UML State Machine: Multi-thread-based concurrency code generation

Van Cam Pham, Ansgar Radermacher, Sebastien Gerard
CEA-List, Laboratory of Model-Driven Engineering for Embedded Systems (LISE)
Gif-sur-Yvette, France
Email: first-name.lastname@cea.fr

***Abstract*—Event-driven architecture is an useful way to design and solve the complexity of today systems. Unified Modeling Language State Machine and its visualization are a powerful means to the modeling of the logical behavior of such architecture. Over the years of research and development, many approaches focus on reducing the gap between the modeling and implementation world. However, despite many useful UML State Machine concepts for modeling and designing complex system, the support of existing generation approaches is still limited to simple cases, especially when considering concurrency of *doActivity* and orthogonal regions, pseudo states such as history, and different event types.**

In order for wider integrating MDE into software development today, one task, among other important tasks, is needed to generate code for all modeling concepts with respect to the semantics. This paper presents a code generation approach whose objective is to clean the above issues. The approach combines IF-ELSE constructions of programming languages and the state pattern with our support for concurrency. Even supporting multi-thread-based concurrency, binary files compiled from and the event processing speed of runtime execution of code generated by our approach are smaller 20 and 30 [?] times, respectively, in comparison with to the boost library ==> should have compare with others.

I. Introduction

The wide application of Internet of Things [1] drives the complexity of embedded systems today rapidly increases. Today embedded applications directly interacting with running environment does not run in standalone mode but also react to the system environment changes. Event-driven architecture [2] is an useful way to design such systems, in which events come to the system either from outside (data) or inside the system itself (time event or internal changes). The Unified Modeling Language State Machine (USM) [3] standardized by OMG and its visualization are a powerful means to the modeling of the logical behavior of such systems. USM defines how systems behave when there are changes occurred.

The rise in use of the Model-Driven Engineering (MDE) approach promotes the automation in software development. MDE relies on two paradigms, abstraction and automation [4], which are recognized as very efficient for dealing with complexity. The most useful advantage of MDE is to bring the ability to automatically generating code from diagram-based modeling languages such as USM to executable code. The gap between the modeling world, which consists of software architects, who prefer using graphical languages such as USM, and the implementation world, which involves programmers, is therefore reduced.

However, on the one hand, OMG seeks to raise the usefulness of UML State Machine by providing more modeling concepts supporting software architects for modeling and designing complex systems. On the other hand, the support of existing generation approaches, over the years of research and development, is still limited to simple cases, especially when considering concurrency of *doActivity* and orthogonal regions, pseudo states such as history, and different events. This again enlarges the gap between the UML State Machine semantics and the actual generated code. Specifically, the following lists some issues of current approaches:

- Most of existing tools and approaches only focus on the sequential aspect while the concurrency such as the *doActivity* behavior of states and orthogonal regions is not taken into account.
- The support for pseudo states such as history, choice and junction is poor while these are very helpful in modeling.
- Code generated from tools such as [5] and [FXU] is heavily dependent on their own libraries, which makes the generated code not portable.
- Issues of event processing speed, executable file size, runtime memory consumption, and UML semantic-conformance of generation code.

In order for wider integrating MDE into software development today, one task, among other important tasks, is needed to seamlessly bring the theory specified by the UML specification and the Precise Semantics of UML Composite Structures (PSCS) [6] into practice of complex software development with respect to the semantics as much as possible. To do it, this paper presents a code generation approach whose objective is to clean the above issues.

We present a novel approach combining the IF-ELSE constructions of programming languages and the state pattern [XXX] with our support for concurrency. Although supporting multi-thread-based concurrency, binary files compiled from and the event processing speed of runtime execution of code generated by our approach are smaller 20 and 30 [?] times, respectively, in comparison with to the boost library ==> should have compare with others.

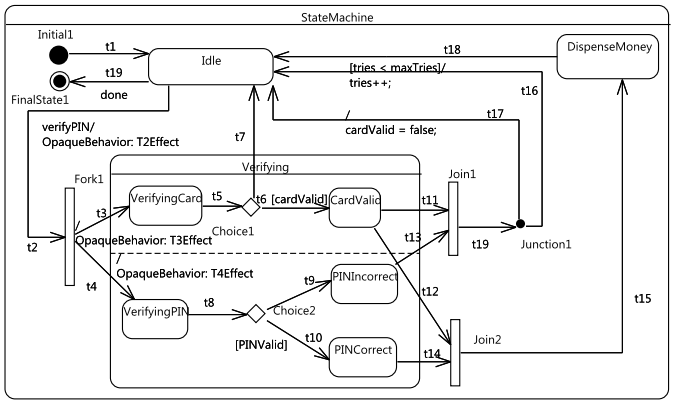

Fig. 1. ATM State machine example

The remaining of this paper is organized as follows: A motivating example is given in Section II. Section III presents preliminary USM concepts in a formal way. Thread-based concurrency is analyzed in Section IV. Based on the analysis, a code generation approach is proposed Section V. The implementation and empirical evaluation are reported in Section VI. Section VII argues related work. The conclusion and future work are presented in Section VIII.

II. Motivating example

III. Background definition

Definition III.1. A directed graph $G = \{V, Ed\}$ consists of a finite set V of vertexes, and a set Ed of edges. An edge connects a source vertex to a target vertex. The source and target vertexes of an edge ed are obtained by $src(ed)$ and $tgt(ed)$.

Definition III.2. A UML vertex $v \in V$ has a kind $v.kind \in$ *{initial, final, state, comp, conc, join, fork, choice, junction, enpoint, expoint, history}*. Each vertex v has a name and we write $v.name$.

Definition III.3. A region $r \in \mathcal{R}$ is composed of one or several vertexes, and contained by a state s. We write $ctner(r) = s$ and $subvts(r)$ is its sub-vertices set.

Definition III.4. A UML state s is a vertex v where v.kind $\in$ state, comp, conc. s has an $entry$, an $exit$ and a $doActivity$ action. A composite state cs contains one or more vertexes. We write $subvts(cs)$ is a set of vertexes contained by cs and $ctner(v)$ refers to the containing state of the vertex v. A concurrent state contains more than one region.

Definition III.5. An action $act \in ActLang$ is a set of statements written in an object-oriented programming language $ActLang$. A guard is a boolean expression written in $ActLang$.

Definition III.6. A transition $t \in T$ is an edge connecting two vertexes. A transition has a guard $guard(t)$, an effect $effect(t)$, and is associated with a set of events $\subset$ E. We write $events(t)$ as the associated set of events. A transition has a type $t.type \in \{trigger, triggerless, guardless\}$ and a kind $t.kind \in external, local, internal$.

Definition III.7. A *TimeEvent* te an internal event and specifies the time of occurrence d relative to a starting time. The latter is specified when a state, which accepts the time event, is entered.

Definition III.8. A *Signal* sig is data described by its attributes.

Definition III.9. A *SignalEvent* se is associated with a signal sig and is occurred if sig is received by a component, which is an active UML class.

Definition III.10. A *ChangeEvent* che is associated with a boolean expression $ex(che)$ written in $ActLang$. che is emitted if $ex(che)$ changes from true(false) to false(true).

Definition III.11. A *CallEvent* ce is associated with an operation $op(ce)$. ce is emitted if there is a call to $op(ce)$.

Suppose that for each vertex $v \in V$, its incoming and outgoing transition lists are extracted by the functions $incomings$ and $outgoings$, respectively. For a list l, the function $head$ is used to get the first element of the list. If $v.kind = conc$, suppose $regions(v)$ is the region set contained by v. Given a transition t:

- $t.type = trig$ if $\#events(t) > 0$.
- $t.type = tless$ if $\#events(t) = 0$.
- $t.type = gdless$ if $(guard(t) = true \vee \nexists guard(t)$.
- $t.type = triggdless$ if $\#events(t) = 0 \wedge (guard(t) = true \vee \nexists guard(t))$.

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

Definition III.12. A state machine sm is a graph specified by $\{V, T\}$ associated with a set of events E. A state machine is a special composite state which has no incoming and no outgoing transitions. A root vertex v is a direct sub-vertex of the state machine, $ctner(v) = sm$. The set of regions contained by sm is written $\mathcal{R}$.

For each vertex $v \in V$, we write the following sets $T_{ins}(v) = incomings(v), T_{outs}(v) = outgoings(v)$, $t_{first} = head(t_{outs})$; transitive transition sets $T^+_{ins}(v)$ and $T^+_{outs}(v)$ are sets of transitions incoming to and outgoing from, respectively, v or direct or indirect sub-vertexes of v.

Definition III.13. Transitive container $ctner^+(v)$ of a vertex v of a state machine sm is defined as following:

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

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

In the example in Fig. 1, we have:

$$ctner^{+}(Idle) = \{StateMachine\},$$
$$ctner^{+}(Choice1) = \{Verifying, StateMachine\}.$$

A state machine $sm = \{V, T\}$ associated with E is validated if, for each $v \in V$, the constraints listed in Table I are hold.

//should have some definition of compound transition

Definition III.14. Current active configuration Cfg of a UML state machine sm is a set of candidate UML states which are able to process an incoming event.

IV. Thread-based Concurrency

A. Thread-based concurrency analysis

While concurrency is an important aspect defined by the UML State machine specification, especially hierarchical and concurrent state machines with *doActivity*s for states, most of existing approaches do not take into account. This is non-trivial since concurrency is dynamic in UML state machine since the number of threads used for concurrency is non-deterministic.

For example, assuming that *Idle* is the current active state of the ATM state machine in Fig. 1 and a *verifyPIN* event is coming. The *doActivity* behavior of *Idle doActivity(Idle)* (if has) is terminated, *exit(Idle)* and the *effect(t2)* (*T2Effect*) are executed sequentially. These actions are run in a state machine main thread which reads incoming events from a "first in, first out" (FIFO) priority queue. Fig. 2 shows the activity diagram representing the concurrency of the state machine example when processing the *verifyPIN* event, in which each activity partition represents a thread. The completion of *effect(t2)* is followed by *effect(t3)* and *effect(t3)*, which are run concurrently since the transitions owning these effects outgo from a fork pseudo state. Two threads *T3Run* and *T4Run* associated with *effect(t2)* and *effect(t3)*, respectively, are created by *FORK*. The entry action *entry(Verifying)* of *Verifying* is executed following the termination of the two threads.

After *entry(Verifying)* completion, the UML specification says that *doActivity(Verifying)*, *entry(VerifyingCard)* and *entry(VerifyingPIN)* should be concurrently executed, which is represented by a fork node, in which a *Start* signal is sent to *VerifyingDoRun* in order for commencing *doActivity(Verifying)*. As the *Verifying* state, the *doActivity*s of the states *VerifyingCard* and *VerifyingPIN* are also concurrently started. Also, upon the completion of *entry(VerifyingCard)* and *entry(VerifyingPIN)*, the main thread completes the processing of the *verifyPIN* event, reads next events from the queue or waits for next event occurrences.

If no event is coming, and *doActivity(VerifyingCard)* and *doActivity(VerifyingPIN)* are long actions (e.g. forever loops inside), the state machine remains its active configuration and

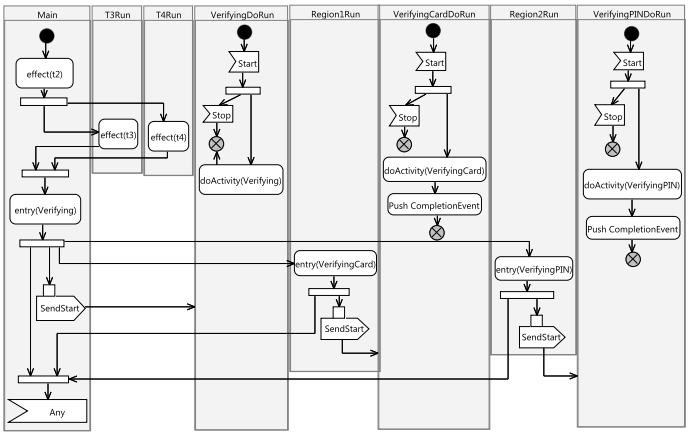

Fig. 2. Concurrency of the ATM when receiving the *verifyingPIN* event

three concurrent actions including *CheckForEvents*, *doActivity(VerifyingCard)*, and *doActivity(VerifyingPIN)* are permanently run.

It is worth noting that the termination time of *doActivity(VerifyingCard)* and *doActivity(VerifyingPIN)* is non-deterministic. However, whenever one of those completes, a completion event associated with the state corresponding to the completed *doActivity* is generated and pushed to the event queue. For illustration, assuming that *doActivity(VerifyingCard)* terminates before *doActivity(VerifyingPIN)*. As the activity diagram in Fig. 3, the Main thread checks the *CompletionEvent* upon the completion of *doActivity(VerifyingCard)*. *exit(VerifyingCard)* and *effect(t5)* are then executed sequentially. If *cardValid* is computed as true as the result of *doActivity(VerifyingCard)* and *exit(VerifyingCard)*, the Main thread simply executes *effect(t6)* and *entry(CardValid)* before waiting for other events.

In contrast, Main sends *Stop* signals to stop *doActivity(VerifyingPIN)* and *doActivity(Verifying)*, executes exit actions, effects and entry actions in an appropriate order (see Fig. 3) and waits for other events.

So far, we see that the number of concurrent actions is not constant but changes timely. Each action can either deterministically or non-deterministically terminate. In this sense, deterministic actions (DAs) prevent the Main thread from going to the waiting-for-event point. In other words, pending events in the queue are only read and processed once all deterministic actions complete. Therefore, we redefine the run-to-completion paradigm of UML state machine as following:

Definition IV.1. Run-to-completion means that, in the absence of exceptions or asynchronous destruction of the context class object or the state machine execution, a pending Event occurrence is dispatched only after the completion of all deterministic actions commenced by the processing of the current event. At this point, a stable state configuration has been reached

TABLE I
STATE MACHINE CONSTRAINTS

- If $v.kind = initial$ then $\#T_{outs}(v) = 1 \wedge \#T_{ins}(v) = 0 \wedge t_{first}.type = triggdless$.
- If $v.kind = final$ then $\#T_{outs}(v) = 0$.
- If $v.kind \notin \{state, comp, conc\}$ then $\forall t \in T_{outs}(v) : src(t) \neg = tgt(t)$.
- If $T_{auto} = \{t \in T_{outs} | \#events(t) = 0\}$, $T_{ng} = \{t \in T_{auto} | guard(t) = true \vee \nexists guard(t)\}$ then $\#T_{ng} <= 1$.
- $\#T^{+}_{ins}(v) > 0 \vee \#T_{outs}(v)^{+} > 0$.
- If $v.kind = comp$ then $\#subvertexes(v) > 0$.
- If $v.kind = conc$ then $\#regions(v) > 0 \wedge (\forall r \in regions(v) : \#subvertexes(r) > 0)$.
- $\#regions(sm) = 1$.
- If $v.kind = fork$ then $\#T_{ins}(v) > 0 \wedge \#T_{outs}(v) > 1 \wedge (\forall t \in T_{outs}(v) : t.type = triggdless \wedge ctner(tgt(t)).kind = conc)$.
- If $v.kind = join$ then $\#T_{ins} > 1 \wedge \#T_{outs}(v) = 1 \wedge (\forall t \in T_{ins}(v) : t.type = triggdless \wedge (\exists s \in ctner^{+}(src(t)), s.kind = conc)) \wedge head(T_{outs}).type = triggdless$.
- If $v.kind \in choice, junction$, then $\#T_{ins}(v) > 0 \wedge \#T_{outs}(v) > 1 \wedge (\exists! out \in T_{outs}(v) : out.type = gdless)$.
- If $v.kind \in enpoint, expoint$, then $ctner(v).kind \in \{comp, conc\} \wedge \#T_{ins}(v) > 0 \wedge \#T_{outs}(v) = 1 \wedge head(T_{outs}(v).type = triggdless)$.
- If $v.kind = history$ then $ctner(v).kind \in \{comp, conc\} \wedge (if v.kind = comp$ then $\exists! v \in ctner(v).subvertexes | v.kind = history) \wedge \#T_{ins} > 0$.

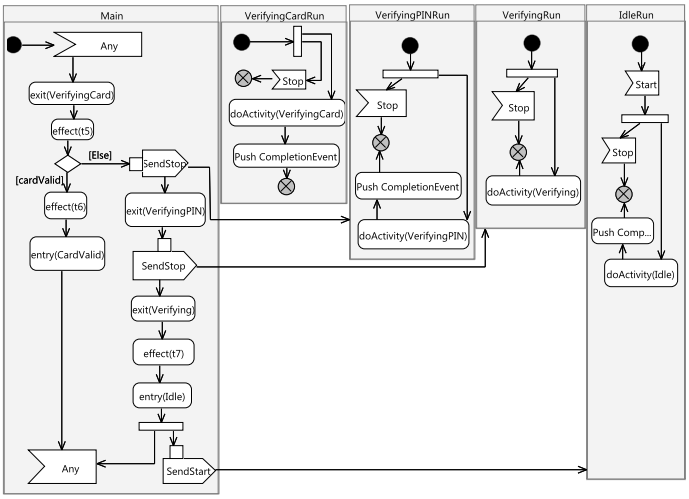

Fig. 3. Concurrency of the ATM when *doActivity* of *VerifyingCard* completes before that of *VerifyingPIN*

In the example, some of DAs are as followings: *effect(t2)*, *effect(t3)*, *effect(t4)*, *entry(Verifying)*, *entry(VerifyingCard)*, *entry(VerifyingPIN)* and non-deterministic actions (NDAs) as followings: *doActivity(Verifying)*, *doActivity(VerifyingCard)* and *doActivity(VerifyingPIN)*.

B. Thread-based design of generated code

Each NDA is run in parallel with the main thread which reads and dispatch events from the event queue. Each is associated with a thread which is initialized at the state machine initialization moment. The number of threads associated with NDAs is therefore equal to that of the NDAs. 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. An NDA is one of the followings:

- *doActivity* of each state if has. The number of *doActivity* $n_{do} = \#\{s \in V | \exists doActivity(s)\}$
- Sleep function associated with a *TimeEvent* which counts ticks and emits a *TimeEvent* once completes: $n_{te} = \#\{e \in E | e \text{ is a time event}\}$.
- Change detect function associated with a *ChangeEvent* which observes a variable or a boolean expression and pushes an event to the queue if changes happen: $n_{che} = \#\{e \in E | e \text{ is a change event}\}$.

Therefore, the concurrency has the number of initial threads $n_{threads} = n_{do} + n_{te} + n_{che}$ plus a main thread which reads events from the event queue, and sends start and stop signals to these initial threads.

Now we consider spontaneous threads which are created by *FORK* to run DAs, joined until and destroyed once DAs complete. The followings describe different types of DAs:

- Actions executed when entering/exiting an orthogonal region, which can be: execute a chain of transition effects contained by the region before entering a stable sub-state or exiting the region completely.
- Effects of transitions outgoing from a $fork$ and those incomings to a $join$.

The spontaneous threads follow a paradigm in which if a thread $parent$ creates a set of threads $children$, $parent$ must wait until $children$ complete their associate methods. These threads are created in one of the following cases:

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

C. Deadlock avoidance

Deadlock is one of the main issues in designing multi-thread applications, in which two competing actions wait for the other to finish. In our case,

V. Code generation pattern

A. Assumption

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

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

B. State transformation

Suppose that we want to generate a state machine sm whose states are listed by $lstates$. A common state interface $IState$ is created. The interface contains three methods, namely, *entry*, *exit*, and *doActivity* corresponding to three state actions, respectively. To preserve the hierarchy of composite states, the interface also has two attributes called *activeStates* and *previousStates* referring to active sub-states *actives* , previous active sub-states *previousStates* in case of the presence of history states, and a list of deferred event identifiers.

Each UML state is transformed into an instance of the interface associated with a state ID (which is a child element of an enumeration) inside the active class C. When initialization, each instance refers its methods to the actual methods implemented in C. In C++, this referring is done by using the powerful mechanism function pointer. In other object-oriented languages such as Java, this is done with anonymous subclasses of the interface. Listing 1 and **??** show the interface and its instances associated with the states of the state machine in C++ and Java, respectively, in which S0 is one of $lstates$. *NUM_STATES* is the number of states in the state machine. The actions of the states are implemented in the active class C and named depending on the name of the states. In the following sections, we only consider C++ as out *ActLang*. The discussion of other object-oriented languages are much similar since these share the same concepts,

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

```
typedef struct IState {
  int pres[2];   int actives[2];
  EventId defEvents[2];
  void (C::*entry)();  void (C::*exit)();  void (C::*doActivity)();
} IState;
class C {
private:
  IState states[NUM_STATES];
public:
  C() {
    states[S0_ID].entry = &C::S0_entry;
    defEvents[0] = EVENT_MAX; defEvents[1] = EVENT_MAX;
    ...
  }
  void S0_entry {...}
}
```

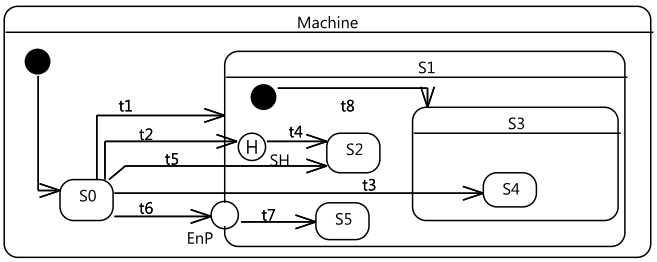

Fig. 4. Example illustrating different ways entering a composite state

Listing 2. Example code generated for doActivity

```
void doActivity(int stateId) {
  isStarts[stateId] = false;
  while(true) {
    mutex[stateId].lock();
    while(!isStarts[stateId]) {
      mutex[stateId].wait();
    }
    states[stateId].doActivity();
    isStarts[stateId] = false;
    mutex[stateId].unlock();
    if (!isStops[stateId]) {
      if (stateId == IDLE_ID || stateId == DISPENSEMONEY_ID ...) {
        pushCompletionEvent(stateId);
      }
    }
  }
}
```

C. Region transformation

Each region is transformed into an entering and an exiting method. While the region entering method controls how a region r is entered from an outside transition t, which is satisfied $src(t) \notin subvts(r)$, the exiting method exits completely a region by executing exit actions of sub-states from innermost to outermost.

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

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

All of the entering ways execute the entry action of the containing composite state $entry(ctner(r))$ after $effect(t)$. *doActivity(ctner(r))* is then signaled to be run in a waiting thread (see IV for thread-based concurrency). The action execution afterward is different from each way. To illustrate, we use an example as in Fig. 4 with *S1* as a target composite state. *t1* and *t3* are in the ways of 1 and 3, respectively, while *t2, t5* in the way 2.

The entering method associated with the region r of *S1* has a parameter $enter_mode$ indicating how actions should be executed. $enter_mode$ takes values depending the number of transitions coming to the composite state $S1$: $\#values(s) = \#\{v \in subvts(s) | v.kind = initial\} +$
$\#\{v \in subvts(s) | \exists t \in T_{ins}(v), src(t) \notin subvts(s)\} +$

$\#\{v \in subvts^+(s) \setminus subvts(s) | \exists t \in T_{ins}(v), src(t) \notin subvts^+(s)\}$. In this case these values are $\{DEFAULT = 0, SH\_MODE = 1, S2\_MODE = 2, S4\_MODE = 3\}$. Listing 3 shows the C++-like example code generated for r.

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

For each value in $values(s)$, the region of $S1$ is entered and executes different actions. By default, the active sub-state of the region is set before the execution of any effect associated with the initial transition starting from the pseudo initial state of the region. $S3$ is set as active sub-state of $S1$. Entering on a direct sub-state (*S2*) sets the active sub-state of *S1* directly to *S2*. In case of an indirect sub-state ($S4$), the entry action of the sub-state ($S3$) of $S1$ is executed before $S4$ is set as the active-sub state of $S3$ and the execution of $entry(S4)$. It is worth noting that after the execution of each entry, a start signal is sent to activate the sleeping thread associated with *doActivity* of the corresponding state (see **??** for thread-based concurrency design).

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

D. Event and transition transformation

1) Events: An event enumeration *EventId* is created whose children are event identifiers associated with events. Each event e) is also transformed into a method mtd_e in the context class C. Suppose $levents$ is the list of events which can be processed by the state machine sm. Besides the explicitly defined events of the state machines, $levents$ contains a special event called $CompletionEvent$. The latter is, following the UML specification, an implicit event triggering triggerless transitions. It is emitted when either *doActivity* of an atomic state finishes its execution or all regions of a composite state have reached to a final state. The other events are transformed as followings:

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

$CompletionEvent$ has the highest priority. Others are equal by default but their priority is configurable.

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

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

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

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

Listing 4. Generation process for an event

```
∀ item ∈ Lm(e)
  ∀s ∈ Map_e(item)
    T_s = {t ∈ T_trig(e) | src(t) = s}
    ∀t ∈ T_s
      GENERATE_STATE_EVENT_CHECK(s, t, e)
      GENERATE_GUARD(t)
      GENTRANS(s, t, tgt(t))
```

The procedure in Listing 4 describes how the generation process works with an event. It first finds the innermost active states which are able to react e by orderly looping over Lm_e. For each transition outgoing from

an innermost state, code for active states and deferral events, guard checking and transition code segments are generated by $GENERATE\_STATE\_EVENT\_CHECK$, $GENERATE\_GUARD(t)$ and *GENTRANS*, respectively. If the identifier of e is equal to one of the events listed in $defEvents$ of the corresponding state (not shown in this paper), it is deferred by putting it to a deferral event queue managed by the main thread, which also pushes the deferred events back to the main queue once one of the pending events is processed.

Generally, *GENTRANS* generates code for transitions between any vertexes satisfying the constraints described in Section III. Algorithm 1 shows how the transition code generation works. The generated code is bounded by the deferral events, active states, and guard checking.

Algorithm 1 Code generation for transition

Input: A source v_s, a target vertex v_t and a transition t
Output: Code generation for transition

```
1:  procedure genTrans(v_s, v_t, t)
2:      H_s ← v_s ∪ ctner^+(v_s)
3:      H_t ← v_t ∪ ctner^+(v_t)
4:      s_ex ∈ H_s, s_en ∈ H_t | ctner(s_ex) = ctner(s_en)
5:      //Generate IF-ELSE statements for junctions
6:      if s_ex is a state then
7:          for r ∈ regions(s_ex) do
8:              FORK(RegionExit(r))
9:          end for
10:         //Generate JOIN for threads created above
11:         //Generate sendStopSignal to s_ex
12:         exit(s_ex)
13:     end if
14:     if v_t.kind = join then
15:         for in ∈ Tins(v_t) do
16:             FORK(effect(in))
17:         end for
18:         //Generate JOIN for threads created above
19:     else
20:         effect(t)
21:     end if
22:     if s_en is a state then
23:         entry(s_en)
24:         //Generate sendStartSignal to s_en
25:         if s_en.kind ∈ {conp, conc} then
26:             for r ∈ regions(s_en) do
27:                 FORK(RegionEnter(r))
28:             end for
29:             //Generate JOIN for threads created above
30:         end if
31:     else
32:         //Generate for pseudo states by patterns
33:     end if
34: end procedure
```

In the first place, Algorithm 1 looks for the composite states s_{ex} and s_{en} at the highest level to be exited and entered, respectively, by using Algorithm **??** (line 2-4). If the transition t is part of a compound transition, which involves some $junction$s, IF-ELSE statements are generated first (as PSCS says $junction$ is evaluated before any action), as described by the following list. The composite state is exited by calling the associated exiting region methods (FORK and JOIN for orthogonal regions)and followed by the generated code of transition effects. Entering region methods are then called once the above code completes the execution. If the target v_t of the transition t is a pseudo state, the generation algorithm corresponding to the pseudo state type is called. These algorithms are shown as the below list.

- $join$: Use $GENTRANS$ for v's outgoing transition.
- $fork$: Use $FORK$ and $JOIN$ for each of outgoing transitions of v.
- $choice$: For each outgoing, an $IF-ELSE$ is generated for the guard of the outgoing together with code generated by $GENTRANS$ (see Listing 6).
- $junction$: As a static version $choice$, a $junction$ is transformed into an attribute $junc_attr$ and evaluated before any action executed in compound transitions (see Listing 6). 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 $pres$ of $IState$. Restoring the active states using the history is exampled as in Listing 3. The entering method is executed as default mode at the first time the corresponding composite state is entered (see Listing 3).
- *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.
- $enpoint$: If $enpoint$ has no outgoing transition, the corresponding composite state is entered by default. Otherwise said, $GENTRANS$ is called to generate code for the outgoing transition.
- $expoint$: The code for the unique transition outgoing from $expoint$ is generated by using $GENTRANS$.
- $terminate$: The code executes the exit action of the innermost active state, the effect of the transition and destroys the state machine object.

3) Example Code: Listing 5 shows a code segment generated for the processing of the event $verifyingPIN$. It first checks whether $Idle$ is the current active state, in which $activeStateID$ is the identifier of the current root active state. The $doActivity$ behavior of $Idle$ is then stopped upon receiving a stop signal (line 2). The effect of $t2$ is executed after the execution of $exit(Idle)$ (line 3-4). $effect(t3)$ and $effect(t4)$ are then concurrently executed by using $FORK$ and $JOIN$ (line 5-8) since the owning transitions outgo from a $fork$. The execution of $emtry(Verifying)$ (line 11) then follows the changing the root active state to $Verifying$ (line 10). $doActivity(Verifying)$ is triggered in its own thread upon receiving a start signal (line 11) followed by concurrently entering the two orthogonal regions of $Verifying$ with

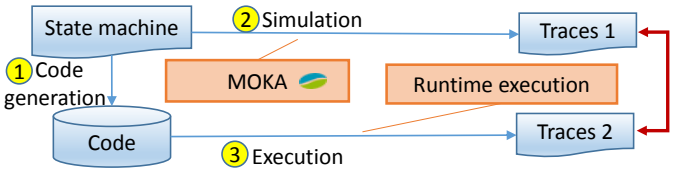

Fig. 5. Semantic conformance evaluation methodology

appropriate modes.

Listing 5. Example code generated for event *verifyingPIN*

```
if (activeStateID == IDLE_ID) {
  sendStopSignal(IDLE_ID);  exit_Idle();
  effect_t2();
  thread_t3 = FORK(effect_t3);  thread_t4 = FORK(effect_t4);
  JOIN(thread_t3);  JOIN(thread_t4);
  activeStateID = VERIFYING_ID;  entry_Verifying();
  sendStartSignal(VERIFYING_ID);
  th_r1 = FORK(R1Enter(VERIFYINGCARD_MODE));
  th_r2 = FORK(R2Enter(VERIFYINGPIN_MODE));
  JOIN(th_r1);  JOIN(th_r2);
}
```

Listing 6. Example code generated for *Join*1 and *Junction*1

```
if((states[VERIFYING_ID].actives[0]==CARDVALID_ID)
&&(states[VERIFYING_ID].actives[1]==PININCORRECT_ID)) {
  Junction1 = 0; //else outgoing transition of Junction1
  if (tries < maxTries) {Junction1 = 1;}
  FORK(R1Exit); FORK(R2Exit);
  //JOIN ...
  sendStopSignal(VERIFYING_ID); exit(VERIFYING_ID);
  FORK(effect_t11); FORK(effect_t13);
  //JOIN ...
  if (Junction1=1) {
    tries++;
    activeStateID = IDLE_ID;     entry(IDLE_ID);
    sendStartSignal(IDLE_ID);
  } else {
    cardValid = false;
    activeStateID = IDLE_ID;     sendStartSignal(IDLE_ID);
  }
}
```

VI. EXPERIMENTS

A. Semantic conformance of runtime execution

To evaluate the semantic conformance of runtime execution of generated code, we use a set of examples provided by Moka [7]. Moka is a model execution engine offering Precise Semantics of UML Composite Structures [6]. Fig. 5 shows our method. We first use our code generator to generate code (Step (1)) from the Moka example set. Step (2) simulates the examples by using Moka to extract the sequence (*SimTraces*) of observed traces including executed actions. The sequence (*RTTraces*) of traces is also obtained by the runtime execution of the code generated from the same state machine in a Step (3). The generated code is semantic-conformant if the sequences of traces are the same for both of the state machine and generated code [8]. The current version of Moka does not support simulation for *TimeEvent* and history pseudo states, we therefore leave experiments for *TimeEvent* as future work.

Within our scope as previously defined 30 examples of the Moka example set are tested. *SimTraces* and *RTTraces* for each case are the same. This indicates that, within our study scope, the runtime execution of code generated by our generator can produce traces semantically equivalent to those obtained via simulation.

VII. RELATED WORK

Implementation and code generation techniques for USMs are closely related to the forward engineering of our RTE.

Switch/if is the most intuitive technique implementing a "flat" state machine. The latter can be implemented by either using a scalar variable [9] and a method for each event or using two variables as the current active state and the incoming event used as the discriminators of an outer switch statement to select between states and an inner one/if statement, respectively. The double dimensional state table approach [10] uses one dimension represents states and the other one all possible events. The behavior code of these techniques is put in one file or class. This practice makes code cumbersome, complex, difficult to read and less explicit when the number of states grows or the state machine is hierarchical. Furthermore, these approaches requires every transition must be triggered by at least an event. This is obviously only applied to a small sub-set of USMs.

State pattern [11], [10] is an object-oriented way to implement flat state machines. Each state is represented as a class and each event as a method. Separation of states in classes makes the code more readable and maintainable. This pattern is extended in [12] to support hierarchical-concurrent USMs. However, the maintenance of the code generated by this approach is not trivial since it requires many small changes in different places.

Many tools, such as [5], [13], apply these approaches to generate code from USMs. Readers of this paper are recommended referring to [14] for a systematic survey on different tools and approaches generating code from USMs.

Double-dispatch (DD) pattern in [15] in which represent states and events as classes. Our generation approach relies on and extends this approach. The latter profits the polymorphism of object-oriented languages. However, DD does not deal with triggerless transitions and different event types supported by UML such as *CallEvent*, *TimeEvent* and *SignalEvent*. Furthermore, DD is not a code generation approach but an approach to manually implementing state machines.

VIII. CONCLUSION

ACKNOWLEDGMENT

This work is motivated by....

REFERENCES

[1] S. Li, L. D. Xu, and S. Zhao, "The internet of things: a survey," *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[2] B. M. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, 2006.

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

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

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

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

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

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

[9] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 1998, vol. 3. [Online]. Available: http://portal.acm.org/citation.cfm?id=1088874

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

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

[12] I. A. Niaz, J. Tanaka, and others, "Mapping UML statecharts to java code." in *IASTED Conf. on Software Engineering*, 2004, pp. 111–116. [Online]. Available: http://www.actapress.com/PDFViewer.aspx?paperId=16433

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

[14] E. Domínguez, B. Pérez, A. L. Rubio, and M. A. Zapata, "A systematic review of code generation proposals from state machine specifications," pp. 1045–1066, 2012.

[15] V. Spinke, "An object-oriented implementation of concurrent and hierarchical state machines," *Information and Software Technology*, vol. 55, no. 10, pp. 1726–1740, Oct. 2013.