From UML State Machine to code and back again!

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

***Abstract*—UML state machines and their visual representations are much more suitable to describe logical behaviors of system entities than equivalent text based description such as IF-THEN-ELSE or SWITH-CASE constructions. 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. In this paper, we address the round-trip engineering of UML state-machine and its related generated code. 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. We implemented a prototype and conducted several experiments on different aspects of the round-trip engineering to verify the proposed approach.**

I. Introduction

The so-called Model-Driven Engineering (MDE) approach relies on two paradigms, abstraction and automation [1]. It is recognized as very efficient for dealing with complexity of today system. Abstraction provides simplified and focused views of a system and requires adequate graphical modeling languages such as Unified Modeling Language (UML). Even, if the latter is not the silver bullet for all software related concerns, it provides better support than text-based solutions for some concerns such as architecture and logical behavior of application development. UML state machines (USMs) and their visual representations are much more suitable to describe logical behaviors of system entities than any equivalent text based descriptions. The gap from USMs to system implementation is reduced by the ability of automatically generating code from USMs [2], [3], [4], [3].

Ideally, a full model-centric approach is preferred by MDE community due to its advantages [5]. However, in industrial practice, there is significant reticence [6] to adopt it. 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. The code modified by programmers and the model are then inconsistent. Round-trip engineering (RTE) [7] is proposed to synchronize different software artifacts, model and code in this case [8]. RTE enables actors (software architect and programmers) to freely move between different representations [8] and stay efficient with their favorite working environment.

Unfortunately, current industrial tools such as for instance Enterprise Architect [9] and IBM Rhapsody[10] only support structural concepts for RTE such as those available from class diagrams and code. Compared to RTE of class diagrams and code, RTE of USMs and code is non-trivial. It requires a semantical analysis of the source code, code pattern detection and mapping patterns into USM elements. This is a hard task, since mainstream programming languages such as C++ and JAVA do not have a trivial mapping between USM elements and source code statements.

For software development, one may wonder whether this RTE is doable. That is, why do the industrial tools not support the propagation of source code modifications back to original state machines? Several possible reasons to this lack are (1) the gap between USMs and code, (2) not every source code modification can be reverse engineered back to the original model, and (3) the penalty of using transformation patterns facilitating the reverse engineering that may not be the most efficient (e.g. a slightly larger memory overhead).

This paper addresses the RTE of USMs and object-oriented programming languages such as C++ and JAVA. The main idea is to utilize transformation patterns from USMs to source code that aggregates code segments associated with a USM element into source code methods/classes rather than scatters these segments in different places. Therefore, the reverse direction of the RTE can easily statically analyze the generated code by using code pattern detection and maps the code segments back to USM elements. Specifically, in the forward direction, we extend the double dispatch pattern presented in [11]. Traceability information is stored during the transformations. We implemented a prototype supporting RTE of state-machine and C++ code, and conducted several experiments on different aspects of the RTE to verify the proposed approach. To the best of our knowledge, our implementation is the first tool supporting RTE of SM and code.

To sum up, our contribution is as followings:

- An approach to round-tripping USMs and object-oriented code.
- A first tooling prototype supporting RTE of USMs and C++ code.
- An evaluation of the proposed approach including:
  - An automatic evaluation of the proposed RTE approach with the prototype.

– A comparison and collaboration of two software development practices including working at the model level and at the code level.
– A lightweight evaluation of the semantic conformance of the runtime execution of generated code.

The remaining of this paper is organized as follows: Our proposed approach is detailed in Section II. The implementation of the prototype is described in Section III. Section IV reports our results of experimenting with the implementation and our approach. Section V shows related work. The conclusion and future work are presented in Section VI.

II. APPROACH

This section presents our RTE approach. At first, it sketches USM concepts supported by this study. The outline and the detail of the approach are presented afterward.

A. Scope

A USM describes the behavior of an active UML class which is called context class. A USM has a number of possible states and well-defined conditional transitions. A state is either an atomic state, a composite state that is composed of sub-states and has at most one active sub-state at a certain time, or a concurrent state which has several active sub-states at the same time. Only one of the inner states of the USM can be active at a time. Transitions between states can be triggered by external or internal events. An action 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. A transition can be external, local, or internal.

A composite state can have one or several entry/exit points. An entry/exit point can have multiple incoming transitions and exactly one outgoing transition which has no triggering event and guard. The transition outgoing from an entry point of a composite state ends on either a sub-state or an entry point of one of the sub-states of the composite state. The transition outgoing from an exit point ends on either an exit point of the parent state or a state/an entry point in the same region. A concurrent state is entered by either an incoming transition ending on its border, or several incoming transitions outgoing from a *fork* and ending on sub-states of the containing regions.

B. Approach outline

Our RTE approach is based on the double-dispatch pattern presented in [11] for mapping from USM to a set of classes with embedded code fragments, and traceability-mapping management in RTE. Fig. 1 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 and vice versa. Each class of the intermediate model contains attributes, operations and method bodies (block of text) associated with each implemented

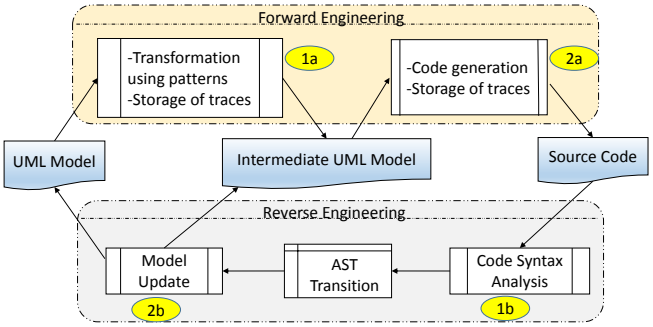

Fig. 1. Outline of state machine and code RTE

operation. The transformation utilizes several patterns which will be presented later. A tracing information table is created in the transformation to be used in the backward direction of the RTE. The intermediate model is then used as the input of a code generator to create source code. This generation step also puts a mapping from UML classes to object-oriented source code in a second table.

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 II-D2 for the detail of the analysis). This transformation takes as input the tracing tables, 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 some 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 transforms a USM into an intermediate UML model.

1) Transformation of states: This sub-section describes the transformation of states to the intermediate model. Each state of the USM is transformed into a UML class. Each UML class representing an atomic state inherits from a base state class. The latter defines a reference to the context class, a process event operation for each event in the USM and other operations as the double-dispatch (DD) approach in [11]. A state class *s* also has an attribute referring to the state class associated with the composite state containing *s*. A composite state class has an attribute pointing to a state instance indicating the active sub-state of the composite state and a *dispatchEvent* operation (see II-C5) dispatching incoming events to the appropriate active state. A composite and a concurrent state class inherit from a base composite and base concurrent state class, respectively, which also inherit from the base state class.

2) Transformation of events: Each event is transformed into a UML class. Three different event class types corresponding to the UML event types *CallEvent*, *SignalEvent* and *TimeEvent* are differentiated. 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

operation associated with the *CallEvent*. 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 enters the component through a port typed by the signal. The implementation view of this scenario depends on the mapping of component-based to object-oriented concepts. In the following, we choose the mapping done in Papyrus Designer [12]. In this mapping, the signal is transferred to the context class by an operation provided by the class at the associated port. Therefore, the transfer of a signal event becomes similar to that of *CallEvent*. 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 the event to the event queue of the component. Therefore, the processing of a *SignalEvent* is the same as that of a *CallEvent*. In the following sections, the paper only considers *CallEvent* and *TimeEvent*.

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 [13] and puts the time event in the event queue of the context class.

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. 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.

Four ways of entering a composite state are differentiated. Three of these including a transition ending (1) on the border, (2) on a sub-state or (3) on a history state of a composite state are detailed in [11]. In the last one, a transition t_{ex} ends (4) on an entry point of a composite state. Semantically, (4) is similar to (2) since both have the same sequential operations: executing the entry action of the composite state, execute the effect of the outgoing transition of the entry point t_{in} in (4) or the transition $t_{default}$ from an initial pseudo state to the sub-state in (2). The transition t_{in} is not allowed to have a guard or a trigger event similarly to the semantics of $t_{default}$.

Exiting a composite state is executed through exit points inversely to entry points.

4) Storage of tracing information: The tracing information generated in the transformation is contained in a table. Mappings from USM concepts to UML classes are mainly one-to-one except for attributes referring composite state or sub-state.

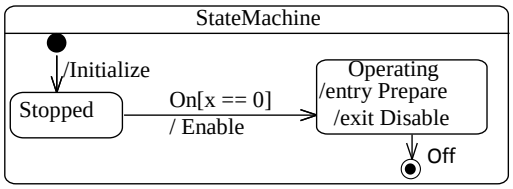

Fig. 2. An example of USM for tracing table

TABLE I
TRACING TABLE OF STATE MACHINE AND CLASS INTERMEDIATE MODEL

UML state machine concepts	UML class concepts
sPrefix::Client (Class)	Root::Client (Class)
Root::Client::StateMachine (StateMachine)	tPrefix::StateMachine (Class)
sPrefix::Stopped (State)	tPrefix::Stopped (Class)
sPrefix::Operating (State)	tPrefix::Operating (Class)
sPrefix::On (CallEvent)	tPrefix::On (Class)
sPrefix::fromStoppedToOperating (Transition)	tPrefix::StateMachine::transition (Operation)
sPrefix::Initialize(OpaqueBehavior)	tPrefix::Initialize (Operation)
sPrefix::Enable (OpaqueBehavior)	tPrefix::Enable (Operation)

The table therefore only keeps identifiers as qualified names and types of elements in the USM model and the associated elements in the UML class model. A part of tracing table for the USM example in Fig. 2 is shown in Table I in which *sPrefix* and *tPrefix* are *Root::Client::StateMachine::TopRegion* and *Root::Client::PerClass_Client*, respectively.

In Fig. 2, the USM is contained in, for instance, a *Client* component, *Root* is the name of the source model. States of the USM are contained in a region *TopRegion*. In the intermediate model, a package named *PerClass_Client* is created to contain all of transformed classes including ones associated with events and states. This package eases the maintenance of source code as well as the backward transformation of the RTE. The transition *fromStoppedToOperating* is transformed into an operation transition inside the USM class which contains *Stopped*. *Initialize*, *Enable*, *Prepare*, and *Disable* are transformed into operations in the context class *Client*. It is worth noting that there can be several transitions outgoing from a state. Therefore, more than one transition in USM can be mapped to the same qualified name in the tracing table. In order to differentiate different transitions in the intermediate model, the qualified name of a transition operation in the intermediate model is combined with the source state and the triggering event. From this tracing table, it is easy to look back the original USM elements from the elements in the intermediate model in the backward direction.

5) Code generation: The intermediate model is then used as input of a template-based object-oriented code generator. Mappings from UML classes to object-oriented are trivial one-1-one. Listing 1 shows a code segment generated from the USM in Fig. 2. The *dispatchEvent* method implemented in the base composite state class delegates the incoming event processing to its active sub-state. If the event is not accepted by the active sub-state, the composite state processes it. *OnEntryAction* and *OnExitAction* overwrite abstract methods

Algorithm 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);}
}
```

which are defined in the base state class and called by the entry and exit methods, respectively. *Stopped* accepts an *On* event by implementing a corresponding *processEvent* method. The transition method from the *Stopped* to the *Operating* state checks the guard condition by calling an associated method in the context class, then executes the transition action, changes the active state and finally enters the target state by calling the entry operation. The machine enters the final state by setting the active state to null meaning that the behavior of the region containing the final state has completed. The generated code statements are identical to the USM semantics and it is easy to modify the behavior of the USM by code.

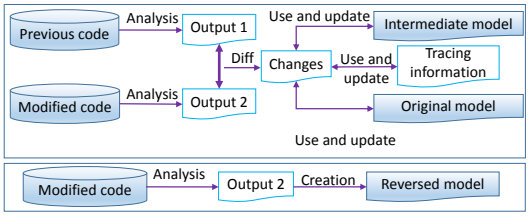

Fig. 3. Overall method for reversing code to state machine

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. 3. 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. 3), 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. 3). *Output1* and *Output2* are then compared with each other to detect actual semantic changes which are about to be propagated to the original model.

2) 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. 5 presents the EMF [14] 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. The pattern list is as followings:

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/Exit action: Similar to the entry action pattern but implements *onExitAction*/*onActivity*, respectively. According

the USM semantics, if an atomic state has outgoing *triggerless* transitions, *onEntryAction* appeals the *triggerless* transition method of the ancestor state class following the *onActivity* call. For composite states, the *triggerless* transition method is called after the its active sub-state is a final state (NULL from implementation perspective) and its activity completes. Note that thread-based parallelism of *doActivity* is implicitly implemented (not presented here) and called by the base state classes which are generated by the generator. Therefore, for each state class added to the code side, developers only need to overwrite the appropriate action methods from the base state classes. This facilitates the analysis and extraction of state actions/events to reconstruct USMs from code within the reverse engineering task.

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 class: A call event class inherits from the base event class. The call event class contains attributes typed by the parameter types of the operation associated with the call event. This pattern is detected if the types of attributes of the event class match with the types of parameters of one of the methods in the context class. There is therefore an ambiguity for an event class to choose an associated operation if more than one operation detected matches the event class. Hence, this pattern poses a restriction that operations associated with events must either have different parameter types or different number of parameters. To overcome this issues, a naming convention used for *CallEvent* classes is used. The event class name is prefixed with the associated operation name. If the event class name does not follow the naming convention, the reverse is refused. Another possible solution targeting this ambiguity is to have a user interaction in case of having more than one operation matching with the event. Having an interaction allows the pattern detection get rid of ambiguity and therefore provides appropriate USM models. A signal event is treated as a *CallEvent* as previously described.

Time event: 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. Two types of transition methods correspond to trigger and *triggerless* transitions. Both *parameterize* its source state class. The trigger transition method is associated with the event triggering the transition. 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. The transition action and the guard are optional. Several if/else statements can appear in a *triggerless* transition method body. The body of local transition methods only checks its guard and executes the transition effect.

Transition action/guard: Transition actions and guards are implemented in the context class.

Algorithm 2 Semantic Analysis

Input: AST of code and a list of state classes stateList
Output: Output of semantic analysis

```
1:  for s in stateList do
2:      for a in attribute list of s do
3:          if a and s match child parent pattern then
4:              put a and s into a state-to-ancestor map;
5:          end if
6:      end for
7:      for o in method list of s do
8:          if o is onEntryAction || o is onExitAction then
9:              analyzeEntryExit(o);
10:         else if o is processEvent then
11:             analyzeProcessEvent(o);
12:         else if o is setInitDefaultState & s is composite
    then
13:             analyzeInitDefaultState(s);
14:         else if o is timeout & s is a timedstate then
15:             analyzeTimeoutMethod(o);
16:             analyzeProcessEvent(s, o);
17:         end if
18:     end for
19: end for
```

Algorithm 2 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. Fig. 4 shows the partitioning used for matching code segments to USM elements. Each partition consists of a code segment and the corresponding model element which are mapped in the backward direction. For instance, the *Stopped* class in code is detected as a representation since it inherits from the base class *State*.

3) 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.

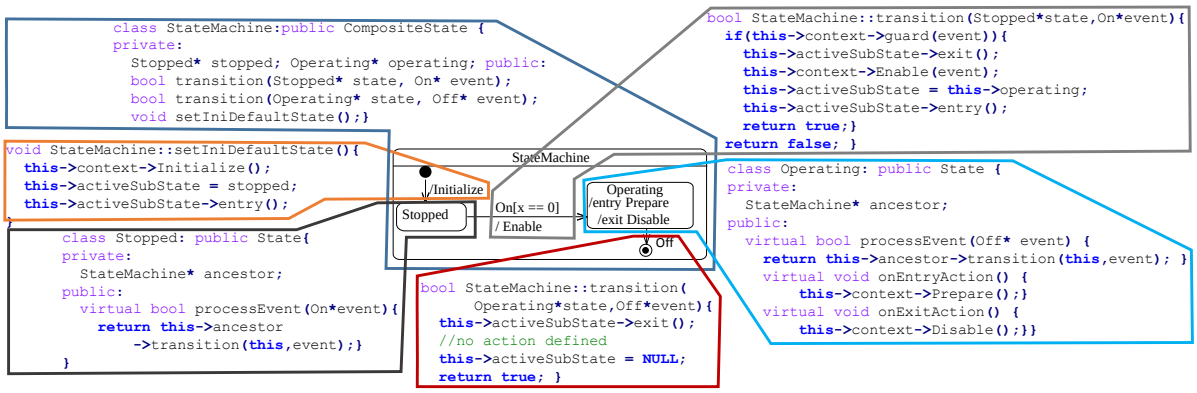

Fig. 4. USM element-code segment mapping partition

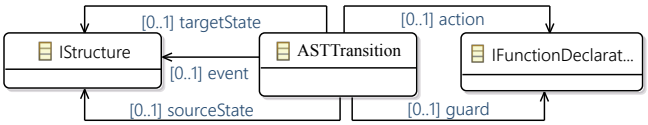

Fig. 5. Transitions output from the analysis

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.

4) Updating the original USM from modified code: In contrast to the previous sub-section, if an intermediate model is existing, lists of states and transitions are retrieved from the intermediate model. The algorithm for detecting state and transition changes is not presented here due to space limitation. Basically, the algorithm takes as input lists of state names, transitions, ancestor maps, which link a state name to its parent state name, extracted from the intermediate model and the modified code, respectively. The algorithm results in lists of state names, transitions to be added/deleted/updated/moved. It first examines the list of state names extracted from the modified code Lc to find which state to be added. A state is considered as a to-be-updated state if its name exists in both of the two lists of state names and its ancestor names in both of the ancestor maps are identical. If the latter condition is not satisfied, the state is considered as being moved to another composite state. Remaining states (not added/updated/moved) in the state list extracted from the intermediate model are added to the to-be-deleted state list. The transition and event change detection is similar to that of states but, due to the space limitation, it is not detailed. For transition change detection, instead of checking by name, the source and target state names of transitions and the associated event name in transitions are used.

The changes detected by the algorithm are then used in a change propagation step which updates the original USM. Events, states and transitions are sequentially processed in order. The processing of elements to be deleted results in deleting corresponding elements in the USM. As previously described, the mapping information for elements in code and the intermediate model is also stored in a table. Each to-be-deleted element in code is associated with an element in the intermediate model. Therefore, it is trivial to retrieve the model element in the intermediate model associated with the to-be-deleted element.

The found model element in turn helps identify the associated element in the USM by using the mapping table between the USM and the class model. For each deleted event in code, the associated event class in the class model and the event in the USM are deleted. Deleted states and transitions are similarly propagated. A deletion of a transition includes deleting its guard, triggers and transaction action.

For each event class added to code, a UML class is added to the intermediate model and a UML event to the USM. For each added state class, its ancestor state is retrieved through the mapping tables, a new state is then created as a sub-state of the ancestor state. Entry and exit actions are added to the new state afterward. A moved state is handled by looking for the associated state, the old and new ancestor state in the USM, and moving the associated state to the new ancestor state. Each added transition is propagated by creating a new transition in the USM and retrieving source and target states from the mapping tables. An update is executed by looking in the mapping tables for elements in the USM associated with elements updated in code.

For example, assuming that we need to adjust the USM example shown in Fig. 2 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.

III. Implementation

The proposed approach is implemented in a prototype existing as an extension of the Papyrus modeler [15]. Each USM is created by using the state machine diagram implemented by Papyrus to describe the behavior of a UML class. Low-level USM actions are directly embedded in form of a block of code written in specific programming languages such as **C++/JAVA** into the USM. **C++** code is generated by the prototype but other object-oriented languages can be easily generated. The code generation consists of transforming the USM original containing the state machine to UML classes and eventually to code by a code generator following the proposed approach. The code generator can generate code for hierarchical and concurrent USMs. In the reverse direction, code pattern detection is implemented as described in the previous section to analyze USM semantics. If the generated code is modified, two options are supported by the prototype to make the USM and code consistent again. One is to create a new model containing the USM from the modified code in the same Eclipse project and the other one is to update the original USM by providing as input the intermediate model and the original model. At the writing moment, the prototype does not support the reverse of concurrent USMs and pseudo states, which are *history*, *join*, *fork*,*choice*, and *junction*.

IV. Experiments

In order to evaluate the proposed approach, we answer three questions stated as followings.

RQ1: A state machine *sm* is used for generating code. The generated code is reversed by the backward transformation to produce another state machine *sm'*. Are *sm* and *sm'* identical? In other words: whether the code generated from USMs model can be used for reconstructing the original model. This question is related to the *GETPUT* law defined in [16].

RQ2: **RQ1** is related to the static aspect of generated code. **RQ2** targets to the dynamic aspect. In other words, whether the runtime execution of code generated from USMs by the generator is semantic-conformant [17]?

RQ3: RTE allows developers to freely move between and modify model and code. Specifically, in software development projects, some traditional programmers might want to practice with code in a traditional way and some MDE developers may prefer working with models. What is the development/maintenance cost comparison between the two practices by comparing the number of steps needed to do equivalent actions?

This section reports our experiments targeting the three questions. Three types of experiments are conducted and presented in Subsections IV-A, IV-B, and IV-C, respectively.

A. Reversing generated code

This experiment is targeting **RQ1**. Fig. 6 shows the experimental methodology to answer **RQ1**. The procedure for this experiment, for each original UML model containing a state machine, consists of 3 steps: (1) code is generated from an **original model**, (2) the generated code is reversed to a **reversed model**, and (3) the latter is then compared to the **original state machine**.

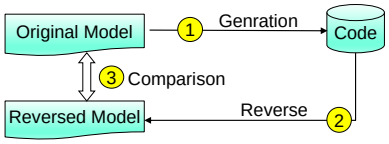

Fig. 6. Evaluation methodology to answer RQ1

TABLE II
Three of model results of generation and reverse: Abbreviations are atomic states (AS), composite states (CS), transitions (T), call events (CE), time events (TE)

Test ID	AS	CS	T	CE	TE	Is reverse correct?
1	47	33	234	145	40	Yes
2	42	38	239	145	36	Yes
..	..	..	..	..	..	Yes
300	41	39	240	142	37	Yes

Random models containing hierarchical state machines are automatically generated by a configurable model generator. The generator can be configured to generate a desired average number of states and transitions. For each model, a context class and its behavior described by a USM are generated. Each USM contains 80 states including atomic and composite states, more than 234 transitions. The number of lines of generated C++ code for each machine is around 13500. Names of the generated states are different. An initial pseudo state and a final state are generated for each composite state and containing state machine. Other elements such as call events, time events, transition/entry/exit actions and guards are generated with a desired configuration. For each generated call event, an operation is generated in the context class which is also generated. The duration is generated for each time event.

Table II shows the number of several types of elements in the generated models, including the comparison results, for 3 of the 300 models created by the generator. We limited ourselves to 300 models for practical reasons. No differences were found during model comparison. The results of this experiment show that the proposed approach and the implementation can successfully do code generation from state machines and reverse.

B. Semantic conformance of runtime execution

a) Bisimulation for semantic-conformance: To evaluate the semantic conformance of runtime execution of generated code, we use a set of examples provided by Moka [18]. Moka is a model execution engine offering Precise Semantics of UML Composite Structures [17]. Fig. 7 shows our method. We first use our code generator to generate code (Step (1)) from the Moka example set. Step (2) simulates the examples by using Moka to extract the sequence (*SimTraces*) of observed traces including executed actions. The sequence (*RTTraces*) of traces is also obtained by the runtime execution of the code generated from the same state machine in a Step (3). The generated code is semantic-conformant if the sequences of traces are the same for both of the state machine and

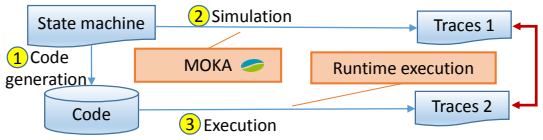

Fig. 7. Semantic conformance evaluation methodology

generated code [19]. The current version of Moka does not support simulation for *TimeEvent* and history pseudo states, we therefore leave experiments for *TimeEvent* as future work.

For example, Fig. 8 (a) shows a USM example with triggerless transitions (*autotransitions*) *T3*. The USM contains two states, *Waiting*, which is the initial state, and *Incrementing*, which increases an integer number from 0 to 5 by using the effect of *T3*. The latter also has a guard checking whether the number is less than 5. The increase is executed after the USM receives an event named *start* to transition the initial state *Waiting* to *Incrementing*. Suppose that executions of the effects of *T3* and *T4* produce traces <*T3*> and <*T4*> (by using MOKA, e.g.), respectively. Due to the guard of *T3*, the effect of *T3* is executed five times followed by an execution of the effect of *T4*. After the completion of the USM, the obtained sequence of traces is <*T3*><*T3*><*T3*><*T3*><*T3*><*T4*>. The sequence obtained by the runtime execution of the code generated from this USM must be equivalent. *RTTraces* is obtained by simply printing logging information for each action (effect).

Within our scope as previously defined 30 examples of the Moka example set are tested. *SimTraces* and *RTTraces* for each case are the same. This indicates that, within our study scope, the runtime execution of code generated by our generator can produce traces semantically equivalent to those obtained via simulation.

After experimenting with our code generator, we compare our results to the observed traces obtained by executing code generated Umple [20]. We find that the obtained traces in case of Umple are not UML-compliant in triggerless transitions and some cases of event processing. Specifically, for the example in Fig. 8 (a), code generated by Umple only produces <*T3*> as the trace sequence. Umple does not support events which are accepted by sub-states and the corresponding composite state as in Fig. 8 (b) in which both *S1* and *S21* accept the event *Continue*. As the processing event example in Fig. 8 (b), assuming that there is an event *Continue* incoming to the state machine which has a current configuration *(S1, S21)* as current active states. While, according to the UML specification, the incoming event should be processed by the inner states of the active composite/concurrent state if the inner states accept it, otherwise the parent state does. Therefore, the next configuration should be *(S1, final state)* and the *T22Effect* effect of the transition *T22* should be executed.

b) Finite state machine: In addition to the experiment using MOKA, we evaluate the semantic-conformance by using deterministic finite state machines (FSMs). The latter is a mathematical model of computation and also a simplified version of UML state machine. In this experiment, we use

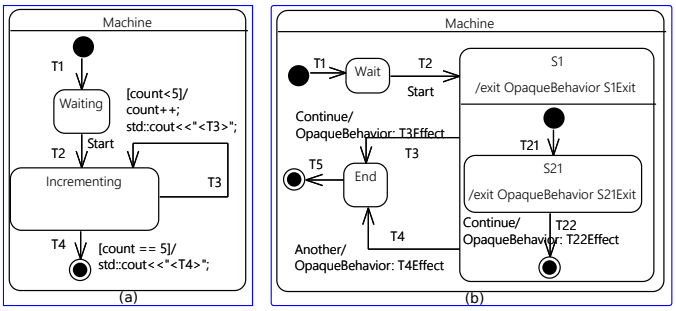

Fig. 8. Self-triggerless transition and event processing example

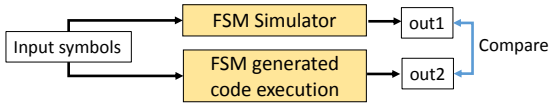

Fig. 9. FSM experiment method

FSMs for recognizing input symbols. Each FSM contains many atomic states. The active state of the FSM can be changed following the acceptance of an input symbol. Fig. 9 shows our method to experiment. For each FSM, we create an equivalent USM. Each input symbol of the FSM is considered as an event of the USM. We use the FSM simulator in [21] to generate and simulate FSMs. For each FSM, a list of observed states is recorded as output (*out1*) of the simulation for each symbol list. The latter is also the input of the generated code runtime execution of the equivalent USM which produces an output *out2*. We then compare *out1* and *out2*.

We limit the number of FSMs to 20 and the number of symbol list for each FSM to 30 for practical concerns. 600 sequences of states obtained by the simulation and a same number of sequences taken by the runtime execution are respectively compared and found being equal. This results that our code generation approach can produce semantic-conformant code in case of FSM.

C. Development/maintenance cost

To compare the development/maintenance cost, we investigate steps need to be done in generated code and models, respectively, to do semantically equivalents. For example, to add a state, on one hand, one step needed in USM diagrams is *dragging & dropping the state notation to the appropriate parent*. On the other hand, three code modifications are (1) *create a state class inheriting from the base state and its constructors*, (2) *add to the parent state class an attribute*, and (3) *add a line of code to initialize the state attribute in the parent state constructor*. Table III shows the number of steps needed for each operation. In this table, model manipulations are the winner in most of cases due to graphical representation advantages but code manipulations are still useful and comparable.

In software development, programmers might modify the generated code, the modifications might violate structures of code or USM semantics. To resolve this issue, as previously described, we provide a semantic analysis that partly and

TABLE III
COST COMPARISON

Description	Model	Code
Add a state	1	3
Add a transition	1	3
Add entry/exit action	2	2
Add transition action (effect) or guard	2	2
Update action/guard	1	1
Redirect target state of a transition	1	1
Create a call event to a transition	3	6
Create a time event to a transition	3	5
Delete a state	1	2
Delete a transition	1	3
Delete entry/exit action	1	2
Delete transition action	1	2
Delete a call event	2	many
Delete a time event	2	many

loosely inspects the AST of generated code. This inspection approach always reverses the code to the USM as well as the code is state machine-compliant even though the code is not compiled. This approach is very useful in practice in which programmers might partly modify code, automatically update the original USM by our RTE, and automatically re-generate state machine-compliant code. This re-generation does no more than completing missing elements in code meaning that all previous changes are preserved. This practice is also limitedly supported by Fujaba [22] in which activity and collaboration diagrams are partly synchronized with JAVA.

V. RELATED WORK

Our work is motivated by the desire to reduce the gap between and synchronize artifacts at different levels of abstraction, model and code in particular, in developing reactive systems. Specifically, the usage of USMs for describing the logical behavior of complex, reactive systems is indispensable. In the following sections, we compare our approach to related topics recorded in the literature.

A. Implementation and code generation techniques for USMs

Implementation and code generation techniques for USMs are closely related to the forward engineering of our RTE.

Switch/if is the most intuitive technique implementing a "flat" state machine. The latter can be implemented by either using a scalar variable [2] and a method for each event or using two variables as the current active state and the incoming event used as the discriminators of an outer switch statement to select between states and an inner one/if statement, respectively. The double dimensional state table approach [3] uses one dimension represents states and the other one all possible events. The behavior code of these techniques is put in one file or class. This practice makes code cumbersome, complex, difficult to read and less explicit when the number of states grows or the state machine is hierarchical. Furthermore, these approaches requires every transition must be triggered by at least an event. This is obviously only applied to a small sub-set of USMs.

State pattern [4], [3] is an object-oriented way to implement flat state machines. Each state is represented as a class and each event as a method. Separation of states in classes makes the code more readable and maintainable. This pattern is extended in [23] to support hierarchical-concurrent USMs. However, the maintenance of the code generated by this approach is not trivial since it requires many small changes in different places.

Many tools, such as [10], [9], apply these approaches to generate code from USMs. Readers of this paper are recommended referring to [24] for a systematic survey on different tools and approaches generating code from USMs.

Double-dispatch (DD) pattern in [11] in which represent states and events as classes. Our generation approach relies on and extends this approach. The latter profits the polymorphism of object-oriented languages. However, DD does not deal with triggerless transitions and different event types supported by UML such as *CallEvent*, *TimeEvent* and *SignalEvent*. Furthermore, DD is not a code generation approach but an approach to manually implementing state machines.

B. Round-trip engineering

Our RTE is related to synchronization of model-code and models themselves that a large number of approaches support. This paper only shows the most related approaches.

Model-code synchronization

Commercial and open-source tools such as [9], [10] only support RTE of architectural model elements and code. Systematic reviews of some of these tools are available in [25].

Some RTE techniques restrict the development artifact to avoid synchronization problems. Partial RTE and protected regions [26] aim to preserve code changes which cannot be propagated to models. This approach separates the code regions that are generated from models from regions which are allowed to be edited by developers. This form of RTE is unidirectional only and does not support iterative development [27] Our approach does not separate different regions but supports a semantic code analysis in the reverse direction.

Fujaba [22] offers an RTE environment. An interesting part of Fujaba is that it abstracts from Java source code to UML class diagrams and a so-called story-diagrams. Java code can also be generated from these diagrams. RTE of these diagrams and code works but limited to the naming conventions and implementation concepts of Fujaba which are not UML-compliant.

Model synchronization

RTE of models is tackled by many approaches categorized by its model transformation from total, injective, bi-directional to partial non-injective transformations [7]. Techniques and technologies, such as Triple Graph Grammar (TGG) [28] and QVT-Relation [29], allow synchronization between source and target elements that have non-injective mappings. These techniques require a mapping model to connect the source and target models which need to be persisted in a model store [30]. Mappings between USMs and code in our approach are non-bijective. We only use simple tables for storing tracing information.

Differently from other approaches, ours is specific to RTE of USMs and code. The goal is to provide a full model-code synchronization supporting for rapidly, iteratively, and efficiently developing reactive systems.

VI. Conclusion

This paper presented a novel approach to RTE from USMs to code and back. The forward process of the approach is based on different patterns transforming USM elements into an intermediate model containing UML classes. Object-oriented code is then generated from the intermediate model by existing code generators. In the backward direction, code is analyzed and transformed into an intermediate whose format is close to the semantics of USMs. USMs are then reconstructed or updated from the intermediate format.

The paper also showed the results of several experiments on different aspects of the proposed RTE with the tooling prototype. Specifically, the experiments on the correctness, semantic conformance of code, and the cost of system development/maintenance using the proposed RTE are conducted. Although, the reverse direction only works if manual code is written following pre-defined patterns, the semantics of USMs is explicitly present in generated code and easy to follow/modify.

While the semantic conformance of code generated is critical, the paper only showed a lightweight experiment on this aspect. A systematic evaluation is therefore in future work. We will also compare our code generation approach with commercial tools such as Rhapsody and Enterprise Architect. Furthermore, as evaluated in [7], the approach inheriting from the double-dispatch trades a reversible mapping for a slightly larger overhead. For the moment, the approach does not support RTE of concurrent state machines and several pseudo-states. Hence, future work should resolve these issues.

Acknowledgment

The work presented in this paper is supported by the European project SafeAdapt, grant agreement No. 608945, see *http://www.SafeAdapt.eu*. The project deals with adaptive system with additional safety and real time constraints. The adaptation and safety aspects are stored in different artifacts in order to achieve a separation of concerns. These artifacts need to be synchronized.

References

[1] G. Mussbacher, D. Amyot, R. Breu, J.-m. Bruel, B. H. C. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill, J. Kienzle, and M. Schöttle, "The Relevance of Model-Driven Engineering Thirty Years from Now," *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 183–200, 2014.

[2] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 1998, vol. 3.

[3] B. P. Douglass, *Real-time UML : developing efficient objects for embedded systems*, 1999.

[4] A. Shalyto and N. Shamgunov, "State machine design pattern," *Proc. of the 4th International Conference on.NET Technologies*, 2006.

[5] 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, 2012.

[6] 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.

[7] 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.

[8] S. Sendall and J. Küster, "Taming model round-trip engineering," *Proceedings of Workshop Best Practices for Model-Driven Software Development*, p. 13, 2004.

[9] SparxSystems, "Enterprise Architect," Sep. 2016. [Online]. Available: http://www.sparxsystems.eu/start/home/

[10] IBM, "Ibm Rhapsody." [Online]. Available: http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/

[11] 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.

[12] "Papyrus Designer." [Online]. Available: https://wiki.eclipse.org/Papyrus_Designer

[13] I. Niaz and J. Tanaka, "Mapping UML statecharts to java code." *IASTED Conf. on Software Engineering*, pp. 111–116, 2004.

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

[15] CEA-List, "Papyrus Homepage Website," https://eclipse.org/papyrus/.

[16] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, May 2007.

[17] OMG, "Precise Semantics Of UML Composite Structures," no. October, 2015.

[18] "Moka Model Execution." [Online]. Available: https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution

[19] J. O. Blech and S. Glesner, "Formal verification of java code generation from uml models," in *... of the 3rd International Fujaba Days*, 2005, pp. 49–56.

[20] O. Badreddin, T. C. Lethbridge, A. Forward, M. Elasaar, and H. Aljamaan, "Enhanced Code Generation from UML Composite State Machines," *Modelsward 2014*, pp. 1–11, 2014.

[21] F. Simulator, "FSM Simulator," http://ivanzuzak.info/noam/webapps/fsm_simulator/.

[22] T. Kleín, U. A. Nickel, J. Niere, and A. Zündorf, "From uml to java and back again," University of Paderborn, Paderborn, Germany, Tech. Rep. tr-ri-00-216, September 1999.

[23] I. A. Niaz, J. Tanaka, and others, "Mapping UML statecharts to java code." in *IASTED Conf. on Software Engineering*, 2004, pp. 111–116.

[24] E. Domínguez, B. Pérez, A. L. Rubio, and M. A. Zapata, "A systematic review of code generation proposals from state machine specifications," pp. 1045–1066, 2012.

[25] D. Cutting and J. Noppen, "An Extensible Benchmark and Tooling for Comparing Reverse Engineering Approaches," *International Journal on Advances in Software*, vol. 8, no. 1, pp. 115–124, 2015. [Online]. Available: https://ueaeprints.uea.ac.uk/53612/

[26] D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. New York, NY, USA: John Wiley & Sons, Inc., 2002.

[27] S. Jörges, "Construction and evolution of code generators: A model-driven and service-oriented approach," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 1–265, 2013.

[28] H. Giese and R. Wagner, "Incremental model synchronization with triple graph grammars," in *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 543–557.

[29] Q. Omg, "Meta Object Facility (MOF) 2 . 0 Query / View / Transformation Specification," *Transformation*, no. January, pp. 1–230, 2008.

[30] G. Bergmann, I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations: Change (in) the rule to rule the change," *Software and Systems Modeling*, vol. 11, no. 3, pp. 431–461, 2012.