UML State Machine: Towards formalizations and full 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*—The so-called model-driven engineering approach relies on two paradigms, abstraction and automation, recognized as very efficient for dealing with complexity of today system. UML state machine and their visual representations are much more suitable to describe logical behaviors of system entities than any equivalent text based description such as IF-THEN-ELSE or SWITH-CASE constructions. Although many industrial tools and research prototypes can generate executable code from such graphical language, their support only focus special cases of UML state machine, especially non-concurrent state machines. While UML state machine concepts such as concurrency, pseudo states such as history, fork, junction and time events are widely used by software architects, the code generation of the existing approaches does not take these concepts into account. To blur the boundaries between the modeling and coding worlds, a code generator should be able to generate code from all concepts of modeling languages with respect to the semantics described by the language specifications.**

To this end, this paper proposes an approach to generate code from all UML state machines with all of its concepts with respect to the Precise Semantics of UML, especially concurrent state machines which are based on multi-thread-based design.

I. Introduction

The so-called Model-Driven Engineering (MDE) approach relies on two paradigms, abstraction and automation [1]. It is recognized as very efficient for dealing with complexity of today system. Abstraction provides simplified and focused views of a system and requires adequate graphical modeling languages such as Unified Modeling Language (UML). Even, if the latter is not the silver bullet for all software related concerns, it provides better support than text-based solutions for some concerns such as architecture and logical behavior of application development. UML state machines (USMs) and their visual representations are much more suitable to describe logical behaviors of system entities than any equivalent text based descriptions. The gap from USMs to system implementation is reduced by the ability of automatically generating code from USMs [2], [3], [4], [3].

Ideally, a full model-centric approach is preferred by MDE community due to its advantages [**?**]. However, in industrial practice, there is significant reticence [**?**] to adopt it. On one hand, programmers prefer to use the more familiar textual programming language. On the other hand, software architects, working at higher levels of abstraction, tend to favor the use of models, and therefore prefer graphical languages for describing the architecture of the system. The code modified by programmers and the model are then inconsistent. Round-trip engineering (RTE) [5] is proposed to synchronize different software artifacts, model and code in this case [6]. RTE enables actors (software architect and programmers) to freely move between different representations [6] and stay efficient with their favorite working environment.

Unfortunately, current industrial tools such as for instance Enterprise Architect [7] and IBM Rhapsody[8] only support structural concepts for RTE such as those available from class diagrams and code. Compared to RTE of class diagrams and code, RTE of USMs and code is non-trivial. It requires a semantical analysis of the source code, code pattern detection and mapping patterns into USM elements. This is a hard task, since mainstream programming languages such as C++ and JAVA do not have a trivial mapping between USM elements and source code statements.

For software development, one may wonder whether this RTE is doable. That is, why do the industrial tools not support the propagation of source code modifications back to original state machines? Several possible reasons to this lack are (1) the gap between USMs and code, (2) not every source code modification can be reverse engineered back to the original model, and (3) the penalty of using transformation patterns facilitating the reverse engineering that may not be the most efficient (e.g. a slightly larger memory overhead).

This paper addresses the RTE of USMs and object-oriented programming languages such as C++ and JAVA. The main idea is to utilize transformation patterns from USMs to source code that aggregates code segments associated with a USM element into source code methods/classes rather than scatters these segments in different places. Therefore, the reverse direction of the RTE can easily statically analyze the generated code by using code pattern detection and maps the code segments back to USM elements. Specifically, in the forward direction, we extend the double dispatch pattern presented in [9]. Traceability information is stored during the transformations. We implemented a prototype supporting RTE of state-machine and C++ code, and conducted several experiments on different aspects of the RTE to verify the proposed approach. To the best of our knowledge, our implementation is the first tool supporting RTE of SM and code.

To sum up, our contribution is as followings:

- An approach to round-tripping USMs and object-oriented code.

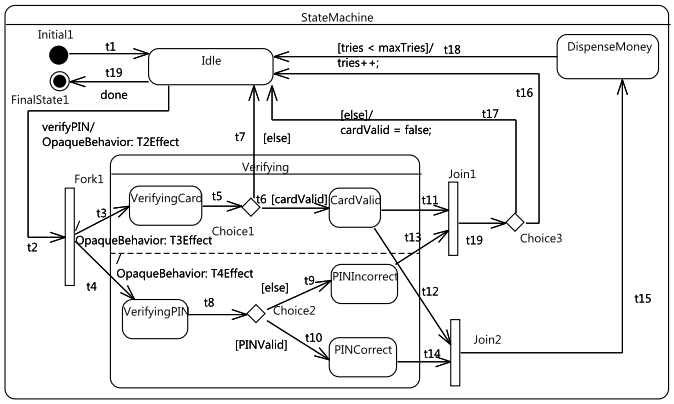

Fig. 1. ATM State machine example

- A first tooling prototype supporting RTE of USMs and C++ code.
- An evaluation of the proposed approach including:
  - An automatic evaluation of the proposed RTE approach with the prototype.
  - A comparison and collaboration of two software development practices including working at the model level and at the code level.
  - A lightweight evaluation of the semantic conformance of the runtime execution of generated code.

The remaining of this paper is organized as follows: Our proposed approach is detailed in Section **??**. The implementation of the prototype is described in Section **??**. Section **??** reports our results of experimenting with the implementation and our approach. Section **??** shows related work. The conclusion and future work are presented in Section **??**.

II. Motivating example

III. Formalization

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 ∈ V has a kind $v.kind \in$ *{initial, final, state, composite, concurrent, 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: $ctner(r) = s$.

Definition III.4. A UML state s is a vertex v where v.kind ∈ state, composite, concurrent. s has an $entry$, an $exit$ and a $doActivity$ action. A composite state cs contains one or more vertexes. We write subvertexes(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 ⊂ 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 contains data.

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 $v.kind = concurrent$, suppose $regions(v)$ is the region set contained by v. Given a transition t:

- $t.type = trigger$ if $\#events(t) > 0$.
- $t.type = triggerless$ if $\#events(t) = 0$.
- $t.type = guardless$ if $(guard(t) = true \vee \nexists guard(t)$.
- $t.type = triggerguardless$ 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} and T^+_{outs}:

$$T^+_{ins}(v) = \begin{cases} T_{ins}(v) & v.kind \notin \{composite, concurrent\} \\ T_{ins}(v) \cup \bigcup_{sub \in subvertexes(v)} T^+_{ins}(sub) & v.kind \in \{composite, concurrent\} \end{cases} \quad (1)$$

$$T^{+}_{outs}(v) = \begin{cases} T_{outs}(v) & v.kind \notin \{composite, concurrent\} \\ T_{outs}(v) \cup \bigcup_{sub \in subvertexes(v)} T^{+}_{outs}(sub) & v.kind \in \{composite, concurrent\} \end{cases} \quad (2)$$

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

$$ctner^{+}(v) = \begin{cases} sm & ctner(v) = sm \\ ctner(v) \cup ctner^{+}(ctner(v)) & otherwise \end{cases} \quad (3)$$

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 following constraints are hold:

- If $v.kind = initial$ then $\#T_{outs}(v) = 1 \wedge \#T_{ins}(v) = 0 \wedge t_{first}.type = triggerguardless$.
- If $v.kind = final$ then $\#T_{outs}(v) = 0$.
- If $v.kind \notin \{state, composite, concurrent\}$ 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 $v.kind = composite$ then $\#subvertexes(v) > 0$.
- If $v.kind = concurrent$ then $\#regions(v) > 0 \wedge (\forall r \in regions(v) : \#subvertexes(r) > 0)$.
- $\#regions(sm) = 1$.
- If $v.kind = fork$ then $\#T_{ins}(v) > 0 \wedge \#T_{outs}(v) > 1 \wedge (\forall t \in T_{outs}(v) : t.type = triggerguardless \wedge ctner(tgt(t)).kind = concurrent)$.
- If $v.kind = join$ then $\#T_{ins} > 1 \wedge \#T_{outs}(v) = 1 \wedge (\forall t \in T_{ins}(v) : t.type = triggerguardless \wedge (\exists s \in ctner^{+}(src(t)), s.kind = concurrent)) \wedge head(T_{outs}).type = triggerguardless$.
- If $v.kind \in choice, junction$, then $\#T_{ins}(v) > 0 \wedge \#T_{outs}(v) > 1 \wedge (\exists! out \in T_{outs}(v) : out.type = guardless)$.
- If $v.kind \in enpoint, expoint$, then $ctner(v).kind \in \{composite, concurrent\} \wedge \#T_{ins}(v) > 0 \wedge \#T_{outs}(v) = 1 \wedge head(T_{outs}(v).type = triggerguardless)$.
- If $v.kind = history$ then $ctner(v).kind \in \{composite, concurrent\} \wedge (if v.kind = compositethen \exists! v \in ctner(v).subvertexes | v.kind = history) \wedge \#T_{ins} > 0$.

Definition III.14. A transition graph τ is an acyclic directed graph ($\mathcal{T}_r$, $\mathcal{P}$, $\mathcal{T}$) where $\mathcal{P}$ is a set of vertexes, and $\mathcal{T}_r$ and $\mathcal{T}$ are sets of transitions. $\mathcal{T}_r$ is called the set of root transitions of the graph. Following conditions are satisfied:

- $\forall t \in \mathcal{T}_r, src(t)$ is a state.
- $\forall p \in \mathcal{P}, p.kind \notin \{state, composite, concurrent\}$.

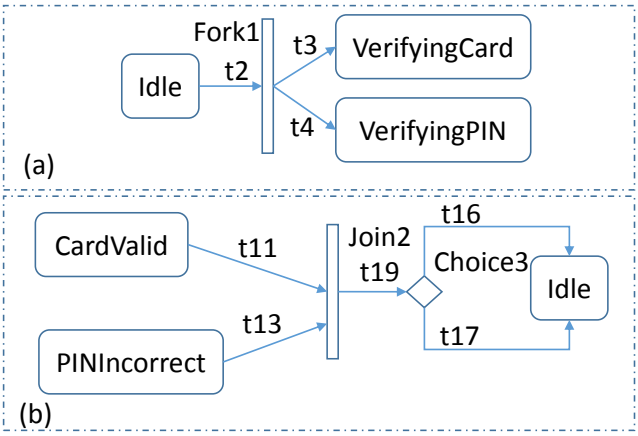

Fig. 2. Transition graphs

- $\forall t \in \mathcal{T}, src(t)$ is a pseudo state.

A traversal from the root transitions of a transition graph to a stable state configuration is a compound transition. A state machine can contain multiple transition graphs. Fig. 2 (a) and (b) show two transition graphs τ_1 and τ_2 of the ATM state machine, respectively, in which

$$\tau_1 = (\mathcal{T}_r, \mathcal{P}, \mathcal{T}) = (\{t2\}, \{Fork1\}, \{t3, t4\})$$

and

$$\tau_2 = (\{t11, t13\}, \{Join2, Choice3\}, \{t19, t16, t17\}).$$

Given a state, Algorithm 1 presents how to calculate transition graph set whose root transitions outgo from s.

For example, applying this algorithm to all states of the state machine example in 1, we can calculate other transition graphs which are:

$$\tau_3 = (\mathcal{S}_3, \mathcal{L}_3, \mathcal{P}_3, \mathcal{T}_3) = (\{VerifyingCard\}, \{Idle, CardValid\}, \{Choice1\}, \{t5, t6, t7\}),$$

$$\tau_4 = (\{VerifyingPIN\}, \{PINIncorrect, PINCorrect\}, \{Choice2\}, \{t8, t9, t10\}),$$

and

$$\tau_5 = (\{CardValid, PINCorrect\}, \{DispenseMoney\}, \{Join2\}, \{t12, t14, t15\}).$$

Definition III.15. 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.

Given sm is a flat state machine, whose vertex kinds are not in $\{composite, concurrent\}$, $\#Cfg(sm) = 1$ when the

Algorithm 1 Transition graphs calculation

Input: A state s of a state machine
Output: A set of transition graphs $\mathcal{GT}$

procedure CALCULATETRANSGRAPHS(s)
  $\mathcal{GT} \leftarrow \emptyset$
  for $out \in T_{outs}(s)$ **do**
    if $tgt(out)$ is not a state **then**
      $\tau \leftarrow (\mathcal{T}_r, \mathcal{P}, \mathcal{T}) = \{\emptyset, \emptyset, \emptyset\}$
      $\mathcal{P} \leftarrow \mathcal{P} \cup tgt(out)$
      $\mathcal{T}_r \leftarrow \mathcal{T}_r \cup out$
      if $tgt(out).kind = join$ **then**
        $\mathcal{T}_r \leftarrow \mathcal{T}_r \cup T_{ins}(tgt(out))$
      end if
      $nexts \leftarrow FINDTRS(tgt(out))$
      $\mathcal{T} \leftarrow \mathcal{T} \cup nexts$
      $\mathcal{GT} \leftarrow \mathcal{GT} \cup \{\tau\}$
    end if
  end for
end procedure

Input: A vertex v
Output: Transition paths starting from v to atomic states

procedure FINDTRS(v)
  $outs \leftarrow T_{outs}(v)$
  for $out \in T_{outs}(v)$ **do**
    if $tgt(out)$ is not a state **then**
      $outs \leftarrow$
$outs \cup FINDTRS(tgt(out))$
    else if then$tgt(out).kind \in composite, concurrent$
      for $sub \in subvertexes(tgt(out)), sub.kind = initial$ **do**
        $outs \leftarrow outs \cup FINDTRS(sub)$
      end for
    end if
  end forreturn $outs$
end procedure

system (active class C) is running. Cfg is also defined for composite/concurrent states. For each active composite/concurrent state cs, we write $subactives(cs)$ as a set of active sub-states of cs. If $cs.kind = composite$ then $\#subactives(cs) = 1$, otherwise $\#subactives(cs) > 1$. A transitive active set $subactives^+$ of an active state s is defined as following:

Therefore, we write:

$$Cfg(sm) = subactives^+(subactives(sm)) \quad (4)$$

A. Event dispatching

An external event incoming to the system or an internal event emitted by the system is dispatched by checking whether the innermost active states accept the event or not. Algorithm 2, in which $isAccepted(s, e)$ is $true$ if the event e is accepted by the state s, shows how an event should be dispatched by the system.

In Algorithm 2, if the active state of the state machine sm

Algorithm 2 Event dispatching

Input: An incomming event e, active state s_active of the running state machine sm
Output: Event is processed

procedure DISPATCHEVENT(s_active, e)
  $ret \leftarrow false$
  if $s\_active.kind \in composite, concurrent$ **then**
    for $sub \in subactives(s_active)$ **do**
      @concurrentExec
$(ret = DISPATCHEVENT(sub, e) \vee ret)$
    end for
  end if
  Wait for all cocurrent executions terminate
  if $(s\_active.kind = state) \vee !ret$ **then**
    if $isAccepted(s\_active)$ **then**
      $ret \leftarrow true$
      $processEvent(s\_active, e)$
    end if
  end ifreturn ret
end procedure

is atomic, a $processEvent$ procedure is executed to transition the active state to another state. Otherwise said, the active state delegates the event processing to its active sub-state. If the event is not consumed by the active sub-state, the processing of the former is delegated back to the parent state of the latter. In the following section, we show how to change the active state of the state machine.

B. Event processing semantics

There are three ways of transitioning from one state to another state including (1) direct transitioning in which the source state is directly connected to the target state by a transition, and (2) indirect transitioning in which multiple transitions and pseudo states in a transition graph are involved.

1) Direct transition: An active state can be changed from a source state to a target state either within the same region or not. In the first case, the transition t connecting the two states is in the same region. This case is simply that the source state ends its activity and all of its active sub-states. In the second case, given a source state s_{src}, a target state s_{tgt}, and the transition t connecting the two states, the following sub-cases are differentiated:

- $s_{src} = ctner(s_{tgt}) \vee s_{src} \in ctner^+(s_{tgt})$
- $s_{tgt} = ctner(s_{src}) \vee s_{tgt} \in ctner^+(s_{src})$
- $s_{src} = s_{tgt}$
- $\exists s^*_{src} \in ctner^+(s_{src}) \wedge \exists s^*_{tgt} \in ctner^+(s_{tgt}) : ctner(s^*_{src}) = ctner(s^*_{tgt})$.

The first and second sub-cases are occurred when the transition lies between a composite state and one of its transitive sub-states. The transition execution depends on the kind of t which is either local or external. The third sub-case is a special case in which only the transition effect is executed. Algorithm 3 shows the execution order of these three sub-cases.

2) Indirect transition:

$$subactives^{+}(s) = \begin{cases} s & s.kind \notin \{composite, concurrent\} \\ \bigcup_{sub \in subactives(s)} subactives^{+}(sub) & s.kind \in \{composite, concurrent\} \end{cases} \quad (4)$$

Algorithm 3 Transition Execution 1

Input: A transition t with its source and target states as s_{src} and s_{tgt}
Output: Transition is correctly executed

```
1: procedure EXECUTETRANS(t, s_src, s_tgt)
2:     ret ← false
3:     if s_active.kind ∈ composite, concurrent then
4:         for sub ∈ subactives(s_active) do
5:             @concurrentExec
6: (ret = DISPATCHEVENT(sub, e) ∨ ret)
7:         end for
8:     end if
9:     Wait for all cocurrent executions terminate
10:    if (s_active.kind = state)∨!ret then
11:        if isAccepted(s_active) then
12:            ret ← true
13:            processEvent(s_active, e)
14:        end if
15:    end ifreturn ret
16: end procedure
```

IV. Code generation

A. Assumption

To give the formalization of the code generation, we assume that we want to generate from the state machine to an object oriented programming language $ActLang$. Assuming that our code generator contains primitive functions supporting for generating $ActLang$ as following:

- $genClass(n, generals, itfs)$ creates a class with its name, parent class set, and implemented interfaces as *n*, *generals*, and *itfs*.
- $genMtd(n, c, type, params)$ creates a method m with its name as n inside the class c, its return type as $type$, and $params$ as its parameter set.
- $genAttr(n, c, type, multiplicity)$ creates an attribute named n in the class c and typed by $type$. The create attribute is an array if $multiplicity > 1$, otherwise a simple attribute.
- $genEnum(n)$ and $genEnumLit(enum, n)$ create an enumeration and its enumeration literal, respectively.
- $genBody(m, body)$ adds a body to a method. The body is a string which contains a list of statements.
- $createParalle(t, seg)$ generates a mechanism which allows the segment code seg run in a thread t. Similarly, $genWait(t), genJoin(t)$.
- $genMutex(size)$ creates an array of mutexes with *size* as the number of items of the array.
- $synchronize(seg)$ generates a mechanism which allows the segment code seg run safely (can be either based on *POSIX pthread* or *Java synchronize* mechanism).
- $toString(stts)$ is used to convert a list of statements $stts$ into a readable string which can be add to a method as its body.
- Concatenation of two strings $str1$ and $str2$ is concisely described as $str1 + str2$.
- *WHILE*, *FOR IF*, *ELSE* are symbols representing while and for loops, if and else statements.
- *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. Code generation algorithm

1) 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 2 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 {
  IState** previousStates;
  IState** actives;
  EventId* defEvents;
  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;
    ...
  }
  void S0_entry {...}
}
```

Listing 2. IState interface and annonymous sub-classes in Java

```
public interface IState {
  public IState[] previousStates;
  public IState[] actives;
  public EventId defEvents;
  public void entry();
  public void exit();
  public void doActivity();
}
class C {
private IState states[NUM_STATES];
public C() {
  states[S0_ID] = new IState() {
    public void entry() {
      S0_entry();
    }
    ...
  }
}
public void S0_entry() {...}
}
```

The procedure to generate the code for states is shown in Listing 3. It first creates the state interface $IState$ (in C++, it is either a class or a struct). The array attribute is then created with the number of states as its size. Each state is also associated with a state ID which is a child of an enumeration. Finally, the constructor of C is created to initialize and make methods of the attribute instances refer to *entry/exit/doActivity* action methods of C. The implementation of action methods in the context class C is similar to the delegation pattern proposed by the authors in [10] but dramatically decreases the memory consumption since only one common interface for all states is created instead of a class for each state in [10].

Listing 3. Procedure to create code for states

```
IState = genClass('IState', ∅, ∅);
stateIdEnum = genEnum('StateIdEnum');
foreach s in lstates
  genEnumLit(stateIdEnum, s.name + '_ID');
  mtd = genMtd(s.name + '_entry', C,
                        null, null);
  genBody(mtd, toString(entry(s)));
  ...
genEnumLit(stateIdEnum, 'NUM_STATES');
genAttr('states', C, IState, NUM_STATES);
genMtd(C.name, C, null, null);
```

2) Region transformation:

3) Event transformation: An event enumeration *EventId* is created whose children are event identifiers associated with events. Each event is also transformed into a method 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 orthogonal regions of a composite state have reached to a final state.

UML defines five types of events including *CallEvent*, *SignalEvent*, *TimeEvent*, *ChangeEvent*, and *Any*. A transition triggered by an *Any* event is meant to be fired by any of the other events. 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:

$$dp(v) = \begin{cases} 1 & v \text{ is a root vertex} \\ dp(ctner(v)) + 1 & otherwise \end{cases} \quad (5)$$

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

The procedure in Listing 4 describes how to generate the body of the method associated with an event. It generates the code checking for active states respecting the UML semantics in which the innermost states process the incoming event first. To do this, it first looks in the source state list $S_{trig(e)}$ for the innermost states that accept the event triggering its outgoing transitions. If these found states are children of a concurrent state, $genStateCheck$ generates the checking codes run in parallel, which will be described later in IV-B4. Otherwise said, sequential code is generated.

Listing 4. Procedure to create code event processing

```
for item in Lm(e)
  if (item.kind = conc)
    for s in Map_e(item)
      genStateCheck()
  else
    for s in Map_e(item)
      genStateCheckWithElse
```

4) Thread-based Concurrency:

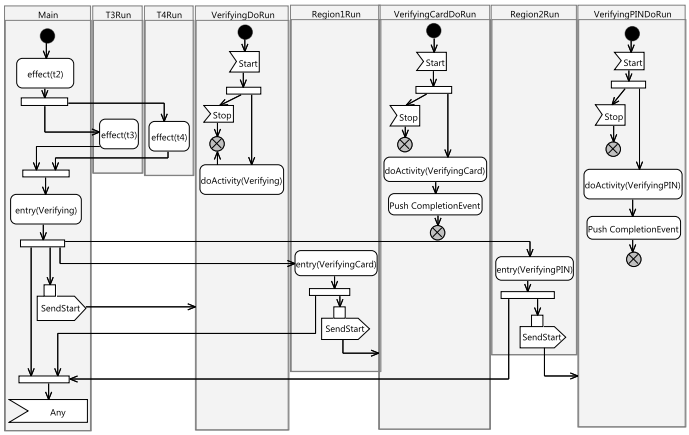

Fig. 3. Concurrency of the ATM when receiving the *verifyingPIN* event

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. 3 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 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. 4, 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. 4) 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

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* $n_{do} = \#\{s \in V | \exists doActivity(s)\}$

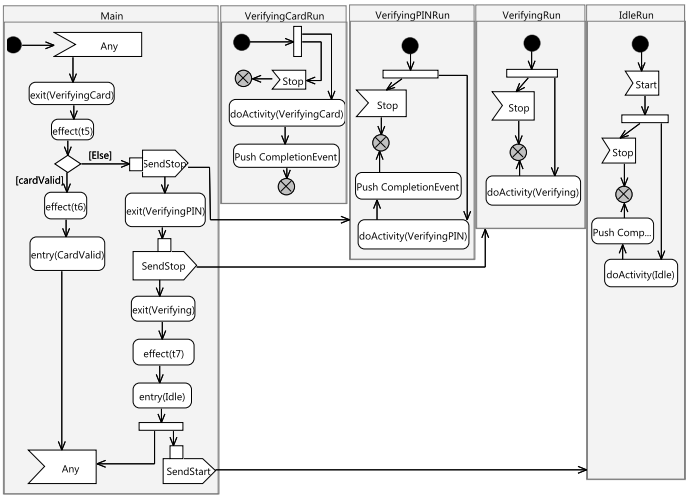

Fig. 4. Concurrency of the ATM when *doActivity* of *VerifyingCard* completes before that of *VerifyingPIN*

- 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 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: $n_{regionthreads} = \#\{r \in \mathcal{R} | ctner(r).kind = concurrent\}$
- Effects of transitions outgoing from a $fork$ and those incomings to a $join$:
  $\mathcal{J} = \{v \in V | v.kind = join\}$
  $\mathcal{F} = \{v \in V | v.kind = fork\}$

$$n_{FJ\_threads} = \sum_{v \in \mathcal{F}} \#T_{outs}(v) + \sum_{v \in \mathcal{J}} \#T_{ins}(v)$$

.

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) Example of generated code:

V. EXPERIMENTS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

VI. CONCLUSION

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

ACKNOWLEDGMENT

This work is motivated by....

REFERENCES

[1] 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.

[2] 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

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

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

[5] T. Hettel, M. Lawley, and K. Raymond, "Model synchronisation: Definitions for round-trip engineering," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5063 LNCS, 2008, pp. 31–45.

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

[7] SparxSystems, “Enterprise Architect,” Sep. 2016. [Online]. Available: http://www.sparxsystems.eu/start/home/
[8] IBM, “Ibm Rhapsody.” [Online]. Available: http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/
[9] 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.
[10] I. Niaz and J. Tanaka, “Mapping UML statecharts to java code.” *IASTED Conf. on Software Engineering*, pp. 111–116, 2004.