Enhancing Architecture-Implementation Conformance with Change Management and Support for Behavioral Mapping

Yongjie Zheng
Institute for Software Research
University of California, Irvine
Irvine, California, USA
zhengy@ics.uci.edu

Richard N. Taylor
Institute for Software Research
University of California, Irvine
Irvine, California, USA
taylor@ics.uci.edu

***Abstract*—It is essential for software architecture to be consistent with implementation during software development. Existing architecture-implementation mapping approaches are not sufficient due to a variety of reasons, including lack of support for change management and mapping of behavioral architecture specification. A new approach called 1.x-way architecture-implementation mapping is presented in this paper to address these issues. Its contribution includes deep separation of generated and non-generated code, an architecture change model, architecture-based code regeneration, and architecture change notification. The approach is implemented in ArchStudio 4, an Eclipse-based architecture development environment. To evaluate its utility, we refactored the code of ArchStudio, and replayed changes that had been made to ArchStudio in two research projects by redoing them with the developed tool.**

***Keywords*-software architecture; architecture change management; architecture-implementation mapping**

I. INTRODUCTION

Architecture-implementation mapping [1-4] is a process of converting software architecture to and from implementation with the goal of maintaining their conformance with respect to certain criteria. It directly determines the degree to which software architecture, a set of principal design decisions made about a software system [5], can be used in development to improve software productivity and quality. The mapping process, however, is often done manually by a team of software developers based on their understandings of the architecture. The limitations are obvious: low efficiency and error proneness. The difficulty comes from the fact that there is a conceptual gap between software architecture and implementation, and both artifacts are under constant changes during software development.

A number of architecture-implementation mapping approaches have been developed to automate this process. They either rely on after-the-fact consistency checking to detect the inconsistency (*correct-by-detection*), or apply technologies like code generation to avoid the inconsistency (*correct-by-construction*). Another perspective with which to look at existing approaches is assessing which artifacts can be manually changed during software development. From this angle, there are approaches of *one-way mapping* and *two-way mapping*. Table I presents the classified approaches. The italicized words represent instances of each approach.

TABLE I. ARCHITECTURE-IMPLEMENTATION MAPPING

	Correct-by-construction	**Correct-by-detection**
One-way mapping	1. Full code generation: *Domain-specific MDD*[6], *DSSA*[9] 2. Architecture refinement: *SADL*[10]	1. Reverse engineering: *Reflexion model*[1] 2. Runtime monitoring & verification: *Pattern-Lint*[7], *DiscoTect*[8], *ArchSync*[3]
Two-way mapping	1. Code generation & separation: *EMF*[11], *DiaSpec*[12] 2. Architecture framework: *Myx.fw*[13], *UniCon*[14] 3. Unifying descriptions: *ArchJava*[2], *Archface*[4]	

Correct-by-detection approaches detect the architecture-implementation inconsistency by extracting or inducing the architecture from the code and comparing the obtained architecture with the existing architecture. They essentially assume the relative constancy of software architecture. This is why there is no two-way mapping of correct-by-detection. Approaches in the category require the software be relatively complete or even executable so that reverse engineering or runtime verification can be done. For this reason, they are more appropriate for use in software maintenance rather than software development. In contrast, correct-by-construction aims to avoid inconsistencies from the very beginning. One-way mapping approaches in this category try to achieve the goal through complete code generation or step-wise architecture refinement, both of which face the challenge of bridging the abstraction gap. What they are trying to do is essentially promoting software development to the level of software architecture, so that source code editing can be completely avoided. However, this only works for some highly specialized domains or specific architecture styles.

What seems most promising is two-way mapping of correct-by-construction. Approaches in this category recognize the essential role of both architecture and code during software development, and provide certain forms of inconsistency-prevention mechanisms, such as separation of generated and non-generated code, and use of pre-defined architecture implementations. What makes these approaches not sound, however, is when architecture and implementation changes occur, or when the architecture contains information that

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012, Zurich, Switzerland

goes beyond structure, e.g. system dynamics. Specifically, the existing approaches in this category are often found deficient in the following aspects.

- Mapping code changes to architecture. This is essentially a hard problem of machine-based abstraction. It can be partially addressed by automatically generating code from the architecture and forbidding manual changes to generated code. With current code separation mechanisms (e.g. filling-in-blanks), however, this only works under the assumption that programmers are highly disciplined. Even so, accidental changes are still a possibility.
- Mapping architecture changes to code. Complete code regeneration with primitive merge support (e.g. EMF's *JMerge*) is usually used for this purpose. In the cases where user-defined code already exists and needs to be preserved during code regeneration, this method deteriorates quickly into a manual mapping. What makes the problem even worse is that programmers often have to figure out by themselves what was changed in the architecture.
- Support for the behavioral mapping. Software architecture encompasses both structural and behavioral decisions of the system under development. In contrast, most architecture-implementation mapping approaches are structure-oriented only. This is mainly because architecture behavioral specification (e.g. UML's sequence diagrams) is not complete enough to generate code from, and its corresponding code is inevitably mixed with user-defined dynamic details. Protection of architecture-prescribed code becomes extremely difficult in this situation.

Given all these challenges, we developed a new correct-by-construction approach of architecture-implementation mapping. The approach is called *1.x-way architecture-implementation mapping*. Different from previous work, it only allows manual changes to be initiated in the architecture ("1") and a separated portion of the code (".x"), with architecture-prescribed code updated solely through code generation. This is enabled by a new code separation approach, *deep separation*, where architecture-prescribed code and user-defined code of each architecture component are separated into two independent program elements (e.g. classes). In this way, mistaken changes of architecture-prescribed code are completely suppressed. User-defined code is also prevented from being overwritten during code regeneration. In particular, the behavioral architecture-implementation mapping can be supported with modeled system dynamics generated into a program construct that is not manually modifiable by programmers.

Additionally, an architecture change model is developed to explicitly record and analyze various architecture changes. Based on it, specific support is provided to map specific kinds of architecture changes to code. Most architecture changes can be automatically mapped to the code through an architecture-based code regeneration mechanism. For architecture changes that may require modifications to user-defined code, architecture change notifications are also generated and automatically sent across the separation boundary. All these features are applicable to both structural and behavioral architecture specifications.

The rest of the paper is organized as follows. Section II introduces the 1.x-way mapping approach, including its deep separation and change management mechanisms. Section III illustrates how behavioral mapping is supported in 1.x-way mapping. Section IV presents the implementation of the approach in ArchStudio. Section V describes the evaluation of 1.x-way mapping, including its current limitations. Section VI discusses the related work with the focus on representative two-way mapping approaches. Finally, Section VII concludes the paper.

II. 1.X-WAY ARCHITECTURE-IMPLEMENTATION MAPPING

Software architecture in 1.x-way mapping is modeled as a configuration of components with executions of significance (i.e. behaviors) defined by UML-like sequence diagrams and state diagrams. A component is a locus of computation and state in a system. It communicates with other components through explicitly defined interfaces, each of which contains a list of operations. The architecture-modeling notation used in 1.x-way mapping is *xADL 2.0* [15], an extensible, XML-based architecture description language. Java is used as the programming language in our current implementation.

Figure 1 shows an overview of 1.x-way architecture-implementation mapping. It is assumed that all the development activities shown in the figure take place in an integrated software development environment (IDE), where the tools used for creating and managing the system at different abstraction levels are able to communicate with each other and share information. A typical example of such an environment is the Eclipse platform.

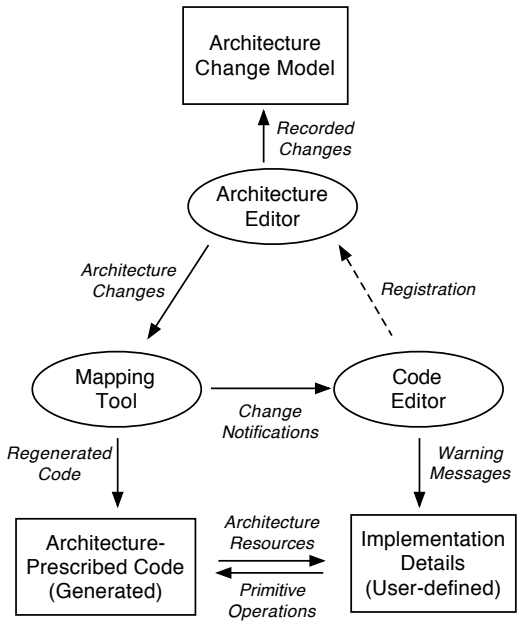

Figure 1. An overview of 1.x-way architecture-implementation mapping.

As shown in the figure, the implementation of each architecture component is separated into two independent program elements: *architecture-prescribed code* and *implementation details*. Architecture-prescribed code is automatically generated. It codifies all the externally visible information of a component that is specified in the architecture, including its identity, interfaces, and properties. User-defined code contains the internal implementation of a component that is to be manually developed by programmers. On top of them, three tools (represented by ovals in the figure) work closely in the IDE to map architecture changes to the code. *Architecture Editor* is responsible for the manipulation of architecture models. In particular, it maintains an explicit change model that records and classifies all the considered architecture changes. *Mapping Tool* is able to automatically map most of the changes to code without requiring manual work on the code, given that all the category information (e.g. *componentChanges*, *linkChanges*, etc.) is recorded in the change model. For those architecture changes that may require modifications to the user-defined code, change notifications are sent to *Code Editor*. In response, warning messages are prompted in the code to highlight changes that have to be made. To reduce the number of unnecessary notifications, a plug-in can be built to allow programmers to register for particular architecture changes. We are currently making this one of our future tasks, which is why it is represented by a dashed line in the figure.

Important technologies developed in 1.x-way mapping include: deep separation of generated and non-generated code, an architecture change model, architecture-based code regeneration, and architecture change notification. Each is introduced in the following subsections.

A. Deep Separation

A new code separation mechanism, *deep separation* or *linguistic separation*, is developed in 1.x-way mapping to decouple generated and non-generated code. It separates architecture-prescribed code (generated) and user-defined code (non-generated) of each component into two independent program elements (e.g. classes), and relies on program composition mechanisms to explicitly integrate separated code. Specifically, the user-defined code of a component provides a set of low-level operations, or *primitive operations*, from which high-level operations in the architecture-prescribed code are constructed. Meanwhile, available *architecture resources* (e.g. required interfaces, architecture properties) are passed to the user-defined code for use in the implementation of those low-level operations. This is essentially different from existing code separation approaches such as filling-in-blanks and subclassing. These approaches are called *shallow separation* or *spatial separation* in this research work, because their code is physically separated, but is still coupled and implicitly integrated by some inherent language relationship (same class, inheritance, etc.).

A calculator application is used as an example in this paper to illustrate how 1.x-way mapping works. Its structural architecture is shown in Figure 2. This is not a complex application. However, it provides concrete situations in which automatically maintaining the architecture-implementation conformance is difficult. Simply speaking, the calculator works as follows. The *GUI* component is responsible for collecting user's input of digits and operators, and displaying both intermediate and final results; the *Controller* component accepts calculation requests from *GUI*, and either pushes entered digits and operators to the corresponding stack or sends them to *Math Unit* for calculation, depending on which state it is in and what the input value is. The *Register* component saves the intermediate result that is to be displayed. Whenever the value in *Register* is changed, *GUI* is notified and has its display field updated correspondingly.

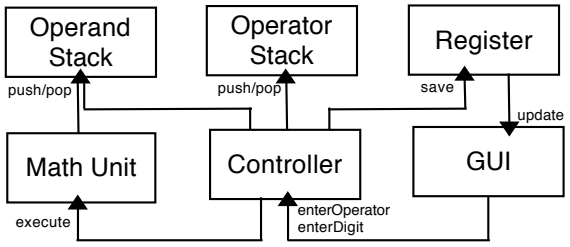

Figure 2. Structural architecture of the calculator application.

The code below shows the implementation of the *Controller* component with deep separation enforced. Other components in the figure can be implemented in the same way. Two classes (*ControllerArch, ControllerImp*) and one interface (*IControllerImp*) are created for the component. *ControllerArch* is architecture-prescribed code that is automatically generated. The interface (*IController*) that it implements (thus, operations that are included: *enterOperator* and *enterDigit*) and the references to connected components (line 3-5) are all from definitions in the architecture. Significantly, the operations in *ControllerArch* are implemented by simply passing requests to the user-defined code *ControllerImp* (line 11 and 14). This goes through a reference (*_imp*, line 2) to *IControllerImp* that is explicitly maintained in *ControllerArch*. *IControllerImp* serves as a contract between *ControllerArch* and *ControllerImp*, and is also automatically generated. It contains a list of operations that *ControllerArch* expects *ControllerImp* to provide. One of them is *setArch* (line 18), which is essential to the implementation of all architecture components, while the other operations in the list are component specific. It is through *setArch* that *ControllerArch* passes itself as a reference to available architecture resources to *ControllerImp* (line 8). What it also implies is the existence of a variable (*_arch*, line 23) of the type *ControllerArch*, or architecture-prescribed code, in the corresponding implementation *ControllerImp*, which is to be manually developed by programmers.

```
01: class ControllerArch implements IController{
02:  IControllerImp _imp;
03:  IOperatorStk _operatorStk;
04:  IOperandStk _operandStk;
05:  … //References to other connected components
06:  public ControllerArch(){
07:   _imp = new ControllerImp();
08:   _imp.setArch(this);
09:  }
```

```
10:  public void enterOperator(String opcode){
11:   _imp.enterOperator(opcode);
12:  }
13:  public void enterDigit(String digit){
14:   _imp.enterDigit(digit);
15:  }
16: }
17: interface IControllerImp{
18:  public void setArch(ControllerArch arch);
19:  public void enterOperator(String opcode);
20:  public void enterDigit(String digit);
21: }
22: class ControllerImp implements IControllerImp{
23:  ControllerArch _arch;
24:  public void setArch (ControllerArch arch){
25:   _arch = arch;
26:  }
27:  public void enterOperator(String opcode){…}
28:  public void enterDigit(String digit){…}
29: }
```

Compared with shallow separation, deep separation provides a better protection of generated code in the sense that programmers' modifications are completely precluded from the program element where generated code is located. Thus, any accidental changes of architecture-prescribed code are avoided. In addition, architecture-prescribed code and user-defined code are relatively independent. Should the *architecture* change and the code require regeneration later, only the architecture-prescribed code is overwritten. Another benefit of deep separation is that it opens up the opportunity of support for the behavioral architecture-implementation mapping, assuming the architecture behavioral specification can be automatically translated to code and there is a way to apply deep separation to the code. This is further discussed in Section III. Finally, chances still exist that the architect may need to manually edit the generated code due to the limitation of current modeling [16] and code generation [17] technologies. It is important to note that this does not affect the validity of the above design, because what is essential about deep separation is that architecture-prescribed code and user-defined code are separated and integrated in the specified way, and programmers' changes are limited to user-defined code (e.g. by a configuration management system).

B. Architecture Change Model

A significant challenge that 1.x-way mapping faces is how to map architecture changes to the code after the architecture is first implemented. This consists of two specific tasks: mapping changes to the architecture-prescribed code, and mapping across the separation boundary to the user-defined code. In particular, different architecture changes often have different impacts on the implementation of an architecture component. For example, redirecting a link between components supposedly does not affect component implementations, given that an architecture component is an independently deployable unit of composition [5]. In contrast, removing an interface from a component requires changes to both architecture-prescribed code and user-defined code of the component. Thus, architecture change management in 1.x-way mapping must be able to differentiate different kinds of architecture changes, and automatically map them to code in specific ways.

Figure 3 shows architecture changes and how they are managed differently in 1.x-way mapping. Considered changes include link changes, component changes, and behavior changes. Of these different changes, link changes are relatively easy to handle, simply by regenerating code that is responsible for bootstrapping the program and instantiating components with connection information. For the rest of this paper, we will mainly focus on the mapping of component changes and behavior changes to code. Notice that *Update Component* and *Update Behavior* both consist of a number of low-level operations, such as add interface to a component or remove a participant from a sequence diagram. They are not shown in the figure for brevity.

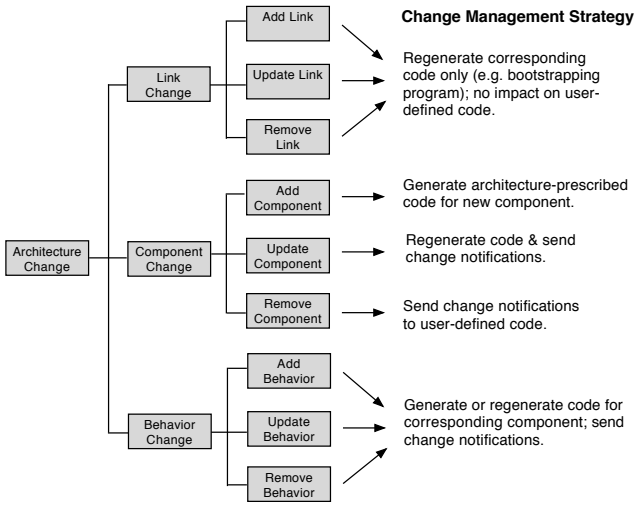

Figure 3. Architecture changes in 1.x-way mapping.

To support the mapping of architecture changes to the code, an architecture change model is maintained in 1.x-way mapping. The implementation of the model is specifically discussed in Section IV. Basically, all elements are monitored. All the considered changes made to the architecture are automatically recorded and classified. Special markers are placed in the change model after changes are successfully mapped to the code. The architecture change model plays an important role in 1.x-way mapping, primarily because it provides the information based on which architecture-based code regeneration and architecture change notifications are performed. Additionally, the change model makes architecture changes an independent artifact. This enables more advanced activities, such as change analysis and refinement.

C. Architecture-Based Code Regeneration

In principle, all architecture changes require the update of architecture-prescribed code. A dominant approach to do this is brute force regeneration. It completely regenerates code for the architecture regardless of what is changed. This approach suffers from a scalability issue [18], since even a small, localized change may require regenerating a disproportionately large part of the code. A recent code regeneration strategy is so called *incremental change*. Different from complete regeneration, it only regenerates code for the changed portion, such as an added interface or a removed

link. The problem is that it may break the system structure and consistency if the regenerated portion is in a highly cohesive entity, such as an architecture component. 1.x-way architecture-implementation mapping uses an *architecture-based code regeneration* mechanism that sits between complete regeneration and incremental change. It addresses the above problems by only regenerating code for modified components. While for each modified component, complete regeneration is enforced. With complete regeneration for each modified component, the integrity of component implementation is protected and structure or inconsistency issues are avoided. Meanwhile, the property of loose coupling between components makes incremental change at the component level a reasonable design to reduce the amount of code regeneration.

D. Architecture Change Notification

Different from architecture-prescribed code, it is not possible to automatically update user-defined code when architecture changes happen. What can be done at best is to send change-related information across the separation boundary to user-defined code. An architecture change notification contains information describing what element is changed in the architecture. The architect's comments when making those changes may also be included to give programmers more information, such as the rationale of a specific change. All the notifications are shown in the code editor in the form of warning messages. In particular, the user-defined code of a component only gets notifications for changes that are made to that component. This is to reduce unnecessary change notifications to decrease the manual analysis work needed to process the notifications. To further reduce the number of unnecessary notifications, a plug-in can be built to allow programmers to register for particular architecture changes. The registration can then be sent to and saved in the architecture as traceability information. When a registered change happens, notifications will be sent correspondingly.

III. Support For Behavioral Mapping

An essential task of architecture-implementation mapping is to have the modeled information correctly preserved in the code. For 1.x-way mapping to support the behavioral mapping, it is important that system behaviors be modeled in a form that is amenable to code generation. In particular, there must be a way to enforce deep separation to the corresponding code. The change management mechanisms of 1.x-way mapping can then be applied as presented earlier.

A. Architecture Behavioral Modeling

Architecture behavioral modeling in 1.x-way mapping is based on the general UML [19] state diagram and sequence diagram with some adaptations made. Our adapted state diagram captures the runtime behavior of a specific architecture component in terms of its state changes. Our sequence diagram defines a sequence of interaction calls between a component and its connected components in one of its provided operations. Furthermore, in our current implementation, both call sequences and state changes are linear. More advanced control structures such as iteration and branch could be added as an extension to the current work. In addition, all object-specific features are not supported in our diagrams since the modeled elements are components (which may e.g. be implemented as a set of classes) rather than objects in the object-oriented sense.

Figure 4 presents an example of two behavioral diagrams defined for the calculator application. The state diagram on the left models state changes of the *Controller* component. It starts from the state of *WaitForInput*, and switches between *WaitForNumber*, *WaitForOperator*, and *EnteringNumber* in response to invocations of its provided operations: *enterDigit*, *enterOperator*, and *enterMR* (i.e. memory recall). For some state transitions, an additional action is also specified (following "/" in the transition label) to be called before the target state is entered. The sequence diagram on the right depicts how the *MathUnit* component collaborates with the *OperandStack* component in its *execute* operation, assuming a binary operator is to be calculated.

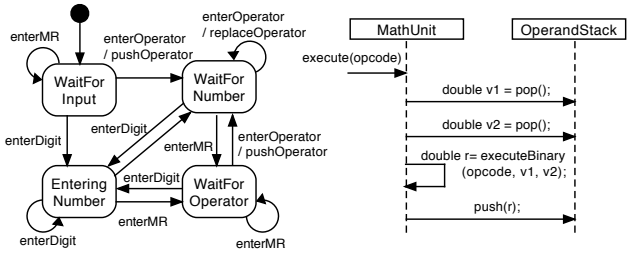

Figure 4. Example of a behavioral architecture definition.

Generating code from state diagrams is a common practice with many CASE tools [20]. In contrast, generating code from UML sequence diagrams is rarely supported. This is partially because some necessary information for code generation is missing in standard sequence diagrams, such as object assignment and how objects are passed between message calls [21]. In particular, the implementation of a sequence diagram is often found spread over the code (e.g. classes) of the participants that are involved. This makes the code generation process very difficult since the code generator not only needs to identify the right place to generate code, but also has to deal with conflicts caused by diagrams that have overlapping message calls.

In response, we added two additional restrictions to our adapted sequence diagram. First of all, we require that variable assignments and parameter passing must be explicitly represented for each message call. In Figure 4, this is how variables of *v1*, *v2*, *opcode*, and *r* are assigned and passed along. Second, all message calls in our sequence diagram must start from the component whose operation is defined by the diagram. In other words, only operations that are directly called in the specified operation are considered in the diagram. With these restrictions made, code can be easily generated from our sequence diagram. There are still situations where minor editing may be needed on generated code, for example, to deal with Java exceptions that cannot be captured in a sequence diagram. This is a limitation that is imposed by the current modeling technology, and is not inherent to the design of 1.x-way mapping.

B. Applying Deep Separation to Behavioral Code

Applying deep separation to the implementation of our adapted sequence diagram is relatively straightforward. The basic structure of architecture-prescribed code, user-defined code, and the contract interface between them are similar to what is presented in Section II. The only difference is how the operation specified by the sequence diagram is implemented in the architecture-prescribed code. Instead of passing the request to user-defined code, the implementation of the operation is now populated from what is defined in the diagram with each message call directly translated to a line of code. The code below is a portion of the generated architecture-prescribed code of *Math Unit* based on what is defined in Figure 4. The implementation of the *execute* operation (line 4-7) directly comes from the corresponding diagram. Note that a reference to the target component is prefixed to each interaction call (e.g. *_operandStk* before *pop*). This is a variable that was created during code generation, which is elaborated in Section IV.

```
01: class MathUnitArch implements IMathUnit{
02:  …//The basic structure is not changed.
03:  public void execute(String opcode){
04:   double v1= _operandStk.pop();
05:   double v2= _operandStk.pop();
06:   double r= executeBinary(opcode, v1, v2);
07:   _operandStk.push(r);
08:  }
09:  …//The implementation of other operations
10: }
```

Applying deep separation to the implementation of our state diagram is based on the state pattern [22]. Again, no changes are required on user-defined code and the contract interface compared with what is presented in Section II. A primary change on architecture-prescribed code is that multiple classes are generated for it, with a class created for each state. A class named *xxxArch* is also created as presented earlier. It is the place where all the structure information of the component is implemented, serving as a container that the state classes referenced. Figure 5 shows the basic structure of the architecture-prescribed code of *Controller*, whose state changes are defined in Figure 4.

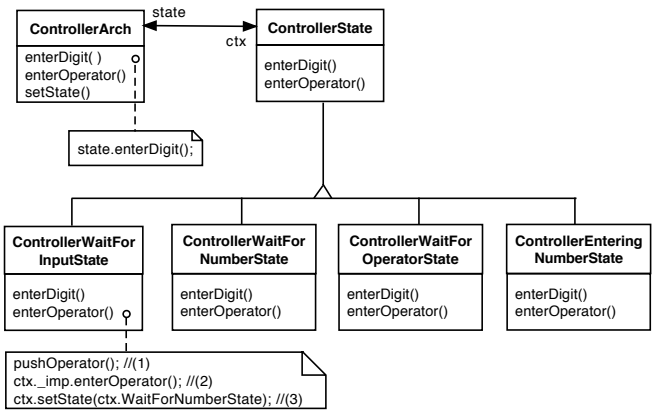

Figure 5. Architecture-prescribed code of the *Controller* component.

Specifically, three types of classes are generated. *ControllerArch* maintains a *state* variable that represents the current state of the component, and provides a *setState* method to change the current state. The operations in *ControllerArch* then redirect requests to the current state since they may be implemented differently in different states. Note that only a portion of *ControllerArch* is shown in the figure, while the rest of it (e.g. the reference to user-defined code) is same as what is defined in Section II. *ControllerState* is the abstract class that defines the behavior that a particular state of the component must have. *ControllerWaitForInputState* and the other three classes next to it are subclasses of *ControllerState*. Each implements a specific state. Operations in these subclasses are implemented by (1) calling the associated action if there is one defined in the state diagram; (2) calling the same operation in user-defined code where state-independent activities may be specified; (3) identifying the successor state in case a state transition is triggered.

IV. IMPLEMENTATION

1.x-way architecture-implementation mapping is implemented in ArchStudio 4 [23], a tool integration environment that is fully integrated within the Eclipse platform as a plug-in project. ArchStudio supports developing, visualizing, and analyzing architecture models using the xADL language. The implementation of 1.x-way mapping consists of four specific tasks: (1) recording architecture changes; (2) analyzing and refining changes; (3) building a code generator; (4) sending change notifications. Task (1) is implemented by adding recording logics to an existing ArchStudio tool. It is relatively independent of Task (2), (3), and (4), which together form the *Mapping Tool* shown in Figure 1. They communicate through production and consumption of the constructed architecture change model. The implementation of each task is presented in the following subsections.

A. Architecture Change Recording

The implementation of recording architecture changes is integrated into an ArchStudio visualization tool, *Archipelago*, which provides a graphical architecture-editing interface. In order to record changes made to an architecture, the xADL schema must be extended. Figure 6 shows an example of recorded architecture changes. As can be seen, a new element <*archChange*> is added to the root (<*xArch*>) of the architecture description. Under the new element, there are multiple <*changes*> elements (annotated with their starting time in the figure), each of which represents a change *session* that includes a series of specific changes as listed in Figure 3. The status of each change session is either "*mapped*" or "*unmapped*", depending on whether a map-to-code process (initiated by user through the selection of a menu item) is done on the session or not. A new "*unmapped*" session will be created automatically if the architecture is modified, and no "*unmapped*" change session exists. A tricky issue in architecture change recording is how to deal with removal of architecture elements. Recording the removal action itself is not a challenge, but the problem is that we often need the information of the removed element during the map-to-code process. With the current design of Archipelago, the element

will be removed permanently from the architecture once that removal action is triggered and saved. Our solution is to create a complete copy of the removed element in the architecture change model, so that Archipelago can remove the original element as before. The copy is used for information retrieval during code mapping, and is removed after the process is successfully done. Again, this special design reflects another advantage of architecture change model.

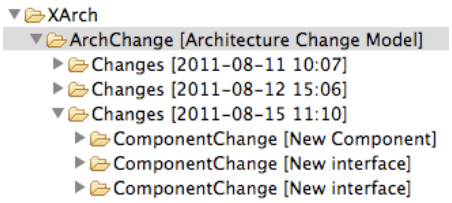

Figure 6. An example of architecture change model.

B. Change Analysis and Refinement

Architecture change model only records "*raw*" changes. In other words, they simply reflect what have been done to the architecture, and are relatively independent of how the changes are mapped to the code. In particular, some recorded changes may not be of interest to the map-to-code process at all. For example, consider the following scenario. The architect created a new component, worked on it a little bit, but somehow found this component not necessary and removed it. All these actions are recorded into the change model. However, as far as their impact on the code is concerned, nothing should be done during code mapping, assuming no other components made a reference to this component yet. This example highlights another important step in the implementation of 1.x-way mapping: analysis and refinement of recorded architecture changes.

As mentioned earlier in this section, the map-to-code process is performed per change session. Each item (e.g. *addComponent*) in the change session that is being processed will go through a specially designed filtering logic. The result of running this filtering logic on a change session are so-called "*refined*" changes that consist of a set of discrete change sets: *addedComponent* (components that were added in this change session), *updatedComponent* (components that were updated in this change session), and so on. In particular, the intersection of these change sets is guaranteed to be null. This makes sure that each changed architecture element be mapped to code in an unambiguous way.

The filtering logic codifies a set of rules that define under what condition a specific change item should be discarded or merged with a previous change item. For example, one filtering rule specifies that whenever a *removeXXX* (e.g. *removeComponent*) change item comes in, the set of *addedXXX* should be checked. If the entity to be removed exists, corresponding entry in the *addedXXX* set should be removed and the *removeXXX* change item is discarded; otherwise add the entity associated with *removeXXX* to the *removedXXX* set. In this way, all the changes in the scenario discussed above will be discarded in the end, and the code remains.

C. Code Generation

The JET technology [24] of Eclipse Modeling Project is used in the implementation of 1.x-way mapping to build a code generator. Figure 7 shows an overview of the code generation process. In general, a JET code generator only requires three inputs: complied *JET Template*, *XML input*, and *Configuration Variables*. An additional input, *Refined Architecture Changes*, is used here to enable architecture-based code regeneration, which ensures that it only regenerates code for modified components. *Architecture Model* and *Configuration Variables* provide required parameters for code generation. In particular, a configuration panel is developed in 1.x-way mapping for this purpose. It provides an opportunity for user to tune the code generation process by changing specific parameters, such as the name of generated classes. Note that the panel is preloaded with values retrieved from the architecture description. After code generation is done, user entered values will be written back to the architecture to update corresponding parameters. In this way, a simple record-and-replay is enabled, as is the creation of traceability links between the architecture and generated code.

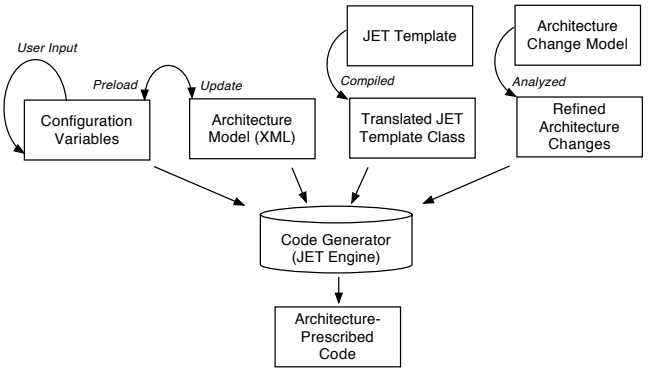

Figure 7. Code generation in 1.x-way mapping.

JET Template codifies specific code generation rules. It defines how a generated variable or class should be named and manipulated by default. Significantly, it enforces the deep separation mechanism in the generated code as described earlier in Sections II and III. Different strategies are taken to generate code for architecture elements that are structural only versus those that are linked to a behavioral diagram. Linking structural and behavioral architecture definitions is not a challenge in general based on the XML technology. It is somewhat difficult, however, to identify an operation that is defined by a sequence diagram from other operations of the same component, because they all exist as methods in a Java interface file. Our solution is to add a Javadoc *@see* tag to an operation in the Java file when a sequence diagram is defined for it, as exemplified below. The tag serves as a link to the corresponding diagram by which the code generator loads information and generates code.

```
01: /**
02:  * @see interactionffea0a1b-0c463028
03:  */
04: public void execute(String opcode);
```

D. Sending Change Notification

The architecture change notification of 1.x-way mapping is built upon the Eclipse *Markers* technology [25], which is used in Eclipse to annotate specific locations within a resource. Figure 8 shows an example of generated architecture change notifications and how they are displayed. Clicking any of these messages will lead user to corresponding code.

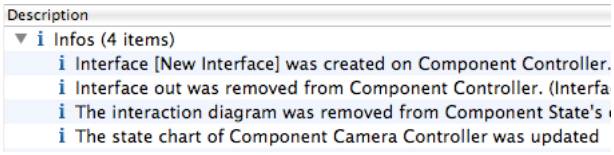

Figure 8. An example of architecture change notifications.

V. EVALUATION

1.x-way architecture-implementation mapping is evaluated from the following three aspects. First of all, we validate if deep separation as the basic requirement of 1.x-way mapping is applicable to the implementation of a real program of significant complexity. Next, we focus on the claimed features of 1.x-way mapping when it is used to manage changes of significant size that are made to the above program. In particular, we want to evaluate how frequent an architecture change can be mapped to code automatically, semi-automatically, or even manually, and how often an accidental change of architecture-prescribed code may happen given that it is supposed to be avoided with 1.x-way mapping. Finally, we evaluate how the involvement of behavioral mapping affects the statistics collected in the second step.

We applied the 1.x-way mapping approach to ArchStudio, a pre-existing architecture development environment where our 1.x-way mapping tool was built and integrated. ArchStudio has been used in several companies and dozens of universities. Its code was accompanied by an explicit xADL-based architecture model that consisted of 56 components and 42 connectors, corresponding to more than 85KSLOC. The architecture matched the code structure, but the code had been manually written. A primary benefit of exercising 1.x-way mapping with ArchStudio is that we could replay architecture and code changes that had been made to ArchStudio in a research project by redoing each change with our tool. The changes made were for a specific task, and happened before 1.x-way mapping was developed. Thus, biases were avoided. Notice that the involved architecture changes in this research project (and some other projects) were structure only. We then replayed changes that had been made to ArchStudio in the development of our own 1.x-way mapping tool, where behavioral models and changes were extensively involved.

A. Evaluation I: Applicability of Deep Separation

Before 1.x-way mapping was applied, the ArchStudio code had to be refactored based on the deep separation mechanism. This was also to answer the first question raised at the beginning of this section about the applicability of deep separation. A tool that can support this refactoring process is not yet available. What we did was generate the architecture-prescribed code for each component first (to a temporary file), used the generated code as a reference to identify corresponding code in the existing code, and finally decoupled them from implementation details using the deep separation mechanism. We found and resolved a number of problems during the refactoring process, and came to the following conclusions.

First, it is generally possible to apply deep separation to the code of a real program. We were able to enforce deep separation to the code of most of ArchStudio components. In particular, we found deep separation greatly facilitates the use of programming framework and code library, both of which are the norm rather than the exception in today's software development and extensively exist in the implementation of ArchStudio. The ArchStudio code is built upon an architecture framework called *myx.fw* that provides abstract base classes for components and connectors. It also restricts the way that certain architecture variables be initialized, and provides lifecycle methods for each component to override. Manually writing the framework-specific code is not only tedious, but also error prone. In the evaluation, we made a special template that included the routine code and had them automatically generated as part of the architecture-prescribed code. Both software productivity and the usability of the programming framework were improved. In addition, the ArchStudio code is greatly dependent on some application-neutral code libraries. For example, one such library provides APIs for accessing and manipulating the xADL document where architecture specifications are saved. Mixing architecture-prescribed code and the library code impedes the evolution of both parts. With deep separation enforced, the use of system library can be encapsulated in user-defined code. In some cases, we were even able to simply use a class from a code library as user-defined code since all required operations were already provided by the class.

Second, a configurable code generator is necessary to facilitate the application of deep separation. It provides a chance for the architect to address variations from predefined rules during code generation, so that the generated architecture-prescribed code does not have to be manually modified. One such variation happens when architecture-prescribed code of a component needs to declare the implementation of additional interfaces other than those that are specified in the architecture. This is particularly the case if the component contributes views, editors, and other GUI elements. In ArchStudio, a special implementation strategy was taken to address an architecture mismatch problem between its *myx.fw* framework and Eclipse plug-in mechanism in terms of who controls loading and instantiating the ArchStudio GUI [26]. Consequently, extensive interface implementation and class extension were involved. Another case where a configurable code generator is necessary is when user-defined code of a component needs to be initialized in a special way. In our evaluation, this happened when the implementation of a component used a special technology, e.g. Java dynamic proxy class, but the code generator was not sophisticated enough to express it.

Finally, deep separation may not be applicable in some special situations. There were components in ArchStudio whose implementation was spread over two Eclipse plug-in projects. To keep the changes minimal, we initially made the architecture-prescribed code and user-defined code located in the two projects respectively. This gave us a "circle detected in build path" error. We then found that it was caused by an inherent requirement of deep separation: architecture-prescribed code and user-defined code of each component must be accessible to each other. In the context of Eclipse plug-ins, what this meant was that the two plug-in projects mentioned above had to add each other to their dependency list. Unfortunately, it was not allowed by the Eclipse plug-in mechanism. This is a situation that fails deep separation. However, we believe its impact is of a limited range considering that (1) this failure is Eclipse specific; (2) it is in general not a good practice to have the implementation of a component spread over projects. In particular, the problem can be easily solved by either moving the code into one project or adding a code agent that acts as user-defined code in the project where the architecture-prescribed code is located.

B. Evaluation II: Change Management

After the ArchStudio code was refactored for deep separation, we focused on a research project called *ACTS* where both architecture and code changes were made to ArchStudio to build a traceability tool. As mentioned earlier, all the involved architecture changes were structure only. 11 components were added to the ArchStudio architecture, and around 15KSLOC were written. The project repository was available for public access at [27]. With the help of Eclipse's *Subclipse* plug-in and the *Trac* system, we were able to recover the work done between January 2008 when a branch was created for the project, and the end of October 2008 when the branch was finally merged to the ArchStudio trunk. A PhD student and a Masters student worked together on the project, and in total made 112 commits during this period. On average, there was one commit every two to three days.

We manually made all the recovered architecture changes, mapped them to code with the 1.x-way mapping tool, and again manually made recovered code changes. Our overall strategy was starting a new architecture change session for each repository commit where architecture changes were involved. There were also commits that were either for small bug fixes or made code changes only. They were simply merged to the previous "*unmapped*" change session, which was mapped to code when the next architecture-related commit arrived. In the case when more than one component was added to the architecture in a single commit, we assumed that they were developed in the top-down order and replayed that way. This was considering that the architecture of ArchStudio was organized into layers, with each component depending on its adjacent upper layer. Repeating this process finally generated 22 architecture change sessions, which in total consisted of 130 architecture changes.

As a result of applying 1.x-way mapping, about two thirds of the changes were mapped to code in a completely automatic manner: only the architecture-prescribed code was updated while the user-defined code remained. The rest were semi-automatically handled, meaning that manual modifications were also required in user-defined code. Table II shows the result of the evaluation. It is basically consistent with what is presented in Figure 3. One thing to note is that only 28 of 64 *Update Component* changes actually required users' manual work. This was because of the change analysis and refining process discussed in Section IV, where some changes were either discarded or addressed in the mapping of other changes. They were considered as being automatically handled since no manual work was actually required.

TABLE II. THE MAPPING OF ARCHITECTURE CHANGES IN ACTS

	Auto	Semi-auto	Total
Link Changes	49	x	49
Add Component	x	14	14
Update Component	36	28	64
Remove Component	3	x	3
Total	88	42	130

Code changes were more difficult to evaluate in this case, because the ACTS tool as an extension of ArchStudio was also built upon the *myx.fw* framework. The framework reduced the chance where architecture-prescribed code could be accidently changed given that it hard coded and encapsulated a portion of architecture implementation, including message exchange among components, architecture topology, and so on. Additional coding constraints were also induced due to the use of the framework, as mentioned earlier. In the evaluation, we decided to consider both direct changes of architecture-prescribed code and violations of the *Myx* coding constraints as errors that should be avoided. In this way, the effect of the framework was greatly negated. 16 such errors ended up being found in the implementation of the 11 ACTS components, all of which were successfully avoided during our replay with the help of 1.x-way mapping. One error discovered, for example, was that a reference to a connected component was initialized by simply calling its constructor. This did not break the architecture-code conformance, but it was not allowed by the *myx.fw* framework.

The risk to the validity of this evaluation is that there were some development changes that could not be recovered from examining the commit history of the project repository, and thus, failed to be replayed during the evaluation. The use of the *myx.fw* framework to some extent also affected the result of how 1.x-way mapping could help to prevent mistaken changes of architecture-prescribed code, even though measures were already taken to keep the effect minimal as discussed above. We believe our collective results are sustainable despite of these risks mainly because (1) all the essential changes and milestones of the project were covered in the evaluation, which was verified by one of original developers of the project; (2) the use of various software framework is quite popular in complex software development. In the end, both of these risks can be fully addressed by doing a long-term study of exploiting 1.x-way mapping in real software development, preferably industrial use.

C. Evaluation III: Support for Behavioral Mapping

To evaluate how the automation of change mapping would vary from the results collected earlier when behavioral changes were involved, we interestingly remade changes done to ArchStudio in the development of our 1.x-way mapping tool itself with the developed tool [28]. This *self-demonstration* was enabled by the Eclipse's development and runtime workbench [25]. It served as a compliment to our major evaluation with ACTS. Same methodology was applied, except that a number of adapted sequence diagrams and state diagrams were defined. For example, one sequence diagram enforced how the mapping tool interacted with the architecture modeling environment and code editor as described at the beginning of Section II. As the result of the evaluation, 118 architecture changes were recorded. 92 of these changes (about 78 percent) were automatically mapped to code, with the rest semi-automatically handled. Notice that the automation rate was even higher than what we got from replaying the ACTS project. This was because most behavioral changes actually do not require user's manual intervention. The only case in the evaluation that we had to manually respond to a behavioral change was when a sequence diagram was removed. Correspondingly, we had to manually implement the specified operation in the user-defined code.

VI. Related Work

The Eclipse Modeling Framework (EMF) [11] is a modeling framework and code generation facility that exploits the facilities provided by Eclipse. It supports automatic code generation, allows manual modification in specified regions, and provides a simple merge support called *JMerge*. Specifically, EMF adds a "*@generated*" tag in the Javadoc comments of generated classes and methods. It is the presence or absence of such tags that determine whether the associated code elements should be updated or left alone during code regeneration. As discussed in Section I, this mechanism does not solve the conformance issue between the model and code. An extreme case could be that all tags are removed and model changes finally have to be manually mapped to code.

DiaSpec [12] is a domain-specific ADL that integrates a new concept called *interaction contract*. As part of the architecture description, interaction contract describes the allowed interactions between components. In particular, its implementation is generated and encapsulated into a programming framework that is not modifiable by programmers. At this point, it is similar to 1.x-way mapping. The difference is that interaction contract uses *subclassing* to decouple its user-defined and generated code. Additionally, it provides no change management as 1.x-way mapping does, and simply relies on the Java complier to detect mismatches between existing code and new generated code. This is not sufficient for architecture changes that do not cause a compilation error. Moreover, subclassing as a shallow separation mechanism may also cause incompatibility between generated code and existing class hierarchies [29].

ArchJava [2] is an architecture-implementation mapping approach that unifies software architecture with implementation. A significant advantage of ArchJava is that it is able to maintain communication integrity or structural conformance. This is enforced by program rules and the ArchJava compilers. Compared with ArchJava, 1.x-way mapping is able to modify, extend, and reuse architecture and code independently, whereas ArchJava mixes the architecture and implementation in one artifact. 1.x-way mapping also has a wider work scope than ArchJava. For example, inter-component connections can only be implemented with method calls in ArchJava.

Archface [4] is similar to ArchJava in terms of using architecture description as part of the implementation. It develops a new interface mechanism that plays a role as ADL at the design phase and as a programming interface at the implementation phase. Archface exploits technologies of Aspect-Oriented Programming (AOP), such as *pointcut* and *advice*, to specify the collaboration among components. This makes its implementation limited to aspect-oriented programs. In addition, Archface relies on a complier to detect the architecture-code inconsistencies and does not support architecture change notifications.

An architecture framework is a piece of software that acts between a particular architectural style and a set of implementation technologies. It facilitates the architecture-implementation mapping by providing fairly well understood implementations, which assist developers in implementing systems that conform to the prescriptions and constraints of a specific architecture style. Examples include the *myx.fw* framework [13] and UniCon's built-in implementations of architecture connectors [14]. An architecture framework helps to establish the initial conformance between the architecture and code, but it does not support the mapping of changes, especially architecture changes that happen afterwards. As a result, an additional mapping approach, such as 1.x-way mapping, is required to manage the architecture-implementation conformance.

VII. Conclusion

1.x-way mapping addresses the issue of architecture-implementation conformance in the presence of development changes to both artifacts. Its novel features include suppression of mistaken changes of architecture-prescribed code, explicit recording and analysis of architecture changes, automatic mapping of specific kinds of architecture changes to code in specific ways, and support for behavioral mappings. The utility of 1.x-way mapping is demonstrated by applying it to the code and evolutions of ArchStudio, where the approach is developed and integrated. Not only unnecessary mappings are avoided, but also mappings that should be made are automated in specific ways. The extensive application of 1.x-way mapping in complex software development, however, requires further development of modeling and code generation technologies.

Acknowledgment

This work is supported in part by the National Science Foundation under grants CCF-0917129, CCF-0820222, and IIS-0808783. We would also like to thank Eric M. Dashofy for reviewing the paper and providing helpful comments.

REFERENCES

[1] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: bridging the gap between design and implementation," IEEE Transactions on Software Engineering, vol. 27, pp. 364-380, 2001.

[2] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation," in 24th International Conference on Software Engineering Orlando, FL: ACM, 2002, pp. 187-197.

[3] J. A. Diaz-Pace, J. P. Carlino, M. S. Blech, A. , and M. R. Campo, "Assisting the synchronization of UCM-based architectural documentation with implementation," in Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009.

[4] 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 (ICSE '10), 2010. ACM, pp. 75-84.

[5] R. N. Taylor, N. Medvidovic, and E. Dashofy, Software Architecture: Foundations, Theory, and Practice: Wiley, John & Sons. 2010.

[6] S. Kelly and J.-P. Tolvanen, Domain-Specific Modeling: Enabling Full Code Generation: Wiley-IEEE Computer Society Press., 2008.

[7] M. Sefika, A. Sane, and R. H. Campbell, "Monitoring Compliance of a Software System with Its High-Level Design Models," in Proceedings of the 18th international conference on Software engineering Berlin, Germany 1996, pp. 387 - 396.

[8] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "DiscoTect: A System for Discovering Architectures from Running Systems," in International Conference on Software Engineering, Edinburgh, Scotland, 2004.

[9] R. v. Ommering, F. v. d. Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," IEEE Computer, vol. 33, pp. 78-85, March 2000.

[10] M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct Architecture Refinement," IEEE Transactions on Software Engineering, vol. 21, pp. 356-372, 1995.

[11] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, EMF: Eclipse Modeling Framework (2nd Edition): Addison-Wesley Professional, 2008.

[12] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In Proceeding of the 33rd international conference on Software engineering (ICSE '11). 2011. ACM, pp. 431-440.

[13] Myx white paper: http://www.isr.uci.edu/projects/archstudio/myx.html

[14] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," IEEE Transactions on Software Engineering, vol. 21, pp. 314-335, April 1995.

[15] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language," in Working IEEE/IFIP Conference on Software Architecture (WICSA 2001) Amsterdam, The Netherlands, 2001.

[16] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, Documenting Software Architectures: Views and Beyond: Addison Wesley, 2002.

[17] J. Herrington, Code Generation in Action: Manning Publications Co., 2003.

[18] Bran Selic. 2003. The Pragmatics of Model-Driven Development. IEEE Softw. 20, 5 (September 2003), 19-25.

[19] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition), Addison-Wesley Professional, 2003.

[20] I. A. Niaz and J. Tanaka. Code Generation from UML Statecharts. In Proc. 7th IASTED Int. Conf. on Software Engineering and Application, pages 315–321, 2003.

[21] G. Engels, R. Hücking, S. Sauer and A. Wagner, UML collaboration diagrams and their transformation to java. In Proceedings of the 2nd international conference on The unified modeling language: beyond the standard (UML'99), Robert France and Bernhard Rumpe (Eds.). Springer-Verlag, Berlin, Heidelberg, 473-488.

[22] E. Gamma, R. Helm, R. Johnson et al., Design Patterns: Elements of Reusable ObjectOriented Software, Reading, MA: Addison‐Wesley Professional, 1995.

[23] E. M. Dashofy, H. Asuncion, S. A. Hendrickson, G. Suryanarayana, J. C. Georgas, and R. N. Taylor, "ArchStudio 4: An Architecture-Based Meta-Modeling Environment," in 29th International Conference on Software Engineering (ICSE 2007). vol. Informal Research Demonstrations, Companion Volume Minneapolis, MN, 2007, pp. 67-68.

[24] JET. http://eclipse.org/modeling/m2t/?project=jet#jet

[25] E. Clayberg and D. Rubel, Eclipse Plug-ins: Addison-Wesley Professional; 3 edition, 2008.

[26] Eric M. Dashofy, Supporting Stakeholder-driven, Multi-view Software Architecture Modeling. Ph.D. Dissertation, University of California, Irvine. 2007.

[27] ACTS. http://tps.ics.uci.edu/svn/projects/archstudio4/branches/traceability/

[28] 1.x-Way Mapping. http://tps.ics.uci.edu/svn/projects/archstudio4/branches/arch-imp-mapping/trunk/

[29] F. Budinsky, M. Finnie, and P. Yu, "Automatic Code Generation from Design Patterns," IBM Systems Journal, vol. 35, pp. 151--171, 1996.