

From UML State Machine to code and back again!

Van Cam Pham, Ansgar Radermacher, Sébastien Gérard
CEA-List, Laboratory of Model Driven Engineering for Embedded Systems (LISE), Gif-sur-Yvette,
France
first-name.last-name@cea.fr

ABSTRACT

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 SWITCH-CASE constructions. Although many industrial tools and research prototypes can generate executable code from such graphical language, generated code could be manually modified by programmers. After code modifications, round-trip engineering is needed to make the model and code consistent, which is a critical aspect to meet quality and performance constraint required from project manager today. Unfortunately, current UML tools only support structural concepts for round-trip engineering such as those available from class diagrams. In this paper, we address the round-trip engineering of UML state-machine and its related generated code. We propose a round-trip engineering approach consisting of a forward process which generates code by using transformation patterns, and a backward process which is based on code pattern detection to update the original state machine model from the modified generated code. We implemented a prototype and conducted several experiments on different aspects of the round-trip engineering to verify the proposed approach.

CCS Concepts

•Computer systems organization → Embedded systems; *Redundancy*; Robotics; •Networks → Network reliability;

Keywords

Round-trip engineering, code generation, state machine, UML, C++, reverse engineering

1. INTRODUCTION

The so-called Model-Driven Engineering (MDE) approach relies on two paradigms, abstraction and automation [24]. 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) [33]. Even, if the latter is not the silver bullet for all software related concerns, it provides hence better support than text-based solutions for some concerns such as architecture and logical behavior of application development. 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 SWITCH-CASE [8] constructions. The gap from UML state machines to system implementation is reduced by the ability of automatically generating code from UML state-machines [4, 8, 12, 12, 13, 31].

However, on one hand maintaining the code generated from existing approaches is non-trivial [26]. On the other hand our observation is that it is very difficult to come up with formalizations that yield such elegant generation solutions [19]. In other words, generated code must be manually modified to build fully operational applications. The modified generated and the model are then inconsistent. Round-trip engineering (RTE) [6, 18, 35] is proposed to synchronize different software artifacts, model and code in this case [30], thus enabling actors (software architect and programmers) to freely move between different representations [30] and stay efficient with their favorite working environment.

Unfortunately, current industrial tools such as for instance Enterprise Architect [32], Visual Paradigm [29], and AndroMDA [1] only support structural concepts for RTE such as those available from class diagrams and code. RTE of state machines and code is a hard task, as it is essentially an RTE. Compared to RTE of class diagrams and code, RTE of state-machines requires a semantics analysis of source code, code pattern detection and mapping these patterns into UML state-machine concepts. The reason to this hard is that in mainstream programming languages such as C++ and JAVA, there are not equivalents between state-machines and source code statements.

For software development, one may wonder if this RTE is doable. That is, why do the industrial tools not support the propagation of source code modifications back to original state machines? One reason is the gap between state-machines and code, and the other reason could be that in the mind of these tools' vendors, users always make changes to models rather than to code. Generated code, in these tools, is therefore not supposed to be changed directly.

This paper addresses the RTE of UML state-machines and object-oriented programming languages such as C++ and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE '16 Singapore

© 2016 ACM. ISBN 123-4567-24-567/08/06...\$15.00

DOI: 10.475/123_4

JAVA. The forward engineering of the approach takes as input a state-machine and executes two transformations. The first is UML to UML by utilizing several transformation patterns such as the double-dispatch approach presented in [34] and the second is a generation of code from the transformed UML. Traceability information is stored, during the transformations. In the backward direction, a verification is executed by the code pattern detection to verify the correctness of the code before the backward process taking as input the modified generated code, the UML classes, the original state-machine and mapping information together merges changes from code to the state-machine. 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 UML state-machines and object-oriented code.
- A first tooling prototype supporting RTE of UML state-machines and C++ code.
- An evaluation of the proposed approach including:
 - An automatic evaluation of RTE correctness of the proposed approach with the prototype.
 - A complexity analysis of the approach and performance evaluation.
 - 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: Section 2 shows related work. Our proposed approach is detailed in Section 3. The implementation of the prototype is described in Section 4. Section 5 reports our results of experimenting with the implementation and our approach. The conclusion and future work are presented in Section 6.

2. RELATED WORK

Two main topics directly related to our study are identified. One is the implementation techniques and code generation for UML SMs and the other one is RTRIP.

2.1 Implementation and code generation for UML SMs

Main approaches including switch/if, state table and state pattern are investigated.

Switch/if is the most intuitive technique implementing a "flat" state machine. Two types of switch/if are supported. The first one uses a scalar variable representing the current active state [8]. A method for each event processes the variable as a discriminator in switch/if statement. The second one uses a double nested switch/if and has two variables to represent the current active state and the event to be processed [12]. The latter are used as the discriminators of an outer switch statement to select between states and an inner one/if statement to decide how the event should be

processed. The behavior code of the two types 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, the first approach lets the code scatter in different places. Therefore, maintaining or modifying such code of complex systems is very difficult.

In [12, 13] the authors also propose a double dimensional state table in which one dimension represents states and the other one all possible events. Each cell of the table is associated with a function pointer meaning that the state associated with a dimension index of the cell is triggered by the event associated with the other dimension. This technique is efficient for flat and simple state machines. As the switch/if technique, this approach gets cumbersome and non-trivial to maintain since states and events represented by indexes of the table are not explicit. Furthermore, this approach requires every transition must be triggered by at least an event. This is obviously only applied to a very small sub-set of UML state machines.

State pattern [4, 12, 31] is an object-oriented way to implement state machines. Each state is represented as a class and each event as a method. The event processing is executed by a delegation from the state machine context class to sub-states. Separation of states in classes makes the code more readable and maintainable. Unfortunately, this technique only supports flat state machines. The authors in [26] extend this pattern to overcome its limitations. However, the maintenance of the code generated or implemented by this approach is not trivial since it requires a lot of code to write and many small changes in different places. This is critically impractical when dealing with large state machines. Furthermore, similar to the state table, this approach also poses the requirement of having at least one event for transition.

Double-dispatch pattern is proposed in [34] as a new technique to implement state machines. States and events are represented as classes. Our generation approach relies on this approach. The latter provides some 1-1 mappings from state machines to object-oriented code and the implementation technique is not dependent on a specific programming language. However, the approach does not deal with triggerless transitions and different event types supported by UML such as *CallEvent*, *TimeEvent* and *SignalEvent*. Furthermore, the proposed approach is not a code generation approach but an approach to manually implementing state machines.

Readers of this paper are recommended referring to c[11] for a systematic survey on different approaches generating code from state machine/state charts.

2.2 Round-trip engineering

There have been a large number of approaches to RTRIP of model and code, and models themselves. This paper only shows the most related approaches.

RTE of models is tackled by many approaches categorized from total, injective, bi-directional to partial non-injective transformations [18]. Van Paesschen et al [28] proposed a RTRIP for one-to-one mappings meaning that one element of a source model is associated with exactly one element in the target model. Triple Graph Grammar (TGG) is introduced in [16, 22] to allow to synchronize between a source and a target pattern.

Foster et al [14] uses the concepts of lenses to define RTRIP. A lens consists of a *get* function, which produces a target artifact (model/code) from a source artifact (model/code), and a *putback* function, which takes as input the modified target and old source artifact to update the source artifact. The paper also defines two RTRIP laws that a RTRIP must satisfy. These laws are the base for the evaluation of our approach that will be presented in Section 5.

Partial RTRIP and *protected regions* are introduced in [15] and [21] to preserve code modifications which cannot be reflected to models. This approach separates the code part which is generated from models and the part which is allowed to be changed by programmers. This form of RTRIP is unidirectional only and does not support iterative development [20].

The authors in [5] propose a syntactic synchronization technique for domain-specific modeling languages (DSML) and code. The approach uses an Abstract Syntax Tree (AST) metal-model to model source. Changes in code detected by using an algorithm proposed in [10] to compare the AST instance of the current code with the last synchronized one are merged to the instance of DSML. However, an AST is very low level and it is not clear to have mappings from DSML instances to AST instances. Furthermore, although there is an example for illustrating the technique, a systematic evaluation of the approach is not presented to show its scalability.

3. APPROACH

This section presents our RTRIP approach. At first, it sketches UML SM concepts supported by this study. The outline and the detail of the approach are presented afterward.

3.1 Scope

A SM has a number of possible states and well-defined conditional transitions between states. A state is either an atomic state, a hierarchical state that is composed of sub-states and has at most one active sub-state at a certain time, or a concurrent state which could have several active sub-states at the same time. Only one of the inner states of the SM can be active at a time. The active state of the machine can be changed to another state triggered by external or internal events. An action can also be activated by the trigger in transitioning from one state to another one. A state can have associated actions such as *entry/exit/doActivity* executed in the running of the SM.

A composite state can have one or several entry/exit points. An entry/exit point has exactly one outgoing transition. The latter has no triggering event and guard. The transition outgoing from an entry point ends on a sub-state or an entry point of one of the sub-states of the composite state. The transition outgoing from an exit point ends on either an exit point of the parent state or a state/an entry point in the same region. A concurrent state is entered by either an incoming transition ending on the border or several incoming transitions outgoing from a *fork* and ending on sub-states of the contained regions.

3.2 Approach outline

Our RTRIP approach is based on the double-dispatch pattern presented in [34] for mapping from UML SM to UML classes and traceability-mapping management in RTRIP. Fig.

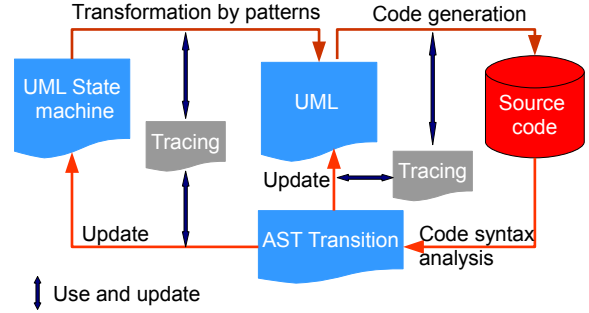


Figure 1: Outline of State machine and code RTE

1 shows the outline of our RTRIP approach consisting of a forward and a backward process. In the forward process, a SM is transformed into UML classes which contain attributes, operations and a block of text as method body associated with each implemented operation. The transformation uses several patterns which will be presented later. A tracing information table is created in the transformation to be used in the backward direction of the RTRIP. The UML classes are then used as the input of a classical code generator to create source code. This generation step also puts mapping from UML classes to object-oriented source code in a second mapping table.

In the backward (reverse) direction, when the source code is modified, a verification process checks whether the modified code conforms to the SM semantics (see Section C for the detail of the verification). The backward transformation takes as input the tracing tables, the created UML classes and the SM to update these models sequentially. While the forward process can generate code from hierarchical and concurrent SMs, the backward one only works for hierarchical machines excluding some pseudo-states which are *history*, *join*, *fork*, *choice* and *junction*. These features are in future work.

3.3 From UML state machine to UML classes

This section describes the forward process. The latter consists of transforming UML SM elements (see 3.3.1, 3.3.2, 3.3.3) into the intermediate model, storing tracing information (see 3.3.4) and code generation (see 3.3.5) from the intermediate model.

3.3.1 Transformation of states

This sub-section describes the transformation of states to the intermediate model.

This paper considers a component as a UML class called context class. Each state of the SM is transformed into a UML class in an intermediate model. Each UML class representing a state inherits from a base state class. The latter defines a reference to the context class, a process event operation for each event in the SM and other operations as the double-dispatch (DD) approach in [34]. A state class *s* also has an attribute referring to the state class associated with the composite state containing *s*. A composite state class has an attribute pointing to a state instance indicating the active sub-state of the composite state and a *dispatchEvent* operation (see 3.3.5) dispatching incoming events to the appropriate active state. An example of this transformation is shown in Fig. 2. The *ParentState* and *SubState* are ver-

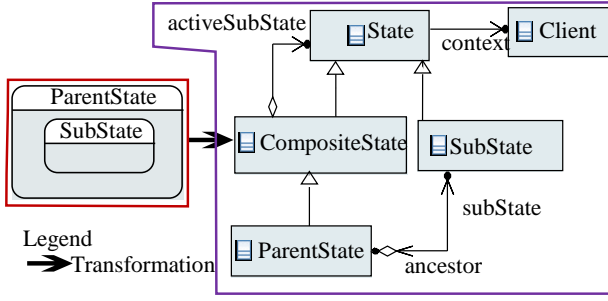


Figure 2: Transformation from hierarchical state to class diagram

texes of the SM describing the *Client* component, for instance. The *State* and *CompositeState* classes are library classes. The *ParentState* inherits from the *CompositeState* class since it is a hierarchical state.

3.3.2 Transformation of events

DD has no means to convey data of events in the SM and considers every event as the same. In our approach, each event is transformed into a UML class that can contain data. Three different event class types corresponding to the UML event types *CallEvent*, *SignalEvent* and *TimeEvent* are differentiated. An event class associated with a *CallEvent* inherits from a base event class and contains the parameters in form of attributes typed by the same types as those of the operation associated with the *CallEvent*. The operation must be a member of the provided interface of a port of the context class (a component as described above). For example, a call event *CallEventSend* associated with an operation named *Send*, which has two input parameters typed by *Integer*, is transformed into a class *CallEventSend* having two attributes typed by *Integer*. When a component receives an event, the event object is stored in an event queue.

A signal event enters the component through a port typed by the signal. From the implementation view, this signal is transferred to the component by an operation provided by the component at the associated port. Therefore, the transfer of a signal event becomes similar to that of *CallEvent*. For example, a signal event containing a data *SignalData* arrives at a port *p* of a component *C*. The transformation derives an interface *SignalDataInterface* existing as the provided interface of *p*. *SignalDataInterface* has only one operation *pushSignalData* whose body will be generated to push the event to the event queue of the component. Therefore, the processing of a *SignalEvent* is the same as that of a *CallEvent*. In the following sections, the paper only considers *CallEvent* and *TimeEvent*.

A *TimeEvent* is considered as an internal event. The source state class of a transition triggered by a *TimeEvent* executes a thread to check the expiration of the event duration as in [25] and puts the time event in the event queue of the component.

3.3.3 Transformation of transitions and actions

In this paper, actions, transition guards and effects in the SM are considered as an operation associated with a block of code describing the actions behavior. Each action is transformed into an operation in the transformed context

class. *Entry/Exit/doActivity* actions have no parameters while transition actions and guards accepting the triggering event object have access to the event data. *doActivity* is implicitly called in the *State* class and executed in a thread. A transition is transformed into an operation taking as input the source state object and the event object similarly to DD. Transitions transformed from triggerless transition which has no triggering events accept only the source state object as a parameter.

Four ways of entering a composite state are differentiated. Three of these including a transition ending (1) on the border, (2) on a sub-state or (3) on a history state of a composite state are detailed in [34]. In the last one, a transition t_{ex} ends (4) on an entry point of a composite state. Semantically, (4) is similar to (2) since both have the same sequential operations: executing the entry action of the composite state, execute the effect of the outgoing transition of the entry point t_{in} in (4) or the transition $t_{default}$ from an initial pseudo state to the sub-state in (2). The transition t_{in} is not allowed to have a guard or a trigger event similarly to the semantics of $t_{default}$.

Exiting a composite state is executed through exit points inversely to entry points. In our implementation presented in Section 4 entry points and exit points are supported in both directions of the RTTRIP.

3.3.4 Storage of tracing information

The tracing information generated in the transformation is contained in a table. Mappings from UML SM concepts to UML classes are mainly one-to-one except for attributes referring composite state or sub-state. The table therefore only keeps identifiers as qualified names and types of elements in the SM model and the associated elements in the UML class model. A part of tracing table for the SM example in Fig. 3 is shown in Table 1.

In Fig. 3, the SM is contained in, for instance, a *Client* component, *Root* is the name of the source model. States of the SM are contained in a region *TopRegion*. In the intermediate model, a package named *PerClass_Client* is created to contain all of transformed classes including ones associated with events and states. This package eases the maintenance of source code as well as the backward transformation of the RTRIP. The transition *fromStoppedToOperating* is transformed into an operation transition inside the SM class which contains *Stopped*. *Initialize*, *Enable*, *Prepare*, and *Disable* are transformed into operations in the context class *Client*. It is worth noting that there can be several transitions outgoing from a state. Therefore, more than one transition in SM can be mapped to the same qualified name in the tracing table. In order to differentiate different transitions in the intermediate model, the qualified name of a transition operation in the intermediate model is combined with the source state and the triggering event. From this tracing table, it is easy to look back the original SM elements from the elements in the intermediate model in the backward direction. This transformation can be implemented as an in-place transformation but it would surprise users. Furthermore, the intermediate model should be used only as a bridge to the code and hidden to users.

3.3.5 Code generation

The intermediate model is then used as input of a template-based object-oriented code generator. The use of an inter-

Table 1: Tracing table of state machine and class intermediate model

UML state machine concepts	UML class concepts
Root::Client::StateMachine::TopRegion::Stopped (State)	Root::Client (Class)
Root::Client::StateMachine (StateMachine)	Root::Client::PerClass_Client::StateMachine (Class)
Root::Client::StateMachine::TopRegion::Stopped (State)	Root::Client::PerClass_Client::Stopped (Class)
Root::Client::StateMachine::TopRegion::Operating (State)	Root::Client::PerClass_Client::Operating (Class)
Root::Client::StateMachine::TopRegion::On (CallEvent)	Root::Client::PerClass_Client::On (Class)
Root::Client::StateMachine::TopRegion::Initialize (OpaqueBehavior)	Root::Client::PerClass_Client::Initialize (Operation)
Root::Client::StateMachine::TopRegion::Enable (OpaqueBehavior)	Root::Client::PerClass_Client::Enable (Operation)

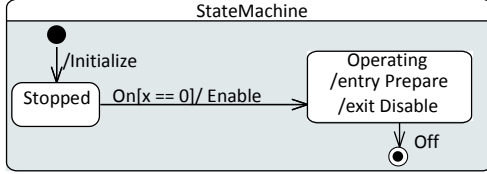


Figure 3: An example of SM for tracing table

mediate model facilitates the transformation from the SM to code and vice versa. Furthermore, the code generation process can use existing generators. Mappings from UML classes to object-oriented are trivial one-1-one. Listing 1 shows a code segment generated from the SM in Fig. 3. The *dispatchEvent* method implemented in the base composite state delegates an incoming event processing to its active sub-state. If the event is not accepted by the active sub-state, the composite state processes it. *OnEntryAction* and *OnExitAction* overwrite abstract methods which are defined in the base state class and called by the entry and exit methods, respectively. *Stopped* accepts an *On* event by implementing a corresponding *processEvent* method. The transition method from the *Stopped* to the *Operating* state checks the guard condition by calling an associated method in the context class, then executes the transition action, changes the active state and finally enters the target state by calling entry. The machine enters the final state by setting the active state to null meaning that the behavior of the region containing the final state has completed. The generated code statements are intuitively similar to the UML SM semantics and it is easy to modify the behavior of the SM by code. For example, if we would like to change the default state, we only need to modify the *setIniDefaultState* method by assigning the attribute *activeSubState* to the attribute *operating* that represent an instance of the state *Operating*.

```

class CompositeState: public State {
protected:
    State* activeSubState;
public:
    bool dispatchEvent(Event* event) {
        bool ret = false;
        if (activeSubState != NULL) {
            ret = activeSubState->dispatchEvent(event);
        }
        return ret || event->processFrom(this);
    }
}
StateMachine::StateMachine(Client* ctx){
    this->context = ctx;
    stopped = new Stopped(this, ctx);
    operating = new Operating(this, ctx);
    this->setIniDefaultState();
    this->activeSubState->entry();
}

```

```

void StateMachine::setIniDefaultState() {
    this->context->Initialize();
    this->activeSubState = stopped;
}
bool StateMachine::transition(
    Stopped* state, On* event) {
    if (this->context->guard(event)) {
        this->activeSubState->exit();
        this->context->Enable(event);
        this->activeSubState = this->operating;
        this->activeSubState->entry();
        return true;
    }
    return false;
}
bool StateMachine::transition(
    Operating* state, Off* event) {
    this->activeSubState->exit();
    //no action defined
    this->activeSubState = NULL;
    return true;
}
class Stopped: public State {
private:
    StateMachine* ancestor;
public:
    virtual bool processEvent(On* event) {
        return ancestor->transition(this, event);
    }
}

```

Listing 1: A segment of C++ generated code

3.4 Merging from modified code to UML SM

This section describes the backward process.

3.4.1 Method Overall

The generated code can be modified by adding/removing/changing states, transitions, actions. The modified generated code then needs to be reversed back to the SM to make the artifacts consistent. The overall method for backward transformation is shown in Fig. 4. The modified code is first verified by partly inspecting the code syntax and semantics to guarantee that it is reversible. There are cases in which not all code modifications can be reversed back to the SM. The verification also produces an output (*output2*) whose format is described later. If the intermediate model or the original SM is absent (the lower part of Fig. 4), a new intermediate model and a new SM are created. In the contrary, the previous code taken, for instance, from control versioning systems is also verified to have its output (*output1*) (the upper part of Fig. 4). *Output1* and *Output2* are then compared with each other to detect actual semantic changes which are about to be propagated to the original model.

3.4.2 Semantic Verification

The output of the semantic verification contains a list of event names, a list of state names, a list of transitions in

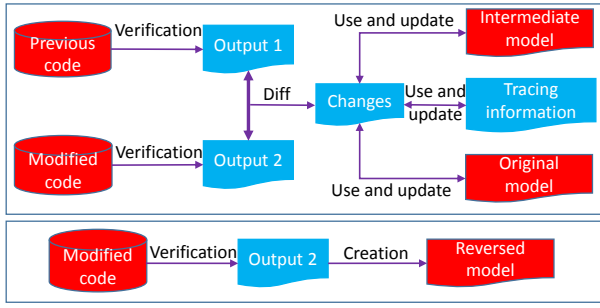


Figure 4: Overall method for reversing code to state machine

which each has a source state, a target state, a guard function, an action function and an event represented in so called abstract syntax tree (AST) transition [15]. For example, Fig. 6 presents the EMF [17] representation of transitions in a C++ AST in which *IStructure* and *IFunctionDeclaration* represent a structure and a function in C++, respectively. Each state name is also associated with an ancestor state, an entry action, an exit action, a default sub-state and a final state. The output is taken by analyzing the AST. The verification process consists of recognizing different patterns. The pattern list is as followings:

State: A state class inherits from the base state class or the composite base state class. For each state class, there must exist exactly one attribute typed by the state class inside another state class. The latter is the ancestor of the state class.

Composite state: A composite state class (CSC) inherits from the base composite state. For each sub-state the CSC has an attribute typed by the associated sub-state class. The CSC also implements a method named *setInitDefaultState* to set its default state. The CSC has a constructor is used for initializing all of its sub-state attributes at initializing time.

Entry action: If a state has an entry action, its associated state class implements *onEntryAction* that calls the corresponding action method implemented in the context class.

Activity/Exit action: Similar to the entry action pattern but implements *onExitAction/onActivity*, respectively. *onActivity* is implicitly called by the base state class and executed in a thread. According the UML state machine semantics, if the state has outgoing *triggerless* transitions, *onEntryAction* appeals the *triggerless* transition method of the ancestor state class following the *onActivity* call.

Event processing: If a state has an outgoing transition triggered by an event, the state class associated with the state implements the *processEvent* method having only one parameter typed by the event class transformed from the event. The body calls the corresponding transition method of the ancestor class.

CallEvent class: A call event class inherits from the base event class. The call event class contains attributes typed by the parameter types of the operation associated with the call event. This pattern is detected if the types of attributes of the event class match with the types of parameters of one of the methods in the context class. There is therefore an ambiguity for an event class to choose an associated opera-

tion if more than one operation detected matches the event class. Hence, this pattern poses a restriction that operations associated with events must either have different parameter types or different number of parameters. To overcome this issues, a naming convention used for *CallEvent* classes is used. The event class name is prefixed with the associated operation name. If the event class name does not follow the naming convention, the reverse is refused. Another possible solution targeting this ambiguity is to have a user interaction in case of having more than one operation matching with the event. Having an interaction allows the pattern detection get rid of ambiguity and therefore provides appropriate SM models. A signal event is treated as a *CallEvent* as previously described.

Time event: A transition is triggered by a *TimeEvent* if the state class associated with its source state implements the timed interface. The duration of the time event is detected in the transition method whose name is formulated as *"transition" + duration*.

Transition: Transition methods are implemented in the ancestor class of the source state class. Two types of transition methods correspond to trigger and *triggerless* transitions. Both *parameterize* its source state class. The trigger transition method has an additional parameter typed by the event class associated with the event triggering the transition. The body of external and internal transition methods contains ordered statements including exiting the source state, executing transition action (effect), changing the active state to the target or null if the target is the final state, and entering the changed active state by calling entry. The body can have an if statement to check the guard of the transition. The transition action and the guard are optional. Several if/else statements can appear in a *triggerless* transition method body. The body of local transition methods only checks its guard and executes the transition effect.

Transition action/guard: Transition actions and guards are implemented in the context class.

Algorithm 1 Semantic verification

Input: AST of code and a list of state classes stateList

Output: Output of semantic verification

```

1: for s in stateList do
2:   for a in attribute list of s do
3:     if a and s match child parent pattern then
4:       put a and s into a state-to-ancestor map;
5:     end if
6:   end for
7:   for o in method list of s do
8:     if o is onEntryAction || o is onExitAction then
9:       verifyEntryExit(o);
10:    else if o is processEvent then
11:      verifyProcessEvent(o);
12:    else if o is setInitDefaultState & s is composite
13:      then
14:        verifyInitDefaultState(s);
15:      else if o is timeout & s is a timedstate then
16:        verifyTimeoutMethod(o);
17:        verifyProcessEvent(s, o);
18:      end if
19:    end for
20:  end for

```

Algorithm 1 shows the algorithm used for verifying code

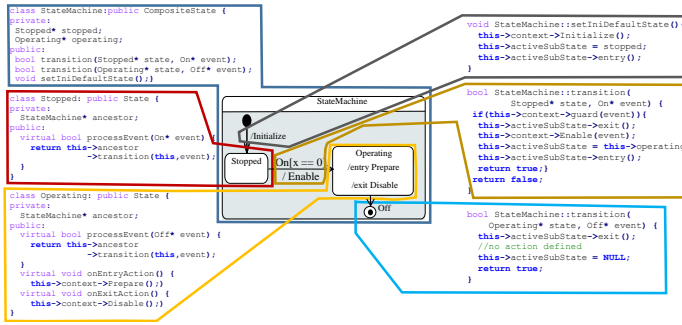


Figure 5: SM element-code segment mapping partition

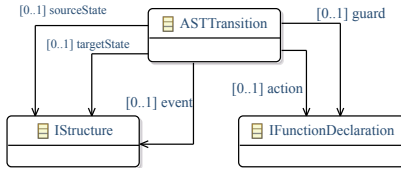


Figure 6: Transitions output from the verification

semantics. Because of limited space, *verifyEntryExit*, *verifyProcessEvent*, *verifyInitDefaultState*, *verifyTimeoutMethod* and *verifyProcessEvent* are not presented but they basically follow the pattern description as above. In the first step of the verification process, for each state class, it looks for an attribute typed by the state class, the class containing the attribute then becomes the ancestor class of the state class. The third steps checks whether the state class has an entry or exit action by looking for the implementation of the *onEntryAction* or *onExitAction*, respectively, in the state class to recognize the *Entry/Exit* action pattern. Consequently, event processing, initial default state of composite state and time event patterns are detected following the description as above. Fig. 5 shows the partitioning used for matching code segments to SM elements. Each partition consists of a code segment and the corresponding model element which are mapped in the backward direction. For instance, the *Stopped* class in code is detected as a representation since it inherits from the base class *State*.

3.4.3 Construction of SM from verification output

If an intermediate model is not present, a new intermediate model and a new SM are created by a reverse engineering and transformation from the output of the verification process. The construction is straightforward. At first, states are created. Secondly, UML transitions are built from the AST transition list. Action/guard/triggering event of a UML transition is created if the associated AST transition has these.

3.4.4 Updating the original SM from modified code

In contrast to the previous sub-section, if an intermediate model is existing, lists of states and transitions are retrieved from the intermediate model. The output of the verification process and that of the intermediate model transformation are compared to each other to detect semantics changes of the modified code. The algorithm for detecting state and transition changes is shown in Algorithm 1.

The algorithm takes as input lists of state names, transitions, ancestor maps extracted from the intermediate model and the modified code, respectively. The algorithm results in lists of state names, transitions to be added/deleted/updated/moved. It first examines the list of state names extracted from the modified code *Lc* to find which state to be added. A state is considered as a to-be-updated state if its name exists in both of the two lists of state names and its ancestor names in both of the ancestor maps are identical. If the latter condition is not satisfied, the state is considered as being moved to another composite state. Remaining states (not added/updated/moved) in *Li* are added to the to-be-deleted state list. The transition and event change detection is similar to that of states but, due to the space limitation, it is not detailed. For transition change detection, instead of checking by name, the source and target state names of transitions and the associated event name in *Tc* and *Ti* are used.

The changes detected by the algorithm are then used in a change propagation step which updates the original SM. Events, states and transitions are sequentially processed in order. The processing of elements to be deleted results in deleting corresponding elements in the SM. As previously described, the mapping information for elements in code and the intermediate model is also stored in a table. Each to-be-deleted element in code is associated with an element in the intermediate model. Therefore, it is trivial to retrieve the model element in the intermediate model associated with the to-be-deleted element.

The found model element in turn helps identify the associated element in the SM by using the mapping table between the SM and the class model. For each deleted event in code, the associated event class in the class model and the event in the SM are deleted. Deleted states and transitions are similarly propagated. A deletion of a transition includes deleting its guard, triggers and transaction action.

Algorithm 2 Change detection

Input: *Li*, *Lc*, *Ti*, *Tc*, *mapI*, *mapC* are lists of state names, transitions, ancestor map extracted from intermediate model and modified code, respectively

Output: *adS*, *delS*, *uptS*, *movS* are lists of added, deleted, updated and moved states respectively. *adT*, *delT*, *uptT* are lists of added, deleted and updated transitions

```

1: for c in Lc do
2:   if !Li.contains(c) then
3:     adS.put(c);
4:   else
5:     if mapC.get(c) = mapI.get(c) then
6:       uptS.put(c);
7:     else
8:       movS.put(c, mapI.get(c), mapC.get(c))
9:     end if
10:    Li.remove(c);
11:  end if
12: end for
13: for i in Li do
14:   delS.put(i)
15: end for
16: detecttransitionchanges;
```

For each added event in code, an event is added to the

class model and in turn to the SM. For each added state, its ancestor state is retrieved through the mapping tables, a new state is then created and attached to the ancestor. Entry and exit actions are also added to the new state afterward. A moved state is handled by looking for the associated state, the old and new ancestor state in the SM, and moving the associated state to the new ancestor. Each added transition is propagated by creating a new transition in the SM and retrieving source and target states from the mapping tables. An update is executed by looking in the mapping tables for elements in the SM associated with elements updated in code. It is worth noting that this algorithm detects a renaming of an event or state as a deletion followed by an addition.

For example, assuming that we need to adjust the SM example shown in Fig 3 by adding a guard to the transition from *Operating* to the final state. The adjustment can be ordered by either modifying the SM model or the generated code. In case of code, the associated transition function in Listing 1 is edited by inserting an if statement which calls the guard method implemented in the context class. The transition function becomes as in *Li*. The algorithm in Algorithm 2 adds the transition function into the updated list since it finds that the source state, the target state and the event name of the transition is not changed. By using mapping information in the mapping table, the original transition in the SM is retrieved. The guard of the original transition is eventually created.

4. IMPLEMENTATION

The proposed approach is implemented in a prototype existing as an extension of the Papyrus modeler [9]. Each SM is created by using a SM diagram and contained in a component. Low-level SM actions are directly embedded in form of a block of code written in specific programming languages such as **C++/JAVA** into the SM. **C++** code is generated by the prototype but other object-oriented languages can be easily generated since the approach relies on existing code generators from the intermediate model. The code generation consists of transforming the SM to UML classes and eventually to code by a code generator following the proposed approach. The code generator can generate code for hierarchical and concurrent SMs. In the reverse direction, code pattern detection is implemented as described in the previous section to verify SM semantics in code. If the generated code is modified, two options are supported by the prototype interface to make the SM and code consistent again. One is to create a new model containing the SM from the modified code in the same Eclipse project and the other one is to update the original SM by providing as input the intermediate model and the original model. At the writing moment, the prototype does not support the reverse of *doActivity*, concurrent SMs and pseudo states which are *history*, *join*, *fork*, *choice*, and *junction*.

5. EXPERIMENTS

In order to evaluate the proposed approach and the prototype, we answer two questions related to two laws of RTRIP [14].

RQ1: A state machine *sm* is used for generating code. The generated code is reversed by the backward transformation to produce another state machine *sm'*. Are *sm* and *sm'*

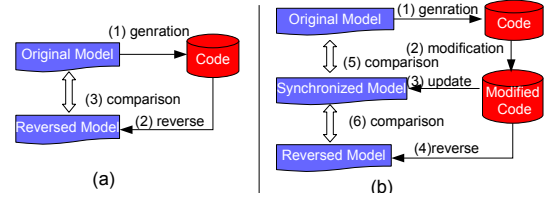


Figure 7: Evaluation methodology to answer RQ1

identical? In other words: whether the code generated from UML state machines model can be used for reconstructing the original model.

RQ2: A state machine *sm* is used for generating code. The generated code is modified by adding/deleting/modifying elements such as states, transitions, or events. The modified code is then reversed by merging changes to *sm*. Are modifications in the modified code propagated to *sm*?

This section reports our experiments targeting to the two questions. Two types of experiments are conducted. For each type, the number of elements in models are taken into account by a JAVA program. Fig. 7 (a) and (b) show the evaluation methodologies to answer **RQ1** and **RQ1**, respectively. Additionally, the time complexity and performance analysis of our approach is presented. Results of a lightweight experiment on the semantic conformance of run-time execution of the generated code are also shown afterward.

Furthermore, in software development projects, some traditional programmers might want to practice with code in a traditional way and some MDE developers may prefer working with models. Therefore, it is necessary to compare the development/maintenance cost between the two practices by comparing the number of steps needed to do the same action.

5.1 Reversing generated code

This experiment is targeting **RQ1**. 300 hierarchical state machines are randomly automatically generated. Each of these has 80 states including atomic and composite states, and more than 234 transitions. The number of elements is unrealistically big but it is artificially used to show the scalability of the approach. The number of lines of generated code for each machine is around 13500. Names of the generated states are different. An initial pseudo state and a final state are generated for each composite state and containing state machine. Other elements such as call events, time events, transition/entry/exit actions and guards are associated with an appearance probability sensing that if a random number is less than the probability, the element associated with the probability is generated. For each generated call event, an operation is generated in the context class which is also generated. The duration is also generated for each time event.

The set up information for the SM generation is shown in Table 2. Code is generated from each state machine. The generated code is reversed to a state machine. The latter is then compared to the original one by using information of SM such as the number of states, transitions.

Table 3 shows some of the generated models which have the same information as the models created by doing the backward process of the generated codes. The results of this

Table 2: Set-up information for model generation

Description	Value
Number of generated states	80
Number of generated transitions	>234
Probability of having an event for transition	0.8
Probability of having CallEvent for transition	0.7
Probability of having an entry/exit action for state	0.7
Probability of having a transition action and guard	0.7

Table 3: Three of model results of generation and reverse: Abbreviations are atomic states (AS), composite states (CS), transitions (T)

Test ID	AS	CS	T	Is reverse correct?
1	47	33	234	Yes
2	42	38	239	Yes
..	Yes
300	41	39	240	Yes

experiment show that the proposed approach and the implementation can successfully do code generation from state machines and reverse.

5.2 Change propagation

We manually created two state machines (model level): one describing Java Thread life-cycle [2] and the other one representing a telephone presented in [33]. For each SM code is generated. Code is then manually modified. Each modification test consists of one or several actions described in Table 5. The original SM is updated by doing a backward process from the modified generated code with the presence of the intermediate and original model. The updated SM ($sm_{updated}$) is in turn compared with the SM created ($sm_{reversed}$) by the reverse engineering (see Fig. 7). A modification test is passed if the corresponding models $sm_{updated}$ and $sm_{reversed}$ are the same. The Table 4 shows the number of test cases (Tests) applied to each model, of passed test cases (Passed tests) and the result of change propagation experiment. The table shows that our approach is able to update the original state machine following code-side modifications.

5.3 Time complexity and performance

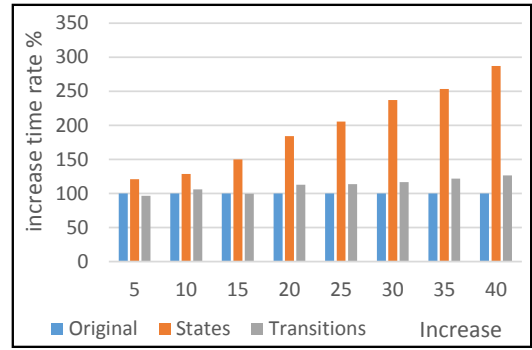
To analyze the time complexity, we consider two tasks: semantic verification and SM construction from the verification output.

For each state, the semantic verification consists of the following phases: (1) detecting composite/sub-state pattern, (2) loop over all methods of a state class, (3) detecting entry action pattern, (4) detecting exit action pattern, (5) detecting processing *CallEvent*, (6) detecting processing *TimeEvent*, and (7) detecting default state pattern.

Due to the space limitation, we cannot present the detail

Table 4: Change propagation experimental results

State machine	Tests	Passed tests	Is change propagation passed?
Java Thread	20	20	Yes
Telephone	30	30	Yes

**Figure 8: Performance impact comparison between states and transitions**

of the complexity of each phase. To sum up, the semantic verification has a worst-case complexity (abbreviations n_s , n_t , n_{ce} , n_a are the number of states, transitions, call events and actions, respectively) $C_1 = n_s(n_s^2 + 9n_t^2 + 6n_t n_s + 2n_a n_{ce}) = O(n_s^3) + O(n_t n_s^2) + O(n_s n_t^2) = O(n^3)$ with $n = \max(n_t, n_s)$. The worst-case occurs if a state can accept all incoming events, all transitions have the same source state and all states contain each other. This case is unrealistic.

The SM construction from the verification output has a worst-case time complexity $C_2 = O(n_s^2) + O(n_s n_t) = O(n^2)$. Therefore, the reverse engineering has a worst-case complexity of $O(n^3)$ with $n = \max(n_s, n_t)$.

To analyze the performance of reverse engineering, we randomly generate 5 models with base set up information in which the numbers of states and transitions are 20 and 50, respectively. We use a Dell Latitude E554 laptop with a 2.1GHz Intel Core i7 with 16 Gb of RAM. The running time of the reverse for the generated code associated with these models is measured. To analyze the impact of state and transition to the reverse performance, we increase the number of states and transitions by five, alternatively. The models resulting from the increase are used for generating code. The running time of reverse engineering the new generated code is measured. For each measurement, three times are computed, the median of these measured values are retained.

Fig. 8 shows the increase of the number of instances for states and transitions, and the increase time rate, which is the execution time for reversing modified models divided by the execution time for reversing the original model. The median execution time for reversing the original model is 64557 ms. The results show that the number of states has a higher performance impact than the number of transitions. When the number of added states grows, the running time for reverse also grows quickly. Whereas, in case of transitions, the difference is small and not clear as we analyze that the worst-case complexity never occurs.

5.4 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 [3]. Moka is a model execution engine offering Precise Semantics of UML Composite Structures [27]. Fig. 9 shows our method. We first use our code generator to generate code (Step (1)) from Moka examples. Step (2) simulates

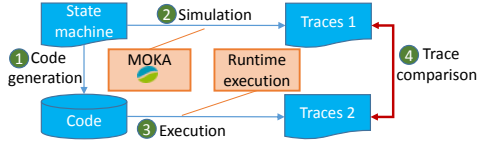


Figure 9: Semantic conformance evaluation methodology

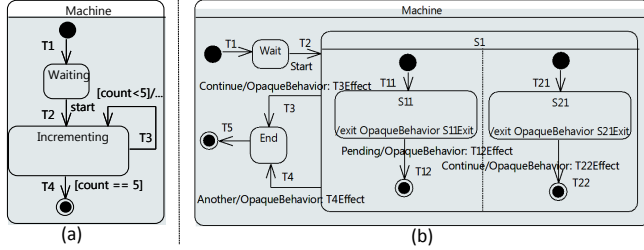


Figure 10: Semantic conformance evaluation methodology

the examples by using the Moka model execution engine to extract the sequence of observed traces including bypassed states and executed actions. The sequence 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 [7]. [To be continued]

5.5 Development/maintenance cost

To compare the development/maintenance cost, we investigate steps needed in generated code and models having the equivalent semantics. For example, to add a state, on one hand, two steps are needed in diagrams including (1) specifying the parent state and (2) dragging & dropping the state notation to that parent. On the other hand, three code modifications are (1) create a state class inheriting from the base state and its constructors, (2) add to the parent state class an attribute, and (3) add a line of code to initialize the state attribute in the parent state constructor. Table 5 shows the number of steps needed for each operation. In this table, model manipulations are the winner in most of cases because of graphical representation advantages but code manipulations are still useful and comparable.

In software development, programmers might modify the generated code, the modifications might violate structures of code or SM semantics. To resolve this issue, as previously described, we provide a semantic verification that partly and loosely inspects the AST of generated code. This inspection approach always reverses the code to the SM as well as the code is state machine-compliant even though the code is not compiled. This approach is very useful in practice in which programmers might partly modify code, automatically update the original SM by our RTRIP, and automatically regenerate state machine-compliant code into the remaining application code. This re-generation does no more than completing missing elements in code meaning that all previous changes are preserved. This practice is also limitedly supported by Fujaba [23] in which activity and collaboration

Table 5: Cost comparison

Description	Model	Code
Add a state	2	3
Add a transition	3	3
Add entry/exit action	2	2
Add transition action	2	2
Update action	1	1
Redirect target state of a transition	1	1
Create a call event to a transition	3	6
Create a time event to a transition	3	5
Delete a state	2	2
Delete a transition	1	3
Delete entry/exit action	1	2
Delete transition action	1	2
Delete a call event	2	many
Delete a time event	2	many

diagrams are partly synchronized with JAVA.

6. CONCLUSION

This paper presented a novel approach to round-trip engineering from UML state machines to code and back. The forward process of the approach is based on different patterns transforming UML state machine concepts such as states, transitions and events into an intermediate model containing UML classes. Object-oriented code is then generated from the intermediate model by existing code generators for programming languages such as C++ and JAVA. In the backward direction, code is analyzed and transformed into an intermediate whose format is close to the semantics of UML state machines. UML state machines are then straightforwardly constructed or updated from the intermediate format.

The paper also showed the results of several experiments on different aspects of the proposed approach with the tooling prototype. Specifically, the experiments on the correctness, performance, semantic conformance of code, and the cost of system development/maintenance using the proposed round-trip engineering are conducted. Although, the reverse direction only works if manual code is written following predefined patterns, the semantics of state machines is explicitly and intuitively present and easily to follow.

While the semantic conformance of code generated is critical, the paper only showed a lightweight experiment on this aspect. The reason is that the implementation of the prototype takes a lot of time. A systematic evaluation is therefore in future work. Furthermore, as evaluated in [7], the approach inheriting from the double-dispatch trades a reversible mapping for a slightly larger head. The reverse does not work concurrent state machine and several pseudo-states. Hence, future work should resolve these issues.

References

- [1] AndroMDA Model Driven Architecture Framework - Homepage. <http://andromda.sourceforge.net/>.
- [2] Java 6 thread states and life cycle UML protocol state machine diagram example. <http://www.uml-diagrams.org/examples/>

java-6-thread-state-machine-diagram-example.html?context=stm-examples.

- [3] Moka Model Execution.
- [4] T. Allegrini. Code Generation Starting From Statecharts Specified in UML. 9, 2002.
- [5] L. Angyal, L. Lengyel, and H. Charaf. A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 463–472, Mar. 2008.
- [6] U. Aßmann. Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science*, 82(5):33 – 41, 2003. {SC} 2003, Workshop on Software Composition (Satellite Event for {ETAPS} 2003).
- [7] J. O. Blech and S. Glesner. Formal verification of java code generation from uml models. In *... of the 3rd International Fujaba Days*, pages 49–56, 2005.
- [8] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*, volume 3. 1998.
- [9] CEA-List. Papyrus Homepage Website. <https://eclipse.org/papyrus/>.
- [10] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, June 1996.
- [11] E. Domínguez, B. Pérez, A. L. Rubio, and M. A. Zapata. A systematic review of code generation proposals from state machine specifications, 2012.
- [12] B. P. Douglass. *Real-time UML : developing efficient objects for embedded systems*. 1999.
- [13] C. Duby. Class 265 : Implementing UML Statechart Diagrams. *Proceedings of Embedded Systems Conference Fall*, (April), 2001.
- [14] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.
- [15] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
- [16] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In *Model Driven Engineering Languages and Systems*, pages 543–557. Springer, 2006.
- [17] R. Gronback. Eclipse Modeling Project.
- [18] 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)*, volume 5063 LNCS, pages 31–45, 2008.
- [19] S. Jarzabek and H. D. Trung. Flexible generators for software reuse and evolution: Nier track. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 920–923, May 2011.
- [20] S. Jörges. Construction and evolution of code generators: A model-driven and service-oriented approach. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7747:1–265, 2013.
- [21] S. Kelly and J. P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. 2007.
- [22] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. *University of Paderborn*, 2007.
- [23] T. Kleín, U. A. Nickel, J. Niere, and A. Zündorf. From uml to java and back again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.
- [24] 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. Kienzie, 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)*, pages 183–200, 2014.
- [25] I. Niaz and J. Tanaka. Mapping UML statecharts to java code. *IASTED Conf. on Software Engineering*, pages 111–116, 2004.
- [26] I. A. Niaz, J. Tanaka, and others. Mapping UML statecharts to java code. In *IASTED Conf. on Software Engineering*, pages 111–116, 2004.
- [27] OMG. Precise Semantics Of UML Composite Structures. (October), 2015.
- [28] E. Paesschen, W. Meuter, and M. D’Hondt. *Model Driven Engineering Languages and Systems: 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005. Proceedings*, chapter Self-Sync: A Dynamic Round-Trip Engineering Environment, pages 633–647. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
- [29] V. Paradigm. Visual Paradigm Homepage Website. <http://www.visual-paradigm.com/>, 2015. [Online; accessed 01-Sept-2015].
- [30] S. Sendall and J. Küster. Taming Model Round-Trip Engineering.
- [31] A. Shalyto and N. Shamgunov. State machine design pattern. *Proc. of the 4th International*, 2006.
- [32] SparxSystems. Enterprise Architect, Sept. 2014.
- [33] O. M. G. A. Specification and C. Bars. OMG Unified Modeling Language (OMG UML). *Language*, (November):1 – 212, 2007.
- [34] V. Spinke. An object-oriented implementation of concurrent and hierarchical state machines. *Information and Software Technology*, 55(10):1726–1740, Oct. 2013.

- [35] B. Steffen, T. Margaria, and C. Wagner. *Round-Trip Engineering*, chapter 96, pages 1044–1055.