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 (model) are efficient to design the behavior and structure, respectively, of event-driven architecture. In Model Driven Engineering (MDE), code can be automatically generated from the models. Nevertheless, current MDE tools and approaches are not sufficient to describe and generate the fine-grained behavior. Many UML tools only produce skeleton code which is then fine-tuned by programmers. The modifications in code might violate the architecture correctness must be synchronized to the model to make architecture and code consistent. However, current approaches cannot handle the synchronization in this case because of the abstraction gap between architecture and code. This paper proposes a bidirectional mapping between code and architecture model specified by UML composite structure and State Machine diagrams. The proposed mapping is used as a means to a synchronization methodological pattern 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], [4]. 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], [4], [5].

On the one hand, current UML tools and approaches are not sufficient to describe fine-grained behavior of the architecture using models. These tools only produce skeleton code [6], which must then be tailored by programmers for fine-grained code. Furthermore, there is perception gap between diagram-based and textual languages. While programmers prefer to use the more familiar combination of a programming language and Integrated Development Environment (IDE), software architects, working at higher levels of abstraction, tend to favor the use of modeling languages for describing the system architecture [7]. The modifications of generated code might violate the architecture correctness, which is not easy to detect due to the lack of a bidirectional mapping between the architecture and code [8].

On the other hand, in continuous development, the architects might change the architecture for new functionalities or requirements while the programmers might still tailor the current architecture. This results that the architecture model and code are concurrently modified.

The modifications made in model and code muse be synchronized to make the architecture and code consistent. However, current UML tools and approaches cannot synchronize the modifications due to the lack of a bidirectional mapping between architecture model specified by the aforementioned diagrams and code.

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, XSeparation 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 [7] for synchronization.

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

II. Approach

This section presents our approach to establish a bidirectional mapping between architecture model and code.

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 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 an existing 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 **Intermediate code**, which conforms to the extended language. The **Intermediate code** is written similarly to the **Standard code**, which conforms the existing programming language, and contains the syntax surface for the additional constructs. By this way, the **Intermediate code** can seamlessly reuse

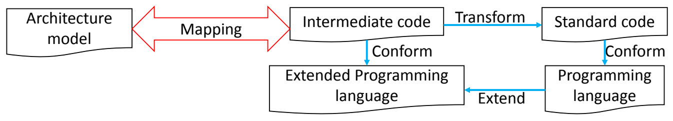


Fig. 1. Approach overview

legacy code written in the existing 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 **Intermediate code**. The **Intermediate code** is transformed to the **Standard code**, which is used for compilation.

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*. The latter is associated with the number of currently stored items (*numberOfItems*), the capacity (*MAX_SIZE*), and other attributes and operations used for validating incoming items and the availability of the queue. 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. The *DataQueuing* state verifies the availability of the queue to decide to either add the item to the queue or discard it.

Our additional constructs added to the existing programming language are categorized into structural and behavioral constructs.

1) Structural constructs:: Three constructs, *Part*, *Port*, and *binding*, are introduced to represent UML *part*, *port*, and *connector*, respectively, because these UML concepts do not have correspondences in the existing programming language. Fig. 2 d shows the intermediate code containing our additional constructs for the producer-consumer example.

In the intermediate code, the *Part*s and *Port*s constructs are template classes. The template parameters of the latter specify the types of *Part*s or required/provided interfaces of *Port*s. For example, lines 3-5 show the three parts corresponding to the ones defined in the model. Lines 21 and 25 show ports with required interfaces and lines 29-30 show ports with provided interfaces.

UML connectors for communicating UML parts through UML ports are mapped to method calls (bindings) in the intermediate code. Lines 7-8 shows two invocations to the *bindPorts* function, 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) for port bindings.

Other model elements defined in the class diagram are mapped to the corresponding elements in the intermediate code. For example, 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. The latter are divided into three sub-categories: vertex, event, and transition, which are specified into three parts: topology, events, and transition table in the intermediate code.

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

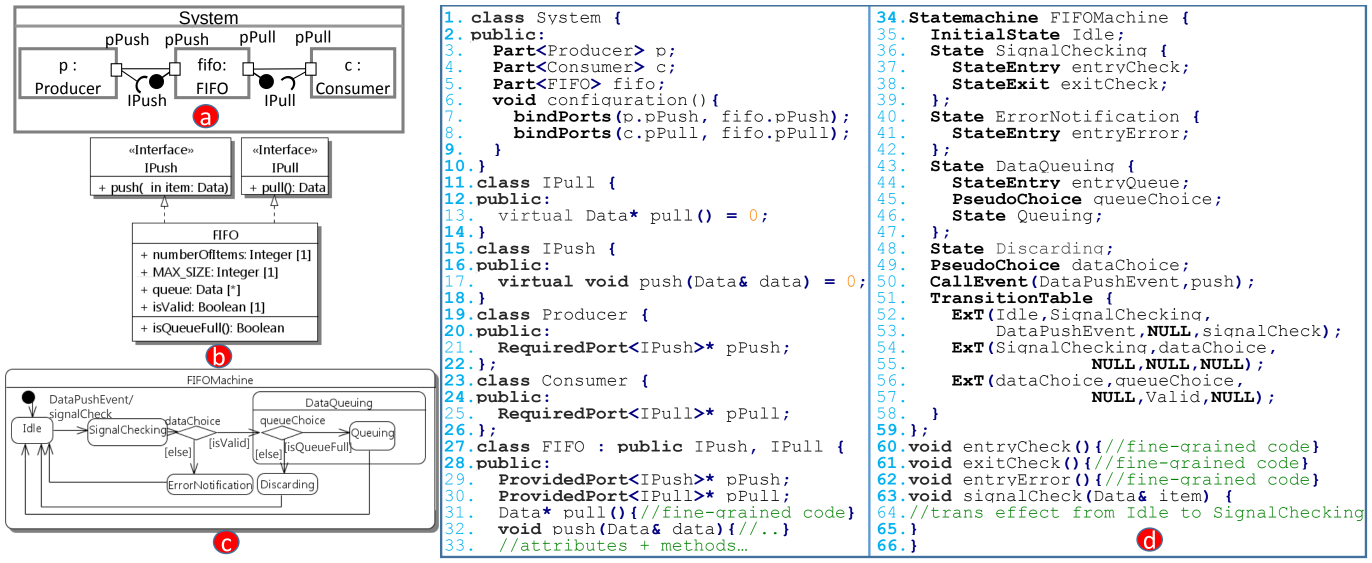
State actions such as state entry/exit/doActivity are declared as attributes of the state. These actions must be implemented in the owning class and has no parameter. For example, *Idle* is an initial state. The *SignalChecking* state (lines 36-39) is declared with state actions, *entryCheck* and *exitCheck*.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

Concurrent states with orthogonal regions in the intermediate code are not shown here due to space limitation. A State machine, state, or region in the intermediate code can declare pseudo states having similar syntax as attribute declarations of a class. The type of pseudo state attributes is one of *{PseudoEntryPoint, PseudoExitPoint, PseudoInitial, PseudJoin, PseudoFork, PseudoChoice, PseudoJunction, PseudoShallowHistory, PseudoDeepHistory, PseudoTerminate}*, which correspond to the pseudo states defined in UML State Machine.

Events: We support different constructs for 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.

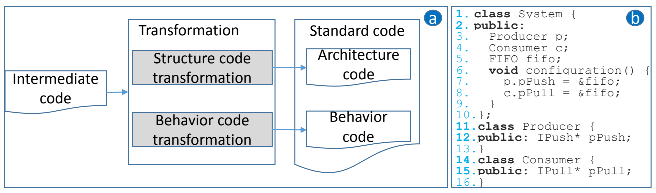
Transition table: It describes the transitions associated with UML transitions specified in the UM state machine at the model level. Three kinds of UML transitions, *external*, *local*, and *internal*, are supported but this paper only presents external transitions. The difference between these kinds is clearly stated in UML and beyond the scope of this paper.

For example, line 50 shows a call event, which is emitted whenever there is an invocation the *push* method if the FIFO class. The processing of the emitted event transitions from *Idle* to *SignalChecking* and executes the *signalCheckingEffect* transition method method associated with the transition if the current active state is *Idle*. If so, the data item brought by the invocation will be checked for validity and further put to the queue or discarded. Note that *signalCheckingEffect* has the same formal parameters with the *push* method.



```
1.  class System {
2.  public:
3.    Part<Producer> p;
4.    Part<Consumer> c;
5.    Part<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);
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. }
```

Fig. 2. Architecture model and generated intermediate code



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

Fig. 3. XSeparation compiler's architecture

Deferred event: A state can declare *deferred events* using our additional construct *DeferredEvent*. The deferred events are used for state machine execution to delay the processing of low-priority events when a certain state is active.

C. Transformation

The transformation transforms the intermediate code into the **Standard code**. Fig. 3 shows the transformation process. The latter consists of two grayed sub-transformations: **Structure code transformation** (SCT) and **Behavior code transformation** (BCT).

Structure code transformation: SCT transforms the additional structural constructs into standard code. Fig. 3 shows a segment of standard code generated from the producer-consumer example. Each port with required interface is transformed into a pointer attribute (lines 12 and 15) while each part into an object attribute (lines 3-5). A binding (bindPorts) in the configuration is transformed into an assignment which refers the pointer associated with a port with provided interface to the corresponding implementation. For example, the *pPush* pointer of *Producer* is referred to the fifo, which implements the *IPush* interface (line 7). When a method is called through a port, for example *push* through the *pPush* port, the corresponding method implemented in FIFO is invoked.

Behavior code transformation: It transforms the behavioral constructs for state machines into standard code similarly to code generation approaches from UML State Machines in MDE tools such as Rhapsody. Our goal is to support all of the features of state machines. However, the existing tools only support a subset of state machine concepts, e.g. Rhapsody does not support junctions, truly concurrent execution of orthogonal regions [10], and all of the UML event types.

We extend an existing code generation pattern with the use of IF/ELSE to support all of the state machine features. Due to space limitation, the details of pattern and evaluation for state machine runtime execution semantics are not presented in this paper.

III. Preliminary Evaluation Results and Plan

The mapping is used in the implementation of our model-code synchronization tool [7], which is an extension of the Papyrus modeling tool [11]. We show our preliminary results and plan to evaluate our approach in combination with our synchronization methodological pattern presented in [7].

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

Semantic-conformance: The motivation is to evaluate the preservation of semantics of UML in the standard code. The precise semantics of UML State Machine is defined by the Precise Semantics of UML State Machine (PSSM) [12] standardized by OMG with a test suite consisting 66 test cases. 62 out of 66 cases passed. There remains for future some special cases containing, for example, a transition from an entry point to an exit point, failing.

Standard code efficiency: We target event-driven embedded systems, which are resource-constrained. Hence, event processing performance and memory usage are critical. We compare the efficiency of the standard code with the standard code generated by UML tools such as Rhapsody and source code libraries. The results show that the standard 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. Comparison and Related work

The goal of proposing a bidirectional mapping is to synchronize model-code to preserve modifications made in the artifacts and keep them consistent. Hence our work is related to following tools and approaches.

Reverse engineering tools: Several tools [5], [13] support code generation from UML models and reverse engineering using a bidirectional mapping between UML class diagrams and object-oriented code. However, when it comes to UML composite structure and state machines, no tools support a bidirectional mapping and synchronization.

Separation: Separation [14] uses specialized comments such as @generated NOT to preserve code modified by programmers by separating the programmer-modified code from generated areas. 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 [15]. *xMapper* [15] overcomes limitations of the separation by separating generated and modified code in different programming constructs. However, it does not allow to map code elements back to architecture elements.

Text-based modeling languages (TMLs): TMLs such as Umple [16] and Earl Grey (EG) [17] can have bidirectional mapping to UML elements, then reduce the perception gap between diagram-based and textual languages. The difference to our approach is that we extend an existing programming language to connect to architecture while the TMLs are simply textual descriptions of UML models. To efficiently use the TMLs, programmers are forced to change their working environment while our approach does not impose that. Furthermore, the TMLs do not support full features of the aforementioned UML diagrams.

Language extension: BSML-mbeddr [18] is an state machine-based programming language integrated into the C language. It introduces the "big and small steps" semantics, which deviates the UML standard's. BSML-mbeddr only specifies state and region concepts. The Boost library [**?**] also tries to build a bidirectional mapping between C++ and UML State Machine. However, Boost does not support different UML events and component-based concepts. Our approach is inspired of ArchJava [19], which adds structural concepts such as part and port to Java. However, ArchJava does not provide a mapping between of architecture behavior and code. The communication between two ports uses method calls instead of interfaces as in UML and our approach. ArchJava makes it not Java anymore and facilities of IDEs such as auto-completion are not aware.Finally, BSML-mbeddr, Boost, and ArchJava are a code-centric approach whereas our mapping is to enable the model-code synchronization.

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 [7].

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

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

[6] 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.

[7] 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.

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

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

[11] 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.

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

[13] N. Magic, “Magic Draw,” https://www.nomagic.com/products/magicdraw.html, 2016, [Online; accessed 14-Mar-2016].

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

[15] 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.

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

[17] M. Mazanec and O. Macek, “On general-purpose textual modeling languages.” Citeseer, 2012.

[18] Z. Luo and J. M. Atlee, “BSML-mbeddr: Integrating semantically configurable state-machine models in a c programming environment,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2016. New York, NY, USA: ACM, 2016, pp. 105–117.

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