Mid-term report: Code generation and model-code synchronization

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] 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.

Yet, still in another survey [7], the authors compare code-centric and model-centric approaches by asking the participants, who mostly come from Canada, United States and other countries such as United Kingdom, France, Indian, Australia and Singapore, questions related to different activities in software development such as fixing a bug, creating efficient software and a system as quickly as possible. The results shows that most activities tend to be easier in a model-centric approach but many participants believe that a code-centric approach is much easier a model-centric approach. This confirms the importance of automatically keeping model and code synchronized.

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 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.

To sum up, the thesis can be divided into two related major

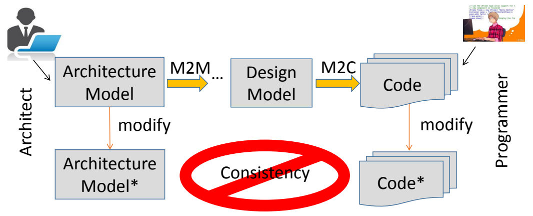

Fig. 1. Concurrent modifications made to different artifacts

parts as followings:

A. Major topics

- **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, 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 of *doActivity* and orthogonal regions, pseudo states such as history, and different events, e.g. Rhapsody does not support *doActivity*, junction or truly concurrent execution orthogonal regions[1] or Sinelabore [12] restricts choice and junction, and does not support fork, join, terminate, junction and concurrent states. Tools such as Enterprise Architect [13], Rhapsody [14] only support CallEvent and TimeEvent while SignalEvent and ChangeEvent (see III-A for formal definitions) are missing. This again enlarges the gap between the UML State Machine semantics and the actual generated code. Furthermore, the synchronization between UML State Machine and generated code is not supported by any other approaches, even though it is critical.

In reality, UML State Machine might be used for different use-cases, either with RTE or not. We consider the second topic in different use-cases. Fig. 2 shows three cases. There are

[1]IBM Rhapsody and UML differences, http://www-01.ibm.com/support/docview.wss?uid=swg27040251

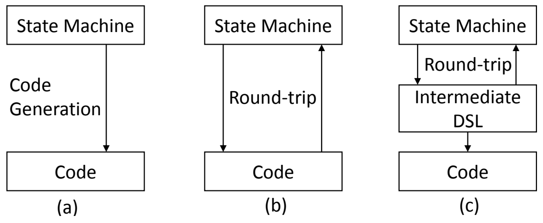

Fig. 2. Different usages of UML State Machine code generators and RTE: (a) state machines are generated to code and modifications only made to the state machines; (b) code generated from state machines is modified and synchronized directly with the state machines; and (c) generated code is modified indirectly via a text-based domain specific language (DSL).

trade-offs between these. In the use-case (a), code generators can generate efficient code from state machines with full features (this is detailed in this report). The case (b) (see V-B) only supports a small sub-set of modeling features and trades a reversible mapping with a memory overhead. The case (c) is a harmonization of the other two but restricts developers to a domain specific language (DSL).

B. 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 seamlessly make the behavior of the code generated from UML State Machine complied with the semantics specified by the Precise Semantics of UML State Machine (PSSM) [15]. To do it, the objective of this part is 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.

C. Contributions

The contributions of this report are the followings:

- An extensible generic methodology, which synchronize two different system artifacts (model and code).
- A complete generation solution from UML State Machine to code with respect to the Precise Semantics State Machine (PSSM).
- A specific methodology derived from the generic one supporting the synchronization of UML State Machine and code.

- A method for synchronization based on incremental model transformations, which prevent one artifact from begin overwritten when propagating changes from the other artifact.

D. 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 complete code generation solution from UML State Machine with respect to concurrency semantics. Section IV presents some ongoing works including a specific synchronization of UML State Machine and code, which is derived from the generic pattern and the approach as in Fig. 2 (c), 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 [16], 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 [17])**.** 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 [17] 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.

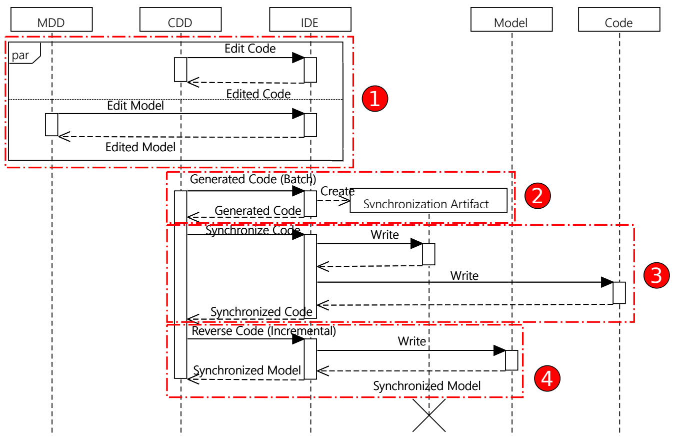

Fig. 3. Synchronization process, in which the model and the code are concurrently edited with code as the synchronization artifact (CDD = Code-Driven Developer, MDD = Model-Driven Developer). The API calls for Model and Code are represented generically as "Read" and "Write".

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. A COMPLETE CODE GENERATION SOLUTION FOR UML STATE MACHINE

This section describes a complete generation solution for UML State Machine, especially when considering the concurrency aspect. It first reminds background definitions in a formal way and some assumptions on output generated code. Subsection III-B continues with the concurrency design for generated code based on multi-thread. Overview of a code generation approach based on the design is then discussed.

A. Background definition

Definition III.1. A directed graph $G = \{V, Ed\}$ consists of a finite set V of vertexes, and a set Ed of edges. An edge connects a source vertex to a target vertex. The source and target vertexes of an edge ed are obtained by $src(ed)$ and $tgt(ed)$.

Definition III.2. A UML vertex $v \in V$ has a kind $v.kind \in$ *{initial, final, state, comp, conc, join, fork, choice, junction, enpoint, expoint, history}*.

Definition III.3. A region $r \in \mathcal{R}$ is composed of one or several vertexes, and contained by a state s. We write $owner(r) = s$ and $vertices(r)$ is its sub-vertices set.

Definition III.4. A vertex is either a UML state or a pseudo-state. A UML state s is a vertex and $s.kind \in$ {state, comp, conc}. s has an $entry$, an $exit$ and a $doActivity$ action. A composite state cs contains one or more vertexes. We write $vertices(cs)$ is a set of vertexes contained by cs and, inversely, $owner(v)$ refers to the containing state of the vertex v.

Definition III.5. An action $act \in ActLang$ is a set of statements written in an object-oriented programming language $ActLang$. A guard is a boolean expression written in $ActLang$.

Definition III.6. A transition $t \in T$ is an edge connecting two vertexes. A transition has a guard $guard(t)$, an effect $effect(t)$, and is associated with a set of events $\subset$ E. We write $events(t)$ as the associated set of events. A transition has a type $t.type \in \{trig, tless, gdless, triggdless\}$ and a kind $t.kind \in external, local, internal$.

Definition III.7. An event is one of the followings:

- A *TimeEvent* te specifies the time of occurrence d relative to a starting time. The latter is specified when a state, which accepts the time event, is entered.
- A *SignalEvent* se is associated with a signal sig, whose data are described by its attributes and is occurred if sig is received by a component, which is an active UML class.
- A *ChangeEvent* che is associated with a boolean expression $ex(che)$ written in $ActLang$. che is emitted if $ex(che)$ changes from true (false) to false (true).
- A *CallEvent* ce is associated with an operation $op(ce)$. ce is emitted if there is a call to $op(ce)$.

Suppose that for each vertex $v \in V$, its incoming and outgoing transition lists are extracted by the functions $incomings$ and $outgoings$, respectively. If $v.kind = conc$, suppose $regions(v)$ is the region set contained by v.

The behavior of an active class C is described by using a state machine whose definition is as following:

Definition III.8. A state machine sm is a graph specified by $\{V, T\}$ associated with a set of events E. A state machine is a special composite state which has no incoming and no outgoing transitions. A root vertex v is a direct sub-vertex of the state machine, $owner(v) = sm$. The set of regions contained by sm is written $\mathcal{R}$.

For each vertex $v \in V$, we write the following sets $T_{ins}(v) = incomings(v), T_{outs}(v) = outgoings(v)$, $t_{first} = head(t_{outs})$; transitive transition sets $T^{+}_{ins}(v)$ and $T^{+}_{outs}(v)$ are sets of transitions incoming to and outgoing from, respectively, v or direct or indirect sub-vertexes of v.

Definition III.9. Transitive container $owner^{+}(v)$ of a vertex v of a state machine sm is defined as following:

$$owner^{+}(v) = \begin{cases} sm & owner(v) = sm \\ owner(v) \cup owner^{+}(owner(v)) & otherwise \end{cases} \tag{1}$$

Likewise, $vertices^{+}(v)$ is a set of transitive sub-vertexes.

A state machine $sm = \{V, T\}$ is validated if an associated set of constraints is validated. The set is not presented here due to space limitation.

Definition III.10. Current active configuration Cfg of a UML state machine sm is a set of candidate UML states which are able to process an incoming event.

B. Thread-based Concurrency for UML State Machine

This Subsection describes our design for generated code. The design is based on an example-based analysis, which is not presented here due to space limitation.

1) Thread-based design of generated code: The concurrency of concurrent UML State Machines is based on multi-thread, in which there are permanent and spontaneous threads. While permanent threads (PTs) are created once and live as long as the state machine is alive, spontaneous threads (STs) are spawned in active for a while. Each PT is initialized at the state machine initialization. The design of threads is based on the thread pool pattern, which initializes all threads at once, and the paradigm "wait-execute-wait". In the latter, a thread **waits** for a signal to **execute** its associated method and goes back to the **wait** point if it receives a stop signal or its associated method completes. Each PT is associated with one of the following actions:

- *doActivity* of each state if has.
- Sleep function associated with a *TimeEvent* which counts ticks and emits a *TimeEvent* once completes.
- Change detect function associated with a *ChangeEvent* which observes a variable or a boolean expression and pushes an event to the queue if changes happen.
- State machine main thread,which reads events from the event queue, and sends start and stop signals to these initial threads.

Now we consider STs which are spawned by a parent thread, joined until and destroyed once the associated methods complete.

The STs follow a paradigm in which if a thread $parent$ spawns a set of threads $children$, $parent$ must wait until $children$ complete their associate methods. These threads are spawned in one of the following cases:

- A thread is created for each effect of transitions' outgoing from a *fork* or incoming to a *join*.

- Entering a concurrent state s, after the execution of $entry(s)$, a thread is also created for each orthogonal region.
- Exiting a concurrent state s, before the execution of $exit(s)$, a thread is also created for each region to exit the corresponding active sub-state.

2) Deadlock avoidance: Each PT is associated with a mutex for synchronization communication in the multi-thread-based generated code. The mutex must be locked before the method associated with the thread is executed. The mutex associated with the main thread preserves the run-to-completion semantics since some event such as *CallEvent* can be processed synchronously and some asynchronously. Each event processing must lock the main mutex before executing the actual processing.

C. Assumption

Assuming that we want to generate from the state machine to an object oriented programming language *ActLang*, which is C++-like and supports multi-threading as following functions and resource control as mutexes.

- A mutex has three methods $lock$, $unlock$, and $wait$, which automatically unlocks the mutex and waits until it receives a signal.
- *FORK(func)* creates a thread (lightweight process) associated with the function/method *func* and *JOIN(theThread)* waits until the method associated with the thread *theThread* completes.

D. Code generation pattern

1) State transformation: A common state interface $IState$ is created. The interface contains three methods, namely, *entry*, *exit*, and *doActivity* respectively corresponding to three state actions. To preserve the hierarchy of composite states, the interface also has two attributes called *actives* and *previousActives* referring to current and previous active sub-states in case of the presence of history states, and a list of deferred event identifiers.

Each UML state is transformed into an instance of the interface associated with a state ID (which is a child element of an enumeration) inside the active class C. During initialization, each instance delegates its methods to suitable implementation, e.g. function pointers in C++. For example, Listing 2 in Appendix shows the interface and its instances.

Each *doActivity* is associated with a permanent thread and a mutex. The *doActivity* thread is initialized, waits for a start signal, executes the *doActivity* code, generates a completion if the state is atomic and still active, and goes back to the waiting point as the paradigm above. Listing 3 in Appendix shows a code segment for *doActivity* threads.

2) Region transformation: Each region is transformed into an entering and exiting method. While the entering method controls how a region r is entered from an outside transition t

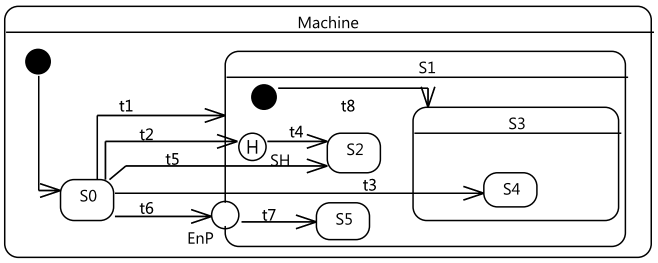

Fig. 4. Example illustrating different ways entering a composite state

($src(t) \notin vertices(r)$), the exiting method exits completely a region by executing exit actions of sub-states from innermost to outermost.

A region r is entered by either a transition t ending at the border of its containing state or on a sub-vertex (direct or indirect), depending on how the state machine is designed. The following lists different ways r may be entered:

- Way 1: entering by default: $tgt(t) = owner(r) \wedge src(t) \notin vertices(r)$.
- Way 2: entering on a direct sub-vertex: $tgt(t) \in vertices(r) \wedge src(t) \notin vertices(r)$.
- Way 3: entering on an indirect sub-vertex: $owner(tgt(t)) \in vertices^{+}(r) \wedge src(t) \notin vertices(r)$.

All of the entering ways execute the entry action of the containing composite state $entry(owner(r))$ after $effect(t)$. *doActivity(owner(r))* is then signaled to be run in its associated thread. The actions afterwards are different for each way. To illustrate, we use an example as in Fig. 4 with *S1* as a target composite state. *t1*, *t3* and *t6* are in the ways of 1 and 3, respectively, while *t2, t5* in the way 2.

The entering method associated with the region r of *S1* has a parameter $enter\_mode$ indicating how actions should be executed. The number of modes depends on the number of transitions coming to the composite state $S1$ specified as: $\#modes(s) =$
$\#\{v \in vertices(s) | v.kind = initial\} +$
$\#\{v \in vertices(s) | \exists t \in T_{ins}(v), src(t) \notin vertices(s)\} +$
$\#\{v \in vertices^{+}(s) \setminus vertices(s) | \exists t \in T_{ins}(v), src(t) \notin vertices^{+}(s)\}$. The detail of how these modes are implemented in specific languages are not discussed here. The readers are recommended to read the example in Listing 4 in Appendix.

Transitioning from a vertex to a pseudo-state of the composite state (transition from $S0$ to SH is a particular case) is not as simple as that of two states. It needs a systematic approach which generates code for a transition outgoing from a vertex to any other one. This is detailed in the next section.

3) Event and transition transformation:

a) Events: An event enumeration *EventId* is created whose children are event identifiers associated with events. Each event e) is also transformed into a method mtd_e in the context class C. Besides the explicitly defined events of the state machines, the event list contains a special event called $CompletionEvent$, which is implicitly implemented as an asynchronous *CallEvent*. For each event type, the transformation is realized as followings:

- *CallEvent ce*: The associated operation $op(ce)$ can be either synchronous or asynchronous. When $op(ce)$ is called, it waits and locks the main mutex protecting the run-to-completion semantics, and executes mtd_{ce}. Contrarily, the parameters of the asynchronous operation are used to create a signal which is transformed similarly to the case of $SignalEvent$.
- *SignalEvent se*: $SignalEvent$ is asynchronous. The signal associated with se is written into the event queue of the active class C by an operation which takes as input the signal.
- *TimeEvent te*: A thread $teThread$ associated with te is created and initialized at the initialization of the state machine. Within the execution of $teThread$, the method associated with te waits for a signal, which is sent after the execution of the entry of a state $s \in \{v \in V | \exists t \in T_{outs}(v), te \in events(t)\}$, to start sleeping for a duration d of te. When the duration expires, te is emitted and written to the event queue if s is still active.
- *ChangeEvent che*: Similar to $TimeEvent$, a thread $cheThread$ is initialized at initialization but the associated method mtd does not wait for a signal to start. mtd periodically checks whether the value of the associated boolean expression $ex(che)$ changes by comparing the current value with the previous value. If a change happen, che is committed to the event queue.
- *Any*: any of the above events can trigger the associated transitions.

As above presented, all asynchronous incoming events are stored in a FIFO priority queue, in which each event type has a configurable priority. $CompletionEvent$ always has the highest priority. Others are equal by default.

b) Transitions: Each event is associated with a list of transitions. We suppose $T_{trig}(e)$ is the transition list, which can be triggered by the event e, and $S_{trig}(e) = \{src(t) | t \in T_{trig}(e)\}$. In other words, $S_{trig(e)}$ is a set of states which are the source states of the transitions in $T_{trig}(e)$. To present how the body of event methods is generated, we define functions as followings:

- Vertex depth $dp(v)$ is defined as: $dp(v) = 1$ if v is a root vertex, otherwise $dp(v) = dp(owner(v)) + 1$.
- $Map_e(s) \subset S_{trig(e)} | \forall sub \in Map_e(s) : owner(sub) = s$, $Prt(e) = \{s \in V | Map_e(v) \neq \emptyset\}$. $Prt(e)$ is an ordered list whose length is $len(Prt\{e\})$ and elements are accessed by index (get). The order of $Prt(e)$ is defined as: $\forall i, j \leq len(Prt\{e\})$,
if $i < j, dp(Prt(e).get(i)) \geq dp(Prt(e).get(j))$.

The procedure in Listing 1 describes how the generation process works with an event. It first finds the innermost active states which are able to react e by orderly looping over Lm_e. For each transition outgoing from an innermost state, code for active states and deferral events, guard checking and transition code segments are generated by $GENERATE\_STATE\_EVENT\_CHECK$, $GENERATE\_GUARD(t)$ and *GENTRANS*, respectively. If the identifier of e is equal to one of the events listed in $defEvents$ of the corresponding state (not shown in this paper), it is deferred by putting it to a deferral event queue managed by the main thread, which also pushes the deferred events back to the main queue once one of the pending events is processed.

Listing 1. Generation process for an event

```
∀ item ∈ $Lm(e)$
  $\forall s \in Map_e(item)$
    $T_s = \{t \in T_{trig}(e) | src(t) = s\}$
    $\forall t \in T_s$
      $GENERATE\_STATE\_EVENT\_CHECK(s, t, e)$
      $GENERATE\_GUARD(t)$
      $GENTRANS(s, t, tgt(t))$
```

Depending on the target of t, $GENERATE\_STATE\_EVENT\_CHECK$ can generate single or multiple active state checking code. The latter occurs if $tgt(t)$ is a $join$. The detailed discussion on these is not presented due to space limitation. Listing 5, line 1-2, shows an example for multiple checking.

Generally, *GENTRANS* generates code for transitions between any vertexes satisfying the constraints described in Section III-A. The detail of *GENTRANS* is not presented here. Each pseudo state is transformed as the followings:

- $join$: Use $GENTRANS$ for v's outgoing transition.
- $fork$: Use $FORK$ and $JOIN$ for each of outgoing transitions of v.
- *choice*: For each outgoing, an $IF - ELSE$ is generated for the guard of the outgoing together with code generated by $GENTRANS$ (see Listing 5).
- *junction*: As a static version *choice*, a $junction$ is doubly evaluated. The first evaluation is before any action executed in compound transitions (see Listing 5). The output value of the first evaluation is used to determine which transition outgoing from $junction$ is taken in a second evaluation.
- *shallow history*: The identifiers of states to be exited are kept in *pres* of $IState$. Restoring the active states using the history is exampled as in Listing 4. The entering method is executed as default mode at the first time the corresponding composite state is entered (see Listing 4). The previous active sub-states are updated by saving the active state identifier to *previousActives* before exiting the region containing the history.
- *deep history*: Saving and restoring active states are done at all state hierarchy levels from the composite state containing the deep history down to atomic states. Updating *pres* is committed before exiting the region, which is directly or indirectly contained by a parent state, in which a deep history is present.

- $enpoint$: If $enpoint$ has no outgoing transition, the corresponding composite state is entered by default. Otherwise said, $GENTRANS$ is called to generate code for the outgoing transition.
- $expoint$: The code for the unique transition outgoing from $expoint$ is generated by using $GENTRANS$.
- $terminate$: The code executes the exit action of the innermost active state, the effect of the transition and destroys the state machine object.

IV. Ongoing and Future Work

A. Towards synchronization of UML State Machine and programming languages

1) Problem statement:

- Although code generation from UML State Machine is proposed by multiple approaches, a complete generation solution is not achieved yet (see I-A). Furthermore, none of existing code generators allows round-trip engineering or synchronization. This is, as previously discussed, one of the reasons why software development practitioners are reluctant to adopt MDE into practice.
- 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. Even, in simple cases where only states and transitions are used, the generated code size has a slightly larger memory overhead (see V-B) compared to traditional approaches such as if/else or nested switch statements.

2) Idea overview:

a) ***General idea****:*
The general idea is to provide a mechanism from which software engineering stakeholders can profit in practice, e.g. software architects work with diagram-based languages while programmers are productive with text-based languages. This study is the realization of the approach in Fig. 2. To do it, A text-based domain specific language is proposed. This proposed language is descriptive and seamlessly integrated with diagrams and programming languages. The idea is inspired by the model-oriented programming language *Umple*[2]. The purpose of the proposed language is to provide a text-based DSL compliant to the UML State Machine specification, which is not correctly defined by Umple.

The language allows 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, and are key characteristics used to differentiate this proposed language from other non UML-compliant languages such as Umple and ThingML[3]. This proposed language is 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 the generation solution in Section III. This language is a convergence of the generic pattern and the complete code generation to provide an efficient and productive way to develop reactive, real-time and embedded systems.

b) ***Advantages****:*
Followings are advantages of this proposition:

- Software architects are freely to describe the behavior of systems by using state machine diagrams.
- Programmers are freely to work with a text-based environment to be productive.
- State machines used for generating code are not restricted to simple or non-concurrent cases as in other approaches.
- At anytime, generated code and software architects-built diagrams are consistent by using the generic synchronization pattern.

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

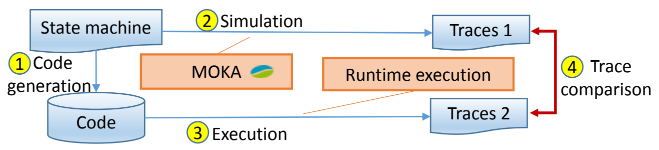

Fig. 5. Bi-simulation to test the semantic-conformance of generated code runtime execution

B. 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.

Two approaches can be candidates for the verification. The first is to use bi-simulation (see Fig. 5), in which a state machine is simulated by the MOKA engine [18], 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. This work is published in **MODELSWARD** 2016[4]. The second one is a round-trip engineering approach for UML State Machine and object-oriented code. A paper based on this work is submitted to **SEW**[5] 2016.

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. From UML State Machine to code and back again

Although many industrial tools and research prototypes can generate executable code from such a graphical language, generated code could be manually modified by programmers. After code modifications, round-trip engineering is needed to make the model and code consistent, which is a critical aspect to meet quality and performance constraint required for software systems. Unfortunately, current UML tools only support structural concepts for round-trip engineering such as those available from class diagrams.

This study addresses the round-trip engineering of UML state-machine and its related generated code as sketched in Fig. 2 (b). We propose an approach consisting of a forward process which generates code by using transformation patterns, and a backward process which is based on code pattern detection to update the original state machine model from the modified code.

This round-trip engineering only works for a limited sub-set of UML State Machine features. Specifically, only hierarchical state machines without pseudo states such as fork, join, junction, histories, choice are supported. Furthermore, to achieve the reverse direction of the RTE, the code generation of this approach uses a reversible mapping, which is an extension of the state pattern [19], [20]. Although the maintainability and collaboration (because of the RTE) are gained, compared to existing approaches such as switch/if [21], this pattern adds a slight memory overhead (around 13%).

[4]MODELSWARD 2016, http://www.modelsward.org/?y=2016

[5]IEEE SEW 2016, https://www.fedcsis.org/2016/sew

VI. Conclusion

Model-Driven Engineering is considered an efficient way to dealing with the complexity of systems today. MDE relies on two mechanisms abstraction and automation. The latter are realized by artifact synchronizations. Despite many advantages of MDE, the adoption level of MDE into industries is still not high as expected. In order to raise the adoption level, the core technologies of MDE need to be enhanced. 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 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 complete generation solution for UML State Machine are chosen to be detailed in this report because of their importance. Other works are also mentioned in the report. The overview of some ongoing and future works is also described to provide readers a plan for the next steps of the thesis deployment.

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] 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
[13] SparxSystem, “Enterprise Architect,” http://www.sparxsystems.com/products/ea/, 2016.
[14] IBM, “"ibm rational rhapsody",” http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/, 2016, [Online; accessed 14-Mar-2016].
[15] OMG, “Precise Semantics Of UML Composite Structures,” no. October, 2015.
[16] ——, “Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0,” http://www.omg.org/spec/SPEM/2.0/PDF, 2008.
[17] 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.
[18] “Moka Model Execution,” https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.
[19] A. Shalyto and N. Shamgunov, “State machine design pattern,” *Proc. of the 4th International*, 2006. [Online]. Available: http://wscg.zcu.cz/rotor/net{_}2006/papers{_}2006/!proceedings{_}short{_}papers{_}2006.pdf{\#}page=55
[20] B. P. Douglass, *Real-time UML : developing efficient objects for embedded systems*, 1999.
[21] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 1998, vol. 3. [Online]. Available: http://portal.acm.org/citation.cfm?id=1088874

VII. Appendix

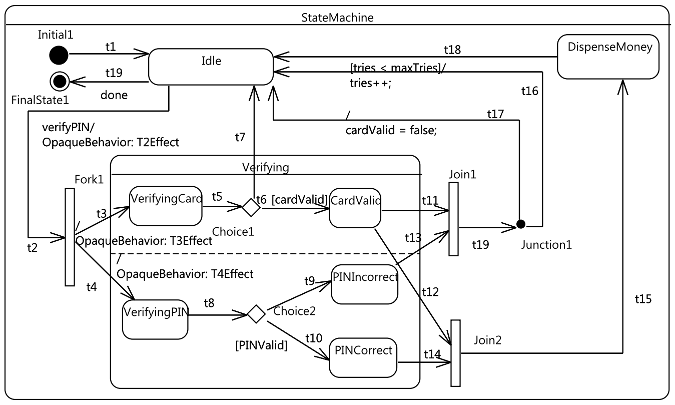

Fig. 6. ATM State machine example

Listing 2. IState interface and function pointers in C++, in which S0 is one

```
typedef struct IState {
  int previousActives[2];    int actives[2];
  EventId defEvents[2];
  void (C::*entry)();   void (C::*exit)();
  void (C::*doActivity)();
} IState;
class C {
private:
  IState states[NUM_STATES];
public:
  C() {
     states[S0_ID].entry = &C::S0_entry;
     defEvents[0] = EVENT_MAX;
     defEvents[1] = EVENT_MAX;
     ...
  }
  void S0_entry {...}
}
```

Listing 3. Example code generated for doActivity, the method in takes as input a state id to use and call the appropriate mutex and *doActivity*,

```
void doActivity(int stateId) {
  isStarts[stateId] = false;
  while(true) {
    mutex[stateId].lock();
    while(!isStarts[stateId]) {
      mutex[stateId].wait();
    }
    states[stateId].doActivity();
    isStarts[stateId] = false;
    mutex[stateId].unlock();
    if (!isStops[stateId]) {
      if (stateId == IDLE_ID || stateId ==
    DISPENSEMONEY_ID ...) {
         pushCompletionEvent(stateId);
      }
    }
  }
}
```

Listing 4. Example code generated for the region of S1

```
void S1Region1Enter(int enter_mode){
  if (enter_mode == DEFAULT) {
    states[S1_ID].actives[0] = S3_ID;
    states[S3_ID].entry();  sendStartSignal(
    S3_ID);
    S3Region1Enter(DEFAULT);
  } else if (enter_mode == S2_MODE) { //entry
    states[S1_ID].actives[0] = S2_ID;
    states[S2_ID].entry();  sendStartSignal(
    S2_ID);
  } if (enter_mode == SH_MODE) {
    StateIDEnum his;
    if (states[S1_ID].pres[0] != STATE_MAX){
      his = states[S1_ID].pres[0];
    } else {
      his = S2_ID;
    }
    states[S1_ID].actives[0] = his;
    states[his].entry();  sendStartSignal(his)
    ;
    if (S3_ID == his) {
      S3Region1Enter (S3_REGION1_DEFAULT);
    }
  } else if (enter_mode == S4_MODE) {
    states[S1_ID].actives[0] = S3_ID;
    states[S3_ID].entry();  sendStartSignal(
    S3_ID);
    S3Region1Enter(S4_MODE);
  } else if (enter_mode == ENP_MODE) {...}
```

Listing 4 shows the C++-like example code generated for entering the state $S1$ in 4. In this case the entering model values are $\{DEFAULT = 0, SH\_MODE = 1, S2\_MODE = 2, S4\_MODE = 3, ENP_MODE = 4\}$. By default, the active sub-state of the region is set after the execution of any effect associated with the initial transition, $S3$ is set as active sub-state of $S1$. It is worth noting that after the execution of each entry, a start signal is sent to activate the waiting thread associated with *doActivity* of the corresponding state.

Listing 5. Example code generated for $Join1$ and $Junction1$

```
if((states[VERIFYING_ID].actives[0]==
    CARDVALID_ID)
  &&(states[VERIFYING_ID].actives[1]==
    PININCORRECT_ID)) {
  Junction1 = 0; //else outgoing transition of
     Junction1
  if (tries < maxTries) {Junction1 = 1;}
  FORK(R1Exit); FORK(R2Exit);
   //JOIN ...
   sendStopSignal(VERIFYING_ID); exit(
  VERIFYING_ID);
  FORK(effect_t11); FORK(effect_t13);
   //JOIN ...
   if (Junction1=1) {
     tries++;
     activeStateID = IDLE_ID;     entry(
  IDLE_ID);
     sendStartSignal(IDLE_ID);
   } else {
     cardValid = false;
     activeStateID = IDLE_ID;
  sendStartSignal(IDLE_ID);
   }
}
```