE {
      TRANSITION(S111, c1, NULL, NULL, NULL);
    }
  };
  //class member declarations
};
```

Listing 2. Final code after synchronization

```
#include "SMDefinitions.h"
```

In our approach, we use C++ macros to define USMs in

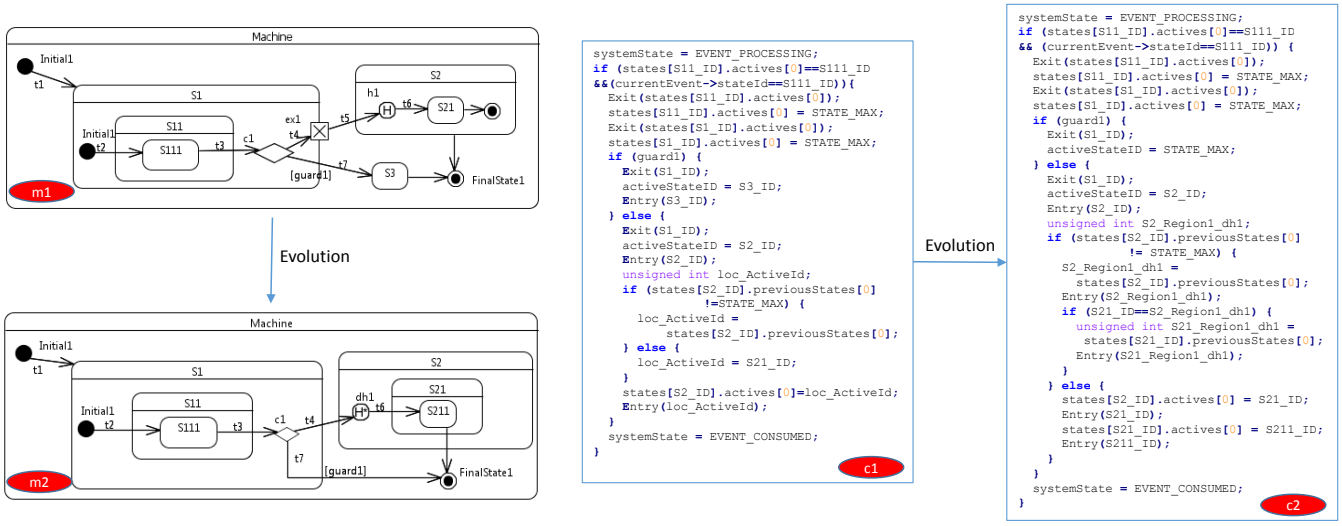

Fig. 9. State machine and code evolution example

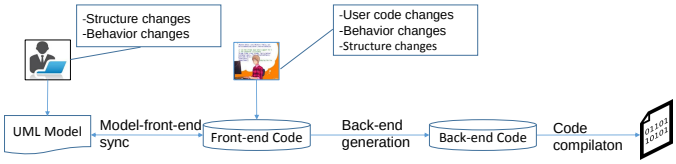

Fig. 10. Approach for synchronization of USMs and code

a front-end generated code. Instead of directly generating the runnable code, we derive a C++ front-end code (see Fig. 10), which is easy for C++ developers to integrate and modify USM in code effectively. Listing 2 shows the C++ version with additional constructs of the USM example in Fig. 9 **(m1)**. The constructs are written in the same source file of existing languages and synchronized with UML State Machine in diagrammatic forms by using the generic pattern proposed in Section II. Generated code with full UML State Machine features is achieved by using a comprehensive generation solution for USMs, which is not detailed in this report due to space limitation.

The additional constructs allow programmers productively write **UML-compliant and complete state machines** without restrictions in a textual way. **UML-compliance and completeness** focus on the full support for UML State Machine features and the semantics of generated code. This approach *adapts USMs to existing programming languages*, which differs from other text-based modeling such as Umple[1] and ThingML[2]. The latter follow: *adapting existing languages into USMs*.

[1]Umple, http://cruise.eecs.uottawa.ca/umple/
[2]ThingML, http://thingml.org/

B. Synchronization of generated and user-modified code with template evolution

In MDE, implementation code is generated from models using generation templates. When models or templates change, code needs to be changed accordingly. However, when these artifacts are concurrently modified, for some reasons, making these consistent again are challenging. Some approaches using specialized comments such as *@generated* and *@non generated* to specify which code areas should be untouched or modified, respectively. When model elements and code change, code areas (user-modified code) marked by *@non generated* are preserved. It means that changes made to models or templates are not propagated to some code areas. This may lead to the dead code problem, in which the user-modified code is not used in the new generated code. Fig. 11 shows an example consisting of an interface, a generation template, and the corresponding generated code, which is tagged using the specialized comment *@non generated code*.

Nothing remains unchanged, hence the interface, the template, and the generated code can be, for some reasons, modified as, for example, in Fig. **??**. Existing approaches preserve the user-modified code marked by *@non generated code*. Therefore, the modified code in Fig. **??** is not overwritten. Thus, the changes made to the *sum* operation of the interface and the template are not propagated to the implementation code. Alternatively, the users can force the generator to overwrite the existing code to propagate the model and template changes to the code. Consequently, the modifications made to code are lost due to the overwriting.

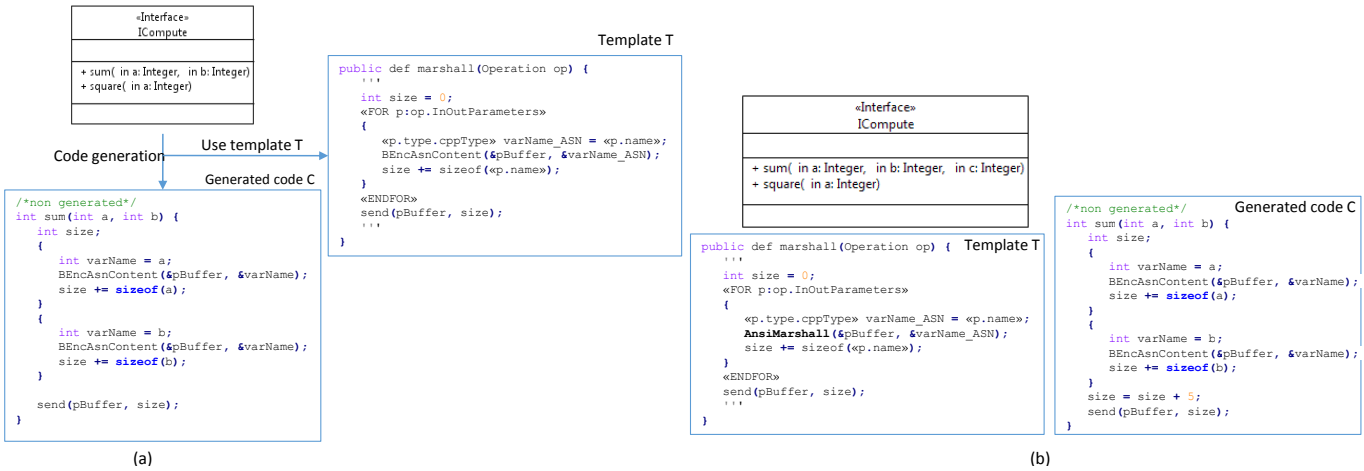

Fig. 11. (a): Interface, template and generated code; (b) artifacts evolution

Listing 3. Final code after synchronization

```
//@non generated
int sum(int a, int b, int c) {
  int size;
  {
    int varName = a;
    AnsiMarshall(&pBuffer, &varName);
    size += sizeof(a);
  }
  {
    int varName = b;
    AnsiMarshall(&pBuffer, &varName);
    size += sizeof(b);
  }
  {
    int varName = c;
    AnsiMarshall(&pBuffer, &varName);
    size += sizeof(c);
  }
  size = size + 3;
  send(pBuffer, size);
}
```

The goal of our approach is to propagate model and template changes while keeping the user modifications made to the code. Our approach generates a code *C1* generated from the modified model and template. *C1* is then compared to the code *C* generated from the origin model and template. A difference *diff1* is computed by the comparison. Similarly, another difference *diff2* is taken by comparing the previous code version and the modified code. *diff1* and *diff2* are then used for detecting conflicts and merging these differences to the code. The final version of the code after synchronization is shown in Listing 3.

C. Verification of semantic-conformance of generated code runtime execution

UML State Machine and its visualizations are widely used for modeling and designing real-time embedded systems. Although many existing approaches can generate code from state machines, the semantic conformance of runtime execution of generated code is not verified yet. The goal of this study is to provide a verification of code generated by the above generation approach.

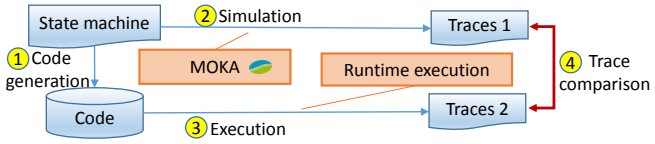

Fig. 12. Bi-simulation to test the semantic-conformance of generated code runtime execution

Two approaches can be candidates for the verification. The first is to use bi-simulation (see Fig. 12), in which a state machine is simulated by the MOKA engine [19], which is a model simulator and offers PSSM. The same state machine is also used for generating code, which is compiled and executed. The execution traces obtained from the simulation and execution are then compared to each other. Generated code for a given state machine is considered semantic-conformant, within the scope of PSSM, if both of the traces are identical. Existing approaches had little chance to do the verification since PSSM is very new. The second one is to utilize model checking techniques, which model checks the generated code conforming to a specification, which is transformed from the state machine.

V. Other Work

This section summaries other works not detailed in this report. The first one is an rule execution scheduling for rule-based incremental transformations, which are directly related to the thesis.

A. A rule execution scheduling-based incremental model transformation

Incremental model transformations (IMT) are used to synchronize different artifacts contributed by the stakeholders. IMTs detect changes on the source model and execute change

rules to propagate updates to the target model. However, the execution of change rules is not straightforward. A rule is only correctly executed if its precondition is satisfied at execution time. The precondition checks the availability of certain source and target elements involved in the rule. If a rule is executed when the precondition is false, either the execution is blocked or stopped. Therefore, the produced target model becomes incorrect. This study presents two approaches to the scheduling of change rule execution in incremental model transformations. These approaches are also applied to the case of model and code synchronization and implemented in a tool named IncRoundtrip that transforms and generates code for distributed systems. We also compare the runtime execution performance of different incremental approaches with batch transformation and evaluate their correctness.

B. A complete generation solution for UML State Machine

The most useful advantage of MDE in software engineering is to bring the ability to automatically generating code from diagram-based modeling languages such as USM to executable code [6]. However, on the one hand, OMG seeks to raise the usefulness of USM by providing more concepts to support software architects for modeling and designing complex systems. On the other hand, the support of existing generation approaches is not fully supported, especially when considering concurrency of *doActivity* and orthogonal regions, pseudo states such as history, and different events. Specifically, the following lists some issues of current approaches:

- Most of existing tools and approaches only focus on the sequential aspect while the concurrency such as the *doActivity* behavior of states and orthogonal regions is not taken into account.
- The support for pseudo states such as history, choice and junction is poor while these are very helpful in modeling.
- Issues of event processing speed, executable file size, runtime memory consumption, and UML semantic-conformance of generation code.

In order for wider integrating MDE into software development today, one task is to seamlessly make the behavior of the code generated from UML State Machine complied with the semantics specified by PSSM. The objective of this paper is to clean the above issues by offering an efficient, complete, and UML-compliant code generation solution for UML State Machine.

Our approach combines the IF-ELSE constructions and an extension of state pattern [20] with our support for concurrency. Code generated by our approach is tested under the PSSM. Although supporting multi-thread-based concurrency, binary files compiled from and the event processing speed of runtime execution of code generated by our approach are dramatically smaller than some manual implementation approaches and code generation tools such as Sinelabore [21] (which generates code from USMs created by various modeling tools such as Magic Draw [22], Enterprise Architect [23]), QM [24], which generate code for active objects [25], and C++ libraries (Boost Statechart [26], Meta State Machine (MSM) [27], C++ 14 MSM-Lite [28], and functional programming like-EUML[29]).

VI. Conclusion

This report presents the scope of the thesis, whose focus is to improve the core technologies of MDE such as artifact transformation, synchronization and code generation.

The report starts with describing the research method and major topics covered and developed in the deployment of the thesis. Different approaches around the technologies are proposed for wider integrating MDE into industrial practice to gain software quality and productivity. The proposed approaches are designed to support a continuous collaboration between software architects and programmers allowing each to use the working practices of choice

Two approaches including a generic pattern for artifact synchronization and a round-trip engineering methodology for UML State Machines and code are detailed in this report. Furthermore, the report also mentioned ongoing works focusing on complete UML State Machine and code synchronization and co-evolution of models, code, and generation templates to provide readers the comprehension for next steps of the thesis deployment. The overview of some other works is also briefly described.

References

[1] S. Li, L. D. Xu, and S. Zhao, "The internet of things: a survey," *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[2] B. Selic, "What will it take? A view on adoption of model-based methods in practice," *Software & Systems Modeling*, vol. 11, no. 4, pp. 513–526, Oct. 2012.

[3] J. A. Cruz-Lemus, M. Genero, M. E. Manso, and M. Piattini, *Model Driven Engineering Languages and Systems: 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ch. Evaluating the Effect of Composite States on the Understandability of UML Statechart Diagrams, pp. 113–125. [Online]. Available: http://dx.doi.org/10.1007/11557432_9

[4] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 633–642.

[5] N. C. C. Brown, M. Kolling, and A. Altadmri, "Position paper: Lack of keyboard support cripples block-based programming," in *Proceedings of the 2015 Blocks and Beyond Workshop*, Atlanta, GA, USA, 2015.

[6] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144–161, Sep. 2014.

[7] A. Forward and T. C. Lethbridge, "Problems and Opportunities for Model-Centric Versus Code-Centric Software Development," *Proceedings of the 2008 international workshop on Models in software engineering - MiSE '08*, p. 27, 2008.

[8] T. Hettel, M. Lawley, and K. Raymond, "Model synchronisation: Definitions for round-trip engineering," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5063 LNCS, 2008, pp. 31–45.

[9] S. Sendall and J. Kuster, "Taming Model Round-Trip Engineering."

[10] S. Sendall and J. Küster, "Taming model round-trip engineering," in *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, Vancouver, Canada, 2004.

[11] E. J. Chikofsky, J. H. Cross, and others, "Reverse engineering and design recovery: A taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13–17, 1990.

[12] OMG, "Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0," http://www.omg.org/spec/SPEM/2.0/PDF, 2008.

[13] H. Giese and R. Wagner, "Incremental Model Synchronization with Triple Graph Grammars," in *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, Genova, Italy, 2006.

[14] V. Spinke, "An object-oriented implementation of concurrent and hierarchical state machines," *Information and Software Technology*, vol. 55, no. 10, pp. 1726–1740, Oct. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913000670

[15] "Papyrus/Designer/code-generation - Eclipsepedia." [Online]. Available: http://wiki.eclipse.org/Papyrus/Designer/code-generation

[16] I. Niaz and J. Tanaka, "Mapping UML statecharts to java code." *IASTED Conf. on Software Engineering*, pp. 111–116, 2004. [Online]. Available: http://www.actapress.com/PDFViewer.aspx?paperId=16433

[17] R. Gronback, "Eclipse Modeling Project." [Online]. Available: http://www.eclipse.org/modeling/emf/

[18] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: Connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 187–197. [Online]. Available: http://doi.acm.org/10.1145/581339.581365

[19] "Moka Model Execution," https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.

[20] I. A. Niaz, J. Tanaka, and others, "Mapping UML statecharts to java code." in *IASTED Conf. on Software Engineering*, 2004, pp. 111–116. [Online]. Available: http://www.actapress.com/PDFViewer.aspx?paperId=16433

[21] SinelaboreRT, "Sinelabore Manual," http://www.sinelabore.com/lib/exe/fetch.php?media=wiki:downloads:sinelaborert.pdf. [Online]. Available: http://www.sinelabore.com/lib/exe/fetch.php?media=wiki:downloads:sinelaborert.pdf

[22] No Magic, Inc., "Magic Draw," https://www.nomagic.com/products/magicdraw.html, 2016.

[23] SparxSystem, "Enterprise Architect," http://www.sparxsystems.com/products/ea/, 2016.

[24] QM, "Qm," http://www.state-machine.com/qm/, 2016, [Online; accessed 14-May-2016].

[25] R. G. Lavender and D. C. Schmidt, "Active object," *An Object Behavioral Pattern for Concurrent*, 1995.

[26] B. Library, "The Boost Statechart Library," http://www.boost.org/doc/libs/1_61_0/libs/statechart/doc/index.html, 2016, [Online; accessed 04-July-2016].

[27] MSM, "Meta State Machine," http://www.boost.org/doc/libs/1_59_0_b1/libs/msm/doc/HTML/index.html, 2016, [Online; accessed 04-July-2016].

[28] "State Machine Benchmark." [Online]. Available: http://boost-experimental.github.io/msm-lite/benchmarks/index.html

[29] "State Machine Benchmark." [Online]. Available: http://www.boost.org/doc/libs/1_61_0/libs/msm/doc/HTML/ch03s04.html