**Complete Code Generation from UML State Machine**

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 archi- tecture. In Model Driven Engineering, executable code can be automatically generated from state machines. However, existing generation approaches 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 Ma- chine. 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.

# 1INTRODUCTION

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 [XX] are proposed to auto- matically translate USMs into executable code in the context of Model-Driven Engineering (MDE) (Muss- bacher et al., 2014).

However, despite many advantages of MDE and USMs, 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, ).

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 ap- proaches 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 flat- tening into simple ones with a combinatorial explo- sion of states, and excessive generated code (Badred- din 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, 2016). Designers are then restricted to a subset of USM concepts during design.

**Efficiency:** Code generated from tools such as Rhap- sody (IBM, ) and FXU (Pilitowski and Derezin˜ska, 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, 2015). This standard is not (yet) taken into account for val- idating 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 gen- erated from USMs with full concepts to reduce the modeling-implementation gap.

The proposed pattern extends IF-ELSE construc- tions with our support for concurrency. Runtime ex- ecution 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 gen- erated code; and (3) application of the tool to a case study.

We assume that readers of this paper have knowl- edge about UML State Machine and its basic execu- tion semantics.

The remaining of this paper is organized as fol-

lows: Section 2 describes the modeling of applica- tions 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 Section 5. The im- plementation 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.

# 2STATE 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 re- ceived by the class are stored in an event queue which is maintained by the class at runtime for later process- ing. 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 rela- tive 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 emit- ted if this accepting state remains active longer that the relative time of occurrence. Once emitted, it trig- gers 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 auto- matically 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 ac-

tive. 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.

# 3FEATURES

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 in- ternal. Hence, the tool improves flexibility of using UML State Machines to express architecture behav- ior. For the moment, our tool cannot deal with transi- tions 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, an *entry points* and *exit points* represent entering points and exit points of a compoiste state, respectively. They provides encap- sulation of the insides of the state. The *entry points* allow users to customize the way to enter the com- posite 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 formal- izing the Precise Semantics of UML State Machine (PSSM) is under standardization of the OMG. It de- fines 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 semanti- cally 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 con- figuration is not specified by UML because the spec- ification wants to be abstract. We allow to determine these values through a specific profile. Note that the configuration information might not be needed in dy- namic 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 com- piled 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 ma- chines to process.

**Concurrency:** Concurrency aspects in state ma- chines including doActivity of states, orthogonal re- gions, event detection, and event queue management are handled by the execution of multiple threads. Cur- rently, we use POSIX threads for concurrency.

# 4CONCURRENCY

This section describes our design of concurrency aspects of state machines in generated code at run- time.

## Thread-based design

The concurrency of USMs is based on multi- ple 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 asso- ciated method and goes back to the **wait** point if it receives a stop signal or its associated method com- pletes. Each PT is associated with one of the follow- ing actions:

* + - *doActivity* of each state if has any.
    - Sleep function associated with a time event which

counts ticks and emits the event once it completes.

* + - Change detect function associated with a change

event which observes a variable or a boolean ex-

pression 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 sig-

nals to other PTs.

STs which are spawned by a parent thread, joined until and destroyed once the associated methods com- plete. The STs follow a paradigm in which the spawn- ing 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.

## Thread communication

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

**Run-to-completion:** The event process must follow the run-to-completion semantics of UML State Ma- chines. The semantics means that the state machine completes processing of each event before starting processing the next event. If all events are asyn- chronous, the main thread processes events by read- ing one-by-one from the event queue. However, be- cause we allow call events to be synchronous, the pro- cessing 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 execut- ing the actual processing. In generated code, lock and unlock are implemented using signals and conditions in POSIX (POSIX, ).

**Multi-threaded problems checking:** We use POSIX threads to realize concurrency in UML State Ma- chines. We use the Valgrind DRD tool (DRD, ) to check multi-thread problems such as data races, dead- lock, and misuse of POSIX threads API in generated code derived from the PSSM test suite. The gen- erated code is free of multi-thread errors. The re- sult shows that code generated by our tool potentially avoids multi-thread problems.

# 5CODE GENERATION PATTERN

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

## State

A common state type *IState* is created. The type has two attributes called *actives*, to preserve the hier- archy of composite states, and *previousActives* refer- 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 (which is a child element of an enumeration). During

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

1 **t y p e d e f s t r u c t** I S t a t e *{*

**i n t** p r e v i o u s A c t i v e s [ 2 ] ; **i n t** a c t i v e s [ 2 ] ;

3 *}* I S t a t e ;

**c l a s s** C *{*

5 **p r i v a t e** :

I S t a t e s t a t e s [ STATE MAX ] ;

7 **p u b l i c** :

**void** e n t r y ( S t a t e I d i d ) *{*

9 **s w i t c h** *{* i d *} {*

**c as e** S0 ID :

11 / /a c t i o ncodef o reachs t a t e

**break** ;

13 / /codef o ro t h e rs t a t ea c t i o n s

*}*

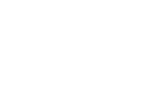
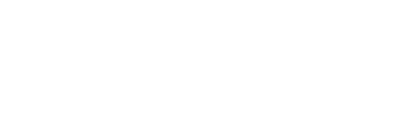
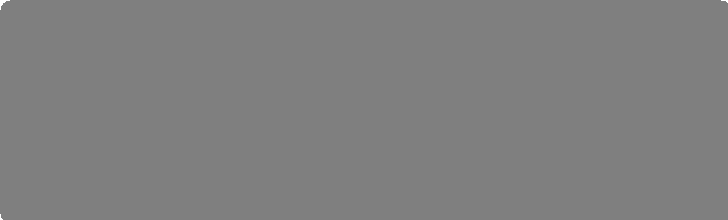


Figure 1: Example illustrating different ways entering a composite state

## Region

15 *}*

*}*

Listing 2: Example code generated for doActivity

**while** ( **t ru e** ) *{*

2 p t h r e a d m u t e x l o c k (& mutex [ s t a t e I d ] ) ;

**while** ( ! i s S t a r t s [ s t a t e I d ] ) *{*

4 / /a w a i ts t a r ts i g n a l

p t h r e a d c o n d w a i t (& cond , &mutex [ s t a t e I d ] ; *}*

6 d o A c t i v i t y ( s t a t e I d ) ;

i s S t a r t s [ s t a t e I d ] = **f a l s e** ;/ /r e s e tw a i tf l a g

8 p t h r e a d m u t e x u n l o c k (& mutex [ s t a t e I d ] ) ;

**i f** ( ! i s S t o p s [ s t a t e I d ] ) *{*

10 **i f** ( s t a t e I d == S0 ID *| |* . . . ) *{* / /a t o m i cs t a t e s

push Completion Event ( s t a t e I d ) ;

12 *}*

*}*

14 *}*

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 are 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 corre- sponding 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 control- ling 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 correspond- ing 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 doActiv- ity. The generated code typically follows the common paradigm in POSIX threads (POSIX, ).

Our approach considers regions as elements to be transformed. Specifically, each region two methods: entering and exiting are added. 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 in- nermost to outermost.

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**: enter- ing at a direct or an indirect sub-vertex of composite states. The two entering ways execute the entry ac- tion of the containing composite state after the tran- sition 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 en- tering 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, *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.

Transitioning from a vertex to a sub-vertex of the composite state (transition from *S*0 to *SH* is a partic- ular 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 sim- pler than that of entering. It basically executes the exit actions of all the active sub-states from innermost to outermost.

Listing 3: Example code generated for the region of S1

vo i d S 1 Region 1 Enter ( i n t e n t e r m o d e ) *{*

2 **i f** ( e n t e r m o d e == DEFAULT) *{*

s t a t e s [ S1 ID ] . a c t i v e s [ 0 ] = S3 ID ;

4 e n t r y ( S3 ID ) ; s e n d S t a r t S i g n a l ( S3 ID ) ; S 3 Region 1 Enter ( DEFAULT) ;

6 *}* **e l s e i f** ( e n t e r m o d e == S2 MODE) *{*

/ / . .

8 *}* **i f** ( e n t e r m o d e == SH MODE) *{*

State IDEnum h i s ;

10 **i f** ( s t a t e s [ S1 ID ] . p r e v i o u s A c t i v e s [ 0 ] ! = STATE MAX) *{*

h i s = s t a t e s [ S1 ID ] . p r e v i o u s A c t i v e s [ 0 ] ;

12 *}* **e l s e** *{*

h i s = S2 ID ;

14 *}*

s t a t e s [ S1 ID ] . a c t i v e s [ 0 ] = h i s ;

16 e n t r y ( h i s ) ; s e n d S t a r t S i g n a l ( h i s ) ;

**i f** ( S3 ID == h i s ) *{*

18 S 3 Region 1 Enter ( S3 REGION1 DEFAULT ) ;

*}*

20 *}* **e l s e i f** ( e n t e r m o d e == S4 MODE) *{*

s t a t e s [ S1 ID ] . a c t i v e s [ 0 ] = S3 ID ;

22 e n t r y ( S3 ID ) ; s e n d S t a r t S i g n a l ( S3 ID ) ; S 3 Region 1 Enter ( S4 MODE) ;

24 *}* **e l s e i f** ( e n t e r m o d e == ENP MODE) *{* . . . *}*

## Event

Similar to the approach in (Niaz et al., 2004), one method is generated for each event. An event enu- meration *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 im- plicitly implemented. A completion event is fired when either the execution of the *doActivity* of sim- ple/atomic state completes or all regions of a com- posite 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 previ- ously mentioned, and executes the event processing (see 4.2).

**SignalEvent**: An API *sendSignal* is created for en- vironment code to interact and send an instance of the signal associated with the event is written into the event queue. When the API is called, an event is emit- ted and written into the event queue.

**TimeEvent**: A thread associated with the event is cre- ated 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 own- ing 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 automat- ically generated. They do not only affect the value of *x/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 al- ways has the highest priority. Others are equal by de- fault. Event type, priority, identifier, associated state *stateId* of completion events, and signal data are spec- ified 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.

## Transitions

Each event triggers a list of transitions. We sup- pose *Ttrig*(*e*) is the transition list triggered by the event *e*, and *Strig*(*e*) is a depth-ordered (from innermost to outermost) set of the source states of the transitions in *Ttrig*(*e*).

Algorithm 1 describes how to generate the body of an event method. It first finds the innermost ac- tive states which are able to react *e* by orderly loop- ing over *Strig*(*e*). This is to ensure that, in case of multiple transitions triggered by the event, the generated code for the transitions outgoing from in- nermost states will be executed. For each transi- tion from an innermost state, code for active states and deferred events, guard checking, and transi- tion 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 pa- per), *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 events back to the main queue once one of the pending events is processed and the active state is changed.

For a transition *t*, *GEN CHECK* can generate sin- gle 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

**Algorithm 1** Code generation for events

**Require:** Event *e*

**Ensure:** Code generation process for event method

1: **procedure** EVENTGENPROCESS(*e*)

2: **for** *∀* s *∈ Strig* (*e*) **do**

3: *Ts* = *{t ∈ Ttrig* (*e*)*|src*(*t*) = *s}*

4: **for** *∀t ∈ Ts* **do**

5: *GEN CHECK*(*s, t, e*)

6: *GEN GUARD*(*t*)

7: *GEN TRANS*(*s, t, tgt*(*t*))

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

**i f** ( e v e n t . s t a t e I d == S6 ID *| |* e v e n t . s t a t e I d == S7 ID ) *{*

2 **i f** ( s t a t e s [ S6 ID ] . a c t i v e s [ 0 ] == S7 ID && s t a t e s [ S6 ID ] . a c t i v e s [ 1 ] == S8 ID ) *{*

4 t h r e a d r 1 =FORK( S 6 Region 1 Exit ) ; t h r e a d r 2 =FORK( S 6 Region 2 Exit ) ;

6 JOIN ( t h r e a d r 1 ) ; JOIN ( t h r e a d r 2 ) ; s e n d S t o p S i g n a l ( S6 ID ) ; **e x i t** S 6 ( ) ;

8 t h r e a d t 1 4 =FORK( e f f e c t ( t 1 4 ) ) ; t h r e a d t 1 5 =FORK( e f f e c t ( t 1 5 ) ) ;

10 JOIN ( t h r e a d t 1 4 ) ; JOIN ( t h r e a d t 1 5 ) ; e f f e c t t 1 6 ( ) ;

12 a c t i v e S t a t e I D = STATE MAX;/ /i n a c t i v es t a t e

*}*

**Algorithm 2** Code generation for transition

**Require:** A source *vs* , a target vertex *vt* and a transition *t*

**Ensure:** Code generation for transition

1: **procedure** GEN TRANS(*vs* , *vt* , *t*)

2: Find *sex* and *sen* as vertexes in the same region and directly or indi- rectly containing/being *vs* and *vt* , respectively.

3: Generate IF-ELSE statements for junctions

4: **if** *sex* is a state **then**

5: **for** *r ∈* regions of *sex* **do**

6: *FORK*(*RegionExit*(*r*)) //create thread for exiting region

7: Generate JOIN for threads created above

8: Generate sendStopSignal to *sex*

9: *exit*(*sex* ) //exit the state

10: **if** *vt* is a pseudo state join **then**

11: **for** *in ∈* incoming transitions of *vt* **do**

12: *FORK*(*e f f ect*(*in*)) //create thread for transition effect

13: Generate JOIN for threads created above

14: **else**

15: *e f f ect*(*t*) //execute transition effect

16: **if** *sen* is a state **then**

17: *entry*(*sen* ) //state entry

18: Generate sendStartSignal to *sen*

19: **if** *sen* is a composite state **then**

20: **for** *r ∈* regions of *sen* **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

14 *}*

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 orthog- onal regions, which actually exit *S7* and *S8*. Then, *exit(S6)* is executed before the concurrency of transi- tion effects *t14* and *t15* is taken into account.

*GEN TRANS* is able to generate code for transi- tions between two vertexes. Algorithm 2 shows how it works. The generated code is contained by the de-

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

**i f** ( a c t i v e R o o t S t a t e == S1 ID ) *{*

2 j u n c = 0 ;/ /o u t g o i n g **t r a n s i t i o n** t 9ofj u n c

**i f** ( guard ) *{* j u n c = 1 ; *}*

4 / /E x i ts u b s t a t e sofS1andS1 e f f e c t ( t 9 ) ;

6 **i f** ( j u n c == 0) *{*

e f f e c t ( t 1 1 ) ;

8 *}* **e l s e** *{*

e f f e c t ( t 1 0 )

10 *}*

FORK( e f f e c t ( t 1 2 ) ) ; FORK( e f f e c t ( t 3 ) ) ;

12 / /JOIN. . .== *>* c o n c u r r e n te x e c u t i o n

/ /E n t e rs t a t eS6,S7andS8

14 *}*

ferral events, active states, and guard checking.

Firstly, Algorithm 2 looks for the *sex* and *sen* ver- texes contained in the same region and respectively containing the source and target vertexes of the tran- sition *t*. For example, *sex* and *sen* in case of the *t*3 tran- sition 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 in- volves some *junction*s, IF-ELSE statements for junc- tions are generated first (as PSSM says *junction* is evaluated before any action). The composite state is exited by calling the associated exiting region meth- ods (FORK and JOIN for orthogonal regions) in Step 3 and followed by the generated code of transition ef- fects (Step 4 and 5), respectively. If the parent state *sen* of the target vertex *vt* 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 *vt* of the transition *t* is a pseudo state, the generation pat- tern 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- ternal transitions. Due to space limitation, the detail of generating local and internal transitions is not dis- cussed here but the only difference is that the com- posite state containing the transitions is not exited.

Table 1: Pseudo state code generation pattern

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

# 6EMPIRICAL STUDY

The pattern is implemented in the Papyrus De- signer tool (pap, ) - an extension of the UML mod- eling tool Papyrus (CEA-List, ). Papyrus Designer supports component-based modeling and code gener- ation. The behavior of a component in Papyrus De- signer 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 imple- menting threads for concurrency. This section reports our experiments with the standalone on the semantic- conformance and efficiency of generated code.

## Semantic conformance of runtime execution

This section presents our results found during ex- periments with our tool to answer the following re- search question.

To evaluate the semantic conformance of runtime execution of generated code, we use a set of examples provided by Moka (mok, ), which is a model execu- tion engine offering PSSM (and also part of the Pa- pyrus 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.

|  |  |
| --- | --- |
| 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*. |
| junctio | n As a static version *choice*, a *junction* is transformed into an attribute *juncattr* and evaluated before any action executed in compound tran- sitions (see Listing 5, lines 2-3 and 6-10). The value of *juncattr* 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 re- gion, 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 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 *expoint* is generated by using *GEN TRANS*. If *expoint* has multiple incoming transitions from orthogonal regions, it is generated as a *join* to multiple-check the source states of these incomings. |
| termina | teThe code executes the exit action of the innermost active state, the effect of the transition and destroys the state machine object. |

**Step 2** We simulate the execution of the **State ma- chine** 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), en- tering (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 event. For pseudo-states, histories are not supported. Thus, our evaluation result is limited to the current specification of PSSM.

**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.

State machine

1. Code generation

Code

1. Simulation

MOKA



1. Execution

Traces 1 Runtime execution

Traces 2

1. Trace comparison

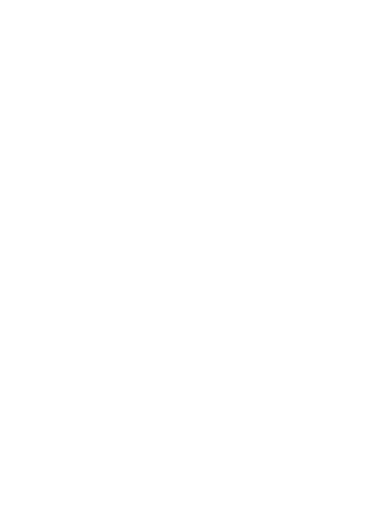
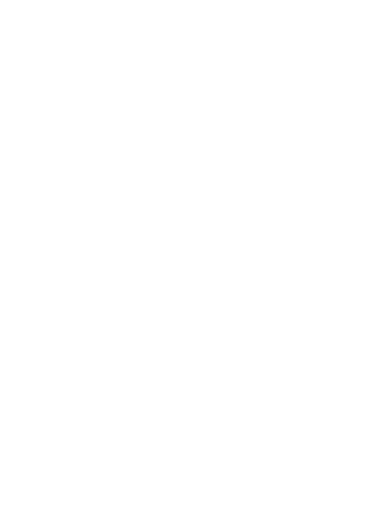
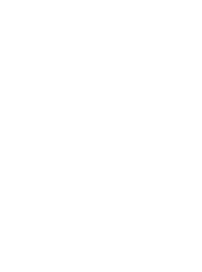
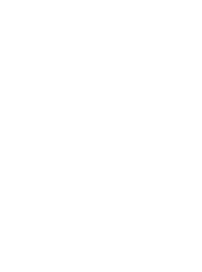
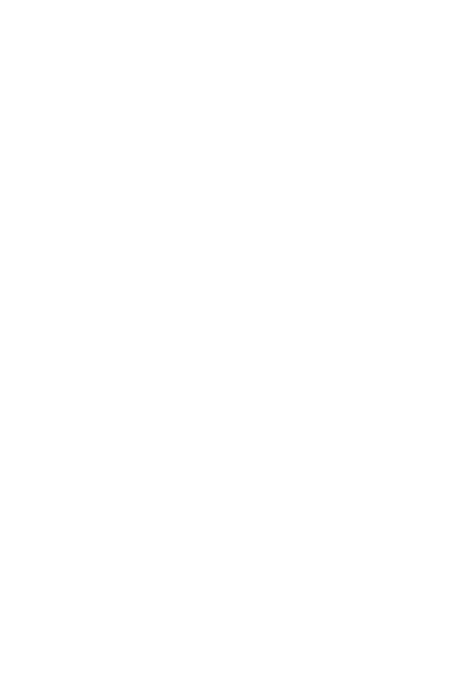
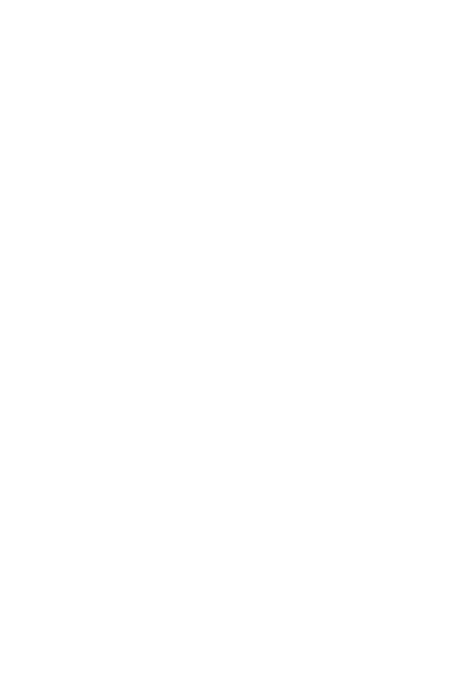
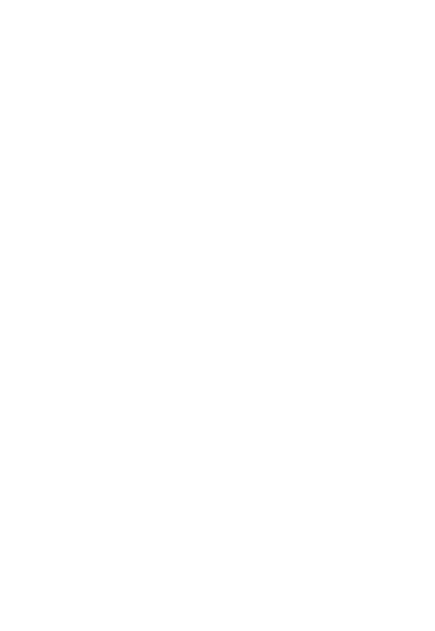
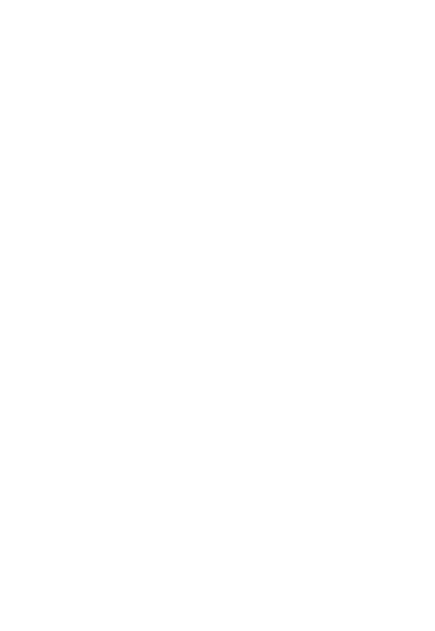
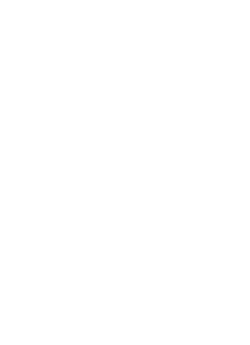
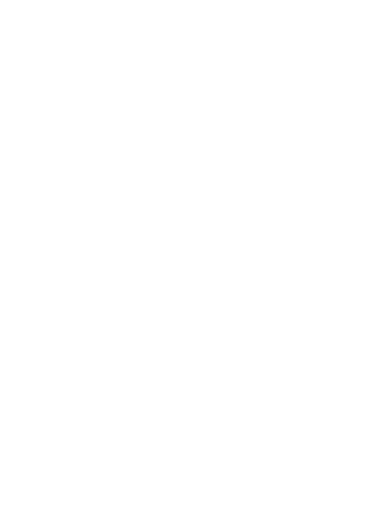
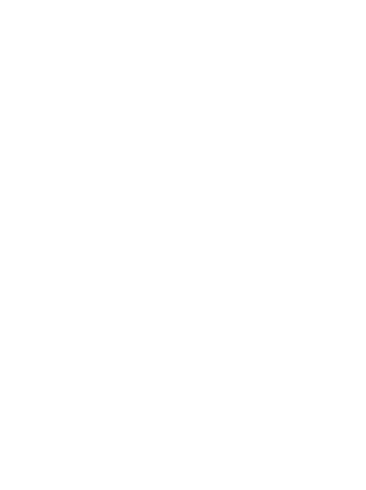
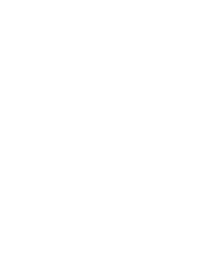
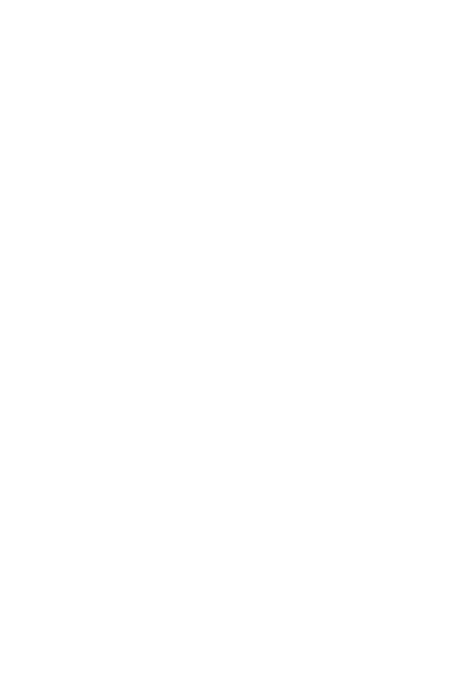
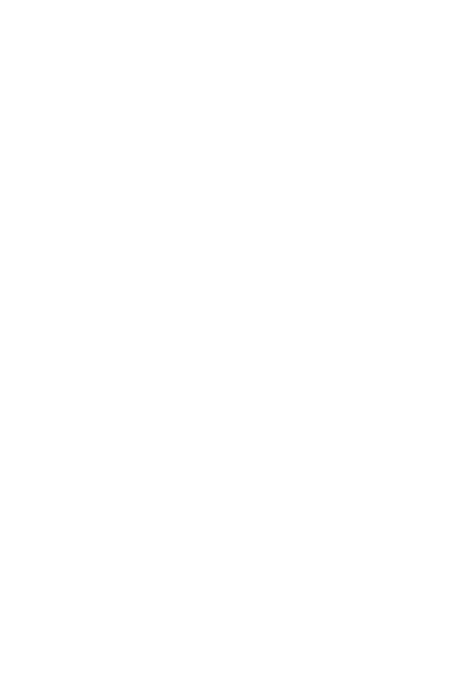
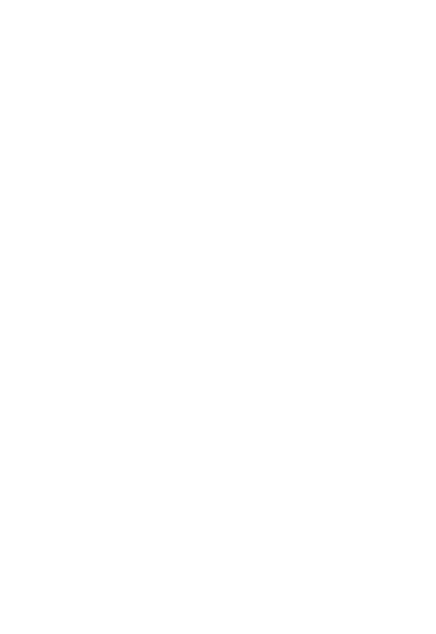
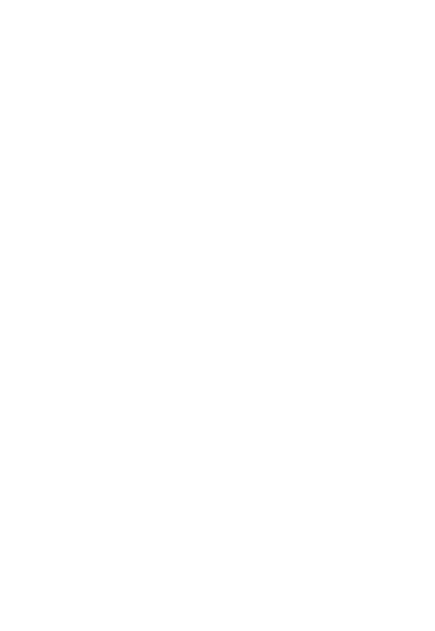
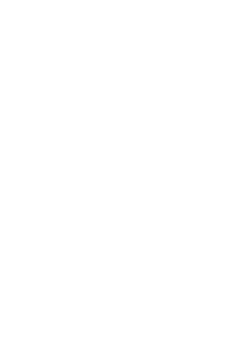
Figure 2: Semantic conformance evaluation methodology

## Benchmarks

execution time (ms)

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

Figure 3: Event processing speed for the benchmarks



700

600

500

400

300

200

100

0

Simple benchmark

Composite benchmark

Min Outlier Max Outlier

***RQ2:*** *Runtime performance and memory usage are undoubtedly critical in real-time and embedded sys- tems. Particularly, in event-driven systems, the per- formance 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 pre- ferred benchmark used by the Boost C++ libraries (boost, 2016) in (ben, ). One simple example only consists of atomic states and the other both atomic and composite states.

We compared our tool with tools such as Sinela- bore (which generates efficient code for Magic Draw (Magic, 2016), Enterprise Architect (SparxSysemx, 2016)), Quantum Modeling (QM) (QM, 2016) (which generates code for event-driven active object frame- works (Lavender and Schmidt, 1996)) , Boost Stat- echart (Library, 2016), Meta State Machine (MSM) (MSM, 2016), C++ 14 MSM-Lite (ben, ), and func- tional programming like-EUML(EUM, ).

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 mil- lions of events are generated and processed by the simple example and more than 4 millions for the com- posite example. Processing time is measured for each case.

### Performance

Fig. 3 shows the event processing performance of the approaches for the two benchmarks. In the nor- mal compilation mode (postfix N), Boost Statechart, MSM, MSMLite, EUML are quite slow and not dis- played in the box-plot.

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 ex- periment. The figure also shows that the optimization of GCC is significant. In normal mode only the per- formance of Sinelabore, QM, and our tool is accept- able. The event processing speed of MSM, MSM Lite

Figure 4: Event processing performance in optimization mode



%

Performance comparison in optimization mode

Simple benchmark

200 Composite benchmark

179,9

150

106,8

100

100

100 107,1

100

70,7 78,6

75,5

51,3

56,4

50

42,7

0

and EUML is too slow without GCC optimizations.

### Memory usage

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

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

For runtime memory consumption, we use the Valgrind Massif profiler (Mas, ; Nethercote and Se- ward, 2007) to measure memory usage. Table 3 shows the memory consumption measurements in- cluding stack and heap usage for the composite ex- ample. 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, Sinela- bore, QM, and Our tool, respectively.

IntersectionStateMachine

Initial1 HighwayOpen

SwitchingHighwayToFarmroad

SwitchingFarmroadToHighway

FarmwayOpen



TrafficLightStateMachine

Red OnRed

OnRed\_Yellow

Yellow

Initial1

OnYellow

Red\_Yellow OnGreen Green

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 76.03 | 75.5 | 75.8 | 75.5 | 75.8 | 75.7 | 76.38 |

Farmway C



Intersection

TrafficLight

1

1

+ highway

+ farmroad

Highway

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

Highway

C

Farmway

minimum time for the highway open is elapsed; and

Figure 5: Traffic Light Controller (left) and its class dia-

gram (right).

# 7TRAFFIC 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 (**?**).

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 de- sign in Yasmine (Yasmine, ), 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 ma- chine. The state machines of *Intersection* and *Traf- ficLight* are shown in Fig. 6 (left and right, respec- tively). All of the states of *IntersectionStateMachine*, except *FarmwayOpen*, are composite. The details of *SwitchingHighwayToFarmroad* and *SwitchingFarm- roadToHighway* are actually shown on the yasmine site (Yasmine, ).

The conditions for switching from the state *High- wayOpen* to *SwitchingHighwayToFarmroad* are: (1) a

(2) the sensors emit a signal.

To show the usability and practicality of UML events, two alternative designs can be specified by us- ing 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 transi- tion from *WaitingForHighwayMinimum* to *Minimum- TimeElapsed*, and a signal event deferred by the *Wait- ingForHighwayMinimum* state. When *HighwayOpen* becomes active, its active sub-state remains *Waiting- ForHighwayMinimum* 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 *MinimumTimeE- lapsed* in case the time event is fired. The signal event is then processed to finish the execution of *Highway- Open* and activate the farmway.

The other design utilizes a change event instead of deferred events for switching from *WaitForPrecondi- tions* to a final state. Two flags *timeFlag* and *detect- Flag* are used. The *WaitForPreconditions* state has two internal transitions. One is triggered by a signal event associated with the signal C and calls a transi- tion effect to update *detectFlag* to true. The other one triggered by a time event sets *timeFlag* to true. The expression associated with the change event updates from false to true once two flags *timeFlag* and *detect- Flag* are set to true. The periodic evaluation time is configured as 10ms.

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

The support of UML events (change events and time events) and deferred events does not only pro- vide designers more options to specify but also sim- plify system behaviors. It can also reduce the num- ber of states. For example, the numbers of sub-states of *HighwayOpen* with the use of deferred events and



DetectorOn

f1

a



DetectorOn

f1

b

Figure 7: Alternative state machine designs for the *High- wayOpen* state

change events are two and one, respectively, while Yasmine requires three states. However, deferred events might make the design more difficult to under- stand because of its specialized semantics.

# 8RELATED WORK

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

Switch/if is the most intuitive technique for imple- menting 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 ta- ble approach (Douglass, 1999) uses one dimension for representing states and the other one for all possi- ble 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 sup- ports flat state machines. This pattern is extended in (Niaz et al., 2004) to support hierarchical state ma- chines. Recently, a double-dispatch (DD) pattern pre- sented in (Spinke, 2013) extends (Niaz et al., 2004) to support maintainability by representing states and events as classes, and transitions as methods. How- ever, as the results shown in (Spinke, 2013), these pat- terns require much memory because of an explosion of the number of classes and use dynamic memory al-

location, which is not preferred in embedded systems. It is worthy noting that none of these approaches pro- vides implementation for all of state machine pseudo states as well as events.

Tools such as (SparxSystems, 2016; IBM, ) apply different patterns to generate code. However, as men- tioned in Section 1, true concurrency, some pseudo- states, and UML events are not supported. FXU (Pil- itowski and Derezin˜ska, 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 gener- ation for different languages such as C++ and Java from state machines. However, Umple does not sup- port 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 pat- tern, to produce small footprint, and the pattern in (Niaz et al., 2004), to preserve state hierarchy. Fur- thermore, we define patterns to transform all of USM concepts including states, pseudo states, transitions, and events. Therefore, users are flexible to create their USM conforming to UML without restrictions.

# 9CONCLUSION

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 Designer tool. We evaluated our tool by conducting experiments on the semantic-conformance and effi- ciency 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 ef- ficiency, we used the benchmark defined by Boost to compare code generated by our tool with other ap- proaches. The results showed that our tool produces efficient code that runs fast in event processing and is small in executable size.

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

# REFERENCES

DRD: a thread error detector. <http://cs.swan.ac.uk/> csoliver/ok-sat-

Forward, A. Perceptions of software modeling: a sur- vey of software practitioners.

IBM. Ibm Rhapsody.

library/internet html/doc/doc/Valgrind/3.8.1/html/drd-IBM (2016). IBM Rhapshody and

manual.html.

Moka Model Execution.

UML differences. [http://www-](http://www-/)

01.ibm.com/support/docview.wss?uid=swg27040251.

https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecutio[nO. nline; accessed 04-July-2016].

Papyrus Software Designer. https://wiki.eclipse.org/Papyrus Software Designer.

State Machine Benchmark. [http://boost-](http://boost-/) experimental.github.io.

State Machine Benchmark.

Valgrind Massif. <http://valgrind.org/docs/manual/ms-> manual.html.

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 (MOD- ELSWARD), 2014 2nd International Conference on*, pages 235–245. IEEE.

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

Balser, M., Ba¨umler, 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 verifica- tion 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 (2016). Boost C++. [http://www.boost.org/.](http://www.boost.org/) [Online; accessed 04-July-2016].

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

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.

Dom´ınguez, E., Pe´rez, B., Rubio, A. L., and Zapata,

M. A. (2012). A systematic review of code gen- eration proposals from state machine specifica- tions.

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

Knapp, A. (2004). Semantics of UML State Ma-

chines.

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

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

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

MSM (2016). Meta State Machine. <http://www.boost.org/doc/libs/1> 59 0 b1/libs/msm/doc/HTML/ind [Online; accessed 04-July-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., Kien- zle, J., and Scho¨ttle, 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 in- strumentation. In *ACM Sigplan notices*, vol- ume 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 (2015). Precise Semantics Of UML Composite Structures. (October).

Pilitowski, R. and Derezin˜ska, A. (2007). *Code Gen- eration and Execution Framework for UML 2.0 Classes and State Machines*, pages 421–427. Springer Netherlands, Dordrecht.

POSIX. POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/.

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

Shalyto, A. and Shamgunov, N. (2006). State ma- chine design pattern. *Proc. of the 4th Interna- tional Conference on.NET Technologies*.

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

SparxSystems (2016). Enterprise Architect.

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 implemen- tation of concurrent and hierarchical state ma- chines. *Information and Software Technology*, 55(10):1726–1740.

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. The classic farmroad example. [http://yasmine.seadex.de/yasmine.html.](http://yasmine.seadex.de/yasmine.html)