# RAOES: Round-trip Engineering for Event-Driven Embedded System

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*—Event-driven architecture is an useful way to design memory-constrained embedded systems. Unified Modeling Language State Machine and its visualization are a powerful means to the modeling of the logical behavior of such architecture. 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.**

**In this paper, we tackle 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 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 in their preferred practice.**

## I. Introduction

Event-driven architecture is used for designing memory-constrained embedded systems [1]. UML state machines (USMs) and their visual representations are an efficient means to describing, analyzing and implementing high level logic behaviors of event-driven systems. 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], [4], [6], [7], [8].

Ideally, a full model-centric approach is preferred by MDE community due to its advantages [9]. However, in industrial practice, there is significant reticence [10] 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.

The code modified by programmers and the model are then inconsistent. Round-trip engineering (RTE) [11] is proposed to synchronize different software artifacts, model and code in this case [12]. 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 [6] and Enterprise Architect [13]. Some approaches such as [14], [15] allow the co-evolution of component-based diagram elements and code.

The *behavior* RTE allows programmers to partially modify behavioral code in limited areas by separating the generated and non-generated code [6], [16] using some specialized comments such as `@generated NOT`. 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 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* [17].

`Deep separation` is proposed by the authors in [17] to overcome the limitation of the current separation mechanisms. However, as the authors state that, this approach does not allow to modify the system architecture- and behavior-prescribed code in text-based development environment.

In this paper, we tackle the problem of synchronization between model, which includes both structural and behavioral elements, and C++ code for developing event-driven systems. Specifically, the system architecture is specified via UML class diagrams and the behavior via USMs. 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` [18] and `Archface` [19] 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` [20] and `EUML` [21]. 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 of RAOES.

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 our synchronization strategy. The implementation of RAOES is described in Section VII. Section VIII reports our results of experimenting with RAOES. Section IX shows related work. The conclusion and future work are presented in Section XI.

## II. Background definition

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

This section presents the background of UML State Machine. Fig. 1 shows the state machine meta-model used in this paper. The latter covers the 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.

A vertex is either a UML state or a pseudo-state. A state with is `composite` if it contains at least one region; `orthogonal/concurrent` if contains more than one region; otherwise `simple`. A state can have an $entry$, an $exit$ and a $doActivity$ action. A vertex is a pseudo state if it is not a state. A connection point is a vertex, whose kind $\in \{enpoint, expoint\}$, and contained by a composite state.

[1]It is possible to have multiple state machines in the same active class. However, for simplification, this paper assumes that an active class only holds a state machine

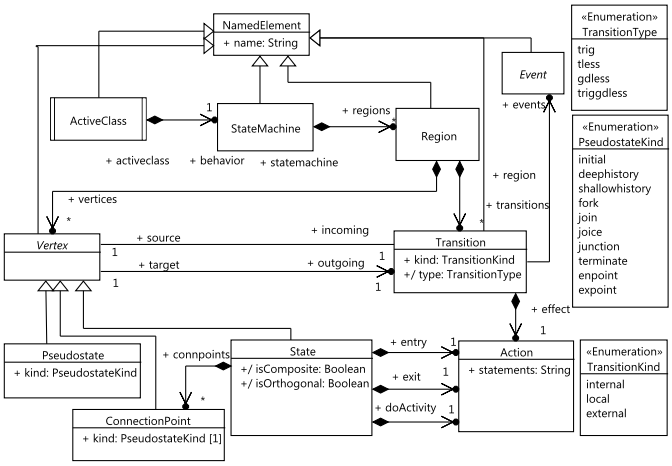


Fig. 1. State machine meta-model

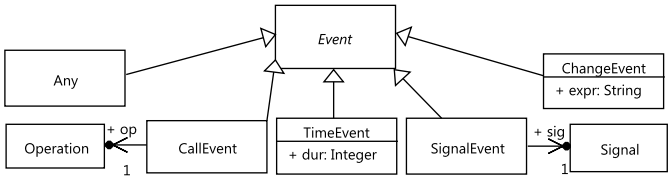


Fig. 2. State machine event meta-model

The detailed semantics of each pseudo state is clearly defined in the UML specification [24] and beyond the scope of this paper.

A transition $t \in T$ is an edge connecting two vertexes, `source and target`. A transition $t$ can have a guard $guard(t)$, an effect $effect(t)$, and is associated with a set of events. A transition is either `external`, `local`, or `internal` whose semantics is defined by the specification.

UML defines five event types. Fig. 2 shows the meta-model of the event types. The description of the events is briefly stated as the following.

**Definition II.1.** An event is one of the followings:

- 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 actors in software development process who will use our model-code RTE technique to collaborate during development. Some basic concepts related to the actors and use-cases, which will be offered by RAOES, are also defined in this section.

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

**Definition II.2.** A model-driven developer (MDD) is an actor who uses the model as the main working artifact. Code-driven developer (CDD) is an actor who uses the code as the main working artifact.

The code, produced from the model automatically, is consistent with the model. A software architect is a kind of the model-driven developer who edits the model to specify the system architecture.

A programmer is a specialization of the code-driven developer. Indeed, programmers may modify our C++ front-end code, such as editing methods, attributes or state machines textually.

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

**Definition II.3** (Batch code generation [22])**.** Batch code generation is a process of generating code from a model, from scratch. Any existing code is overwritten by the newly generated code.

**Definition II.4** (Incremental code generation)**.** Incremental code generation is the process of taking as input an edited model and existing code to update the code by propagating editions in the model to the code.

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

**Definition II.5** (Batch reverse engineering)**.** Batch reverse engineering is a process of producing a model from code, from scratch. The existing model is overwritten by the newly produced model.

**Definition II.6** (Incremental reverse engineering)**.** Incremental reverse engineering is the process of taking as input a edited code, and an existing model, and then updating the model by propagating editions in the code to the model.

In Section VI, the use-cases are integrated into our process, which covers model-code synchronization and is detailed in Section VI.

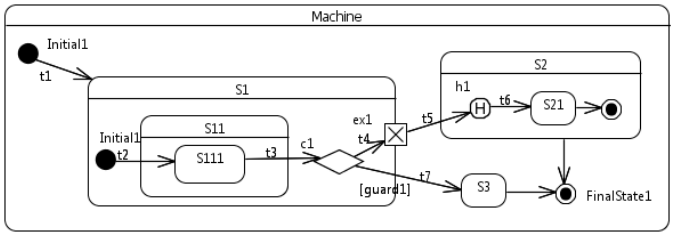


Fig. 3. A USM example

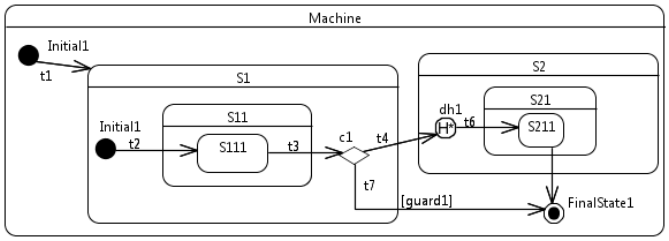


Fig. 4. The evolved version of the USM shown in Fig. 3

## III. Motivating example

Let's consider a collaboration scenario between software architects and programmers in developing an event-driven paradigm-based system `System`. The latter's behavior is described by using a USM as in Fig. 3. This USM is artificial, and is extracted and customized from the origin in [23]. This USM consists of some simple, composite, and pseudo states such as `choice`, `connection point expoint`, and `junction`.

Although many tools have the ability to generate code from USMs, a few such as IBM Rhapsody [6] 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, 5 (a) and (b) show the code segments generated for the transition outgoing from the state `S111` of the example by using our tool and Rhapsody, respectively.

In this scenario, we assume that, 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 high-level logic behavior by using modeling tools. Assuming that, for some reasons, the USM should be evolved to the next version as in Fig. 4.

For simplification, we assume that no effects are associated with the transitions in the examples. In Fig. 5 (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. If the transition

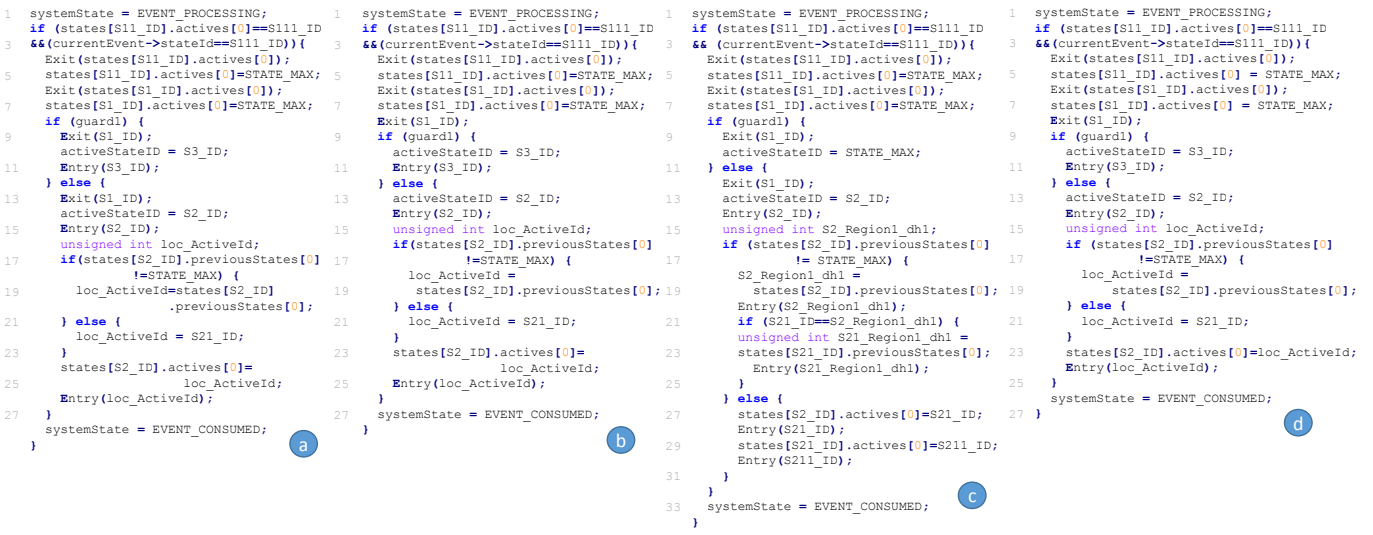


Fig. 5. Codes generated from the state machine example in Fig. 3 by using our tool (a) and Rhapsody (b), and their respective evolved versions

`t4` is selected, the exit action of `S1` is called (line 13) and followed by the entry action (line 15) and the restoration of the previous active sub-state of `S2` (lines 16-26). If the other transition `t7` is taken, `S1` and `S3` are exited and entered (lines 9-11), respectively.

The generated code in Fig. 5 (b) for the evolved version of the example in Fig. 4 differs from that of Fig. 5 (a) by the way the history of `S2` is restored. Fig. 5 (b) executes a deep restoration in lines 14-32 if `guard` is evaluated as false.

It is worth noting that, the codes are generated following some patterns, which are not explicitly understandable for the programmers to capture the control flow of the USM associated with the code. Hence, it is quite challenging to modify the topology of the USM at the code level. Even, if the programmers could understand and modify the code following the defined patterns, which require a very high discipline, it is still very difficult for RTE tools to decipher and reflect the code changes to the model.

Furthermore, different code generation patterns produce different code looks. As a result, it is very hard, if not impossible, to find common rules to reconstruct the original state machine from the code. This is since existing tools such as Rhapshody supporting round-trip engineering 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, using the heavy modeling tools, which are usually pricey, does not guarantee the seamless collaboration between the different 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 event-driven systems' development. In the latter, the behavior of active objects is specified by using UML State Machines. To do it, RAOES defines a mechanism interface 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. 5, RAOES produces a front-end C++ code. The latter plays as an intermediate representation, which is C++-conformant. Fig. 6 shows how RAOES is different from the existing approaches.

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, by using RAOES, the generated front-end code for the example in Fig. 3 and its evolved version are presented in Fig. 7. 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.

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

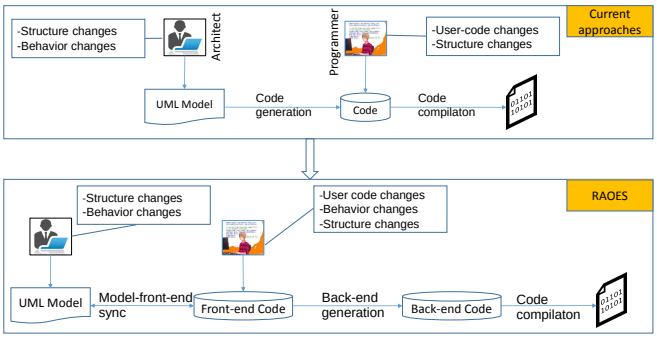


Fig. 6. From existing approaches to RAOES

```
1  class System {
2    STATE_MACHINE(Machine) {
3      INITIAL_STATE(S1,S1_entry,S1_exit,NULL){
4        INITIAL(Initial1);
5        STATE(S11, S11_entry, S11_exit) {
6          STATE(S111, S111_entry, S111_exit);
7        };
8        CHOICE(c1);
9        EXIT_POINT(ex1);
10     };
11     STATE(S2, S2_entry, S2_exit) {
12       SHALLOW_HISTORY(h1);
13       STATE(S21, S21_entry, S21_exit);
14       FINAL_STATE(S2_final);
15     };
16     STATE(S3, S3_entry, S3_exit);
17     FINAL_STATE(FinalState1);
18     //Event table definitions
19     //Transition table
20     TRANSITION_TABLE {
21       TRANSITION(S111,c1,NULL,NULL,NULL);
22       TRANSITION(c1,ex1,NULL,NULL,NULL);
23       TRANSITION(c1,S3,NULL,NULL,NULL);
24     }
25   };
26   void S1_entry() {
27     //Entry action for S1
28   }
29   //...
30   //class member declarations
31 };
```

a

```
1  class System {
2    STATE_MACHINE(Machine) {
3      INITIAL_STATE(S1,S1_entry,S1_exit,NULL){
4        INITIAL(Initial1);
5        STATE(S11, S11_entry, S11_exit) {
6          STATE(S111, S111_entry, S111_exit);
7        };
8        CHOICE(c1);
9      };
10     STATE(S2, S2_entry, S2_exit) {
11       DEEP_HISTORY(dh1);
12       STATE(S21, S21_entry, S21_exit) {
13         STATE(S211, S211_entry, S211_exit);
14       };
15     };
16     FINAL_STATE(FinalState1);
17     //Event table definitions
18     //Transition table
19     TRANSITION_TABLE {
20       TRANSITION(S111,c1,NULL,NULL,NULL);
21       TRANSITION(c1,dh1,NULL,NULL,NULL);
22       TRANSITION(c1,FinalState1,NULL,NULL,NULL);
23     }
24   };
25   void S1_entry() {
26     //Entry action for S1
27   }
28   //...
29   //class member declarations
30 };
31
```

b

Fig. 7. Front-code evolution

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.

## 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. 7, 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 ';'

**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.

**Example:** Lines 2-10 in Fig. 7 (a) represents the state `S1` of the USM example in Fig. 3 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. 7 represent a connection exit point and a shallow history

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

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

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.

**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 since 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, effect');'

`local`→'LOCAL_TRANSITION('src,tgt,guard,evt,effect');'

`internal` → 'INT_TRANSITION('src, guard, evt, effect');'

**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.
- `effect` 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, `effect` becomes NULL.

**Example:**

## VI. Process for synchronization

This section describes our generic artifact synchronization methodology pattern to synchronize a model with USMs and front-end code. We assume that an IDE used by software architects and programmers offers the set of use-cases defined in Section II-B. Our methodology allows concurrent modifications made to the model and front-end code so that both can be used for the full implementation of a system, rather than just architectural design for the former, and code implementation for the latter.

We propose two synchronization strategies for this scenario. The general approach 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 a `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 8 shows the first synchronization strategy based on using front-end code as the SA. The general steps of the

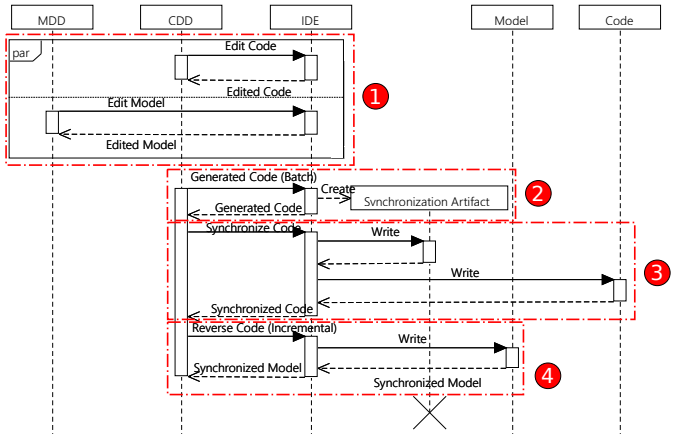


Fig. 8. Synchronization process, in which the model and the code are concurrently edited with code as the SA (CDD = Code-Driven Developer, MDD = Model-Driven Developer, Code = C++ front-end code). The API calls for Model and Code are represented generically as "Read" and "Write".

process shown in Figure 8 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.

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

A prototype is implemented to realize our approach. This section presents the architecture and implementation detail of this prototype based on the Eclipse Modeling Framework (EMF). The latter provides many facilities such as UML Java libraries, to ease the development. Fig. 9 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.

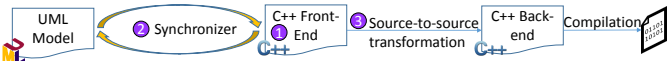


Fig. 9. RAOES's architecture

### 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. This front-end is then used by a source-to-source (C++-to-C++) transformation to generate a C++ back-end code. 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.

The underlying of some concepts in our implementation is shown in Listing 1. To easily recognize the state machine element types in RAOES, we use specialized names to 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 describing the behavior of an active class 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 transition row in the transition table is generated.

Step 4 Each state *entry/exit/doActivity* or transition effect is transformed into a method, which is referred by the USM's topology written in RAOES. The body of the transformed methods can be embedded directly

Listing 1. The underlying representation of the C++ front-end

```
#define STATE(name,ent,ex)  class STATE___ ## name ## ___ ## ent ## ___ ## ex ## ___
#define INITIAL_STATE(name,ent,ex,effect) class INITIAL_STATE___ ## name ## ___ ## ent ## ___ \
                    ## ex ## ___ ## EFFECT ## ___ ## effect
#define REGION(name)  class REGION___ ## name ## ___
#define STATE_MACHINE(name) class STATE_MACHINE___ ## name ## ___
#define CALL_EVENT(name,operation) class CALL_EVENT___ ## name ## ___OPERATION___ ## operation {};
#define TIME_EVENT(name,duration) class TIME_EVENT___ ## name ## ___DURATION___ ## duration {};
#define SIGNAL_EVENT(name,signal) class SIGNAL_EVENT___ ## name ## ___SIGNAL___ ## signal {};
#define SIMPLE_EVENT(name) class SIGNAL_EVENT___ ## name ## ___SIGNAL___NULL{};
#define CHANGE_EVENT(name,expression) class CHANGE_EVENT___ ## name ## ___EXPRESSION___ \
        {const char* func() {return #expression;}};
#define TRANSITION_TABLE void TRANSITION_TABLE___operation(int transition_len)
#define TRANSITION(source, target, guard, event, effect) transition_len = strlen("transition") + \
    strlen(#source) + strlen(#target) + strlen(#guard) + strlen(#event) + strlen(#effect);
```

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 [25].

*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 affected by the changes made to the model. Therefore, it needs to know which model elements have been changed. Hence, we implement a model listener which is based on the EMF transaction mechanism.

The listener is hooked to the Papyrus modeling tool - a set of EMF plug-ins. It detects different changes made to the model. Fig. 10 shows the model change classification. 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 actions. The latter are used

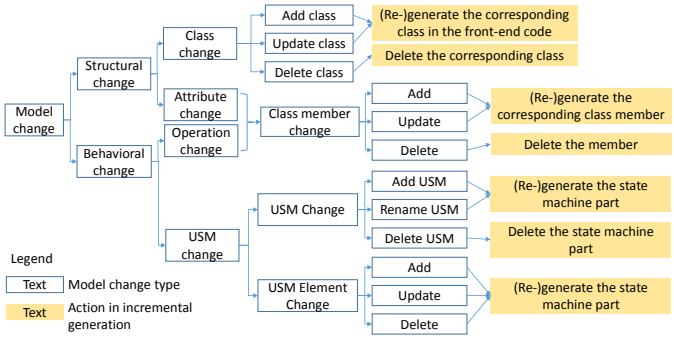


Fig. 10. Model change classification and management in incremental generation

to update the front-end code. For example, in Fig. 10, 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, if model elements are unchanged, the associated code elements are kept intact.

*b) Incremental reverse engineering from front-end code to model:* This is the inverse direction of the incremental code generator. Similarly to the latter, it needs to have a code listener, which detects 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. 10. 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. 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. 11 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


Fig. 11. Source-to-source transformation via reverse engineering and code generation

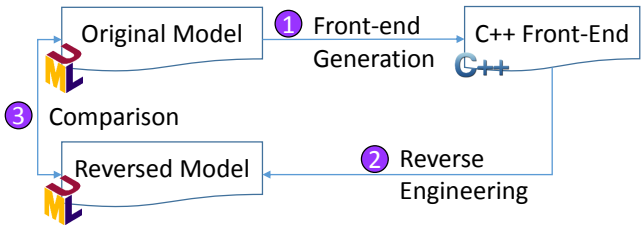


Fig. 12. Evaluation methodology to answer RQ1

generators. Although there are many approaches and tools supporting code generation for USMs, a complete approach is still missing [26], especially when considering concurrency. Therefore, in order to support full synchronization of USMs and code, we need to design an approach to generating code for USMs with full features.

Our code generation approach combines the state pattern in [27] and IF/ELSE constructions, and extends the support of these patterns for all of pseudo states and events defined in USMs. The detail of this generation approach is not presented here due to space limitation.

## VIII. Evaluation

In order to evaluate RAOES, we conducted experiments focusing on different aspects. The followings present the research questions and the experimental results.

### A. Reversing generated code

**RQ1**: A state machine `sm` is used for generating the front-end code. The latter is reversed engineered to produce another state machine `sm'`. Are `sm` and `sm'` identical? In other words: whether the front-end code generated from USMs model can be used for reconstructing the original model. This question is related to the *GETPUT* law defined in [28].

Fig. 12 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:

- Step 1 `C++ front-end code` is generated from an `original model`.
- Step 2 The `C++ front-end code` is reverse engineered to a `reversed model`
- Step 3 The `reversed model` is then compared to the `original model`.

Random models are automatically generated by a configurable model generator. The latter can generate a desired average number of vertexes, transitions, and events. For each model, a context class and its behavior described by a USM

TABLE I
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 |

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 I 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

**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)?.

To evaluate the semantic conformance of runtime execution of the back-end code, we use a set of USM examples provided by `Moka` [29]. The latter is a model execution engine offering PSSM. Fig. 13 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. Table II shows the test results for each state machine concept provided

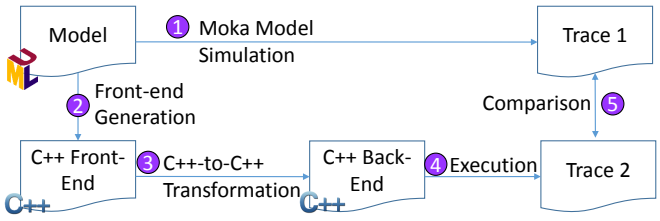


Fig. 13. Semantic conformance evaluation methodology

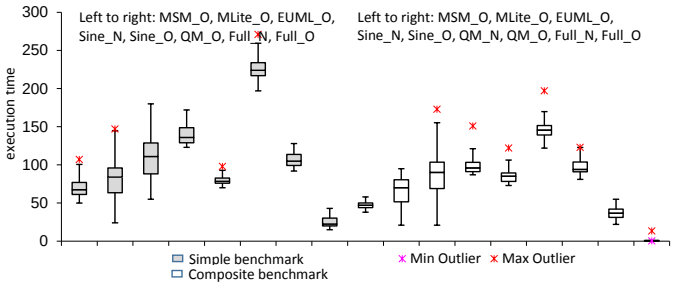


Fig. 14. Event processing speed for the benchmark

by PSSM. The results are promising that RAOES passes 62/66 tests. In fact, RAOES fails with some wired tests such as transitions outgoing 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, on event support, only `SignalEvent` is specified. On pseudo-states, histories are not supported. Thus, our evaluation result is limited to the current specification of PSSM.

**Threats to validity:** Internal threat is that, all test cases of the PSSM test suite are contained in a single model file. However, the input to our experiments requires a test case per model file. Furthermore, operation behaviors, in PSSM, are defined by activities while our prototype defines behaviors as code blocks embedded into models. Therefore, we manually re-create these tests and convert activities into programming language code.

### C. Benchmarks

In this section, we present the results obtained through the experiments on some optimization aspects of generated code. Specifically, two questions related to memory consumption and runtime performance of generated code are posed.

**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?

**Experimental dataset:** Two state machine examples are obtained by the preferred benchmark used by the Boost C++ libraries [30] in [31]. One simple example [32] only consists of atomic states and another [33] both atomic and composite states. Tools such as Sinelabore (which efficiently generates code from UML State Machines created by various modeling tools such as Magic Draw [34], Enterprise Architect [35]) and QM [**?**], which generate code from state machines, and C++ libraries (Boost Statechart [36], Meta State Machine (MSM) [20], C++ 14 MSM-Lite [31], and functional programming like-EUML[21]) 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.

*1) Speed:*

**Results:** Table III shows the median of event processing time. In the normal compilation mode (N), Boost Statechart, MSM, MSMLite, EUML are quite slow. 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. Fig. 14 compares the performance of these approaches to that of our approach. 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:*

**Result:** Table IV 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[37] to measure memory usage. Table V 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).

TABLE II
SEMANTIC-CONFORMANCE TEST RESULTS (NUMBER OF PASSED/TOTAL TESTS)

| Behavior | Choice | Deferred Events | Entering | Exiting | Entry | Exit | Event | Final | Fork | Join | Transition | Terminate | Others |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5/6 | 3/3 | 6/6 | 5/5 | 4/5 | 5/5 | 3/3 | 9/9 | 1/1 | 2/2 | 2/2 | 11/14 | 3/3 | 2/2 |

TABLE III
EVENT PROCESSING SPEED IN MS

| Test | SC | | MSM | | MSM-Lite | | EUML | | Sinelabore | | QM | | PSM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | N | O | N | O | N | O | N | O | N | O | N | O | N | O |
| Simple | 13706 | 1658 | 5250 | 71 | 834 | 79 | 10868 | 110 | 141 | 80 | 286 | 229 | 107 | 25,4 |
| Composite | 5353 | 821 | 3546 | 47 | 517 | 65 | 4225,6 | 92 | 100 | 86 | 146 | 98 | 36,5 | 1,40 |

TABLE IV
EXECUTABLE SIZE IN KB

| Test | SC | | MSM | | MSM-Lite | | EUML | | Sinelabore | | QM | | PSM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 V
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 |

## IX. RELATED WORK

Code generation from state machines has been received huge attention in automated software development and many approaches are proposed. This section mentions some usual patterns and how our approach differs. A systematic review of proposals is presented in [38].

Switch/if is the most intuitive technique implementing a "flat" state machine. The latter can be implemented by either using a scalar variable [3] 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 [4] uses one dimension represents states and the other one all possible events. These approaches require a transformation from hierarchical and concurrent USMs to flatten ones. However, the semantics of USMs containing pseudo states such as histories or join/fork are hardly preserved during the transformation.

State pattern [5], [4] 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 [27] to support hierarchical-concurrent USMs. Recently, a double-dispatch (DD) pattern presented in [39] extends [27] to support maintainability by representing states and events as classes, and transitions as methods. However, as the results shown in [39], these patterns require much memory because of explosion of classes and uses dynamic memory allocation, which is not preferred in embedded systems.

Tools, such as [6], [13], apply different patterns to generate code. However, as mentioned in Section I, true concurrency and some pseudo-states are not supported. FXU [40] is the most complete tool but generated code is heavily dependent on their own library and C# is generated.

Our approach combines the classical switch/if pattern, to produce small footprint, and the pattern in [27], to preserve state hierarchy. Furthermore, we define pattern to transform all of USM concepts including states, pseudo states, transitions, and events. Therefore, users are flexible to create there USM conforming to UML without restrictions.

### A. Comparison to text-based state machine specification

-RAOES: Adapt state machine model 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.

-RAOES: programmers write and maintain the USM-based behavior part in the same class/file containing the active class.

-RAOES: full USM features are supported.

-RAOES: automatically synchronize the code with the system model specified by UML.

-RAOES: defines the state machine topology separately from the transition table and event definition.

## X. DISCUSSION

## XI. CONCLUSION

We presented an approach whose objective is to provide a complete, efficient, and UML-compliant code generation solution from UML State Machine. The design for concurrency of generated code is based on multi-thread. The code generation pattern set extends the IF-ELSE/SWITCH patterns and the state pattern extension. The hierarchy of USM is kept by our simple state structure.

We evaluated our approach by implementing a prototype PSM and conducting experiments on the semantic-conformance and efficiency of generated code. The former is tested under PSSM that 62/66 tests passed. For efficiency, we used the benchmark defined by Boost to compare code generated by PSM to other approaches. The results showed that PSM produces code that runs faster in even processing and is smaller in executable size than those of other approaches (in the paper scope).

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] 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

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

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

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

[7] 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

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

[9] 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.

[10] 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.

[11] 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.

[12] S. Sendall and J. Küster, "Taming Model Round-Trip Engineering."

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

[14] 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.

[15] 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.

[16] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[17] 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.

[18] 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.

[19] 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.

[20] 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].

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

[22] 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.

[23] L. Shuang, "Formalizing UML State Machine Semantics for Automatic VerificationâĂŞthe PAT Approach." [Online]. Available: http://www.comp.nus.edu.sg/~pat/uml/techreport/uml_sm_semantics.pdf

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

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

[26] 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.

[27] 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

[28] 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.

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

[30] boost, "Boost C++," http://www.boost.org/, 2016, [Online; accessed 04-July-2016].

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

[32] 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].

[33] ——, "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].

[34] N. Magic, "Magic Draw," https://www.nomagic.com/products/magicdraw.html, 2016, [Online; accessed 14-Mar-2016].

[35] SparxSysemx, "Enterprise Architect," http://www.sparxsystems.com/products/ea/, 2016, [Online; accessed 14-Mar-2016].

[36] 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].

[37] "Valgrind Massif." [Online]. Available: http://valgrind.org/docs/manual/ms-manual.html

[38] 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.

[39] 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.

[40] R. Pilitowski and A. Dereziñska, *Code Generation and Execution Framework for UML 2.0 Classes and State Machines*. Dordrecht: Springer Netherlands, 2007, pp. 421–427. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-6268-1_75