XSeparation: A Code Generation Approach for Round-trip Engineering of Model and Code

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*—UML State Machines and composite structure are prove to efficiently capture and simplify the complexity in designing the behavior and structure, respectively, of event-driven and cyber-physical system architecture. Model Driven Engineering (MDE) tries to generate fully executable code from architecture models. Nevertheless, because of abstraction gap between architecture and implementation, and lack of means to describe fine-grained behaviors, e.g. UML State Machine and Sequence diagrams are not sufficient, current MDE tools and approaches put code directly in the model in order to achieve full code generation. It is, however, not favorable because the architecture should only hold design decisions. Hence, code generated from the tools must be tailored by programmers for fine-grained code. In this case, the architecture correctness might be violated, which raises the problem of consistency between architecture and implementation code. This paper tackles the problem of synchronization between object-oriented code and architecture model for the co-evolution of these artifacts. We propose XSeparation - an extreme separation code generation and synchronization for enabling the bidirectional traceability between architecture model and code. We implemented XSeparation a prototype based on the Papyrus modeling tool and evaluated it by developing a software application for LEGO.**

I. Introduction

UML State Machines and composite structure are prove to efficiently capture and simplify the complexity in designing the behavior and structure, respectively, of event-driven and cyber-physical system architecture [1], [2]. A number of code generation approaches have been proposed in the context of Model-Driven Engineering (MDE) [3] to automate the process of translating the software architecture represented by these models into implementation [1], [4], [5].

Ideally, a full model-centric approach is preferred by MDE community due to its advantages [6] such as complex project management, model-based system analysis, abstraction, and automation. However, in industrial practice, there is significant reticence [7] to adopt it.

On one hand, it is frequent that programmers modify architecture information during implementation because of abstraction gap between architecture and implementation. Current industrial MDE tools such as IBM Rhapsody put fine-grained behaviors/computational algorithms directly in the architecture model as blocks of text to generate full code. However, it is not favorable because the architecture should only hold design decisions. Hence, code generated from the current MDE tools must be tailored by programmers for fine-grained code. In this case, the architecture correctness might be violated, which is not easy to detect due to the lack of a bidirectional traceability between the architecture and code [8]. Some approaches [9] prevent modifications in code through complete code generation. However, the latter only works for very highly specialized domains and specific architecture styles [10].

On the other hand, in software evolution, continuous development and maintenance, the architects might change the architecture for new functionalities or requirements while the programmers might still implement the current architecture or modify the code for various reasons such as code level optimization, bug fixing, refactoring. In MDE tools, the regeneration of code from the changed architecture would overwrite the modifications made by programmers in code. Some tools such as Eclipse Modeling Framework (EMF) [11] separate code areas, which could be modified by the programmers, to preserve the code changes by using some specialized comments such as *@generated NOT*. However, current separation mechanisms require the programmers to be very highly discipline. Furthermore, even so, accidental changes are still possible [10]. The *1.x-way architecture mapping deep separation* approach [10] overcome the limitations of these separation mechanisms by generating *architecture-prescribed code* in a class separating from user-code written in an other class. However, deep separation does not allow to modify the architecture at the code level.

The modifications made in model and code raise the consistency and synchronization problem [12]. If the latter is not solved, modifications in the code are not reflected to the model. As a result, the model does not reflect the actual running system, which even worse entails that model-based activities such as architecture and behavior analysis, or testing are obsolete, hence many of the advantages of MDE would disappear. In order to solve this problem, which hinders the adoption of MDE in practice, it is necessary to have a code generation process, which establishes a way to trace code modifications back to the model.

In this paper, we propose **XSeparation** - an extreme separation code generation and synchronization approach, which enables the bidirectional traceability between architecture model and code. XSeparation lifts the *deep separation* approach to an extreme level of separation. XSeparation provides an adequate support for programmers to control both structure and behavior of a component at the code level.

Our work is motivated by both industry and research. From the latter, MDE, for flexibility, allows the architecture model and the generated code to evolve concurrently [13]. Furthermore, R. N. Taylor et al. [14] pointed out an important research direction, in which key design decisions may be made in implementation (code) and evolution of architecture must be seamlessly propagated to the code. This implies the fluid moving from architecture to code and vice versa. Additionally, synchronization of model and code is also considered as an important need by the MODELS community [15].

For industry, one of the reasons that impede the adoption of MDE is the perceived gap between diagram-based modeling languages and textual languages. On one hand, programmers prefer to use the more familiar combination of a programming language and Integrated Development Environment (IDE). On the other hand, software architects, working at higher levels of abstraction, tend to favor the use of modeling languages for describing the system architecture.

Our contribution is summarized as followings:

- XSeparation - code generator, mode-code synchronizer, and compiler for enabling a fluid moving between architecture and implementation.
- Evaluations of XSeparation based on a case study.

The remainder of this paper is organized as follows: Section II presents the overview of XSeparation code generation. Section IV and V describe how XSeparation is applied to code generation from architecture structure and behavior, respectively, for enabling bidirectional traceability between model and code. Section VI shows how to synchronize concurrent modifications architecture model and XSeparation-generated code. The development of a case study is presented in Section VIII to evaluate XSeparation. Section IX shows related work. The conclusion and future work are presented in Section X.

II. XSeparation code generation in a nutshell

This section presents overview of XSeparation. For a component whose structure is described by UML components, parts, ports, and connectors and behavior by a UML State Machine, the structure- and the behavior-prescribed code are generated within the same object-oriented class.

Fig. 1 shows the overview of the code generation in XSeparation. A component *System* is generated to code of an object-oriented class, which consists of five code parts defined as followings:

- **Component structure-prescribed code:** It is generated from the component structure, which is described by UML class and component diagram structural concepts such as *property*, *port*, *part*, and *connector*.
- **Behavior-prescribed code:** It is generated from the component behavior, which is described by UML State Machine diagrams and UML-defined events such as *CallEvent*, *TimeEvent*, *SignalEvent*, and *ChangeEvent*.
- **State machine action code:** It is used to define fine-grained action code for the state machine actions such as state entry/exit/doActivity and transition effect. This code can also be generated from the model in case of industrial tools such as IBM Rhapsody and Enterprise Architect, which allow to embed the fine-grained code within the model as blocks of text. This code, which will be invoked by the state machine behavior of the component, has an agreement (*BP-U Agreement*) with **Behavior-prescribed code**. The details of *BP-U Agreement* are presented in Section V.
- **User-filled skeleton code:** This code contains object-oriented methods generated from UML operations defined in the model for the component. Similar **State machine action code**, users can fill fine-grained code within the model for the operations. The methods can be called by **State machine action code** or **Component structure-provided implementation**.
- **Component structure-provided implementation:** A component might provide some interfaces through its ports in order for other components to interact with. This code part is for the implementation of the interfaces. If **Architecture-prescribed code** contains a port, which provides an interface and is not delegated (connected) to an other port of one of the sub-components (see Section IV for more details).

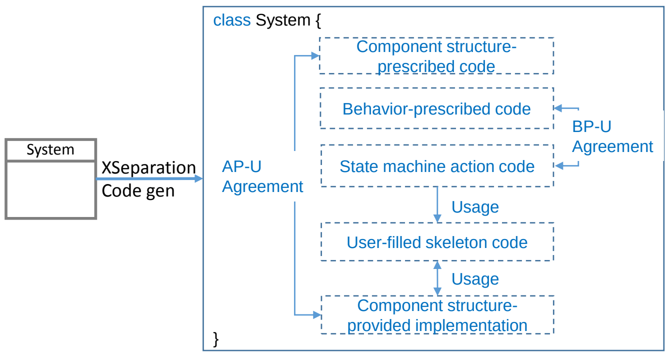
System
XSeparation Code gen
class System {
Component structure-prescribed code
Behavior-prescribed code
State machine action code
User-filled skeleton code
Component structure-provided implementation
}
AP-U Agreement
BP-U Agreement
Usage
Usage

Fig. 1. XSeparation overview

Although the five parts are ideally separate from each other as in Fig. 1, these parts, except **Behavior-prescribed code**, can be populated in an interleaved way.

In the following sections, we will present the details of these parts with an illustrative example.

III. Illustrative example

We consider a producer-consumer example developed using component-based engineering. Fig. 2 (left) shows the component-based model of the example. The *p* producer produces data items and sends to a first-in first-out passive communication channel instance *fifo*. The latter stores data in order for the consumer to pull it. The class diagram of *FIFO* is explored in Fig. 2 (right). FIFO persists data items in a sized queue attribute, namely, *queue*. The latter is associated with the number of currently stored items (*numberOfItems*), the capacity (*MAX_SIZE*), and other attributes and operations used for validating incoming data items and the status of the queue.

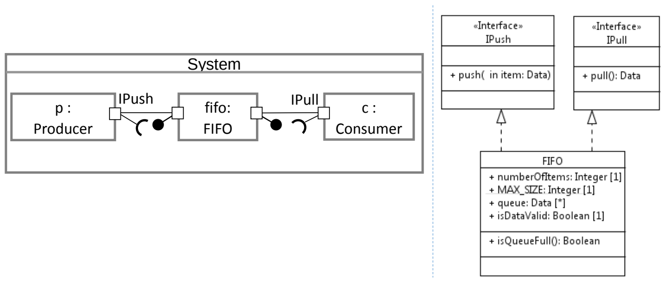

Fig. 2. Architecture of System (left) and Class diagram of FIFO (right)

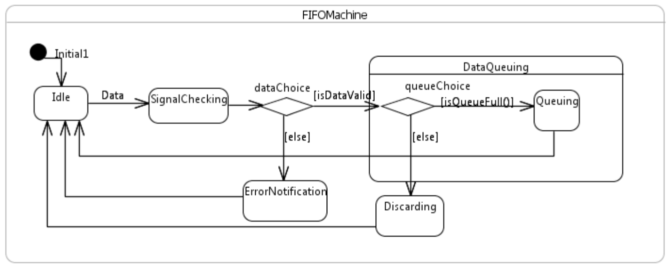

Fig. 3. FIFO's UML State Machine

The producer's port with *IPush* as its required interface is connected to the port of FIFO with *IPush* as its provided interface so that the producer and FIFO can interact with each other through their respective port. Because FIFO provides two interfaces, it implements these.

The behavior of *FIFO* is described by using a UML State Machine as in Fig. 3. The machine activates the *Idle* state as initial state. The latter waits for an item to come to the *fifo* component (through the *pPush* port, for instance). The item is then checked for its validity by using a *choice* pseudo state. The *DataQueuing* state verifies the status of the queue to decide to either add the item to the queue or discard it.

In the next sections, this example will be used for illustrating how XSeparation works.

IV. XSeparation for Architecture

This section presents how XSeparation generates **Component structure-prescribed code** and **Component structure-provided implementation**. **Component structure-prescribed code** is controlled by the UML component modeling concepts such as component, connector, part, and port. These concepts are not directly mapped to the object-oriented code. Therefore, we customize an object-oriented language by adding more specific constructs to it in order to be able to establish a bidirectional traceability between the architecture and the code.

Listing 1 shows the C++ code generated from the architecture model of *System* by using XSeparation. A UML component is mapped to a class while additional constructs: part and port, reflect UML parts and ports, respectively. A UML connector contained by a parent component (System in this case) is mapped to an invocation to the *bindPorts* method, which takes as input two declared ports of the sub-components of the parent. The **bindPorts** method must be called under the unique configuration declared within the parent component for wiring its sub-components' ports.

Listing 1. Architecture-prescribed code generated from the architecture model in Fig. 2 (left)

```
 1 class System {
   public:
 3   part Producer p;
     part Consumer c;
 5   part FIFO fifo;
     configuration {
 7     bindPorts(p.pPush, fifo.pPush);
       bindPorts(c.pPull, fifo.pPull);
 9   }
   }
11 class IPull {
     virtual Data* pull() = 0;
13 }
   class IPush {
15   virtual void push(Data& data) = 0;
   }
17 class Producer {
   public:
19   RequiredPort<IPush> pPush;
   };
21
   class Consumer {
23 public:
     RequiredPort<IPull> pPull;
25 };
   class FIFO : public IPush, IPull {
27 public:
     ProvidedPort<IPush> pPush;
29   ProvidedPort<IPull> pPull;
     Data* pull(){
31     //fine-grained code for pull
     }
33   void push(Data& data){
       //fine-grained code for push
35   }
     int numberOfItems;
37   const int MAX_SIZE = 100;
     Data queue[MAX_SIZE];
39   bool isDataValid;
     bool isFullQueue() {
41     //fine-grained code
     }
43 }
```

Each part is typed similarly to that of class attributes. Each port can provide and/or require an interface. For the UML *FIFO* component, its two ports provide two interfaces *IPush* and *IPull*, which respectively define the two UML operations *push* and *pull*. These interfaces are mapped to object-oriented interfaces, which in C++ are classed with all of its methods defined as *pure virtual*. Following the AP-U Agreement between as described previously in Section II, **Component structure-provided implementation** is generated in the FIFO class. The implementation should consist of the concrete definitions of the *push* and *pull* methods, which are originally declared in the two interfaces IPush and IPull (class in C++).

The UML FIFO component also contains some UML properties and operations. These members are mapped directly to the corresponding concepts in object-oriented code, class attributes and methods in particular. Their generated code is **User-filled skeleton code** as the lines 34-40 in Listing 1. **User-filled skeleton code** code can use or be used by **Component structure-provided implementation**.

Interaction between components through ports: Given the generated code as in Listing 1, we now show how a component, through its ports, can interact with other components. A component containing a port with a required interface

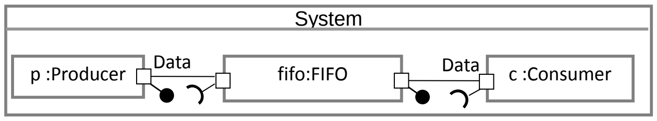

Fig. 4. Component-based architecture example with data port

can call the defined services/methods implemented within the component containing an other port with the interface as provided.

Listing 2 shows a C++ code segment of the components *Producer* and *Consumer*. The *sendDataToFifo* and *pullDataFromFifo* methods can be either written by programmers or generated from the example model. If written by the programmers, these methods will be synchronized with the model because there is a clear mapping between object-oriented methods and UML operations. The producer pushes data items to the fifo by invoking the *push* method through the *pPush* port and the consumer pulls items by invoking *pull* through *pPull*.

Listing 2. Code for interacting between components

```
class Producer {
  void sendDataToFifo(Data& item) {
    pPush->push(data);
  }
}
class Consumer {
  Data* pullDataFromFifo() {
    pPull->pull();
  }
}
```

Data flow port: A port can also provide/require a data type, a class in this case, to become a data flow port. A data flow port is useful when being used with UML State Machine signal events, which will be detailed in Section V. Data items flow from a port providing to a port requiring the items. In the examples, data ports can be used instead of ports with interfaces. Data ports are defined to support the explicit understanding of "physical" system data flow.

Fig 4 shows the component diagram of the example in using data ports. The *p* producer provides data items to *fifo* through their respective ports *pProvideData* and *pRequireData*. The code generated by XSeparation for the data ports is shown in Listing 3.

Listing 3. System using data ports

```
class Producer {
public:
  ProvidedDataPort<Data> pProvideData;
}
class FIFO : public IPush, IPull {
public:
  RequiredDataPort<Data> pRequireData;
  ProvidedDataPort<Data> pProvideConsumer;
}
class Consumer {
public:
  RequiredDataPort<Data> pRequireData;
}
```

In system implementation, the producer provides data items to the fifo via the port *pProvideData* by calling *pProvideData.sendData(item)* as an object-oriented way. In case that the behavior of the component is described by a state machine, the *sendData* method will fire a signal event in order for the state machine to handle it.

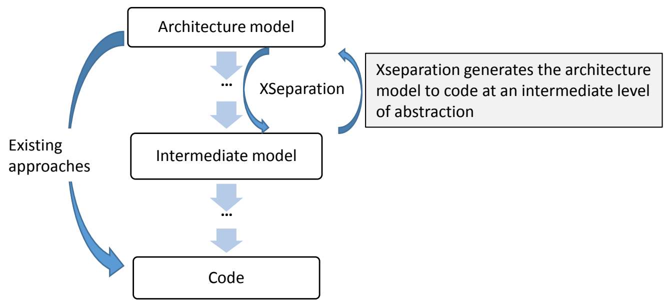
Architecture model
...
XSeparation
Xseparation generates the architecture model to code at an intermediate level of abstraction
Existing approaches
Intermediate model
...
Code

Fig. 5. Comparison of XSeparation with other code generation approaches

Discussion: Compared to other code generation approaches, the philosophy of XSeparation is different. Usually, a code generation approach refines the architecture model at high level of abstraction to the code through a chain of transformations. On one hand, the latter help to modularize the code generation process, hence make it easy to implement. On the other hand, the traceability from the code back to the model becomes very hard.

Fig. 5 shows how different XSeparation is from the other approaches. XSeparation, differently from other approaches, generates code at an intermediate abstraction level. Because component concepts such as part, port, and connectors are not directly mapped to object-oriented code, XSeparation proposes to preserve the abstraction level of these concepts in code by introducing additional programming constructs to object-oriented code as shown in the example in Listing 1. Other modeling concepts such as properties and operations are generated to code using a trivial mapping used by many industrial code generation tools such as Rhapsody.

XSeparation and the *1.x-way mapping deep separation* approach are similar in the philosophy of separating completely **Component structure-prescribed code** and **User-filled skeleton code** in different constructs, the two approaches are fundamentally not the same. *Deep separation* ignores modifications made to the architecture at the code level, hence much assumes that the programmers are not interested in modifying architecture. It, however, contradicts with the need pointed out by R. N. Taylor et al. [14], which allows to fluidly moving between architecture design and coding tasks

XSeparation, on the other hand, puts the architecture information in code at the right abstraction level with appropriate additional constructs. By this way, XSeparation, compared to *deep separation*, adds the right for programmers to have more support for not only understanding but also modifying the architecture at the code level. Because the abstraction level of component modeling concepts are preserved, XSeparation enables the reflection of modifications made in code to the model.

We have presented how XSeparation works for the architecture structure. In the next section, XSeparation for the architecture behavior described by UML State Machines will be detailed.

V. XSeparation for Behavior

This section describes the application details of XSeparation to the generation of behavior code. The behavior of a UML component is described by a UML State Machine[1].

A. Reminding of UML State Machines

A UML State Machine is efficient to describe the behavior of a component [1], [2]. It has a number of vertexes (state and pseudo state) and well-defined conditional transitions. A state is either an atomic state or a composite state, which is composed of sub-states. A composite state can have one or several active sub-states at the same time. Transitions between states can be triggered by external or internal events. An action (effect) can also be activated by the trigger while transitioning from one state to another. A state can have associated actions such as *entry/exit/doActivity* executed when the state is entered/exited or while it is active, respectively. A pseudo state is a transient vertex in UML State Machine diagrams to connect multiple transitions into more complex state transitions paths.

Tools such as IBM Rhapsody [5] and Papyrus-RT [1] generate code from UML State Machines. However, modifications in generated code cannot be reflected to the models. This is because there is no trivial mapping between UML State Machine concepts and object-oriented code.

Our XSeparation presented in the following subsection will generate code which enables the bidirectional traceability between UML State Machine concepts and object-oriented code.

B. Application of XSeparation for UML State Machine

XSeparation generates and synchronizes **Behavior-prescribed code** and **State machine action code** from UML State Machines describing the behavior of a component. As previously discussed, one of the main issues that hinders the synchronization of UML State Machines and code is the lack of mapping between these artifacts. Applying XSeparation to UML State Machines, code generated by XSeparation is at the right abstraction level, which contains additional constructs for state machines for tracing from the code back to the model.

Back to the UML State Machine example in Fig. 3 for illustration, XSeparation generates code as in Listing 4. The **Behavior-prescribed code** is divided into three parts: topology, events, and transition table, which are described in the belows.

Topology: A topology describes the hierarchical structure of a UML State Machine in generated code. As the example in Listing 4, the root of the topology is define by *state_machine*. Other elements such as *region*, *state*, and *pseudo state* are defined as sub-elements. The followings give the syntax of some elements in the generated code and the semantics mapped to the well-defined semantics in the UML specification [22].

[1]It is possible to have several UML State Machines for modeling the behavior within a component but for simplication, we consider only one.

Listing 4. Code generated from the FIFO State Machine

```
class FIFO : public IPush, IPull {
public:
  ProvidedPort<IPush> pPush;
  ProvidedPort<IPull> pPull;
  //Behavior-prescribed code
  Statemachine FIFOMachine {
    InitialState Idle;
    State SignalChecking {
      StateEntry entryCheck;
      StateExit exitCheck;
    };
    State ErrorNotification {
      StateEntry entryError;
    };
    State DataQueuing {
      StateEntry entryQueue;
      PseudoChoice queueChoice;
      State Queuing;
    };
    State Discarding;
    PseudoChoice dataChoice;
    CallEvent(DataPushEvent, push);
    TransitionTable {
      Transition(Idle, SignalChecking,
        DataPushEvent, NULL, signalCheckingEffect);
      Transition(SignalChecking, dataChoice,
            NULL, NULL, NULL);
      Transition(dataChoice, queueChoice,
            NULL, isDataValid, NULL);
    }
    configuration {
      QueueSize = 50;
      PeriodicTime = 20;
    }
  };

  //State machine action code
  void entryCheck(){
    //action code for entry of SignalChecking
  }
  void exitCheck(){
    //action code for exit of SignalChecking
  }
  void entryError(){
    //action code for entry of ErrorNotification
  }
  void signalCheckingEffect(Data& item) {
    //effect for transition from Idle to
    SignalChecking
  }

}
```

1) State and regions:

Syntax:

- *state machine* → 'Statemachine' name {subvertices}';'
- *state* → 'State' {state_action}; subvertices
- *state* → 'InitialState' name {state_action;effect?;subvertices}
- *concurrent state* → 'State' name {state_action; regions}
- *regions* → region; regions
- *region* → 'Region' name { subvertices };
- *state_action* → 'StateEntry/StateExit/StateDoActivity' actionName
- *subvertices* → state/concurrent state/pseudo state; subvertices
- *effect* → 'TransitionEffect' effName;

Semantics:

- *name*: the unique identifier of a state machine, a state, or a region.
- *actionName*: The name of the entry/exit/doActivity action method of the state. This method is implemented in the corresponding class, whose behavior is described by the state machine, and has no parameter.
- *initial_state*: A state is defined as an initial state, which has an incoming transition outgoing from a pseudo initial state within the same region or composite state.
- *effName*: For initial state, this is the transition effect associated with the initial transition.

- *concurrent state*: The representation of a concurrent state. The latter is composed of a set of regions. Each region contains a set of vertices, which for each is either a state or a pseudo state.

Example: Lines 6-35 in Listing 4 shows the code generated from the FIFO UML State Machine Fig. 3. *Idle* is defined as the initial state. The *SignalChecking* and *ErrorNotification* states (lines 8-14) are declared with methods written the **State machine action code** part following the **BP-U Agreement** for state. The latter is specified as following: the entry/exit/doActivity action of a state, if declared within the topology, must be implemented as a method of the component/class. For example, *entryCheck*, *exitCheck*, and *entryError* are implemented in the component class FIFO (lines 37-46).

2) Pseudo state:

Syntax: Pseudo states have similar syntax as below. In this latter, *pseudo_keyword* type is one of *{PseudoEntryPoint, PseudoExitPoint, PseudoInitial, PseudJoin, PseudoFork, PseudoChoice, PseudoJunction, PseudoShallowHistory, PseudoDeepHistory, PseudoTerminate}*, which correspond to the pseudo states defined in UML State Machine. *name* is the unique name of the pseudo state.

- *pseudo state* → 'pseudo_keyword' name

Example: *PseudoChoice(dataChoice)* and *PseudoChoice(queueChoice)* in Listing 4 represent the generated code for two *choice* pseudo states in the FIFO UML State Machine example.

Events: UML State Machine specifies four event types: *CallEvent*, *TimeEvent*, *SignalEvent*, *ChangeEvent*. The code generated by XSeparation for these events has syntax as followings:

- *CallEvent* → 'CallEvent' '('name, op');'
- *TimeEvent* → 'TimeEvent' '('name, dur');'
- *SignalEvent* → 'SignalEvent'<sig> name
- *ChangeEvent* → 'ChangeEvent' '('name, expr');'

Semantics: Essentially, each field in the syntax carries known semantics defined in the UML specification.

name	The unique identifier for an event.
op	The name of the operation associated with a *CallEvent* and implemented in the active class.
dur	The duration associated with a *TimeEvent* and specified as millisecond.
sig	The name of the signal class type (a UML signal is transformed into an object-oriented class) associated with a *SignalEvent*.
expr	The expression associated with a *ChangeEvent*. This expression is periodically evaluated to check whether its boolean value is changed.

Transition table: There are three kinds of UML transitions: *external*, *local*, and *internal*.

Syntax:

- *external* → 'Transition' '('src, tgt, guard, evt, eff');'
- *local*→'LocalTransition('src,tgt,guard,evt,eff');'
- *internal* → 'IntTransition('src, guard, evt, eff');'

Semantics:

src/tgt	The name of the source/target vertex of the transition. This name must be defined in the topology.
guard	A boolean expression representing the transition's guard and NULL If the transition is not guarded.
evt	The name of the event triggering the transition. evt must be one of the defined events as above or NULL if the transition is not associated with any event.
eff	The name of the method, which defines the effect of the transition or NULL if the transition has no effect. For **BP_U Agreement**, this method must be implemented by the component. If *evt* is a *SignalEvent*, the method has an input parameter typed as the signal class associated with the event. If *evt* is a *CallEvent*, the method has the same formal parameters as the method associated with *evt*.

Example: *CallEvent(DataPushEvent, push)* at line 22 of Listing 4 specifies that an event is fired whenever the *push* method, which is implemented by *FIFO* for the *IPush* interface provided by the *pPush* port, is invoked. The event firing activates the transition from *Idle* to *SignalChecking* and executes the *signalCheckingEffect* method associated with the transition if the current active state is *Idle*. If so, the data item brought by the invocation will be checked for validity and further put to the queue or discarded. Note that *signalCheckingEffect* has the same formal parameters with the *push* method.

State machines generated by XSeparation will be compiled to executable files by using our XSeparation compiler, which is presented in Section VII. The executable files run in an asynchronous mode, in which according to UML State Machine, events, except CallEvent, are stored in an event queue. Furthermore, the boolean expression of ChangeEvent is periodically evaluated. The size of the event queue and the periodic evaluation time can be configured within the topology of the state machines. Lines 31-34 in Listing 4 configures the size of the queue as 50 and the periodic evaluation time is 20 millisecond. Note that this configuration is different from the structure configuration, which is used for wiring different components through explicitly defined ports.

In the next section, we will show how to synchronize the XSeparation-generated code with the architecture model in case of concurrent modifications.

VI. Synchronization

In our previous work [XX], a methodological pattern is proposed to synchronize model and object-oriented code in case of concurrent modifications made in the model and code. The pattern especially requires the availability of several use-cases as followings:

- **Batch code generation**: generates and overwrites any existing code from model.
- **Incremental code generation**: updates the code by propagating changes from the model to the code.
- **Batch reverse engineering**: creates and overwrites any existing model from code.
- **Incremental reverse engineering**: updates the model by propagating changes from the code to the model.

The batch code generation and reverse engineering are straightforwardly supported by using the bidirectional traceability between the architecture model and code created by XSeparation. The incremental code generation (ICG) and incremental reverse engineering (IRE) needs a classification and management of modifications made in the model and code.

Model modification classification and management: Table I shows our classification and management of actions for propagating modifications in model to code. They can be detected by a model listener. We make a distinction between structural and behavioral modifications, which result in creating/removing/regenerating the corresponding code part. Although only add/remove/update are detected, the moving of a model element can be combined as a removal following by an addition. Some particular modifications requires the corresponding actions to respect and preserve the user code. For example, if the *sendDataToFifo* method in Listing 2 is renamed to *sendDataFromProducerToFifo* at the model level, the corresponding action consists of several steps: (1) identify the method, at the code level, associated with the operation, at the model level, using the old name *sendDataToFifo*, which is recorded by the model listener; (2) rename the method to *sendDataFromProducerToFifo* while keeping its parameters and body intact.

Code modification classification and management: Modifications types in code are similar to that of model consisting of structural and behavioral modifications. However, we do not support the removal of classes in code because this kind of modification causes conflicts and syntax errors, which require the code to be manually re-factored for reconciliation. We believe that an automatic mechanism brought by the regeneration of code caused by the removal can better handle the refactoring. If the FIFO class is removed in the code, for example, the *fifo* in *System* in Listing 1 must be retyped or removed. If not, a compilation error is raised.

Using these use-cases and the bidirectional traceability facilitated by XSeparation during code generation, the concurrent modifications in model and code can be synchronized by our previous methodological pattern.

After synchronization, to be executable, the XSeparation-generated code needs to be compiled. In the next section, we will present the architecture of our XSeparation compiler for compilation.

VII. XSeparation Compiler

XSeparation generates object-oriented code containing additional constructs for bidirectional traceability. Hence, a compilation process is needed to transform the XSeparation-generated code into machine code. This section presents the architecture of the XSeparation compiler for this compilation process.

Fig. 6 shows the compilation process, which takes as input the XSeparation-generated code, and produces the binary files by three two steps: ① generation of executable object-oriented code by the XSeparation compiler from XSeparation-generated code; ② production of binary files from the executable code by using object-oriented compilers such as GCC for C++, Javac for Java.

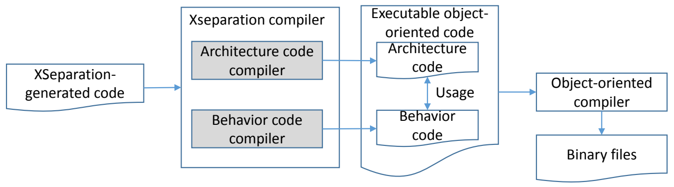

Fig. 6. XSeparation compiler's architecture

The XSeparation compiler consists of two grayed submodules: **Structure code compiler**, which takes as input the **Component structure-prescribed code**, **User-filled skeleton code**, and **Component structure-provided implementation code** parts to produce **Structure code**, while **Behavior code compiler** takes as input the other parts to generate **Behavior code**.

Listing 5. Executable code generated by XSeparation compiler for the structure of *System*

```
1  class System {
   public:
3    Producer p;
     Consumer c;
5    FIFO fifo;
     void configuration() {
7      p.pPush = &fifo;
       c.pPull = &fifo;
9    }
   };
11 class Producer {
   public: IPush* pPush;
13 }
   class Consumer {
15 public: IPull* pPull;
   }
```

Structure code compiler: Listing 5 shows an executable **Structure code** segment generated for the *System* example using ports with interfaces. Each port is generated to a pointer attribute (lines 12 and 15) while each part to an object attribute (lines 3,4,5). A configuration is transformed into a method named *configuration*. When a method is called through a port as in Listing 2, for example *push* through the *pPush* port, the corresponding method implemented in FIFO is invoked.

Behavior code compiler: The significance of the **Behavior code compiler** is the ability to generate executable code from the **Behavior-prescribed code** and **State machine action code**. It is similar to the code generation from UML State Machines in tools such as Rhapsody and Sinelabore [16]. However, usually, only a subset of UML State Machine concepts is supported by these tools, e.g. Rhapsody does not support junctions, and truly concurrent execution of orthogonal regions [17] (see [18] for more detail) and the support for pseudo states such as history, choice and junction is poor [20], [16]. The concurrency of the orthogonal regions is often implemented sequentially [19].

XSeparation compiler, on the other hand, supports code generation from state machines with full features. Due to space limitation, the details of patterns and evaluation for state machine code generation are not presented in this paper.

In the next section, we will implement XSeparation as an extension of the Papyrus modeling tool and evaluate it by

TABLE I
MODEL CHANGE CLASSIFICATION AND MANAGEMENT

Element type		Modification type	Action
Structure	Part/Port/Connector	Add/Remove/Update	Regenerate Component structure-prescribed code
	Class/Component/Interface	Add/Remove/Update	Create/Remove/Update the corresponding code
	Property	Add/Remove/Update	Create/Remove/Regenerate the corresponding class attribute
Behavior	Operation	Add/Remove/Update	Create/Remove/Regenerate the corresponding method with keeping its method body
	UML State Machine	Create	Generate Behavior-prescribed code and State machine action code
		Remove	Remove Behavior-prescribed code and State machine action code from the corresponding component
		Update	Regenerate Behavior-prescribed code and State machine action code with respect to the existing user-code
	UML State Machine concept (state, transition, pseudo state, event)	Create/Remove/Update	Regenerate Behavior-prescribed code and State machine action code with respect to the existing user-code

developing a case study of software application for LEGO.

VIII. Evaluation

This section presents our experiments with a case study of developing a software application for LEGO to evaluate the feasibility and usability of XSeparation. The latter is implemented as an extension of the Papyrus modeling tool, which is based on Eclipse. Architecture models in UML and XSeparation-generated C++ are synchronized. Our XSeparation compiler supports POSIX systems, in which pthreads are used for executing concurrency specified in UML State Machines. For incremental code generation in XSeparation, a model listener based on IncQuery is hooked to the modeling tool to detect model modifications.

A. Case study development

B. Usability

IX. Comparison and Related work

A. Round-trip engineering and co-evolution

Some RTE techniques restrict the development artifact to avoid synchronization problems. Partial RTE and protected regions are introduced in [23] to preserve code editions which cannot be propagated to models. The mechanisms as discussed in Section I are used for the separation of generated and non-generated code. EMF implements these techniques to allow users to embed user-code replacing the default generated code. Yu et al. [13] propose to synchronize user-code and generated code through bidirectional transformation. This latter does not, however, allow modifications in regions beyond the marked ones and thus prohibits the programmers from changing USMs' topology. *Deep separation* proposed in [10] overcomes the limitations of the separation mechanisms. However, it does not allow to modify the system architecture-prescribed code in text-based development environment.

The idea of adding more constructs for object-oriented languages to contain architecture information in XSeparation is similar to ArchJava [24] and Archface [8]. ArchJava is a Java-based language extension, which adds structural component concepts such as part and port to Java to support the co-evolution of architecture structure and Java implementation. Archface is a contract description language, which supports components and connectors, between design and implementation using concepts such as *pointcut* in Aspect programming to reason about the design and code correctness. These approaches, however, do not provide the traceability between the behavior of the architecture and code. The user-code and generated code are not separated as in XSeparation and *1.x-way mapping* to allow modifications made in both architecture and code. They also do not have a synchronization mechanism in case of concurrent development. Furthermore, the communication between two ports uses method calls of object-oriented languages instead of interfaces conforming to the UML specification. In addition, while ArchJava makes it not Java anymore and facilities of Java's IDEs such as intelligent completion are not aware, the XSeparation-generated code in our implementation can use all IDE functionalities.

Regarding the co-evolution of component-based model and code, the authors in [25] use a super model, from which source code and component model can be derived as views for modifications. However, code modifications made outside of their (limited) editor and violating their rules, e.g. only method bodies allow to be modified in code, are not updated to the super model.

The authors in [26] propose to synchronize code with platform specific models by using a three-way merging approaches. However, the approach only deals with syntactic synchronization while the code semantics such as state machines in source code is not taken into account.

B. Language engineering

Several languages such as PlantUML [27], Umple [28], and Earl Grey (EG) [29] support the text-based modeling (Textual ML) of UML. UML class and State Machine elements are usually available in these languages. However, they lack the explicit support for event types definitions used in UML. Furthermore, these languages do not allow the programmers to reuse the existing syntax of C++ but redefine it in their own language and IDE. By this way, they need a new complete compiler, which requires a lot of engineering tasks to develop. Contrarily, XSeparation integrates the modeling features into C++, only requires a light-weight compiler, and enables using legacy code in development. XSeparation allows the programmers stay with their familiar C++ syntax and existing favorite

IDEs. Hence, XSeparation profits all benefices of IDEs such as intelligent completion and easy to implement. Furthermore, XSeparation allows to use all specific C++ features such as function pointers for program performance, which are not available in the Textual MLs.

mbeddr [30] proposes an extensible C-based programming language. The idea of *mbeddr* is similar to Umple's by introducing a new editor to mix high-level and low-level code for effective embedded system development. *mbeddr* is a code-centric approach and UML event types are not explicitly supported whereas XSeparation enables a model-code hybrid and concurrent development and maintenance approach focusing on the collaboration between software architects and programmers, and the evolution of the artifacts.

In [31], the authors propose to integrate graphical and textual editors for UML profiles. The goal is to allow developers to work graphically and textually, which is similar to our goal. However, this approach is dependent on Eclipse technologies and embeds all modeling concepts to textual editors while XSeparation only introduces partially to allow programmers to be familiar with the syntax of their favorite programming language.

X. Conclusion

We have presented XSeparation- code generation, synchronization, and compilation, which enable round-trip engineering of architecture model and code for developing and maintaining reactive event-driven software application using Model-Driven Engineering. XSeparation code generation improves the bidirectional traceability between architecture model and code by proposing to generate code at an appropriate level of abstraction. The architecture model and XSeparation-generated code are then used as input in a model-code synchronization methodological pattern. An XSeparation compiler was developed to compile the XSeparation-generated code. XSeparation is implemented based on the Papyrus modeling tool for the case of UML and C++.

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.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

References

[1] E. Posse, "Papyrusrt: Modelling and code generation."

[2] J. O. Ringert, B. Rumpe, and A. Wortmann, "From software architecture structure and behavior modeling to implementations of cyber-physical systems," in *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, vol. P-215, 2013, pp. 155–170.

[3] S. Kent, "Model driven engineering," in *International Conference on Integrated Formal Methods*. Springer, 2002, pp. 286–298.

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

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

[6] B. Selic, "What will it take? a view on adoption of model-based methods in practice," *Software & Systems Modeling*, vol. 11, no. 4, pp. 513–526, 2012.

[7] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 633–642.

[8] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: a contract place where architectural design and code meet together," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 75–84.

[9] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[10] Y. Zheng and R. N. Taylor, "Enhancing architecture-implementation conformance with change management and support for behavioral mapping," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 628–638.

[11] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[12] Y. Zheng and R. N. Taylor, "A classification and rationalization of model-based software development," *Software & Systems Modeling*, vol. 12, no. 4, pp. 669–678, 2013.

[13] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "Maintaining invariant traceability through bidirectional transformations," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 540–550.

[14] R. N. Taylor and A. van der Hoek, "Software design and architecture the once and future focus of software engineering," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 226–243. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.21

[15] R. Van Der Straeten, T. Mens, and S. Van Baelen, "Challenges in model-driven software engineering," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 35–47.

[16] SinelaboreRT, "Sinelabore Manual," http://www.sinelabore.com/lib/exe/fetch.php?media=wiki:downloads:sinelaborert.pdf. [Online]. Available: http://www.sinelabore.com/lib/exe/fetch.php?media=wiki:downloads:sinelaborert.pdf

[17] IBM, "IBM Rhapshody and UML differences," http://www-01.ibm.com/support/docview.wss?uid=swg27040251, 2016, [Online; accessed 04-July-2016].

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

[19] O. Badreddin, T. C. Lethbridge, A. Forward, M. Elasaar, and H. Aljamaan, "Enhanced Code Generation from UML Composite State Machines," *Modelsward 2014*, pp. 1–11, 2014.

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

[21] R. Pilitowski and A. Dereziñska, *Code Generation and Execution Framework for UML 2.0 Classes and State Machines*. Dordrecht: Springer Netherlands, 2007, pp. 421–427. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-6268-1_75

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

[23] K. Czarnecki, M. Antkiewicz, and C. H. P. Kim, "Multi-level customization in application engineering," *Communications of the ACM*, vol. 49, no. 12, p. 60, Dec. 2006.

[24] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE, 2002, pp. 187–197.

[25] M. E. Kramer, M. Langhammer, D. Messinger, S. Seifermann, and E. Burger, "Change-driven consistency for component code, architectural models, and contracts," in *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. ACM, 2015, pp. 21–26.

[26] 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*, Mar. 2008, pp. 463–472.

[27] "Plantuml," http://plantuml.com/.

[28] T. C. Lethbridge, A. Forward, and O. Badreddin, "Umplification: Refactoring to incrementally add abstraction to a program," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 220–224.

[29] M. Mazanec and O. Macek, "On general-purpose textual modeling languages." Citeseer, 2012.

[30] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "mbeddr: an extensible c-based programming language and ide for embedded systems," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 2012, pp. 121–140.

[31] S. Maro, J.-P. Steghöfer, A. Anjorin, M. Tichy, and L. Gelin, "On integrating graphical and textual editors for a uml profile based domain specific language: An industrial experience," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2015. New York, NY, USA: ACM, 2015, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2814251.2814253