RAOES: Round-trip Engineering for Event-Driven Embedded System Development and Evolution

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

***Abstract*—UML State Machine and its visualizations are efficient to model the logical behavior of event-driven embedded systems. Model Driven Engineering generates executable code from state machines. The generated code can then be modified by programmers. Round-trip engineering is a technique used to propagate changes made to code to the original model. However, existing round-trip engineering tools and approaches mainly focus on structural parts of the system model such as those available from UML class diagrams. This paper tackles the problem of collaboration between software architects and programmers in developing event-driven embedded systems using UML State Machine to describe the behavior. We propose RAOES-a round-trip engineering and synchronization of model and C++ code, with which the software architects and programmers can freely switch between model and code to be efficient with their preferred practice.**

I. Introduction

UML state machines (USMs) and their visual representations are efficient to describe, analyze and implement high level logic behaviors of event-driven embedded systems [1]. A number of code generation approaches and industrial tools have been proposed in the context of Model-Driven Engineering (MDE) [2] to automate the process of translating USMs into implementation [3], [4], [5], [6], [5], [7], [8], [9].

Ideally, a full model-centric approach is preferred by MDE community due to its advantages [10]. However, in industrial practice, there is significant reticence [11] to adopt it. On one hand, programmers prefer to use the more familiar programming language. On the other hand, software architects, working at higher levels of abstraction, favor the use of models, and therefore prefer graphical languages for describing the system architecture high level logic behavior [11], [12].

The back-and-forth switching between between model and code raises the consistency and synchronization problem. Round-trip engineering (RTE) [13] is proposed to synchronize different software artifacts, model and code in this case [14]. RTE enables actors (software architect and programmers) to freely move between different representations and stay efficient with their favorite working environment. In other words, RTE enables both model and code to be considered as development artifact.

Approaches proposed for RTE are categorized as *structure* and *behavior* RTE. The former refers to synchronization of structural concepts such as those available from class diagrams and code, and is supported by industrial tools such as IBM Rhapsody [7] and Enterprise Architect [15]. Some approaches such as [16], [17] allow the co-evolution of component-based diagram elements and code.

The *behavior* RTE is usually supported very limitedly. This is because there is no trivial mapping from behavior model such as USM and code. Consequently, it is very difficult to reflect behavior code changes to the original model. Approaches for the *behavior* RTE often allow programmers to partially modify behavioral code in limited areas by separating the *generated* and *non-generated code* [7], [18] using some specialized comments such as *@generated NOT*. Approaches and tools in this category use an incremental code generation, which preserves the user-code changes in the areas marked as non-generated. However, "*current separation mechanisms require the programmers are highly discipline. Furthermore, even so, accidental changes are still possible*" [19].

In this paper, we tackle the problem of synchronization between model, which includes both structural and behavioral elements, and *C++* code, which meets resource-constrained requirements for developing event-driven embedded systems. Specifically, the system architecture is specified via UML class diagrams and the behavior via USMs. Component and composite structure diagrams for architecture description will be included in future work. To support the architects and the programmers at the modeling and programming level, respectively, equivalently, our goal is to allow the synchronization of USMs with full features and code. The latter should be efficient (small in size and fast in event processing speed) to be fit into resource-constrained systems.

Our proposed technique RAOES is inspired by *ArchJava* [20] and *Archface* [21] whose goal is to allow the co-evolution of architecture and implementation in Java by introducing additional constructs to Java. Our approach adds USM-based constructs to connect C++ to the USMs. Instead of directly generating C++ code from models as the existing tools, RAOES produces a C++ front-end code, which contains our added constructs. The programmers are free to modify not only the high level logic behavior described by USMs but also the user code by making changes to the C++ front-end code.

The introduction of the front-end is similar to *MSM* [22] and *EUML* [23]. However, these front-ends use a lot of C++ templates, which make the code difficult to write and understand. Furthermore, they support only a limited subset of USM, especially events defined by UML are not correctly supported.

In RAOES, the C++ front-end is merged into and written in the usual C++ code. The front-end is then used for generating a back-end code, which is actually used for compilation to binary files. Furthermore, using our strategy defined in this paper, the front-end code is also synchronized with the model when there are concurrent modifications.

To sum up, our contribution is as followings:

- RAOES: A round-trip engineering approach for developing event-driven systems using UML State Machines and C++.
- The implementation of RAOES based on the Eclipse Modeling Framework (EMF) and the Papyrus tool.
- Experimental evaluations by experimenting with RAOES and a case study simulation.

Even though this presented work is specific to C++ and embedded systems, the general idea can be applied to other object-oriented programming languages such as Java and to other domains.

The remainder of this paper is organized as follows: Section II presents the background. Section III and IV describe the motivation and overview of RAOES. The syntax of RAOES's front-end is detailed in Section V. Section VI shows RAOES's synchronization strategy. The implementation of RAOES is described in Section VII. Section VIII presents our research questions to evaluate RAOES. A simulation case study of a Traffic Light Controller in Section IX is used to show the practicality of RAOES. Section X shows related work. The conclusion and future work are presented in Section XI.

II. Background

This section reminds the background definitions of UML State Machine (see II-A) and Model-Driven Round-Trip Engineering (see II-B) in a formal way.

A. UML State Machine

Fig. 1 shows the meta-model of UML State Machine whose detailed semantics is clearly defined in the UML specification [24] and beyond the scope of this paper. The metamodel covers full concepts defined by UML State Machine and adapts some features dedicated to our development method. Specifically, the behavior of an *ActiveClass* is described by a state machine[1]. Furthermore, instead of using the *Behavior* UML meta-element, we introduce *Action* to define actions of state and transition, which actually are user-code. *Action* is convenient in many modeling tools such as IBM Rhapsody and Enterprise Architect to allow the developers to write the low level code (user-code) directly at the model level.

A transition connects two vertexes.It can have a *guard*, an $effect$, and is triggered by a set of events. A transition is either *external*, *local*, or *internal*.

UML defines five event types. The description of the events is briefly stated as the following.

[1]For simplification, this paper assumes that an active class only holds a USM although it can have more

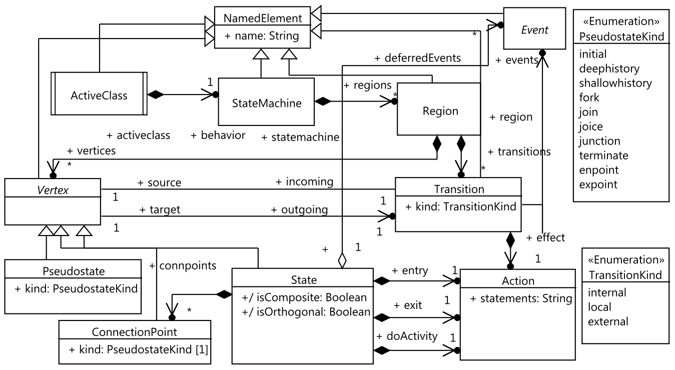


Fig. 1. State machine meta-model

- A *TimeEvent* specifies the time of occurrence *dur* relative to a starting time. The latter is specified when a state, which accepts the time event, is entered.
- A *SignalEvent* is associated with a signal *sig*, whose data are described by its attributes, and occurs if *sig* is received by a component, which is an active UML class.
- A *ChangeEvent* is associated with a boolean expression *expr* written in C++. *ChangeEvent* is emitted if *expr* changes from true (false) to false (true).
- A *CallEvent* is associated with an operation *op*. *CallEvent* is emitted if there is an invocation to *op*.
- An *Any* event is any of the above events.

B. Model-Driven Round-trip Engineering

This section defines the main capabilities, as use-cases, related to forward and reverse engineering. A developer (software architect or programmer) can either use `Generate Code (Batch)` or `Generate Code (Incremental)` related to forward engineering, and `Reverse Code (Batch)` or `Reverse Code (Incremental)`, related to reverse engineering. The definitions of these are as followings.

Definition II.1. *Batch code generation* [25] is a process of generating code from a model, from scratch. Any existing code is overwritten by the newly generated code. In contrast, *Incremental code generation* takes as input an edited model and existing code to update the code by propagating editions in the model to the code.

Definition II.2. *Batch reverse engineering* is a process of producing a model from code, from scratch. The existing model is overwritten by the newly produced model. *Incremental reverse engineering* takes as input an edited code, and an existing model to update the model by propagating editions in the code to the model.

In Section VI, the definitions are integrated into our process for model-code synchronization.

III. Motivating example: The impossibility of direct reverse engineering generated code

Let's consider a collaboration scenario between software architects and programmers in developing an event-driven paradigm-based system *System*. Fig. 2 (a) and (b) show the

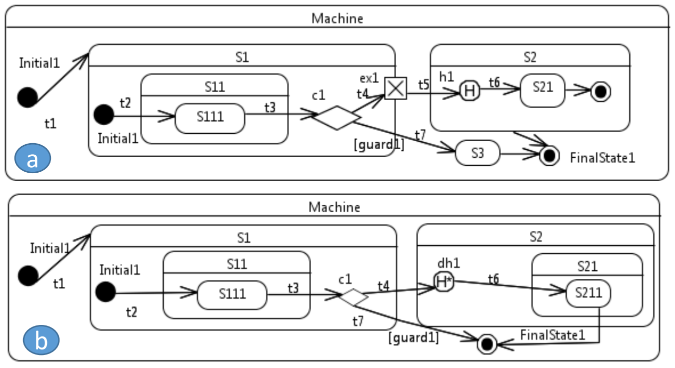


Fig. 2. A USM example (a) and its evolved version (b).

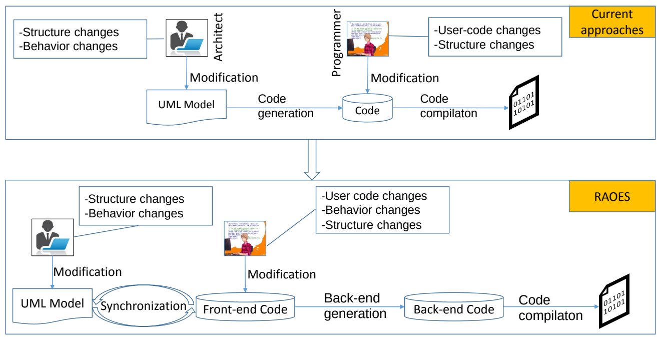


Fig. 4. From existing approaches to RAOES

```
1   systemState = EVENT_PROCESSING;
    if (states[S11_ID].actives[0]==S111_ID
3   &&(currentEvent->stateId==S111_ID)){
      Exit(states[S11_ID].actives[0]);
5     states[S11_ID].actives[0]=STATE_MAX;
      Exit(states[S1_ID].actives[0]);
7     states[S1_ID].actives[0]=STATE_MAX;
      if (guard1) {
9       Exit(S1_ID);
        activeStateID = S3_ID;
11      Entry(S3_ID);
      } else {
13      Exit(S1_ID);
        activeStateID = S2_ID;
15      Entry(S2_ID);
        unsigned int loc_ActiveId;
17      if(states[S2_ID].previousStates[0]
                  !=STATE_MAX) {
19        loc_ActiveId=states[S2_ID]
                       .previousStates[0];
21      } else {
          loc_ActiveId = S21_ID;
23      }
        states[S2_ID].actives[0]=
25                          loc_ActiveId;
        Entry(loc_ActiveId);
27    }
      systemState = EVENT_CONSUMED;
29  }
```

a

```
1   systemState = EVENT_PROCESSING;
    if (states[S11_ID].actives[0]==S111_ID
3   && (currentEvent->stateId==S111_ID)){
      Exit(states[S11_ID].actives[0]);
5     states[S11_ID].actives[0]=STATE_MAX;
      Exit(states[S1_ID].actives[0]);
7     states[S1_ID].actives[0]=STATE_MAX;
      if (guard1) {
9       Exit(S1_ID);
        activeStateID = STATE_MAX;
11    } else {
        Exit(S1_ID);
13      activeStateID = S2_ID;
        Entry(S2_ID);
15      unsigned int S2_Region1_dh1;
        if (states[S2_ID].previousStates[0]
17                 != STATE_MAX) {
          S2_Region1_dh1 =
19          states[S2_ID].previousStates[0];
          Entry(S2_Region1_dh1);
21        if (S21_ID==S2_Region1_dh1) {
          unsigned int S21_Region1_dh1 =
23        states[S21_ID].previousStates[0];
            Entry(S21_Region1_dh1);
25        }
        } else {
27        states[S2_ID].actives[0]=S21_ID;
          Entry(S21_ID);
29        states[S21_ID].actives[0]=S211_ID;
          Entry(S211_ID);
31      }
      }
33    systemState = EVENT_CONSUMED;
    }
```

b

Fig. 3. Codes generated from the state machine example in Fig. 2 by using our tool (a) and Rhapsody (b), and their respective evolved versions. The dashed underlined code segment should evolve to the simple underlined code.

current and evolved USM behaviors of *System*. This USM consists of some simple, composite, and pseudo states such as *choice*, *connection point expoint*, and *junction*.

A few tools such as IBM Rhapsody [7] and ours are able to deal with this example because generating code for pseudo states such as *expoint* and *history* is not as simple as states. Fig. 3 (a) and (b) show the code segments generated for the transition outgoing from the state *S111* of the example and its evolved version by using our tool, respectively.

For simplification, we assume that no effects are associated with the transitions in the examples. In Fig. 3 (a), the code segment checks whether the state *S111* is active (lines 2-3). If so, the exit actions of *S111* and *S11* are executed sequentially (lines 4 and 6). The sub-states of *S11* and *S1* also become inactive by setting the appropriate values to *STATE_MAX* (lines 5 and 7). The segment then evaluates *guard1* (lines 8 and 12) to dynamically select which transition outgoing from the choice *c1* should be taken into account. The exit action of *S1* (line 13), the entry action (line 15) and the restoration of the previous active sub-state of *S2* (lines 16-26) are called if *guard1* is false. Otherwise, *S1* and *S3* are exited and entered (lines 9-11), respectively. The code in Fig. 3 (b) differs from that of Fig. 3 (a) by the way the history of *S2* is restored. Fig. 3 (b) executes a deep restoration in lines 14-32 if *guard1* is evaluated as false.

The code generation patterns are not explicitly understandable for the programmers to capture the control flow of the USM. Hence, it is challenging to modify the topology of the USM at the code level. Even, if the programmers could understand and modify the code , it is still very difficult for RTE tools to decipher and reflect the code changes to the model. Furthermore, it is very hard, if not impossible, to find common rules to reconstruct the original state machine from the code. This is the reason why existing RTE tools such as Rhapsody have no way to recover the modified code to the original USM.

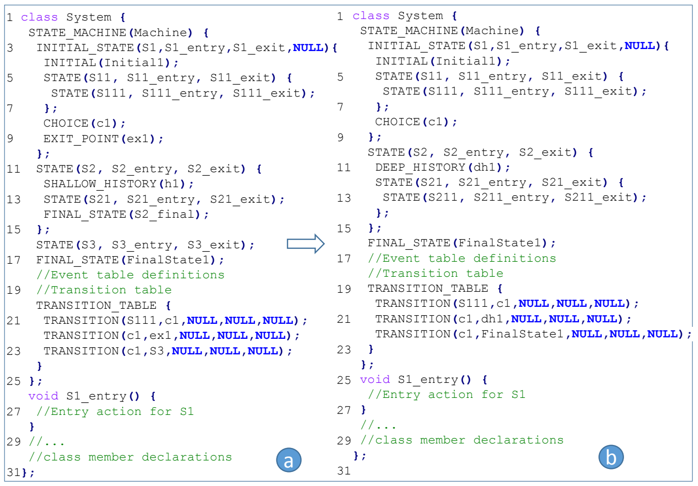
Consequently, to interfere the high-level logic behavior of the systems, the programmers must use the click-and-select mechanism of modeling tools, which are, as previously, not encouraged for the programmers to be efficient. Furthermore, it does not guarantee the seamless collaboration between the favored practices of the programmers and software architects.

In the next section, we show how RAOES can handle this collaboration problem.

IV. RAOES Overview

The goal of RAOES is to seamlessly support the collaboration of software architects and programmers in developing event-driven systems. In the latter, the behavior of active objects is specified by using UML State Machines. As previously described, it is very difficult to reconstruct the original model from the generated code since there is no bijective mapping between these artifacts. Therefore, RAOES defines a mechanism interface textually embedded inside the active objects, which are defined by object-oriented classes. This mechanism acts as a role to communicate the C++ programming language to USM so that the traceability between model and code in the reverse direction of the RTE can be eased.

Specifically, in the code generation process, instead of directly generating C++ code as in Fig. 3, RAOES produces a front-end C++ code. The latter plays as an intermediate



```
1 class System {
   STATE_MACHINE(Machine) {
3   INITIAL_STATE(S1,S1_entry,S1_exit,NULL){
     INITIAL(Initial1);
5    STATE(S11, S11_entry, S11_exit) {
      STATE(S111, S111_entry, S111_exit);
7    };
     CHOICE(c1);
9    EXIT_POINT(ex1);
    };
11  STATE(S2, S2_entry, S2_exit) {
     SHALLOW_HISTORY(h1);
13   STATE(S21, S21_entry, S21_exit);
     FINAL_STATE(S2_final);
15  };
    STATE(S3, S3_entry, S3_exit);
17  FINAL_STATE(FinalState1);
    //Event table definitions
19  //Transition table
    TRANSITION_TABLE {
21   TRANSITION(S111,c1,NULL,NULL,NULL);
     TRANSITION(c1,ex1,NULL,NULL,NULL);
23   TRANSITION(c1,S3,NULL,NULL,NULL);
    }
25 };
   void S1_entry() {
27  //Entry action for S1
   }
29 //...
   //class member declarations
31};
```

```
1 class System {
   STATE_MACHINE(Machine) {
    INITIAL_STATE(S1,S1_entry,S1_exit,NULL){
     INITIAL(Initial1);
5    STATE(S11, S11_entry, S11_exit) {
      STATE(S111, S111_entry, S111_exit);
7    };
     CHOICE(c1);
9   };
    STATE(S2, S2_entry, S2_exit) {
11   DEEP_HISTORY(dh1);
     STATE(S21, S21_entry, S21_exit) {
13    STATE(S211, S211_entry, S211_exit);
     };
15  };
    FINAL_STATE(FinalState1);
17  //Event table definitions
    //Transition table
19  TRANSITION_TABLE {
     TRANSITION(S111,c1,NULL,NULL,NULL);
21   TRANSITION(c1,dh1,NULL,NULL,NULL);
     TRANSITION(c1,FinalState1,NULL,NULL,NULL);
23  }
   };
25 void S1_entry() {
    //Entry action for S1
27 }
   //...
29 //class member declarations
  };
31
```

Fig. 5. Front-code evolution

representation, which is C++-conformant. Fig. 4 shows how RAOES is different from the existing approaches. RAOES separates codes for the structural and behavioral part. The connections between these parts are realized by using simple name binding mechanisms.

For example, by using RAOES, the generated front-end code for the example in Fig. 2 (a) and (b) are presented in Fig. 5. The USM defining the behavior of the active class *System* is defined inside the class. The USM is written in a description-like language. The topology of the USM is explicitly and hierarchically described. All USM features can be represented in RAOES's front-end. Hence, we allow to fully generate code from USMs.

In RAOES, the programmers can modify not only structural and user-code parts, which are offered by advanced round-trip engineering tools such as Rhapsody and Enterprise Architect, but also the high-level logic behavior specified by USM. The modification is realized by making changes to the front-end code. For example, if the programmers want to modify the high level behavior of the USM, they can modify the hierarchical description of the USM in the class *System*; The user-code (*Action*) can be simply modified as usual C++ method code; Other parts such as user-created attributes or methods can be freely modified.

The front-end closely connects to the USM concepts to make programmers easy to modify the state machine. The front-end merges the USM description into the active class *System* and keeps the class members intact. Therefore, the programmers are free to work with C++ as their practice. This is especially our difference and advantage over some text-based state machine languages such as Umple[2] and ThingML[3]. The latter languages adapt existing languages and the programmers' habit into USMs by providing a new language with a new editor (usually defined in Eclipse Xtext). However, the new editor usually does not support the IDE utilities such as syntax highlights and intelligent completion for mainstream programming languages.

[2]Umple, http://cruise.eecs.uottawa.ca/umple/

[3]ThingML, http://thingml.org/

V. RAOES's LANGUAGE SYNTAX

This section presents how USMs can be represented in the front-end language of RAOES. A USM in RAOES is defined inside an active class (C++ class). It consists of three parts: topology, event definition, and transition table. The followings describe the details of these parts.

A. Topology

A topology describes how the hierarchy of a USM is organized. As the example in Fig. 5, the root of the topology is define by *STATE_MACHINE*. Other elements such as *region*, *state*, and *pseudo state* are defined as sub-elements. The followings give the syntax of some elements and the semantics of RAOES mapped to the well-defined semantics in the UML specification [24].

1) State and regions:

Syntax:

- *state machine* → 'STATE_MACHINE('name') {vertices};'
- *state* → 'STATE ('name, ent, ex');'
- *state* → 'INITIAL_STATE ('name, ent, ex, effect');'
- *concurrent state* → 'STATE ('name, ent, ex') {' regions '};'
- *regions* → region; regions
- *region* → 'REGION (' name ') ' vertices ';'
- *deferred event* → 'DEFER(' eventId ');'
- *event queue size* → 'CONFIGURATION('

Semantics:

- *name*: the unique identifier of a state machine, a state or a region.
- *ent/ex*: The name of the entry/exit action method associated with the state. These methods are implemented in the active class and have no parameter. If a state does not have an entry/exit action, *ent/ex* becomes *NULL*.
- *INITIAL_STATE*: A state is defined as an initial state, which has an incoming transition outgoing from a pseudo initial state within the same region or composite state.
- *effect*: For initial state, this is the transition effect associated with the initial transition. If the latter does not have an effect, *effect* is specified as *NULL*.
- *concurrent state*: The representation of a concurrent state. The latter is composed of a set of regions. Each region contains a set of vertices, which for each is either a state or a pseudo state.
- *eventId*: The identifier of a defined event (see V-B), which is deferred by the corresponding state.

Example: Lines 2-10 in Fig. 5 (a) represents the state *S1* of the USM example in Fig. 2 (a) and its vertices. *S1* is defined as the initial state of the USM, and has *entry* and *exit* actions bound to the methods *S1_entry* (lines 26-28) and *S1_exit* (not shown) implemented in the active class. The direct *S11* and indirect *S111* sub-state are defined hierarchically (lines 5-7) with the associated actions.

2) Pseudo state:

Syntax:

- *enpoint* → 'ENTRY_POINT ('name');'
- *expoint* → 'EXIT_POINT ('name');'
- *initial* → 'INITIAL ('name');'
- *final* → 'FINAL_STATE ('name');'
- *join* → 'JOIN ('name');'
- *fork* → 'FORK ('name');'
- *choice* → 'CHOICE ('name');'
- *junction* → 'JUNCTION ('name');'
- *shallow history* → 'SHALLOW_HISTORY ('name');'
- *deep history* → 'DEEP_HISTORY ('name');'

Example:

EXIT_POINT(ex1); and *SHALLOW_HISTORY(h1);* in Fig. 5 represent a connection exit point and a shallow history with their respective name.

B. Events

Events represent all USM events a USM can react. As defined in Section II, there are four USM event types: *CallEvent*, *TimeEvent*, *SignalEvent*, *ChangeEvent*.

Syntax:

- *CallEvent* → 'CALL_EVENT' '('name, op');'
- *TimeEvent* → 'TIME_EVENT' '('name, dur');'
- *SignalEvent* → 'SIGNAL_EVENT' '('name, sig');'
- *ChangeEvent* → 'CHANGE_EVENT' '('name, expr');'
- *SimpleEvent* → 'SIMPLE_EVENT' '('name');'

Semantics: Essentially, each field in the syntax carries known semantics defined in the UML specification and Section II:

name	The unique identifier for an event.
op	The name of the operation associated with a *CallEvent* and implemented in the active class.
dur	The duration associated with a *TimeEvent* and specified as millisecond.
sig	The name of the signal associated with a *SignalEvent*.
expr	The expression associated with a *ChangeEvent*. This expression is periodically evaluated to check whether its boolean value is changed.

SimpleEvent is a specialized *SignalEvent* without specifying an explicit signal. It is not explicitly standardized by UML but provided by tools such as QM [9] for practical reasons.

Example:

- *CALL_EVENT(CE1, method1)*: A *CallEvent* occurs if the method *method1* in the active class is called.
- *SIGNAL_EVENT(SE, Sig)*: A *SignalEvent* occurs if an instance of *Sig* is sent to the active class using its provided method *sendSig*.
- *TIME_EVENT(TE5ms, 5)*: A *TimeEvent* occurs after 5 millisecond from the moment the timer starts by entering some state.

C. Transitions

As previously defined, there are three kinds of transitions: *external*, *local*, and *internal*.

Syntax:

- *external* → 'TRANSITION' '('src, tgt, guard, evt, eff');'
- *local*→'LOCAL_TRANSITION('src,tgt,guard,evt,eff');'
- *internal* → 'INT_TRANSITION('src, guard, evt, eff');'

Semantics:

src	The name of the source vertex of the transition. This name must be defined in the topology.
tgt	The name of the target vertex of the transition.
guard	A boolean expression representing the transition's guard. If the transition is not guarded, *guard* is NULL.
evt	The name of the event triggering the transition. evt must be one of the defined events. If the transition is not associated with any event, *evt* is NULL.
eff	The name of the method, which defines the effect of the transition. The method is implemented in the active class. If *evt* is a *SignalEvent*, the method has an input parameter typed as the signal associated with the event. If *evt* is a *CallEvent*, the method has the same parameters as the operation associated with *evt*. If the transition has no effect, *eff* becomes NULL.

Example:

- *TRANSITION(S1,S2,guard1,CE1,effect1)*: A transition from *S1* to *S2* which is fired if there is an appeal to the method *method1* and the value of *guard1* is true. *method1* is associated with the *CallEvent CE1* as the example above. Furthermore, an action *effect1* is executed during the transition fire. Listing 1 shows how to write *effect1* (lines 4-6) corresponding to the signature of *method1* (line 1-3) following the above semantics.
- *TRANSITION(S11,S2,NULL,SE,effect2)*: A transition, triggered by the signal event *SE*, and executes an effect *effect2*. *effect2* in Listing 1 has a parameter typed by the signal *Sig* associated with the event *SE*.

Listing 1. A segment of C++ front-end code

```
1 void method1(int p1, int p2) {
    //method1 body
3 }
  void effect1(int p1, int p2) {
5   //effect1 body
  }
7 void effect2(Sig& s) {
    //effect2 body
9 }
```

VI. Process for synchronization

This section describes RAOES's process to synchronize a model with USMs and front-end code in case that these artifacts concurrently evolve. We assume that a used integrated development environment (IDE) offers the use-cases defined in Section II-B. The process allows concurrent modifications made to the model and front-end code so that both can be used for the full implementation of a system.

We propose two synchronization strategies. The rationale behind our strategies is to represent one artifact (model/code) in the language of its corresponding other artifact (code/model). For this, we define a concept of *synchronization artifact*:

Definition VI.1 (Synchronization artifact)**.** An artifact used to synchronize a model and its corresponding front-end code is called a synchronization artifact. It is an image of one of the artifacts, either the model or the front-end 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 (SA) 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 SA, two strategies are proposed: one in which the SA is code, and the other in which the SA 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, since he prefers to work exclusively with code).

Figure 6 shows the first synchronization strategy based on using front-end code as the SA. The general steps of the process shown in Figure 6 are described as follows:

Step 1	Both the model and code may be edited concurrently.
Step 2	First we create a SA from the edited model by generating front-end code in batch mode. This SA is code and it is an image of the edited model.
Step 3	The SA is synchronized with the edited code. Since the SA is code itself, this step is done with the `Synchronize Code` use-case of the IDE.
Step 4	Once SA and edited code are synchronized, the former is reversed incrementally to update the edited model.

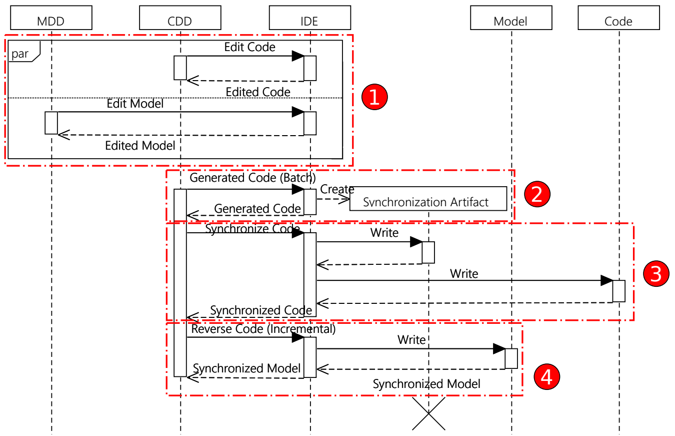


Fig. 6. Synchronization process, in which the model and the code are concurrently edited with code as the SA (CDD = Code-Driven Developer = Programmer, MDD = Model-Driven Developer = Software Architect, Code = C++ front-end code). The API calls for Model and Code are represented generically as "Read" and "Write".

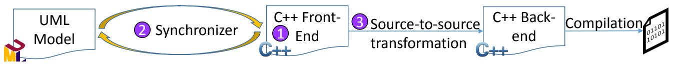


Fig. 7. RAOES's architecture

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

The actors 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.

VII. RAOES Implementation Method

This section presents the architecture and implementation detail of a RAOES prototype based on the Eclipse Modeling Framework (EMF). The latter provides many facilities such to ease the development. Fig. 7 shows the RAOES's architecture. The latter consists of the C++ front-end extending C++, a synchronizer between the model and the front-end, and a source-to-source transformation. The implementation of these modules is presented in the followings. Although some Eclipse facilities are used, the implementation method is generic and can be applied to other development environments.

A. C++ front-end implementation

The purpose of introducing the front-end is to ease and reduce the programmers' effort in modifying the topology of USMs, including all features. Therefore, the front-end should be easily parsed by inspecting the *Abstract Syntax Tree (AST)* of C++. The front-end is presented in a hierarchical way. Hence, we use the class hierarchy in C++ to represent the underlying. This is of course not the only one way to define the front-end. To easily recognize the state machine element types in RAOES, we use specialized names embedded in the macro definitions of RAOES.

In this implementation, hierarchical elements such as *State Machine*, *State*, *Region* in concurrent states, and events are implemented as classes. Pseudo states, which are contained by either a region or state (e.g. connection points), are implemented as class attributes. Each transition is associated with a statement in a method representing the transition table. By using this implementation, the front-end is compilable with C++ compilers such as GCC.

B. Synchronizer

The synchronizer consists of three sub-modules: a front-end code generator from the model, a reverse engineering from front-end to UML, and a synchronization. The implementation of the latter is as followings:

1) The front-end generator: The front-end code consists of two parts: state machine and class members. The former is generated by Step 1-4 and the latter by Step 5 in the following steps.

Step 1 The UML State Machines in UML models are verified whether they are valid or not. If valid, the regions and vertexes of each state machine are inspected to generate the state machine topology in the RAEOS's language.

Step 2 All possible events reactivated a USM are collected and inspected. For each event, the appropriate event representation in RAOES is represented.

Step 3 For each transition, a row in the transition table is generated.

Step 4 Each state action or transition effect is transformed into a method, which is used for binding in the USM's topology written in RAOES. The method body is embedded directly in the model level through the specialized element *OpaqueBehavior*. The latter is in fact supported by most of the existing state machine code generation tools.

Step 5 For each active class, the structural and usual operation parts are generated by using the Papyrus C++ code generator [26].

2) Reverse engineering: The reverse engineering consists of inspecting and analyzing the front-end code, and convert and abstract to the model. It is composed of two steps: reversing the state machine part and the class member part.

Step 1 Parsing the state machine part in the front-end code by using the specialized names as above to recognize state machine element types. The reconstruction of the state machine from the recognized elements is then straightforward. If there are actions including *entry/exit/doActivity* of state and transition effect, the corresponding methods implemented in the active class are parsed and reversed.

Step 2 For each class in written in RAOES, all class members except the members belonging to the state machine part are reversed engineered.

3) Synchronization: As presented in Section VI, the synchronization of the model and front-end code requires not only a batch generator and reverse engineering as described in VII-B1 and VII-B2, respectively, but also their respective incremental versions. The latter are presented in the followings.

a) Incremental front-end code generator: The incremental generator only regenerates the code parts associated with the changed model elements. We implement a model listener which is based on the EMF transaction mechanism (other mechanisms of modeling tools can be used). The listener is hooked to the Papyrus modeling tool to detect model changes. Fig. 8 shows the model change classification.

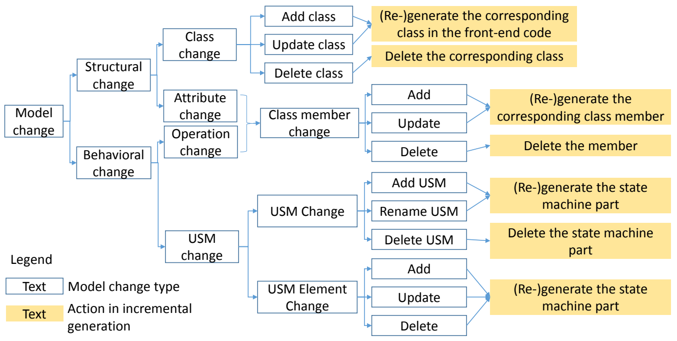


Fig. 8. Model change management in incremental generation

Each change to the model is either a structural or behavioral change. The former is an update/deletion/addition of class or attribute while the latter of operation or USM concept such as vertex, transition or event. Model changes trigger different updates to the front-end code. For example, in Fig. 8, when an attribute or operation is changed (update/delete/add), the associated element in code is also changed, respectively. If a USM concept is changed, the USM written in RAOES's language is regenerated. By using the incremental generator, the code elements associated with unchanged model elements are kept intact.

b) Incremental reverse engineering from front-end code to model: Similarly to the incremental code generator, there needs to be a code listener to the changes made to the code. In Eclipse, we implemented the listener on top of C/C++ Development Tool (CDT). The code changes are also classified as in the model change in Fig. 8. The change management actions propagate the code changes to the model similarly to the other way. Hence, we do not go to details of this implementation.

c) Conflict resolution strategy:

C. Transformation

The transformation takes as input the C++ front-end code to generates the C++ back-end code which is used for compilation and execution. We implemented this transformation based on the reverse engineering as previously presented and a state machine code generation.

Fig. 9 describes the transformation is realized in two steps. Step 1 reverse engineers the front-code to UML models with USMs and Step 2 generates the back-end code via USM code generators. Although there are many approaches and tools supporting code generation for USMs, a complete approach is still missing [27], especially when considering concurrency and the support of event types. Therefore, in order to provide full synchronization of USMs and code, we design an approach to generating code from USMs with full features to not restrict developers in modeling and coding.

Our code generation approach combines the state pattern in [28] and IF/ELSE constructions, and extends the support of these patterns for all of pseudo states and events defined in USMs. The detail of the generation is not presented here due to space limitation. For example, the generated back-end code for the USM examples in Fig. 2 is shown in Fig. 5.

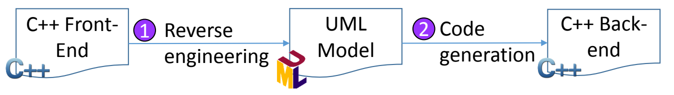


Fig. 9. Source-to-source transformation via reverse engineering and code generation

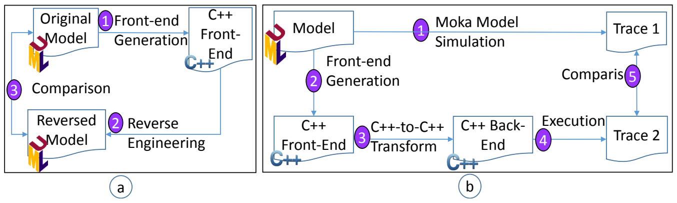


Fig. 10. Evaluation methodology to answer RQ1 (a) and RQ2 (b)

VIII. Experiments report

In order to evaluate RAOES, we conducted experiments focusing on different aspects. Our research questions are as followings:

RQ1 Whether the front-end code generated from a model with USMs can be used for reconstructing the original model. This question is related to the *GETPUT* law defined in [29].

RQ2 The back-end code is used for compilation. Does the runtime execution of the back-end code is semantic-conformant to Precise Semantics for UML State Machines (PSSM)?

RQ3 Runtime performance and memory usage is undoubtedly critical in real-time and embedded systems. Particularly, in event-driven systems, the performance is measured by event processing speed. Does code generated by the presented approach outperform existing approaches and use less memory?

In the followings, Subsections VIII-A, VIII-B, and VIII-C report the experimental results for RQ1, RQ2, and RQ3, respectively.

A. Reversing generated code

Fig. 10 (a) 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) **C++ front-end code** is generated from an **original model**; (2) the **C++ front-end code** is reverse engineered to a **reversed model**; and (3) the **reversed model** is then compared to the **original model**.

We developed a configurable generator to derive random models. The configuration information consists of an active class and its state machine behavior with desired average numbers of vertexes, transitions, and events. Each USM contains 100 vertices , more than 234 transitions, more than 50 events.

300 radom models are produced 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 RAOES can successfully do C++ front-end code generation from state machines and reverse.

B. Semantic conformance of runtime execution

To evaluate the semantic conformance of runtime execution of the back-end code for **RQ2**, we use a set of USM examples provided by *Moka* [30]. The latter is a model execution engine offering PSSM. Fig. 10 (b) shows our method, which consists of the following steps:

Step 1 For a *model* from the Moka example set, we simulate its execution by using Moka to extract a sequence of traces *Trace 1*.
Step 2 A *C++ front-end* code is generated from the *model* using the front-end generator implemented in Section VII-B1.
Step 3 The *C++ front-end* is used as input for generating a *C++ back-end* code using the source-to-source transformation.
Step 4 The *C++ back-end* is compiled for execution to obtain a sequence of traces *Trace 2*.
Step 5 *Trace 1* and *Trace 2* are compared.

The *C++ back-end* is semantics-conformant if *Trace 1* and *Trace 2* are the same.

PSSM test suite consists of 66 test cases totally for dedicated to different elements. The results are promising that RAOES passes 62/66 tests including: behavior (5/6), choice (3/3), deferred events (6/6), entering (5/5), exiting (4/5), entry(5/5), exit (3/3), event (9/9), final state (1/1), fork (2/2), join (2/2), transition (11/14), terminate (3/3), others (2/2). In fact, RAOES fails with some wired tests such as transitions from an *enpoint* to an *expoint*. This is, as our observation, never used in practice. Furthermore, as the UML specification says that transitions outgoing from an *enpoint* of a composite state should end on one of the sub-vertexes.

However, this evaluation methodology has a limitations that it is dependent on PSSM. Currently, PSSM is not fully defined. Specifically, only *SignalEvent* is supported. On pseudo-states, histories are not supported. Thus, our evaluation result is limited to the current specification of PSSM.

C. Benchmarks

In this section, we present the results obtained through the experiments on some efficiency aspects of back-end code to answer **RQ3**.

Two state machine examples are obtained by the preferred benchmark used by the Boost C++ libraries [31] in [32]. One simple example [33] only consists of atomic states and the other [34] both atomic and composite states.

Code generation tools such as Sinelabore (which efficiently generates code for Magic Draw [35], Enterprise Architect [36]) and QM [9] , and C++ libraries (Boost Statechart [37], Meta State Machine (MSM) [22], C++ 14 MSM-Lite [32], and functional programming like-EUML[23]) are used for evaluation.

Experimental procedures: We use a Ubuntu virtual machine 64 bit (RAM, memory, Ghz??) hosted by a Windows 7 machine. For each tool and library, we created two applications corresponding to the two examples, generated C++ code and compiled it two modes: normal (N), by default GCC compiler, and optimal (O) with options -O2 -s. 11 millions of events are generated and processed by the simple example and more than 4 millions for the composite example. Processing time is measured for each case.

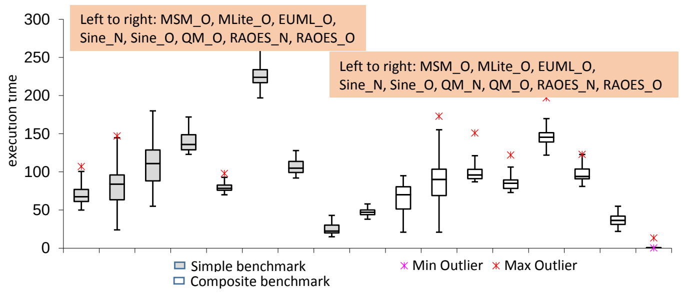


Fig. 11. Event processing speed for the benchmark

1) Speed: Fig. 11 shows the event processing performance of the approaches. In the normal compilation mode (N), Boost Statechart, MSM, MSMLite, EUML are quite slow and not displayed in the box-plot. Only Sinelabore and QM are performantly comparable with our approach. The table also shows that the optimization of GCC is significant. MSM and MSMLite run faster than Sinelabore and QM.

Our approach processes faster around 40 milliseconds than the fastest approach within the scope of the experiment. It is seen that, even without GCC optimizations, code generated by our approach significantly runs faster than that of EUML and QM with the optimizations. When compiled with the optimizations, our approach improves the event processing speed. Even, in case of composite, our approach does not produce any slowness compared to the simple example.

2) Binary size and runtime memory consumption: Table I shows the executable size for the examples compiled in two modes. It is seen that, in GCC normal mode, Sinelabore generates the smallest executable size while our approach takes the second place. When using the GCC optimization options, QM and our approach require less static memory than others.

Considering runtime memory consumption, we use the Valgrind Massif profiler[38] to measure memory usage. Table II shows the measurements for the composite example. Compared to others, code generated by our approach requires a slight overhead runtime memory usage (1KB). This is predictable since the major part of the overhead is used for C++ multi-threading using POSIX Threads and resource control using POSIX Mutex and Condition. However, the overhead is small and acceptable (1KB).

IX. Traffic Light Controller simulation

In order to assess the usability and practicality of UML State Machines and RAOES, we consider a simplified Traffic Light Controller (TLC) system as a case study, which is extracted from [39]. TLC controls an intersection of a busy highway and a little-used farm-way.

The system is shown in Fig. 12.

To apply RAOES, a software architect and a programmer participated to the development. The class system design is similar to the object-oriented one presented in [40]. Each class's behavior is described by USMs. However, the state machine describing the behavior of *Intersection* in our design

TABLE I
EXECUTABLE SIZE IN KB

Test	SC		MSM		MSM-Lite		EUML		Sinelabore		QM		RAOES	
	N	O	N	O	N	O	N	O	N	O	N	O	N	O
Simple	320	63,9	414,6	22,9	107,3	10,6	2339	67,9	16,5	10,6	22,6	10,5	21,5	10,6
Composite	435,8	84,4	837,4	31,1	159,2	10,9	4304,8	92,5	16,6	10,6	23,4	21,5	21,6	10,6

TABLE II
RUNTIME MEMORY CONSUMPTION IN KB. COLUMNS (1) TO (7) ARE SC, MSM, MSM-LITE, EUML, SINELABORE, QM, AND OUR APPROACH, RESPECTIVELY.

Test	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Composite	76.03	75.5	75.8	75.5	75.8	75.7	76.5

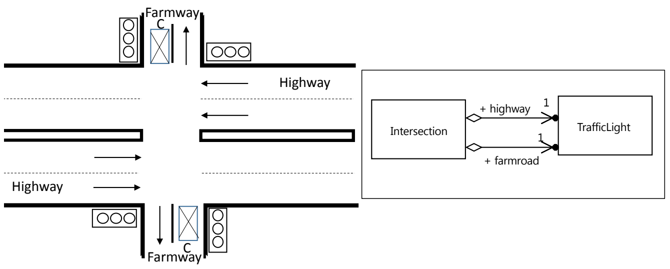


Fig. 12. Traffic Light Controller (left) and its class diagram (right).

is invented by two ways utilizing *ChangeEvent*s and the deference of events in USM, respectively.

The design of behaviors of *Intersection* and *TrafficLight* is shown in Fig. 13 (left and right, respectively). It is worth noting that the states of *IntersectionStateMachine*, except *FarmwayOpen* are composite. The details of *SwitchingHighwayToFarmroad* and *SwitchingFarmroadToHighway* are actually shown on the yasmine site [41].

The requirements for switching from the state *HighwayOpen* to *SwitchingHighwayToFarmroad* are: (1) a minimum time for the highway open is elapsed; and (2) the sensors signal. Fig. 14 (b) and (d) shows the alternative designs for the composite state *HighwayOpen*, and Fig. 14 (a) and (c) displays their respective version written in C++ front-end. The design in ((a) and (b)) uses a *TimeEvent* and the event deference. Specifically, when *HighwayOpen* becomes active, its active sub-state remains *WaitingForHighwayMinimum* as long as the minimum time. If the a signal event is emitted from the detector, the event is delayed until the active sub-state becomes *MinimumTimeElapsed*. The event is then processed to finish the execution of *HighwayOpen* and active the farmway.

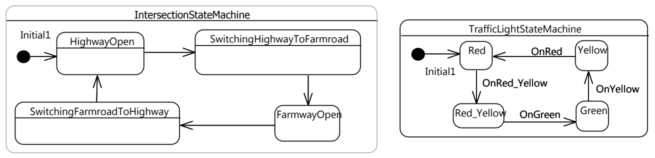
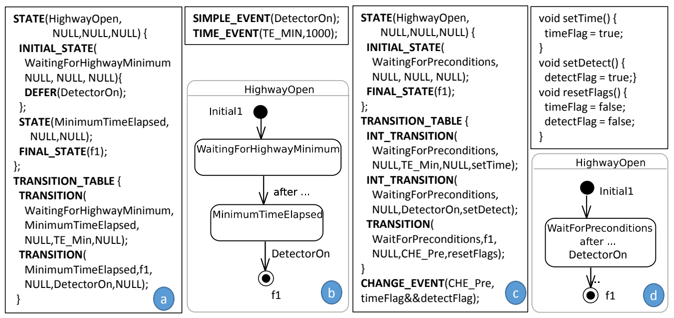


Fig. 13. State machines for describing the behavior of Intersection (left) and TrafficLight (right)



```
STATE(HighwayOpen,
        NULL,NULL,NULL) {
  INITIAL_STATE(
    WaitingForHighwayMinimum
    NULL, NULL, NULL){
    DEFER(DetectorOn);
  };
  STATE(MinimumTimeElapsed,
     NULL,NULL);
  FINAL_STATE(f1);
};
TRANSITION_TABLE {
  TRANSITION(
    WaitingForHighwayMinimum,
    MinimumTimeElapsed,
    NULL,TE_Min,NULL);
  TRANSITION(
    MinimumTimeElapsed,f1,
    NULL,DetectorOn,NULL);
 }
```

```
SIMPLE_EVENT(DetectorOn);
TIME_EVENT(TE_MIN,1000);
```

```
STATE(HighwayOpen,
      NULL,NULL,NULL) {
  INITIAL_STATE(
    WaitingForPreconditions,
    NULL, NULL, NULL);
  FINAL_STATE(f1);
};
TRANSITION_TABLE {
  INT_TRANSITION(
    WaitingForPreconditions,
    NULL,TE_Min,NULL,setTime);
  INT_TRANSITION(
    WaitingForPreconditions,
    NULL,DetectorOn,setDetect);
  TRANSITION(
    WaitForPreconditions,f1,
    NULL,CHE_Pre,resetFlags);
}
CHANGE_EVENT(CHE_Pre,
timeFlag&&detectFlag);
```

```
void setTime() {
  timeFlag = true;
}
void setDetect() {
  detectFlag = true;}
void resetFlags() {
  timeFlag = false;
  detectFlag = false;
}
```

Fig. 14. Alternative state machine design for the *HighwayOpen* state

The other design utilizes a *ChangeEvent* for triggering the switching the active sub-state of *HighwayOpen* from *WaitForPreconditions* to a final state. The expression associated with the change event changes from false to true once two flags *timeFlag* (for *minimum time elapsed*) and *detectFlag* (for *vehicle detected on the farmroad*) are set. *timeFlag* is set by executing the effect *setTime* of the internal transition of *WaitForPreconditions* triggered by the *TE_MIN* time event. And *detectFlag* is set by the method *setDetect* once a *DetectorOn* event occurs to trigger another internal transition of *WaitForPreconditions*.

The architect creates the state machine for *Intersection* for better understanding. For the programmer, he develops the low-level behavior and creates the state machine for *TrafficLight* textually, from scratch. These actors worked in parallel and synchronized their assignment after finishing. The synchronization is realized by our previously presented process. For simulation, we reuse the detector class developed in [41] to automatically generate *DetectorOn/DetectorOff* signals.

X. COMPARISON AND RELATED WORK

A. *Round-trip engineering and co-evolution*

Some RTE techniques restrict the development artifact to avoid synchronization problems. Partial RTE and protected regions are introduced in [42] to preserve code editions which cannot be propagated to models. The mechanisms as discussed in Section I are used for the separation of generated and non-generated code. EMF implements these techniques to allow users to embed user-code replacing the default generated code. Yu et al. [43] propose to synchronize user-code and generated code through bidirectional transformation. However, the mechanisms does not allow modifications in regions beyond the marked ones and thus prohibit the programmers from changing USMs' topology. *Deep separation* proposed in [19] overcomes the limitations of the current separation mechanisms. However, it does not allow to modify the system architecture- and behavior-prescribed code in text-based development environment.

The underlying idea of RAOES is similar to ArchJava [20], [21], and the approach in [44]. The latter utilizes Java annotations to preserve architecture intention in the source code. However, these approaches try to embed architectural information into the code, specifically Java, while RAOES embeds the behavior represented by USMs into C++.

B. Language engineering

Several languages such as PlantUML[4], Umple [45], and Earl Grey (EG) [46] support the text-based modeling of UML State Machine (TML). UML class and state machine elements are usually available in these languages. However, they lack the explicit support for event types definitions used in UML. Furthermore, these languages do not allow the programmers to reuse the existing syntax of C++ but redefine it in their own language and IDE. By this way, they need a new complete compiler, which requires a lot of engineering tasks to develop. Contrarily, RAOES integrates the modeling features into C++ and only requires a slight compiler for the modeling features. RAOES allows the programmers stay familiar with their C++ syntax and existing favorite IDEs while familiarity of these TMLs are also questionable in [46]. Followings list differences in comparison between RAOES and the TMLs.

- RAOES adapts USM features to existing programming languages while Umple or TextUML does inversely, hence RAOES profits all benefices of IDEs such as intelligent completion and easy to implement. Furthermore, RAOES allows to use all specific C++ features such as function pointers for program efficiency, which are not available in the the TMLs.
- In RAOES, the programmers write and maintain the USM-based behavior part in the same class/file containing the active class.
- RAOES support full USM features.
- RAOES automatically synchronizes the code with the system model specified by UML.
- RAOES defines the state machine topology separately from the transition table and event definition.

mbeddr [47] proposes an extensible C-based programming language to support high-level abstractions for embedded system implementation. The idea of *mbeddr* is similar to Umple's by introducing a new editor to mix high-level and low-level code for effective embedded system development. Furthermore, *mbeddr* is a code-centric approach and UML event types are not explicitly supported whereas RAOES is a model-code hybrid approach focusing on the collaboration between software architects and programmers and the evolution of artifacts.

In [48], the authors embeds behavior models into Java by representing, for example, states as classes, transitions as annotations, guards as methods. Although the programmers can modify the code following these patterns, the code size is large because of class explosion.

C. Code generation patterns and tools

Tools such as IBM Rhapsody [7], Enterprise Architect [36], and Papyrus-RT [3] support only the structure RTE for UML class diagram concepts and code generation from UML State Machines. Techniques for generating code from USM such as SWITCH/IF, state table [5] and state pattern [6], [28] are proposed. A systematic review of code generation approaches is presented in [49]. However, only a subset of USM features is supported and generated code requires much memory because which is not preferred in embedded systems [50]. RAOES offers code generation for all USM concepts. Therefore, users are free and flexible to create there USM conforming to UML without restrictions.

[4]PlantUML, http://plantuml.com/

XI. Conclusion

We have presented RAOES-a round-trip engineering approach for effective collaboration between software architects and programmers in developing and maintaining reactive embedded systems using UML State Machines for describing behaviors. RAOES introduces a C++ front-end code lying between models and actual executable code. RAOES proposes a framework for synchronizing the models and the front-end code, and generating executable code from the front-end. RAOES is implemented as an extension of the Papyrus modeling tool.

We evaluated RAOES by conducting experiments on the round-trip engineering correctness, the semantic-conformance and efficiency of generated code. 300 random models are tested for the RTE correctness. The conformance is tested under PSSM that 62/66 tests passed. For efficiency of code, RAOES produces code that runs faster in event processing time and is smaller in executable size than those of other approaches (in the paper scope).

For the moment, RAOES supports the co-evolution of UML class and state machine diagrams and code. In the future, we will integrate component-based concepts into C++ to make the front-end more powerful and effective. Our wish is to embed component-connector and interaction component information into C++ to stay synchronized with composite structure diagrams in UML and elaborate the application range such as distributed embedded systems.

However, code produced by PSM consumes slightly more memory than the others. Furthermore, some PSSM tests are failed. Therefore, as a future work, we will fix these issues by making multi-thread part of generated code more concise.

References

[1] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/1182807.1182811

[2] S. Kent, "Model driven engineering," in *International Conference on Integrated Formal Methods*. Springer, 2002, pp. 286–298.

[3] E. Posse, "Papyrusrt: Modelling and code generation."

[4] 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

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

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

[7] IBM, "Ibm Rhapsody." [Online]. Available: http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/
[8] 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
[9] QM, "Qm," http://www.state-machine.com/qm/, 2016, [Online; accessed 14-May-2016].
[10] 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.
[11] 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.
[12] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 471–480. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985858
[13] 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.
[14] S. Sendall and J. Küster, "Taming Model Round-Trip Engineering."
[15] SparxSystems, "Enterprise Architect," Sep. 2016. [Online]. Available: http://www.sparxsystems.eu/start/home/
[16] M. Langhammer, "Co-evolution of component-based architecture-model and object-oriented source code," in *Proceedings of the 18th international doctoral symposium on Components and architecture*. ACM, 2013, pp. 37–42.
[17] M. E. Kramer, M. Langhammer, D. Messinger, S. Seifermann, and E. Burger, "Change-driven consistency for component code, architectural models, and contracts," in *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. ACM, 2015, pp. 21–26.
[18] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.
[19] Y. Zheng and R. N. Taylor, "Enhancing architecture-implementation conformance with change management and support for behavioral mapping," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 628–638.
[20] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE, 2002, pp. 187–197.
[21] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: a contract place where architectural design and code meet together," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 75–84.
[22] MSM, "Meta State Machine," http://www.boost.org/doc/libs/1_59_0_b1/libs/msm/doc/HTML/index.html, 2016, [Online; accessed 04-July-2016].
[23] "State Machine Benchmark." [Online]. Available: http://www.boost.org/doc/libs/1_61_0/libs/msm/doc/HTML/ch03s04.html
[24] OMG, "Precise Semantics Of UML Composite Structures," no. October, 2015.
[25] 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.
[26] "Papyrus/Designer/code-generation - Eclipsepedia." [Online]. Available: http://wiki.eclipse.org/Papyrus/Designer/code-generation
[27] 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.
[28] I. A. Niaz, J. Tanaka, and others, "Mapping UML statecharts to java code." in *IASTED Conf. on Software Engineering*, 2004, pp. 111–116. [Online]. Available: http://www.actapress.com/PDFViewer.aspx?paperId=16433
[29] 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.
[30] "Moka Model Execution." [Online]. Available: https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
[31] boost, "Boost C++," http://www.boost.org/, 2016, [Online; accessed 04-July-2016].
[32] "State Machine Benchmark." [Online]. Available: http://boost-experimental.github.io/msm-lite/benchmarks/index.html
[33] Boost, "Simple CDPlayer Example," http://www.boost.org/doc/libs/1_45_0/libs/msm/doc/HTML/ch03s02.html#d0e424, 2016, [Online; accessed 14-May-2016].
[34] ——, "Composite CDPlayer Example," http://www.boost.org/doc/libs/1_45_0/libs/msm/doc/HTML/ch03s02.html#d0e554, 2016, [Online; accessed 14-May-2016].
[35] N. Magic, "Magic Draw," https://www.nomagic.com/products/magicdraw.html, 2016, [Online; accessed 14-Mar-2016].
[36] SparxSysemx, "Enterprise Architect," http://www.sparxsystems.com/products/ea/, 2016, [Online; accessed 14-Mar-2016].
[37] B. Library, "The Boost Statechart Library," http://www.boost.org/doc/libs/1_61_0/libs/statechart/doc/index.html, 2016, [Online; accessed 04-July-2016].
[38] "Valgrind Massif." [Online]. Available: http://valgrind.org/docs/manual/ms-manual.html
[39] R. H. Katz and G. Borriello, "Contemporary logic design," 2005.
[40] Yasmine, "The classic farmroad example," http://yasmine.seadex.de/The_classic_farmroad_example.html.
[41] "Farmroad Example." [Online]. Available: http://yasmine.seadex.de/The_classic_farmroad_example.html
[42] K. Czarnecki, M. Antkiewicz, and C. H. P. Kim, "Multi-level customization in application engineering," *Communications of the ACM*, vol. 49, no. 12, p. 60, Dec. 2006.
[43] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "Maintaining invariant traceability through bidirectional transformations," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 540–550.
[44] H. B. Christensen and K. M. Hansen, "Towards architectural information in implementation (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 928–931.
[45] T. C. Lethbridge, A. Forward, and O. Badreddin, "Umplification: Refactoring to incrementally add abstraction to a program," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 220–224.
[46] M. Mazanec and O. Macek, "On general-purpose textual modeling languages." Citeseer, 2012.
[47] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "mbeddr: an extensible c-based programming language and ide for embedded systems," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 2012, pp. 121–140.
[48] M. Balz, M. Striewe, and M. Goedicke, "Embedding behavioral models into object-oriented source code."
[49] 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.
[50] 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.