XSeparation: A Code Generation Approach for Bidirectional Traceability between Model and Code

Van Cam Pham, Ansgar Radermacher, Sebastien Gerard, Shuai Li
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 efficient to capture and simplify the complexity in designing the behavior and structure, respectively, of event-driven architecture. Model Driven Engineering (MDE) tries to generate full code from executable models. To achieve it, models must contain very detailed information. Nevertheless, current MDE tools and approaches are not sufficient to describe fine-grained behavior of the architecture. Current MDE tools therefore allow to manually put action code directly within the architecture model by limited editors, in which code errors might be produced during generated code compilation. In this case, the programmers tend to directly modify the code in familiar integrated development environments. Furthermore, some tools only produce skeleton code which is then fine-tuned by programmers. The modifications in code in these cases might violate the architecture correctness, which raises the problem of consistency and synchronization 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 - a set of utilities for enabling the bidirectional traceability and synchronization between architecture model and code. We implemented XSeparation in 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. The ultimate goal of MDE is to create executable models for generating full implementation code.

On the one hand, to generate full code, models must contain very detailed information. Nevertheless, current MDE tools and approaches are not sufficient to describe fine-grained behavior of the architecture using models. Therefore, to achieve the goal of complete code generation, some industrial MDE tools put fine-grained action code within the architecture model as blocks of text. MDE tools usually support limited editors for manual fine-grained coding tasks, which might produce errors during compilation of the generated code. When it happens, the developers tend to directly modify/correct the generated code for successful compilation because the code can be opened and easily modified within a familiar integrated development environment of the developers. Furthermore, some architecture-centric MDE approaches [7] only produce skeleton code from architecture models. The generated code in this case must then be tailored by developers for fine-grained code. The modifications of generated code in these cases might violate the architecture correctness, which is not easy to detect due to the lack of a bidirectional traceability between the architecture and code [8].

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 tailor the current architecture or modify the code for various reasons such as code level optimization, bug fixing, refactoring. This results that the architecture model and code are concurrently modified.

The modifications made in model and code raise the consistency and synchronization problem. If the latter is not solved, modifications in the code are not reflected to the model. Consequently, 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 elements back to the model.

This paper describes **XSeparation** - an code generation approach which enables the bidirectional traceability between architecture model and code. Current programming language elements are at lower level of abstraction than software architectures. To establish a bidirectional traceability between architecture model and code, the core idea of XSeparation is to leverage an object-oriented language by creating additional constructs for expressing architectural information. The established traceability is then combined with our synchronization methodological pattern presented in [12] for synchronization.

From a research perspective, this study aims to improve flexibility in MDE to allow the architecture model and the generated code to co-evolve while keeping these two consistent [9]. Furthermore, R. N. Taylor et al. [10] pointed out an important research direction, in which key design decisions

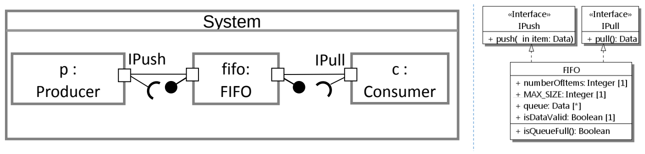

Fig. 1. Architecture of System (left) and Class diagram of FIFO (right)

may be made in implementation (code) and evolution of architecture must be seamlessly propagated to the code [5]. 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 [11].

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 III presents the overview of XSeparation code generation. Section IV, V, **??**, and VI describe the details of XSeparation including: XSeparation for architecture structure, behavior, synchronization, and compilation, respectively. The development of a case study is presented in Section VII to evaluate XSeparation. Section VIII discusses related work. The conclusion and future work are presented in Section IX.

II. Illustrative example

We consider a producer-consumer example developed using component-based engineering. Fig. 1 (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. 1 (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 items and the status of the queue.

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. 2. 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.

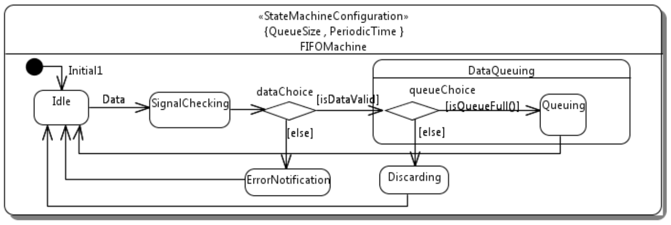

Fig. 2. FIFO's UML State Machine

III. 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. 3 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 in the belows.

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* (see V for details).

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 **Component structure-prescribed code** contains a required port with interface, which is not delegated (connected) to an other port of one of the sub-components (see Section IV for more details), **Component structure-provided implementation** is generated.

Although the five parts are ideally separate from each other as in Fig. 3, these parts, except **Behavior-prescribed code**, can be populated in an interleaved way.

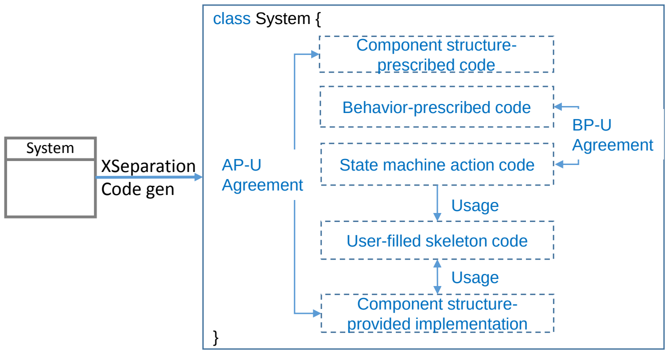
System
XSeparation
Code gen
class System {
Component structure-prescribed code
Behavior-prescribed code
State machine action code
User-filled skeleton code
Component structure-provided implementation
AP-U Agreement
BP-U Agreement
Usage
Usage
}

Fig. 3. XSeparation overview

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

IV. XSeparation for Architecture structure

XSeparation generates **Component structure-prescribed code** and **Component structure-provided implementation**. **Component structure-prescribed code** is derived UML component modeling concepts such as component, connector, part, and port. These concepts are not directly mapped to the object-oriented code. XSeparation customizes 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 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.

Each part is typed as class attributes. A UML port is either required with a unique required interface, or provided with a unique provided interface, or bidirectional with one required and one provided interface, which is transformed into an object of either **ProvidedPort** or **RequiredPort** or **BidirectionalPort**, respectively. 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 classes with all of its methods defined as *pure virtual*. In AP-U Agreement mentioned in Section III, **Component structure-provided implementation** should consist of concrete realizations of the *push* and *pull* methods, which are originally declared in the two interfaces *IPush* and *IPull* (class in C++).

A connector between two ports can be assemble, if the two parts containing the ports are in the same parent component, or delegate, if one of the parts is itself a part of the other.

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

```
1  class System {
2  public:
3    Part<Producer> p;
4    Part<Consumer> c;
5    Part<FIFO> fifo;
6    configuration {
7      bindPorts(p.pPush, fifo.pPush);
8      bindPorts(c.pPull, fifo.pPull);
9    }
10 }
11 class IPull {
12 public:
13   virtual Data* pull() = 0;
14 }
15 class IPush {
16 public:
17   virtual void push(Data& data) = 0;
18 }
19 class Producer {
20 public:
21   RequiredPort<IPush>* pPush;
22 };
23
24 class Consumer {
25 public:
26   RequiredPort<IPull>* pPull;
27 };
28 class FIFO : public IPush, IPull {
29 public:
30   ProvidedPort<IPush>* pPush;
31   ProvidedPort<IPull>* pPull;
32   Data* pull(){
33     //fine-grained code for pull
34   }
35   void push(Data& data){
36     //fine-grained code for push
37   }
38   int numberOfItems;
39   const int MAX_SIZE = 100;
40   Data queue[MAX_SIZE];
41   bool isDataValid;
42   bool isFullQueue() {
43     //fine-grained code
44   }
45 }
```

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

Listing 2. Code for interacting between components

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

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. 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 *fifo* by invoking the *push* method through the *pPush* port and the consumer actively pulls items by invoking *pull* through *pPull*.

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

For example, let's replace the bound ports with interfaces of the producer and fifo with data ports, in which the producer's port provides and that of the *fifo* receives data items. The other ports of the *fifo* and the consumer are not changed. The code generated by XSeparation for the data ports is shown in Listing 3. The *p* producer provides data items to *fifo* through their respective ports *pProvideData* and *pRequireData*.

Listing 3. System using data ports

```
1  class Producer {
2  public:
3    ProvidedDataPort<Data>* pProvideData;
4
5    void sendToFifo(Data& item) {
6      pProvideData->sendSignal(item);
7    }
8  }
9  class FIFO : public IPush, IPull {
10 public:
11   RequiredDataPort<Data>* pRequireData;
12   ProvidedPort<IPull>* pPull;
13 }
```

In implementation, the producer provides data items to the fifo via the port *pProvideData* by calling *pProvideData->sendSignal(item)* (line 6) as an object-oriented way. In case that the behavior of the component is described by a state machine, the *sendSignal* method will fire a signal event in order for the state machine to handle it.

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. Although the latter help to modularize the code generation process, hence make it easy to implement, the traceability from the code back to the model becomes very hard because the abstraction gap degree increases at each transformation.

Fig. 4 shows how different XSeparation is from the other approaches. We argue that synchronization of architecture model and executable code is a very hard problem, even impossible because component concepts such as part, port, and connectors are not directly mapped to object-oriented code. To deal with it, XSeparation generates code at an intermediate abstraction level. We propose to preserve the abstraction level of these concepts by representing and introducing additional equivalent classes/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.

Although introducing additional constructs is not really new when compared to Archface [8] and ArchJava [13], these approaches are very language-specific. XSeparation is, on the other hand, purely object-oriented, hence better suitable to object-oriented programmers. Furthermore, the programmers can write interaction code between components (Listing 2)

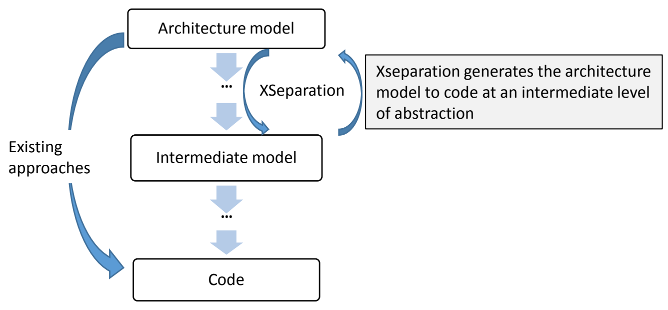
Architecture model
XSeparation
Xseparation generates the architecture model to code at an intermediate level of abstraction
Existing approaches
Intermediate model
Code

Fig. 4. Comparison of XSeparation with other code generation approaches

by the way, which fully profits advantages such as auto-completion of familiar integrated development environments such as Eclipse without having to install any additional tools/plug-ins. XSeparation also enables full integration with legacy code and interaction with existing frameworks.

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. [10] as previously discussed.

XSeparation, gives programmers the ability 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 architecture behavior to generate code (**Behavior-prescribed code** and **State machine action 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 has vertexes (state and pseudo state) and well-defined conditional transitions to efficiently describe the behavior of a component [1], [2]. 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 state. A state can have associated actions such as

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

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.

XSeparation in the following enables the bidirectional traceability between UML State Machine concepts and object-oriented code.

B. Application of XSeparation for UML State Machine

Applying XSeparation to UML State Machines, XSeparation-generated code contains our syntactic sugar of additional constructs for explicitly representing domain-specific concepts of state machines such as states and transitions. The goal is to embed specific concepts of state machines as internal programming concepts into general-purpose host programming languages such as C++ and Java so that the embedded concepts are valid syntaxes. By this way, from a programmer's perspective, tooling supports for programming facilities are available with any IDE. Furthermore, state machine concepts are so expressive that UML State Machines can be reconstructed from the code.

Listing 4. Code generated from the FIFO State Machine

```
1  class FIFO : public IPush, IPull {
2  public:
3    ProvidedPort<IPush>* pPush;
4    ProvidedPort<IPull>* pPull;
5    //Implementation of push and pull
6    //Behavior-prescribed code
7    Statemachine FIFOMachine {
8      InitialState Idle;
9      State SignalChecking {
10       StateEntry entryCheck;
11       StateExit exitCheck;
12     };
13     State ErrorNotification {
14       StateEntry entryError;
15     };
16     State DataQueuing {
17       StateEntry entryQueue;
18       PseudoChoice queueChoice;
19       State Queuing;
20     };
21     State Discarding;
22     PseudoChoice dataChoice;
23     CallEvent(DataPushEvent, push);
24     TransitionTable {
25       Transition(Idle, SignalChecking,
26         DataPushEvent, NULL, signalCheckingEffect);
27       Transition(SignalChecking, dataChoice,
28             NULL, NULL, NULL);
29       Transition(dataChoice, queueChoice,
30             NULL, isDataValid, NULL);
31     }
32     configuration {
33       QueueSize = 50;
34       PeriodicTime = 20;
35     }
36   };
37
38   //State machine action code
39   void entryCheck(){
40     //action code for entry of SignalChecking
41   }
42   void exitCheck(){
43     //action code for exit of SignalChecking
44   }
45   void entryError(){
46     //action code for entry of ErrorNotification
47   }
48   void signalCheckingEffect(Data& item) {
49     //effect for transition from Idle to SignalChecking
50   }
51
52 }
```

For the example in Fig. 2 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 state machine hierarchy in XSeparation-generated code. In Listing 4, the root of the topology is a *Statemachine*. Other elements such as *region*, *state*, and *pseudo state* are hierarchically defined as sub-elements. Xseparation-generated code for a state resembles to class declaration except that the state uses its own keyword *State* instead of *class*. Each element has a unique name.

State actions such as state entry/exit/doActivity are declared as attributes of the state. These actions have relationships with **State machine action code** as **BP-U Agreement**. In the latter, the declared actions must be implemented in the corresponding class (the class containing the **State machine action code**) and has no parameter. For example, in Listing 4, *Idle* is an initial state, which can declare an initial transition effect implemented in **State machine action code**. The *SignalChecking* and *ErrorNotification* states (lines 9-15) are declared with state actions, whose names are identical with the methods written in the **State machine action code** part following the **BP-U Agreement** for state. The *entryCheck*, *exitCheck*, and *entryError* methods are implemented in the class FIFO (lines 38-47).

A concurrent state (not generated for the example) is composed of a set of regions. A region can be generated to **Region** *topRegion {vertices}*. Each region contains a set of vertices, which for each is either a state or a pseudo state.

Pseudo states have similar syntax as attribute declarations of a class. The type of pseudo state attributes is one of *{PseudoEntryPoint, PseudoExitPoint, PseudoInitial, PseudJoin, PseudoFork, PseudoChoice, PseudoJunction, PseudoShallowHistory, PseudoDeepHistory, PseudoTerminate}*, which correspond to the pseudo states defined in UML State Machine. For 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: Four UML event types including *CallEvent*, *TimeEvent*, *SignalEvent*, and *ChangeEvent* are supported.

- A *SignalEvent* is associated with a signal type *sig*, whose data are described by attributes, and occurs if an instance of *sig* is received by a component through its data port by invoking *sendSignal* as previously discussed.
- A *CallEvent* is associated with an operation *op* of the component class containing the state machine. *CallEvent* is emitted if there is an invocation to *op*.
- A *TimeEvent* specifies the time of occurrence *dur* relative to a starting time. The latter is specified as the time when a state, which accepts the time event, is entered. In other words, the state, which is the source vertex of a transition triggered by a time event, will remain active for a maximal amount of time specified by the time event.
- A *ChangeEvent* is associated with a boolean expression *expr*. *ChangeEvent* is emitted if the value of *expr* changes from false to true.

Call events are synchronous meaning that the processing runs within the thread calling the operation associated with a call event. Other events are asynchronous meaning that on receiving these events are stored in an event queue which is maintained at runtime for later processing. Note that not like call and signal events, time and change events are automatically fired inside the component.

Transition table: There are three kinds of UML transitions: *external*, *local*, and *internal*. The difference between these kinds is clearly stated in UML and beyond the scope of this paper. The syntax for these kinds is as followings:

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

The semantics of the syntax are as followings:

- *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*.

For example, *CallEvent(DataPushEvent, push)* at line 23 in 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.

Code generated for a signal event has syntax as **SignalEvent**<*sig*> *name*, in which *sig* is the name of the associated signal class (a UML signal is transformed into an object-oriented class). The signal type must be declared as required data of one of data ports of the component in **Component structure-prescribed code**. Sending of a signal instance to the port might fire an asynchronous signal event for a state machine to handle in case that the containing component behavior of the port is specified via the state machine. Due to space limitation, we do not go to details of TimeEvent and ChangeEvent.

Configuration: Asynchronous events are stored in an event queue managed by the component at runtime. Furthermore, the boolean expression of ChangeEvent is periodically evaluated. The size of the queue and the periodic evaluation time can be configured within the topology of the state machines. Others such as event priorities are configurable but not implemented for the moment. Lines 32-35 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. UML, however, remains abstractly and does not specify these configuration values. We then define a profile support for UML State machine configuration. Fig. 2 shows the configuration values annotating the state machine example.

Deferred event: A state can declare *deferred events* by introducing attributes typed by our additional construct *DeferredEvent* and named as event names to be deferred. The deferred events are used for state machine execution to delay

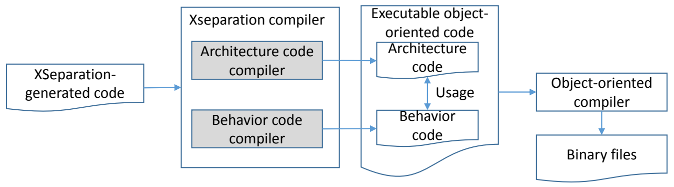

Fig. 5. XSeparation compiler's architecture

the processing of low-priority events. The execution semantics of deferred events is that: given a current active state with declared deferred events and an event queue, the deferred events, if in front of the queue, will be moved to a deferred set and pushed back to the front of the queue once a non-deferred event is processed. In other words, the deferred events will not be processed while the state remains active.

Discussion: Behavior-prescribed code is actually a textual declarative description of UML State Machines, which is supported by some UML textual languages such as PlantUML [**?**], Umple [14], and Earl Grey (EG) [15]. However, XSeparation does not create a completely new language as the UML textual languages. XSeparation for behavior follows the idea of introducing an internal programming language within an existing host programming language such as C++ and Java [16] so that programmers can use the combination of familiar programming languages and IDEs. Because XSeparation features legacy code reuse and traditional advantages of object-oriented languages such as inheritance. Tooling for supporting facilities such as auto-completion for coding state machines in the XSeparation-generated description is much more easier and requires less effort to build than other languages such as Umple (see VIII for more details). The generated code looks similarly to class hierarchies, which are not only familiar with programmers but also easy to understand the state machine topology.

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

VI. XSeparation Compiler

This section presents the architecture of the XSeparation compiler. Fig. 5 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++ and Javac for Java.

The XSeparation compiler consists of two grayed sub-modules: **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**.

Structure code compiler: A verification is firstly executed to check the well-formedness of components and ports. We basically verify three port rules: (1) every port with a required interface or provided data must be bound to another port; (2) if two ports are connected by a assemble connector, the provided interface/data of one port must identical to or an extension of the required interface/data of the other port; and (3) if the connector is delegate, the provided interface/data of one port must identical to or an extension of the provided interface/port of the other port;

Once the rules are verified, executable code can be generated. Listing 5 shows a **Structure code** segment generated from 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.

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

```
1  class System {
2  public:
3    Producer p;
4    Consumer c;
5    FIFO fifo;
6    void configuration() {
7      p.pPush = &fifo;
8      c.pPull = &fifo;
9    }
10 };
11 class Producer {
12 public: IPush* pPush;
13 }
14 class Consumer {
15 public: IPull* pPull;
16 }
```

Behavior code compiler: It generates executable code from the **Behavior-prescribed code** and **State machine action code** similarly to code generation approaches from UML State Machines in MDE tools such as Rhapsody and Sinelabore [17]. However, 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 [18] (see [19] for more detail) and the support for pseudo states such as history, choice and junction is poor [20], [17]. XSeparation compiler supports full features of state machines. XSeparation compiler has following features.

Completeness: XSeparation compiler supports all state machine vertexes and transitions including all pseudo states and transition kinds. Hence, XSeparation compiler improves flexibility of using state machines to express architecture behavior. For the moment, the only issue with the compiler is that it cannot deal with transitions from an entry point to an exit point.

Event support: XSeparation compiler utilizes four UML event types and deferred events, which are able to express synchronous and asynchronous behaviors and exchange data between components.

UML-conformance: A recent specification formalizing the precise semantics of UML State Machine is under standardization of OMG. It defines a test suite with 66 test cases for certifying the conformance of runtime execution of code generated from UML State Machines. We have experimented XSeparation compiler with the test suite. The traced execution results of 62/66 test cases comply with the standard and are a good hint that the execution is semantically correct. Due to space limitation, the details of patterns and evaluation for state machine code generation semantics are not presented in this paper.

State machine configuration: XSeparation allows to configure the event queue size and periodic time for evaluation of change events.

Event API: Generated code in XSeparation compiler provides APIs for environment code to invoke operations or send data signals through component ports. The invocations and sending will automatically fire events for state machines to process.

In the next section, XSeparation will be implemented as an extension of the Papyrus modeling tool and evaluated by developing a case study of software application for LEGO.

VII. Evaluation

This section presents our implementation prototype and experiments with a case study of developing a software application for LEGO to evaluate the feasibility and usability of XSeparation.

A. *Implementation*

XSeparation is implemented as an extension of the Papyrus modeling tool, which is based on Eclipse. Intermediate C++ code is generated from architecture models in UML. Our XSeparation compiler supports executable code executed within POSIX systems, in which pthreads are used for concurrency in UML State Machines such as doActivity, parallel regions, and infinite loop for event dispatching. The additional constructs added to C++ are engineered in a way that allows programmers to use facilities such as auto-completion of any C++ IDE to write fine-grained code without installing additional tools. The programmers can stay within their familiar IDE such as Eclipse CDT and Visual Studio. This is one of our advantages over other approaches which will be discussed in Section VIII.

In order to synchronize the modified intermediate code with the model, we apply our synchronization methodological pattern presented in [12]. Following this pattern, we implemented a synchronizer from code to model. It contains the following use-cases:

- **Batch code generation**: generates and overwrites existing intermediate code from model.
- **Incremental code generation**: propagates changes from the model to the mediate code.
- **Batch reverse engineering**: creates and overwrites any existing model from the intermediate code.
- **Incremental reverse engineering**: propagates changes from the intermediate code to the model.

The batch code generation and reverse engineering are straightforwardly supported by using the bidirectional traceability between the architecture model and the intermediate code created by XSeparation. The implementation of incremental code generation (ICG) and reverse engineering (IRE)

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

is based on management of modifications made in model and code.

Incremental code generation: A model listener is developed to detect changes in model. Table I shows our detected modifications and how to propagate these changes to the intermediate code. 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.

Incremental reverse engineering: Similarly to incremental code generation, we developed a code listener to detect code modifications in CDT. We detect (1) structural modifications such as addition/removal/update of part/port/attribute, and (2) behavioral modifications. Currently, our implementation does not support removal of C++ classes because it might cause conflicts and syntax errors. For example, if the FIFO class is removed in the code, the *fifo* in *System* in Listing 1 must be retyped or removed. If not, a compilation error is raised. Furthermore, for state machines, we only synchronize modifications in **State machine action code** back to the model.

After implementation, we then use the prototype with its use-cases to develop a software application for LEGO in order to evaluate XSeparation through the tooling prototype. The followings present the case study development for the applicability evaluation and a manual evaluation of the usability.

B. Case study-based evaluation

1) Evaluation purpose:

2) Case study development scenarios: Description of the application...

The case study was developed by a developer **Dev**. **Dev** used the Papyrus Designer tool, which features component-based and model-driven development in UML and full C++ code generation through embedding of fine-grained code as blocks of texts within a limited text editor.

During the development, an architecture model is created. However, the developer felt difficult, unfamiliar, and annoyed to write code with the limited editor. **Dev** then refused to use that editor and programmed in CDT to be familiar and effective with C++ facilities such as syntax highlights and auto-completion. **Dev** then copied the code from CDT to the model and regenerated the code. The developer felt inefficient, prone-to-error, and lack comprehension of the architecture information during code writing and copying.

Given the above issues, **Dev** wants to try a synchronization approach, which at least can automatically synchronize modifications in fine-grained behavior to the model.

Application of XSeparation: Two developers: **Dev** and another developer use Git to collaborate. Two distinct scenarios are emulated: (1) The developers modify both model and code; and (2) A collaboration scenario between **Dev**, as a programmer, and the other as a software architect.

Scenario 1: In this scenario, **Dev** managed the application revision by using Git. **Dev** used Papyrus Designer to create the architecture model, and generates intermediate and executable code from it. She then filled fine-grained behavior in the intermediate code using CDT. After that, **Dev** then used XSeparation to regenerate the executable code and compiled it. The modified intermediate code is then automatically synchronized back to the original architecture model by using XSeparation. Following each synchronization, **Dev** committed and pushed both of the model, the intermediate code, and the executable code to Git.

Scenario 2: **Dev** and an other developer **OtherDev** use Git to control versions of the case study project and work together. In the beginning, the component-based architecture structure model is created by the two developers using Papyrus Designer. Intermediate and executable code are then generated from the model by XSeparation. The model and codes are then pushed to the Git repository as an initial version. Each developer took in charge of development of several components. For each corresponding component, each of the developers then described its course-grained behavior via a UML State machine, regenerated the intermediate code, wrote fine-grained code for the component, and synchronized the modified intermediate code with the model by using XSeparation. After synchronization, the developer tried to pulled the artifacts from the remote Git repository. If the remote model was updated by the other developer, both of the models were merged with each other using the Papyrus model merger [21]. After merging, the intermediate code is then regenerated. The two artifacts are then pushed to the remote repository.

C. Findings

To evaluate the usability of XSeparation, several questions are used for asking developers for how effective XSeparation is when compared to full code generation approaches using fine-grained code directly embedded within models.

a) Is model-code synchronization in XSeparation better than full code generation with fine-grained code within models in general?:

b) Is modifying architecture at the code level more effective than at the model level?:

c) Is modifying fine-grained code at the code level better than code embedded within models for full code generation?:

d) Is concurrent development using XSeparation effective?: 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.

VIII. Comparison and Related work

A. Round-trip engineering and co-evolution

Some RTE techniques restrict the development artifact to avoid synchronization problems. Partial RTE is introduced in [22] to preserve code modifications which cannot be propagated to models. It separates non-generated code from generated areas. EMF implements these techniques to allow users to replace the default generated code with user-code. However, this mechanisms requires programmers to be highly discipline. Even so, there is still a possibility for the programmers to produce accidental changes, which cannot be reflected to the model [23]. Yu et al. [9] propose to synchronize user-code and generated code through bidirectional transformation. This latter does not, however, allow modifications in regions beyond the marked ones. *Deep separation* proposed in [23] overcomes the limitations of these separation mechanisms. However, it does not allow to modify the system architecture at the code level.

In [24], source code and component model can be derived as views from a super model for modifications in an editor for co-evolution. 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.

In [25], the authors integrate graphical and textual editors for UML profiles to allow developers to work in both of the representations. However, this approach is dependent and embeds all modeling concepts to textual editors while XSeparation only introduces necessary concepts in order to keep programmers being familiar with of their favorite programming language.

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

B. Language engineering

Text-based modeling languages (Textual ML) of UML such as PlantUML [**?**], Umple [14], and Earl Grey (EG) [15] support UML class and State Machine elements. However, these languages lack the explicit support for event types definitions used in UML. Furthermore, Textual MLs require a lot of engineering tasks to develop a tooling support such as compiler and editor. Contrarily, XSeparation only requires a light-weight compiler and enables using favorite IDEs. XSeparation does not impose programmers to change programming environments.

In [27] BSML-mbeddr is proposed as an state machine-based programming language integrated into the C language. Big and small steps are defined in the language semantics. However, the latter is not UML-compliant. BSML-mbeddr only specifies state and region concepts, and does not allow different event types as XSeparation does. Furthermore, BSML-mbeddr is a code-centric approach whereas XSeparation enables a model-code concurrent development, maintenance, and co-evolution.

ArchJava [13] adds structural component concepts such as part and port to Java. The Archface [8] contract description language 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 of the architecture behavior 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 as in UML. In addition, while ArchJava makes it not Java anymore and facilities of Java's IDEs such as auto-completion are not aware, XSeparation-generated code can use these facilities as discussed in IV.

IX. Conclusion

We have presented XSeparation- a set of utilities including 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 as an extension of the Papyrus modeling tool for the case of UML and C++.

We successfully applied XSeparation to the development of a case study. The latter is an software application for LEGO. An evaluation of usability of XSeparation was also conducted to assess that: XSeparation improved the work flow of Model-Driven Development because XSeparation allows model-code synchronization, which then maximizes the effectiveness and advantages of both programming and modeling

For the moment, the approach is implemented for UML and C++. In future, we want to instantiate XSeparation for Java to have synchronization of UML with a range of general-purpose object-oriented programming languages.

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] 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.

[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] 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.

[10] 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

[11] 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.

[12] V. C. Pham, S. Li, A. Radermacher, and S. Gérard, "Foster software architect and programmer collaboration," in *21th International Conference on Engineering of Complex Computer Systems, ICECCS 2016, Dubai, United Arab Emirates, November 6-8, 2016*, 2016, pp. 1–10.

[13] 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.

[14] 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.

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

[16] G. Hinkel, "Change propagation in an internal model transformation language," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9152, 2015, pp. 3–17.

[17] 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

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

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

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

[21] "Collaborative Modeling with Papyrus," http://eclipsesource.com/blogs/2015/04/13/collaborative-modeling-with-papyrus-emf-compare-and-egit/.

[22] 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.

[23] 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.

[24] 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.

[25] 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

[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] Z. Luo and J. M. Atlee, "Bsml-mbeddr: Integrating semantically configurable state-machine models in a c programming environment," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2016. New York, NY, USA: ACM, 2016, pp. 105–117. [Online]. Available: http://doi.acm.org/10.1145/2997364.2997372