Synchronization of Architecture Design Model and Code in Reactive System Development

Van Cam Pham, Ansgar Radermacher, Sebastien Gerard, Shuai Li
CEA, LIST, Laboratory of Model-Driven Engineering for Embedded Systems (LISE), P.C. 174, Gif-sur-Yvette, France
Email: first-name.lastname@cea.fr

***Abstract*—Architecting a software system has become one of the most important tasks during development. UML State Machine and Composite Structure are efficient to design reactive system architectures. In Model Driven Engineering (MDE), code can be automatically generated from the models.**

However, current UML tools and approaches are not sufficient to describe and generate the fine-grained behavior. Hence, these tools allow manually putting programming code as blocks of text embedded within the model for the fine-grained behavior.This manual coding practice in a limited editor loses programming flexibility and might produce errors during compilation of the generated code. In this case, programmers tend to directly modify the code using familiar integrated development environments. The modifications in code, which may violate the architecture correctness, must be synchronized with the model to make architecture and code consistent. Current approaches cannot handle the synchronization of model and code in case of UML State Machine and Composite Structure elements because there is a significant abstraction gap between architecture and code.

This paper proposes an approach to enable this synchronization. The approach consists of a bidirectional mapping between code and the architecture model, and a synchronization mechanism, which allows to synchronize concurrent modifications made in model and code. The approach is then evaluated through the development of an embedded software case study - a Lego Car factory. The evaluation shows that our approach can automatically synchronize model and code, and eventually improve collaboration between software architects and programmers.

I. Introduction

System architecting has become one of the most important tasks during development. Unified Modeling Language (UML) has been widely used in Model-Driven Engineering (MDE) [1] and become an industry de factor standard to describe and document architecture of complex systems [2], [3] despite the emergence and disappearance of a number of architecture description languages [4]. UML Class, Composite Structure, and State Machine diagrams prove to well capture a reactive system architecture [5], which is useful for designing flexible, loosely-coupled and scalable embedded systems [6], [7]. In MDE, implementation can be automatically produced from architecture models specified by the UML models in order to raise software productivity and eliminate bugs [5].

Current UML tools and approaches are not sufficient to exploit the fine-grained behavior of the architecture. The approach in [8] uses the Action Language for Foundational UML (ALF) [9] to express the fine-grained behavior for generating fully operational code. The approach in [10] even proposes a model compiler, which directly converts ALF code into binary code. However, ALF is currently not widely used in software development as much as mainstream programming languages such as Java or C++. Furthermore, with the model compilation, many optimizations at the code level, proposed by many tools and compilers such as GCC, become useless.

Industrial MDE tools such as IBM Rhapsody [11] and Papyrus-RT [5] put manually written fine-grained programming code such as C++, instead of ALF, within the architecture model to generate fully operational code. These tools force the programmers to use limited textual editors supported by these tools for manual coding. This later practice often lacks programming facilities such as auto-completion appeared in modern integrated development environments (IDEs) and might produce errors during compilation of the generated code with the manual code. In such a case the programmers tend to directly modify the generated code for a successful compilation by using a familiar IDE of the programmers.

In continuous development, 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 for algorithmic/computational code, bug fixing, and refactoring. This results that the architecture model and code are concurrently modified.

The modifications made in model and code must be synchronized to make the architecture and code consistent. To deal with it, several approaches such as separation [12] and reverse engineering [11] use specialized comments to separate user-modified code and generated code for preserving the user code. However, these approaches only work for modifications within the user area under assumption that the programmers are highly disciplined [41]. Some tools such as IBM Rhapsody are able to do round-trip engineering: modifications in code can be propagated back to model and vice versa. However, only UML class diagram concepts are supported by the round-trip engineering tools. One of the reasons, that makes the propagation of the modifications in the code back to the model hard, is the lack of a bidirectional mapping between the architecture model specified by the aforementioned models and code [13].

This paper addresses the synchronization of code and architecture design model specified by the aforementioned UML models. Our approach consists of a mechanism of bidirectional mapping between the architecture design model and code, and a synchronization mechanism. We argue that current program-

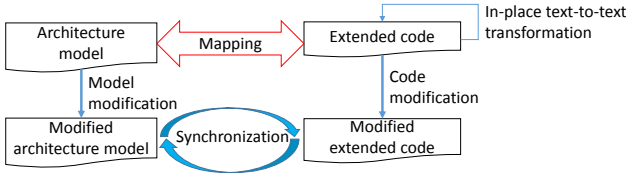

Fig. 1. Approach overview

ming language elements are at lower level of abstraction than software architectures. To establish a bidirectional mapping, our approach leverages the abstraction level of an existing object-oriented language by creating additional constructs for expressing architectural information. Our synchronization mechanism then uses the established mapping mechanism to propagate modifications in code back to model.

The contributions of this paper are as followings:

- A bidirectional mapping mechanism between architecture model and code.
- A synchronization mechanism which uses the proposed mapping as a means to ease the synchronization.
- An evaluation of the approach through a case study.

This paper assumes that the readers have knowledge about component-based and UML State machine and Composite Structure concepts.

The remainder of this paper is organized as follows: Section II describes the overview of our approach. The bidirectional mapping mechanism is presented in Section III. Section IV gives the details of the synchronization mechanism. A case study is used for evaluation of our approach in Section V. Section VI openly discusses scalability and perspectives of our approach. We discuss related work in Section VII. The conclusion and future work are presented in Section VIII.

II. Approach Overview

This section presents the overview of our approach. The latter consists of a mapping mechanism between architecture model and code, and a synchronization mechanism using the mapping as a means to synchronize the model and code.

Fig. 1 shows the overview of our approach. The mapping mechanism is formed by a bidirectional mapping and a text-to-text transformation and described as belows.

Architecture model: The architecture designed by using the UML Class, Composite Structure, and State Machine concepts. The architecture model might be modified by the software architects or indirectly updated by propagating code modifications back to the model.

Mapping: The mapping consists of a set of correspondences [14] between elements of UML and the extended language (see the extended language and code). This mapping is bidirectional and is used as a means to ease the model-code synchronization.

Extended language and code: To build a bidirectional mapping between architecture model and code, we raise the abstraction level of a standard object-oriented programming language, whose language elements are at a lower level of abstraction than architecture elements [15]. We tailor a standard object-oriented language by adding additional/ad-hoc programming constructs for the UML elements that have no direct representations in common programming languages, notable UML ports, connectors, and State Machine elements. We choose these model elements because ports and connectors are widely proposed in many Architecture Description Languages (ADLs) [16], and state machines are suitable to the modeling of the behavior of components in reactive systems. The *extended language* is the standard language with the additional constructs. It is as close as possible to the existing standard language in order to be suitable for programmers perception to minimize additional learning efforts. The **Extended Code** conforms to the *extended language*. Programmers can therefore change the architecture model or the fine-grained code behavior at the code level while the code modifications can be reflected to the model by our synchronization mechanism. The additional constructs are created by using specialized mechanisms of the standard language such as templates, and macros in C++ or annotations in Java. The **Extended code** containing the additional constructs syntactically conforms to the standard programming language. By this way, the **Extended code** can seamlessly reuse legacy code or library written in the standard programming language. This is especially important because current complex systems rely on a lot of library support [17].

In the following sections, we use the terms *Extended Code* and *code* interchangeably and *Extended language* refers to the standard language with our additional constructs.

In-place transformation: The additional programming constructs are used to ease the connections from code to architecture model. However, they are natively not executable. The in-place transformation utilizes the information in the extended code to complement the programming additional constructs so that these constructs are executable.

Model modification: Software architects make modifications/changes to the architecture design model during development.

Code modification: Programmers make modifications/changes to the extended code during development.

Synchronization: Once the model and/or the code are/is modified, the synchronization reflects modifications made in one artifact to the other artifact and vice-versa.

In the following sections, we give the details of the mapping and synchronization.

III. Bidirectional Mapping Mechanism

In this section, we describe the mapping mechanism, which consists of a bidirectional mapping (see III-A) and a text-to-text transformation (see III-B).

A. Bidirectional mapping through an example

We present here our bidirectional mapping between UML-based architecture models and code through a producer-consumer example, whose architecture is specified by Fig. 2 (a), (b), and (c). The *p* producer sends data items to a first-in first-out component *FIFO* storing data. The *FIFO* queue has a limited size, the number of currently stored items (*numberOfItems*) and the *isQueueFull* operation for validating

TABLE I
MAPPING BETWEEN UML AND EXAMPLES OF EXTENDED LANGUAGE (1)

UML	Extended Language	Code example in Fig. 2
Port requiring an interface *I*	Attribute typed by *RequiredPort*<I>	Ports *pPush* and *pPull* at lines 19 and 22
Port providing an interface *I*	Attribute typed by *ProvidedPort*<I>	Ports *pPush* and *pPull* at lines 26-27
Bidirectional port providing *R* and *P*	*BidirectionalPort<R,P>*	Not shown in this paper
Connector	Binding	Lines 7-8
State Machine	*StateMachine*	The FIFO state machine at lines 31-51
State	*State/InitialState*	State *SignalChecking* at lines 33-36
Region	*Region*	Not shown in this paper
Pseudo state	Attribute typed by pseudo type	The *dataChoice* pseudo state at line 42
Action/Effect	Method	Methods at lines 52-57
Transitions	Transition table	Transition table at lines 44-50
Event	Event	The call event at line 43
Deferred Event	State attribute typed by deferred event type	Not shown in this paper
Transition guard	Method returning a boolean	The valid method at lines 58-59 as the transition guard at line 49

its availability. The *pPush* port of the producer with *IPush* as required interface is connected to the *pPush* port of *FIFO* with *IPush* as provided interface. The producer and *FIFO* can interact with each other through their respective port. *FIFO* also provides the *IPull* interface for the consumer to get data items. *FIFO* implements the two interfaces as in Fig. 2 (b), lines 28-29.

The behavior of *FIFO* is described by using a UML State Machine as shown in Fig. 2 (c). Initially, the *Idle* state is active. The state machine then waits for an item to arrive at the *fifo* component (through the *pPush* port). The item is then checked for its validity before verifying the fullness of the queue to decide to either add to the queue or discard it.

Table I and II show some of the UML meta-classes and our ad-hoc equivalent constructs in the extended language. The constructs are categorized into *structural* (four upper rows in Table II and II) and *behavioral constructs* (nine lower rows in Table II). We explain these constructs in the followings.

1) Structural constructs: This section explains the programming constructs in the bidirectional mapping corresponding to the modeling concepts used for modeling the architecture structure, including *Port* and *Binding*.

Port: Template-based programming constructs proposed in the extended language correspond to architecture component interaction points specified by UML ports. *RequiredPort<T>* and *ProvidedPort<T>* (see Table I) are equivalent to UML unidirectional ports, which have only one required or provided interface. The *T* template parameter is an interface in code (e.g. *interface* in Java or class with *pure virtual* methods in C++) equivalent to the interface required/provided by a UML port. *BidirectionalPort<R,P>* (not shown) is also proposed to map to UML bidirectional ports, which have one required *R* and one provided *P* interface.

Lines 19 and 22 show ports with a required interface and lines 26-37 show ports with a provided interface of the *Producer*, *Consumer*, and *FIFO* classes respectively.

TABLE II
MAPPING BETWEEN UML AND EXAMPLES OF EXTENDED LANGUAGE (2)

UML and MARTE	Extended Language	Example in Listing 1
In flow port	Attribute typed by *InFlowPort<Sig>*	Ports *pInData* at line 10
Out flow port	Attribute typed by *OutFlowPort<Sig>*	Ports *pOutData* at lines 3 and 11
Bidirectional flow port	Attribute typed by *InOutFlowPort<Sig>*	Not shown in this paper
UML Signal	A class	Not shown in this paper

Binding: A binding (see Table I, row 4) connects two ports. It is equivalent to a UML connector connecting two UML ports. A binding is a method call to our predefined method *bindPorts*. Lines 7-8 show two invocations of *bindPorts*, which takes as input two ports (the two ports of the producer and fifo, for example). Each code class associated with a UML component contains a single configuration (as a method in lines 6-9) containing bindings. The configuration method is restricted to contain only invocations to *bindPorts* for synchronization ease.

Other elements in the UML class diagram are mapped to the corresponding elements as in industrial tools such as IBM Rhapsody and Enterprise Architect. For example, the *p, fifo, c* UML parts are mapped to the composite attributes of the *System* class; the UML operations and properties are mapped to the class methods and attributes (not shown in the paper), respectively; the UML interfaces (*IPush* and *IPull*) mapped to classes with pure virtual methods (lines 11-17) (in C++).

From a programming perspective, in order to call the services/methods provided by the interface of a provided or bidirectional port from a required or bidirectional port, we provide attribute interfaces as members of the port additional constructs as followings: a *requiredIntf* attribute (a class attribute in Java or a pointer attribute in C++, e.g.) typed by the required interface of the required or bidirectional port. For example, for calling the *push* method implemented by the fifo from the producer, a programmer can write *pPush.requiredIntf->push(data)* in fine-grained code of the producer.

Flow port: UML only provides service ports having provided and/or required interfaces. Some UML extensions/profiles such as MARTE [18] define *flow ports* to support both client-server like and data-flow like communication schemas. Flow ports enable message-driven and data flow oriented communication between components, where messages flowing across ports represent data items. A flow port can be *in/out/inout* and the flow of data items through the port can be from outside to inside/from inside to outside/bidirectional, respectively. The data items exchanged between components through flow ports are modeled in terms of UML signals. Our mapping mechanism also contains programming constructs corresponding to these flow ports. Table II shows our mapping and examples for flow ports.

Let's redesign the producer-consumer example using flow ports. The data items flow from the producer/the fifo to the fifo/the consumer through the connectors between the producer

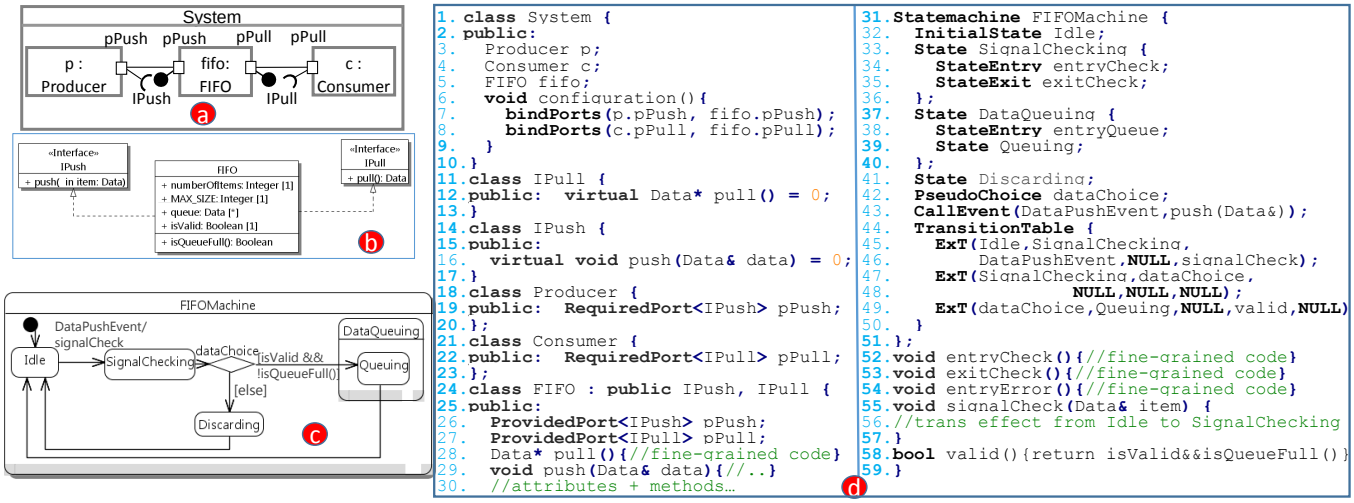

Fig. 2. Architecture model and generated extended code

and the fifo, and the fifo and the consumer, respectively. The changes made to the design in Fig. 2 (a) are as follows:

- *pPush* and *pPull* of the producer and the fifo become *OutFlowPort* ports, namely, *pOutData* in the respective class.
- *pPush* and *pPull* of the fifo and the consumer become *InFlowPort* ports, namely, *pInData* in the respective class.

Listing 1 shows the generated code for the flow port-based producer-consumer example. The constructs *OutFlowPort* and *InFlowPort* are used. From an implementation perspective, the producer sends data items to the fifo via the port *pOutData* by calling *pOutData.intf->push(item)* (lines 4-6). Similarly to the service ports, flow ports also have an interface attribute *intf* for writing code to send signals from a component to another component. In case the behavior of the receiving component (the fifo, for example) is described by a state machine, the *push* method will fire a signal event in order for the state machine to handle it. The details of signal events are discussed in sub-section III-A2.

Listing 1. Producer-consumer using flow ports

```
1  class Producer {
2  public:
3    OutFlowPort<Data> pOutData;
4    void sendToFifo(Data& item) {
5      pOutData.intf->push(item);
6    }
7  }
8  class FIFO : public IPush, IPull {
9  public:
10   InFlowPort<Data> pInData;
11   OutFlowPort<IPull> pOutData;
12 }
```

In the next sub-section, we present our proposed additional programming constructs corresponding to the modeling concepts for the architecture behavior, in particular UML state machine elements.

2) Behavioral constructs: In our approach, UML State Machines are used for modeling the behavior of components. Our behavioral programming constructs correspond to UML State Machine concepts at the modeling level. These behavioral constructs are grouped into three parts: topology, events, and transition table in the extended code.

Topology: A topology contains the constructs to describe the state machine hierarchy. The root of the topology is specified via the *StateMachine* as in Fig. 2. Other elements such as *region*, *state*, and *pseudo state* are hierarchically defined as state-machine (direct/indirect) sub-elements.

State actions such as *entry/exit/doActivity*s are declared within the corresponding state in the extended code as state attributes. These actions must be implemented as methods in the owning class and have no parameter. For example, *Idle* is an initial state. The *SignalChecking* state (lines 36-39) is declared with the state actions, *entryCheck* and *exitCheck*. The *FIFO* class implements the methods *entryCheck* and *exitCheck* (lines 60-61) for the state actions. Programmers can write fine-grained code for these action methods.

Concurrent states with orthogonal regions in the extended code are not shown here due to space limitation. Pseudo states can be declared within *Statemachine/states/regions* in the extended code, the syntax is similar to class attribute declarations. For example, line 49 in Fig. 2 declares the *dataChoice* choice pseudo state mapping to the corresponding pseudo state in the *FIFOMachine* model.

Events: In UML, four event types including *CallEvent*, *TimeEvent*, *SignalEvent*, and *ChangeEvent* are defined for modeling reactive systems using UML State Machines. The semantics of these events are clearly defined in the UML specification and briefly described in the below list.

- A call event is associated with an operation/method and emitted if the operation is invoked.
- A signal event is associated with a UML signal type containing data. When a component, whose behavior is described by a UML State Machine, receives a message/signal through its in/inout flow ports, a signal event is automatically emitted and stored in an event queue to be processed by the state machine later.
- A time event specifies the time of occurrence relative to a starting time, defined as the time when a state with an outgoing transition triggered by the time event is entered. The time event is emitted if the state remains active longer than the time of occurrence to trigger the transition.
- A change event has a boolean expression and is fired if the expression's value changes from false to true.

The processing of call events is synchronous meaning that it runs within the thread of the operation caller. The processing of other events is asynchronous meaning that these events received by a component are stored in an event queue, which is maintained by the component for later processing. We support all of these events with the same semantics.

Transition table: It describes the mapping of our syntactical constructs to UML transitions at the model level. Three kinds of UML transitions, *external*, *local*, and *internal*, are supported but this paper only presents external transitions. The difference between these kinds is clearly stated in UML and beyond the scope of this paper. Lines 45-46 in Fig. 2 shows an external transition, which is from *Idle* ti *SignalChecking*, triggered by the *DataPushEvent* call event declared with the state machine, and has *signalCheck* as transition effect. The *signalCheck* method is implemented within the *FIFO* class owning the state machine, and has the same formal parameters as the *push* method associated with the call event.

For example, line 50 shows a call event, which is emitted whenever there is an invocation of the *push* method of the *FIFO* class. The processing of the emitted event activates the transition from *Idle* to *SignalChecking*, and executes the *signalChecking* transition effect method. The data item returned by the invocation will be checked for validity and further put to the queue or discarded.

Deferred event: A state can declare *deferred events* by introducing attributes typed by our class-like additional construct *DeferredEvent* and named as event names to be deferred. A deferred event will not be processed while the state remains active. The deference of events is used to postpone the processing of some low-priority events while the state machine is in a certain state.

B. In-place text-to-text transformation

To remind, the previously generated extended code contains our template-based and syntactic additional programming constructs and user fine-grained code. This extended code itself is not compilable, executable, and debug-able. That is why an in-place text-to-text (T2T) transformation is needed.

For each component class containing our additional constructs such as ports, bindings, and state machine elements in the extended code, the T2T transformation creates an additional source code file/class, namely *delegatee*, complementing with the generated extended code. The created delegatee classes/files are in the same repository with the extended code but hidden from developer perspectives and not intended to be modified by the developers. These delegatee files play as role of "dynamic library code files" executing the runtime semantics and logic specified by the additional constructs.

Fig. 3 shows the relationships and control flows between a component in the extended code and its corresponding generated delegatee code as followings:

- A component class in the extended code owns a composite attribute typed by its delegatee class similarly to the delegation design pattern [19].
- The execution of the component is passed off to the delegatee class through the owned composite attribute.

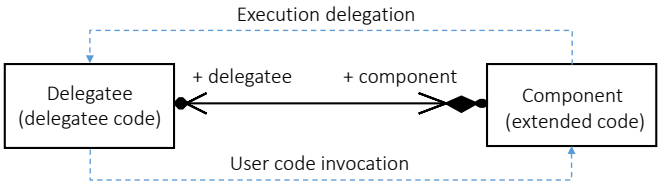

Fig. 3. Delegation from a component to its delegatee

- The user code (e.g., state entry/exit actions or transition effects) in the component is invoked by the delegatee execution.

The process of generating delegatee code from the additional constructs is actually a code generation from component-based design and state machines. This code generation is supported by several tools such as IBM Rhapsody [11] and Enterprise Architect [20]. However, the code generated from these tools is mixed with fine-grained behavior code such as state action or transition effect code, which makes the reconstruction of state machines and component-based elements from code impossible. Our code generation, on the other hand, completely separates the delegatee code and user code in different files.

Port transformation: Each port is transformed into code elements within the corresponding delegatee as followings:

- *ProvidedPort*: A getter for getting the implementation of the port's interface.
- *RequiredPort*: A setter for setting the *requiredIntf* required interface attribute of the port.
- *BidirectionalPort*: A setter and a getter equivalently to a required port and a provided port.
- *InFlowPort*: A getter for getting the implementation of the interface containing a *push* method for other components to send signals.
- *OutFlowPort*: A setter for setting the *intf* interface attribute of the port.
- *InOutFlowPort*: A getter and a setter equivalently to an in-flow port and an out-flow port.

Listing 2 shows code segments for the delegatee classes associated with the producer and *FIFO*. A *set_pPush* method (lines 4-6) and two getter methods (*get_pPush* and *get_pPull* at lines 11-12) are generated for the required port of the producer and the two provided ports of the FIFO, respectively.

It is worth noting that the bodies of *get_pPush* and *get_pPull* are different because of the *DataPushEvent* call event associated with the *push* method (see State machine transformation for more details).

Binding transformation: For each component with a configuration, a *createConnections* method is created within the associated delegatee class. Each binding in the configuration is transformed into a statement in *createConnections()*. The statement binds the interface attribute of a required port to a concrete implementation of the interface provided by the component at the other end of the connector. Listing 2 shows *createConnections* at lines 27-32 generated for the configuration of *System*. The binding between the *pPush* producer required port and the *pPush* fifo provided port is transformed into a statement calling the corresponding getter and setter to assign the interface attribute of the required port to the corresponding implementation.

State machine transformation: The transformation creates code within the delegatee class for executing the runtime logics

of the state machine in the extended code. The details of the pattern used in this transformation is presented in our previous work [21]. Our transformation supports all UML State Machine concepts including transitions, states, pseudo states, and events. This paper does not repeat this transformation but describes an example how the delegation of the state machine execution from the extended code to the delegatee works.

Listing 2. Delegatee classes of the producer and FIFO

```
1  class ProducerDelegatee {
2  public:
3    Producer* component;
4    void set_pPush(IPush* impl) {
5      component->pPush.requiredIntf=impl;
6    }
7  }
8  class FIFODelegatee: public IPush {
9  public:
10   FIFO* component;
11   IPush* get_pPush() {return this;}
12   IPull* get_pPull() {return component;}
13   void processDataPushEvent(Data& sig) {
14    if (active_state==IDLE){//check the Idle state active
15     component->signalCheck(sig);//transition effect
16     active_state=SIGNALCHECKING;//set active state
17    }
18   }
19   void push(Data& data) {
20     processDataPushEvent(data);
21     component->push(data);
22   }
23 }
24 class SystemDelegatee{
25 public:
26   System* component;
27   void createConnections(){
28     component->p.delegatee.set_pPush(
29       component->fifo.delegate.get_pPush());
30     component->p.delegatee.set_pPull(
31       component->fifo.delegate.get_pPull());
32   }
33 }
```

In Listing 2, the code at lines 13-22 shows how a *DataPushEvent* call event is emitted and processed. In the FIFO state machine in Fig. 2, an instance of the *DataPushEvent* call event, which is associated with the *push* method of *FIFO*, is emitted if *push* is called. Therefore, whenever there is a call to *push* through the *pPush* port of the producer, the call event should be emitted and processed. Hence, in the getter of *FIFODelegatee* for the *pPush* provided port of *FIFO*, instead of returning the fifo as *get_pPull*, *get_pPush* returns the delegatee as *this*. The call to *push* from the producer is then delegated to *push* of *FIFODelegatee* instead of that of *FIFO*. An event is then emitted and processes synchronously by the *processDataPushEvent* method. The latter checks whether the current active state is *Idle* (line 14). If so, the *signalCheck* transition effect in the extended code component (lines 55-57 in Fig. 2) is called and followed by changing the current active sub-state to *SignalChecking* (line 16).

By this transformation, on one hand, the programmers can execute and debug the extended code and modify the architecture at the code level while keeping the mapping between the model and the code bidirectional. On the other hand, the model can be reconstructed from the extended code by utilizing the proposed bidirectional mapping if there are modifications in the extended code, either to the architecture or the fine-grained behavior.

In the next section, we will show how the modifications in the model and the extended code can be synchronized by our synchronization mechanism.

IV. Synchronization Mechanism

In our previous work [22], a model-code synchronization mechanism in case of concurrent modifications is proposed. The mechanism 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 proposed bidirectional mapping between the architecture model and code. The incremental code generation (ICG) and incremental reverse engineering (IRE) needs a classification and management of modifications made in the model and code.

Incremental code generation: Table III shows our management of actions for propagating modifications in model to 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 detected as a combination of a removal followed by an addition.

Incremental reverse engineering: Our IRE, with specificity in text, is similar to change-driven transformation (CDT) [23]. The latter listens to changes made in a model and uses predefined rules to propagate the changes back to another model. However, CDT cannot be applied directly to propagate changes in code back to model because detection of changes in code is non-trivial. The approach in [24] records every developer operation by creating a specific code editor. However, this approach is not reliable because it is hard to monitor all of developer modifications if the developer uses different editors to modify the code.

In our approach, we do not record all of modifications made to code elements. We use a *File Tracker* to detect which files are changed by developers. This kind of tracking is much easier to realize and more realistic than the above approaches. It is also supported by several tools such as Git. The details of our approach are shown in Fig. 4.

The file tracker monitors all of the code files generated from the architecture model. The tracker does not listen to specific changes to programming language elements such as renaming an attribute or adding a method. After all of modifications made in the code, the tracker returns a list of files which were changed during modification. We do not allow renaming or deleting a class because doing these modifications at the code level requires doing some additional re-factorings. For example, deleting a class requires re-typing class attributes typed by this deleted class. We believe that working at the model level is more suitable to these modifications because the re-factorings can be done through code re-generation from the modified model. Furthermore, the IRE then needs to propagate

TABLE III
MODEL CHANGE CLASSIFICATION AND MANAGEMENT

Element type		Modification type	Action
Structure	Part/Port/Connector	Add/Remove/Update	Regenerate and re-transform code for the component containing the modified model element.
	Class/Component/Interface	Add/Remove/Update	Create/Remove/Update the corresponding code file(s). If the modification is remove or rename (update), regenerate the classes/components/interfaces that depend on the removed/renamed element. For example, regenerate the classes containing attributes typed by the deleted types to avoid unknown type problems during compilation.
	Property	Add/Remove/Update	Regenerate the corresponding class
Behavior	Operation	Add/Remove/Update	Regenerate the corresponding class
	UML State Machine	Add/Remove/Update	Regenerate and transform code for the component containing the state machine

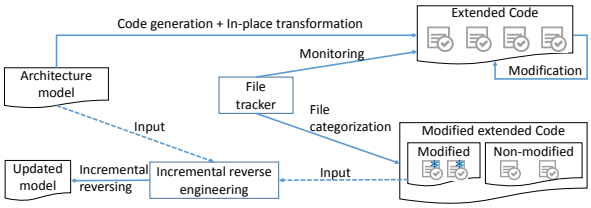

Fig. 4. Incremental reverse engineering with file tracker

modifications in the code within the class scope (attributes, methods, state machines) back to the model.

The modified files and the model are then used as input for reverse engineering. For each modified file, the IRE for each code element in the file follows a **Update-Create-Delete** strategy as followings:

- Update: Finding a model element matching with the code element by using the name and type of the code element to look for the corresponding model element. If exists, using the information of the code element to update the model element. For example, if we modify the *entryCheck* method body in Fig. 2 (d), the IRE will automatically propagate the changed body to the architecture model as a block of text.
- Create: If do not find a matching model element, creating a model element corresponding to the code element. For example, if a programmer adds a state to the state machine example in Fig. 2 (d), a UML state will be created in the model.
- Delete: UML elements (attributes, ports, connectors, methods, state machines, and events), which are not touched during the IRE, are deleted because these elements are implicitly removed by programmers during code modification.

It is worth noting that a renaming in code will be considered as an addition followed by a deletion at the model level.

In the next section, we describe our experiments to evaluate our approach.

V. EVALUATION RESULTS

The approach is integrated into Papyrus Designer - an extension of Papyrus [25]. Papyrus Designer features component-based design in UML and full C++ code generation through embedding of fine-grained code as blocks of texts. Formerly, it does not support model-code synchronization. This section presents our evaluation results through experimentation.

The objective of experimentation is to evaluate the synchronization correctness of the synchronization of architecture model and code through a case study. Three cases for the synchronization correctness are to be answered: (1) can the extended code be used to reconstruct the original architecture model? (2) if the extended code is modified, can the code modifications be propagated back to the model? and (3) if both extended code and model are concurrently modified, can our approach make the model and code consistent again?

We use our approach to develop an embedded software case study for LEGO. The LEGO car factory consists of small LEGO cars used for simulating a real industrial process [26]. It is chosen for the evaluation because it is a real world embedded system with enough complexity.

Previously, it was developed by using Papyrus Designer tool, which . However, the developer felt difficult, unfamiliar, and annoyed to write fine-grained code within a limited textual editor supported by the tool. The developer then refused to use that editor and programmed in CDT instead to be familiar and effective with programming facilities such as syntax highlights and auto-completion. She then copied the code from CDT back to the model and regenerated the code. The developer felt inefficient, prone-to-error, and lacked comprehension of the architecture information during code writing and copying. Furthermore, programming activities such as creation of methods are much easier in the code than in the model. Because of these reasons, the LEGO car factory is suitable to the evaluation.

A LEGO car is composed of four modules: chassis, front, back, and roof.The communication between these modules is based on Bluetooth and master-slave like. In the latter, the chassis acts as the master while the other modules as the slaves. Each slave module consists of five components: bluetooth communication controller, convoyer, robotic arm, press, and shelf. The behavior of each component is described by a UML state machine. The components communicate with each other through a data flow like communication schema. Previously, each component holds references (pointers in C++), modeled as UML associations, to other components to exchange data between the components. Furthermore, API invocations between these components were also aided by these references. The latter, however, made the design not purely component-based and not reusable.

To adopt a fully component-based approach, we use flow ports to exchange data items/signals between the components within a module. API invocations are re-designed by using

service ports. Fig. 5 shows the UML composite structure diagram for the *front* module without showing detailed structures of each of its components. Furthermore, for simplification, only flow ports are shown in the figure. The names of the signals/messages exchanged between flow ports of the components are annotated above connectors between the flow ports. The three flow port types are used. For example, the controller can send the *StopProcess* signal to the other four components through its ports. The robotic arm component can send/receive *StopProcess* instances to the conveyor/from the controller through its bidirectional flow port respectively. Note that the processing of signals incoming to a component via its ports is realized by the component's state machine.

1) Reconstruction of model from code: In this first experimentation, for each (origin) module without fine-grained code embedded as blocks of text, we generated its corresponding extended code. The model used for code generation contains 97 classes, 111 connectors, 119 ports, 15 state machines with 240 vertexes, 296 transitions and 11 events including signal and time events. From the extended code, a reversed model was created by using the batch reverse engineering. This reversed model was then compared with the origin module model. No differences were detected for the four modules. This result assesses that our approach and its implementation can be used for reconstructing a model from its corresponding generated code.

2) Propagation of code modifications back to model: In this second experimentation, for each of the extended code generated as above, we added class attributes and fine-grained code to each component. Specifically, [how many?] state actions, transition effects, and class methods are created following the rules described in Section III. Furthermore, API invocations between components through UML associations and C++ references are re-factored with the use of service ports and invocations through the required interface of a service port.

After enrichment for the generated code, we automatically propagated the code modifications back to the corresponding module model by using IRE. We then manually checked the module model for updates as followings:

- UML properties are created in the model corresponding to the class attributes in the code.
- UML state actions and transition effects are created within the model. Each has a block of text containing the fine-grained code filled in the extended code.
- UML operations are created in the model corresponding to the class methods in the code.

All of the model elements corresponding in the modified elements in the code were found in the updated model. This result assesses that our approach and its implementation can propagate modifications in code back to model.

3) Synchronization of concurrent modifications: To assess our approach in case of concurrent modifications, we emulated a typical situation, in which the Lego architecture model, named the original model, and its generated code are concurrently modified. To simplify the emulation, we only introduced modifications to method bodies in the code (become the **modified code**) and additions of UML ports, connectors and states in the model (become the **modified model**).

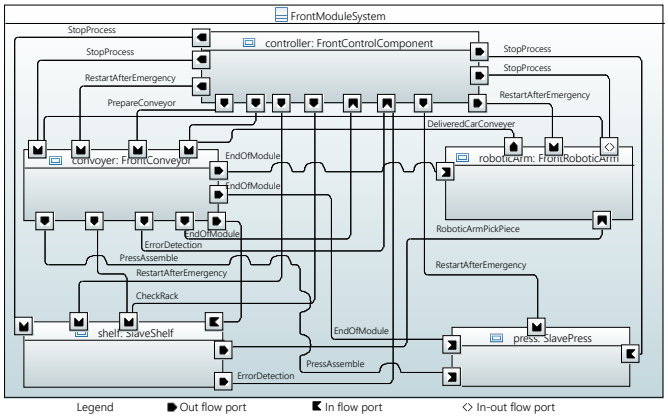

Fig. 5. Composite structure diagram of the front module for flow ports

TABLE IV
LEGO CAR FACTORY WITH AND WITHOUT SYNCHRONIZATION

Module	Lines of code		Binary size (KB)	
	Without sync	With sync	Without sync	With sync
Chassis	9659	**11193**	256	**264**
Front	9660	**12246**	248	**268**
Roof	9725	**XX**	248	**XX**
Back	9703	**XX**	248	**XX**

To synchronize the **modified model** and **modified code**, we used our previous synchronization mechanism [22], whose steps are as followings:

- Step 1: Through the IRE of the **modified code**, the **original model** is updated and becomes the **code-updated original model**.
- Step 2: Using EMF Compare to merge the model modifications from the **modified model** to the **code-updated original model**. The latter becomes the **synchronized model**.
- Step 3: Re-generate to create the **synchronized code** from the **synchronized model** by using batch code generation.

To assess the consistency of the **synchronized model** and **synchronized code**, we use batch reverse engineering to create a **reconstructed model**. We then compare the **reconstructed model** with the **synchronized model**. No differences were found during this comparison. We assess that our approach can synchronize architecture model and code in case of concurrent modifications.

After the experimentations of the synchronization, we compared the generated code in our approach with the code generated from Papyrus Designer without the use of the synchronization. Table IV shows the comparison of the codes generated with and without our synchronization approach. The number of lines of and the binary size compiled, by an ARM 32 bit Linux-based cross compiler, from code generated with our approach (With sync) are larger than those of the approach without synchronization. However, the difference between the binary sizes of the executables is very subtle: our approach produces around 3% larger (for the Chassis module), normalized to the binary size in case of the approach without synchronization. This assesses that the proposed approach does not only bring the ability to synchronize UML-based architecture models and code, but also keeps the generated

code quality acceptable.

After the case study-based evaluation of our approach, we examines how scalable the approach is and our perspectives in the next section.

VI. Open Discussion and Scalability

Our approach is about modeling, design, and implementation of software architecture, especially monolithic architectures. In this section we describe the differences between our approach and architecture description languages [27], which provide precise and formal notations for describing architectures. We also discuss how our approach can be applied to distributed architectures, to ALF, and the main limitation of our approach in applying to synchronize architecture models with different programming languages.

Architecture description language: The idea of our approach is to have architectural information right at the code level by proposing additional ad-hoc programming constructs.At a first glance, the code written with these additional constructs looks like the code in ADLs. However, our approach and the ADLs are fundamentally different. The ADLs are often used for architecture description, analysis, and verification [28]. The Smalltalk-based SCL component language [29] is even operational. However, its acceptance does not gain as much as mainstream object-oriented languages. Our additional constructs remain the extended language as an object-oriented programming language dedicated for system development including debugging and execution. The code written in ADLs can be translated into programming language code. But, the development activities such as compilation, debugging, and execution of this translated code is completely separated from the ADLs code. Our approach, on the other hand, allows the use of architectural information elements directly at the code level for these development activities. Furthermore, the additional constructs in our approach are often used by programmers while the ADLs by architecture analysts.

Distributed architecture: A question to our approach is whether it can be used for development of distributed architectures such as client-server architecture. Currently, it is only applied to a monolithic architecture. However, we believe that we can apply the approach to distributed architectures with some extensions. In our previous work [30], we propose to use *interaction components* to model the communication between components within a distributed system. For a client-server system model as in Fig. 6 (a) in which the client requests services from the server through a remote interface *IPush*. The connector between the components ports is transformed into an *ic* interaction component typed by *AsyncCall* as in Fig. 6 (b). The *AsyncCall* interaction component is modeled as UML components/classes and then translated to middleware-based interaction implementation such as ZeroMQ (see [30] for more details). In the generated extended code in Fig. 6, the system at the client-side contains the client, the *ic* interaction component, and a binding from the *p* required port of the client to the *q* provided port of *ic*. The system at the server-side, on the other hand, consists of *server*, *ic*, and a binding from the *q*

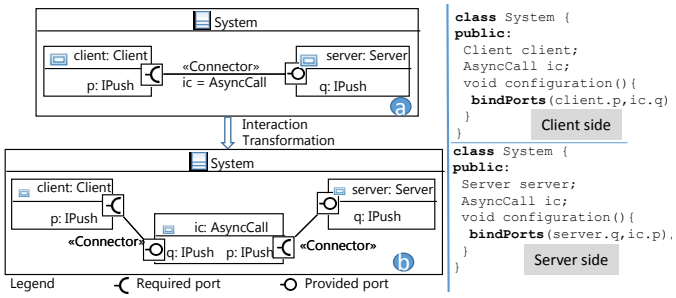

Fig. 6. Client-server example including interaction component transformation, client-side, and server-side code

provided port of *server* and the *p* required port of *ic*. *AsyncCall* realizes the communication implementation between the two sides. For example, for each call from the client through its port, *AsyncCall* establishes a socket connection, marshals and send the call's parameters to the *ic* interaction component at the server-side. This latter indeed demarshals the received data to extract the parameters values sent by the client, and calls the corresponding method of the server via the *p* port of *ic*.

The propagation of modifications in the model and the code is as followings:

- If the model is modified:
  - If the client is modified, propagate the modifications to the client-side code.
  - If the server is modified, propagate the modifications to the server-side code.
  - If the interaction component is modified, propagate the modifications to the interaction component code at both of the sides.
- If the code is modified:
  - If the client at the client-side is modified, propagate the modifications to the client component at the model level.
  - If the server at the server-side is modified, propagate the modifications to the server component at the model level.
  - If the interaction component is modified at the either side, propagate the modifications to the interaction component at the model level.

Action Language for Foundational UML (ALF): Back to the introduction section, we mentions the use of ALF as a fine-grained behavioral expression language for operation bodies to preserve modeling consistency and enable a full model-centric approach, and full-fledged code generation for embedded systems [31]. The main limitation that makes this approach not sound is that ALF is not common, even unknown in the programming community with very strong and widely used programming languages such as Java and C+. We believe that a harmonization of using ALF and common programming languages would benefit advantages of both of the modeling and programming community. Therefore, a further step, that builds upon our approach and synchronizes ALF-based fine-grained behaviors with programming code, should be included in future work. This extension might be based on the mapping between ALF and C++ proposed in [32].

Language support limitation: Our approach is based on using built-in programming language features to add new programming constructs to the language. Therefore, it is dependent on the programming language that we want to synchronize with UML. It is currently applicable to C++ with macros, Java and C# with annotations, and Python with macropy [33]. More

languages should be investigated to elaborate the approach.

VII. Related work

Our work is about synchronizing UML-based architecture model and code. Hence, the following tools and approaches are related to our work.

Reverse engineering and synchronization: Several tools such as Enterprise Architect [34] and IBM Rhapsody [11] support code generation from UML Class and State Machine models, and reverse engineering from code to UML classes. Systematic reviews of these tools are available in [35]. A few approaches [36], [37], [38] are able to recover components from object-oriented code, based on heuristic algorithms. Extracting state machines from sequential code is also supported in [39], [40], using static analysis of source code. However, the recovered models in these approaches are mainly used for system understanding, rather than development. The recovered models are often different from the original models.

Some techniques use specialized comments [12] such as @generated NOT to preserve code modified by programmers from generated code. EMF allows replacing the default generated code with user-code. However, this approach does not intend to synchronize model-code using a bidirectional mapping as ours. Furthermore, if accidental changes happen to the special comments, modified code cannot be preserved [41]. *xMapper* [41] overcomes limitations of the separation by separating generated and modified code in different classes. However, as the authors state that, it does not allow reverse engineering code elements back to architecture elements.

The three-way merging approach in [42] synchronizes 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.

Architecture Description Language (ADL): ADLs provide precise and formal notations, such as components, , connector and ports for system architecture description. Systems described by ADLs can be analyzed and validated for requirements from early development phase. The difference to our approach is clearly discussed in Section VI.

Change-driven propagation: 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.

Textual modeling languages (TMLs): TMLs [43] such as Umple [44] unifies modeling and programming text-based modeling languages. These languages can have bidirectional mapping to UML elements. The difference to our bidirectional mapping approach is that, because we extend a programming language, the extended code in our approach is valid to the programming language and can be processed by standard compilers such as GCC for C++ while the TMLs are not. In our approach, programmers can use favorite IDEs while the programmers are forced to change their working environment to efficiently use the TMLs. In [45], 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 on EMF and embeds all modeling concepts to textual editors while our approach only introduces necessary concepts in order to keep programmers being familiar with of their favorite programming language.

Language extension: The *P* [46] and *P#* [47] state machine-based programming languages are proposed by Microsoft for safe event-driven programming. *P#* adds only a subset of state machine constructs, state and transition, to C#. A system in these languages is composed of state machines communicating with each other through events. However, the semantics of state machines in these languages is not UML-compliant and component-based concepts are not supported by these languages. The idea of adding more constructs for object-oriented languages is similar to ArchJava [48]. ArchJava [48] adds structural concepts such as part and port to Java to support the co-evolution of architecture structure and Java implementation. However, ArchJava does not provide a mapping between of architecture behavior and code. Furthermore, ArchJava is not standard Java and then not executed with standard Java Virtual Machine, and facilities of IDEs such as auto-completion are not compatible.

VIII. Conclusion

We have presented an approach for synchronization of object-oriented programming code and architecture model specified in using UML Class, Composite Structure and State Machine concepts, notable components, ports, connectors, and state machine elements. The approach is based on extending an existing object-oriented programming language by introducing additional ad-hoc programming constructs for modeling concepts that have no representation in common programming languages such as Java and C++. The extended code written with the additional constructs is used as input of an in-place text-to-text transformation to generate other code files to be executable. The synchronization mechanism synchronizes the extended code with the model in case of concurrent modifications.

To evaluate the correctness, the practicality, and the feasibility of the synchronization, we used the approach to develop an embedded software case study - Lego Car factory. Our approach can synchronize concurrent modifications in model and code. Furthermore, the quality of generated code is also preserved in the synchronization approach, in compared to the approach without synchronization. We also discuss the applicability and scalability of our approach in applying it to distributed architectures, synchronization of ALF and programming languages for profiting advantages of both of the modeling and programming world.

The main limitation of our approach is the applicability to the synchronization of architecture model with different programming languages. For the moment, the approach is applicable to C++ and Java. In future, we will investigate other programming languages. The discussed perspectives will also be further detailed for elaborating our approach.

REFERENCES

[1] 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.

[2] R. Hilliard, "Using the uml for architectural description," in *International Conference on the Unified Modeling Language*. Springer, 1999, pp. 32–48.

[3] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144 – 161, 2014.

[4] R. Pandey, "Architectural description languages (adls) vs uml: a review," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 1–5, 2010.

[5] E. Posse, "Papyrusrt: Modelling and code generation (invited presentation)," in *Proceedings of the International Workshop on Open Source Software for Model Driven Engineering, Ottawa, Canada, September 29, 2015.*, 2015, pp. 54–63.

[6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 29–42.

[7] "Reactive Manifesto," http://www.reactivemanifesto.org/, [Online available 10/4/2017].

[8] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "Towards translational execution of action language for foundational uml," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*. IEEE, 2013, pp. 153–160.

[9] OMG, "Action Language for Foundational UML (Alf): Concrete Syntax for a UML Action Language," 2013, version 1.1, OMG Document Number: formal/2013-09-01.

[10] A. Charfi, C. Mraidha, S. Gérard, F. Terrier, and P. Boulet, "Toward optimized code generation through model-based optimization," in *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, 2010, pp. 1313–1316.

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

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

[13] 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.

[14] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 1st ed. Morgan & Claypool Publishers, 2012.

[15] L. De Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, 2012.

[16] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 3, pp. 213–249, Jul. 1997. [Online]. Available: http://doi.acm.org/10.1145/258077.258078

[17] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin, "Automatic model generation from documentation for java api functions," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 380–391.

[18] OMG, "A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems," 2011, version 1.1 formal/2011-06-02.

[19] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[20] SparxSystems, "Enterprise Architect," Sep. 2016. [Online]. Available: http://www.sparxsystems.eu/start/home/

[21] V. C. Pham, A. Radermacher, S. Gérard, and S. Li, "Complete code generation from uml state machine," in *5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017*, 2017.

[22] 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.

[23] I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 342–356.

[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. Gérard, C. Dumoulin, P. Tessier, and B. Selic, "Papyrus: A uml2 tool for domain-specific language modeling," in *Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems*, ser. MBEERTS'07. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 361–368.

[26] "Lego Car Factory," http://robotics.benedettelli.com/lego-car-factory/, [Online avalable: 22/3/2017].

[27] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[28] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum, "Integrating architecture description languages with a standard design method," in *Proceedings of the 20th international conference on Software engineering*. IEEE Computer Society, 1998, pp. 209–218.

[29] L. Fabresse, C. Dony, and M. Huchard, "Scl: a simple, uniform and operational language for component-oriented programming in smalltalk," in *International Smalltalk Conference*. Springer, 2006, pp. 91–110.

[30] V. C. Pham, Ö. Gürcan, and A. Radermacher, "Interaction Components Between Components based on a Middleware," in *1st International Workshop on Model-Driven Engineering for Component-based Software Systems (ModComp'14)*, 2014.

[31] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "On the generation of full-fledged code from uml profiles and alf for complex systems," in *Information Technology-New Generations (ITNG), 2015 12th International Conference on*. IEEE, 2015, pp. 81–88.

[32] F. Ciccozzi, "On the automated translational execution of the action language for foundational uml," *Software & Systems Modeling*, pp. 1–27, 2016.

[33] Macropy, "Macros in Python," https://github.com/lihaoyi/macropy, 2017, [Online; accessed 24-March-2017].

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

[35] D. Cutting and J. Noppen, "An Extensible Benchmark and Tooling for Comparing Reverse Engineering Approaches," *International Journal on Advances in Software*, vol. 8, no. 1, pp. 115–124, 2015.

[36] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit, "Extraction of component-based architecture from object-oriented systems," in *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*. IEEE, 2008, pp. 285–288.

[37] J.-M. Favre, "G/sup see: a generic software exploration environment," in *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*. IEEE, 2001, pp. 233–244.

[38] M. von Detten, M. C. Platenius, and S. Becker, "Reengineering component-based software systems with archimetrix," *Software & Systems Modeling*, vol. 13, no. 4, pp. 1239–1268, 2014.

[39] M. Abadi and Y. A. Feldman, "Automatic recovery of statecharts from procedural code," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 238–241.

[40] T. Sen and R. Mall, "Extracting finite state representation of java programs," *Softw. Syst. Model.*, vol. 15, no. 2, pp. 497–511, May 2016.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: safe asynchronous event-driven programming," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 321–332, 2013.

[47] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson, "Asynchronous programming, analysis and testing with state machines," *SIGPLAN Not.*, vol. 50, no. 6, pp. 154–164, Jun. 2015.

[48] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Software Engineering, 2002. ICSE 2002*. IEEE, 2002, pp. 187–197.