Bidirectional Mapping between Architecture and Code for Synchronization

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 models are efficient to design the behavior and structure of architectures. In Model Driven Engineering (MDE), code can be automatically generated from the models. Nevertheless, current UML tools only produce skeleton code which is then fine-tuned by programmers. The modifications in code, which may violate the architecture correctness must be synchronized to the model to make architecture and code consistent. However, current approaches cannot handle the synchronization when there is a significant abstraction gap between architecture and code. This paper proposes to ease synchronization between model and code, through a bidirectional mapping between code and architecture models specified by UML composite structure and state machine. The proposed mapping is used as a means to a synchronization methodological pattern proposed in our previous work, which allows concurrent modifications made in model and code, and keeps them synchronized. We plan to evaluate different aspects of the approach.**

I. Introduction

Unified Modeling Language (UML) has been widely used in Model-Driven Engineering (MDE) to describe architecture of complex systems [1]. Event-driven architecture is useful for designing embedded systems [2]. UML class, composite structure, and State Machine diagrams prove to well capture such architecture structure [3]. Approaches have been proposed in the context of Model-Driven Engineering (MDE) to automatically translate the architecture represented by the UML diagrams into implementation [3].

Current UML tools and approaches are not sufficient to exploit the fine-grained behavior of the architecture. These tools only produce skeleton code [4], which must then be tailored by programmers for fine-grained and algorithmic code. The modifications of generated code might violate the architecture correctness. When it happens, to keep the model and the code, the modifications in the code must be reflected back to the model. To deal with it, several approaches such as separation [5] and reverse engineering [6] use specialized comments to separate user-modified code and generated code. However, these approaches only work if the modifications are only introduced in a comments-separated area. One of the reasons makes the reflection of the modifications in the code back to the model hard is due to the lack of a bidirectional mapping between the architecture model specified by the aforementioned diagrams and code [7].

This paper describes an approach which enables the bidirectional mapping between architecture model and code. We argue that current programming language elements are at lower level of abstraction than software architectures. To establish a bidirectional mapping, our approach leverages the abstraction level of an object-oriented language by creating additional constructs for expressing architectural information. The established mapping is then combined with our synchronization methodological pattern presented in [8].

The remainder of this paper is organized as follows: Section II describes our mapping approach. Section III presents our evaluation plan. We discuss related work in Section IV. The conclusion and future work are presented in Section V.

II. Approach for bidirectional mapping

This section presents our bidirectional mapping approach.

A. Approach overview

Current programming language elements are at a lower level of abstraction than architecture elements [9]. To establish a bidirectional mapping, we therefore raise the abstraction level of a standard programming language by introducing additional programming constructs. We demonstrate the case, in which the programming language is C++.

Fig. 1 shows the overview of our approach. From a standard programming language, the additional constructs are created to form an **Extended Programming Language**, which is the working language for programmers. We establish a bidirectional mapping between the architecture model and the **Extended code**, which conforms to the extended language. The latter is semantically closer to the architecture since it adds language constructs for modeling concepts that have no direct representation in common object-oriented languages, notably ports, connectors and state-machine elements. At the same time, it is as close as possible to the existing standard language in order to minimize additional learning efforts.

The additional constructs are created by using specialized mechanisms of the standard language such as templates, and macros in C++ and annotations in Java. Therefore, the **Extended code** containing the additional constructs are syntactically valid to the standard programming language. By this way, the **Extended code** can seamlessly reuse legacy code written in the standard programming language and programming facilities such as syntax highlights and auto-completion in Integrated Development Environments (IDEs) can be fully used for assisting the development of the **Extended code**. The

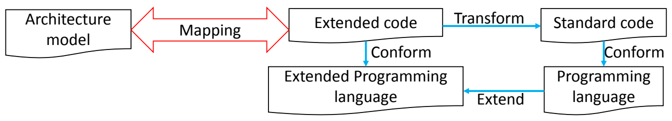


Fig. 1. Approach overview

TABLE I
MAPPING BETWEEN UML AND EXTENDED LANGUAGE

UML	Extended Language	Code example in Fig. 2
Port requiring an interface *I*	Attribute typed by *RequiredPort*<I>	Ports *pPush* and *pPull* at lines 21 and 25
Port providing an interface *I*	Attribute typed by *ProvidedPort*<I>	Ports *pPush* and *pPull* at lines 29-30
Connector	Binding	Lines 7-8
State Machine	*StateMachine*	The FIFO state machine at lines 34-59
State	*State/InitialState*	State *SignalChecking* at lines 36-39
Region	*Region*	Not shown
Pseudo state	Attribute typed by pseudo type	The *dataChoice* pseudo state at line 49
Action/Effect	Method	Methods at lines 60-65
Transitions	Transition table	Transition table at lines 51-58
Event	Event	The call event at line 50

Extended code is transformed to the **Standard code** and then the two code are combined with each other to execute.

In the next subsection, we present the additional constructs with an illustrative example.

B. Bidirectional mapping through an example

We present our additional constructs 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 for the consumer to pull it. The data items are saved in a sized queue attribute, *queue* associated with some class members such as the number of currently stored items (*numberOfItems*) and the *isQueueFull* operation for validating 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. *FIFO* also provides the *IPull* interface for the consumer to pull data items. FIFO implements the two interfaces in Fig. 2 b.

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

Table I shows some of the UML meta-classes and the equivalent constructs in the extended language. The constructs are categorized into *structural* (three upper rows) and *behavioral constructs* (seven lower rows).

1) Structural constructs:: We explain the constructs used for representing the architecture structure.

Port: Template-based constructs are proposed in the extended language to map with the architecture structure specified by UML ports. *RequiredPort<T>* and *ProvidedPort<T>* (see Table I) are equivalent to UML uni-directional ports, which have only one required or provided interface. The *T* template parameter is the 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 and one provided interface.

Lines 21 and 25 show ports with a required interface and lines 29-30 show ports with a provided interface of the *Producer*, *Consumer*, and *FIFO* classes.

Binding: A binding (see Table I, row 3) connects two ports equivalently to a UML connector connecting two UML ports. A binding is a method call to our predefined method *bindPorts*. Lines 7-8 shows two invocations to *bindPorts*, which takes as input two ports (the two ports of the producer and fifo, for example). Each class associated with a UML component contains a single configuration (as a method in lines 6-9) containing bindings.

Other model elements defined in the class diagram are mapped to the corresponding elements in the extended code. 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 the classes with pure virtual methods (lines 11-18) (in C++).

2) Behavioral constructs:: These constructs are used to map to UML State Machine concepts categorized into three parts: topology, events, and transition table in the extended code.

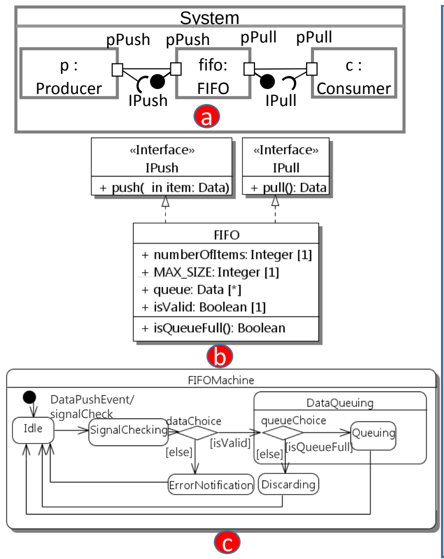
Topology: A topology contains the constructs mapping with UML vertexes and regions to describe the state machine hierarchy. The root of the topology is specified via the *StateMachine* as in Fig. 2. Other elements such as *region*, *state*, and *pseudo state* are hierarchically defined as its sub-elements.

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

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, which have syntax 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: Events are defined in UML are mapped to our constructs, which support four UML event types including *CallEvent*, *TimeEvent*, *SignalEvent*, and *ChangeEvent*. The semantics of these events are clearly defined in the UML specification and beyond the paper's scope.

Transition table: It describes our syntactical constructs mapping with UML transitions at the model level. Three kinds of UML transitions, *external*, *local*, and *internal*, are supported



```
1. class System {
2. public:
3.   Producer p;
4.   Consumer c;
5.   FIFO fifo;
6.   void 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.class Consumer {
24.public:
25.   RequiredPort<IPull> pPull;
26.};
27.class FIFO : public IPush, IPull {
28.public:
29.   ProvidedPort<IPush> pPush;
30.   ProvidedPort<IPull> pPull;
31.   Data* pull(){//fine-grained code}
32.   void push(Data& data){//..}
33.   //attributes + methods…
```

```
34.Statemachine FIFOMachine {
35.  InitialState Idle;
36.  State SignalChecking {
37.    StateEntry entryCheck;
38.    StateExit exitCheck;
39.  };
40.  State ErrorNotification {
41.    StateEntry entryError;
42.  };
43.  State DataQueuing {
44.    StateEntry entryQueue;
45.    PseudoChoice queueChoice;
46.    State Queuing;
47.  };
48.  State Discarding;
49.  PseudoChoice dataChoice;
50.  CallEvent(DataPushEvent,push(Data&));
51.  TransitionTable {
52.    ExT(Idle,SignalChecking,
53.        DataPushEvent,NULL,signalCheck);
54.    ExT(SignalChecking,dataChoice,
55.                NULL,NULL,NULL);
56.    ExT(dataChoice,queueChoice,
57.                NULL,Valid,NULL);
58.  }
59.};
60.void entryCheck(){//fine-grained code}
61.void exitCheck(){//fine-grained code}
62.void entryError(){//fine-grained code}
63.void signalCheck(Data& item) {
64.//trans effect from Idle to SignalChecking
65.}
66.}
```
d

Fig. 2. Architecture model and generated extended code

but this paper only presents external transitions.

For example, line 50 shows a call event, which is emitted whenever there is an invocation to the *push* method of the *FIFO* class. The processing of the emitted event fires the transition from *Idle* to *SignalChecking*, and executes the *signalCheckingEffect* method, which is the effect of the transition. 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.

C. Transformation

The transformation creates **Standard code** combined with the extended code to form executable code so that the programmers can compile, debug, and execute the extended code. At the same time, the latter is able to map back to the model.

To do it, for each class containing bindings or state machine elements, we generate a *controller* class in the standard code to control the state machine execution and create connections between parts through ports (see lines 15-19 and 5-10 in Listing 1). The written fine-grained code in the extended code will be called by the controlling (standard) code or the controller classes. Listing 1 shows the generated standard code. *SystemController* is associated with *System* by referring to the system object through a pointer *pSys*. It also declares a *controller* attribute typed by *FIFOController* associated with *FIFO*. During code execution, the standard code makes the required and provided interface of the *pPush* ports of *Producer* and *FIFO*, respectively, refer to the controller of *FIFO*. The *required* and *provided* attributes are actually declared within the port constructs as described in II-B in order for programmers to write fine-grained code. In this case, when a programmer want to call the *push* method provided by *FIFO*, she/he only needs to write *pPush.required->push(data)*, which will call the *push* method implemented by the FIFO controller. The *pPush* ports refer to *controller* because a call event associated with *push* is declared within *FIFOStateMachine*. In order to process the event emitted by calling the *push* method of *FIFO* through its *pPush* port, an appropriate method for event processing is called before the call to *push* of *FIFO* through a reference to *FIFO* (lines 20-23).

Listing 1. Generated controller classes

```
1  class SystemController {
2  public:
3    System* pSys; FIFOController controller;
4    void configuration(){
5    controller.pFifo=&pSys->fifo;
6      pSys->p.pPush.required=&controller;
7      pSys->fifo.pPush.provided=&controller;
8      pSys->c.pPull.required=pSys->fifo;
9      pSys->fifo.pPull.provided=pSys->fifo;
10   }
11 }
12 class FIFOController: public IPush {
13 public:
14   FIFO* pFifo;
15   void processDataPushEvent(Data& sig) {
16     //check the Idle state active
17     //call the transition effect pFifo->signalCkeck(sig)
18     //set the SignalChecking state active
19   }
20   void push(Data& data) {
21     processDataPushEvent(data);
22     pFifo->push(data);
23   }
24 }
```

The interfaces of the *pPull* ports of *FIFO* and *Consumer* are referred to the fifo because no event should be emitted and processed in case of an invocation to the *pull* method.

By this transformation, 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.

III. Preliminary Evaluation Results and Plan

The mapping is used in the implementation of our model-code synchronization tool [8] as an extension of Papyrus [10]. This section shows our preliminary results and plan to evaluate our approach in combination with our synchronization [8].

Correctness: We will evaluate the correctness of the synchronization of architecture model and code using our proposed mapping. Three cases are planned: (1) can the extended code and mapping be used to reconstruct the original architecture model? (2) if the extended code is modified, can the modifications made be propagated back to the model? and (3) if both extended code and model are concurrently modified, can the mapping be used for synchronization?

Semantic-conformance: The motivation is to assess the preservation of semantics of UML in the code. In [11], we show our experiments to test the runtime execution of the code against the precise semantics of UML State Machine standardized by OMG in [12] with a test suite consisting 66 test cases. 62 out of 66 cases passed leaving for future to fix failed test cases.

Standard code efficiency: We target resource-constrained embedded systems. Hence, event processing speed and memory usage are critical. We compare the efficiency of the generated code with the code generated by some UML tools and source code libraries in [11]. The results show that the code in our approach runs fast and requires little memory.

Feasibility and scalability: We plan to use the mapping with our synchronization approach to develop a case study, which is an embedded software for LEGO. The mapping and synchronization are feasible and scalable if the development is efficiently successful.

IV. Related work

Our work is related to following tools and approaches.

Reverse engineering tools: Several tools [6] create code from UML models using a bidirectional mapping between UML class concepts and object-oriented code. However, when it comes to UML composite structure and state machines, no tools support a bidirectional mapping and synchronization.

Separation: Specialized comments [5] such as `@generated NOT` are used to preserve code modified by programmers by separating the modified code from generated code. However, this approach does not intend to synchronize model-code using a bidirectional mapping as ours. Furthermore, if accidental changes happen to the comments, modified code cannot be preserved [13]. *xMapper* [13] overcomes limitations of the separation by separating generated and modified code in different classes. However, it does not allow to map code elements back to architecture elements.

Textual modeling (TMLs) and architecture description (ADLs) languages: TMLs such as Umple [14] and ADLs can have bidirectional mapping to UML elements. The difference is that the extended code in our approach is syntactically valid and compilable by standard compilers such as GCC for C++ while the TMLs and ADLs are not. In ours, programmers can use the familiar combination of a standard language and IDE [8] while the programmers are forced to change their working environment to efficiently use the TMLs.

Language extension: The Boost library [15] tries to build a bidirectional mapping between C++ and UML State Machine. However, Boost does not support different UML events and component-based concepts. ArchJava [16] adds structural concepts such as part and port to Java. However, ArchJava does not provide a mapping between of architecture behavior and code. ArchJava makes it not Java anymore and facilities of IDEs such as auto-completion are not aware.

V. Conclusion

We have presented an approach for establishing a bidirectional mapping between code and architecture model specified by UML class, composite structure, and state machine diagrams. The idea is to raise the abstraction level of an existing programming language. The aim of the approach is to use the mapping as input in our model-code synchronization methodological pattern presented in [8].

For the moment, the approach is implemented for UML and C++. In future, we will extensively evaluate the approach for different aspects: synchronization correctness, feasibility, and scalability.

References

[1] 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.

[2] 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.

[3] E. Posse, “Papyrusrt: Modelling and code generation.”

[4] 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.

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

[6] IBM, “Ibm Rhapsody.” [Online]. Available: http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/

[7] 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.

[8] 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.

[9] 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.

[10] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic, “19 papyrus: A uml2 tool for domain-specific language modeling,” in *Model-Based Engineering of Embedded Real-Time Systems*. Springer, 2010, pp. 361–368.

[11] 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.

[12] OMG, “Precise Semantics of UML State Machines (PSSM) Revised Submission,” 2016, [Revised Submission, ad/16-11-01].

[13] 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.

[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.

[15] boost, “Boost C++,” http://www.boost.org/, 2016, [Online; accessed 04-July-2016].

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