Fostering Software Architect and Programmer Collaboration

Anonymous Authors
Institution
Address
City, State, Country
Email

ABSTRACT

Event-driven architecture is an useful way to design and solve the complexity of today 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. However, despite many useful UML State Machine concepts, the support of existing generation approaches is still limited to simple cases, especially when considering concurrency and pseudo states such as history, junction, and event types.

This paper provides a complete and efficient code generation approach from UML State Machine. The proposal combines IF-ELSE-SWITCH constructions of programming languages and the state pattern with our support for concurrency. Our approach is tested under the Precise Semantics Of State Machine. Efficiently, even supporting multi-thread-based concurrency, binary files compiled from and the event processing speed of runtime execution of code generated by our approach are significantly improved when comparing to other approaches.

CCS Concepts

•**Software and its engineering** → *Software development methods; Collaboration in software development;*

Keywords

Round-trip engineering, Model-driven engineering, Model code co-evolution, UML, C++, Eclipse Modeling Tools

1. INTRODUCTION

The Unified Modeling Language State Machine (USM) [31] and its visualization, standardized by the OMG [23], are a powerful means to model the logical behavior of system complexity. The so-called Model-Driven Engineering methodology relies on two paradigms, abstraction and automation [20].Automation is to reduce the gap between the modeling world and the implementation world by bringing the ability to automatically generating code from diagram-based modeling languages such as USM to executable code.

However, on the one hand, the usefulness of UML State Machine is raised by providing more concepts supporting software architects for modeling and designing complex systems. On the other hand, existing code generation approaches have some issues regarding completeness and efficiency of generated code. Specifically, the following lists some issues of current approaches:

- Existing tools and approaches mainly focus on the sequential aspect while the concurrency of *doActivity* states and orthogonal regions is not taken into account. We investigated on industrial leading tools. Rhapsody does not support *doActivity*, junctions, and truly concurrent execution of orthogonal regions [15] (see 2 for formal definitions). The concurrency of the orthogonal regions is often implemented sequentially. Rhapsody and Enterprise Architect [29] only support *CallEvent* and *TimeEvent* while *SignalEvent* and *ChangeEvent* are missing.
- The support for pseudo states such as history, choice and junction is poor [28, 29] while these are very helpful in modeling.
- Code generated from tools such as [14] and FXU [24] is heavily dependent on their own libraries, which make the generated code not portable.
- Issues of event processing speed, executable file size, and UML semantic-conformance defined by a recent work on the Precise Semantics of State Machine (PSSM) [22].

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

Our approach combines the IF-ELSE constructions and an extension of state pattern [21] with our support for concurrency. Code generated by our approach is tested under the PSSM. Although supporting multi-thread-based concurrency, binary files compiled from and the event processing speed of runtime execution of code generated by our approach are dramatically smaller than some manual implementation approaches and code generation tools.

DOI: 10.475/123_4

To sum up, the contributions of this paper are as followings:

- A multi-thread-based design and complete code generation approach for concurrent USMs.
- An empirical on the semantic-conformance and efficiency of code generated by the proposed approach.

The remaining of this paper is organized as follows: Section 2 presents preliminary USM concepts in a formal way. Thread-based concurrency is designed in Section 3. Based on the design, a code generation approach is proposed Section 4. The implementation and empirical evaluation are reported in Section 5. Section 6 argues related work. The conclusion and future work are presented in Section 7.

2. BACKGROUND DEFINITION

DEFINITION 1. *A UML vertex $v \in V$ has a kind $v.kind \in$ {initial, final, state, comp, conc, join, fork, choice, junction, enpoint, expoint, history}.*

DEFINITION 2. *A region $r \in \mathcal{R}$ is composed of one or several vertexes, and contained by a state s. We write $owner(r) = s$ and $vertices(r)$ is its sub-vertices set.*

DEFINITION 3. *A vertex is either a UML state or a pseudostate. A UML state s is a vertex and $s.kind \in$ {state, comp, conc}. s has an entry, an exit and a doActivity action. A composite state cs contains one or more vertexes. We write $vertices(cs)$ is a set of vertexes contained by cs and, inversely, $owner(v)$ refers to the containing state of the vertex v.*

DEFINITION 4. *An action $act \in ActLang$ is a set of statements written in an object-oriented programming language $ActLang$. A guard is a boolean expression written in $ActLang$.*

DEFINITION 5. *A transition $t \in T$ is an edge connecting two vertexes. The source and target vertexes of an edge ed are obtained by $src(t)$ and $tgt(t)$. A transition has a guard $guard(t)$, an effect $effect(t)$, and is associated with a set of events $\subset E$. We write $events(t)$ as the associated set of events. A transition has a type $t.type \in \{trig, tless, gdless, triggdless\}$ and a kind $t.kind \in external, local, internal$.*

DEFINITION 6. *An event is one of the followings:*

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

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

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

DEFINITION 7. *A state machine sm is a graph specified by $\{V, T\}$ associated with a set of events E. A state machine is a special composite state which has no incoming and no outgoing transitions. A root vertex v is a direct sub-vertex of the state machine, $owner(v) = sm$. The set of regions contained by sm is written $\mathcal{R}$.*

DEFINITION 8. *Transitive container $owner^+(v)$ of a vertex v of a state machine sm is defined as following:*

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

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

DEFINITION 9. *Current active configuration Cfg of a UML state machine sm is a set of candidate UML states which are able to process an incoming event.*

3. THREAD-BASED CONCURRENCY

This section describes our design of concurrency for generated code.

3.1 Thread-based design of generated code

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

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

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

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

- A thread is created for each effect of transitions outgoing from a *fork* or incoming to a *join*.
- Entering a concurrent state s, after the execution of $entry(s)$, a thread is created for each orthogonal region.
- Exiting a concurrent state s, before the execution of $exit(s)$, a thread is also created for each region to exit the corresponding active sub-state.

3.2 Deadlock avoidance

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

4. CODE GENERATION PATTERN

4.1 Assumption

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

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

4.2 State transformation

A common state interface *IState* is created. The interface contains three methods, namely, *entry*, *exit*, and *doActivity* respectively corresponding to three state actions. The benefice of using these methods is to increase performance in invoking state actions. To preserve the hierarchy of composite states, the interface also has two attributes called *actives* and *previousActives* referring to current and previous active sub-states in case of the presence of history states.

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

Listing 1 shows the interface and its instances. *NUM_STATES* is the number of states in the state machine. In the following sections, we only consider *ActLang* as a C++-like. The discussion of other object-oriented languages are much similar since these share the same concepts.

```
typedef struct IState {                                  1
  int previousActives[2];   int actives[2];              2
  void (C::*entry)();  void (C::*exit)();                3
  void (C::*doActivity)();                               4
} IState;                                                5
class C {                                                6
private:                                                 7
  IState states[NUM_STATES];                             8
public:                                                  9
  C() {                                                  10
    states[S0_ID].entry = &C::S0_entry;                  11
    ...                                                  12
  }                                                      13
  void S0_entry {...}                                    14
}                                                        15
```

Listing 1: IState interface and function pointers in C++

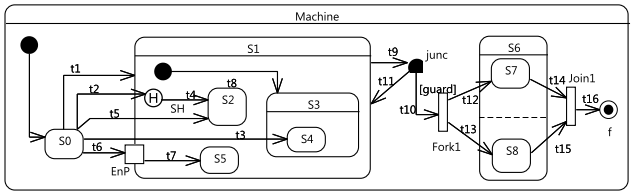

Figure 1: Example illustrating different ways entering a composite state

Each *doActivity* is associated with a permanent thread and a mutex. The *doActivity* thread is initialized, waits for a start signal, executes the *doActivity* code, generates a completion if the state is atomic and still active, and goes back to the waiting point as the paradigm above. Listing 2 shows a code segment for *doActivity* threads. The method in the Listing takes as input a state id to use and call the appropriate mutex and *doActivity*, respectively.

```
void doActivity(int stateId) {                                  1
  isStarts[stateId] = false; //wait flag                        2
  while(true) {                                                 3
    mutex[stateId].lock();                                      4
    while(!isStarts[stateId]) {                                 5
      mutex[stateId].wait();} //await start signal              6
    states[stateId].doActivity();//execute doActivity           7
    isStarts[stateId] = false;//reset wait flag                 8
    mutex[stateId].unlock();                                    9
    if (!isStops[stateId]) {                                    10
      if (stateId == S0_ID || ...) { //atomic states            11
        pushCompletionEvent(stateId);                           12
      }                                                         13
    }                                                           14
  }                                                             15
}                                                               16
```

Listing 2: Example code generated for doActivity

4.3 Region transformation

Our approach considers regions as elements to be transformed. Specifically, each region is transformed into an entering and exiting method. While the entering method controls how a region r is entered from an outside transition t ($src(t) \notin vertices(r)$), the exiting method exits completely a region by executing exit actions of sub-states from innermost to outermost.

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

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

All of the entering ways execute the entry action of the containing composite state $entry(ctner(r))$ after $effect(t)$.

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

The entering method associated with the region r of *S1* has a parameter *enter_mode* indicating how actions should be executed. *enter_mode* takes values depending the number of transitions coming to the composite state $S1$: $\#values(s) =$

$\#\{v \in vertices(s) | v.kind = initial\} +$
$\#\{v \in vertices(s) | \exists t \in T_{ins}(v), src(t) \notin vertices(s)\} +$
$\#\{v \in vertices^{+}(s) \backslash vertices(s) | \exists t \in T_{ins}(v), src(t) \notin vertices^{+}(s)\}.$

The detail of how these modes are implemented in specific languages are not discussed here. Listing 3 shows the C++-like example code generated for r.

```
void S1Region1Enter(int enter_mode){                                    1
if (enter_mode == DEFAULT) {                                            2
  states[S1_ID].actives[0] = S3_ID;                                     3
  states[S3_ID].entry();  sendStartSignal(S3_ID);                       4
  S3Region1Enter(DEFAULT);                                              5
} else if (enter_mode == S2_MODE) {                                     6
  //..                                                                  7
} if (enter_mode == SH_MODE) {                                          8
  StateIDEnum his;                                                      9
  if (states[S1_ID].previousActives[0]!=STATE_MAX){                     10
    his = states[S1_ID].previousActives[0];                             11
  } else {                                                              12
    his = S2_ID;                                                        13
  }                                                                     14
  states[S1_ID].actives[0] = his;                                       15
  states[his].entry();  sendStartSignal(his);                           16
  if (S3_ID == his) {                                                   17
    S3Region1Enter (S3_REGION1_DEFAULT);                                18
  }                                                                     19
} else if (enter_mode == S4_MODE) {                                     20
  states[S1_ID].actives[0] = S3_ID;                                     21
  states[S3_ID].entry();  sendStartSignal(S3_ID);                       22
  S3Region1Enter(S4_MODE);                                              23
} else if (enter_mode == ENP_MODE) {...}                                24
```

Listing 3: Example code generated for the region of S1

By default, the active sub-state of the region is set after the execution of any effect associated with the initial transition, $S3$ is set as active sub-state of $S1$. Entering on a direct sub-state (*S2*) sets the active sub-state of *S1* directly to *S2*. In case of an indirect sub-state ($S4$), the entry action of $S3$ is executed before $S4$ is set as the active-sub state of $S3$ and the execution of $entry(S4)$. It is worth noting that after the execution of each entry, a start signal is sent to activate the waiting thread associated with *doActivity* of the corresponding state.

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

4.4 Event and transition transformation

4.4.1 Events

Similar to the approach in [21], one method mtd_e is generated for each event e. An event enumeration *EventId* is created whose children are event identifiers associated with events. Besides the explicitly defined events of the state machines, the event list of a state machine *sm* contains a special event called *CompletionEvent*, which is implicitly implemented as a *CallEvent*. For each event type, the pattern is realized as in Table 1.

As above presented, all asynchronous incoming events are stored in a FIFO priority queue, in which each event type has a configurable priority. *CompletionEvent* always has

Table 1: Event code generation pattern

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

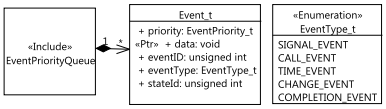

Figure 2: Event data structure

the highest priority. Others are equal by default. Fig. 2 shows the generic structure of events stored in the queue. Event type, priority, identifier, associated state (*stateId*) and data are specified in the structure. The associated state *stateId* is used to define the scope of *Completion Event*s. The event method associated with *Completion Event* executes a check on *stateId* (see Listing 5, line 1). The event data contain marshaled[1] parameters of *SignalEvent*'s signal or *CallEvent*'s parameters.

4.4.2 Transitions

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

- Vertex depth $dp(v)$ is defined as:

$$dp(v) = \begin{cases} 1 & v \text{ is a root vertex} \\ dp(ctner(v)) + 1 & otherwise \end{cases} \quad (2)$$

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

The procedure in Listing 4 describes how the generation process works with an event. It first finds the innermost active states which are able to react e by orderly looping over Lm_e. For each transition outgoing from an innermost state, code for active states and deferral events, guard checking and transition code segments are generated by $GENERATE\_STATE\_EVENT\_CHECK$, $GENERATE\_GUARD(t)$ and *GENTRANS*, respectively. If the identifier of e is equal to one of the deferred event list of the corresponding state (not shown in this paper), $GENERATE\_STATE\_EVENT\_CHECK$ generates code, which checks whether the event to be deferred and pushes the event to a deferral event queue managed by the main thread, which also pushes the deferred events back to the main queue once one of the pending events is processed.

```
∀ item ∈ Lm(e)                                          1
  ∀s ∈ Map_e(item)                                      2
    T_s = {t ∈ T_trig(e)|src(t) = s}                    3
    ∀t ∈ T_s                                            4
      GENERATE_STATE_EVENT_CHECK(s, t, e)               5
      GENERATE_GUARD(t)                                 6
      GENTRANS(s, t, tgt(t))                            7
```

Listing 4: Generation process for an event

For a transition t, $GENERATE\_STATE\_EVENT\_CHECK$ can generate single or multiple active state checking code.

[1]https://en.wikipedia.org/wiki/Marshalling_(computer_science)

Table 2: Code generation procedure for transition: GENTRANS

Input	A source v_s, a target vertex v_t and a transition t
Output	Generated code for transition
Step 1	Find entry and exit states $H_s = v_s \cup ctner^+(v_s)$, $H_t = v_t \cup ctner^+(v_t)$ $s_{ex} \in H_s, s_{en} \in H_t|ctner(s_{ex}) = ctner(s_{en})$
Step 2	Generate IF-ELSE statements for junctions
Step 3	If s_{ex} is a state For $r \in regions(s_{ex})$ $FORK(RegionExit(r))$ Generate JOIN for threads created above Generate sendStopSignal to s_{ex} $exit(s_{ex})$
Step 4	If $v_t.kind = join$ For $in \in T_ins(v_t)$ $FORK(effect(in))$ Generate JOIN for threads created above
Step 5	If $v_t.kind \neq join$ $effect(t)$
Step 6	If s_{en} is a state $entry(s_{en})$ Generate sendStartSignal to s_{en}
Step 7	If $s_{en}.kind \in \{conp, conc\}$ For $r \in regions(s_{en})$ $FORK(RegionEnter(r))$ Generate JOIN for threads created above
Step 8	If $s_{en}.kind \notin \{conp, conc\}$ Generate for pseudo states by patterns

The latter occurs if $tgt(t)$ is a *join*. The detailed discussion on these is not presented due to space limitation. Listing 5, line 2-3 show a portion of code, with multiple checking, generated for processing *Completion Event* triggering transitions *t14* and *t15* outgoing from *S6* and *S7*, respectively, to *Join1*. In addition, the code portion checks the state associated with the current event, which is a completion emitted upon the completion of either *S6*'s or *S7*'s *doActivity*, and saved in the event queue of the active class *C*. Lines 4-6 of the portion concurrently exits the sub-states of *S6* by using *FORK* and *JOIN* for methods associated with *S6*'s orthogonal regions, which actually exit tiS7 and *S8*. Then, *exit(S6)* is executed before the concurrency of transition effects *t14* and *t15* is taken into account.

```
if(event.stateId==S6_ID||event.stateId==S7_ID){          1
  if (states[S6_ID].actives[0] == S7_ID &&              2
    states[S6_ID].actives[1] == S8_ID) {                3
    thread_r1=FORK(S6Region1Exit);                      4
    thread_r2=FORK(S6Region2Exit);                      5
    JOIN(thread_r1);      JOIN(thread_r2);              6
    sendStopSignal(S6_ID);  exit_S6();                  7
    thread_t14=FORK(effect(t14));                       8
    thread_t15=FORK(effect(t15));                       9
    JOIN(thread_t14);  JOIN(thread_t15);                10
    effect_t16();                                       11
    activeStateID = STATE_MAX;  //inactive state        12
  }                                                     13
}                                                       14
```

Listing 5: Example code generated for completion events triggering transitions t14 and t15

```
if(activeRootState==S1_ID) {                            1
  junc = 0; //outgoing transition t9 of junc            2
  if (guard) {junc = 1;}                                3
  //Exit substates of S1 and S1                         4
  effect(t9);                                           5
  if(junc==0) {                                         6
    effect(t11);                                        7
  } else {                                              8
    effect(t10)                                         9
```

```
  }                                                          10
  FORK(effect(t12)); FORK(effect(t3));                       11
  //JOIN ... ==> concurrent execution                        12
  //Enter state S6, S7 and S8                                13
}                                                            14
```

Listing 6: Example code generated for $Fork1$ and $junc$

Generically, *GENTRANS* generates code for transitions between any vertexes satisfying the constraints described in Section 2. Table 2 shows how the transition code generation works. The generated code is bounded by the deferral events, active states, and guard checking.

In the first place, the procedure in Table 2 looks for the composite states s_{ex} and s_{en} at the highest level to be exited and entered (Step 1), respectively. If the transition t is part of a compound transition (we use the algorithm presented in [5, 16] to compute compound transitions), which involves some *junction*s, IF-ELSE statements for junctions are generated first (as PSSM says *junction* is evaluated before any action). The composite state is exited by calling the associated exiting region methods (FORK and JOIN for orthogonal regions) in Step 3 and followed by the generated code of transition effects (Step 4 and 5), respectively. If the parent state s_{en} of the target vertex v_t is a state (composite state), the associated entry is executed (Step 6). Entering region methods are then called once the above code completes its execution (Step 7). If the target v_t of the transition t is a pseudo state, the generation pattern corresponding to the pseudo-state types is called. These patterns are shown in Table 3.

Note that, the procedure in 2 only applies for external transitions. Due to space limitation, the detail of generating local and internal transitions is not discussed here but the only difference is the composite state containing the transitions is not exited.

5. EMPIRICAL STUDY

The approach is implemented as part of the Papyrus Designer tool [26] and an extension **PSM** of Papyrus [11]. This section reports our experiments with **PSM** on the semantic-conformance (Subsection 5.1) and efficiency (Subsection 5.2) of generated code.

5.1 Semantic conformance of runtime execution

Research question 1: *Does the runtime execution of code generated from USMs by* ***PSM*** *is semantic-conformant to PSSM?*

To evaluate the semantic conformance of runtime execution of generated code, we use a set of examples provided by Moka [1], which is a model execution engine offering PSSM. Fig. 3 shows our method. We first use our code generator to generate code (Step (1)) from the Moka example set. Step (2) simulates the examples by using Moka to extract the sequence (*Traces 1*) of observed traces including executed actions. The sequence (*Traces 2*) of traces is obtained by the runtime execution of the code generated from the same state machine in a Step (3). The generated code is semantic-conformant if the sequences of traces are the same for both of the simulation and generated code execution [6].

PSSM test suite consists of 66 test cases totally. Table 4 shows the test results for each state machine concepts provided by PSSM. The results are promising that our proto-

Table 3: Pseudo state code generation pattern

Pseudo state	Code generation pattern
join	Use $GENTRANS$ for v's outgoing transition (see Listing 5, lines 4-6).
fork	Use $FORK$ and $JOIN$ for each of outgoing transitions of v (see Listing 6, lines 11-12).
choice	For each outgoing, an $IF-ELSE$ is generated for the guard of the outgoing together with code generated by $GENTRANS$.
junction	As a static version *choice*, a *junction* is transformed into an attribute $junc_{attr}$ and evaluated before any action executed in compound transitions (see Listing 6, lines 2-3 and 6-10). The value of $junc_{attr}$ is then used to choose the appropriate transition at the place of *junction*.
shallow history	The identifiers of states to be exited are kept in *previousActives* of *IState*. Restoring the active states using the history is exampled as in Listing 3. The entering method is executed as default mode at the first time the corresponding composite state is entered (see Listing 3, lines 9-19). *previousActives* is updated with the active state identifier before exiting the region containing the history.
deep history	Saving and restoring active states are done at all state hierarchy levels from the composite state containing the deep history down to atomic states. Updating *pres* is committed before exiting the region, which is directly or indirectly contained by a parent state, in which a deep history is present.
entry point	If *enpoint* has no outgoing transition, the corresponding composite state is entered by default. Otherwise said, $GENTRANS$ is called to generate code for each outgoing transition.
exit point	The code for each transition outgoing from *expoint* is generated by using $GENTRANS$. If *expoint* has multiple incoming transitions from orthogonal regions, it is generated as a *join* to multiple-check the source states of these incomings.
terminate	The code executes the exit action of the innermost active state, the effect of the transition and destroys the state machine object.

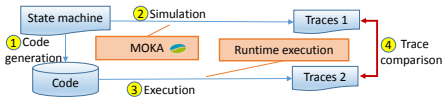

Figure 3: Semantic conformance evaluation methodology

type PSM passes 62/66 tests. In fact, our prototype PSM fails with some wired tests such as transitions outgoing from an entry point to an exit point. This is, as our observation, never used in practice. Furthermore, as the UML specification says that transitions outgoing from an entry point 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.

5.2 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.

Research question 2: *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 [8] in [2]. One simple example [10] only consists of atomic states and another [9] 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 [18], Enterprise Architect [29]) and QM [25], which generate code from state machines, and C++ libraries (Boost Statechart [17], Meta State Machine (MSM) [19], C++ 14 MSM-Lite [2], and functional programming like-EUML[3]) 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.

5.2.1 Speed

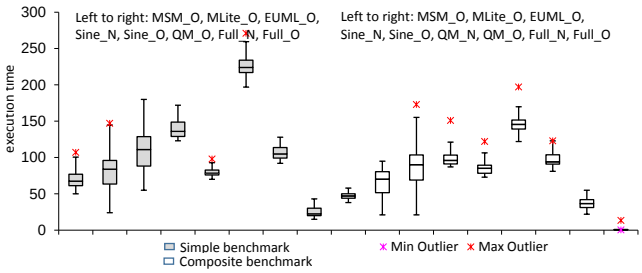

Figure 4: Event processing speed for the benchmark

Results: Table 5 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. 4 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.

5.2.2 Binary size and runtime memory consumption

Result: Table 6 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[4] to measure memory usage. Table 7 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).

6. 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 [12].

Switch/if is the most intuitive technique implementing a "flat" state machine. The latter can be implemented by either using a scalar variable [7] 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 [13] uses one dimension represents states and the

Table 4: Semantic-conformance test results (number of passed/total tests)

Behavior	Choice	Deferred Events	Entering	Exiting	Entry	Exit	Event	Final	Fork	Join	Transition	Terminate	Otl
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 5: 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

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 [13, 27] 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 [21] to support hierarchical-concurrent USMs. Recently, a double-dispatch (DD) pattern presented in [32] extends [21] to support maintainability by representing states and events as classes, and transitions as methods. However, as the results shown in [32], 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 [14, 30], apply different patterns to generate code. However, as mentioned in Section 1, true concurrency and some pseudo-states are not supported. FXU [24] 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 [21], 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.

7. 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] Moka Model Execution.

[2] State Machine Benchmark.

[3] State Machine Benchmark.

[4] Valgrind Massif.

[5] M. Balser, S. Bäumler, A. Knapp, W. Reif, and A. Thums. Interactive verification of uml state machines. In *International Conference on Formal Engineering Methods*, pages 434–448. Springer, 2004.

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

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

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

[9] Boost. 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].

[10] 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].

[11] CEA-List. Papyrus Homepage Website. https://eclipse.org/papyrus/.

[12] E. Domínguez, B. Pérez, A. L. Rubio, and M. A. Zapata. A systematic review of code generation proposals from state machine specifications, 2012.

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

[14] IBM. Ibm Rhapsody.

[15] IBM. IBM Rhapshody and UML differences. http://www-01.ibm.com/support/docview.wss?uid=swg27040251, 2016. [Online; accessed 04-July-2016].

[16] A. Knapp. Semantics of UML State Machines. 2004.

Table 6: 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 7: 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

[17] 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].

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

[19] 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].

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

[21] I. A. Niaz, J. Tanaka, and others. Mapping UML statecharts to java code. In *IASTED Conf. on Software Engineering*, pages 111–116, 2004.

[22] OMG. Precise Semantics Of UML Composite Structures. (October), 2015.

[23] OMG. Object Management Groups. http://www.omg.org/, 2016. [Online; accessed 04-July-2016].

[24] R. Pilitowski and A. Dereziñska. *Code Generation and Execution Framework for UML 2.0 Classes and State Machines*, pages 421–427. Springer Netherlands, Dordrecht, 2007.

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

[26] Qompass. Qompass. https://wiki.eclipse.org/Papyrus_Qompass, 2015. [Online; accessed 01-Sept-2015].

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

[28] SinelaboreRT. Sinelabore Manual. http://www.sinelabore.com/lib/exe/fetch.php?media=wiki:downloads:sinelaborert.pdf.

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

[30] SparxSystems. Enterprise Architect, Sept. 2016.

[31] O. M. G. Specification. UML specification. *Object Management Group pct/07-08-04*, 2007.

[32] V. Spinke. An object-oriented implementation of concurrent and hierarchical state machines. *Information and Software Technology*, 55(10):1726–1740, Oct. 2013.