Complete Code Generation from UML State Machine

Van Cam Pham, Ansgar Radermacher, Sébastien Gérard, Shuai Li
CEA LIST, Saclay, France
{*first-name*}.{*last-name*}*cea.fr*

Keywords: UML State Machine, code generation, semantics-conformance, efficiency, events, C++

Abstract: An event-driven architecture is a useful way to design and implement complex systems. The UML State Machine and its visualizations are a powerful means to the modeling of the logical behavior of such an architecture. In Model Driven Engineering, executable code can be automatically generated from state machines. However, existing generation approaches and tools from UML State Machines are still limited to simple cases, especially when considering concurrency and pseudo states such as history, junction, and event types. This paper provides a pattern and tool for complete and efficient code generation approach from UML State Machine. It extends IF-ELSE-SWITCH constructions of programming languages with concurrency support. The code generated with our approach has been executed with a set of state-machine examples that are part of a test-suite described in the recent OMG standard Precise Semantics Of State Machine. The traced execution results comply with the standard and are a good hint that the execution is semantically correct. The generated code is also efficient: it supports multi-thread-based concurrency, and the (static and dynamic) efficiency of generated code is improved compared to considered approaches.

1 INTRODUCTION

The UML State Machine (USM) (Specification and Bars, 2007) and its visualizations are efficient to model the behavior of event-driven architecture. Tools and approaches are proposed to automatically translate USMs into executable code in the context of Model-Driven Engineering (MDE) (Mussbacher et al., 2014).

However, despite many advantages of MDE and USM, they are not widely adopted as a recent survey revealed (Whittle et al., 2014). This is partially due to poor support for code generation (Forward et al., 2010).

On one hand, the usefulness and semantics of USM are being empowered by OMG by providing more concepts and their precise semantics such as pseudo states and composite state machines. On the other hand, existing code generation tools and approaches have some issues regarding completeness, semantics and efficiency of generated code. Existing approaches either support a subset of USM modeling concepts or handle composite state machines by flattening into simple ones with a combinatorial explosion of states, and excessive generated code (Badreddin et al., 2014a). Specifically, the following lists some of the current issues:

Completeness: Existing tools and approaches mainly focus on the sequential aspect while the concurrency of state machines is limitedly supported. Pseudo states are not rigorously supported by existing tools such as Rhapsody (IBM, 2016a). Designers are then restricted to a subset of USM concepts during design.

Efficiency: Code generated from tools such as Rhapsody (IBM, 2016b) and FXU (Pilitowski and Dereziñska, 2007) depends on the libraries provided by the tool vendor, which makes the generated code non portable. Event processing speed and executable file size of generated code are not optimized (Charfi et al., 2012).

Semantics: The semantics of UML State Machine is defined by a recent OMG-standardized: Precise Semantics of State Machine (PSSM) (OMG, 2016). This standard is not (yet) taken into account for validating the runtime execution semantics of generated code.

Given the above issues, the objective of this paper is to present a novel code generation pattern and its tooling support. The latter offers efficient code generated from USMs with full concepts to reduce the modeling-implementation gap.

The proposed pattern extends IF-ELSE constructions with our support for concurrency. Runtime execution of generated code is experimented with the

PSSM test suite.

To sum up, the contributions of this paper are: (1) an approach and tooling support for code generation from USMs with full features; (2) an empirical study on the semantic-conformance and efficiency of generated code; and (3) application of the tool to a case study.

We assume that readers of this paper have knowledge about UML State Machine and its basic execution semantics.

The remaining of this paper is organized as follows: Section 2 describes the modeling of applications using UML State Machines. Section 3 mentions the features of our tool. Thread-based concurrency is designed in Section 4. Based on this design, a code generation approach is proposed in Section 5. The implementation and empirical evaluation are reported in Section 6. The application of our tool to a case study is presented in Section 7. Section 8 discusses related work. The conclusion and future work are presented in Section 9.

2 STATE MACHINES AND UML EVENTS

This section presents overview of using UML State Machines for modeling and designing reactive software applications. A state machine is used for describing the behavior of either a class in object-oriented design or a component in component-based design. In the following, we commonly use the term *class*.

The state machine processes external and internal events. UML defines four event types: *CallEvent, SignalEvent, TimeEvent, ChangeEvent*. A call event is associated with an operation/method and emitted if the operation is invoked. The processing of call events is synchronous meaning that it runs within the thread of the operation caller. The processing of other events is asynchronous meaning that these events received by the class are stored in an event queue which is maintained by the class at runtime for later processing. A signal event is associated with a UML signal type containing data. It is emitted if the class receives an instance of the signal type. From a programming perspective, we provide an API *sendSignal* to send the signal instance from environment code or other classes to the class and store the event in the queue.

A time event specifies the time of occurrence relative to a starting time. The latter is defined as the time when a state with an outgoing transition triggered by the time event is entered. The time event is emitted if this accepting state remains active longer that the relative time of occurrence. Once emitted, it triggers the transition. In other words, the state, which is the source vertex of a transition triggered by a time event, will remain active for a maximal amount of time specified by the time event. A change event has a boolean expression and is fired if the expression's value changes from false to true. Note that unlike call and signal events, time and change events are automatically fired inside the class.

Deferred events: A state can specify to defer some events. It means that if an event specified as deferred, it will be not processed while the state remains active. The deference of events is used to postpone the processing of some low-priority events while the state machine is in a certain state.

We support all of these events to model event-driven reactive applications.

3 FEATURES

Our pattern and tool has some features compared to other tools as followings:

Completeness: Our tool supports all state machine vertexes and transitions including all pseudo states and transition kinds such as external, local, and internal. Hence, the tool improves flexibility of using UML State Machines to express architecture behavior. For the moment, our tool cannot deal with transitions from an *entry point* to an *exit point*. We believe that these transitions are not used in reality. This is because the contradictory semantics of *entry points* and *exit points*. In UML, *entry points* and *exit points* represent entering points and exit points of a compoiste state, respectively. They provide encapsulation of the insides of the state. The *entry points* allow users to customize the way to enter the composite state instead of the default entering way while the *exit points* allow to customize the exiting way. For example, the *Enp* entry point in Fig. 1 allows the *S5* sub-state of the *S1* composite state to be active instead of *S3* by the default entering way.

Event support: Our tool promotes four UML event types and event deference mechanism, which are able to express synchronous and asynchronous behaviors and exchange data between components/classes.

UML-conformance: A recent specification formalizing the Precise Semantics of UML State Machine (PSSM) is under standardization of the OMG. It defines a test suite with 66 test cases for validating the conformance of runtime execution of code generated from UML State Machines. We have experimented our tool with the test suite. Traced execution results of 62/66 test cases comply with the standard and are, therefore, a good hint that the execution is semantically correct.

State machine configuration: Asynchronous events

such as signal events, change events, and time events are stored in an event queue. A signal event can bring data (message). Our tool allows to configure the event queue size and the maximal size of signals. The configuration is not specified by UML because the specification wants to be abstract. We allow to determine these values through a specific profile. Note that the configuration information might not be needed in dynamic memory allocation. The latter, however, is not recommended in embedded systems.

Efficiency: We conducted experiments on some benchmarks to show that code generated by our tool is efficient and can be used to develop resource-constrained embedded software. Specifically, event processing is fast and the size of executable files compiled from generated code is small.

Event API: Generated code in our tool provides APIs for environment code to invoke operations or send data signals to reactive classes. The invocations and sending will automatically fire events for state machines to process.

Concurrency: Concurrency aspects in state machines including doActivity of states, orthogonal regions, event detection, and event queue management are handled by the execution of multiple threads. Currently, we use POSIX threads for concurrency.

Portability: Currently, our tool generates C++ code. The generated code can run on POSIX systems such as Ubuntu without installing any additional libraries to be able to compile and execute the code. Our code generation pattern and tool can be extended to generate code in other programming languages such as Java which supports threads and mutexes for multi-thread synchronization.

4 CONCURRENCY

This section describes our design of concurrency aspects of state machines in generated code at runtime.

4.1 Thread-based design

The concurrency of USMs is based on multiple threads including 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 time event which counts ticks and emits the event once completes.
- Change detect function associated with a change event 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.

STs which are spawned by a parent thread, joined until and destroyed once the associated methods complete. The STs follow a paradigm in which the spawning parent must wait until its children complete their associated methods. These threads are used for 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, after the entry action of the state, a thread is created for each orthogonal region.
- Exiting a concurrent state, before the exit action of the state, a thread is created for each region to exit the corresponding active sub-state.

4.2 Thread communication

Each PT is associated with a mutex for synchronization in the multi-thread-based generated code. The mutex must be locked before the method associated with the thread is executed.

Run-to-completion: The event process must follow the run-to-completion semantics of UML State Machines. The semantics means that the state machine completes processing of each event before starting processing the next event. If all events are asynchronous, the main thread processes events by reading one-by-one from the event queue. However, because we allow call events to be synchronous, the processing of synchronous and asynchronous events can violate the run-to-completion semantics. To avoid it, a main mutex is associated with the main thread to protect the run-to-completion semantics. Each event processing must lock the main mutex before executing the actual processing. In generated code, lock and unlock are implemented using signals and conditions in POSIX (Butenhof, 1997).

5 CODE GENERATION PATTERN

This section describes our code generation pattern for states, regions, events, and transitions.

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

```
typedef struct IState {
  int previousActives[2];  int actives[2];
} IState;
class C {
private:
  IState states[STATE_MAX];
public:
  void entry(StateId id) {
    switch{id} {
      case S0_ID:
        //action code for each state
        break;
      //code for other state actions
    }
  }
}
```

Listing 2: Example code generated for doActivity

```
while(true) {
  pthread_mutex_lock(&mutex[stateId]);
  while(!isStarts[stateId]) {
    //await start signal
    pthread_cond_wait(&cond, &mutex[stateId];}
  doActivity(stateId);
  isStarts[stateId] = false;//reset wait flag
  pthread_mutex_unlock(&mutex[stateId]);
  if (!isStops[stateId]) {
    if(stateId==S0_ID||...){//atomic states
      pushCompletionEvent(stateId);
    }
  }
}
```

5.1 State

A common state type *IState* is created. The typ has two attributes called *actives*, to preserve the hie archy of composite states, and *previousActives* refe ring to current and previous active sub-states in case of the presence of history states. Each UML state is transformed into an instance of **IState** and a state ID is assigned (which is a child element of an enumeration). During initialization, each instance initializes its attributes to a default value meaning inactive state.

In the following sections, we only consider C++ as a specific generated language. The discussion of other object-oriented languages is much similar since these share the same concepts.

Listing 1 shows the state type and its instances. *STATE_MAX* is the number of states. The state actions such as entry/exit/doActivity are generated to corresponding common methods containing action codes. For example, *entry* in the listing implements all of the state action codes.

State *doActivity*s, as specified by UML, are run concurrently. Each *doActivity* is then run within a permanent thread and a mutex is created for controlling it. Listing 2 shows a code segment for *doActivity* threads. The method *doActivityThread* takes as input a state id to use and call the appropriate mutex and *doActivity*, respectively. The method does nothing and stays in a waiting point if the state corresponding to the input parameter state identifier is inactive (line 5). If the state is active, a start signal is sent to this thread method to start the execution of *doActivity*. The generated code typically follows the common paradigm in POSIX threads (Butenhof, 1997).

5.2 Region

Our approach considers regions as elements to be transformed. Specifically, each region has two methods: entering and exiting. The entering method controls how a region *r* is entered from an outside transition and the exiting method exits completely a region by executing exit actions of sub-states from innermost to outermost.

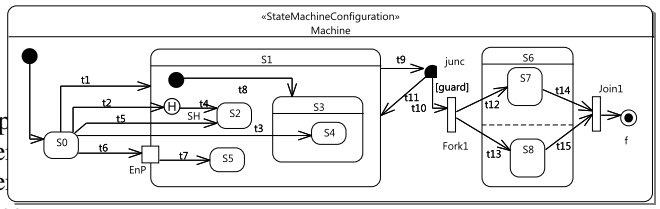

Figure 1: Example illustrating different ways entering a composite state

A region can be entered two different ways: (1) **entering by default**: the transition ends at the border of composite states; and (2) **cross transition**: entering at a direct or an indirect sub-vertex of composite states. The two entering ways execute the entry action of the containing composite state after the transition effect. The executions afterwards are different for each way. To illustrate, we use an example as in Fig. 1 with *S1* as a target composite state. *t1* is in the way (1) while *t2, t5, t6* in the way 2.

The entering method associated with the region of *S1* has a parameter *enter_mode* telling how the entering should be executed. *enter_mode* takes values depending the number of transitions coming to the composite state. The detail of how these modes are implemented in specific languages are not discussed here. Listing 3 shows the generated C++.

By default, the region's active sub-state is set after the execution of any effect associated with the initial transition. Therefore, *S*3 is set as active sub-state of *S*1. Entering at (*S2*) sets the active sub-state of *S1* directly to *S2*. In case of an indirect sub-state (*S*4), the entry action of *S*3 is executed before *S*4 is set as the active-sub state of *S*3 and the entry execution of *S*4. It is worth noting that after the execution of each entry action, a start signal is sent to activate the waiting thread associated with *doActivity* of the corresponding state.

Listing 3: Example code generated for the region of S1

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

Transitioning from a vertex to a sub-vertex of the composite state (transition from *S*0 to *SH* is a particular case) is not as simple as that of two states. This is detailed in the next section.

The method generated for exiting a region is simpler than that of entering. It basically executes the exit actions of all the active sub-states from innermost to outermost.

5.3 Event

Similar to the approach in (Niaz et al., 2004), one method is generated for each event. An event enumeration *EventId* is created whose children are event identifiers associated with events. The event list of a state machine contains explicitly defined events and a special event called completion event, which is implicitly implemented. A completion event is fired when either the execution of the *doActivity* of simple/atomic state completes or all regions of a composite state have reached final states. For each event type, the pattern is realized as followings:

CallEvent: When its associated operation is called, the event processing waits and locks the main mutex protecting the run-to-completion semantics as previously mentioned, and executes the event processing (see 4.2).

SignalEvent: An API *sendSignal* is created for environment code to interact and send an instance of the signal associated with the event by calling it. When the API is called, an event is emitted and written into the event queue.

TimeEvent: A thread associated with the event is created and initialized at the initialization. Within the thread execution, its associated method waits for a signal, which is sent after the execution of the entry of an accepting state, to start sleeping for a duration specified by the event. When the relative time expires, the event is emitted and written to the event queue if the state is still active.

ChangeEvent: Similarly to time events, a thread is initialized and its method waits for a starting signal. The method checks whether the value of the boolean expression of the event is updated from false to true. If so, the event is committed to the event queue. The expression is expressed by attributes of the class owning the state machine. The starting signal is sent if one of the expression's constituents (attributes of the class) changes. We track the changes of the attributes' values by using setters of the attributes. For example, for an expression $x+y>10$, x and y are extracted as constituents. The setters (*setX* and *setY*) are automatically generated. They do not only affect the value of x and y but also send the starting signal to the thread.

As above presented, all asynchronous incoming events are stored in a runtime priority queue, in which each event type has a priority. Completion event always has the highest priority. Others are equal by default. Event type, priority, identifier, associated state *stateId* of completion events, and signal data are specified in an internal structure. The associated state is responsible to specify which atomic/simple state completes its *doActivity* execution or the composite state whose sub-states have reached final states.

5.4 Transitions

Each event triggers a list of transitions. We suppose $T_{trig}(e)$ is the transition list triggered by the event e, and $S_{trig}(e)$ is a depth-ordered (from innermost to outermost) set of the source states of the transitions in $T_{trig}(e)$.

Algorithm 1 describes how to generate the body of an event method. It first finds the innermost active states which are able to react e by orderly looping over $S_{trig}(e)$. This is to ensure that, in case of multiple transitions triggered by the event, the generated code for the transitions outgoing from innermost states will be executed. For each transition from an innermost state, code for active states and deferred events, guard checking, and transition code segments are generated by *GEN_CHECK*, *GEN_GUARD*(t) and *GEN_TRANS*, respectively. If the identifier of e is equal to one of the deferred event list of the corresponding state (not shown in this paper), *GEN_CHECK* generates code, which checks whether the event to be deferred and pushes the event to a deferred event queue managed by the runtime main thread. The latter also pushes the deferred

Listing 4: Example code generated for completion events triggering transitions t14 and t15

```
if(event.stateId==S6_ID||event.stateId==S7_ID){
  if (states[S6_ID].actives[0] == S7_ID &&
    states[S6_ID].actives[1] == S8_ID) {
    thread_r1=FORK(S6Region1Exit);
    thread_r2=FORK(S6Region2Exit);
    JOIN(thread_r1);      JOIN(thread_r2);
    sendStopSignal(S6_ID);  exit_S6();
    thread_t14=FORK(effect(t14));
    thread_t15=FORK(effect(t15));
    JOIN(thread_t14);  JOIN(thread_t15);
    effect_t16();
    activeStateID = STATE_MAX;  //inactive state
  }
}
```

events back to the main queue once one of the pending events is processed and the active state is changed.

Algorithm 1 Code generation for events

Require: Event e
Ensure: Code generation process for event method

procedure EVENTGENPROCESS(e)
  for $\forall$ s $\in S_{trig}(e)$ **do**
    $T_s = \{t \in T_{trig}(e) | src(t) = s\}$
    for $\forall t \in T_s$ **do**
      GEN_CHECK(s,t,e)
      GEN_GUARD(t)
      GEN_TRANS$(s,t,tgt(t))$

For a transition t, *GEN_CHECK* can generate single or multiple active state checking code. The latter occurs if the target of the transition is a pseudo state join because the transitions incoming to a *join* are fired if and only if all of their source states are active. The detailed discussion on these is not presented due to space limitation. Listing 4, lines 2-3 show a portion of the code with multiple checking generated for the completion event processing method. The transitions *t14* and *t15* incoming to *Join*1 are executed if *S6* and *S7* are active. In addition, the code portion checks the state associated with the current completion event emitted upon the completion of either *S6*'s or *S7*'s *doActivity*. In lines 4-6, the code concurrently exits the sub-states of *S6* by using *FORK* and *JOIN*, which are respectively used to spawn and wait for a thread, for the region methods associated with *S6*'s orthogonal regions, which actually exit *S7* and *S8*. Then, *exit(S6)* is executed before the concurrency of transition effects *t14* and *t15* is taken into account.

GEN_TRANS is able to generate code for transitions between two vertexes. Algorithm 2 shows how it works. The generated code is contained by the deferral events, active states, and guard checking.

Firstly, Algorithm 2 looks for the s_{ex} and s_{en} vertexes contained in the same region and respectively containing the source and target vertexes of the transition t. For example, s_{ex} and s_{en} in case of the $t3$ transition are *S*0 and *S*1 contained by the top region. If the transition t is part of a compound transition (we use the algorithm presented in (Balser et al., 2004; Knapp, 2004) 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 lines 4-9 and followed by the generated code of transition effects (lines 10-15). If the parent state s_{en} of the target vertex v_t is a state (composite state), the associated entry is executed (lines 16-18). Entering region methods are then called once the above code completes its execution (lines 19-24). 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 1.

Note that, the procedure in 2 only applies for ex-

Algorithm 2 Code generation for transition

Require: A source v_s, a target vertex v_t and a transition t
Ensure: Code generation for transition

1: **procedure** GEN_TRANS(v_s, v_t, t)
2: Find s_{ex} and s_{en} as vertexes in the same region and directly or indirectly containing/being v_s and v_t, respectively.
3: Generate IF-ELSE statements for junctions
4: **if** s_{ex} is a state **then**
5: **for** $r \in$ regions of s_{ex} **do**
6: $FORK(RegionExit(r))$ //create thread for exiting region
7: Generate JOIN for threads created above
8: Generate sendStopSignal to s_{ex}
9: $exit(s_{ex})$ //exit the state
10: **if** v_t is a pseudo state join **then**
11: **for** $in \in$ incoming transitions of v_t **do**
12: $FORK(effect(in))$ //create thread for transition effect
13: Generate JOIN for threads created above
14: **else**
15: $effect(t)$ //execute transition effect
16: **if** s_{en} is a state **then**
17: $entry(s_{en})$ //state entry
18: Generate sendStartSignal to s_{en}
19: **if** s_{en} is a composite state **then**
20: **for** $r \in$ regions of s_{en} **do**
21: $FORK(RegionEnter(r))$ //create thread for entering region
22: Generate JOIN for threads created above
23: **else**
24: Generate for pseudo states by patterns

Listing 5: Example code generated for *Fork*1 and *junc*

```
if(activeRootState==S1_ID) {
  junc = 0; //outgoing transition t9 of junc
  if (guard) {junc = 1;}
  //Exit substates of S1 and S1
  effect(t9);
  if(junc==0) {
    effect(t11);
  } else {
    effect(t10)
  }
  FORK(effect(t12)); FORK(effect(t3));
  //JOIN ... ==> concurrent execution
  //Enter state S6, S7 and S8
}
```

Table 1: Pseudo state code generation pattern

Pseudo state	Code generation pattern
join	Use *GEN_TRANS* for *v*'s outgoing transition (Listing 4, lines 4-6).
fork	Use *FORK* and *JOIN* for each of outgoing transitions of *v* (see Listing 5, lines 11-12).
choice	For each outgoing, an $IF-ELSE$ is generated for the guard of the outgoing together with code generated by *GEN_TRANS*.
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 5, 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 composite state is entered (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 *previousActives* 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 an *entry point* has no outgoing transition, the composite state is entered by default. Otherwise said, *GEN_TRANS* is called to generate code for each outgoing transition.
exit point	The code for each transition outgoing from an *exit point* is generated by using *GEN_TRANS*. If the *exit point* 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.

ternal transitions. Due to space limitation, the detail of generating local and internal transitions is not discussed here but the only difference is that the composite state containing the transitions is not exited.

6 EMPIRICAL STUDY

The pattern is implemented in Papyrus Designer (LISE,), which is an extension of the UML modeling tool Papyrus (Gérard et al., 2010). Papyrus Designer supports component-based modeling and code generation. The behavior of a component in Papyrus Designer is described by using UML State Machines. The tool allows to use some time notions from the MARTE profile to specify time events. C++ code is generated and runs within POSIX systems such as Ubuntu, in which Pthreads are used for implementing threads for concurrency. This section reports our experiments with Papyrus Designer on the semantic-conformance and efficiency of generated code.

6.1 Semantic conformance of runtime execution

This section presents our results found during experiments with our tool to answer the following research question.

> ***Research question 1:*** *Is the runtime execution of code generated from USMs by our tool semantic-conformant to PSSM?*

To evaluate the semantic conformance of runtime execution of generated code, we use a set of examples provided by Moka (Papyrus, 2016), which is a model execution engine offering PSSM (and also part of the Papyrus modeler). Fig. 2 shows our method. The latter consists of the following steps:

Step 1 For a **State machine** from the Moka example set, we use our code generation tool to generate code.

Step 2 We simulate the execution of the **State machine** by using Moka to extract a sequence **Trace 1** of observed traces including executed actions.

Step 3 The sequence (**Traces 2**) is obtained through the runtime execution of the code generated in Step 1.

Step 5 *Trace 1* and *Trace 2* are compared. The code is semantic-conformant if **Traces 1** and **Traces 2** are the same (Blech and Glesner, 2005).

The PSSM test suite consists of 66 test cases for different state macchine element types. The results are promising: our tool passes 62/66 tests including: behavior (5/6), choice (3/3), deferred events (6/6), entering (5/5), exiting (4/5), entry(5/5), exit (3/3), event (9/9), final state (1/1), fork (2/2), join (2/2), transition (11/14), terminate (3/3), others (2/2). In fact, our tool fails with some tests containing transitions (1) from an *entry point* to an *exit point* or (2) from an entry point/exit point to itself. This is, as our observation, rarely used in practice because of the contradictory semantics of *entry points* and *exit points* as previously discussed.

The results of this evaluation are not enough to prove that our pattern and tooling support preserves the UML State Machine execution properties but are a good hint that runtime execution of generated code is semantically correct (for all but the case identified above).

This evaluation methodology has the limitation that it is dependent on PSSM. Currently, for event support, PSSM only specifies signal events. For pseudo-states, histories are not supported. Thus, our evaluation result is limited to the current specification of PSSM.

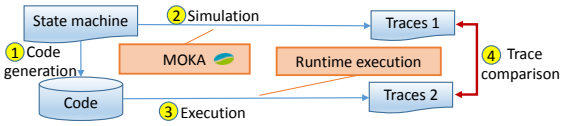

Figure 2: Semantic conformance evaluation methodology

Threats to validity: Operation behaviors in PSSM are defined by activities while our prototype requires fine-grained behavior as blocks of code embedded into models. Therefore, an internal threat is that we manually re-create these tests and convert activities into programming language code.

6.2 Benchmarks

In this section, we present the results obtained through the experiments on some efficiency aspects of generated code to answer the following question.

> ***Research question 2:*** *Runtime performance and memory usage are undoubtedly critical in real-time and embedded systems. Particularly, in event-driven systems, the performance is measured by event processing speed. Are the performance and memory usage of code generated by our tool comparable to existing approaches?*

Two state machine examples are obtained by the preferred benchmark used by the Boost C++ libraries (Boost Library, 2016a) in (Jusiak, 2016). One simple example only consists of atomic states and the other both atomic and composite states.

We compared our tool with tools such as Sinelabore (which generates efficient code for Magic Draw (Magic, 2016), Enterprise Architect (SparxSysems, 2016)), Quantum Modeling (QM) (Quantum Leaps, 2016) (which generates code for event-driven active object frameworks (Lavender and Schmidt, 1996)), Boost Statechart (Boost Library, 2016d), Meta State Machine (MSM) (Boost Library, 2016b), C++ 14 MSM-Lite (Jusiak, 2016), and functional programming like-EUML(Boost Library, 2016c).

We used a Ubuntu virtual machine 64 bit hosted by a Windows 7 machine. For each tool, we created two applications corresponding to the two examples, generated C++ code and compiled it in two modes: normal (N), by default GCC compiler; and optimal (O) with GCC optimization 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.

6.2.1 Performance

Fig. 3 shows the event processing performance of the approaches for the two benchmarks. In the normal compilation mode (postfix N), Boost Statechart, MSM, MSMLite, EUML are quite slow and not displayed in the box-plot.

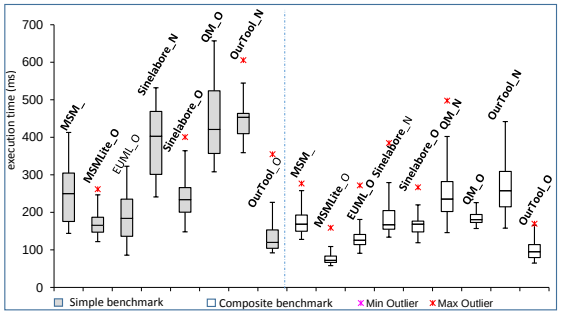

Figure 3: Event processing speed for the benchmarks

In both of the simple and composite benchmarks, in optimization mode (postfix O) MSMLite and our tool run faster than the others in the scope of the experiment. The figure also shows that the optimization of GCC is significant. In normal mode only the performance of Sinelabore, QM, and our tool is acceptable. The event processing speed of MSM, MSM_Lite and EUML is too slow without GCC optimizations.

6.2.2 Memory usage

Table 2 shows the executable size for the examples compiled in two modes. Without optimization, Sinelabore generates the smallest executable size while our approach takes the second place. In GCC optimization mode, MSMLite, Sinelabore and our approach require less static memory than the others.

Let's look closer at the event processing performance in optimization mode in terms of time medians. Fig. 4 shows the figures of the two benchmarks, relative to the performance of Sinelabore (normalized to 100%). For the simple (blue) benchmark, our approach (51.3%) is the fastest. For the composite (red) benchmark, with the support of C++14, the performance in MSMLite (42.7%) is the fastest and ours is the second.

For runtime memory consumption, we use the Valgrind Massif profiler (Valgrind, 2016; Nethercote and Seward, 2007) to measure memory usage. Table 3 shows the memory consumption measurements including stack and heap usage for the composite example. Compared to others, code generated by our approach requires a slight overhead with regard to runtime memory usage (0.35KB). 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 (0.35KB).

Table 2: Executable size in KB

Test	MSM		MSM-Lite		EUML		Sinelabore		QM		Our tool	
	N	O	N	O	N	O	N	O	N	O	N	O
Simple	414,6	22,9	107,3	10,6	2339	67,9	16,5	10,6	22,6	16,6	21,5	10,6
Composite	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 3: Runtime memory consumption in KB. Columns from left to right are SC, MSM, MSM-Lite, EUML, Sinelabore, QM, and Our tool, respectively.

76.03	75.5	75.8	75.5	75.8	75.7	76.38

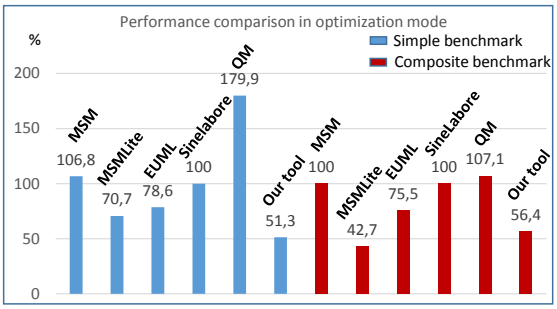

Figure 4: Event processing performance in optimization mode

7 TRAFFIC LIGHT CONTROLLER SIMULATION

In order to assess the usability and practicality of using UML State Machines and events, we applied our tool to a simplified Traffic Light Controller (TLC) system as a case study, which is extracted from (Katz and Borriello, 2005).

TLC controls an intersection of a busy highway and a little-used farm-way as in Fig. 5. Detectors are placed along a farmroad to raise the signal *C* as long as a vehicle is waiting to cross the highway. The highway lights remains green as long as no vehicle is detected on the farmroad. Otherwise, the highway lights should change from yellow to red, allowing the farmroad lights to become green. The farmroad lights stay green only as long as a vehicle is detected on the farmroad and never longer than a set interval to allow the traffic to flow along the highway. If no vehicle or timeout expired, the farmroad lights change from green to yellow to red, allowing the highway lights to return to green. Even if vehicles are waiting to cross the highway, the highway should remain green for a set interval.

The object-oriented class diagram follows the design in Yasmine (Yasmine, 2016), which is a C++11 state machine framework, and is shown in Fig. 5 (right). The behavior of each class is described by a state machine. The state machines of *Intersection* and *TrafficLight* are shown in Fig. 6 (left and right, respectively). All of the states of *IntersectionStateMachine*, except *FarmwayOpen*, are composite. The details of *SwitchingHighwayToFarmroad* and *SwitchingFarmroadToHighway* are actually shown on the yasmine site (Yasmine, 2016).

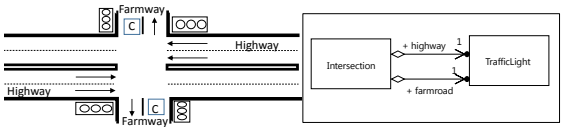

Figure 5: Traffic Light Controller (left) and its class diagram (right).

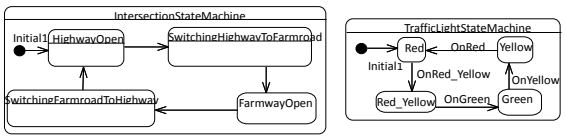

Figure 6: State machines for describing the behavior of Intersection (left) and TrafficLight (right)

The conditions for switching from the state *HighwayOpen* to *SwitchingHighwayToFarmroad* are: (1) a minimum time for the highway open is elapsed; and (2) the sensors emit a signal.

To show the usability and practicality of UML events, two alternative designs can be specified by using time events and change events. Fig. 7 (a) and (b) show the alternates, respectively. The first design in 7 (a) uses a time event, which triggers the transition from *WaitingForHighwayMinimum* to *MinimumTimeElapsed*, and a signal event deferred by the *WaitingForHighwayMinimum* state. When *HighwayOpen* becomes active, its active sub-state remains *WaitingForHighwayMinimum* as long as the minimum time. If a signal C is fired from the detector, a signal event *DetectorOn* is sent to the state machine. The event is, however, not immediately processed but delayed by until the active sub-state becomes *MinimumTimeElapsed* in case the time event is fired. The signal event is then processed to finish the execution of *HighwayOpen* and activate the farmway.

The other design utilizes a change event instead of deferred events for switching from *WaitForPreconditions* to a final state. Two flags *timeFlag* and *detectFlag* are used. The *WaitForPreconditions* state has two internal transitions. One is triggered by a signal event associated with the signal C and calls a transition effect to update *detectFlag* to true. The other one triggered by a time event sets *timeFlag* to true. The

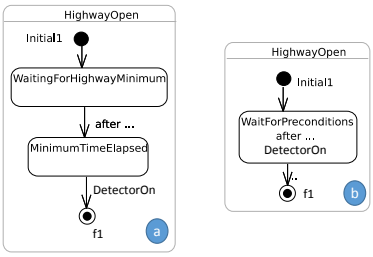

Figure 7: Alternative state machine designs for the *HighwayOpen* state

expression associated with the change event updates from false to true once two flags *timeFlag* and *detectFlag* are set to true. The periodic evaluation time is configured as 10ms.

For simulation of TLC, we reuse the detector class developed in (Yasmine, 2016) to automatically generate *DetectorOn/DetectorOff* signals.

The support of UML events (change events and time events) and deferred events does not only provide designers more options to specify but also simplify system behaviors. It can also reduce the number of states. For example, the numbers of sub-states of *HighwayOpen* with the use of deferred events and change events are two and one, respectively, while Yasmine requires three states. However, deferred events might make the design more difficult to understand because of its specialized semantics.

8 RELATED WORK

Code generation from state machines has received a lot of attention in automated software development. This section mentions some existing code generation patterns and how our approach differs. A systematic review of several proposals is presented in (Domínguez et al., 2012).

Switch/if is the most intuitive technique for implementing a ”flat” state machine. It either uses a scalar variable (Booch et al., 1998) and a method for each event, or using two variables as the 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 state table approach (Douglass, 1999) uses one dimension for representing states and the other one for all possible events. These approaches require a transformation from hierarchical to flatten state machines. However, these approaches are hardly applied to state machines containing pseudo states such as deep history or join/-fork.

The object-oriented state pattern (Shalyto and Shamgunov, 2006; Douglass, 1999) transforms a state into a class and an event into a method. Events are processed by delegating from the class containing the state machine to its sub-state classes. Separation of states in classes makes the code more readable and maintainable. Unfortunately, this technique only supports flat state machines. This pattern is extended in (Niaz et al., 2004) to support hierarchical state machines. Recently, a double-dispatch (DD) pattern presented in (Spinke, 2013) extends (Niaz et al., 2004) to support maintainability by representing states and events as classes, and transitions as methods. However, as the results shown in (Spinke, 2013), these patterns require much memory because of an explosion of the number of classes and use dynamic memory allocation, which is not preferred in embedded systems. It is worth noting that none of these approaches provides implementation for all of state machine pseudo states as well as events.

Tools such as (SparxSystems, 2016; IBM, 2016b) apply different patterns to generate code. However, as mentioned in Section 1, true concurrency, some pseudo-states, and UML events are not supported. FXU (Pilitowski and Dereziñska, 2007) is the most complete tool but generated code is heavily dependent on their own library and C# is generated.

Umple (Badreddin et al., 2014b) is a textual UML programming language, which supports code generation for different languages such as C++ and Java from state machines. However, Umple does not support pseudo states such as fork, join, junction, and deep history, and local transitions. Furthermore, only call events and time events are specified in Umple.

Our approach combines the classical switch/if pattern, to produce small footprint, and the pattern in (Niaz et al., 2004), 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.

9 CONCLUSION

We presented an approach whose objective is to provide a complete, efficient, and UML-compliant code generation from UML State Machines with full features. The design for concurrency of generated code is based on multi-thread of POSIX. The code generation pattern extends the IF-ELSE/SWITCH patterns and uses a hierarchical structure to preserve the state machine hierarchy.

We implemented our pattern as part of the Papyrus modeling tool. We evaluated our tool by conducting experiments on the semantic-conformance and efficiency of generated code. The conformance is tested under PSSM: 62 of 66 tests passed. These results are a good hint that our tool preserves the UML State

Machine semantics during code generation. For efficiency, we used the benchmark defined by the Boost library to compare code generated by our tool to other approaches. The results showed that our tool produces efficient code that runs fast in event processing speed and is small in executable size.

Code produced by our tool, however, consumes slightly more memory than that of the others at runtime. In future work, we will fix this issue by making multi-thread part of generated code more concise. Furthermore, we will use the pattern to support Java code generation from UML State Machines.

REFERENCES

Badreddin, O., Lethbridge, T. C., Forward, A., Elaasar, M., Aljamaan, H., and Garzon, M. A. (2014a). Enhanced code generation from uml composite state machines. In *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*, pages 235–245. IEEE.

Badreddin, O., Lethbridge, T. C., Forward, A., Elasaar, M., and Aljamaan, H. (2014b). Enhanced Code Generation from UML Composite State Machines. *Modelsward 2014*, pages 1–11.

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

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

Booch, G., Rumbaugh, J., and Jacobson, I. (1998). *The Unified Modeling Language User Guide*, volume 3.

Boost Library (2016a). Boost C++. http://www.boost.org/. [Online; accessed 04-July-2016].

Boost Library (2016b). Meta State Machine. `http://www.boost.org/doc/libs/1_59_0_b1/libs/msm/doc/HTML/index.html`. [Online; accessed 04-July-2016].

Boost Library (2016c). State Machine Benchmark. `http://www.boost.org/doc/libs/1_61_0/libs/msm/doc/HTML/ch03s04.html`.

Boost Library (2016d). The Boost Statechart Library. [Online; accessed 04-July-2016].

Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley Professional.

Charfi, A., Mraidha, C., and Boulet, P. (2012). An optimized compilation of uml state machines. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 172–179.

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

Douglass, B. P. (1999). *Real-time UML : developing efficient objects for embedded systems*.

Forward, A., Lethbridge, T. C., and Badreddin, O. (2010). Perceptions of software modeling: A survey of software practitioners. In *in 5th Workshop from Code Centric to Model Centric: Evaluating the Effectiveness of MDD (C2M: EEMDD), 2010. Available: http://www. esi. es/-modelplex/c2m/papers. php*. Citeseer.

Gérard, S., Dumoulin, C., Tessier, P., and Selic, B. (2010). 19 papyrus: A uml2 tool for domain-specific language modeling. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 361–368. Springer.

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

IBM (2016b). Ibm Rhapsody. [Online; accessed 04-July-2016].

Jusiak, K. (2016). State Machine Benchmark. `https://github.com/boost-experimental`. [Online; accessed 20-Oct-2016].

Katz, R. H. and Borriello, G. (2005). Contemporary logic design.

Knapp, A. (2004). Semantics of UML State Machines.

Lavender, R. G. and Schmidt, D. C. (1996). Active Object. *Context*, pages 1–12.

LISE. Papyrus Software Designer. `https://wiki.eclipse.org/Papyrus_Software_Designer`.

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

Mussbacher, G., Amyot, D., Breu, R., Bruel, J.-m., Cheng, B. H. C., Collet, P., Combemale, B., France, R. B., Heldal, R., Hill, J., Kienzle, J., and Schöttle, M. (2014). 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.

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

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

OMG (2016). Precise Semantics of UML State Machines (PSSM) Revised Submission. [Revised Submission, ad/16-11-01].

Papyrus (2016). Moka Model Execution. `https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`. [Online; accessed 01-Nov-2016].

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

Quantum Leaps (2016). Quantum Modeling. http://www.state-machine.com/qm/. [Online; accessed 14-May-2016].

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

SparxSysems (2016). Enterprise Architect. http://www.sparxsystems.com/products/ea/. [Online; accessed 14-Mar-2016].

SparxSystems (2016). Enterprise Architect. http://www.sparxsystems.eu/start/home/. [Online; accessed 20-Nov-2016].

Specification, O. M. G. A. and Bars, C. (2007). OMG Unified Modeling Language (OMG UML). *Language*, (November):1 – 212.

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

Valgrind (2016). Valgrind Massif. http://valgrind.org/docs/manual/ms-manual.html. [Online; accessed 20-Nov-2016].

Whittle, J., Hutchinson, J., and Rouncefield, M. (2014). Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161.

Yasmine (2016). The classic farmroad example. http://yasmine.seadex.de/yasmine.html. [Online; accessed 20-Nov-2016].