Interaction Components Between Components based on a Middleware

Van Cam Pham, Ansgar Radermacher

CEA, LIST, Laboratory of Model driven engineering for embedded systems,
Point Courrier 174, Gif-sur-Yvette, F-91191 France
name.surname@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.

1 Introduction

The so-called Model-Driven Engineering (MDE) approach relies on two paradigms, abstraction and automation [**?**]. 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 [**?**,**?**,**?**,**?**].

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) [?] is proposed to synchronize different software artifacts, model and code in this case [?]. RTE enables actors (software architect and programmers) to freely move between different representations [?] and stay efficient with their favorite working environment.

Unfortunately, current industrial tools such as for instance Enterprise Architect [?] and IBM Rhapsody[?] 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 [?]. 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.
- 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 **??**.

2 Formalization

Definition 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 2. *A UML vertex* $v \in V$ *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 3. *A region* $r \in \mathcal{R}$ *is composed of one or several vertexes, and contained by a state* s*:* $ctner(r) = s$.

Definition 4. *A UML state* s *is a vertex* v *where* $v.kind \in$ *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 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 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 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 8. *A Signal sig contains data.*

Definition 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 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 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 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}:

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

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

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

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

In the example in Fig. **??**, we have: lCr 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 14. *A transition graph τ is an acyclic directed graph ($\mathcal{T}$, $\mathcal{P}$, $\mathcal{T}$) where $\mathcal{S}, \mathcal{L}, \mathcal{P}$ are sets of vertexes and $\mathcal{T}$ is a set of transitions whose source and target vertexes belong to $\mathcal{S} \cup \mathcal{L} \cup \mathcal{P}$. And following conditions are satisfied:*

- $\forall s \in \mathcal{S} \cup \mathcal{L}$:
  - *If $s \in \mathcal{S}$ then s is a state.*
  - *Otherwise s is a state or $s.kind = final$.*
- $\forall p \in \mathcal{P}, p.kind \notin \{state, composite, concurrent\}$.

$\mathcal{S}$ and $\mathcal{L}$ are sets of source and reachable target states of τ, respectively.

A transition graph is composed of one or multiple compound transitions, each of which consists of one/multiple transitions starting from states/pseudo states/pseudo states to pseudo states/states/pseudo states. A state machine can contain multiple transition graphs. Fig. **??** (a) and (b) show two transition graphs τ_1 and τ_2 of the ATM state machine, respectively, in which lCr $\tau_1 = (\mathcal{S}_1, \mathcal{L}_1, \mathcal{P}_1, \mathcal{T}_1) = (\{Idle\}, \{VerifyingCard, VerifyingPIN\}, \{Fork1\}, \{t2, t3, t4\})$ and lCr $\tau_2 = (\mathcal{S}_2, \mathcal{L}_2, \mathcal{P}_2, \mathcal{T}_2) = (\{CardValid, PINIncorrect\}, \{Idle\}, \{Join2, Choice3\}, \{t11, t13, t19, t16, t17\})$.

Definition 15. *A compound transition t_{cp} is a virtual path which starts from one or multiple UML state and ends on one or multiple UML state. A compound transition is specified by a triple {srcs(t_{cp}), trc(t_{cp}), tgts(t_{cp})}, in which source part srcs(t_{cp}) consists of one or multiple states, transition part trc(t_{cp}) consists of multiple transitions, and target part tgts(t_{cp}) consists of one or multiple states.*

[clip, trim=2.0cm 6cm 17.5cm 3cm, width=]figures/transitionGraph.pdf

Fig. 1. Transition graphs

Given a state, Algorithm 1 presents how to calculate transition graphs whose source t_{cp} whose source part contains only a state s.

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**
      $\tau \leftarrow (\mathcal{S}, \mathcal{L}, \mathcal{P}, \mathcal{T}) = \{\emptyset, \emptyset, \emptyset, \emptyset\}$
      $\mathcal{P} \leftarrow \mathcal{P} \cup tgt(out)$
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$
      $\mathcal{T} \leftarrow \mathcal{T} \cup out$
      if $tgt(out).kind = join$ **then**
        $ins \leftarrow$
$\{i \in T_{ins}(tgt(out))|ctner(src(i)) = ctner(s)\}$
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{src(i)|i \in ins\}$
        $\mathcal{T} \leftarrow \mathcal{T} \cup ins$
      $nexts \leftarrow FINDTRANS(tgt(out))$
      $\mathcal{T} \leftarrow \mathcal{T} \cup nexts$
      H $\leftarrow$
$\{tgt(t)|t \in nexts \wedge tgt(t).kind = history\}$
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{src(t)|t \in nexts\} \cup H$
      $\mathcal{L} \leftarrow \mathcal{L} \cup \{tgt(t)|t \in nexts \wedge tgt(t)$ is state $\}$
      $\mathcal{GT} \leftarrow \mathcal{GT} \cup \{\tau\}$

Input: A vertex v
Output: Transition paths starting from v and ending on a state

procedure FINDTRANS(v)
  $nextTrans \leftarrow T_{outs}(v)$
  for $out \in T_{outs}(v)$ **do**
    if $tgt(out)$ is not a state **then**
      $nextTrans \leftarrow$
$nextTrans \cup FINDTRANS(tgt(out))$
  return $nextTrans$

For example, applying this algorithm to all states of the state machine example in **??**, we can calculate other transition graphs which are: lCr $\tau_3 = (\mathcal{S}_3, \mathcal{L}_3, \mathcal{P}_3, \mathcal{T}_3) = (\{VerifyingCard\}, \{Idle, CardValid\}, \{Choice1\}, \{t5, t6, t7\})$, lCr $\tau_4 = (\{VerifyingPIN\}, \{PINIncorrect, PINCorrect\}, \{Choice2\}, \{t8, t9, t10\})$, and lCr $\tau_5 = (\{CardValid, PINCorrect\}, \{DispenseMoney\}, \{Join2\}, \{t12, t14, t15\})$.

Definition 16. *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.*
- $\forall p \in \mathcal{P}, p.kind \notin \{state, composite, concurrent\}$.
- $\forall t \in \mathcal{T}, src(t)$ *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. **??** (a) and (b) show two transition graphs τ_1 and τ_2 of the ATM state machine, respectively, in which lCr $\tau_1 = (\mathcal{T}_r, \mathcal{P}, \mathcal{T}) = (\{t2\}, \{Fork1\}, \{t3, t4\})$ and lCr $\tau_2 = (\{t11, t13\}, \{Join2, Choice3\}, \{t19, t16, t17\})$.

[clip, trim=2.0cm 6cm 17.5cm 3cm, width=]figures/transitionGraph.pdf

Fig. 2. Transition graphs

Definition 17. *A compound transition t_{cp} is a virtual path which starts from one or multiple UML state and ends on one or multiple UML state. A compound transition is specified by a triple {srcs(t_{cp}), trc(t_{cp}), tgts(t_{cp})}, in which source part srcs(t_{cp}) consists of one or multiple states, transition part trc(t_{cp}) consists of multiple transitions, and target part tgts(t_{cp}) consists of one or multiple states.*

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 **??**, we can calculate other transition graphs which are: lCr $\tau_3 = (\mathcal{S}_3, \mathcal{L}_3, \mathcal{P}_3, \mathcal{T}_3) = (\{VerifyingCard\}, \{Idle, CardValid\}, \{Choice1\}, \{t5, t6, t7\})$, lCr $\tau_4 = (\{VerifyingPIN\}, \{PINIncorrect, PINCorrect\}, \{Choice2\}, \{t8, t9, t10\})$, and lCr $\tau_5 = (\{CardValid, PINCorrect\}, \{DispenseMoney\}, \{Join2\}, \{t12, t14, t15\})$.

Definition 18. *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 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:

$$Cfg(sm) = subactives^+(subactives(sm)) \tag{1}$$

Algorithm 2 Transition graphs calculation

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

```
1: procedure CALCULATETRANSGRAPHS(s)
2:     GT ← ∅
3:     for out ∈ T_outs(s) do
4:         if tgt(out) is not a state then
5:             τ ← (T_r, P, T) = {∅, ∅, ∅}
6:             P ← P ∪ tgt(out)
7:             T_r ← T_r ∪ out
8:             if tgt(out).kind = join then
9:                 T_r ← T_r ∪ T_ins(tgt(out))
10:            nexts ← FINDTRS(tgt(out))
11:            T ← T ∪ nexts
12:            GT ← GT ∪ {τ}
```

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

```
13: procedure FINDTRS(v)
14:     outs ← T_outs(v)
15:     for out ∈ T_outs(v) do
16:         if tgt(out) is not a state then
17:             outs ←
18: outs ∪ FINDTRS(tgt(out))
19:         else if  thentgt(out).kind ∈ composite, concurrent
20:             for sub ∈ subvertexes(tgt(out)), sub.kind = initial do
21:                 outs ← outs ∪ FINDTRS(sub)
        return outs
```

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

2.1 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.

Algorithm 3 Event dispatching

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

```
1: procedure DISPATCHEVENT(s_active, e)
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:     Wait for all cocurrent executions terminate
8:     if (s_active.kind = state)∨!ret then
9:         if isAccepted(s_active) then
10:            ret ← true
11:            processEvent(s_active, e)
       return ret
```

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

2.2 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.

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 **??** shows the execution order of these three sub-cases.

Algorithm 4 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:     Wait for all cocurrent executions terminate
8:     if (s_active.kind = state)∨!ret then
9:         if isAccepted(s_active) then
10:            ret ← true
11:            processEvent(s_active, e)
       return ret
```

Indirect transition

3 experiments

4 Conclusion

[1]

References