UNIVERSITY OF CALIFORNIA,
IRVINE

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

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Yongjie Zheng

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor André van der Hoek
Assistant Professor James A. Jones

2012

DEDICATION

To

Mom and Dad.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

ACKNOWLEDGMENTS

First of all, I would like to thank Department of Informatics and Institute for Software Research at University of California, Irvine for providing me a wonderful environment and research resources. I have really enjoyed working on such a great platform. Therefore, I would like to express my gratitude to all the staff in these two organizations, especially Debi, Kiana, Kari, Marty, Dick, David, and Andre. Thank you all for helping me in the past years!

Special thanks goes to my advisor, Professor Richard N. Taylor, who recruited me to a research group that I have been very proud of and let me work on interesting projects. I thank Dick from the bottom of my heart for offering me all the support, guidance, and resources I needed to complete my Ph.D. study. In particular, I really appreciate the trust and space that Dick gave me in research, and always pointing me to the directions that later on I felt very excited about. These significantly fostered the development of my dissertation work. Last, but not least, I thank Dick for inviting me to a delicious homemade Thanksgiving dinner. Yes, that was unforgettable!

I would like to thank my dissertation committee members, Professor André van der Hoek and Professor James A. Jones. Thank you for spending time reading my survey paper, topic proposal, and dissertation, and giving me helpful feedback. I always think the time we spent together in my candidacy exam and topic defense is one of my most valuable experiences during my PhD study. Discussing my research with world-class researchers in two-hour period, in my eyes, is one of most exciting things.

In addition, a thank you to Professor Donald J Patterson and Professor Lizhi Sun, who both joined my candidacy committee. Thank you for your time and comments! What you did really meant a lot to me. I also want to thank Professor Nenad Medvidović of University of Southern California. I met with Neno several times, and he has been helpful to me all the time.

Many thanks to Eric Dashofy. Eric reviewed my ICSE paper, topic proposal, NSF proposal, and gave me challenging questions that I had to think over before I could respond. Those long conversations we once had by email not only made many ideas and concepts clear, but also helped to get my research ready to face a wider research community. Particularly, I want to thank Eric for developing ArchStudio 4, a wonderful system and platform based on which my dissertation work was done. Thank you, Eric, for everything!

Hazel Asuncion and Scott Hendrickson gave me many help when I first came to UC Irvine. Hazel also kindly answered all my questions regarding her ACTS project during the evaluation of my dissertation work. I would like to use this opportunity to express my gratitude to both of them for being such great group mates to me.

I would like to extend my thanks to Justin Erenkrantz, Michael Gorlick, Alegria Baquero, Kyle Strasser, Leyna Cotran, Erik Trainer, Rosalva Gallardo Valencia, HyeJung Choi, Derek Pfister, Francisco Servant, and Fang Deng. Justin, Michael, Alegria, and I worked together on the CREST demo project, which finally generated some impressive results. Kyle is my constant office mate, who comes to the office every day as I do and patiently answered my questions ranging from English to places to buy eyeglasses. Leyna gave a guest lecture to an undergraduate class that I once taught, and I really appreciate the time and effort she spent on that. Erik spent a whole morning in helping me record the demo video of my developed tool. His charming voice made the video absolutely better, and I know that was far more than the lunch I treated him. In addition, HyeJung, Derek, and I took many classes together and we worked together on several class projects. Paco, Fang, and I attended ASE 2011 together, and we had a great time there. Thank you, everyone!

Finally, I give my most sincere gratitude to my family. To my wife Yan and my daughter Janie, I love you so much and thank you for bringing me so much pleasure. To Dad and Mom, who have done so many things for me unconditionally, words are not enough to express my gratitude. Without your support, I do not think I am still on my way to pursuit my academic goal. I would also like to thank my brother Yongsheng Zheng, my sister Dongmei Zheng, my uncle Shubiao Liu, my cousin Gang Liu, and their families. I miss all of you in this big family. I love you all!

CURRICULUM VITAE

Yongjie Zheng

Field of Study

Software engineering, software architecture and design, architecture-implementation mapping, architecture-centric software development.

Education

2012	Ph.D. in Information and Computer Science University of California, Irvine
2005-2007	Department of Computer and Information Science and Engineering University of Florida
2005	M.Phil. in Department of Computing Hong Kong Polytechnic University
2000	B.Eng. in Department of Computer Science and Technology Tsinghua University, Beijing, China

Professional Experience

2008	Summer Intern, Novelics, Inc. Aliso Viejo, California
2007	Summer Intern Innovative Scheduling, Inc. Gainesville, Florida
2001-2003	Software Engineer IBM China Software Development Lab Shanghai, China

Teaching

Instructor. INF 111: Software Tools and Methods (21 students) - Summer 2010.

Teaching Assistant. INF 123: Software Architectures, Distributed Systems, and Interoperability – Spring 2009; INF 117: Project in Software Systems Design – Winter 2009; INF 121: Software Design I – Fall 2008.

Reader. INF 117: Project in Software Systems Design – Spring 2008; ICS 52: Introduction to Software Engineering – Winter 2008; INF 113: Requirement Analysis and Engineering – Fall 2007.

Publications

Yongjie Zheng and Richard N. Taylor. "Enhancing Architecture-Implementation Conformance with Change Management and Support for Behavioral Mapping", Proceedings of ICSE 2012: 34th International Conference on Software Engineering, June 2-9, 2012, Zurich, Switzerland. (to appear).

Yongjie Zheng and Richard N. Taylor. "xMapper: An Architecture-Implementation Mapping Tool", Informal Research Demonstration, Proceedings of ICSE 2012: 34th International Conference on Software Engineering, June 2-9, 2012, Zurich, Switzerland. (to appear).

Yongjie Zheng and Richard N. Taylor. "Taming Changes With 1.x-Way Architecture-Implementation Mapping", Short Paper, Proceedings of ASE 2011: 26th IEEE/ACM International Conference On Automated Software Engineering, November 6–12, 2011, Lawrence, Kansas.

Yongjie Zheng. "1.x-Way Architecture-Implementation Mapping", Doctoral Symposium, Proceedings of ICSE 2011: 33rd International Conference on Software Engineering, May 21-28, 2011, Honolulu, Hawaii.

Yongjie Zheng and Richard N. Taylor. "A Rationalization of Confusion, Challenges, and Techniques in Model-Based Software Development", ISR Technical Report UCI-ISR-11-5, August 2011.

Yongjie Zheng and Alvin T.S. Chan. "MobiGATE: A Mobile Computing Middleware for the Active Deployment of Transport Services", IEEE Transactions on Software Engineering, Jan 2006, vol. 32, no. 1, pp 35-50.

Yongjie Zheng, Alvin T.S. Chan, and Grace Ngai. "Applying Coordination for Service Adaptation in a Mobile Computing Environment", IEEE Internet Computing, Sep/Oct 2006, pp. 61-67, vol. 10, no. 5, IEEE.

Yongjie Zheng, Alvin T.S. Chan, and Grace Ngai. "MCL: A MobiGATE Coordination Language for Highly Adaptive and Reconfigurable Mobile Middleware", Software Practice and Experience, special issue on Auto-adaptive and Reconfigurable Systems, 36(11-12), pp1355-1380, 2006.

Yongjie Zheng and Alvin T.S. Chan. "Coordinated Composition of Services for Adaptive Mobile Middleware", Proc. of the 11th IEEE Symposium on Computers and Communications (ISCC 2006), June 26-29, 2006, Pula, Calkiari, Sardinia, Italy.

Yongjie Zheng and Alvin T.S. Chan. "Stream Composition for Highly Adaptive and Reconfigurable Mobile Middleware", Proceedings of 28th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2004), IEEE Press, 28-30 Sep 2004, Hong Kong.

Yongjie Zheng and Alvin T.S. Chan. "MobiGATE: A Mobile Gateway Proxy for the Active Deployment of Transport Entities", Proceedings of the 2004 International Conference on Parallel Processing (ICPP 2004), 15-18 Aug 2004, Montreal, Quebec, Canada, IEEE.

Yongjie Zheng, Yang Wang, Hanjian Fu, and Drew Schechter. "IBM CrossWorlds Collaborations as a Web Service", IBM WebSphere Developer Technical Journal, Aug 2002.

ABSTRACT OF THE DISSERTATION

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

By

Yongjie Zheng

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2012

Professor Richard N. Taylor, Chair

Software architecture plays an increasingly important role in complex software development. Its further application, however, is challenged by the fact that software architecture, over time, is often found not conformant to its implementation. This is usually caused by frequent development changes made to both artifacts. Against this background, how to automatically maintain architecture-implementation conformance becomes a significant research question. Without this issue resolved, architecture centrality can only exist in ideal situations where developers are highly disciplined or the system under development is relatively simple.

Architecture-implementation mapping is a process that specifically addresses the conformance issue mentioned above. Existing approaches can be roughly classified as one-way mapping and two-way mapping depending on which artifacts can be manually changed. None of them, however, provides a complete solution in the sense that mapping of changes is weakly supported and most approaches can support structural conformance only. In this research study, a new mapping approach called 1.x-way mapping is developed. Its name comes from the fact that it only allows manual changes to be initiated from the architecture ("1") and a separated portion of the code (".x"). 1.x-way mapping advances the area of architecture-implementation mapping with the capability of preventing mistaken changes of architecture-prescribed code by programmers, and supporting automatic mapping of structural and behavioral architecture changes to code.

1.x-way mapping consists of four core mechanisms: a deep separation mechanism, an architecture change model, architecture-based code regeneration, and architecture change notifications. In a nutshell, the architecture-prescribed code and user-defined code of each architecture component are separated into two independent elements. Architecture-prescribed code can only be updated through code regeneration, and programmers' manual changes are limited to user-defined code. All the architecture changes are explicitly recorded and classified in an architecture change model. They are automatically mapped to code through the regeneration of architecture-prescribed code and the delivery of change notifications to user-defined code if necessary. Behavioral architecture-implementation mapping is enabled during this process with system dynamics modeled in a form that can be automatically translated into architecture-prescribed code that cannot be contaminated when programmers work on user-defined code.

Empirical evaluation of the 1.x-way mapping approach consists of three case studies based on ArchStudio 4, an Eclipse-based architecture development environment where 1.x-way mapping is implemented and integrated. In the first case study, we refactored the code of ArchStudio with the deep separation mechanism. In the next two case studies, we re-did, or replayed, changes made to the architecture and code of ArchStudio in two research projects with the help of our new mapping approach. The first project built and integrated an architecture-centric traceability tool in ArchStudio, and the involved changes were structure-oriented only. The second project was the development of the 1.x-way mapping tool, where both structural and behavioral changes

were made to the ArchStudio architecture and code. The purpose of the evaluation is to determine if the 1.x-way mapping approach is applicable to real software development and its features are effective.

1 Introduction

This chapter highlights the importance and challenges of maintaining architecture-implementation conformance in the development of complex software systems. Based on the desire to address these challenges, a new architecture-implementation mapping approach is presented as the contribution of this research study. The research hypothesis and corresponding validation methods are also specified. The organization of the dissertation is given at the end of the chapter as further guidance to the reader.

1.1 Research Question

The increasing complexity of software systems makes continuously improving software productivity and quality difficult as long as it remains based on traditional code-centric development [16]. Meanwhile, software architecture—the set of principal design decisions made about a software system [143]–plays an increasingly important role in software development. To the extent this increase is present, it is due to the increased maturity of related technologies, such as architecture description languages [36, 47, 85, 87], architecture styles [19, 49, 142], and domain-specific/product line software architectures [111, 146]. Architecture-centric development represents the next logical step [11, 100, 141, 154]. Different from traditional software engineering where software architecture is simply seen as a documentation artifact that is peripheral to code development, architecture-centric development emphasizes that software architecture should play an essential role throughout the product lifecycle. It represents a paradigm where software architecture is used not only horizontally—to describe [54] and analyze [71]—but also vertically to synthesize [18], integrate [40], and evolve [114] software systems.

Several different forms of architecture-centric development have emerged in recent years, including model-driven development [52] and architecture-based research [58, 113]. On the one hand, these approaches further reveal the benefits of architecture centrality in software development. On the other hand, none of them dominates "the practice." A primary reason is that the architecture of a software system, over time, is often found not conformant to its implementation [129]. In other words, a solution to a central piece of architecture-centric development, architecture-implementation mapping [3, 42, 103, 123], is still missing. Architecture-implementation mapping is a process of converting architecture specifications to and from source code with the goal of maintaining their conformance with respect to certain criteria. There are a number of architecture-implementation mapping approaches, such as architecture frameworks [88] and code generation [69], but most of them are deficient in the sense that change mapping and behavioral conformance are not supported. As a result, software developers often have to manually maintain architecture-implementation conformance, which is not only time consuming, but also error prone. Generally speaking, software architecture can easily become out of date if the cost of maintaining its conformance to code significantly exceeds that of working on code directly.

The research question that this study addresses is *how to automatically maintain the conformance between software architecture and source code during software development*. By *automatically*, it is meant that both the architect and programmers can solely focus on their own artifacts without worrying about the inconsistency that their development work may cause. Note that programmers' manual work on the code (e.g. to implement a new architecture element) is still acceptable, given that automatic programming [10, 48] is not a practical approach yet at this point. *Conformance* in the above statement means that not only the source code does not lose properties of the architecture, but also that no new properties about the architecture can be inferred from the code. In other words, we require the source code to be a faithful interpretation [98] of the architecture. Depending on what information the architecture may contain, the specific criteria of conformance could be communication integrity [3], relative substitutability [56], and so on.

Automatic maintenance of architecture-implementation conformance directly determines the degree to which software architecture can be used in development to improve software productivity and quality. In particular, a complete resolution to this research question is not yet available at this point. We believe it is primarily due to the following significant challenges that this process involves:

- Software architecture and source code are located at separated abstraction levels and are usually expressed using different conceptual constructs [124]. Many architecture constructs, such as architecture components and connectors, often do not have direct counterparts in the programming paradigm. What this means in the context of architecture-implementation mapping is that the mapping of code back to architecture is essentially an activity of abstraction, which is hard to be fully automated (i.e. machine-based abstraction).
- Both software architecture and source code may be under frequent changes during software development [117]. Many of these changes endanger the conformance established between the architecture and code. In particular, no explicit change management mechanism is provided to either regulate or analyze changes that are made to the two types of artifacts. As a result, not only how to map changes between software architecture and code forms a challenge, but also whether or not to do so

becomes a problem. This is especially the case considering that some changes, such as modifications of implementation details, may or may not affect architecture-implementation conformance.

- Software architecture encompasses many aspects of the system under development, including structure, behavior, and non-functional properties. In contrast, most existing 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 [140]. Protection of architecture-prescribed code becomes extremely difficult in this situation. With respect to non-functional properties, they are not considered in this study as architectural modeling of these properties is not fully supported by the existing technology [26].

Overall, a successful solution of maintaining architecture-implementation conformance must be able to convert architecture to and from source code when development changes occur to either of them. It must do so in an intelligent way, meaning taking different actions for different types of changes. In particular, the conversion process should be able to span across the abstraction gap between the two artifacts [109]. In most cases, this requires the involvement of code generation both structurally and behaviorally.

1.2 Contribution

In this research study, we present a new architecture-implementation mapping approach called *1.x-way mapping*. Different from existing mapping approaches, which are classified as *one-way mapping* and *two-way mapping* depending on which artifacts can be manually changed, 1.x-way mapping only allows changes to be initiated in the architecture ("1") and a separated portion of the code (".x"). Architecture-prescribed code is updated solely through code generation. Overall, 1.x-way mapping consists of the following four core mechanisms:

- Deep separation. 1.x-way mapping separates architecture-prescribed code and user-defined code of each architecture component into two independent language elements (e.g. classes), and relies on an explicit program composition mechanism (e.g. method calls) to integrate the code. This is different from existing code separation approaches, such as filling-in-the-blanks and subclassing, where separated code is implicitly coupled and integrated by some built-in language relationship (same class, inheritance, etc.). Deep separation has several advantages, including comprehensive code protection, enforcement of architecture centrality, and mutual independence of separated code.
- An architecture change model. 1.x-way mapping explicitly maintains an architecture change model that records all the development changes made to the architecture. The architecture change model provides information that is necessary for the mapping of architecture to code, such as the element that is changed, the type of changes, and whether or not these change have been mapped to code.
- Architecture-based code regeneration. 1.x-way mapping updates architecture-prescribed code through code regeneration. It only regenerates code for modified components. For each modified component, complete regeneration is enforced. This special design not only protects the integrity of component implementation, but also reduces the amount of code regeneration.
- Architecture change notification. An architecture change notification contains information describing what element is changed in the architecture. It is sent to user-

defined code when certain architecture changes occur. In this way, programmers can make corresponding changes to their code and still keep the conformance between the architecture and source code.

Specifically, 1.x-way mapping works as follows. Architecture-prescribed code and user-defined details of each architecture component are decoupled into two independent program elements (e.g. classes). Manual code changes are limited to user-defined code, and thus, cannot contaminate the architecture-prescribed code. All the architecture changes are recorded and classified into the architecture change model. Most of these changes can be automatically mapped to code through the architecture-based code regeneration mechanism. For architecture changes that may require modifications to user-defined code, architecture change notifications are also generated and sent across the separation boundary to user-defined code, describing what element is changed in the architecture. In particular, 1.x-way mapping can support the behavioral architecture-implementation mapping with system dynamics modeled in a form that can be automatically translated into code in a way that maintains code separation.

1.3 Hypothesis

The hypothesis of this study is that *1.x-way mapping can be applied in the development of a realistic system to prevent the architecture-prescribed code from being accidentally or intentionally changed by programmers, and support automatic mapping of structural and behavioral architecture changes to code*. 1.x-way mapping and particularly its code separation mechanism should be applicable to the implementation of a real program of significant complexity. 1.x-way mapping must also be able to automatically map specific kinds of architecture changes to code in specific ways. Note that the manual changes to user-defined code for logic completion are not considered as a violation of the hypothesis. Instead, we still consider

1.x-way mapping as being able to support automatic change mapping as long as it can automatically update corresponding architecture-prescribed code and send change notifications to user-defined code when necessary. Significantly, all the above features are applicable to both structural and behavioral architecture specifications. In the current implementation of 1.x-way mapping, behavioral architecture is modeled in the form of UML-like sequence diagrams and state diagrams. Support for other forms of behavioral specifications are pending on the development of corresponding modeling and code generation technologies. This is specifically discussed in Chapter 4. Finally, limitations apply of course: non-functional properties are not considered in this study.

To validate the hypothesis, we perform case studies with a pre-existing software system, ArchStudio 4 [39], a Eclipse-based architecture development environment that is being used at UC Irvine, in several companies, and at several universities. A primary benefit of exercising 1.x-way mapping with ArchStudio 4 is that it has been extended in several independent research projects, where significant changes were made to its architecture and code. Therefore, we can replay, or re-do some of these changes with the help of our 1.x-way mapping approach. The purpose is to determine if 1.x-way mapping (a) can be applied to a real software system to protect its architecture-prescribed code during the development, (b) automatically map architecture changes to the code, and (c) provide support for both structural and behavioral architecture specifications. Applicability and effectiveness are the two dimensions that we will be specifically focused on during the evaluation.

1.4 Organization of the Dissertation

Chapter 1 describes the research question, contribution, and hypothesis of this study. It presents the significance and challenges of maintaining the conformance between software architecture and code during the development.

Chapter 2 provides a classification of existing architecture-implementation mapping approaches. It specifically reviews some representative approaches, with their limitations highlighted. This forms an important motivation of this research study.

Chapter 3 introduces related technologies of 1.x-way mapping, including architecture-centric software development, architecture modeling, code generation, and software change management. These technologies together form the application context of 1.x-way mapping, and many of their pragmatic techniques are reused in the development of 1.x-way mapping. For example, a template-based code generation mechanism is applied in 1.x-way mapping to build its code generator. Thus, it is necessary to give an introduction to these approaches before we elaborate the 1.x-way mapping approach.

Chapter 4 is devoted to the specifics of 1.x-way mapping. It starts from design principles and an overview of the approach, and focuses on the four core mechanisms of 1.x-way mapping: a code separation mechanism, an architecture change model, architecture-based code regeneration, and architecture change notification. Support for the behavioral architecture specifications is also discussed from the perspective of behavioral modeling and applying code separation to the behavioral code. At the end of the chapter, a comparison of 1.x-way mapping and the existing mapping approaches introduced in Chapter 2 is provided. This highlights the new features of 1.x-way mapping.

Chapter 5 focuses on the implementation of 1.x-way mapping. It introduces the implementation environment, specific tasks, tool usage, and lessons learned from the implementation experience. A number of issues are specifically discussed in this chapter, such as how to deal with removed elements during architecture change recording, the analysis and refinement of architecture changes, and the correlation between the behavioral architecture elements and structural elements during code generation.

Chapter 6 is about the validation of 1.x-way mapping. The validation work is deeply rooted in the hypothesis of this study. It consists of three case studies, which are meant to evaluate different aspects of 1.x-way mapping. For each of these case studies, the evaluation method, collected results, threats to validity, and conclusion are presented. At the end of chapter, it is discussed why we believe the results collected from our evaluation can be generalized to the development of other real software systems.

Chapter 7 offers conclusion to this study. It also points out some directions for the future research activities on this topic. Such work is necessary to make 1.x-way mapping more complete and effective in terms of maintaining architecture-implementation conformance. In the long term, these activities have the potential to make architecture-centric development an approach that can be widely adopted in the development of complex software systems.

2 Architecture-Implementation Mapping

Architecture-implementation mapping is a process of converting architecture specifications to and from source code with the goal of maintaining their conformance with respective to certain criteria. A number of approaches have been developed to automate this process. This chapter provides a classification of existing architecture-implementation mapping approaches, and specifically reviews some representative approaches of each category. At the end of the chapter, limitations of existing approaches are summarized to highlight the motivation of this research study.

Current architecture-implementation mapping approaches either rely on after-the-fact consistency checking to detect any inconsistency (*correct-by-detection*), or apply technologies like code generation to avoid inconsistency from the very beginning (*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 perspective, there are approaches of *one-way mapping* and *two-way mapping*. Table 2-1 presents a classification of these approaches. The italicized words represent instances of each approach. 1.x-way mapping is also shown in the table as a preview, although detailed descriptions about it are given in Chapter 4.

Note that correct-by-detection approaches usually detect the architecture-implementation inconsistency by extracting or inducing the architecture from the code, and comparing the obtained architecture with the prescriptive architecture. They assume the relative constancy of software architecture, since source code is the only focus during inconsistency checking. This explains why there is no two-way mapping of correct-by-detection in the table.

	Correct-by-construction	**Correct-by-detection**
One-way mapping	1) Full code generation *Domain-specific MDD [79]*, *DSSA [111]* 2) Architecture refinement *SADL [98]*	1) Reverse engineering *Reflexion model [103]* 2) Runtime monitoring *DiscoTect [153]*, *ArchSync [42]*, *Pattern-Lint [123]*
Two-way mapping	1) Code generation and separation *EMF [135]*, *DiaSpec [20]* 2) Architecture frameworks *myx.fw [34]*, *UniCon [128]* 3) Unifying descriptions *ArchJava [3]*, *Archface [148]* 4) Roundtrip engineering *Fujaba [107]*	
	1.x-way mapping	

Table 2-1: Architecture-implementation mapping.

2.1 One-Way Mapping

One-way architecture-implementation mapping mandates that all manual changes begin from either the architecture or the code (but not both), with the other artifact automatically updated by a mapping approach. In the category of correct-by-construction, the architecture can be manually changed and the mapping is from the architecture to code. The technologies of full code generation and architecture refinement are mostly used for the mapping purpose. In contrast, the mapping is usually from the code to architecture in correct-by-detection and the architecture is assumed to be constant. Reverse engineering and runtime monitoring are two typical technologies that support this reverse mapping process.

2.1.1 Full Code Generation

Full code generation is a representative approach of correct-by-construction. It is extensively used in some Model-Driven Development (MDD) approaches, which aim to make architecture models compilable and executable, and become the main artifacts of development. In essence, what full code generation is trying to do is promote software development to the level of software architecture, so that source code editing can be completely avoided. This faces the challenge of bridging the abstraction gap. On the one hand, software models in MDD must have sufficient detail to enable complete program generation; on the other hand, the models also need to be, and stay, simpler than the corresponding software programs created during this process. As a result of this challenge, full code generation is currently only applicable in some highly specialized domains with the help of domain-specific modeling languages [79], code generators [25], and software architecture [64, 111].

Domain-specific MDD. Domain specificity relates to the applicability of an approach to different domains. It classifies MDD into generic and domain-specific approaches. Generic MDD approaches such as Model Driven Architecture (MDA) [81] use a domain-independent vocabulary and mechanism that are extensible enough to be adapted to different application areas. This is specifically introduced in Section 3.3. In contrast, domain-specific MDD is closely related with the process of domain engineering, whose purpose is to develop domain artifacts that may be used (and reused) in developing applications for a given domain. Examples of reusable domain artifacts in domain-specific MDD include domain-specific languages (DSLs) [137] and application generators [25].

A DSL adopts representation formalisms and modeling constructs of established engineering disciplines – there is no need to learn yet another modeling language. It offers highly efficient constructs to capture design requirements and constraints. Compared with a general modeling language, DSL is more expressive and therefore tackles complexity better, making software development easier and more convenient. Most importantly, DSLs raise the level of abstraction and together with domain-specific generators, can automate the creation of high-quality code. The primary difficulty with DSLs, however, is that each language needs its own set of tools. These tools will need to evolve as the domain evolves. Building and evolving these tools using manual techniques can be expensive.

Application generators are tools for creating application programs from the specifications that capture domain variations [24]. In particular, application generators are usually used for the development of a whole application family, not just a single application. Examples include Bison [60] and LEX [43] that have been widely used in the program compiler area. The working process of an application generator is as follows. The system analyst and system designer build specific applications, while the domain analyst and the domain designer build the application generators used by the system designers. The domain analyst specifies the requirements of an application generator for a range of problems. The domain designer takes these specifications and implements them in a generator. Similarly, the system designer takes the system specification from the system analyst and uses the produced application generator to finally generate applications for customers. To change or modify the product, the system analyst simply changes the system specification and asks the system designer to regenerate the software. In particular, generated programs do not have to be directly modified.

Domain-specific MDD has been successfully applied to databases, user interfaces and program compliers. However, there are two important factors that limit domain-specific MDD from being widely used in many application areas. First, the use of application generators has a very high requirement on the maturity of an application domain. They are only applicable in those limited situations where the domain is so thoroughly understood and bounded that generation is feasible. In addition, application generators and DSLs are difficult to build. They require expert knowledge and skill in both the application domain and building parsers and language translators. In general, the development costs of application generators can be considerably more than the development of an individual application, and must be compared with the long-term benefits of reuse.

DSSA. Domain-specific software architecture (DSSA) promotes the reuse of domain knowledge to a high level of abstraction, and provides a new composition mechanism since software component assembly requires much higher levels of adaptation than the assembly of physical components. A DSSA comprises (1) a reference architecture, which describes a general computational framework for a significant domain of applications; (2) a component library, which contains reusable chunks of domain expertise; (3) an application configuration method for selecting and configuring components within the architecture to meet particular application requirements [143]. As far as the mapping of architecture to implementation is concerned, DSSA is favorable in the following three aspects.

First, the reference architecture in DSSA serves as a foundation based on which a specific architecture could be created through architecture specialization. In other words, the reference architecture raises the level of software reuse to architectural abstractions, and thus, software production is increased.

Second, the library of reusable components in DSSA simplifies architecture implementation to the process of component composition [30]. Significantly, the use of reference architecture reduces component mismatch [55] and simplifies the management of supplier relationships by describing the contexts in which components operate.

Finally, the associated configuration method of DSSA provides potentials for the generative software development, which automatically generates a software system from its requirements specification through the assembly of reusable components. One possible way to select and configure components based on requirements is suggested as follows [77]: the functional requirements are mainly used to identify required components, while non-functional requirements are used to partition components, to select components from alternative ones with the same function, or to select types of connectors between components.

A representative example of applying DSSA techniques in software development practice is the use of the Koala model and architecture description language to create a family of television products in Philips [111]. In essence, Koala is a combination of component models and architecture description languages to deal with product populations. The Koala component model emphasizes context independence through the separation of communication from computation in component development. By this means, different combinations of reusable components can be made for different products. The Koala language extends the Darwin ADL [86] to support the addition of modules between components and a diversity interface mechanism. In particular, the reusable components are parameterized over all configuration-specific information.

It is important to note that the process of developing DSSA could be much more expensive than developing an individual system, and often needs a close cooperation between domain experts and experienced application engineers. It is for this reason that the creation of a DSSA for a domain should be carefully considered based on the evaluation of the expected savings against its building cost. In general, DSSA is mostly used in the development of product-line, or family of applications.

2.1.2 Architecture Refinement

Architecture refinement is the process of mapping an abstract architecture into a lower level architecture that is intended to implement it. It is usually used in the construction of an architecture hierarchy that describes a large software system. In general, an abstract architecture is smaller and easier to understand; a concrete architecture reflects more implementation concerns [12, 90].

A traditional approach to mapping architectures at different levels in a design hierarchy is taken by Rapide [85] through the use of architecture simulation and event pattern mapping. Each architecture instance is associated with an event-based execution model, and the simulation of architecture generates a partially ordered set of events (posets). The predefined event pattern mappings then map posets of events in one architecture into posets in another, based on which the consistency of the two architectures is checked. There are two important limitations about the Rapide approach. First of all, event pattern mappings are defined at the level of architecture instances. This prevents them from being reused by other systems (architecture instances). Second, the consistency check in Rapide only emphasizes functional conformance. However, there may be properties other than behavior equivalence that need to be preserved in the concrete architecture.

Style-based architecture refinement [56] takes a step further along the above-mentioned two aspects, and transforms an abstract architecture into a concrete architecture through a series of small refinements, each of which involves the application of a set of transformation rules. In particular, the rules are defined between architecture styles, a named collection of architectural design decisions that includes a set of constraints put on development to elicit beneficial properties. It permits creation and verification of rules to be done by style specialists, while allowing system designers to simply reuse the result without proof. Another important property of style-based refinement is that it enforces a stronger correctness criterion than functional conformance in the refinement process. In essence, this is necessary because software architecture, as a set of principal design decisions, characterizes multiple aspects of a system, including functional, structural, interaction, and non-functional concerns.

In general, techniques that apply at the level of styles are much more powerful than techniques that apply to instances. This is because the demonstration can be performed once for the styles and then reused many times for instances of those styles. The flip side of the coin is that sometimes it is difficult to build or prove something that works for every instance of a style. David Garlan addresses this problem in their style-based refinement approach by identifying subsets of the systems in a given style, so called substyles, and defining specialized refinements for each substyle, instead of the whole style. This is argued as being more close to what engineers actually do when they implement architectural designs. The primary challenge, however, is that how to identify a suitable substyles for the refinement definition.

A special requirement on architecture refinement is that the process must be correctness-preserving with respect to some criterion. Simply speaking, if the original architecture has some property of concern, the refinement must exactly preserve the property in the derived architecture. A criterion that is mostly seen during architecture refinement is *communication integrity*. That is, concrete architecture components only communicate with the components they are connected to in the abstract architecture. This criterion essentially emphasizes the preservation of structural architecture decisions in the refinement process. Another more flexible criterion used in a style-based refinement approach is *relative substitutability*, which means the concrete architecture must be conformable to the abstract architecture with respect to a set of properties of interest. These properties could be performance, security, or any other system concern. Strong or weak, the proof of refinement correctness is never an easy thing to do given that manual formal reasoning is heavily involved. This is an important reason that most style-based refinement approaches assume the existence of pre-proved refinement patterns, and emphasize the definition of refinement at the reusable style level.

The architecture refinement approach has been successfully used to design an architecture for an operational power control system implemented in 200,000 lines of FORTRAN 77 code [98]. The resulting system has a reference architecture at two levels of detail: the abstract architecture was in a dataflow style, and the concrete architecture was a combination of a call-return style, a shared memory style, and a special process synchronization style. Significantly, the concrete architecture is correct with respect to the abstract architecture.

2.1.3 Reverse Engineering

Reverse engineering is the process of creating higher-level abstractions from source code that are less implementation-dependent [22]. It analyzes a subject system to identify the system's components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction. In general, reverse engineering does not involve changing the subject system. It is a process of examination not change or replication. Figure 2-1 clearly shows how reverse engineering is related with other software development activities, such as forward engineering, reengineering, and restructuring (i.e. refactoring).

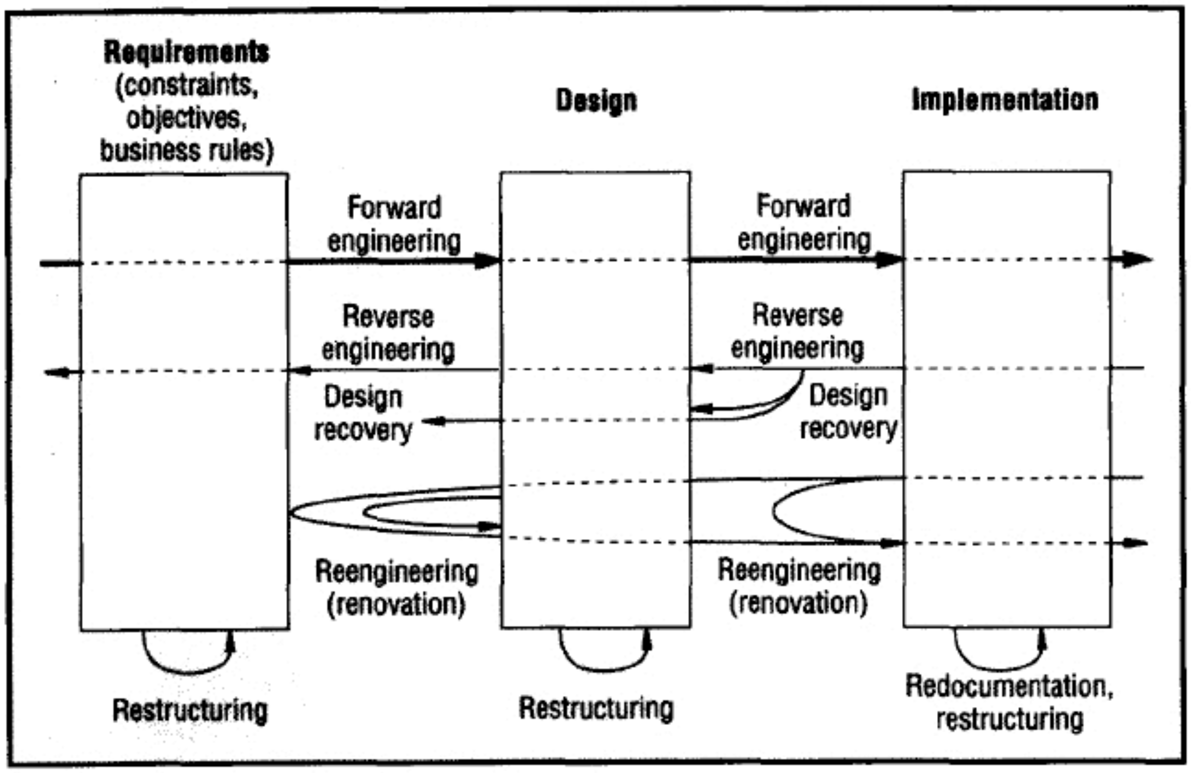


Figure 1. Relationship between terms. Reverse engineering and related processes are transformations between or within abstraction levels, represented here in terms of life-cycle phases.

Figure 2-1: Reverse engineering diagram excerpted from [22].

When applied to maintain architecture-implementation conformance, reverse engineering is reduced to the activity of design recovery shown in the figure and represents an after-the-fact detection technique. It abstracts source models from modified implementations, and compares the original source model with the generated one either to recover lost information or detect side effects [15]. This usually happens during software maintenance, when the system's maintainers, not its designers, must expend many resources to examine and learn the system. In this context, reverse engineering can help them understand the system and make appropriate changes. Note that reverse engineering can be expensive for complex systems; moreover, it is hard to guarantee that the generated model captures the same aspects that the original source model contains, since they may represent two different abstractions of the same implementation.

Reflexion model. A typical example of maintaining architecture-implementation conformance through reverse engineering is the software reflexion model technique [103]. It is used to produce a high-level structure model qualified for system reasoning. Specifically, an engineer defines a high-level model of interest, extracts a source model (such as a call graph) from the source code, and defines a mapping between the two models. A software reflexion model is then computed to determine where the engineer's high-level model does or does not agree with the source model. In essence, a reflexion model summarizes a source model of a software system from the viewpoint of a particular high-level model.

The reflexion model technique is particularly useful for software engineering tasks like software comprehension, and matching designs with implementations. Its primary limitation is that only structure information is used in the process of model comparison. In reality, the design and implementation could diverge from each other along many other aspects, including functional behaviors and non-functional properties. Moreover, the process of model comparison is most often done by human interpretations of the software reflexion model. This prevents it from being widely used in the development of complex software systems.

Another example of combining reverse engineering and after-the-fact consistency checking concerns the architecture of the Linux kernel as presented in [15]. It is akin to the reflexion model in many ways: both need a high level conceptual architecture as a general guide; both use existing source code extraction tools to extract used/defined relationships between functions, variables, and source files; and both have to depend on human interpretation for the comparison of the low-level concrete architecture and the high-level conceptual architecture. The main difference is that the Linux example uses existing documentation and knowledge of related systems to form the conceptual architecture, while the reflexion model assumes that such a conceptual architecture already exists.

2.1.4 Runtime Monitoring

Runtime monitoring is another branch of correct-by-detection mapping approaches. Runtime monitoring approaches infer the system architecture from execution traces or system events that are collected at runtime. Specifically, the runtime monitoring process consists of three specific steps: (1) observing a system's runtime behavior; (2) interpreting that runtime behavior in terms of architecturally meaningful events; (3) representing the resulting architecture. The approaches of runtime monitoring are favorable in terms of being able to check the system behaviors against the original architecture. To do this, the availability of executable software is usually required. Some approaches also demand certain forms of code instrumentation. This prevents dynamic verification from being used at development time, when programs are often not complete enough to be executed.

DiscoTect. DiscoTect [153] is a system for discovering the architectures of running object-oriented systems. It is particularly focused on the problem of bridging the abstraction gap between system observations and architecture effects, which is essentially the second step identified above. DiscoTect develops a language that defines the mappings between implementation patterns and architecture elements. Given a set of implementation conventions or styles and a vocabulary of architecture element types and operations (i.e. architecture styles), a mapping can be defined in that language to capture the way in which runtime events should be interpreted as operations on elements of the architecture style. In particular, the defined mappings can be reused across programs that are implemented in the same style. Specific examples of an implementation style in DiscoTect could be naming specific classes of a system in a pre-defined way. Architecture style in the context of DiscoTect is basically same as what is defined previously, such as a pipe-filter architecture style, except that a list of operators are also defined for a specific architecture style.

A primary limitation of DiscoTect is that it only works when an implementation obeys regular coding conventions. Completely ad hoc bodies of code are unlikely to benefit from the technique. In addition, DiscoTect also requires the identification of an architecture style, so that mappings can be created. Finally, as with other techniques based on runtime monitoring, DiscoTect can only analyze a system that is actually executable.

ArchSync. ArchSync [42] is similar to the reflexion model discussed earlier in many ways. In particular, both systems use pre-existing high-level models as a reference during the extraction of source models. The difference is that the design models supported by ArchSync are Use-Case Maps [26], which model functional scenarios by means of causal paths that cut across design structures. In this way, UCMs are able to capture both structural and behavioral information at a high-level of abstraction. Another difference between ArchSync and reflexion model is that ArchSync generates action scripts that can be used to automatically update the source model when inconsistencies are detected, whereas only simple mappings are created between the extracted model and the pre-existing model in the reflexion model. However, automatic generation of these action scripts or synchronizing UCMs relies on a correlation heuristic technique that may be hard to scale to complex software systems.

Pattern-Lint. Pattern-Lint [123] is a computer-assisted approach for confirming that the implementation of a system maintains its expected design models and rules. Different from traditional "reverse engineering" style analysis, Pattern-Lint improves compliance checking by combining static analysis of data sharing and method calls with code instrumentation-based dynamic visualization. By this means, it is able to support checking for conformance to a variety of design principles, including architectural structure, implementation guidelines, and non-functional properties like high cohesion and low coupling. In particular, many aspects of the checking process are automated, based on an explicit specification of conformance rules concerning different aspects of the system. Compared with other consistency checkers, support for multiple design principles and computer assistance are the two most important advantages of Pattern-Lint. However, Pattern-Lint is essentially an after-the-fact checking tool, so it has to depend on the assumption that the implementation is appropriately constructed. Moreover, its support for other non-functional properties, especially those not visualizable, is still an issue without being explicitly addressed.

2.2 Two-Way Mapping

Two-way mapping approaches recognize the essential roles of both architecture and code during software development, and allow manual changes to be initiated in both artifacts. Compared with one-way mapping, especially technologies like full code generation, two-way mapping is more practical given current modeling and code generation technologies. In particular, the fact that both architecture and source code may be changed is more close to the software development scenario. However, this also means more challenges since both the architecture-to-code mapping and code-to-architecture mapping are involved during this process. As a result, most two-way mapping approaches as described in the following sections can only support structural conformance between architecture and source code. Finally, as noted earlier, two-way mapping is only limited to approaches of correct-by-construction given the existing analysis techniques of correct-by-detection.

2.2.1 Code Generation and Separation

Code generation and separation is commonly used in practice to help maintain the architecture-implementation conformance. Approaches in this category automatically generate architecture-prescribed code, and separate the generated code from user-defined code by using some primitive code separation mechanisms (e.g. filling-in-blanks). These approaches are similar to our 1.x-way mapping approach in that all of them recognize that generated code should be separated from user-defined code and be protected from manual modification. However, it is important to highlight that there are some significant differences.

First of all, the code separation mechanism used in 1.x-way mapping is different from existing separation mechanisms. This is specifically discussed in Chapter 4. Simply speaking, existing code separation mechanisms are called *shallow separation* or *spatial separation* in this study given that their code is physically separated, but is still coupled and implicitly integrated by some inherent language relationship (e.g. same class, inheritance, etc.). In contrast, the code separation mechanism used in 1.x-way mapping is called *deep separation* or *linguistic separation*. Compared with shallow separation, deep separation not only provides a better protection of generated code, but also has some other advantages, such as support for behavioral code. Another difference is that most existing code generation and separation mechanisms do not have explicit change management mechanisms. As a result, the architecture and code become inconsistent soon after the first round of code generation.

Code generation technology is presented in Chapter 3. The existing mechanisms of code separation are specifically discussed below in this section, including filling-in-blanks [135], subclassing [18], and partial classes [99].

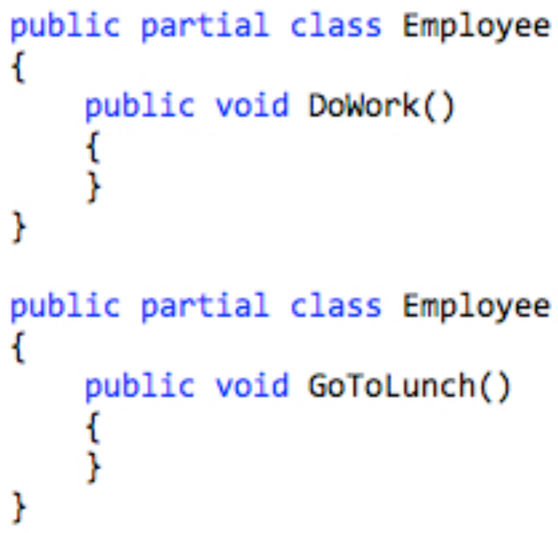
Filling-in-blanks, or protected code region, is a code separation mechanism that has been widely used in some Computer-Aided Software Engineering (CASE) tools [73, 122, 131]. It differentiates generated and non-generated code by including some human-understandable comments, such as “Do not delete”, “To be completed by user”, etc. The generated code is usually class names, method signatures, and pre-defined variables, while non-generated code is mostly the implementation of specific methods. The filling-in-blanks mechanism is easy to implement, but it fails to provide real protections to generated code. This is primarily because its generated and non-generated code are still physically mixed in the same program element, and both are under the control of programmers. In this context, it only works under the assumption that programmers are highly disciplined. Even so, accidental changes to generated code are still a possibility.

Subclassing is another code separation mechanism that is often used for object-oriented programs. For each class generated, the subclassing mechanism generates two classes: a base class that contains generated code and a subclass that contains user’s manual modifications. In particular, any user-specified changes must be made to the subclass only and the user never alters the core base class. The inheritance relationship lets the user re-define or extend operations in generated code, such as adding new operations or adding new instance variables. Should the code require regeneration later, the tool overwrites only the core class. The user’s changes remain unaffected.

Compared with filling-in-blanks, subclassing separates code into two separate elements. This to some extent prevents generated code from being manually modified. A primary limitation of subclassing, however, is that it requires the use of generated code as base classes. This may be a problem if the application that incorporates generated code has already developed its own class hierarchy. Using a programming language (e.g. C++) that supports multiple inheritance partially addresses this problem, but not all languages support this. Besides, the inheritance itself is more about reusing code of an existing object rather than integrating code.

Partial class is a code separation mechanism that was recently developed by Microsoft. Its usage is limited to those programming languages that support this feature, such as C# 2.0 and Visual Basic 2005. Partial class splits the definition of a class, a struct, or an interface over two or more source files. Each source file contains a section of the class definition, and all parts are combined when the application is compiled. Figure 2-2 shows an example of partial class, where the *partial* keyword indicates all the parts of a class. Using partial classes, generated and non-generated code are separated into different source files, while still maintaining the mutual independence.

Partial class currently is not broadly adopted. A primary criticism of partial class is that it breaks the concept of a class being a single entity with a single concern. Partial class, instead, introduces the concept of the part of a class being a single entity with a single concern. The fact that partial classes belong to the same class also incurs additional constraints to separated code. For example, partial classes or separated code of a class cannot contain methods that have the same signature. In other words, the code that can be protected by partial class is still limited by the way that separated code is integrated.



```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

Figure 2-2: An example of partial class.

EMF. Eclipse Modeling Framework (EMF) [135] is a modeling framework and code generation facility that exploits the facilities provided by Eclipse. It supports defining a model using Java interfaces, UML diagrams, or XML schemas, from one of which an *Ecore* model can be created. Ecore is a small and simplified subset of full UML, and is concerned with only one aspect of UML, class modeling. In essence, an EMF model is the class diagram subset of UML; that is, a structure model of the classes, or data, of the application.

The EMF code generator can not only generate corresponding implementation classes for a model, but also a functional editor plug-in integrated into the Eclipse IDE that can be used to create and edit instances of the model. In addition, EMF provides a runtime framework that can work with generated code for the purpose of model change notification and persistence support. EMF-generated code is meant to be modified. EMF uses @generated markers in the Javadoc comments of generated interfaces, classes, methods, and fields to identify the generated parts. It is the presence or absence of such tags that determines whether the associated code elements should be updated or left alone during regeneration.

DiaSpec. DiaSpec [20] 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. The interaction contract uses subclassing to decouple its user-defined and generated code. Additionally, it provides no change management, 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.

2.2.2 Architecture Frameworks

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. 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. The most common example of an architecture framework in industry is the standard I/O library in UNIX, which is actually a bridge between the pipe and filter style and procedural programming languages like C. Two major framework initiatives from academia are *myx.fw* and *UniCon*. Both are specifically introduced in the following.

myx.fw. The myx.fw class framework [34] is an extensible framework of abstract classes for the architecture style of *Myx*. The Myx style is a set of rules for composing the components and connectors of an application like ArchStudio. It provides patterns of composition for synchronous and asynchronous interactions among components. It also provides rules for what kinds of assumptions components may make about each other, ensuring a directed/layered ordering of dependencies among components. The rules of the Myx style include (1) Components are used as the loci of computation; (2) Connectors are used as the loci of communication; (3) Components communicate only through well-defined provided and required interfaces; (4) Components and connectors have two 'faces', 'top' and 'bottom'; (5) Components interact through three distinct patterns: synchronous bottom-to-top procedure call; (6) asynchronous top-to-bottom (notification) messaging; and asynchronous bottom-to-top (request) messaging; (7) Components may only make assumptions about the services provided above them, and may make no assumptions about the services provided below them.

By adhering to these constraints, Myx applications (including ArchStudio) receive certain benefits. Components remain relatively independent from one another, and it is easy to reuse components. Components only communicate through explicit interfaces, so it is easy to rewire components in different configurations without recoding the components themselves.

The Myx framework implements component interconnection and message passing protocols. Components and connectors used in Myx applications are subclasses from the appropriate abstract classes in the framework. This guarantees their interoperability, eliminates many repetitive programming tasks, such as connector implementations, and provides a basis for development of reusable components in Myx.

Specifically, Myx components and connectors are classes that implement an interface called *IMyxBrick*. Myx brick (component and connector) classes must minimally implement only two capabilities. First, Myx bricks must provide zero or more 'lifecycle providers.' A lifecycle provider is a class (possibly the brick's main class itself) that implements four lifecycle methods: *init()*, *begin()*, *end()*, and *destroy()*. These methods are called automatically by the framework as the bricks are created, attached, detached, and destroyed respectively. Second, Myx bricks must provide 'true objects' for provided interfaces, given the identifier of the provided interface. Recall that Myx bricks have explicit provided and required interfaces; these interfaces are associated with objects that implement these interfaces. For each provided interface, a Myx brick must (on demand of the framework) produce the object that implements that interface.

UniCon. UniCon [128] is an ADL for universal connector support, emphasizing the structural aspects of software architecture. Like other ADLs, architectures are modeled in UniCon as a configuration of components and connectors. Components are the locus of computation and state. Each component in UniCon has an interface specification that defines the component's type, and a list of association units – players, whose functionality is pretty much like that of component interfaces in other ADLs. Connectors are the locus of definition of relations among components. Each connector has a protocol specification that defines its connector type, a list of association units – roles, and the properties of roles. The connection process is then mainly about mapping the players of components with roles of connectors.

The novel part of UniCon is that the implementations of connectors in UniCon are all built-in, and could be reused in implementing different architectures. By this means, system developers could focus on the application-specific components, while the management of connectors is under control. The supported connector types include procedure call, data access, Unix-like pipes, remote procedure call, and real-time scheduling. Further specification or customization of each connector is achieved by providing values for certain attributes of a connector type, such as the communication algorithm used, and the maximal connection number allowed. The primary limitation of the UniCon implementation, however, is that built-in connector types are not extensible. This makes it hard for application designers who need a special connector type that is not supported. In addition, connector types in UniCon are chosen opportunistically and organized loosely. A better approach to do this could be through the use of a connector classification framework [91], where from general to specific connectors are organized into categories, types, dimensions, sub-dimensions, and values, based on the provided services and realization mechanisms.

2.2.3 Unified Representations

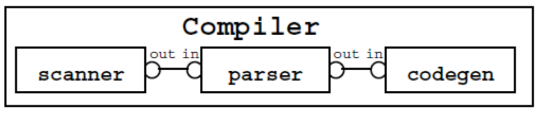
Approaches of unified representations seek to express and enforce structural or behavioral aspects of software architecture *within* source code, typically through the adoption of specially designed programming languages. They embed architecture constructs in a programming language, and rely on program compliers to check for architecture-code conformance. Examples include ArchJava and Archface, both of which develop new program elements to represent architecture constructs in source code. Their benefits are obvious: the co-evolution of software architecture and implementation. However, this also makes it hard to modify, extend, and reuse architecture and code independently given that they are mixed into a single artifact. In addition, approaches of unified representation face a common applicability issue: no other ADLs or programming languages can be supported.

ArchJava. ArchJava [3] is an extension to Java that unifies software architecture with implementation in one language. To allow programmers to describe software architecture, ArchJava adds to Java new language constructs to support components, connections, and ports. Figure 2-3 is excerpted from [3], and shows an example of ArchJava code that represents a graphical compiler architecture and its parser component. The *component*, *port*, and *connect* key words in the figure are defined by ArchJava to represent corresponding architecture elements. In addition, ArchJava enforces some specific rules to protect architecture-implementation conformance. For example, it requires that a component can only communicate with other components through explicitly declared ports – regular method calls between components are not allowed. This makes dependencies explicit, reducing coupling between components. Another rule specifies that each required method must be bound to a unique provided method. All these rules are enforced and checked by the ArchJava compilers.

```
public component class Parser {
  public port in {
    provides void setInfo(Token symbol,
                          SymTabEntry e);
    requires Token nextToken()
                   throws ScanException;
  }
  public port out {
    provides SymTabEntry getInfo(Token t);
    requires void compile(AST ast);
  }

  void parse(String file) {
    Token tok = in.nextToken();
    AST ast = parseFile(tok);
    out.compile(ast);
  }

  AST parseFile(Token lookahead) { ... }
  void setInfo(Token t, SymTabEntry e) {...}
  SymTabEntry getInfo(Token t) { ... }
  ...
}
```



```
public component class Compiler {
  private final Scanner scanner = ...;
  private final Parser parser = ...;
  private final CodeGen codegen = ...;

  connect scanner.out, parser.in;
  connect parser.out, codegen.in;

  public static void main(String args[]) {
    new Compiler().compile(args);
  }

  public void compile(String args[]) {
    // for each file in args do:
    ...parser.parse(file);...
  }
}
```

Figure 2-3: An ArchJava example excerpted from [3].

A primary advantage of ArchJava is that it protects *communication integrity* of the system under development. Namely, the implementation components only communicate directly with the components they are connected to in the architecture. This is based on the ArchJava programming rules described above, and essentially guarantees the structural conformance between software architecture and code. However, ArchJava cannot support behavioral conformance. In addition, the way that the architecture is implemented is limited to those predefined in the language. For example, inter-component connections can only be implemented with method calls in ArchJava.

Archface. Archface [148] 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 aspect weaving and round-trip engineering to maintain the architecture-implementation conformance. Both of these technologies face significant complexities as a program scales. It is not clear how this is tackled in Archface.

2.2.4 Roundtrip Engineering

The goal of software roundtrip engineering is to propagate updates made in derived artifacts back to their source artifacts [6, 21, 127]. Roundtrip engineering is usually used when the source model does not contain all information necessary to implement the complete system or the mapping of models to code is not as good as current day compilers. A typical example is architecture-centric software development that is introduced in next chapter. The mapping of architecture to implementations generates architecture-prescribed code. Since software architecture only contains principal design decisions of a system, generated programs are usually application fragments and skeletons with blanks for the developers to fill with implementation details. However, when the user has the possibility of changing the implementations, he can potentially also change parts of the architecture-prescribed code. Obviously this is a source of trouble, which is specifically addressed by roundtrip engineering. It is important to note that roundtrip engineering is not necessary for some model-driven development approaches, where full code generation is enforced and the source model is the only artifact that could be changed by application developers.

Roundtrip engineering in practice is often implemented by reverse engineering the modified source code to a high-level model, and replacing the previous version of the software model with the generated model. During this process, the information in the original software model is usually not considered, and is simply replaced with the new model. Strictly speaking, this is inconsistent with how roundtrip engineering is defined and the controversy about it still exists. Additionally, as discussed in Section 2.1.3, reverse engineering involves the activity of abstraction and is hard to fully automate. In particular, repeatedly doing it during the development of complex software systems could be expensive.

Fujaba (**F**rom **U**ML to **J**ava **A**nd **B**ack **A**gain). Fujaba [107] is a typical example of using reverse engineering to do roundtrip engineering. Software architecture in Fujaba is modeled as UML class diagrams that capture the system structure and so called *Story-Diagrams* that capture system dynamics. Story-Diagrams are a combination of UML activity diagrams and UML collaboration diagrams. Activity diagrams are used to specify the control flow and each activity contains either pure Java source code or a graph rewrite rule that is translated from a collaboration diagram.

A special feature of Fujaba is that it can reconstruct (i.e. reverse engineer) both structural models and behavioral models from the code that was manually changed, whereas most other systems can only reconstruct structure models. In particular, this is done through static analysis or parsing the modified code. In contrast, the construction of high-level behavioral model from source code is usually done through runtime monitoring discussed in Section 2.1.4. Graph transformation is involved during this process. In addition, Fujaba also requires that the program follow some pre-defined implementation patterns (e.g. naming conventions). A certain amount of annotations also have to be inserted and preserved in the source code. This, as well as the complexity of graph transformation, can be seen as a major limitation of Fujaba.

A recently developed approach of roundtrip engineering is through the application of software traceability [2], which is specifically discussed in Section 3.4. At this point, it is sufficient to know that software traceability concerns the relationships that exist among software artifacts created during development of a software system. Different from reverse engineering, the traceability-based roundtrip engineering process updates or reconciles, instead of replacing, the source models during the return trip. Following the trace links established between derived artifacts (e.g. source code) and the source artifacts, the system either warns the user that the changed artifact is generated from a high level model, or suggests further changes in the source models when manual changes to derived code occur. Automatic update of the source model, however, is not supported.

Finally, it is important to note that software traceability itself is still a research problem. It faces several critical challenges, such as creation, storage, and maintenance of traceability links during software development. This to some extent limits the further development of the traceability-based roundtrip engineering. In particular, the application of this approach in industry is rarely seen at this moment. This situation may improve as the technology of software traceability becomes increasingly mature.

2.3 Problems

Based on the discussion provided above, we believe that none of existing architecture-implementation mapping approaches is a complete solution that can be widely used during complex software development. Overall, correct-by-detection approaches require the program to be relatively complete or even executable in order for reverse engineering or runtime monitoring to be done for inconsistency detection. They are more appropriate for software maintenance, rather than software development. Moreover, prevention is always better than detection, especially considering that some inconsistencies may be expensive to detect and recover. From this perspective, correct-by-construction seems to be a good direction to go for conformance maintenance during development.

One-way mapping of correct-by-construction, however, faces the challenges of complete modeling and full code generation, and in practice can only be applied in some highly specialized domains with the help of DSLs and some other domain specific artifacts. Two-way mapping approaches of correct-by-construction, in contrast, are more practical given current modeling and code generation technologies. The problem is, most approaches in this category, such as architecture frameworks and current code generation approaches, are structure-oriented only and are limited in the mapping of changes between architecture and source code. Below we summarize and specifically discuss the problems of existing (especially two-way mapping) approaches. Resolution of these issues forms an important motivation of this research work.

- **Mapping architecture changes to code**. Maintaining architecture-implementation conformance is not a one-time thing since software architecture may be changed frequently during software development. A single round of code generation helps to improve software productivity, but is far from solving the conformance issue.

Complete code regeneration with primitive merge support (e.g. EMF's *JMerge*) is usually used when architecture changes occur after the first round of code generation. In the cases where user-defined code already exists and needs to be preserved during code regeneration, however, this method deteriorates quickly into a manual mapping. All these difficulties primarily come from the fact that current architecture implementation is usually done in an ad hoc way, and architecture-prescribed code is mixed with implementation details. Existing code separation mechanisms to some extent alleviate this problem. The challenge that they face is how to make separated code work seamlessly, especially when a portion of the code is regenerated. In addition, the programmers often have to figure out by themselves what was changed in the architecture.

- **Mapping code changes to architecture**. This is essentially an abstraction activity, and is hard to fully automate. In particular, the code-to-architecture mapping actually conflicts with the principle of architecture centrality. 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. Another promising approach to this problem is roundtrip engineering based on software traceability. However, the wide application of this technology is still pending on further development of software traceability. Moreover, automatic update of architecture when manual changes to generated code occur is still a critical challenge even with traceability links involved.

- **Support for the behavioral mapping**. Software architecture encompasses both structural and behavioral design decisions of the system under development. In contrast, most architecture-implementation mapping approaches are structure-oriented only. This is usually because architecture behavioral specifications (e.g. UML's sequence diagrams) are not complete enough to generate code from, and most existing code separation mechanisms do not support the separation of architecture-prescribed behavioral code from implementation details. Thus, the corresponding behavioral code is inevitably mixed with user-defined dynamic details. Protection of architecture-prescribed code becomes extremely difficult in this situation. Another challenge is that behavioral architecture specification could involve the interactions of several architecture components, and the corresponding implementation often cross-cuts the implementations of the involved components. Under this circumstance, code separation becomes even harder based on existing code separation mechanisms. The adoption of special modeling languages (e.g. Archface) may help at this point. However, as discussed earlier, these approaches rely on the design of special languages and are hard to be widely used.

3 Related Work

This chapter reviews a number of software development activities that are related to the process of architecture-implementation mapping, including architecture modeling, code generation, architecture-centric development, software traceability, and software change management. Of these different activities, architecture modeling and code generation provide some pragmatic techniques that are reused in our study so that we can focus on the consistency control of our approach; architecture-centric development actually represents an application context of architecture-implementation mapping; finally, software traceability and change management are two areas that are related to architecture-implementation mapping, even though corresponding techniques are not directly applied in this study.

3.1 Architecture Modeling

Software architecture modeling is an important portion of architecture-implementation mapping. Architecture models may be expressed using different modeling languages, and may express different aspects of the system (e.g. structure, behavior, etc.). Significantly, different definitions of software architecture still exist in the current literature. All these variations have an impact on the process of architecture-implementation mapping, and are specifically introduced in the following subsections.

3.1.1 Definition of Software Architecture

Software architecture was first defined by Perry and Wolf as a tuple of *Elements*, *Form*, and *Rationale* [120]. That is, software architecture is a set of architectural or design elements that have a particular form. Specifically, these *elements* could be processing elements, data elements, or connecting elements. The processing elements are those components that supply the transformation on the data elements; the data elements are those that contain the information that is used and transformed; the connecting elements (which at times may be either processing or data elements, or both) are the glue that holds the different pieces of the architecture together into architecture *form*. Finally, an underlying, but integral, part of an architecture is the *rationale* for the various choices made in defining an architecture. The rationale captures the motivation for the choice of architectural style, the choice of elements, and the form.

Shaw and Garlan defined software architecture modeling as a problem of designing and specifying the overall system structure that is beyond the algorithms and data structures of the computation [54]. These structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

Based on these early definitions of software architecture, a number of other definitions were presented in the following decade. For example, a later definition that has been widely adopted states that software architecture of a computing system is the structure or structures of the system, which comprise software components, the externally visible properties (assumptions that other components can make of a component: provided services, shared resources, etc.) of those components, and the relationships among them [26].

All these definitions do not conflict with each other. Instead, they are similar in that they all emphasize the structural perspective of software architecture (the so called 4 "C" model: **C**omponents, **C**onnectors, **C**onfiguration, and **C**onstraints). This to a great extent is based on the modeling technologies of software architecture at that time. For example, most early architecture description languages can only model the system structure. A primary benefit of these traditional definitions is that they are concrete and straightforward to follow. The limitation, however, is also significant. All these definitions of software architecture rely too much on, or are limited by, the development status of modeling technology at the time the definition was given. Thus, they may soon become out of date with the development of corresponding technologies (e.g. extensible architecture modeling). A typical example is Model-Driven Development, which also emphasizes the role of software architecture in the development, but cannot be explained by most traditional architecture definitions since its models usually involve extensive information beyond structure and do not have some well-recognized architecture constructs (e.g. connectors).

The definition we used in this research study defines a software system’s architecture as the set of principal design decisions about the system [143]. We believe this is an accurate characterization of software architecture that is not affected by the limitations of existing modeling technology. Different from the traditional definitions, this definition particularly emphasizes the extensibility of software architecture: different principal design decisions may be included by different sets of stakeholders for a system. It may look abstract given current modeling technologies, but it offers a universal explanation for all architecture-related activities, including MDD mentioned above. With the further development of architecture modeling technologies, especially extensible modeling languages, we believe this definition will be gradually adopted more widely.

Finally, it is important to note that we still follow the earlier definition of software architecture in our implementation of structural architecture specification. Namely, the architecture consists of a configuration of components and connectors. In this way, we can relate our technology to historic architecture models.

3.1.2 Architecture Description Languages

Architecture description languages (ADLs) overcome the informality of most box-and-line descriptions of software architecture, and provide notations and tools for precisely representing and analyzing architectural design decisions. Existing ADLs include Darwin [86], Rapide [85], Wright [4], Acme [57], AADL [47], UML [89], and xADL [32, 36]. Each of them provides certain distinctive capabilities, such as view support, dynamism, and analysis mechanism. A specific classification and comparison of these ADLs is provided in [87], and is not repeated here. Instead, we focus on the description of the xADL and UML languages in this section. Both of these languages are used in this research study with certain extensions or adaptations made.

xADL is an extensible XML-based ADL. The version used in this study is xADL 2.0. Extensible Markup Language (XML) [151] is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. A XML document forms a tree structure that consists of a set of nested elements, each of which is delimited by a start and end tag (a markup construct that begins with "<" and ends with ">"). A XML element may contain additional annotations called attributes. In addition, the XML schema language or XML Schema Definition (xsd) can be used to define the structure of an XML document, such as elements that can appear in a document, the order of elements, and default values of attributes.

Every xADL model is a *well-formed* and *valid* XML document. By well-formed, it means the document conforms to the basic structure of tags and attributes defined in XML. By valid, it means the document is consistent with the defined schema. In particular, xADL 2.0 is defined in a *modular* language design approach. Specifically, its notations are not defined in one large

XML schema block. Instead, xADL 2.0 is defined as a set of XML schemas, and the xADL 2.0 language is simply the composition of all the xADL schemas. Each xADL schema adds a set of features to the language, such as the ability to describe components and connectors, or the ability to indicate some particular elements in the architecture. Figure 3-1 is excerpted from [38] and shows the existing schemas of xADL 2.0.

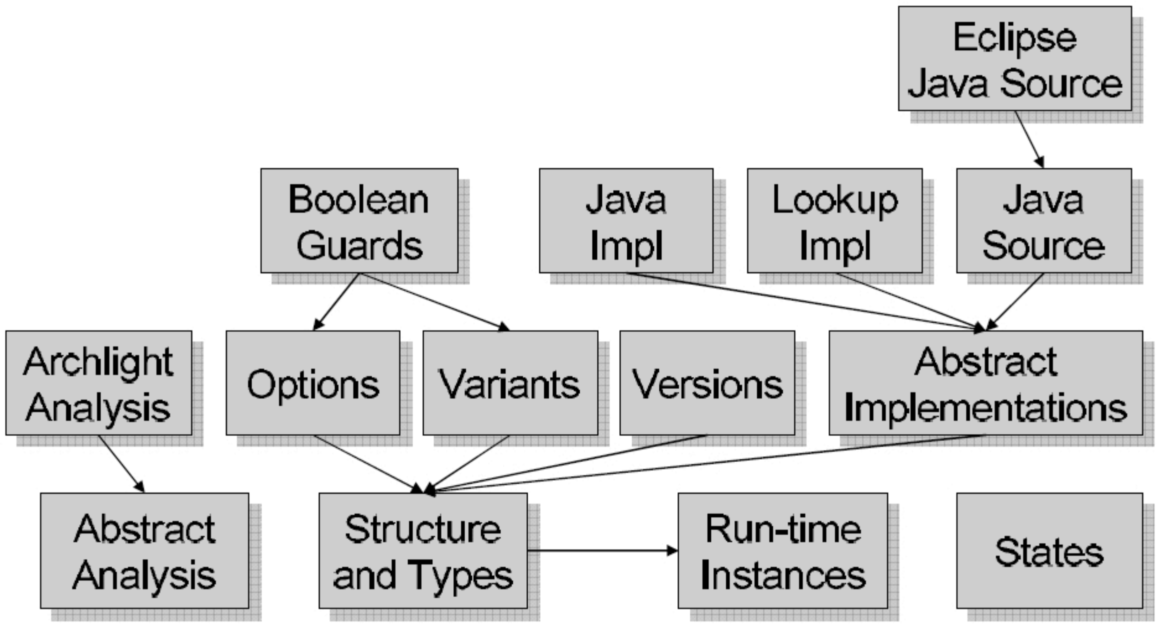


Figure 3-1: xADL schemas and dependencies

Among current xADL schemas shown in the figure, *Structure and Type*, *Instances*, and *Java Implementation* are the ones that are directly related with this study. *Structure and Type* defines basic structural modeling of prescriptive architectures: components, connectors, interfaces, links, as well as types for components, connectors, and interfaces. *Instances* defines basic structural modeling of description architectures: components, connectors, interfaces, and links. *Java Implementation* is related with architecture-implementation mapping. It defines a set of elements to map from structural architecture elements (i.e. components) to Java implementations. New schemas are still necessary for the investigation of this research study, for example, to model architecture changes and interactions among components. These are specifically discussed in Chapter 4.

Another significant advantage of xADL is its tool support [37]. In particular, there is a tool called *Apigen* that can automatically generate new data binding libraries (APIs for parsing, reading, and writing documents) for new xADL features or schemas. The data binding library provides an object-oriented interface to edit xADL documents. On top of it, xADL's tool suite includes a wrapper called "xArchADT", which can cast the object-oriented interface to a "flattened" interface that can be exposed over network-based middleware. Based on the data binding library and xArchADT, new tools can be built to support the exploration of those new features. This makes xADL an ideal language for investigating new architectural approaches and research directions.

Unified Modeling Language (UML) is a standardized modeling language that is mostly used the field of object-oriented software engineering. The standard is managed, and was created, by the Object Management Group (OMG). It was first added to the list of OMG adopted technologies in 1997, and has since become an industry standard for modeling software-intensive systems. Its latest released version is UML 2.3.

UML consists of a number of modeling diagrams, which can be roughly classified into two categories: structure diagrams and behavior diagrams [50]. A typical example of structure diagrams is the UML class diagram, which describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes. It is used in almost all object-oriented methods. A problem of class diagrams is that sometimes they contain too many details, given that they are relatively close to the source code. From this perspective, class diagrams often cannot provide real abstractions as most ADLs do.

Examples of behavior UML diagrams include use case diagrams, sequence diagrams, state diagrams, and activity diagrams. A use case is a set of scenarios (a sequence of interaction steps between a user and a system) tied together by a common user goal. A use case diagram, thus, depicts interactions between a user and a system, and represents an external view of the system. It is rarely used to model the inside working mechanism of a system. In contrast, a sequence diagram describes how groups of objects of a system collaborate in a single use case. Typically, a sequence diagram shows a number of involved objects and the messages that are passed between these objects within the use case. A state diagram describe all of the possible states that a particular object can enter and how the object's state changes as a result of events that reach the object. This usually spans several use cases. Finally, activity diagrams include some special modeling notations, such as branch, merge, and form, to represent parallel behaviors. They are primarily used in workflow modeling and multithreaded programming.

Current usages of UML are primarily in three modes: *UmlAsSketch*, *UmlAsBlueprint*, and *UmlAsProgrammingLanguage* [51]. In the *UmlAsSketch* mode, developers use the UML to help communicate some aspects of a system. The essence is selectivity. This mode is very popular. In contrast, *UmlAsBlueprint* is a UML mode that focuses on completeness. The goal is to express software designs in such a way that the designs can be handed off to a separate group to write the code, much as blueprints are used in building bridges. This mode is how we adopt UML and use it in our approach as described specifically in Section 4.5, except that the code is automatically generated from these UML models. The third mode raises the bar of UML even higher. It tries to use UML as a high level language by extending standard UML and providing executable semantics for it. A typical example is executable UML defined based on action semantics. This is mostly advocated by some MDD approaches introduced later in Section 3.3.1, such as Model-Driven Architecture (MDA). They try to make UML computation complete to achieve a high degree of formality and completeness for the platform independent model (PIM).

3.1.3 Modeling Aspects

An architecture model is an artifact that captures some or all of the design decisions that comprise a system's architecture in a machine-understandable ADL. It may represent different aspects of the system, including structure, behavior, and non-functional properties [80, 147]. Other modeling aspects also include domain variations used by domain-specific approaches and composition specifications for component-based development [67, 138]. In this research study, however, we only focus on the mapping of structural and behavioral architecture specifications to the code.

System structure is a basic aspect that can be captured by most architecture models. As mentioned earlier, existing ADLs model the architecture structure as a configuration of components and connectors with some constraints enforced. Important elements of a structural architecture model include:

- Components. Components are the loci of computation and state in the architecture. A component (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on is required execution context.
- Connectors. Connectors are the loci of communication in the architecture. A connector can be seen as a special component that is tasked with effecting and regulating interactions among components. From our perspective there is no

difference between components and connectors in terms of the mapping to source code. Implementing both as components is sufficient for this research study.

- Interfaces. Interfaces are components' portals to the outside world. There are *provided* and *required* interfaces. A provided interface expresses the services that a component provides, usually in the form of a list of functionally related operations. A required interface is the interface to services provided by other components in a system on which this component depends for its ability to perform its operations.
- Links. Links are connections between elements that define the topology of the architecture.
- Configuration. An architectural configuration is a set of specific associations between the components and connectors of a software system's architecture.

Examples of ADLs that primarily focus on the capture of architectural structure include Darwin, Wright, and UniCon. These languages tend to be semantically precise, but lack breadth and flexibility. They support similar architecture constructs, although mostly in different ways. For example, systems in Darwin are modeled as a set of interconnected components and there is no notion of explicit connectors in Darwin. However, a component that facilitates interactions could still be interpreted as a connector. Interfaces in Wright are simply specified using a notation derived from the Communication Sequential Processes (CSP) [4], which are introduced in next section. This gives Wright the ability to analyze constraints such as deadlock freedom.

A primary limitation of structural architecture models is that they cannot capture interactions among components or dynamic behaviors of a specific component. For example, the structural designs only declare some operations expected or provided in the interface of the participating components, but do not capture the control flow of several small methods scattered around the architectural configuration. Thus, the control flow among these operations is usually lost when mapped to code. This causes the clarity or the underlying rationale of the designs to get lost in the code [94]. In general, a single method only make sense in a larger context and is difficult to be reused independently.

In contrast, a behavior model of a software system captures interactions among system elements or between the system and its external user, the order in which they can be executed, and maybe other aspects of this execution such as timing and concurrency. In essence, the primary challenge of behavioral modeling is that a static method has to be used to describe dynamic aspects of the system. Existing behavioral modeling methods can be classified into those that are based on formal notations and those that are for practical use. Each has its own range of applications, and important limitations as well. Generally, formal methods are expensive and complicated for normal use. In most cases, people would rather write code directly in implementations. Practical behavior models like UML diagrams are informal and still semantically incomplete. It is hard to generate complete code from them without any significant extensions made.

Formal behavioral modeling methods include the use of process algebra [96], Petri nets [101], Actor model [1], and Z notations [134]. Automatic analysis is one of their primary purposes [74]. Process algebras, Petri nets, and the Actor model particularly focus on modeling concurrent computations. Specifically, process algebra is the study of the behavior of parallel or distributed systems by algebraic means. "Process" refers to behavior of a system, and can be simply seen as a series of events, the basic units of a behavior model. Processes are then described by combining events and other simpler processes through a set of pre-defined composition operators. Significantly, the operations must satisfy a set of axioms or laws, based on which advanced analysis and verification can be done. Examples of process algebras include Calculus of Communicating Systems (CCS) [95], Communicating Sequential Processes (CSP) [70], and Pi-calculus [97]. Pi-calculus is an extension to CCS, and addresses the issue of dynamic reconfiguration in a distributed system. One of its important extensions is treating links as ordinary variables (called names) that can be passed among agents. In this way, links between agents can be created and destroyed dynamically. A Petri nets consists of places, transitions, and directed arcs. By changing the meanings of these elements, Petri nets can be used to model control flow, data flow, and state machines of a system. Its major weakness, however, is that Petri net-based models often tend to become too large for analysis even for a modest-size system. Finally, Z is a formal notation that is based on set theory and first-order logic. The Z language focuses on data and its transformations. Systems are specified as sets of schemas. The Z schemas, which can be regarded as generalized type definitions, are used to represent basic constructs. These schemas provide semantics that permit the formal verification of properties of the model. Additional details on Z can be found in [134].

Behavior modeling of software architecture mostly reuses or is based upon the work described above. Wright and Darwin are two ADLs that use existing formal notations to model system behaviors. Wright uses CSP, and Darwin uses FSP (another instance of process algebra). With supportive tool built (e.g. LTSA), models specified in these languages can be analyzed to verify if the system design satisfies some predefined properties, such as deadlock freedom, liveness, and so on. In essence, what Wright and Darwin do is describe the system structure with their own ADL constructs, and use structural descriptions as a framework for behavioral specification. They suffer from the same limitations as those formal notations, high complexity. The architecture analysis and design language (AADL) is an ADL that provides a more practical way to model system behaviors. It has two specially designed elements: *Call Sequences* and *Modes*. Call Sequences describes the interactions between or within components, and Modes represent operational states of a system or a component. Functionally speaking, these two language elements are very similar to interaction and state diagram of UML. From this perspective, we think AADL actually takes a simple, but practical way for behavior description. The risk, as described above, is how to successfully map these informal specifications into code.

3.2 Code Generation

Similar to structural modeling, structural code generation is well understood and not a particular research issue. Architecture-implementation mapping brings a new challenge in this regard, however, which requires both structural code and behavioral code, to be automatically generated from source models. This is hard not only because non-structural modeling is not yet mature, as introduced in previous section, but also because system dynamics are involved and many more variations need to be considered compared with static structural code generation.

Figure 3-2 shows existing code generation approaches and how they treat source code differently. Code can be treated as a model, program, or plain text. Approaches that treat code as model require the definition of a metamodel [9] for the target programming language, and use model transformation approaches [31] for code generation. A typical example is Eclipse's ATL project [45]. It remains to be seen how well these approaches can be practically used in complex software development, especially considering the high complexity that is often involved in model transformation. Approaches that treat code as program are trying to use the target programming language's own metaprogramming ability, e.g. reflection, for code generation [14]. They are limited in the sense that they can only be used to generate structural constructs like classes, methods, and attributes.

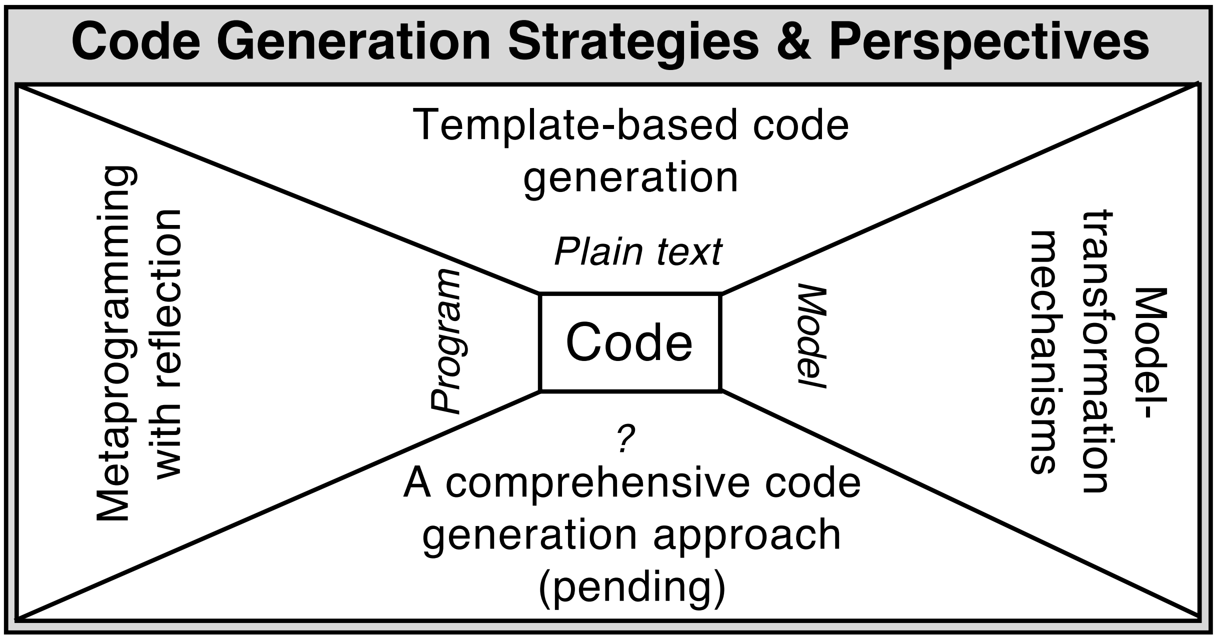

Code Generation Strategies & Perspectives
Template-based code generation
Plain text
Metaprogramming with reflection
Program
Code
Model
Model-transformation mechanisms
?
A comprehensive code generation approach (pending)

Figure 3-2: Different code generation strategies see code differently.

Treating code as plain text, or template-based code generation represents a popular approach [25, 69]. A typical example is Java Server Pages (JSP) that are used to create web pages, where the Java escapes are executed to produce the dynamic portions of the HTML page. A primary advantage of the template-based approach is that templates are independent of the target language. This simplifies the generation of any textual artifacts, including documentation. A primary challenge, however, is verifying the correctness of code embedded in templates that are usually not runnable. Thus, a comprehensive code generation approach that can work as well as a program compiler is still missing. Further development in this area may be pending on a new perspective.

User control, the amount of work that is required from the user, is another important discriminator between various approaches to code generation. There are two extremes. The simplest implementation is one that makes the user responsible for every single generation step.

Such an approach is only practical if the system is of small scale and contains compact mapping steps. Fully automatic systems, on the other hand, hide everything in the code generation process from users by using built-in heuristics to evaluate different mapping possibilities. Such systems, however, work satisfactorily only for restricted domains of application [10]. Against this background, it is desirable if a code generation process is interactive. For example, users may be responsible for making decisions like selection of appropriate transformation rules, while the system automatically applies the selected rule and records users' selections for the purpose of replay. At this point, how to find a balance between user and system control is an important issue that needs careful consideration.

Granularity measures the size of software entities used as the construction unit in the code generation process. From fine to coarse, variations of granularity include programming language constructs, code fragments and skeletons, components, and large-scale domain-specific subroutines [41]. In general, increasing granularity can not only improve software reusability [82], but also contract the implementation space given that reusable constructs usually encapsulate certain implementation decisions from the external. At this point, programming language constructs, such as variables and arrays, provide very little support since it is essentially generating code from scratch. In contrast, large-scale subroutines are much more useful, although limited to some domain-specific approaches like application generators. A typical example of code fragments and skeletons in architecture-implementation mapping is an architecture framework, which specifies key elements of an architecture style in the form of source code. Finally, software components are mostly used in "component-based" software development, where glue code is generated from composition specifications to combine different large-grain components into one application.

3.3 Architecture-Centric Software Development

A system's software architecture is the set of principal design decisions made about it. Architecture-centric development emphasizes that software architecture, instead of being a documentation artifact that is peripheral to code development, should play an essential role throughout the software development lifecycle. Specifically, it requires that all the architecture-related changes start right from the architecture, and be mapped to code through an architecture-implementation mapping tool. Examples of architecture-centric development include model-driven development (MDD) and architecture-based research. A significant difference between these approaches is the modeling notation used and the amount of modeling versus programming in software development [156]. Therefore, different strategies are often taken to maintain architecture-implementation conformance [126].

3.3.1 Model-Driven Development

MDD suggests a paradigm where software design models take the role of traditional programs, and become the main artifact of development. UML and domain specific languages (DSLs) are the main modeling notations of MDD. Code generators are extensively used in MDD to generate code from design models. Based on the amount of generated code, approaches of MDD are divided into two camps [29, 63, 135], which we refer to as *MDD in theory* and *MDD in practice*. MDD in theory aims to make design models compilable and executable, so that software developers can solely focus on abstract models. To achieve the goal, it emphasizes full code generation. Initiatives in this camp include Model-Driven Architecture (MDA) [81], Model-Integrated Computing (MIC) [136], and Software Factories [61]. They are different in various ways, some are generic, others domain specific. However, they all face the same challenge that was discussed in Chapter 2 when it comes to full code generation. Figure 3-3 provides a simplified illustration of these approaches with important artifacts and code generation processes explicitly represented. MDD in practice, in contrast, recognizes the essential role of both design models and implementation. Its generated code is an application skeleton that requires software developers to fill in details. A typical approach of this camp is the Eclipse Modeling Framework (EMF) that was introduced in Chapter 2.

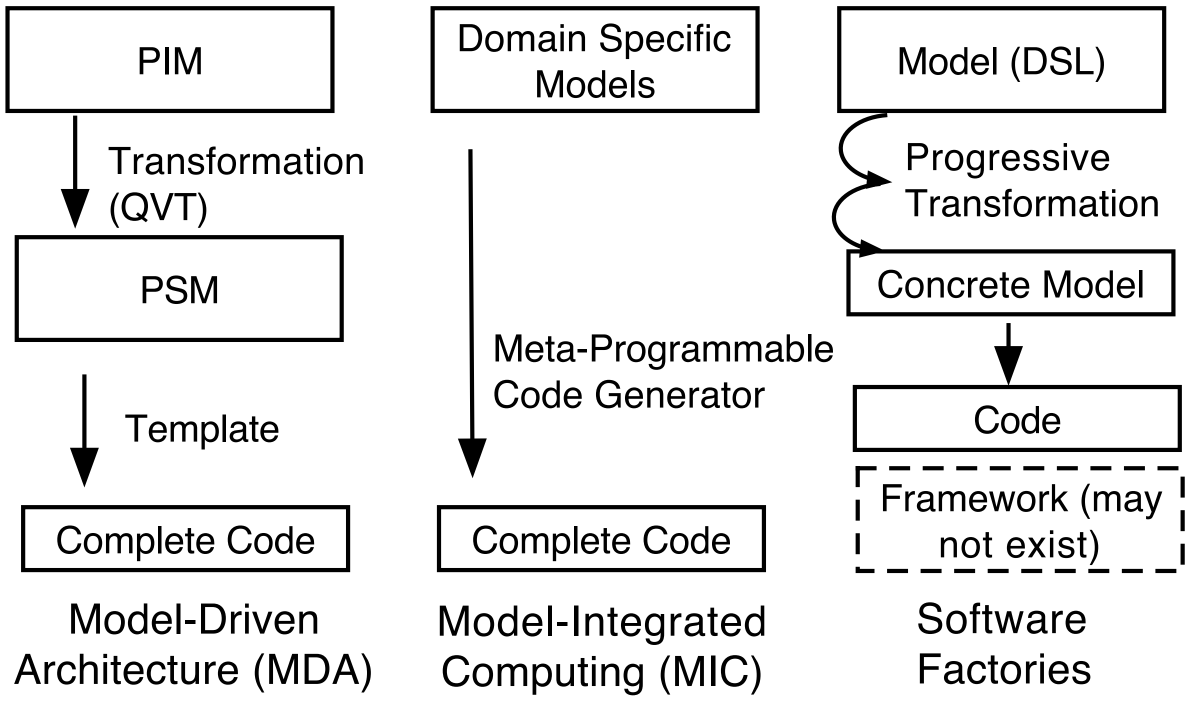

PIM
Transformation (QVT)
PSM
Template
Complete Code
Model-Driven Architecture (MDA)
Domain Specific Models
Meta-Programmable Code Generator
Complete Code
Model-Integrated Computing (MIC)
Model (DSL)
Progressive Transformation
Concrete Model
Code
Framework (may not exist)
Software Factories

Figure 3-3: Model-driven development.

MDA is a conceptual framework in support of model-driven development, defined by the Object Management Group (OMG) in late 2001. The term "architecture" in MDA is used because MDA prescribes certain kinds of models, how those models may be prepared, and the relationships of the different kinds of models. Specifically, software development in the MDA starts with a Platform-Independent Model (PIM) of an application's business functionality and behavior, constructed using Unified Modeling Language (UML) based on OMG's MetaObject

Facility (MOF). This model remains stable as technology evolves, extending and thereby maximizing software reusability. MDA development tools, available now from many vendors, convert the PIM first to a Platform-Specific Model (PSM) and then to a working implementation on virtually any middleware platform [13]: Web Services, XML/SOAP, EJB, C#/.Net, OMG's own CORBA, or others.

One of key challenges that MDA faces is transforming the high-level PIM models to PSMs that tools can use to generate code. Research on model transformations is still immature and there is little experience. The OMG tries to solve the problem by proposing a new standard, Query/View/Transformation (QVT), to address the way transformations are achieved between models whose languages are defined using the MOF. It contains a language for creating views of a model, a language for querying the model, and a language for writing transformation definitions. Some desirable features of transformations in MDA include traceability, incremental changes, and roundtrip engineering.

MIC and Software Factories are both domain-specific approaches. MIC was originally designed for embedded software development. MIC advocates the application of different types of models written in domain-specific modeling languages (DSMLs), and manages the interdependency among models at the meta-level [78]. In particular, MIC develops a meta-programmable generic modeling environment (GME) that allows the creation of models that comply with the static semantics defined in the corresponding metamodel [83].

Like other model-based approaches, an important task for the MIC technology is the generation of embedded systems from domain models. A special property about MIC at this point is that embedded systems frequently consist of many physical and software components that are customizable and reusable in different systems. Thus, the role of the model-based generators in the MIC framework is just to generate the "glue" required to compose the integrated application from library components, consistently parameterize the components, and customize the composition platform.

The primary goal of Software Factories is to industrialize software development, and improves software productivity and predictability. The concept is the confluence of model-driven development, component-based development, and software product lines. These technologies respectively represent three dimensions that software factories are trying to integrate: abstraction, granularity, and specificity. Generally speaking, software factories increase the abstraction level by focusing on high-level software models in application development, improve the granularity of abstractions by increasing the size of the software constructs, and promote the usability or value of abstractions by increasing their specificity to some problem domain.

Framework completion and progressive refinement are two cooperative approaches used in Software Factories to generate an executable from requirements. Specifically, framework completion is the approach where a software framework that specifically addresses a well-defined, narrow problem domain is provided, and the abstractions in a model are used to define how the variability points in the framework must be completed. In the code generation process, only minimal code needs to be generated to fill variability points in a domain-specific framework from a domain-specific model. If it is not possible to build a software framework that can provide a natural platform for implementing a useful DSL, it may be necessary to define another layer of simplifying abstractions into which the first set may be mapped. This second set of abstractions may be easier to implement than the first. The abstractions are then transformed into an executable by a series of steps. This process is called progressive transformations. It is important to note that the transformation process is inherently parameterized, and operates by binding objects in source models as parameter values, and creating or modifying objects in target models.

3.3.2 Architecture-based Research

Architecture-based research specifically refers to the work done in the research community of software architecture, where architecture description languages (ADLs) are usually used as modeling notations. Figure 3-4 shows the architecture-implementation mapping approaches that are mostly used in architecture-based research. All of them have been specifically discussed in Chapter 2. In this section, instead, we focus on two architecture-based development activities, architecture-based self-adaptation and product line architectures. Both of them involve the process of architecture-implementation mapping as discussed below.

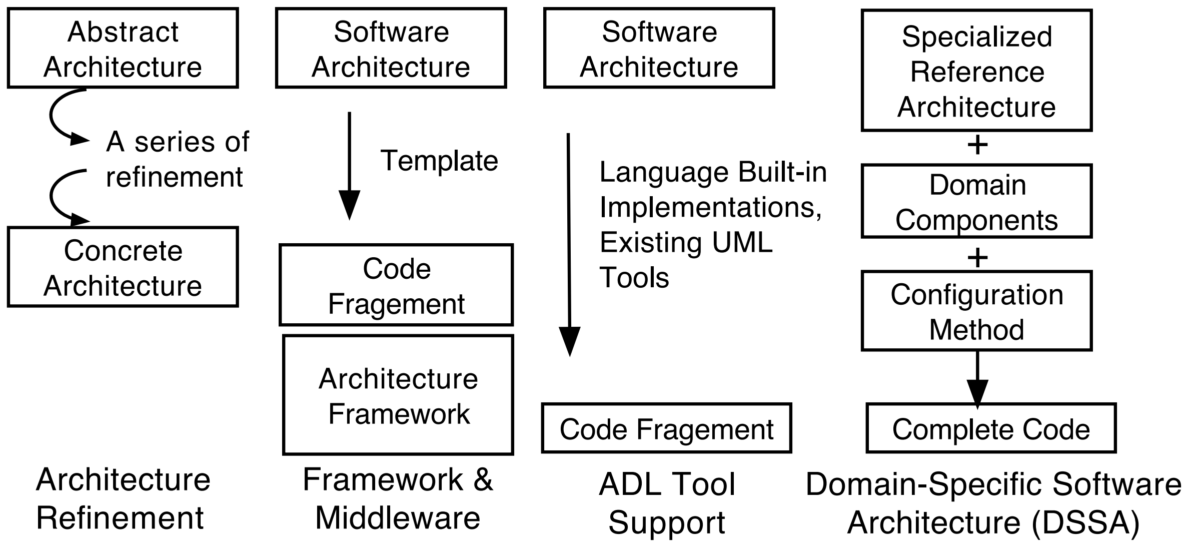


Figure 3-4: Architecture-based research.

Dynamic adaptation refers to the capability of a self-adaptive software system that can modify its own behavior in response to changes in its operating environment (e.g. end-user input, external sensors, etc.). Architecture-based adaptation [113, 115, 154] brings promising results in this regard. This is an approach where changes are first formulated in, and reasoned over, an explicit architectural model when environment changes. Changes to the architectural model (usually at the level of components and connectors) are reflected in modifications to the application's implementation, while ensuring that the model and the implementation are consistent with one another. It is at this point that an architecture-implementation mapping approach has the potential to play a significant role by dynamically mapping architecture changes to code.

An architecture-based infrastructure is described in [112] to support software self-adaptability. Specifically, it separates adaptation activities into two simultaneous processes: adaptation management and evolution management. Adaptation management monitors and evaluates the application and its operating environment, plans adaptation, and deploys change descriptions in architecture terms to the running application. In contrast, evolution management is responsible for actually evolving software and maintaining the consistency between architecture and implementation. A primary contribution of this approach is the development of a comprehensive methodology that integrates different technologies in support of the range of adaptations. Significantly, the inherent properties of software architecture, such as separation of concerns and abstracting away obscuring details, determines that it is the right abstraction level for managing software evolution.

An integral part of the adaptation infrastructure described above is maintaining the consistency between the architectural model and the implementation as changes are applied. Dynamic adaptation exposes additional challenges, such as protecting integrity of adapted systems, and identification of quiescent states when adaptation can safely occur.

Keeping the cost of software changes low is another important requirement in software evolution. This is especially emphasized when making changes that are anticipated before system development starts, so called *anticipated changes* [119]. In contrast, changes that are discussed earlier in the dissertation can be seen as *unanticipated changes*, which could result from requirement changes, system refactoring, or development incidents. Anticipated changes usually occur when developing a family of software products, or a product line. For example, producing a new software product simply by extending a related existing product (e.g. adding an optional capability, customizing for different platforms). At this point, being able to reuse existing code that encapsulates domain, business, and technology information as much as possible becomes very important.

The use of product lines has gradually become a principled form of software reuse over the past decade. This is partially due to the application of product line architectures (PLAs) [68], an architecture-centric approach to product lines. A PLA explicitly specifies variation points (e.g. optional and alternative elements) inside the reference architecture of an entire product line to differentiate products. Implementing a PLA is also a mapping problem, except that multiple products composed of core elements and variation points are involved. During this process, it is important that separation of concerns can be achieved among the different component implementations as it is in the architecture. Otherwise extensive changes have to be made to the code of existing components to introduce variations, and software reusability is compromised. However, separating concerns in the implementation artifacts along preferred boundaries involves significant challenges, especially for those crosscutting concerns that are spread over the system [102].

3.4 Software Traceability

Software traceability represents relationships that exist among software artifacts created during development of software system [7, 133]. It was originally applied in the area of requirements engineering to assess the drift between the software product's actual behavior and the original requirements specified by the customer. The field of software traceability then has grown to accommodate other types of artifact relationships across the software lifecycle with the goal of enhancing software product quality. In particular, software traceability facilitates the important software development tasks of system comprehension, change impact analysis, system debugging, as well as roundtrip engineering, as described in Section 2.2.4.

It is important to highlight that software traceability and conformance management are two related, but different areas. This is especially important when it goes to architecture-implementation mapping. In other words, having correct traceability links established between architecture elements and source code does not necessarily mean that the architecture and code are conformant. For example, an architecture element (e.g. component) may be incorrectly implemented in the linked code element (e.g. class). In this case, the established traceability link is still valid though the architecture and code is not consistent.

Software traceability currently faces some critical challenges, such as automatic creation and maintenance of traceability links, the storage of captured links, and link semantics [2]. Important research progress has already been made in corresponding areas. However, the application of software traceability in practice is still limited. This is primarily due to the overhead of creating and maintaining traceability links, given that many software artifacts (e.g. requirements specification, design, code, test cases, etc.) may exist during software development and each is often under constant change.

ArchTrace [104, 105] is a tool to support the evolution of traceability links between architecture descriptions and corresponding source code. ArchTrace uses a policy-based approach, where different policies specify different actions to take, or constraints that must be satisfied, upon the evolution of either software architecture or source code configuration items. The execution of one policy can result in the triggering of one or more other policies. For example, a policy may define the addition of new traceability links when new versions of source files are available, and another policy may suggest the removal of old traceability links after the execution of the first policy. The effectiveness of ArchTrace is evaluated by replaying the past check-in, check-out data from a real development project, and comparing the set of ideal traceability links with the set of actual traceability links produced by ArchTrace. The result is positive given the experimental settings. In general, ArchTrace is a good complement to the architecture-implementation mappings introduced in this dissertation in the sense that it maintains valid architecture-to-implementation traceability links, based on which architecture change notifications presented in Section 4.4.3 could be fulfilled to help keep architecture descriptions and implementations consistent.

3.5 Software Change Management

Generally speaking, software change management includes many different activities, such as change impact analysis, configuration management, and even regression testing. In this study, however, our focus on change management is limited to the issues of change mappings, change notifications, and change control. We do not examine the problems of version control, building baselines, etc. Existing work in this limited area are reviewed and compared to change management of 1.x-way mapping in this section.

ArchEvol [108] was originally designed as an integration of Eclipse, ArchStudio, and Subversion. It addressed the evolution of the relationships between versions of the architecture and versions of the implementation through the interaction of these three tools. Recent work on it [110] has upgraded ArchEvol to an Eclipse-based development environment that supports concern-driven software development. It maintains an explicit concern model that consists of a hierarchy of concerns (rationales of development decisions) and the links to the code fragments that implement corresponding concerns. Based on the model, concerns can be visualized at both the code and the architecture level. In addition, heuristic and manual techniques are also developed to maintain the concern mapping over time. ArchEvol thus represents a good compliment to 1.x-way mapping, which only focuses on structure and behavior architecture-implementation mapping.

Lighthouse [130] is an Eclipse plug-in built to support the coordination of multiple developers. It develops a new concept called emerging design, an up-to-date representation of the design that is extracted from the developers' code. Basically, Lighthouse collects code changes from each developer (by monitoring their workspace), and presents the emerging design as a diagram that is annotated with additional information about ongoing changes, such as which developer is making what kind of changes to which element. Various filters are also built in Lighthouse to reduce the number of elements shown in the emerging design, so that it can be scaled for use in large software development. Lighthouse represents an efficient change notification mechanism. It is limited in terms of change mapping, however, because developers still have to manually respond to each related change. In addition, privacy may be an issue that endangers its wide use, given that each developer workspace is monitored and transparent to others.

The *CHS* tool maps change-based product line architectures (PLA) to code in a software configuration management (SCM) system [84]. CHS supports the activity of generating a directory structure and skeleton code in the SCM system. In particular, it requires that generated code not be modified by the developers. However, no specific explanations are given regarding to how generated and non-generated code should be separated and integrated, and how changes should be handled differently in both parts. This is primarily because the focus of CHS is on the adoption of a change-based SCM system to map changed-based modeling of a PLA, rather than providing a complete solution to architecture-implementation conformance management. The idea of protecting generated code from being manually modified can also be found in some other research work [17, 18, 92]. As introduced previously, most use so-called spatial separation or shallow separation for this purpose, which is not sufficient for architecture-implementation conformance management.

4 Approach

This chapter presents a new architecture-implementation mapping approach, *1.x-way mapping*. It begins with an introduction of basic design principles and underlying insights of the approach. After that, an overview of 1.x-way mapping is given, and its four core mechanisms are specifically introduced: a code separation mechanism, an architecture change model, architecture-based code regeneration, and architecture change notification. Support for behavioral mapping and other related issues, such as prevention of programmer-induced negative properties are also discussed. At the end of the chapter, a comparison framework is presented to highlight the differences between 1.x-way mapping and the existing mapping approaches described in Chapter 2.

4.1 Design Principles

As stated in Section 1.3, the hypothesis of this research study is that 1.x-way mapping can be applied in the development of a realistic system to prevent its architecture-prescribed code from being changed by programmers, and support automatic mapping of structural and behavioral architecture changes to code. Strictly following this goal, the core design principles of 1.x-way mapping are summarized as follows:

- The implementation of software architecture and changes to it should be regulated. The freedom of implementing architecture and the effort of maintaining architecture-implementation mapping can be seen as a tradeoff to make. The difficulties that traditional software development faces in this regard comes from the fact that architecture is often implemented in ad hoc ways, and architecture-prescribed code is mixed with implementation details. As a result, it is hard for programmers to know either how or when to update architecture when code changes occur. In contrast, approaches like architecture frameworks, EMF, and ArchJava described in Chapter 2 address the issue by regulating the implementation of the architecture through the use of pre-defined code, code separation, or special programming language constructs. Correspondingly, it is relatively easier for them to maintain the architecture-implementation conformance, even though problems still exist as discussed earlier in Section 2.3.

- The best way to map code changes to architecture is to avoid the need for such reverse mapping by protecting generated code from manual modification. As mentioned earlier, the code-to-architecture mapping itself is essentially a problem of abstraction [125], and is hard to be fully automated. Moreover, it conflicts with the principle of architecture centrality in software development, which requires the all architecture related changes should start from the architecture and be mapped to code afterwards. Therefore, we believe a good way to address the difficulty of code-to-architecture mapping is simply to avoid it. This can be done through the regulation of architecture implementation discussed above, for example, by using code separation.
- Architecture-prescribed code should be generated, and updated solely through code generation. Information duplication has been identified as an important cause of inconsistency. In the context of architecture-implementation mapping, this means that the same information exists or is represented in both architecture and source code. The best way to address the problem of information duplication is to follow the principle of *Don't Repeat Yourself* (DRY): every piece of knowledge must have a single, unambiguous, authoritative representation within a system [72]. Code

generation can be used in this process to automatically update representations of the same information in different artifacts.

- Architecture changes should be modeled and manipulated as an independent software artifact. Architecture changes play an important role in architecture-implementation mapping, especially if we want to reduce the impact of code regeneration on non-related code, and notify programmers about the changes that were just made. Explicit modeling of architecture changes not only meets these demands, but also opens up the opportunities for more advanced development activities, such as concurrent architecture changes, and change replay. Moreover, the fact that software architecture is located at a relatively high abstraction level and contains fewer constructs compared with software programs also makes architecture change modeling a possibility [155].
- Only "*executions of significance*" should be modeled in behavioral architecture. Software architecture encompasses *principal* design decisions about a software system, and it should not be expected to be a complete model of the system. This is especially the case when it goes to system dynamics, which could contain overwhelming details that, if represented in the architecture, would greatly degrade the usability of software architecture. Thus, it is advocated in 1.x-way mapping that only executions that are significant enough to be visible at the architecture level, or "executions of significance", should be modeled and mapped to code. Note that the definition of *significance* is up to stakeholders or software architect to decide. Same kind of system behavior may be seen of different importance in the development of different systems. From this perspective, the term "significance" is actually as

subjective as the term “principal” in the definition of software architecture discussed in Section 3.1.1.

4.2 Overview

In this section, we present an overview of 1.x-way mapping. The name comes from the fact that 1.x-way mapping 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. 1.x-way mapping consists of four core mechanisms: deep separation, an architecture change model, architecture-based code regeneration, and architecture change notification. Figure 4-1 shows an overview of 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. Note that the amount of behavior modeling does not affect the effectiveness of 1.x-way mapping. In other words, 1.x-way mapping can also support situations where extensive behavior modeling is involved, although this is not recommended as discussed earlier in this chapter. The architecture modeling notation used in 1.x-way mapping is xADL 2.0, an extensible, XML-based architecture description language. Java is used as the programming language in this exposition. It is assumed that all the development activities shown in the figure take place in an integrated software development environment (IDE) [76, 144, 145], 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 ArchStudio 4, an Eclipse-based tool integration environment where our work is performed.

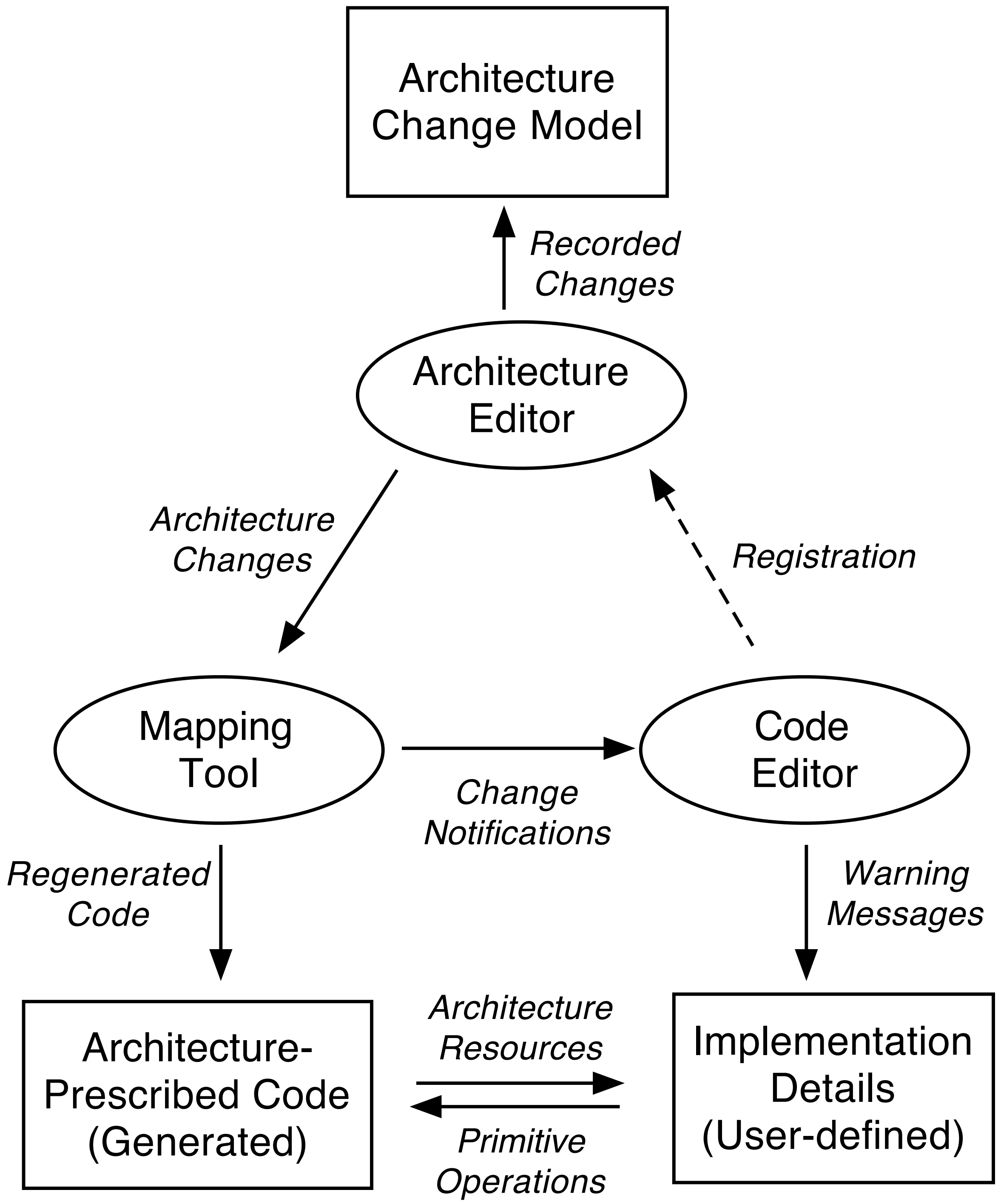


Figure 4-1: An overview of 1.x-way architecture-implementation mapping.

As shown in the figure, the implementation of each architecture component is separated into two independent program elements: architecture-prescribed code and implementation details. The former 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. The latter represents the internal implementation of a component that is to be manually developed by programmers.

On top of the separated code, 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, based on the fact that all the information about the kinds of changes (e.g. *componentChanges*, *linkChanges*, etc.) is recorded in the change model. For those architecture changes that may require modifications to 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 messages, a plug-in could be built to allow programmers to register for particular kinds of architecture changes. This part of the work is future work, which is why it is represented by a dashed line in the figure.

1.x-way mapping consists of four core mechanisms: a deep separation mechanism, an architecture change model, architecture-based code regeneration, and architecture change notification. Each of them is introduced in the following subsections. Support for behavioral mapping, and prevention of programmer-induced negative properties are also discussed.

4.3 Code Separation and Integration

The 1.x-way mapping approach exploits a new code separation mechanism to decouple architecture-prescribed code and user-defined details of each architecture component. The separated code is explicitly integrated by a program composition mechanism (e.g. method calls, software frameworks), which not only supports the integration of separated behavioral code, but also enables mutual independence of separated code. This is essentially different from existing code separation approaches, such as filling-in-blanks and subclassing.

4.3.1 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 (e.g. method calls) to explicitly integrate separated code. Specifically, the user-defined code of a component implements 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 described in Section 2.2.1. Those approaches are called *shallow separation* or *spatial separation* in this work, because their code is physically separated, but is still coupled and implicitly integrated by some inherent language relationship (same class, inheritance, etc.).

Architecture-prescribed code of a component comprises the implementation of all the prescribed information about the component, including its identity, provided and required interfaces, properties, and behavior definitions. It includes knowledge about architecture topology and message exchange among components. In particular, architecture-prescribed code of a component in 1.x-way mapping includes the implementation of a set of operations that are to be provided by the component, and each of them is simply implemented by redirecting requests to corresponding user-defined code of the component. Figure 4-2 shows a set of rules defined in 1.x-way mapping to specify how the architecture-prescribed code of a component should be generated. Notice that these rules are not intended to be exhaustive, given that extensibility is an essential feature of software architecture, and additional rules may have to be defined when new elements are added. In addition, all of them are based on the assumption that Java is used to implement the architecture. With a different programming language used, the rules should be modified correspondingly.

Rule #1: A class is generated for each architecture component as the architecture-prescribed code. The class name by default is the component identity suffixed with "Arch".

Rule #2: The generated class implements all the provided interfaces declared by the corresponding component. Each method in a provided interface, unless defined by a sequence diagram, is implemented by redirecting the request to user-defined code.

Rule #3: Each required interface is implemented as an attribute of that interface type in the generated class.

Rule #4: The architecture-prescribed code of a component maintains an explicit reference to the corresponding user-defined code, and initializes the reference by calling the *setArch()* method in its constructor method.

Figure 4-2: Rules of deep separation for architecture-prescribed code.

User-defined code provides primitive operations that architecture-prescribed code uses to construct higher-level operations. It contains implementation details that are not specified in the architecture, such as how a specific algorithm should be implemented, which system library to use, domain-specific code reuse, or the application of an implementation technology such as web services, CORBA, Java RMI, etc. All these details are to be manually completed by programmers. Compared with architecture-prescribed code, which essentially codifies externally visible characteristics of a component, user-defined code represents the internal implementation of a component. It does not communicate directly with other connected components, and is completely hidden from the externals in 1.x-way mapping. Figure 4-3 lists the rules of deep separation for user-defined code.

Rule #1: The user-defined code of a component implements all the operations requested by the corresponding architecture-prescribed code. In case an interface (e.g. *required operations*) is generated for this purpose, the user-defined code should implement all the methods in the interface.

Rule #2: The user-defined code of a component must maintain an explicit reference to its architecture-prescribed code, and initialize the reference through its provided operation, *setArch*.

Figure 4-3: Rules of deep separation for user-defined code.

A calculator application is used as an example in this dissertation to illustrate how 1.x-way mapping works. Its structural architecture is shown in Figure 4-4. This is not a complex application. However, it provides concrete situations in which automatically maintaining the architecture-implementation conformance is difficult. In short, the calculator works as follows. The *GUI* component is responsible for collecting the 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 its value is changed, *GUI* is notified and updates its display field correspondingly.

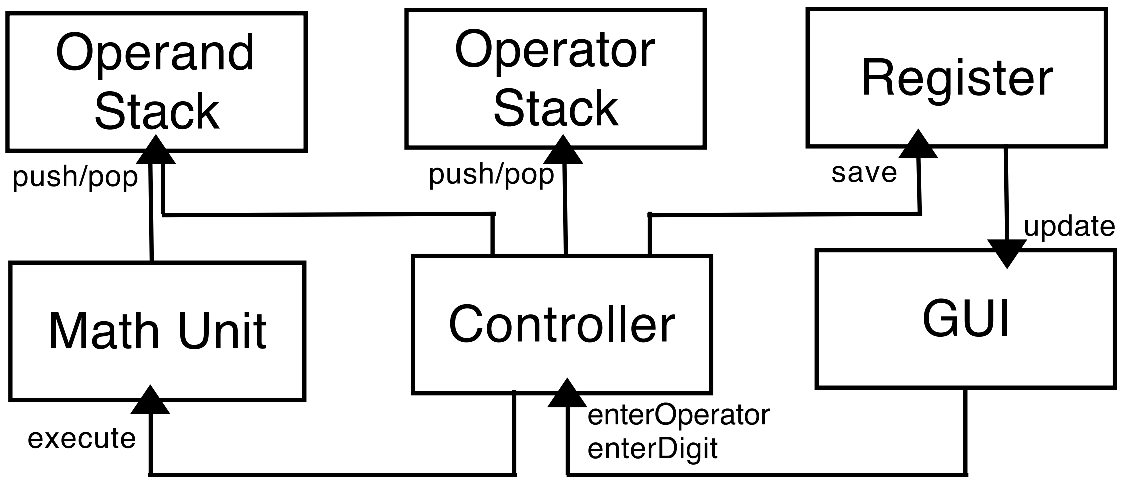


Figure 4-4: Structural architecture of the calculator application.

List 4-1 shows the implementation of the *Controller* component with deep separation enforced. Other components in Figure 4-4 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 (lines 03-06) 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 12 and 15). This goes through a reference (_imp, line 02) 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 19), 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 architectural information to *ControllerImp* (line 09). What it also implies is the existence of a variable (*_arch*, line 24) of the type *ControllerArch*, or architecture-prescribed code, in the corresponding implementation *ControllerImp*, which is to be manually developed by programmers. In the example, an outline of *ControllerImp* (lines 23-30) is also generated for programmers to start with.

```
01: class ControllerArch implements IController{
02:  IControllerImp _imp;
03:  IOperatorStk _operatorStk;
04:  IOperandStk _operandStk;
05:  IRegister _register;
06:  IMathUnit _mathUnit;
07:  public ControllerArch(){
08:   _imp = new ControllerImp();
09:   _imp.setArch(this);
10:  }
11:  public void enterOperator(String opcode){
12:   _imp.enterOperator(opcode);
13:  }
14:  public void enterDigit(String digit){
15:   _imp.enterDigit(digit);
16:  }
17: }
18: interface IControllerImp{
19:  public void setArch(ControllerArch arch);
20:  public void enterOperator(String opcode);
21:  public void enterDigit(String digit);
22: }
23: class ControllerImp implements IControllerImp{
24:  ControllerArch _arch;
25:  public void setArch (ControllerArch arch){
26:   _arch = arch;
27:  }
28:  public void enterOperator(String opcode){…}
29:  public void enterDigit(String digit){…}
30: }
```

List 4-1: Applying deep separation to the implementation of Component *Controller*.

Deep separation is able to prevent mistaken changes of generated architecture-prescribed code given that programmers' modifications to the code are precluded from the program element where generated code is located (e.g. *ControllerArch* and *IControllerImp* in List 4-1). At this point, a configuration management system (e.g. Subversion) can be used to ensure that architecture-prescribed code (e.g. *ControllerArch*) be only updated through code regeneration by the architect. A challenge that deep separation faces is explicitly integrating separated code, whereas this is automatically done by the inherent language relationship with shallow separation. In particular, this has to be done in the presence of frequent changes that may be made to both architecture and source code during software development. The remainder of this chapter specifically discusses how code integration and change management are handled.

4.3.2 Code Integration

The integration process is as suggested in Figure 4-1. The architecture information such as references to other connected components is passed from architecture-prescribed code to user-defined code; Meanwhile user-defined code provides primitive operations to architecture-prescribed code to implement higher-level operations. This is essentially a process of program composition, and can be done either through simple method calls or by building a source code integration framework. In this study, we only use method calls to integrate separated code. Specific analysis is also provided below about how a source code framework may be built for integration.

A straightforward way of integrating separated code in 1.x-way mapping is using object composition or method calls, as illustrated in List 4-1. Operations present in architecture-prescribed code are simply implemented by calling the corresponding operation (e.g. with the same signature) defined in user-defined code. Figure 4-5 further illustrates this by providing a high-level view of the integration process. *Primitive operations* in the figure represents a Java interface that consists of a list of operations that the architecture-prescribed code expects its user-defined code to provide. This is implemented as *IControllerImp* in List 4-1. What is special about Figure 4-5 is that it shows two user-defined implementations providing the same set of operations. This is to highlight the fact that specific implementations are encapsulated from the architecture-prescribed code with deep separation enforced. The architecture-prescribed code only sees the list of provided operations, without being aware of underlying different implementations. The advantage of integrating code with method calls is that it is easy to implement, and does not require redundant code. The disadvantage, however, is that it provides little control over the integration process.

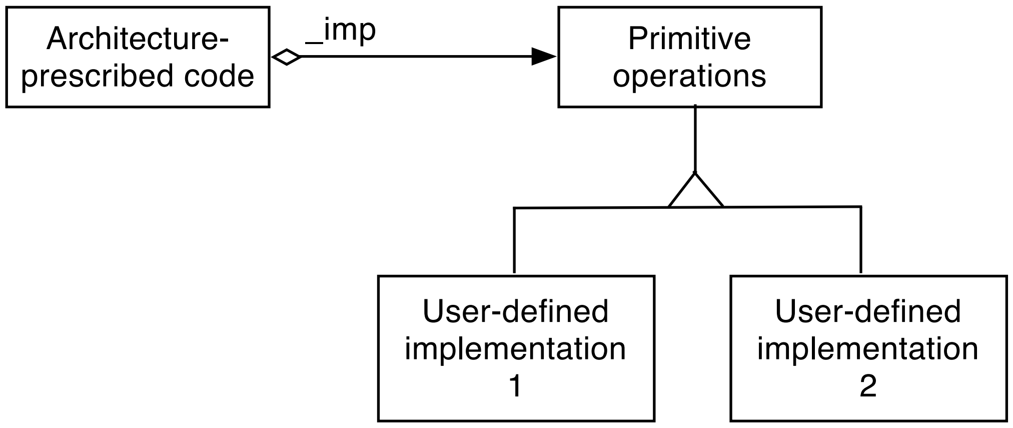

Architecture-prescribed code
_imp
Primitive operations
User-defined implementation 1
User-defined implementation 2

Figure 4-5: Integrating code by method calls.

In contrast, another way to integrate code is using a software framework, an abstraction in which common code providing generic functionality can be selectively overridden or specialized to provide specific functionality [75]. Figure 4-6 suggests an example of such a framework. Compared with method calls, an integration framework not only integrates code, but also gives developers a chance to customize the process to satisfy any special requirements, such as architecture-based dynamic adaptation. In addition, the use of an integration framework also makes it possible for architecture-prescribed code and user-defined code of a component to run on different machines, with the integration framework taking care of network-related communication issues. The downside of using such an integration framework is that additional application independent code is induced. In particular, it is often required that the deployed application be deployed in an environment where the framework is installed.

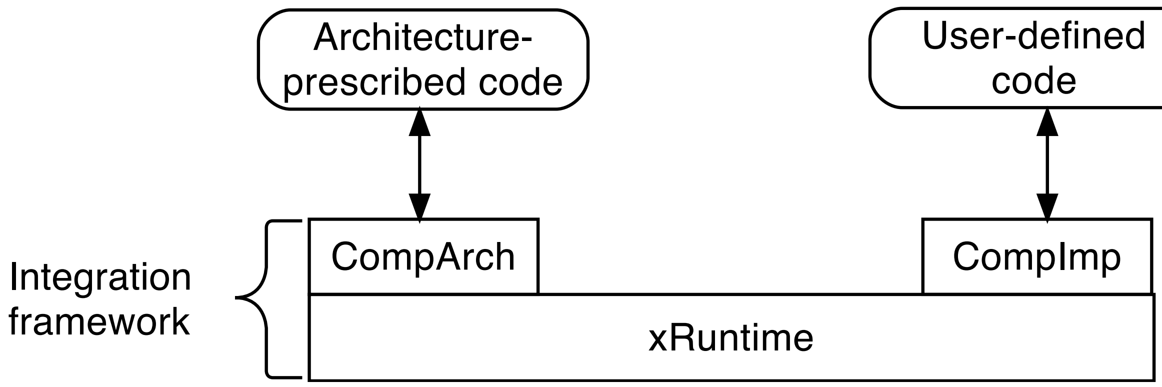


Figure 4-6: An example of code integration framework.

The framework shown in the figure consists of two kernel classes, *CompArch* and *CompImp*, which represent the base classes for architecture-prescribed code and user-defined code of each architecture component. They encapsulate integration related activities from the overlying application, and communicate with the underlying *xRuntime* to make method calls. It is *xRuntime* that explicitly controls which specific operation in user-defined code is called when a request from architecture-prescribed code arrives. The code below in List 4-2 is an example of the generated architecture-prescribed code for a component based on this framework. The request method (line 04) in the code is predefined in *CompArch*. It converts the incoming operation request to a canonical format recognizable by *xRuntime*, where application specific logics are applied before the request is redirected to the implementation developed by the programmer. There, *CompImp* does a reverse process, and translates the received request back to a regular method call.

```
class ControllerArch extends CompArch implements IController{
 …
 public void enterOperator(String opcode) {
  request("enterOperator", opcode);
 }
 public void enterDigit(String digit) {
  request("enterDigit", digit);
 }
 …
}
```

List 4-2: Generated code based on an integration framework.

The presentation above simply highlights another possibility for integrating code in 1.x-way mapping, and is far from being a complete solution. More work is required concerning some specific issues, such as encapsulation of function calls and configuration of processing logics. Addressing some of these issues may require a functional programming language. The payback of having such a framework is as discussed previously: full control can be obtained over how the integration is done, and opportunities for dynamic reconfiguration and runtime reuse are opened up. This is beyond the focus of this study, which is about maintaining architecture-implementation conformance in software development.

4.3.3 Discussion: Deep Separation vs. Shallow Separation

As discussed in Section 2.2.1, separating generated code and non-generated code is not a new idea and has been around for decades. An important difference between the 1.x-way mapping approach and other code separation based approaches is the application of the deep separation mechanism, which is supported by a set of change management mechanisms. The change management of 1.x-way mapping is presented in next section. In this section, we highlight the advantages of deep separation over traditional shallow separation in the context of architecture-implementation mapping.

As mentioned at the beginning of this section, deep separation separates the code of an architecture component into two independent program elements, and relies on external program composition mechanisms to explicitly integrate separate code. In contrast, shallow separation relies on certain program built-in relationships (e.g. same class, inheritance, partial class) to automatically integrate separated code. Due to this essential difference, the code they can protect, the relationship between their separated code, and their potential usages are significantly different as summarized below.

- Deep separation provides more comprehensive code protection. With shallow separation, code integration is done in a static and rigid way. The code that can be separated, and thus protected, is limited to those that can be integrated in the exact pre-defined way and additional constraints are also inevitably induced. For example, filling-in-blanks as discussed in Section 2.2.1 mixes generated and non-generated code in the same program element; subclassing can only support code that can be clearly separated with an inheritance relationship; partial class requires that separated code cannot contain methods of the same signature. This is also an important reason why existing code separation mechanisms often fail to protect behavioral code since there is no appropriate language relationship that can support so. Deep separation, in contrast, relies on a flexible external program composition mechanism (e.g. method calls) to integrate separated code. The code that can be supported is relatively independent of how the code is integrated. Thus, more kinds of code, including behavioral code, can be supported.

- What deep separation essentially reflects is the spirit of code library [82] and virtual machine [119]. Deep separation enforces architecture-centric development: architecture-prescribed code can only be updated through code generation from the architecture. User-defined code plays the role of code library or programming platform in this context, based on which architecture-prescribed code is generated. For example, libraries of architecture implementations can be constructed for various architecture features based on the operations provided by underlying user-defined code without containing implementation details.
- Deep separation makes the separated code of each component mutually independent. Should the code require regeneration later, only the architecture-prescribed code is overwritten. The work on the user-defined code remains unaffected, unless the architecture is radically changed as discussed in next section. Similarly, modifications to user-defined code have no impact on the architecture-prescribed code, assuming the required operations are still provided. This reduces the chance that inconsistency may happen. Moreover, it gives both architect and programmers more freedom to work on their own part compared with shallow separation mechanisms such as filling-in-blanks and partial class.
- Finally, deep separation makes the usage of architecture information (e.g. services provided by other connected components) in user-defined code explicit and manageable. All accesses to architecture resources have to go through a handle (_*arch* in List 4-1) maintained in user-defined code. This opens up opportunities for advanced activities like architecture change requests discussed later in Section 4.4.3 that can be done based on static program analysis. In addition, prevention of user-

induced negative properties is enabled based on deep separation. This is specifically discussed in Section 4.4.4.

4.4 Change Management

Software artifacts are subject to constant changes during the development. Software architecture and code are not exceptions. These changes significantly endanger the conformance between the artifacts. Important techniques developed in 1.x-way mapping in this regard include an architecture change model, architecture-based code regeneration, and architecture change notification. We also discuss in this section how 1.x-way mapping could be extended to prevent programmer-induced negative properties.

4.4.1 Architecture Change Model

A significant challenge that 1.x-way mapping faces is mapping 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. 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 4-7 shows various types of architecture changes and how they are managed 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 dissertation, the focus will be on the mapping of component changes and behavior changes to code. Note 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.

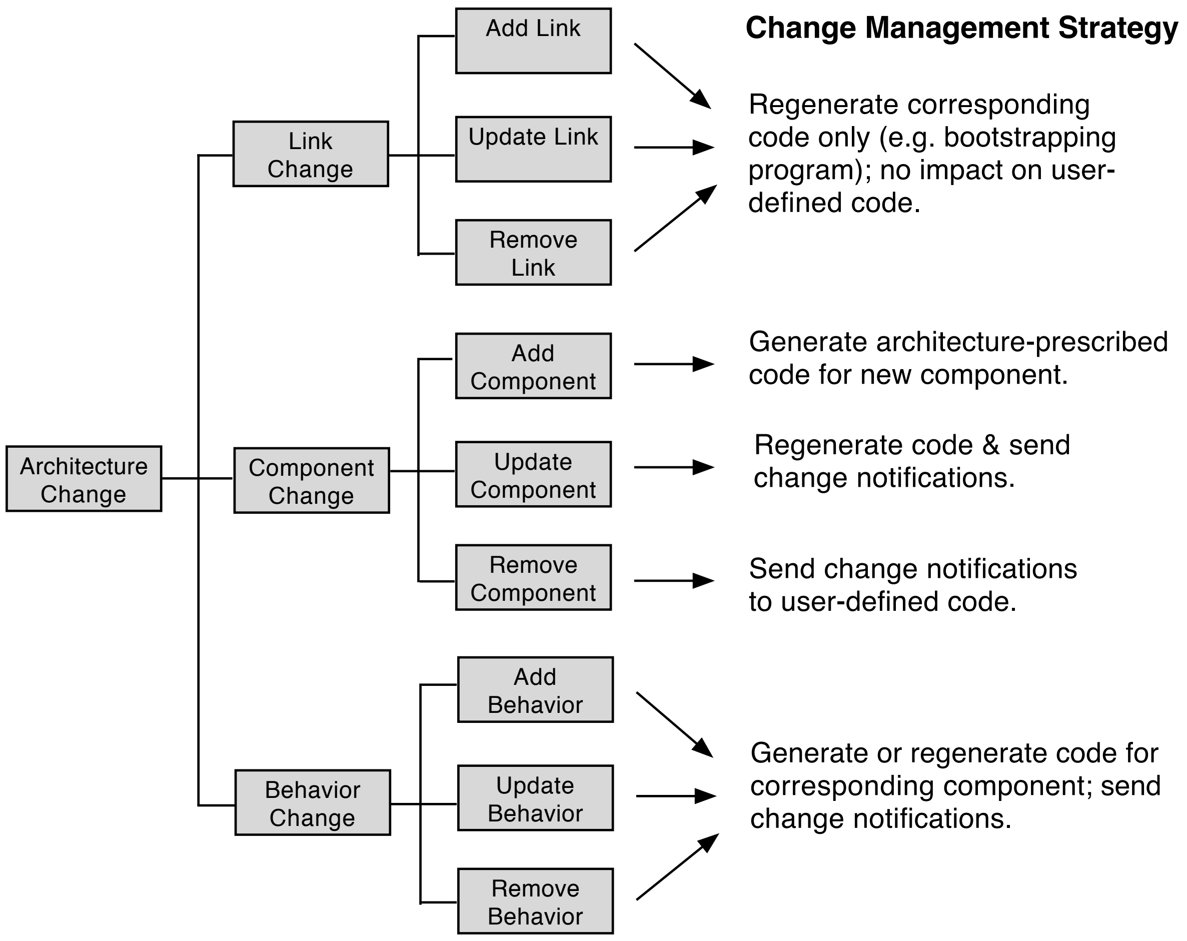

Change Management Strategy
Architecture Change
Link Change
Component Change
Behavior Change
Add Link
Update Link
Remove Link
Add Component
Update Component
Remove Component
Add Behavior
Update Behavior
Remove Behavior
Regenerate corresponding code only (e.g. bootstrapping program); no impact on user-defined code.
Generate architecture-prescribed code for new component.
Regenerate code & send change notifications.
Send change notifications to user-defined code.
Generate or regenerate code for corresponding component; send change notifications.

Figure 4-7: 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 with extensions made to the xADL architecture description language. Basically, all architecture elements are monitored, and all the considered changes made to them are automatically recorded and classified. An initial design of the architecture change model is shown below in List 4-3. As can be seen, a new element <*archChange*> (line 02) is added to the root (<*xArch*>) of the architecture description. Under the new element, there are multiple <*changes*> elements (lines 03 - 11), each of which represents a change session that includes a series of specific changes as listed in Figure 4-7. The status of each change session is either "*mapped*" (line 03) or "*unmapped*" (line 09), depending on whether a map-to-code process is done on the session or not. A new change session will be created automatically if the architecture is modified, and no "unmapped" change session exists.

```
01: <xArch>
02:   <archChange>
03:     <changes status="mapped">
04:       <compChange type="add"> … </compChange>
05:       <linkChange type="remove"> … </linkChange>
06:       … <!-- Other specific changes -->
07:     </changes>
08:     … <!-- Additional change sessions -->
09:     <changes status="unmapped">
10:       … <!-- Specific changes -->
11:     </changes>
12:   </archChange>
13: </xArch>
```

List 4-3: Basic structure of architecture change model.

Having an explicit architecture model can benefit many related activities, such as change analysis, redo/undo, and change reuse. Some of these applications are future work and are further discussed in Chapter 7. Different usages often have different requirements in the content of the created change model. In the design of our architecture change model, we intentionally make it independent of the following process of mapping to code. That said, the change model in 1.x-way mapping contains much more information than those that are needed to either regenerate code or send change notifications as introduced in the following subsections. Basically, all the architecture changes are recorded even though only a portion of them is of interest to the mapping process.

4.4.2 Architecture-based Code Regeneration

In principle, all architecture changes require the update of architecture-prescribed code. An easy approach to accomplish this is brute force regeneration. It completely regenerates code for the architecture regardless of what is changed. As discussed in Section 2.3, this approach suffers from the challenge of conflict resolution. In addition, complete code regeneration is also not scalable in the sense that 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, which only regenerates code for the changed portion, such as an added interface or a removed link. This approach reduces the amount of regenerated code, and thus, minimizes the impact of code regeneration to the rest of the system. The problem is that incremental change may break the system structure and consistency if the regenerated portion is in a highly cohesive entity, such as an architecture component [150]. Moreover, it is often hard to clearly tell which specific part of the modified architecture element should be regenerating code for, given that change impact analysis itself is still a research problem.

The 1.x-way architecture-implementation mapping approach 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. 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. Regenerating code for a specific component does not affect the implementation of other components, and thus the consistency of the whole system. Figure 4-8 illustrates how architecture-based code regeneration is different from complete regeneration and incremental change.

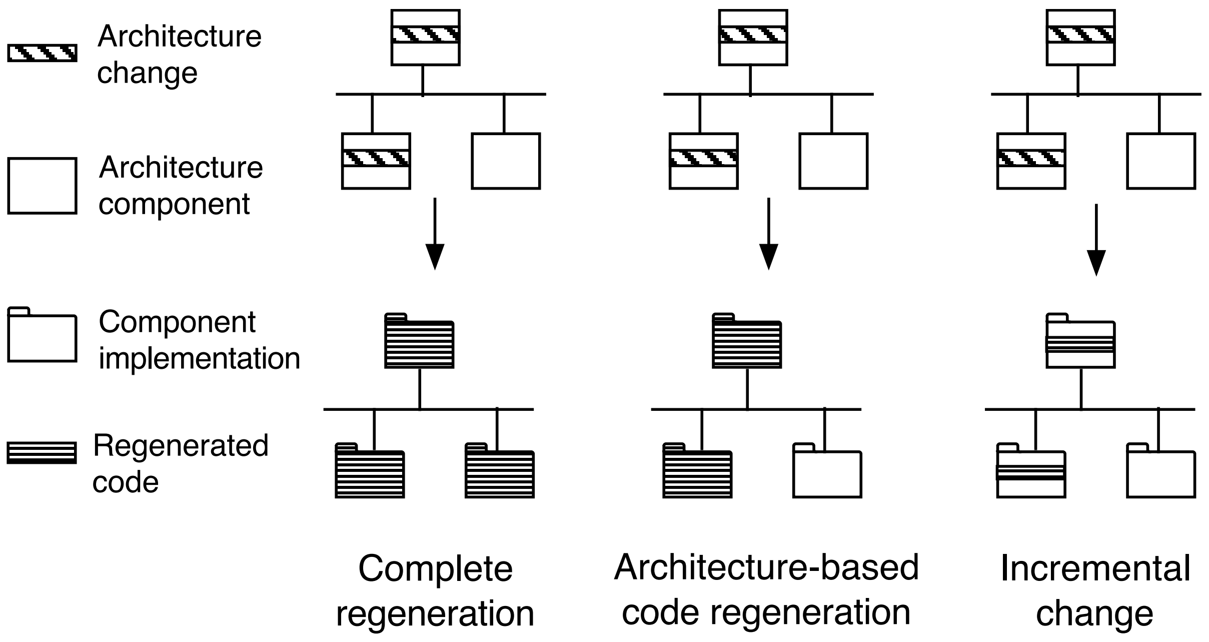


Figure 4-8: Code regeneration mechanisms.

It is important to note that link changes in 1.x-way mapping are still addressed by traditional incremental change mechanism. That is, only corresponding code is regenerated. This is based on the assumption that link changes represent changes of global architecture, and regenerating corresponding code does not affect the structure integrity of each specific component. It is especially the case when a bootstrapping program is used to start a system and initialize components with the connection information, such as the *myx.fw* framework discussed in Section 2.2.2. Architecture-based code regeneration discussed in this section is designed primarily to address component changes, such as adding interfaces or associating a new behavior. At this point, not only architecture-prescribed code of the modified components has to be completely regenerated, but also change notifications have to be sent to the user-defined code.

4.4.3 Architecture Change Notification

In contrast with architecture-prescribed code, it is not possible to automatically update user-defined code when architecture changes happen. If it were, we could simply move the part that can be automatically updated to architecture-prescribed code to improve software productivity. What can be done in general is to send change-related information across the separation boundary to user-defined code. Specifically, two kinds of information can be transferred: (1) what is changed in the architecture and (2) what needs to be changed in user-defined code. In this study, information about (1) is called *architecture change notification* and information about (2) is called *architecture change request*. In this dissertation, 1.x-way mapping only supports architecture change notification. Specific analysis is also provided below about how change requests may be generated, as a basis for future research.

An architecture change notification contains information describing what element (interfaces, properties, etc.) is changed in the architecture. The architect's comments when making those changes may also be included to give programmers more information. Note that this essentially addresses a problem of software architecture failure to capture design rationales that was identified earlier. 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 reduces unnecessary change notifications, decreasing the manual analysis work needed to process the notifications. To further reduce the number of unnecessary notifications, a plug-in could 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 following established traceability links.

In contrast, sending architecture change requests to user-defined code is essentially a problem of change impact analysis [5], a topic about identifying what to modify to accomplish a change. This is necessary because implementations in user-defined code may need specific architecture information, such as details of a required interface. When the element is changed in the architecture, related user-defined code should be updated correspondingly. Existing solutions include the application of transitive closure, inference, and program slicing. However, it remains to be seen how well these approaches can be applied in architecture-implementation mapping.

What is special about 1.x-way mapping is that all the architecture information is accessed explicitly through a single reference in user-defined code, as shown in Section 4.3.1. This facilitates the identification of the places where a changed architecture element is used. What can be done is to create traceability links between architecture elements and their usages in user-defined code through static program analysis. Based on the trace information, change requests can be generated. For example, if a component interface being used by user-defined code is removed, a warning message should be displayed in corresponding lines of user-defined code, just like a compiler works in a programming IDE. The challenge, however, is keeping these traceability links valid given that source code is under constant change as well.

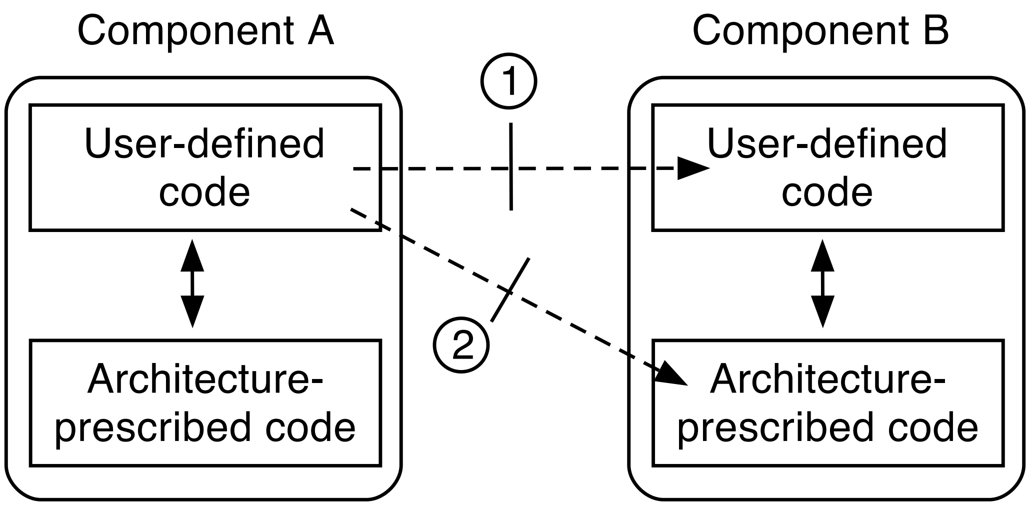
Both architecture change notifications and requests represent an important improvement over existing architecture-implementation mapping approaches. None of existing approaches cited earlier support notification during software development. At best, a program complier is used to detect inconsistencies in the code and return warning messages.

4.4.4 Discussion: Prevention of Programmer-induced Negative Properties

As introduced previously, 1.x-way mapping is able to protect architecture-prescribed code from being contaminated when developers work on user-defined code. What this essentially means is that the code does not lose properties of the architecture (e.g. specified elements). However, programmer-induced changes in user-defined code may also include new negative properties [72]. For example, the implementation of a component may (inappropriately) reference another component that is not connected to it in the architecture, obviously breaking the architecture-implementation conformance. Because programmers are granted full control over the user-defined code, it is technically hard to prevent this from happening during software development. As a result, most existing architecture-implementation mapping approaches either ignore the problem or simply rely on the after-the-fact consistency checking to detect it.

With 1.x-way mapping, a correct-by-construction method, the programmer-induced negative properties can be avoided from the very beginning with appropriate tools. Figure 4-9 illustrates how this may be done based on deep separation of 1.x-way mapping. Dashed lines in the figure represent illegal accesses from a component to the code of another unconnected component. Preventing illegal access to the user-defined code of a component is relatively straightforward. As introduced in Section 4.3, the user-defined code of a component does not communicate directly with other connected components with deep separation enforced. What can be done is to apply an access control mechanism (e.g. name scrambling), so that the user-defined code of a component is only accessible to its architecture-prescribed code, hidden from clients of the component. Illegal accesses to user-defined code from other components are thus prevented.

As to the architecture-prescribed code of a component, it is automatically generated and is not meant to be edited by programmers in 1.x-way mapping. If the architecture-prescribed code of a component is only accessible to the architecture-prescribed code of other components, illegal accesses from the user-defined code can be avoided as well. One possible way to do this is to expose the services, instead of the direct reference, of a component's architecture-prescribed code to other components and hide the mapping of services to references from programmers. As a result, the architecture-prescribed code of a component is accessed in a pre-defined way that is only understandable to the machine. A specific example of such an application is [35], where off-the-shelf middleware was used to implement architecture connectors between components.



1 The user-defined code of a component can only be accessed by its architecture-prescribed code (e.g. through name scrambling).

2 The architecture-prescribed code of a component can only be accessed by the architecture-prescribed code of other components (e.g. through framework encapsulation).

Figure 4-9: Prevention of programmer-induced negative properties.

Finally, user-defined code may also invalidate the architecture by not using certain architecture elements (e.g. a required interface). This can be resolved through the static program analysis mentioned in Section 4.4.3, and is not detailed here. Supporting and evaluating theses approaches is not part of this dissertation, and represents future directions for 1.x-way mapping.

4.5 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 behavioral mapping, it is important that (1) system behaviors be modeled in a form that is amenable to code generation; (2) there is 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. In this study, we simply reuse pragmatic techniques for the above two activities with necessary adaptations made. This allows us to focus on the consistency control of 1.x-way mapping. Each of these two activities, however, could be or is a research area. With new approaches to them available, 1.x-way mapping could support more behavioral models.

4.5.1 Architecture Behavioral Modeling in 1.x-Way Mapping

Architecture behavioral modeling in 1.x-way mapping is based on limited UML state diagrams and sequence diagrams. 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 with respect to one of its provided operations. Furthermore, in our current implementation, both call sequences and state changes do not contain advanced control structures such as iteration and branch. 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 4-10 presents an example of two behavioral diagrams defined for the calculator application introduced in Section 4.3.1. 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 with respect to its *execute* operation, assuming a binary operator is to be calculated. As shown in the figure, *MathUnit* first fetches two operands from *OperandStack* by calling its provided *pop* method, then makes the calculation through a self message call, and finally pushes the result back to the stack with another message call. Note that all the parameter passing and value assignments are explicitly represented in the figure. This is not required in a standard UML sequence diagram, and is an important adaptation that we made in 1.x-way mapping to facilitate behavioral code generation. This is further discussed later in this section.

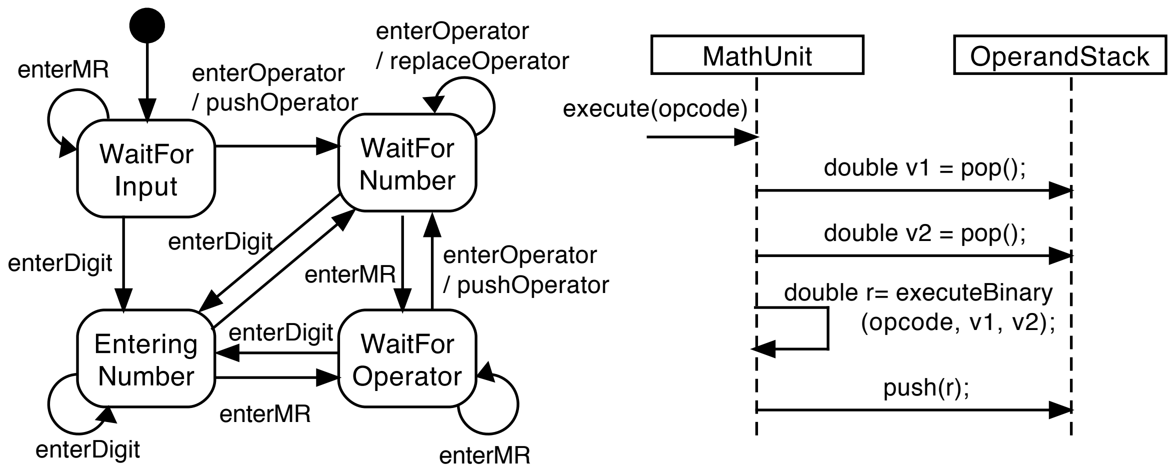


Figure 4-10: Example of a behavioral architecture definition.

Overall, a state diagram in 1.x-way mapping describes all of the possible states that a particular architecture component can get into and how the component's state changes as a result of events (i.e. invocations of the component's provided operations) that reach the component. A state diagram is focused on describing the behavior of a single component in response to external stimuli. As mentioned earlier, a standard UML state diagram contains many features, such as conditional transitions, superstates, and entry/exit events. In this study, however, only selected elements are implemented, allowing us to focus on the mapping of state diagrams to code. Specifically, a state diagram of 1.x-way mapping consists of the following two key elements:

- **State**. A state is represented by a rounded rectangle labeled with the state name in a state diagram. A component starts from an initial state, represented by the closed circle, and can end up in an optional final state, represented by the bordered circle. A state can either change to another state or remain in the original state when an event arrives, but only one transition can be taken out of a given state.
- **Transition**. Transitions are lines with arrowheads in a state diagram. A transition represents movement from one state to another, and is associated with a transition label. The format of a transition label is: *Event / Action*. Event is required, and triggers the transition. In our current implementation, an event is simply implemented as the invocation of a method that is provided by the component. Events can come from the external world, such as end-user input, external sensors, etc. Action is optional. It represents an (usually user-defined) operation that is to be called before the target state is entered. This gives user an opportunity to customize the state transitions.

A sequence diagram in 1.x-way mapping complements a state diagram by describing the interactions among architecture components. It shows a number of participating components and the messages (i.e. procedure calls) that are passed between these components with respect to an operation in one of a component's provided interfaces. Note that it is not recommended in 1.x-way mapping to make a sequence diagram for every single operation of a component. Instead, we believe sequence diagrams are most appropriate for those operations that exhibit interesting behavior, or "executions of significance" as described at the beginning of this chapter. This distinguishes our approach from MDD approaches that try to make UML a programming language, as discussed in Section 3.1.2.

Similar to state diagrams, we limit the standard UML sequence diagram: features like frames and new/delete messages are not included. The resulting sequence diagram primarily consists of:

- **Participant**. A participant component is shown as a box at the top of a dashed vertical line labeled with the component name. The leftmost participant is the component (called *host component*) whose operation is defined in the sequence diagram. From left to right are the components that the messages are sent to.
- **Message**. A message depicts the interactions among participants. It could be asynchronous event notifications, synchronous procedure calls, and so on. In our adapted sequence diagram, all this information is encapsulated and represented in architecture connectors and is not explicitly represented in a sequence diagram. In addition, a message must be labeled with the name and parameters of the corresponding message call. This is specifically discussed later in this section.

Generating code from state diagrams is a common practice with many CASE tools [106]. 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 [46]. In particular, the implementation of a sequence diagram is often found spread over the code (e.g. classes) of the participants that are involved [118]. 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, variable assignments and parameter passing must be explicitly represented for each message call. In Figure 4-10, variables *v1*, *v2*, *opcode*, and *r* are thus explicitly assigned and passed. Second, all message calls must start from the component whose operation is defined by the diagram. In other words, only operations that are directly called in the specified provided interface operation are considered in the diagram. With these restrictions, code can be easily generated. 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.

4.5.2 Applying Deep Separation to Behavioral Code

Once system dynamics are modeled in the supported form so that code can be automatically generated, the next challenge is finding a way to apply deep separation to the corresponding implementation. The methodology presented in Section 4.3.1 is still applicable, with architecture-prescribed code and user-defined code separated into two independent classes.

A difference is how the operation specified by a 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. List 4-4 is a portion of the generated architecture-prescribed code of *Math Unit* based on what is defined in Figure 4-10. The implementation of the *execute* operation (lines 04-07) 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 next chapter.

```
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: }
```

List 4-4: Generated code from a sequence diagram.

Applying deep separation to the implementation of our state diagram is based on the state pattern [53]. Again, no changes are required on user-defined code and the contract interface compared with what is presented in Section 4.3.1. A primary change of architecture-prescribed code is that multiple classes are generated, with each class corresponding to a specific state of the diagram. A class named *xxxArch* is also created as presented earlier, where all the structure information of the component (e.g. references to other connected components, provided interfaces, etc.) is implemented, serving as a container that the state classes reference to. Figure 4-11 shows the basic structure of the architecture-prescribed code of *Controller*, whose state changes are defined in Figure 4-10.

Specifically, three kinds 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 but need not be implemented differently in different states. Note that only a portion of *ControllerArch* is shown in the figure, while the remainder (e.g. the reference to user-defined code) is as defined in Section 4.3.1. *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 (e.g. system logging) may be specified; (3) identifying the successor state in case a state transition is triggered.

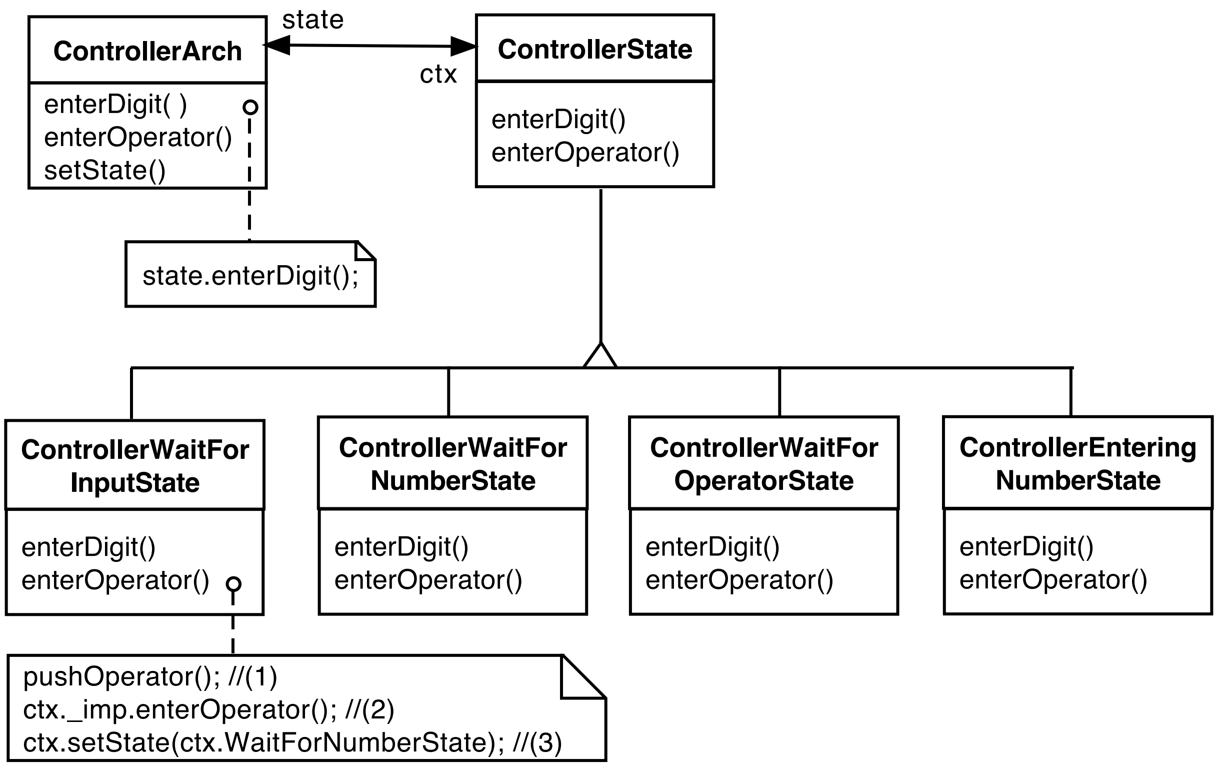


Figure 4-11: Structure of the architecture-prescribed code generated from a state diagram.

List 4-5 further illustrates the generated code from a state diagram by showing a portion of *ControllerArch*'s code that directly comes from the diagram. As can be seen, a list of variables (line 5 - 8) is explicitly maintained in the generated class to represent all the possible states that the component may get into. A variable *state* is also created, maintaining the current state of the component. In addition to initializing user-defined code as presented in Section 4.3.1, the constructor method (lines 11 - 18) now also needs to initialize state variables and set the initial state based on what is defined in the state diagram. A special method called *setState* (lines 20 - 22) is generated for the purpose of changing the current state. Finally, all the provided operations (e.g. *enterOperator* shown in line 24) of the architecture-prescribed code are now implemented by redirecting requests to the current state, which could be *ControllerWaitForInput*, *ControllerWaitForNumber*, *ControllerWaitForOperator*, or *ControllerEnteringNumber*. Corresponding operations are implemented in these classes just as shown in Figure 4-11.

```
01: class ControllerArch implements IController{
02:
03:  … //References to user-defined code, other components.
04:
05:  ControllerState ControllerWaitForInput;
06:  ControllerState ControllerWaitForNumber;
07:  ControllerState ControllerWaitForOperator;
08:  ControllerState ControllerEnteringNumber;
09:  ControllerState state = null;
10:
11:  public void ControllerArch(){
12:   … // Initialization of user-defined code
13:   ControllerWaitForInput = new ControllerWaitForInputState(this);
14:   ControllerWaitForNumber = new ControllerWaitForNumberState(this);
15:   ControllerWaitForOperator = new ControllerWaitForOperatorState(this);
16:   ControllerEnteringNumber = new ControllerEnteringNumberState(this);
17:   setState(ControllerWaitForInput); // Set the initial state
18:  }
19:
20:  public void setState(ControllerState newState){
21:   state = newState;
22:  }
23:
24:  public void enterOperator(String opcode){
25:   state. enterOperator(opcode);  // Redirect to current state.
26:  }
27:
28:  …//The implementation of other operations
29: }
```

List 4-5: Generated code of *ControllerArch*.

4.6 Revisiting Architecture-Implementation Mapping

As presented in Section 1.3, the hypothesis of this study is that 1.x-way mapping supports architecture-centric development, can be applied in the development of a realistic system to prevent its architecture-prescribed code from being manually changed by programmers, and supports automatic mapping of structural and behavioral architecture changes to code. In this section, we explore the hypothesis by comparing 1.x-way mapping with the existing architecture-implementation mapping approaches described in Chapter 2. Significantly, we present a framework consisting of a set of important criteria, which form a perspective for evaluating a specific architecture-implementation mapping approach. The purpose is to highlight how our approach contributes to architecture-implementation conformance. Meanwhile, some remaining challenges are also identified, making the scope of this research study clearer.

Table 4-1 compares architecture-implementation mapping approaches. The approaches are organized into three categories, one-way mapping, two-way mapping, and 1.x-way mapping. The comparison is made along eight dimensions:

- *Architecture model* highlights the type of architecture information that can be mapped to or from the code;
- *Generated code* depicts the form of architecture-prescribed code that is generated;
- *Architecture configuration changes*, *Architecture component changes*, and *Architecture behavioral changes* describe how corresponding changes are mapped from the architecture to the code;
- *Changes of Architecture-Prescribed Code* and *Changes of Implementation Details* represent two kinds of changes that are initiated by programmers, and may break the architecture-implementation conformance.

		One-way mapping	Two-way mapping	1.x-way mapping
	Architecture model	Structure, behaviors, non-functional properties.	Structure only.	Structure, executions of significance.
	Generated code	Complete program.	Code fragments and skeletons that are to be filled with details by developers.	An independent program element (for each component) that cannot be manually modified.
Architect initiated	Architecture link changes	Completely regenerate the code regardless of what is changed in the architecture.	Corresponding code is automatically updated to reflect link changes.	Automatically mapped to the code by regenerating the bootstrapping program.
	Architecture component changes		Regenerate code for the changed part of the component. Unrelated code remains (e.g. with EMF's JMerge). Change notification is not supported.	Completely regenerate code for the changed components, and send change notifications if the user-defined code needs to be updated.
	Architecture behavior changes		Cannot be mapped to the code through code regeneration. Have to be done manually.	Completely regenerate code for the changed components, and send change notifications if the user-defined code needs to be updated.
Programmer initiated	Changes of architecture-prescribed code	Manual changes are not allowed in the code.	Mapped to the architecture through reverse engineering or roundtrip engineering, both of which are of high complexity level.	Manual changes of architecture-prescribed code by programmers are not allowed.
	Changes of implementation details		Rely on the discipline or program compiler to avoid new negative properties.	Programmer-induced negative properties may be avoided with additional tools built as discussed in Section 4.4.4.

Table 4-1: A comparison of architecture-implementation mapping approaches.

The table indicates how these potential problems can be handled. As presented in the table, 1.x-way mapping has the following advantages over existing one-way mapping and two-way mapping approaches.

- More practical with current modeling and code generation technologies. The models supported by 1.x-way mapping capture system structure and some executions of significance (i.e. behaviors). Correspondingly, generated code is architecture-prescribed, separated from user-defined code via the deep separation mechanism. This is more practical compared with complete modeling and full code generation of one-way mapping. In contrast, most two-way mapping approaches currently can only (partially) support the mapping of structural architecture to the code. In practice, its generated code is often mixed with user-defined implementation details, given the limitations of current code separation mechanisms. 1.x-way mapping's position in the middle represents an advantageous level of modeling at this point in the evolution of software development technology. We believe it mixes just the right amount of modeling with programming to maximize the effectiveness of both. Moreover, 1.x-way mapping can be easily extended to support additional models with the development of corresponding technologies.
- A solution to architecture changes. 1.x-way mapping explicitly records and classifies architecture changes in an architecture change model, analyzes and refines recorded changes, and maps different kinds of architecture changes in specific ways. None of these features is supported by either one-way mapping or two-way mapping. Instead, all the existing approaches treat architecture changes simply as ordinary artifact changes and map them to code through complete code regeneration or incremental

changes mentioned earlier. This is obviously insufficient given that different architecture changes may have different impacts on the source code, and some changes combined together may not even affect the code at all.

- Regulation of code changes. As discussed in this chapter, programmers' manual changes to the code can invalidate the architecture by changing architecture-prescribed code, inducing new negative properties, or voiding certain architecture specifications by not making use of it. Of these different changes, 1.x-way mapping can prevent mistaken changes of architecture-prescribed code by programmers. With additional tools, 1.x-way mapping could also resolve the last two challenges as discussed earlier in Section 4.4.4. This is our future work. Two-way mapping approaches to certain extent can also prevent architecture-prescribed code from manual modifications. Some of them (e.g. ArchJava) can even guarantee that no new negative properties are induced in the code with the help of programming rules and compliers.
- Support for behavioral mapping. All the change management features described above can be applied to both structural and behavioral architecture in 1.x-way mapping. This represents another dimension along which our approach advances current technology. This is based on two important insights: (1) only executions of significance should be captured in behavioral architecture specification; (2) the application of deep separation. The first insight alleviates the challenge of complete modeling that one-way mapping faces. The second insight addresses the difficulties of clearly separating generated behavioral code from user-defined code that most two-way mapping approaches face. As discussed earlier, deep separation actually

reflects the spirit of code library and virtual machine. From this perspective, behavioral architecture can be implemented just as part of the architecture-prescribed code that is generated upon a set of low-level operations provided by programmers.

Finally, 1.x-way mapping also has a number of limitations. Some of these limitations are shared by all the existing architecture-implementation mapping approaches, such as not being able to support the mapping of non-functional properties (e.g. security, reliability, etc.) and system dynamics that are modeled using some expressive formal methods (e.g. some types of process algebra). Some other limitations, however, are specific to the design of 1.x-way mapping. First of all, 1.x-way mapping induces one more layer of indirection in the implementation of each architecture component. This may not noticeably affect the system performance given current computational power, but it brings additional challenges when it goes to system integration. In addition, our current investigation of 1.x-way mapping has focused on centralized applications written in object-oriented languages. It remains to be seen how other programming paradigms (e.g. functional programming) and applications where the separated code may be run on different machines can be supported.

5 Implementation

1.x-way architecture-implementation mapping is implemented as a tool named *xMapper* in ArchStudio 4, a tool integration environment that is fully integrated within the Eclipse platform as a plug-in project. This chapter begins with an introduction of the implementation environment, Eclipse and ArchStudio 4. After that important implementation tasks and some challenges are specifically discussed, including architecture change recording, code generation, and change notifications. An application scenario is then presented to illustrate how xMapper may be used in practice to maintain architecture-implementation conformance. Finally, reflections and lessons learned from the implementation experience are summarized at the end of the chapter.

5.1 Implementation Environment

As described in Section 4.2, 1.x-way mapping runs in an integrated software development environment (IDE) so that 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 Eclipse, or particularly ArchStudio 4, where the development activities of architecting and programming for an application co-exist. Another advantage of implementing our approach in such an environment is that the developed tool can be potentially distributed and deployed with the integration environment. This section provides an overview of Eclipse and ArchStudio 4, with the focus on their features (e.g. Eclipse's JET code generation engine) that are used in the implementation work.

5.1.1 Eclipse

Eclipse is an open development platform comprised of extensible frameworks, tools, and runtimes for building, deploying, and managing software across the lifecycle. The Eclipse platform is designed to provide tool providers with mechanisms to use and rules to follow, that lead to seamlessly-integrated tools. Typical examples of these Eclipse-based tools include Eclipse Java development tools (JDT) for Java, Eclipse Modeling Framework, the JUnit testing framework, and our ArchStudio 4 architecture development environment that is introduced later in this section. An important concept of Eclipse is *plug-in*, which is the smallest unit of Eclipse platform function that can be developed and delivered separately [23]. A small tool is written as a single plug-in, whereas a complex tool may have its functionality split across several plug-ins.

Each Eclipse plug-in contributes to the whole in a structured manner, may rely on services provided by another plug-in, and each in turn may provide services on which yet other plug-ins may rely. Specifically, each plug-in has a *manifest* file declaring its interconnections to other plug-ins. The interconnection model is simple: a plug-in declares any number of named *extension points*, and any number of *extensions* to one or more extension points in other plug-ins. A primary advantage of this plug-in mechanism is that each specific plug-in can be more readily reused to build applications not envisioned by the original developers of the plug-in. It represents an important extension mechanism of Eclipse. This is an important reason that many people think "the Eclipse platform is an IDE for anything, and for nothing in particular".

Even the Eclipse platform itself is portioned by the plug-in mechanism. Figure 5-1 shows the major components, and APIs, of the Eclipse platform. The kernel (runtime system) is based on Equinox, an implementation of OSGI framework [62]. All basic functionalities are plug-ins built on top of the kernel, including the Eclipse workbench and workspace.

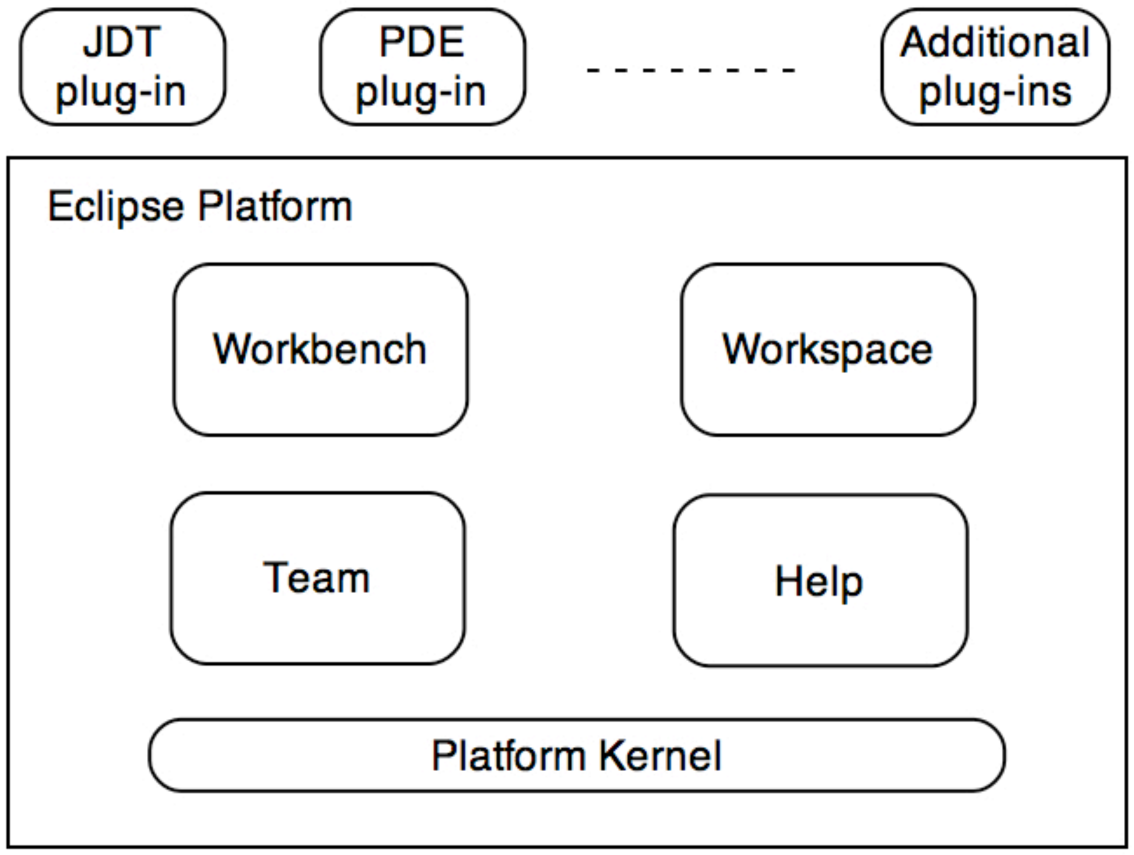


Figure 5-1: The architecture of the Eclipse platform [23].

All the components except *Platform Kernel* shown in the figure are integrated into Eclipse in the form of plug-ins. Even a portion of the Eclipse kernel or runtime is implemented as plug-ins. Generally speaking, *Workbench* provides both UI and non-UI behavior specific to the Eclipse IDE itself, such as projects, project natures, editors, views, and actions. *Workspace* plug-ins display and store user files as projects, source code, and so on. *Team* is a group of plug-ins providing services for integrating different types of source code control management systems (e.g. Subversion) into the IDE. *Help* plug-ins provide documentation for the Eclipse IDE. Finally, *JDT* and *PDE* plug-ins are usually shipped with the Eclipse platform to support development of Java programs and user-defined Eclipse plug-ins.

Eclipse Java Emitter Templates (JET) [44] is a template-based code generation engine that was built on the Eclipse platform. The original version of JET (JET1) is part of the Eclipse Modeling Framework (EMF), while the new version of JET (JET2) includes a number of new features and moved to the Eclipse Model To Text (M2T) project. JET2 operates in the context of Eclipse, and itself is an Eclipse plug-in. It is used in the implementation of the 1.x-way mapping approach to build a code generator as introduced later in next section.

The JET code generation engine loads an XML document that contains code generation parameters, follows user-defined JET templates that consist of both target text (source code in this case) and control tags, and finally generates source code. The generated code could be Java, C, or even some non-executable documents, depending on what is defined in the templates. In particular, Eclipse JET2 supports loading XML documents, and navigating them using *XPath* expressions [152]. This facilitates the implementation of 1.x-way mapping since the xADL architecture description is XML.

JET2 also provides standard JET tag libraries that make it possible to create relatively readable templates (compared with templates of JET1 that are embedded with Java scriptlets). Examples of these tags include <*c:choose*> (conditionally dumping text depending on the value), <*c:get*> (writing out the result of an XPath expression), and so on. These tags are used in JET templates to control the code generation process. In addition, JET can be extended with user-defined tag libraries. This is useful when the standard tag library cannot meet certain code generation needs. Finally, JET2 can read .java files and access the information in JET templates. This is a useful feature for our implementation, considering that some java interface files need to be loaded to generate corresponding methods that are specified in the interface files.

5.1.2 ArchStudio 4

ArchStudio 4 is an architecture development environment integrated within the Eclipse platform as a plug-in project. It supports developing, visualizing, and analyzing architecture models using the xADL language introduced in Section 3.1.2. ArchStudio 4 follows the Myx architecture style and is built upon the *myx.fw* framework described in Section 2.2.2. Users can extend xADL with new features, and automatically generate libraries used for those new features. This makes ArchStudio an ideal platform for investigating new architectural approaches and research directions. In addition, ArchStudio has been used in several companies and universities. However, ArchStudio did not have an automated architecture-implementation mapping tool before xMapper - the 1.x-way mapping tool was developed. With xMapper built and integrated, ArchStudio is now upgraded in terms of support for architecture-centric development.

Similar to Eclipse, extensibility is an important feature of ArchStudio in the sense that new tools can be relatively easily built and integrated into the ArchStudio environment. On the one hand, ArchStudio has provided a number of ancillary tools, such as *Archipelago*, *ArchEdit*, *AIM Launcher*, and *TypeWrangler*. These tools support some essential activities of architecture-centric development, and can be extended to address new architecture concerns. For example, modeling a new architecture concern can be done by adding a new xADL schema, followed by development of Archipelago plug-ins to add visualization support for the schema. On the other hand, new tools that are independent of the tools mentioned above can also be built and integrated with ArchStudio for the purpose of some other development activities, such as software traceability, product line architectures, and the architecture-implementation mapping focused in this study. Again, this process includes the activities of developing new xADL schemas and specific tools to explore new features.

Archipelago is ArchStudio's graphical editor that provides a symbolic point-and-click boxes-and-arrows editing interface. The current version of Archipelago is focused on structural architecture modeling. It supports the action of adding/removing architecture components, interfaces, and links. With regarding to behavioral modeling, only a basic statechart editor was built in Archipelago with many essential features missing, such as identification of a triggering event for a transition. Sequence diagrams, necessary for this study, were not supported in Archipelago. As described earlier in this section, this problem can be solved based on the extensibility support of Archipelago, and particularly its BNA framework.

Central to Archipelago is a framework called *BNA* (Boxes N Arrows), which resembles other graphical editing frameworks such as GEF and JGraph. BNA consists of a number of basic elements, including Things, ThingPeer, BNA Model, BNA Logics, and so on. These elements encapsulate low-level details that control things like how a specific architecture element (e.g. component) is rendered and displayed, and also provide APIs for the overlying application to customize the display of architecture to satisfy their special needs. This to a great extent facilitates modeling of new architecture elements. In particular, the BNA framework also follows a modular design, and can be easily extended to address new modeling concerns. For example, the information of each modeling element and its displaying characteristics are separated into two independent elements (BNA Thing and ThingPeer) in BNA.

ArchEdit is another useful tool of ArchStudio that provides a graphical user interface to syntactically edit software architecture specifications. ArchEdit depicts an architecture description graphically in a tree format, where each node can be expanded, collapsed, and edited like many XML editors. Significantly, the ArchEdit tree is automatically populated from the underlying xADL schemas. It does not have to be changed to provide support for new schemas. This makes ArchEdit a free low-level graphical editor for users who define new xADL schemas.

5.2 Implementation Tasks

This section specifically introduces the implementation of 1.x-way mapping. It starts with an explanation of how ArchStudio 4 is extended to support sequence diagram and structure diagram modeling. As discussed in Section 4.5, this is a task that is not inherent to 1.x-way mapping, but it is necessary for us to further explore the claimed capabilities of 1.x-way mapping. The implementation of 1.x-way mapping itself 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, Archipelago. It is relatively independent of Task (2), (3), and (4), which together form *Mapping Tool* shown in Figure 4-1. They communicate through production and consumption of the constructed architecture change model. The implementation of each task is presented in the following sections.

5.2.1 State Diagram Editor and Sequence Diagram Editor

A sequence diagram editor and a state diagram editor are built into the Archipelago modeling environment as part of our implementation work. Based on them, users can create and manipulate the adapted sequence diagrams and state diagrams described in Section 4.5. Corresponding implementation work includes the creation of new xADL schemas and development of the modeling interface on the basis of the BNA framework. During this process, we closely followed the provided tutorials of xADL and ArchStudio. Figure 5-2 and Figure 5-3 below show screenshots of the developed diagram editors.

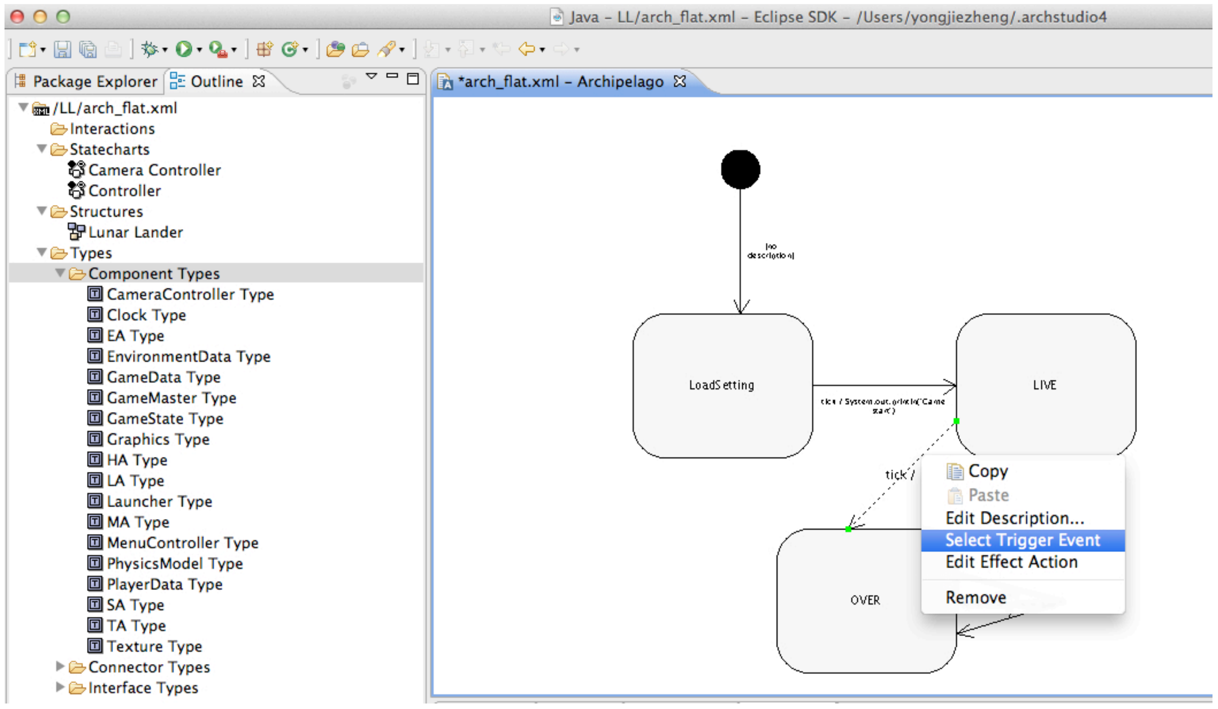
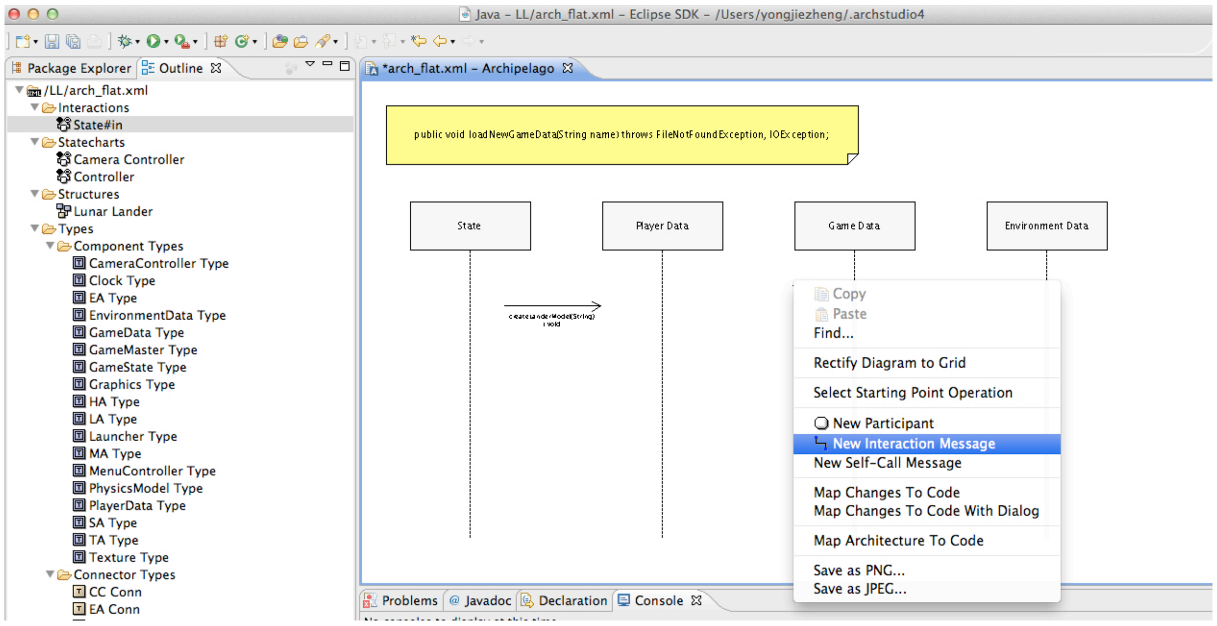


Figure 5-2: A screenshot of the state diagram editor.

As shown in the figure above, the developed state diagram editor supports the creation of states, transitions, and identification of triggering event for a specific transition. The black dot represents the initial state of the diagram. What is not shown in the figure is how a state diagram can be associated with a specific architecture component. This is done through an explicit selection of the corresponding component in a pop-up list when creating a state diagram. The underlying xADL description of a state diagram is not shown here due to its lengthy details and the focus of this study. In short, each state diagram is modeled as a *statechart* element in the xADL description. It consists a number of child elements, including *description*, *linkedComp*, *state*, and *transition*. For the elements of *state* and *transition*, there can be more than one and each contains further details about corresponding elements, such as state type (initial, normal, final), triggering event, and target state.



```
public void loadNewGameData(String name) throws FileNotFoundException, IOException;
```

Figure 5-3: A screenshot of the sequence diagram editor.

Figure 5-3 is a screenshot of the developed sequence diagram editor in Archipelago. The text box at the top of the figure represents the operation that is depicted by the diagram. It is specified at the beginning of creating a sequence diagram. The captured context menu lists several things that can be done in a sequence diagram editor, such as adding a new participant, adding a new message, and so on. The editor also facilitates selecting an interaction message. After the corresponding menu item is selected, a small selection window pops up, containing all the interfaces of the host component (the component whose operation is defined by the diagram). For each of the shown interfaces, a list of specific methods is shown. In this way, the user can select a corresponding method of the interface that is connected to the target participant component, and the signature of the selected method will automatically become the label of that interaction message. This reduces the amount of code that the user has to write, and also serves as a guide for the creation of a message.

A difficult issue in the implementation of both the state diagram editor and the sequence diagram editor described above is the xADL-BNA mapping. Specifically, there must be a way to update a particular part of the diagram (e.g., a component) based on a part of a xADL document. When that part of the xADL document changes, the corresponding symbol should be automatically updated to reflect the changes in the xADL document. In other words, all changes made to the xADL model should cause the corresponding update in the diagram. Generally speaking, this is a problem of maintaining the model-view synchronization that pervades in the development of all graphical editors.

In Archipelago, a BNA-based editor, the synchronization of the xADL and the BNA model is automatically monitored and handled by a class implemented in the BNA framework, *AbstractAutomapSingleAssemblyXArchRelativePathMappingLogic*. To support new modeling capabilities such as the editors of sequence diagrams and state diagrams, all one need do is just create specific logics that extend the above class with some customization code. Examples of these new logics that were created in our implementation include *MapXadlTransitionLogic*, *MapXadlStateLogic*, and *MapXadlParticipantLogic*. This special design keeps things simple and efficient, and represents another benefits of Archipelago and its BNA framework.

Note that the implementation of both the state diagram editor and the sequence diagram editor is not part of the architecture-implementation mapping work. We developed them simply because there were no corresponding editors in Archipelago when we started this research work. The introduction presented in this section highlights some special features of the developed editors. These editors provide us an application context, based on which we can further develop our architecture-implementation mapping tool – xMapper.

5.2.2 Recording Architecture Changes

The recording of architecture changes is integrated into Archipelago, which provides a graphical architecture-editing interface. It supports the action of adding/removing architectural components, interfaces, and links. In order to record changes made to an architecture, the xADL schema must be extended to specify changes in the architecture description. Based on that, specific recording logics are built in Archipelago. Figure 5-4 shows the recorded changes after a series of modifications to an architecture. What is also shown in the figure is a screenshot of the Archipelago structural modeling environment and how a map-to-code process is started.

As can be seen, a new element <*archChange*> is added to the root (<*xArch*>) of the architecture description, which used to contain the <*archStructure*> element only for architecture structure information. 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. The status of each change session is either "*mapped*" or "*unmapped*", depending on whether the map-to-code process 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. This represents an implicit change session management.

In contrast, an alternative way is to allow the architect to create a new change session explicitly, even if there is already an "*unmapped*" change session. By this means, parallel change sessions are enabled. They could be changes that modify different portions of the architecture, for different purposes, and be mapped to the code independently. Users are allowed to switch among different change sessions (e.g. for different tasks), so that the changes they made to the architecture are recorded into the corresponding change session. Another possible application is to visualize the changes of each change session. For example, select a change session on the left panel of Archipelago, and changes in that session will be highlighted in the architecture in the right panel of Archipelago. We believe there are some specific issues to be addressed, such as the relationships (e.g. mutual exclusion, dependency) between concurrent change sessions. What should be noted here is that it is actually the architecture-based code regeneration mechanism presented above that makes parallel change sessions a possibility.

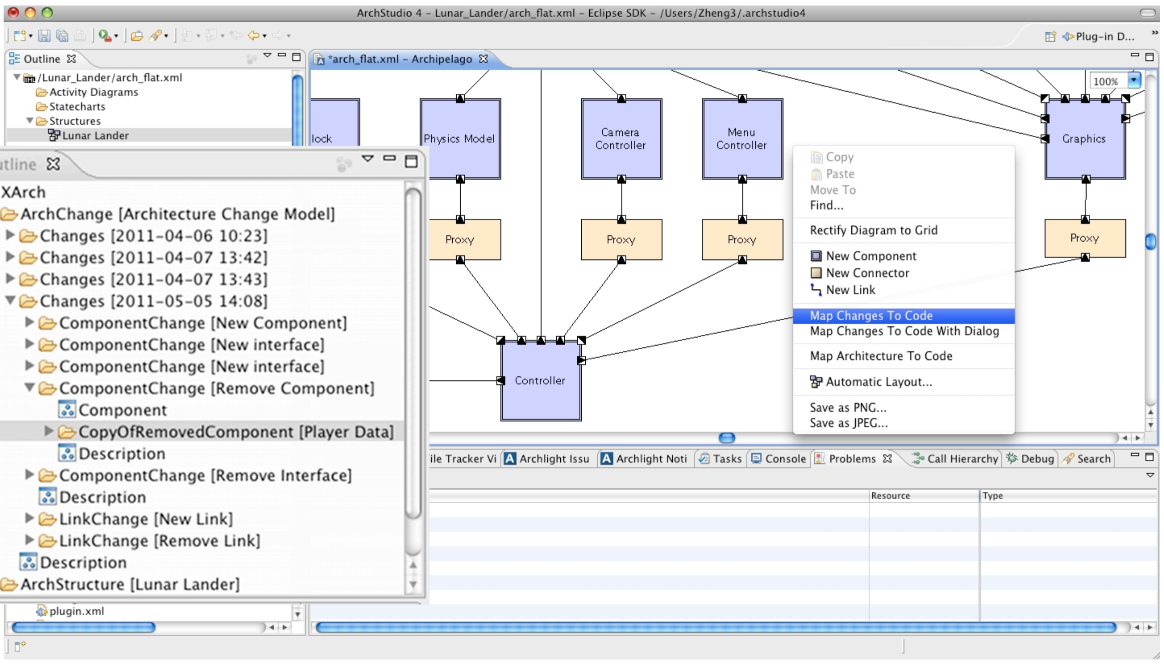

ArchStudio 4 – Lunar_Lander/arch_flat.xml – Eclipse SDK – /Users/Zheng3/.archstudio4
Outline
/Lunar_Lander/arch_flat.xml
Activity Diagrams
Statecharts
Structures
Lunar Lander
XArch
ArchChange [Architecture Change Model]
Changes [2011-04-06 10:23]
Changes [2011-04-07 13:42]
Changes [2011-04-07 13:43]
Changes [2011-05-05 14:08]
ComponentChange [New Component]
ComponentChange [New interface]
ComponentChange [New interface]
ComponentChange [Remove Component]
Component
CopyOfRemovedComponent [Player Data]
Description
ComponentChange [Remove Interface]
Description
LinkChange [New Link]
LinkChange [Remove Link]
Description
ArchStructure [Lunar Lander]
plugin.xml
*arch_flat.xml – Archipelago
100%
Physics Model
Camera Controller
Menu Controller
Graphics
Proxy
Proxy
Proxy
Proxy
Controller
Copy
Paste
Move To
Find...
Rectify Diagram to Grid
New Component
New Connector
New Link
Map Changes To Code
Map Changes To Code With Dialog
Map Architecture To Code
Automatic Layout...
Save as PNG...
Save as JPEG...
Archlight Issu
Archlight Noti
Tasks
Console
Problems
Call Hierarchy
Debug
Search
Resource
Type

Figure 5-4: Architecture change recording in the Archipelago modeling environment.

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 (name, attributes, …) of the removed element during the map-to-code process, such as generation of change notifications. With the current design of Archipelago, the element will be removed permanently from the architecture once that removal action is triggered and saved.

A possible solution is to change the logic of Archipelago: mark the element as something like "*toBeRemoved*", and remove it after the changes are successfully mapped to the code. This solves the problem, but it requires unwanted modifications to Archipelago's existing logic. 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 explicit modeling of architecture changes.

With respect to the logics of recording architecture changes, the Archipelago architecture editor works in two modes: *recording* and *normal*. When the editor is entered or re-entered (e.g. by double-clicking a node in the left panel of Archipelago), the system will check if there is any "*unmapped*" change session in the xADL document. If an "*unmapped*" session exists, the editor will load the change session ID into an environment variable (e.g. *sessionID*) and automatically enter the recording mode – all the changes made afterwards will be recorded into the loaded change session and the "Map Changes To Code" menu item shown in Figure 5-4 is enabled. Otherwise the editor is in the normal mode: the *sessionID* variable will be set to null and the map-to-code menu item is disabled.

Once the editor is entered, there are two kinds of actions that can trigger the mode transition: making changes to the architecture and starting the map-to-code process. The former makes the editor enter the recording state. Note that this may include the creation of a new change session and the action of setting the *sessionID* variable if the editor is originally in the normal mode. In contrast, the latter simply changes the mode from recording to normal and clears the *sessionID* variable.

5.2.3 Change Analysis and Refinement

The architecture change model only records "*raw*" changes. In other words, it simply reflects what has been done to the architecture, and is 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. 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 *removedComponent*. In particular, the intersection of these change sets is guaranteed to be null. This ensures that each changed architecture element be mapped to code in an unambiguous way.

List 5-1 shows the filtering logic of change analysis and refinement in 1.x-way mapping. It 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 *removeX* (e.g. *removeComponent*) change item comes in, the set of *addedX* should be checked first. If the entity to be removed exists, the corresponding entry in the *addedX* set should be removed and the *removeX* change item is discarded. Otherwise the set of *updatedX* is checked. Again if the entity to be removed exists, the corresponding entry in the *updatedX* set should be removed and the *removeX* change item is added to the *removedX* set. Finally, if neither of the above is true, add the entity associated with *removeX* to the *removedX* set. In this way, the scenario discussed above can be successfully addressed, all the changes will be discarded in the end, and the code remains.

```
// X below represents a changed architecture element
for each addX change
  add X to AddedElements

for each removeX change
  if (X is in AddedElements)
    remove X from AddedElements
  else if (X is in UpdatedElements)
         remove X from UpdatedElements
         add X to removedElements
       else
         add X to removedElements

for each updateX change
  if (X is in AddedElements)
    discard updateX
  else
    add X to updatedElements
```

List 5-1: The filtering logic of change analysis.

The filtering logic illustrated above is based on the assumption that all the changes in an architecture change session are recorded and processed in the order of occurrence. In other words, an *addX* change item should always be recorded and processed before either *updateX* or *removeX* of the same architecture element. This is guaranteed by the implementation of our architecture recording and map-to-code process. It also explains a portion of the above filtering logic. For example, an *addX* item can be simply processed by adding *X* into the *addedX* set without checking either *updatedX* or *removedX* since there is no way that *X* could be updated or removed before it is added under the current design.

Finally, note that the filtering logic presented above represents a basic exploration towards this direction. It can be extended or customized with more advanced logics to satisfy some special needs. For example, a new filtering logic may be created to specify that only changes to a specific architecture element is considered and all other changes should be discarded. Or only the changes made by a specific person are processed and mapped to code. All these advanced logics can be developed based on our architecture change model, and this highlights a possible extension of 1.x-way mapping. For now, the filtering logic in List 5-1 is sufficient for us to explore change management and support for behavioral mapping, which is the focus of this study.

5.2.4 Code Generation

1.x-way mapping updates architecture-prescribed code through code regeneration. In particular, code that is regenerated is proportional to the architecture elements that are changed. For link changes mentioned above, 1.x-way mapping regenerates code that is responsible for bootstrapping the program with the connection information. For component changes, in contrast, only the code of the corresponding component is regenerated. The code generator of xMapper is built upon on the JET2 (Java Emitter Templates) technology of the Eclipse Modeling Project. As mentioned earlier, JET2 operates in the context of Eclipse, and itself is an Eclipse plug-in. JET2 supports loading XML documents, and navigating them using XPath expressions. This facilitates the implementation of 1.x-way mapping since the xADL architecture description is XML. JET2 also provides standard JET tag libraries that make it possible to create relatively readable templates.

Figure 5-5 shows an overview of the code generation process. In general, a JET code generator only requires three inputs: a complied *JET Template*, *XML input* (the xADL architecture specification in our case), 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.

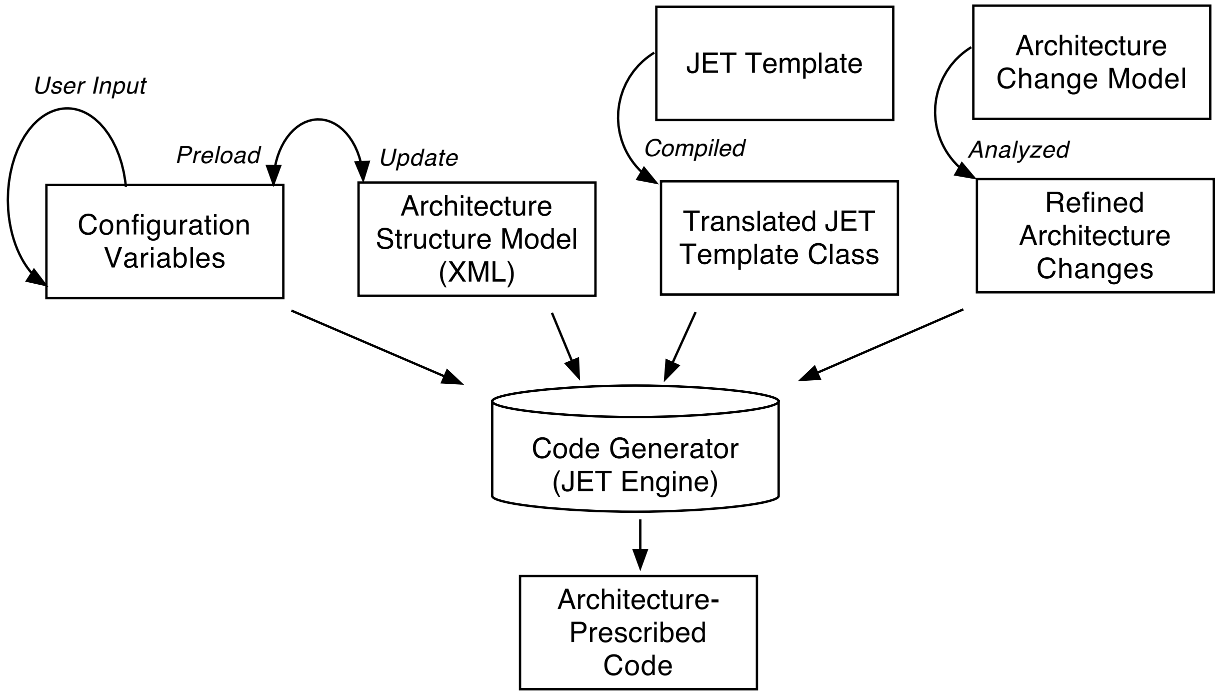
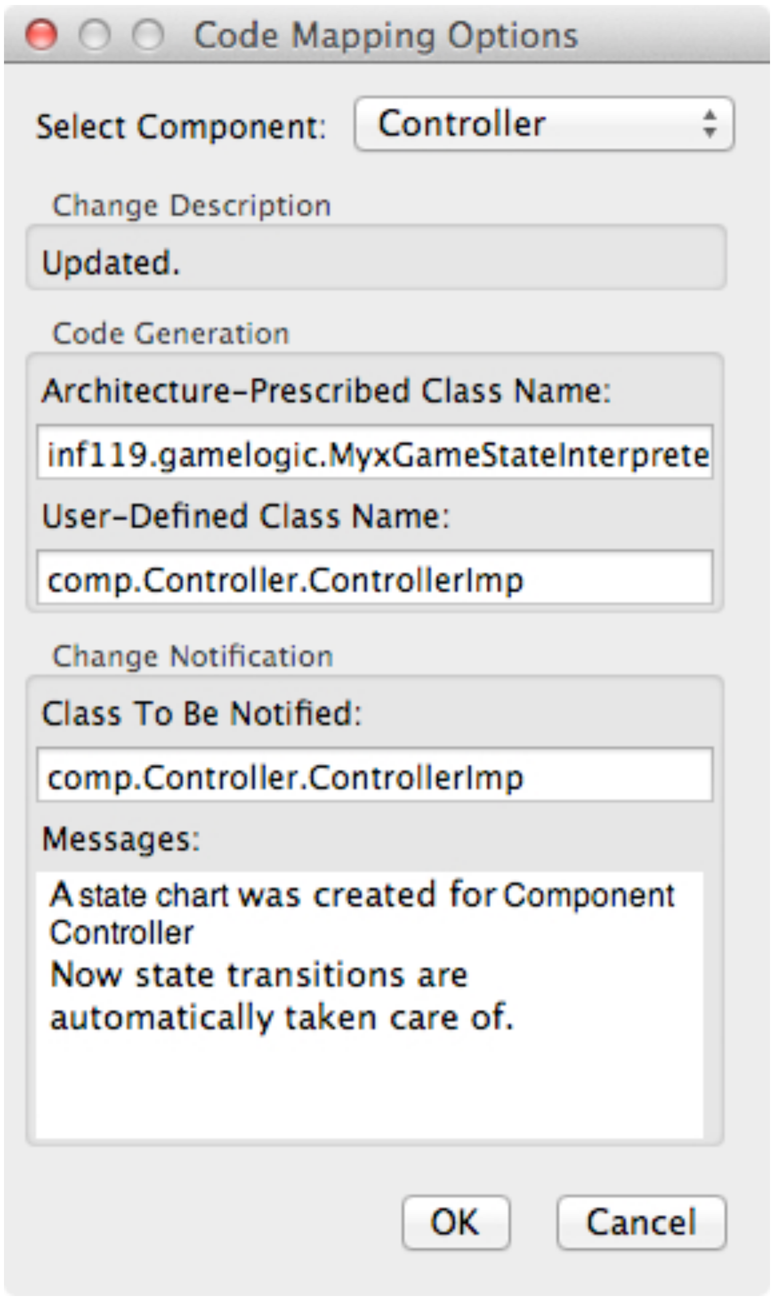

User Input
Preload
Update
Configuration Variables
Architecture Structure Model (XML)
JET Template
Compiled
Translated JET Template Class
Architecture Change Model
Analyzed
Refined Architecture Changes
Code Generator (JET Engine)
Architecture-Prescribed Code

Figure 5-5: Code generation in 1.x-way mapping.

In particular, a configuration panel of code generation is developed as shown in Figure 5-6. It collects *Configuration Variables* as shown in Figure 5-5 and provides an opportunity for the user to tune the code generation process by changing specific parameters, such as the name of generated classes. 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. The panel also allows the user to manually edit generated change notifications, and this is discussed in next section.

Note that more parameters can be changed through the configuration panel as long as there is a way to load and write corresponding information from and to the architecture. Shown here is an illustration of how this can be done. It is by no means to be complete. Generally, the more parameters that can be changed, the more flexible the code generation process is, and the more variations can be addressed during code generation.


Code Mapping Options

Select Component: Controller

Change Description

Updated.

Code Generation

Architecture-Prescribed Class Name:

inf119.gamelogic.MyxGameStateInterprete

User-Defined Class Name:

comp.Controller.ControllerImp

Change Notification

Class To Be Notified:

comp.Controller.ControllerImp

Messages:

A state chart was created for Component Controller
Now state transitions are automatically taken care of.

Figure 5-6: A configuration panel that tunes the mapping process.

The *JET Template* in Figure 5-5 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 Chapter 4. Different strategies are taken to generate code for architecture elements that are structural only versus those that are linked to a behavioral diagram, as discussed in Section 4.5.2. Linking structural and behavioral architecture definitions is not a challenge in general based on the XML technology. It is somewhat difficult, however, to associate an operation in a Java interface file with a sequence diagram so that code can be generated from the corresponding sequence diagram for this operation. 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 in List 5-2 below. The tag serves as a link to the corresponding diagram by which the code generator loads information and generates code.

```
/**
 * @see interactionffea0a1b-0c463028
 */
public void execute(String opcode);
```

List 5-2: Annotating the method defined by a sequence diagram.

Specifically, a number of JET templates were developed in our implementation, including *main.jet*, *comparch.jet*, *icomp.jet*, *compimp.jet*, *comparch_sc.jet*, *abstract_st.jet*, and *concrete_st.jet*. The last three templates were specifically for state diagram based code generation. Each of these templates is briefly introduced below:

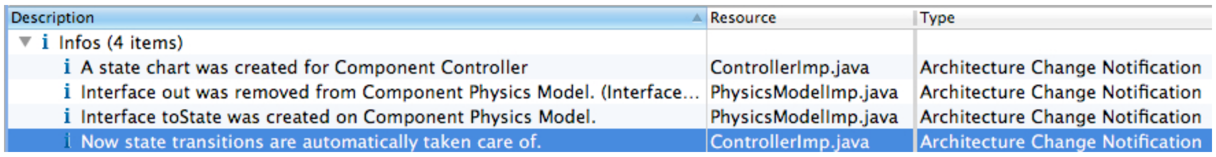
- *main.jet*. This is the entry point when the code generation begins. The template specifies some general information, such as where to dump the generated code and which template to use for a specific architecture element.

- *comparch.jet*. This is the template with which to generate the architecture-prescribed (mostly structural) code of a component. For components that contain the operation defined by a sequence diagram, the operation is generated as described in Chapter 4.
- *icomp.jet*. This template generates the interface between the architecture-prescribed and user-defined code of a component. Basically, it contains a list of specific operations that architecture-prescribed code expects user-defined code to provide.
- *compimp.jet*. This is the template for user-defined code. Note that the generated code is just to provide a starting point for the programmer to work with, and may be manually changed by the programmer.
- *comparch_sc.jet*, *abstract_st.jet*, and *concrete_sc.jet*. These templates control the generation of architecture-prescribe code for state diagrams. Recall what is shown in Figure 4-11, *comparch_sc.jet* is the template for the architecture-prescribed code that provides a container; *abstract_st.jet* is the template for abstract state; *concrete_sc.jet* is the template for concrete state. For the user-defined code of state diagrams, it simply reuses the *compimp.jet* template described above.

5.2.5 Sending Change Notifications

The architecture change notification of 1.x-way mapping is built upon the Eclipse *Markers* technology [23], which is used in Eclipse to annotate specific locations within a resource. For example, the Eclipse Java compiler not only produces class files from source files, but also annotates the source files by adding markers to indicate compilation errors. Eclipse markers provide a mechanism that automates the delivery and display of notifications in user-defined code. 1.x-way mapping generates notifications based on refined changes and architecture information. In particular, the architect is able to review and edit generated notifications through the configuration panel mentioned above. This is to support scenarios where additional information, such as the rationale of a specific change, need to be provided to programmers [130].

Figure 5-7 shows an example of generated notifications and how they are displayed. Clicking any of these messages will lead the user to the corresponding source code. All the messages are automatically generated except the one highlighted, which was manually edited by the architect in the configuration panel shown in Figure 5-6. In addition, note that information (e.g. the name of removed interface) of the message "Interface out was removed …" is actually from the copy created in the architecture change model, as discussed previously in this section.



Description	Resource	Type
Infos (4 items)		
A state chart was created for Component Controller	ControllerImp.java	Architecture Change Notification
Interface out was removed from Component Physics Model. (Interface…	PhysicsModelImp.java	Architecture Change Notification
Interface toState was created on Component Physics Model.	PhysicsModelImp.java	Architecture Change Notification
Now state transitions are automatically taken care of.	ControllerImp.java	Architecture Change Notification

Figure 5-7: An example of architecture change notifications.

There are two choices regarding the persistence of these notifications. We can make them either *shown-and-disappear* or *persistent*, depending on how important the corresponding notification is. Notifications of the former type will be gone automatically once the programmer read them, while notifications of the latter type must be manually removed by the programmer. Thanks to the Eclipse Markers technology, this customization can be easily done with corresponding parameters specified. Another possible way of extending the notification mechanism is to assign a priority level to each specific architecture change notification. For example, we can assign high priority level to those notifications that the architect wants the programmer to react to instantly, such as the implementation of a required component interface. In this way, a better communication between the architecture and the programmer is enabled.

5.3 Tool Usage

The developed xMapper tool helps maintain architecture-implementation conformance in architecture-centric software development, where software architecture plays a central role and drives development activities like software synthesis, evolution, and integration. Architecture-centric development requires that all architecture-related changes should start from the architecture, and be mapped to code afterwards. Traditionally, however, this only works under the assumption that programmers are highly disciplined since an automated tool that could enforce and facilitate this development process has not been fully available yet. Below we present a scenario to illustrate how xMapper can be used by both architect and programmer in the context of architecture-centric software development to overcome this previous shortcoming.

Architect. Mike works for a software project as an architect. He develops an architecture model in ArchStudio using the xADL language. After Mike finishes his work, he right clicks the mouse and selects "Map Architecture To Code" in the pop up menu. At that point, the code generation engine of xMapper is invoked and starts to generate architecture-prescribed code for every component. Meanwhile, a Java interface file is also generated for each component. It consists of the low-level operations (or primitive operations as mentioned earlier) that the architecture-prescribed code of a component expects its user-defined code to provide. The generated interface file is then passed to a corresponding programmer to implement.

Mike puts the generated architecture-prescribed code under the protection of a configuration management system (e.g. Subversion), so that the code can only be updated by him through the next round of code generation. After a while, Mike decides to make some changes to the architecture, either to address a requirement change or for optimization. Once he starts to do that in the ArchStudio modeling environment, a new "unmapped" change session is automatically created in the architecture change model and all the following changes are recorded and classified in it. Without having to worry about implementation details, Mike makes all the architecture changes and simply selects “Map Changes to Code” to get the corresponding code updated. Alternatively, he can also select “Map Changes To Code With Dialog” if he wants to change some default code generation parameters or review and edit generated notifications. As a result, the architecture-prescribed code of changed components is regenerated, necessary notifications are sent to the programmer(s) of the user-defined code, and the change session is closed with its status updated to “mapped”.

Mike may also want to capture some system dynamics or executions of significance in his architecture. It is important in architecture-centric development that behavioral architecture and its changes can be accurately mapped to code. With xMapper, both can be done in an automatic manner as presented above. All Mike needs to do is model system behaviors in the supported forms, UML-like sequence diagrams and state diagrams, and make sure that the developed models are consistent with each other.

Programmer. Jack is a programmer that uses xMapper to collaborate with Mike in the project. He is responsible for the internal implementation of one or more components, and is supposed to implement low-level operations required by the corresponding architecture-prescribed code. During this process, Jack is provided with a reference to the architecture-prescribed code in the form of method parameters. Through this reference, Jack has access to services provided by other components that are connected to the component Jack is working on. With xMapper being used, Jack does not have to worry about whether his work contaminates the architecture-prescribed code or not. Jack’s code is also sustainable to architecture changes and code regenerations that follow since many of these changes (e.g. link changes) only require regenerating architecture-prescribed code. Jack gets notifications for the architecture changes that may require modifications to his code (e.g. component changes). In particular, he only gets notifications for changes that are made to his component. In this way, Jack will not be overwhelmed by unrelated notifications.

5.4 Lessons Learned

Implementing 1.x-way mapping in Eclipse and ArchStudio provides some first-hand understandings about the power and limitations of these two systems. Overall, both of them facilitated our implementation work by offering some out-of-the-box features (e.g. Eclipse Markers used in our implementation of change notifications), a clean extension interface (e.g. Eclipse's plug-ins), or a powerful framework based on which we could build our own tool (e.g. ArchStudio's BNA framework). Some problems, however, were also discovered during this process. In this section, we discuss the lessons learned from our implementation work about these two systems. Doing so is part of our reflection upon the implementation, and also serves as a guide for future development on top of these systems.

5.4.1 About Eclipse

As mentioned in Section 5.1.1, Eclipse has two features that play an important role in our implementation, its plug-in mechanism and the Eclipse JET code generation engine. Our experience with the former includes the application of existing plug-ins and development of our own Eclipse plug-ins. Our experience with JET is more comprehensive in the sense that we not only built a code generator on top of the JET engine, but also made extensions to it (e.g. created our own JET tag). The following are some reflections on this experience.

Eclipse is not a single monolithic program, but rather a small kernel containing a plug-in loader surrounded by hundreds of plug-ins. This modular design lends ArchStudio itself to discrete chucks of functionality that can be more readily reused to build applications not envisioned by Eclipse's original developers. In our implementation, we were able to reuse a number of Eclipse plug-ins, which saved much effort. Some of these plug-ins were visible to us and were explicitly used during our implementation, such as Eclipse Markers and Eclipse JET code generation engine, while some other plug-ins were built-in and were used without being noticed, such as SWT, JFace, and so on. In addition to reusing existing plug-ins, we also developed two of our own, a code generator based on JET and a plug-in that extends the JET tag library. This also turned out to be an easy process with the help of the Eclipse wizard. In particular, Eclipse's workbench can be run in different modes: development and runtime. This makes it possible to test and debug plug-ins without having to stop the current Eclipse workbench.

Our main concern with Eclipse plug-in infrastructure is that all Eclipse plug-ins have to be run in the context of Eclipse, which may sound obvious and reasonable. However, there are some functions that users may want to run independently as a standalone application, for example, to run a JET-based code generator independently. At this point, it would be nicer if Eclipse could expose an interface, from which their plug-ins could be called programmatically. In that way, Eclipse plug-ins would be able to run both inside and outside the Eclipse platform. From another perspective, what this implies is that making your application an Eclipse plug-in if and only if it is dependent on or is contributing to Eclipse development platform.

Our experience with the Eclipse JET code generation engine was exactly as what we expected from an open-source project: powerful but poorly documented. The version we used was JET2, a template-based code generation engine having many nice features compared with the old version of JET, such as accessing XML documents with Xpath expression, a tag library, and the capability of reading .java files. At some point, we were even able to create our own JET tag to extend its functionalities. In our implementation, this happened when we wanted to load a Java file with a fully qualified name that was located in a different project from the project where the code generator was run. By default, JET2 can only load Java files in the same project. However, users can easily add new functionalities by creating their own tags, just as we did. This is an important feature of JET2.

The problem with JET2 primarily comes from its documentation. All the information we could get about JET2 is from its website, which maintains a JET Wiki and several user experience articles [44]. Some of them are already out of date. There is currently no detailed user manual or guide about how to use JET2. There is also a JET forum where JET users can post questions, and the JET project owner was generally responsible for answering those questions. The problem is, the same question was often found asked several times given that a complete document about JET is still missing. We believe this is a problem that JET must resolve to obtain more users, especially given its comprehensive code generation capabilities.

5.4.2 About ArchStudio

Extensibility is an important feature of ArchStudio, and this was directly reflected in our implementation of 1.x-way mapping. First of all, the extensions of xADL schema to model state diagrams, sequence diagrams, and architecture changes were relatively straightforward with the provided tools (e.g. Apigen). Not only a data binding library was automatically generated for new schemas, which offers programmatic interfaces for manipulating architecture descriptions, but also the integrated ArchEdit tool can be used to graphically browse the architecture description that supports the new schemas in a tree format. Even the existing xADL schema was also created with the extension under consideration. For example, in the xArch *Java Implementation* XML Schema, there are two elements defined to link a component to its corresponding source code, *mainClass* and *auxClass*. During our implementation of 1.x-way mapping, we could simply use the former to represent our architecture-prescribed code and use the latter to represent the user-defined code without any additional changes made.

Modularity is another favorable feature of ArchStudio. What this meant in our implementation was that all the new xADL schemas we defined were saved independently and were clearly separated from the existing xADL schemas. They do not interfere with existing schema, and can be easily added or modified. Moreover, the implementation of ArchStudio itself is also well organized. The source code is saved into different packages that correspond to different architecture components, which are loosely coupled. This to a great extent facilitated our implementation. For the majority of our implementation, we could just create new components and work independently of other parts of ArchStudio. The only part of the work where we had to look at the existing code was recording architecture changes, which was integrated into the logics of Archipelago. Even so, the process was still relatively easy, given that most modification logics (e.g. *add component*, *remove link*, etc.) were centrally specified in several files that could be easily located from their class names.

xADL 2.0 is accompanied by a number of tools that provide basic capabilities to xADL users: parsing, editing, serializing, low-level editing, and so on. In particular, the data binding library of xADL and its xArchADT wrapper offer a high-level interface (e.g. *addComponent*, *addInterface*) to edit xADL documents, shielding users from underlying XML details (e.g. tags, attributes, etc.). This is recognized as an important benefit of xADL and ArchStudio. However, it would be better if an interface that accepts the *XPath* expression (the XML Path Language, a query language for selecting nodes from an XML document) [152] could be provided. In that way, it would be more efficient for users to parse the xADL document and directly fetch the element of their interest. At this point, all the read/write operations on the xADL document can only go through the data binding library.

Finally, ArchStudio is still a research-oriented tool. It is not fair to compare it with some commercial tools, such as IBM Rational Software Architect, even though we believe ArchStudio in many aspects outperforms these existing products. In particular, the extensibility and modularity of ArchStudio make it an ideal platform to perform architecture related research. Based on our experience, some further improvements can be made to get ArchStudio even better, such as usability and documentation. Some initial work has already been done in this regard [139]. The 1.x-way mapping tool integrated with ArchStudio also partially addresses the usability issue. For example, users do not have to manually write some framework specific code, which is often found tedious and error prone. As to the documentation, it is fine in terms of how to use ArchStudio. In contrast, the document about how to contribute to ArchStudio as a developer is still limited. For example, there is no specific document talking about how to use the BNA4 framework to build additional graphical elements in Archipelago.

6 Experiments and Validations

This chapter is dedicated to the validation of the 1.x-way mapping approach. It starts from a description of the overall objectives of the validation, and then specifically introduces three case studies that address different aspects of the approach. For each of these case studies, its corresponding introduction in this chapter covers the applied evaluation method, collected results, threats to validity, and conclusions that can be made about 1.x-way mapping based on the case study. At the end of the chapter, justification is also made about why the results collected in our validation can be generalized to other real software systems.

6.1 Objectives

The overall purpose of the validation is to validate the hypothesis presented in Chapter 1-1.x-way mapping can be applied in the development of a realistic system to prevent its architecture-prescribed code from being manually changed by programmers, and support automatic mapping of structural and behavioral architecture changes to code. The emphases of the hypothesis are (1) the approach can be applied to a realistic software system; (2) automatic mapping of architecture changes to code and protection of architecture-prescribed code are valid features; (3) the features listed in (2) are applicable to behavioral architecture specifications. Correspondingly, our validation is specifically focused on these three points. We want to ensure that the 1.x-way mapping approach is practical in real software development, and its features really work as presented earlier in the dissertation.

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. Deep separation is the only *cost* that 1.x-way mapping users have to pay for its support for architecture-implementation conformance discussed above. It directly decides the applicability or feasibility of 1.x-way mapping, especially considering that programming patterns and frameworks are widely adopted in current software development. Thus, it is important to validate that deep separation does not compromise or conflict with existing implementation technologies.

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 frequently an architecture change can be mapped to code automatically, semi-automatically, or even manually, and how often - if ever - an accidental change of architecture-prescribed code may happen given that it is supposed to be avoided with 1.x-way mapping. Note that the manual changes to user-defined code for logic completion should not be considered as a violation of the hypothesis. Instead, we still consider 1.x-way mapping as being able to support automatic change mapping as long as it can automatically update corresponding architecture-prescribed code and send relevant change notifications to user-defined code when necessary.

Finally, we evaluate how the involvement of behavioral mapping affects the statistics collected in the second step. 1.x-way mapping will be considered to be able to support behavioral mapping if all the behavioral architecture changes can be mapped to code automatically or semi-automatically depending on specific change types. If manual changes are involved, they should be specifically studied and analyzed to see if they are caused by the design of the 1.x-way mapping approach.

An ideal way to validate all three dimensions mentioned above would be doing a long-term study of how the 1.x-way mapping tool can be used to manage development changes in a real software project. Such a study is currently pending. Instead, we applied the 1.x-way mapping approach to the evolutions of a pre-existing software application, ArchStudio 4, where our approach was implemented and integrated. We refactored the code of ArchStudio with the deep separation mechanism first. The purpose was to (1) validate the applicability of deep separation in a real system and (2) prepare for the subsequent replaying evaluations and enable us to start an experimental development. After that, we replayed the changes made to ArchStudio in two research projects with the help of our 1.x-way mapping approach. That is, we replayed history, but using the 1.x-way mapping version of the system from the refactoring evaluation. The first project we replayed was Architecture-Centric Traceability for Stakeholders (ACTS) [66]. In this project, structural changes had been made to both the architecture and the code of ArchStudio to build and integrate a tool that automatically captures traceability links between architecture and other development artifacts. Replaying its development history offers an opportunity to fully exercise 1.x-way mapping's capability of managing structural architecture and code changes mentioned above. The second project was the development of xMapper – the tool that implements 1.x-way mapping in ArchStudio. This project was special in the sense that behavioral architecture diagrams and changes had been involved, based on which the behavioral mapping feature can be validated for 1.x-way mapping.

6.2 Evaluation I: Deep Separation of ArchStudio 4

ArchStudio 4 is an Eclipse-based architecture development environment that has been developed and maintained by our research group for many years. Its architecture and code are both open for public access. The code of ArchStudio 4 is accompanied by an explicit architecture model that consists of forty components, corresponding to more than 85KSLOC. A primary benefit of exercising 1.x-way mapping with ArchStudio 4 is that it has been extended in several research projects [8, 68, 139], where significant changes were made to its architecture and code. ArchStudio 4 represents an open source system whose development and evolutions were committed independently (e.g. by different people and for different purposes). In particular, the changes made were for a specific task, and happened before 1.x-way mapping was developed. This provides a chance to re-do the changes based on our 1.x-way mapping approach without inducing biases.

Before 1.x-way mapping was applied, the ArchStudio code had to be refactored [93] based on the deep separation mechanism. This is also to answer the first question raised in Section 6.1 about whether deep separation of 1.x-way mapping is applicable to the implementation of a real program of significant complexity. At this point, we believe ArchStudio 4 is qualified because (1) ArchStudio 4 is a software system that is being used at UC Irvine, in several companies and universities. (2) ArchStudio 4 is a relatively complex system that integrates a number of sophisticated tools and can be extended with more. (3) ArchStudio 4 is built on the *myx.fw* architecture framework introduced in Section 2.2.2 and involves extensive use of programming patterns and code libraries, both of which are the norm rather than the exception in current software development. Thus, we can safely draw the conclusion that deep separation is practical with real software systems if we can successfully apply it to the code of ArchStudio 4. Threats to validity exist as well, and they are also specifically discussed later in this section.

6.2.1 Evaluation Method

Figure 6-1 is ArchStudio 4's (build 4.0.5) architecture model as shown in the Archipelago editor. Due to the size of the diagram, the elements are too small to be clearly seen. The point of including this diagram here is to highlight the layered aspect of the architecture and provide an overview of how ArchStudio's components are organized. In the evaluation, we refactored the ArchStudio code in the bottom-up order. In this way, the impact of the refactoring process on the whole system was kept minimal given that each component in the ArchStudio architecture depends on its adjacent upper layer [33].

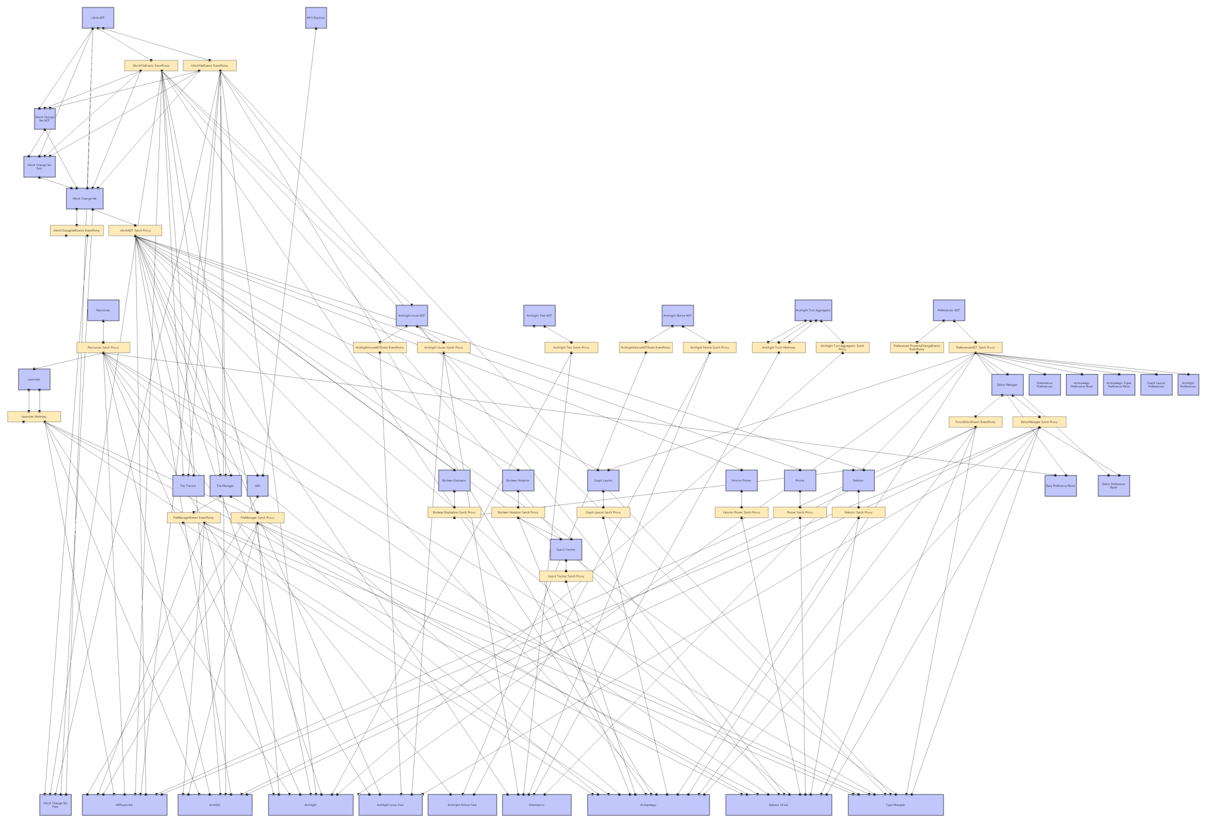


Figure 6-1: ArchStudio 4's architecture.

A tool that can support the process of code refactoring based on deep separation is not yet available. To enable the evaluation, we generated the initial 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 described in Section 4.3. By decoupled, it usually means the process of creating new classes or interfaces, copying and pasting, and changing certain code (e.g. variable names) of existing classes. In some cases, we also needed to update the templates of our code generator to satisfy some special needs as described below.

When problems occurred, for example, the code of some components could not be separated exactly as what is specified by the deep separation due to the application of some technologies, we first tried to modify code generation templates or the configuration panel mentioned in Section 4.4. This was done under the precondition that the changes made do not break the basic principles of deep separation – the architecture-prescribed code and user-defined code of a component are separated into independent elements (classes in this case), and programmer's changes are limited to user-defined code. If the problem remained, we then tried to change the way these components were implemented so that the deep separation mechanism can be finally applied. Again, it must be guaranteed that the same functionality of the component is maintained. At this point, we tried to keep the changes minimal. If neither of the above was applicable, we wrote down the specific problems and did some post hoc analysis to see if these problems were inherent to the design of deep separation, or simply due to the limitations of specific implementation technologies.

We applied deep separation to the implementation of all components in the ArchStudio architecture shown in Figure 6-1. During this process, some similar problems were found and similar solutions ended up being applied. We began to realize that the components could be divided into groups based on how deep separation was applied to them. The code of components in the same group basically could be refactored in the same way for deep separation, and similar strategies were usually used to refactor the code. In addition, we found that the way a component is grouped is closely related with its position in the architecture. Components that are adjacent in the architecture (e.g. located in the same layer) tend to be classified in the same group (i.e. refactored in the same way). For example, components in the bottom layer (e.g. the components that contribute GUI elements) of the architecture were coded in a similar way in ArchStudio, and therefore, their implementations were refactored similarly.

6.2.2 Results

Table 6-1 shows the results of applying deep separation to the code of ArchStudio 4. As we introduced earlier, all the ArchStudio components are divided into different groups based on the strategy taken to refactor their code. Overall, deep separation was successfully applied to the implementation of all the components, although some of them needed special treatment, such as modifications to code generation templates, the configuration panel, or the way the components were implemented. Note that all the changes made were to facilitate the separation process, and none of them were essential to deep separation. Changing templates and the configuration panel was primarily to automate the process of updating architecture-prescribed code, whereas changing the implementation of some components was to make the boundary of its architecture-prescribed code and user-defined code clearer.

Another thing to note is that there were several places where generated code had to be manually edited, for example, to catch/process Java exceptions. This is not recommended in 1.x-way mapping, and can be seen as a compromise we made given current code generation and modeling technologies. However, this does not affect the validity of the deep separation mechanism, 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 programmer's changes are limited to user-defined code. Automatic update of architecture-prescribed code is a final goal, even though it may be hard to completely realize at this point.

#	Components	Problems	Solutions / Comments
1	Pruner, Selector, Version Pruner, Boolean Evaluator, Boolean Notation, Graph Layout, Guard Tracker, Editor Preference Panel, Base Preference Panel, Editor Manager, Schematron Preferences, Archipelago Preference Panel, Archipelago Types Preference Panel, Graph Layout Preferences, Archlight Preferences, Launcher, xArch Change Set Sync	Some variable names (e.g. the references to connected components) did not match those of generated code.	Only some variable names were changed. These components are "good citizens" of deep separation.
2	Archlight Tool Aggregator, Archlight Issue ADT, Preference ADT, Archlight Test ADT, Archlight Notice ADT, File Tracker, File Manager	1. The existence of multiple provided interfaces. 2. Arch-prescribed code had additional interfaces to implement.	Changed the code generation templates and configuration panel.
3	Archlight Issue View, Archlight Notice View, Schematron, xArch Change Set	Implemented Interface IMyxDynamicBrick.	Used a different template.
4	Archipelago, ArchEdit, Archlight, AIMLauncher, Type Wrangler, xArch Change Set View, Selector Driver	1. A special pattern was used to implement these components. 2. Both architecture-prescribed code and user-defined code had to extend some pre-defined classes.	1. Changed the code of the components to make architecture-prescribed and user-defined code clear. 2. Code generator was also changed to address the class extension issue.
5	xArch ADT, AIM, Resources, xArch Change Set ADT	1. Special technology was applied (e.g. Java dynamic proxy class). 2. Reuse of code that was located in different projects.	1. Moved technology-specific code to user-defined part. 2. Moved the reused code to the same project as the arch-prescribed code.

Table 6-1: Results of Evaluation I.

The first group of components shown in the above table can be seen as “good citizens” of deep separation. They did not require any major changes to either the existing code or code generation templates, and deep separation could be easily applied to them. All the changes needed were simply renaming some variables. Close to half of the ArchStudio components fell into this group. On the one hand, this saved us a lot of time and effort in the refactoring process. On the other hand, it reflects the fact that deep separation naturally matches the current implementation of ArchStudio. All these components were developed before 1.x-way mapping, yet most were developed in the spirit of deep separation. Typical examples of components in this group are *Boolean Evaluator*, *Pruner*, *Selector*, and *Editor Manger*. Note that all these components are located in the middle layers of the architecture. They do not interact directly with the user as the components in the bottom layers do, and are not involved with File I/O as the components in the top layers are.

The second group of components could not be refactored as easily as the first group, but they were still successfully handled after we made some changes to either the code generation templates or the configuration panel. Representative components in this group include *File Tracker*, *Archlight Tool Aggregator*, and *Archlight Issue ADT*. They were special primarily because the code generator we first built was not flexible enough to deal with all possibilities during code generation, rather than the deep separation mechanism itself being limited. That said, these components could be potentially moved to the first group with a comprehensive code generator built. For example, some components in this group had multiple provided interfaces, or expected their architecture-prescribed code to implement interfaces that are not specified in the architecture due to the application of certain programming patterns. The former was addressed in our evaluation by making some changes to the code generation templates, while the latter was addressed by allowing users to specify additional interfaces in the configuration panel. Either way, the deep separation mechanism was still successfully applied to these components.

Components in the third group include *Archlight Issue View*, *Archlight Notice View*, *Schematron*. The implementation of these components exploited a special design of the *myx.fw* framework to provide support for runtime dynamism. Their code implemented an interface, *IMyxDynamicBrick*, which requires the implementation of several callback methods so that the framework can notify the code of these components when a link is being connected to or disconnected from their interfaces at runtime. This allows a specific component in the application to support runtime dynamism where it is needed, without complicating components that do not need it [33]. As a result, a different code generation template was used for these components. This also requires our code generator to be highly customizable so that not only specific code generation parameters (e.g. class names) can be changed, but also the template to be used can be specified during the code generation process.

The fourth group primarily consists of components that contribute views, editors, and other GUI elements. Examples include *ArchEdit*, *Archipelago*, *Archlight*, and *Type Wrangler*. These components were implemented in ArchStudio in such a way that addressed a conflict between ArchStudio and Eclipse concerning how the components are instantiated [38]. Simply speaking, there were two classes that implement each architecture component, *xxxEditor* and *xxxMyxComponent* (*xxx* is the name of a specific editor). The former extended the *AbstractArchstudioEditor* class and was loaded by Eclipse, while the later extended *AbstractArchstudioEditorMyxComponent* and was instantiated by ArchStudio. The two base classes encapsulated a mechanism that facilitated the integration of ArchStudio and Eclipse. From the perspective of 1.x-way mapping, both *xxxMyxComponent* and its parent class

AbstractArchstudioEditorMyxComponent could be seen as architecture-prescribed code and were thus processed. The code generator must be highly customizable in this case, so that necessary parameters could be set during the code generation process, including the name of the editor. As to *xxxEditor*, it was seen as user-defined code that extended *AbstractArchstudioEditor*. What was special at this point was how the architecture-prescribed code and user-defined code were integrated. Instead of using the method call mechanism described in Section 4.3, an existing class called *MyxRegistry* was used for integration. It provided the methods of *register*, *map*, and *waitForBrick* that map architecture-prescribed code to its corresponding user-defined code. As a result, deep separation was finally applied to the components in this group, even though changes in both code generator and existing code had to be made during this process.

Finally, the fifth group contains several components that can be seen as special cases, including *Resource*, *AIM*, and *xArch ADT*. *Resource*, for example, used the technology of Java dynamic proxy class that needed to be initialized in a specific way. To refactor its code for deep separation, we moved the technology-specific code to user-defined part, and made initialization customizable during code generation. *AIM* and *xArch ADT* both used existing code as their user-defined code to implement provided operations. In particular, the reused class was located in a separate project. Our first attempt on separating the code gave us a “circle detected in build path” error. After examining the problem, we realized that this was caused by the two plug-in projects, where architecture-prescribed code and user-defined code were located, had to maintain a mutual reference due to the design of deep separation. Our solution was simply moving the reused code to the same project as the architecture-prescribed code. This is further discussed in Section 6.2.4.

6.2.3 Threats to Validity

The threats to validity of our success in applying deep separation include three primary factors. First of all, ArchStudio 4 represents an open-source academic project, where the technologies used are limited to some standard ones, mostly Eclipse technologies. It remains to be seen how deep separation can be used to refactor the code of a proprietary industrial application. The challenges that these systems may bring include the extensive use of accumulated domain-specific code or framework, domain-specific software architecture, or some proprietary development technologies. This is particularly true for those domains that are highly mature and require sophisticated domain-specific knowledge, such as financial planning, traffic scheduling, and so on. A significant amount of legacy code may exist in the software systems of these application areas, and this imposes a challenge to the deep separation mechanism.

Another factor that may endanger the validity of this evaluation is related to the application of *myx.fw* framework in the implementation of ArchStudio 4. One the one hand, the framework provides us an opportunity to fully exercise deep separation with program patterns of the framework, ensuring that they are compatible to each other. On the other hand, the framework to some extent contracts the implementation space given that reusable constructs usually encapsulate certain implementation decisions from its users. In other words, there are less variations of how a component may be implemented with an architecture framework than without a framework. What this means to our deep separation is that a higher requirement on the customizability of the code generator may be imposed. An extreme case could be that some components may not have user-defined code at all, or its architecture-prescribed code contains nothing but a reference to user-defined code.

Finally, my pre-existing knowledge about ArchStudio may be another factor that affects the validity of this evaluation. Before working on 1.x-way mapping, I had the experience of developing a Myx-style lunar lander video game in ArchStudio. Although my role in that project was primarily as an ArchStudio user, instead of an ArchStudio developer or contributor, I still obtained some understandings about ArchStudio, such as how ArchStudio was started and so on. In addition, I also made some modifications to the code of ArchStudio (mostly the *AIM Launcher* tool) to make it support hierarchical architectures. All these may play a role in the success of applying deep separation to the code.

6.2.4 Conclusion

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 comports with 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 the *myx.fw* framework 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 a 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. As discussed earlier, a special implementation strategy was taken in ArchStudio 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. 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 is not compatible with the mutual reference restriction of Eclipse plug-ins. 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 explicitly maintain a mutual reference as illustrated in Section 4.3.1. 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 where deep separation cannot be directly applied. 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.

6.3 Evaluation II: Replaying Changes of ACTS

Architecture-Centric Traceability for Stakeholders (ACTS) is a project that centered on the creation, maintenance, and application of traceability links between software architecture and other artifacts [66]. It tackles issues of catering to multiple stakeholder interests by using custom rules and mashups, and addresses issues that relate to capturing and maintaining links by prospective link capture, concepts from open hypermedia, and rules. As an implementation of the work, a traceability recording and analysis tool was built in ArchStudio and both architecture and code changes were made to ArchStudio. Eleven components were added to the ArchStudio architecture, and around 15KSLOC were written. All the involved architecture changes were structure oriented. The project repository was available for public access at [65].

After the ArchStudio code was refactored for deep separation, we chose to re-do the changes made to ArchStudio in the ACTS project with the help of the 1.x-way mapping approach. That is, we recovered and replayed the development history of ACTS on top of the refactored ArchStudio system. The rationale for choosing ACTS is that its development history was well preserved and extensive architecture changes were involved. An independent branch had been created in the *Subversion* system [28] for this project when it first started, all subsequent development was checked in on a regular basis, and finally the branch was merged into the ArchStudio trunk after the project reached a stable condition. A typical development lifecycle was followed in this project.

6.3.1 Evaluation Method

The process of replaying the development history of a project includes two specific activities: recovering changes and re-doing them with the 1.x-way mapping tool. Before replaying started, we ensured that the evaluation environment was set up appropriately. Figure 6-2 shows the environmental setting of our evaluation, which was based on the Eclipse's development and runtime workbench. The development workbench on the left represents the development environment of the 1.x-way mapping tool. This is a regular Eclipse development environment with the ArchStudio plug-ins installed. Its workspace contained all the ArchStudio packages and the code that we wrote for our mapping tool. The runtime workbench was started through running ArchStudio in the workspace as an Eclipse application. At that point, ArchStudio (including our 1.x-way mapping tool) was automatically installed as plug-ins in the Eclipse runtime workbench. In the workspace of the runtime workbench, we manually imported the ArchStudio packages that were refactored in Evaluation I for deep separation. The evaluation environment was then ready for replaying changes.

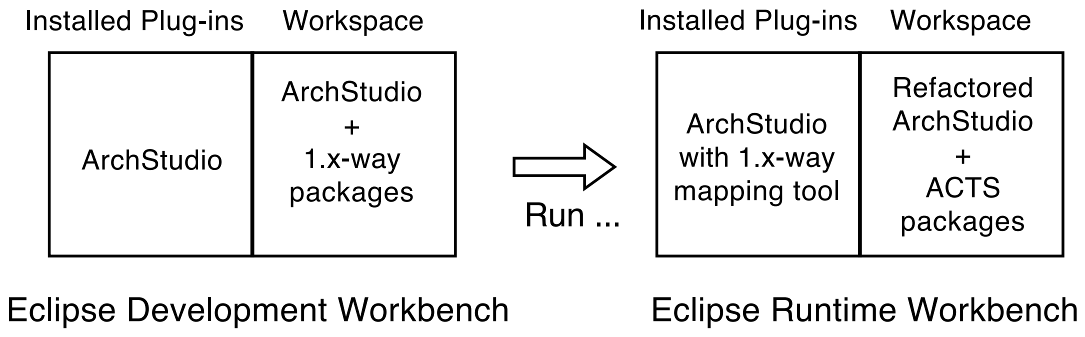
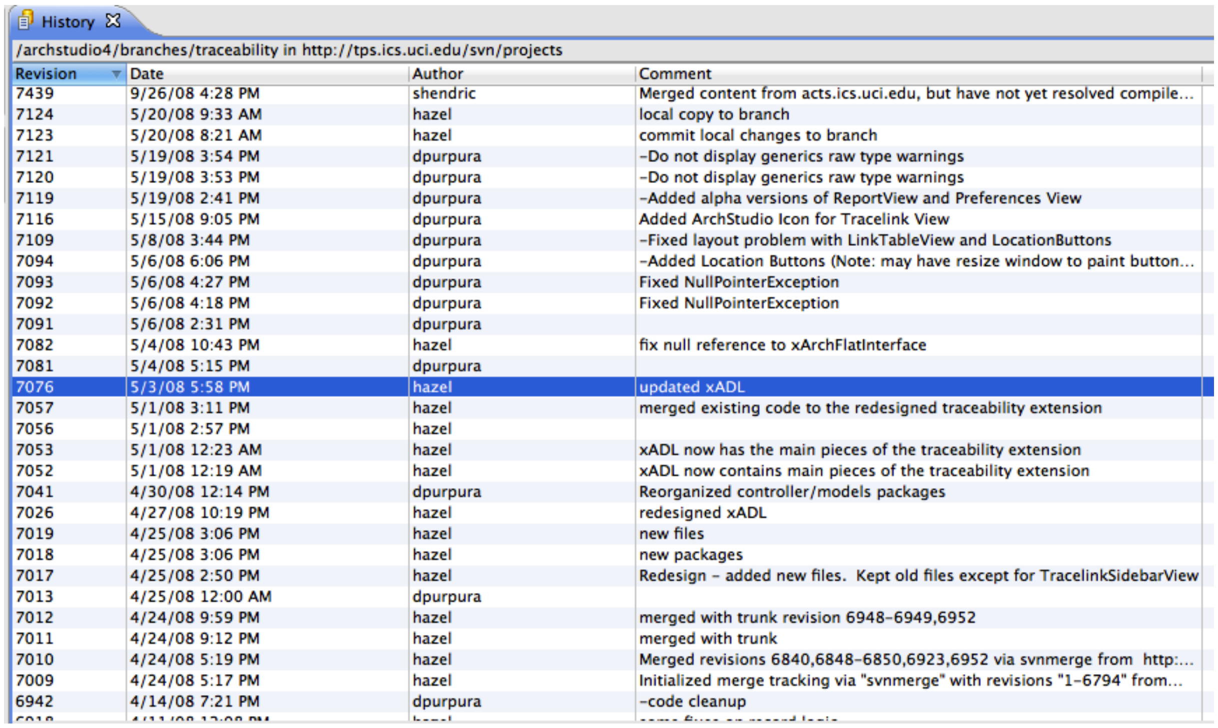

Installed Plug-ins
Workspace
ArchStudio
ArchStudio + 1.x-way packages
Run ...
Installed Plug-ins
Workspace
ArchStudio with 1.x-way mapping tool
Refactored ArchStudio + ACTS packages
Eclipse Development Workbench
Eclipse Runtime Workbench

Figure 6-2: Replaying environment.

With the help of Eclipse's *Subclipse* plug-in [27] and the *Trac* system [132], 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. Figure 6-3 is a screenshot of the project development history shown in *Subclipse*.

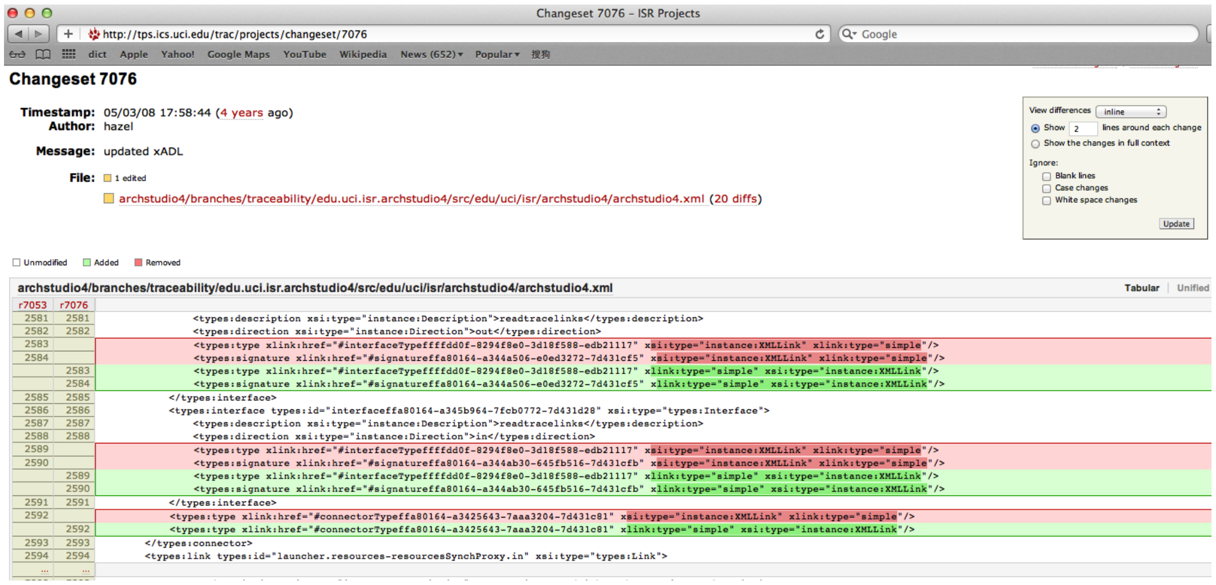
For each of the change entries shown in Figure 6-3, we used the *Trac* system to check details about what specifically was changed. A screenshot of such change details is shown in Figure 6-4. Following this process, all changes checked into the repository could be successfully recovered. Note that our focus was on the architecture changes. In other words, we closely followed the updates made to the ArchStudio xADL document. While for the code changes, many of them were low-level implementation details and were simply ignored. We only cared about the code changes that were associated with an architecture update. Many such code changes were architecture related, such as to implement a component that was just added or update the code of a component that was changed in the architecture.



/archstudio4/branches/traceability in http://tps.ics.uci.edu/svn/projects

Revision	Date	Author	Comment
7439	9/26/08 4:28 PM	shendric	Merged content from acts.ics.uci.edu, but have not yet resolved compile...
7124	5/20/08 9:33 AM	hazel	local copy to branch
7123	5/20/08 8:21 AM	hazel	commit local changes to branch
7121	5/19/08 3:54 PM	dpurpura	-Do not display generics raw type warnings
7120	5/19/08 3:53 PM	dpurpura	-Do not display generics raw type warnings
7119	5/19/08 2:41 PM	dpurpura	-Added alpha versions of ReportView and Preferences View
7116	5/15/08 9:05 PM	dpurpura	Added ArchStudio Icon for Tracelink View
7109	5/8/08 3:44 PM	dpurpura	-Fixed layout problem with LinkTableView and LocationButtons
7094	5/6/08 6:06 PM	dpurpura	-Added Location Buttons (Note: may have resize window to paint button...
7093	5/6/08 4:27 PM	dpurpura	Fixed NullPointerException
7092	5/6/08 4:18 PM	dpurpura	Fixed NullPointerException
7091	5/6/08 2:31 PM	dpurpura	
7082	5/4/08 10:43 PM	hazel	fix null reference to xArchFlatInterface
7081	5/4/08 5:15 PM	dpurpura	
7076	5/3/08 5:58 PM	hazel	updated xADL
7057	5/1/08 3:11 PM	hazel	merged existing code to the redesigned traceability extension
7056	5/1/08 2:57 PM	hazel	
7053	5/1/08 12:23 AM	hazel	xADL now has the main pieces of the traceability extension
7052	5/1/08 12:19 AM	hazel	xADL now contains main pieces of the traceability extension
7041	4/30/08 12:14 PM	dpurpura	Reorganized controller/models packages
7026	4/27/08 10:19 PM	hazel	redesigned xADL
7019	4/25/08 3:06 PM	hazel	new files
7018	4/25/08 3:06 PM	hazel	new packages
7017	4/25/08 2:50 PM	hazel	Redesign - added new files. Kept old files except for TracelinkSidebarView
7013	4/25/08 12:00 AM	dpurpura	
7012	4/24/08 9:59 PM	hazel	merged with trunk revision 6948-6949,6952
7011	4/24/08 9:12 PM	hazel	merged with trunk
7010	4/24/08 5:19 PM	hazel	Merged revisions 6840,6848-6850,6923,6952 via svnmerge from http:...
7009	4/24/08 5:17 PM	hazel	Initialized merge tracking via "svnmerge" with revisions "1-6794" from...
6942	4/14/08 7:21 PM	dpurpura	-code cleanup

Figure 6-3: A screenshot of development history in Subclipse.


Changeset 7076 – ISR Projects
http://tps.ics.uci.edu/trac/projects/changeset/7076
Google
dict
Apple
Yahoo!
Google Maps
YouTube
Wikipedia
News (652)
Popular
搜狗

Changeset 7076

Timestamp: 05/03/08 17:58:44 (4 years ago)
Author: hazel

Message: updated xADL

File: 🟨 1 edited

🟨 archstudio4/branches/traceability/edu.uci.isr.archstudio4/src/edu/uci/isr/archstudio4/archstudio4.xml (20 diffs)

View differences: inline
Show 2 lines around each change
Show the changes in full context
Ignore:
☐ Blank lines
☐ Case changes
☐ White space changes
Update

☐ Unmodified 🟩 Added 🟥 Removed

archstudio4/branches/traceability/edu.uci.isr.archstudio4/src/edu/uci/isr/archstudio4/archstudio4.xml — Tabular | Unified

```
r7053 r7076
2581  2581        <types:description xsi:type="instance:Description">readtracelinks</types:description>
2582  2582        <types:direction xsi:type="instance:Direction">out</types:direction>
2583              <types:type xlink:href="#interfaceTypeffffdd0f-8294f8e0-3d18f588-edb21117" xsi:type="instance:XMLLink" xlink:type="simple"/>
2584              <types:signature xlink:href="#signatureffa80164-a344a506-e0ed3272-7d431cf5" xsi:type="instance:XMLLink" xlink:type="simple"/>
      2583        <types:type xlink:href="#interfaceTypeffffdd0f-8294f8e0-3d18f588-edb21117" xlink:type="simple" xsi:type="instance:XMLLink"/>
      2584        <types:signature xlink:href="#signatureffa80164-a344a506-e0ed3272-7d431cf5" xlink:type="simple" xsi:type="instance:XMLLink"/>
2585  2585      </types:interface>
2586  2586      <types:interface types:id="interfaceffa80164-a345b964-7fcb0772-7d431d28" xsi:type="types:Interface">
2587  2587        <types:description xsi:type="instance:Description">readtracelinks</types:description>
2588  2588        <types:direction xsi:type="instance:Direction">in</types:direction>
2589              <types:type xlink:href="#interfaceTypeffffdd0f-8294f8e0-3d18f588-edb21117" xsi:type="instance:XMLLink" xlink:type="simple"/>
2590              <types:signature xlink:href="#signatureffa80164-a344ab30-645fb516-7d431cfb" xsi:type="instance:XMLLink" xlink:type="simple"/>
      2589        <types:type xlink:href="#interfaceTypeffffdd0f-8294f8e0-3d18f588-edb21117" xlink:type="simple" xsi:type="instance:XMLLink"/>
      2590        <types:signature xlink:href="#signatureffa80164-a344ab30-645fb516-7d431cfb" xlink:type="simple" xsi:type="instance:XMLLink"/>
2591  2591      </types:interface>
2592            <types:type xlink:href="#connectorTypeffa80164-a3425643-7aaa3204-7d431c81" xsi:type="instance:XMLLink" xlink:type="simple"/>
      2592      <types:type xlink:href="#connectorTypeffa80164-a3425643-7aaa3204-7d431c81" xlink:type="simple" xsi:type="instance:XMLLink"/>
2593  2593    </types:connector>
2594  2594    <types:link types:id="launcher.resources-resourcesSynchProxy.in" xsi:type="types:Link">
 ...   ...
```

Figure 6-4: Change details shown in Trac.

After change recovery was done, we manually made all the recovered architecture changes, mapped them to code with the 1.x-way mapping tool, and manually made necessary changes in user-defined code. 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. Figure 6-5 shows the recovered architecture milestones that were successively developed during this process. Note that Component *Trace Criteria View* in version 7026 at some point in the development was removed and does not exist in the later versions.

We also made some changes to our code generator to facilitate the replay of some architecture changes. For example, one change we made was to make our code generator able to load java interface files that were located in a separate project. This was because the ACTS project was created in a separate Eclipse project from ArchStudio, even though ACTS itself was integrated as an ArchStudio tool. Unfortunately, current Eclipse JET tag library does not support this. Thus, we had to extend JET by creating our own tag to do this. Another change we made was for the similar purpose: generating code that is in a different project from the project where code generator is running.

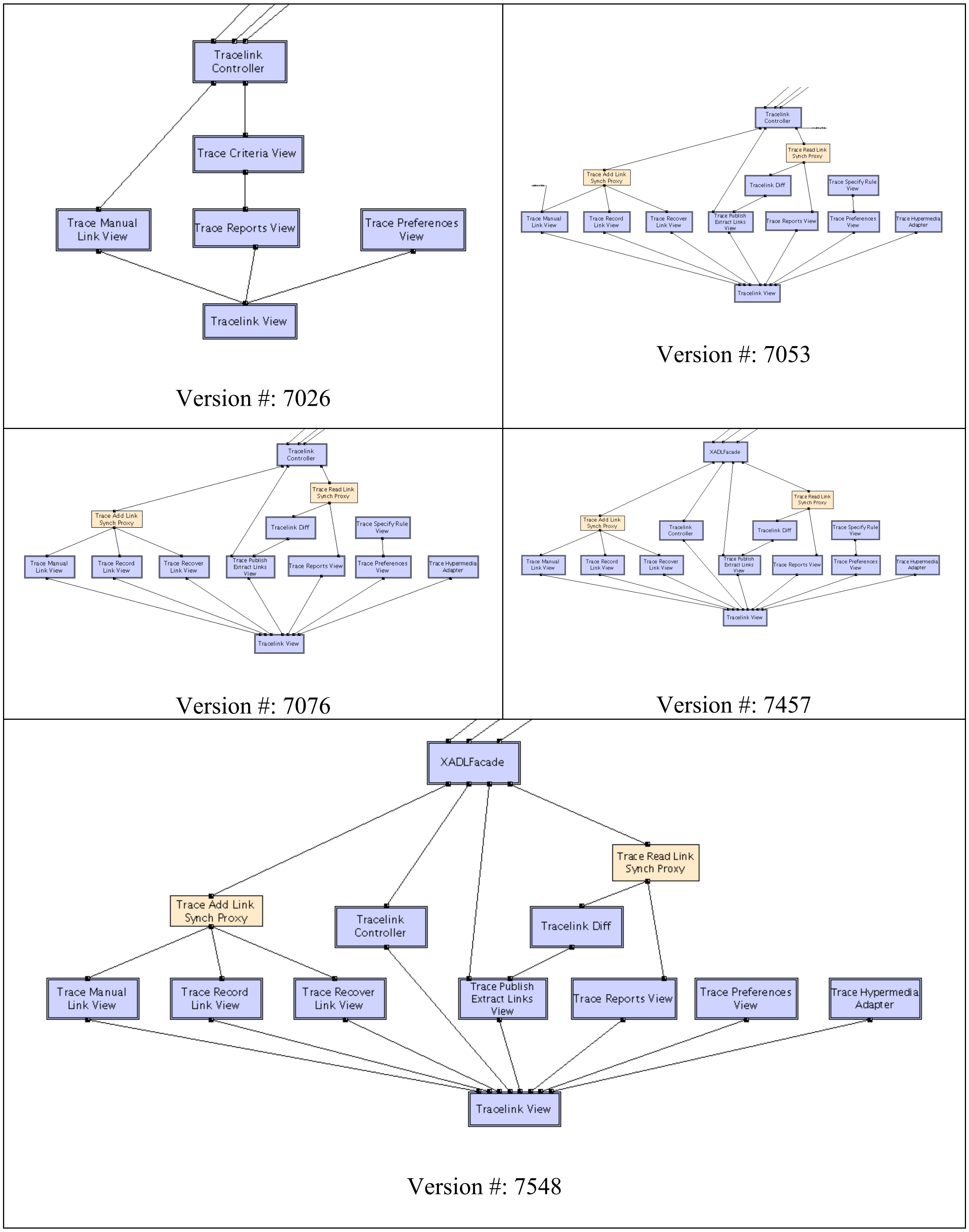


Figure 6-5: Recovered architecture milestones.

6.3.2 Results

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 and appropriate architecture change notifications were sent correspondingly. Table 6-2 shows the details of the evaluation result. It is consistent with what is presented in Figure 4-7.

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

Table 6-2: Results of Evaluation II.

The manual work required for semi-automatically handled changes was primarily to complete application logic. For example, all the *Add Component* changes in the table had to be semi-automatically handled because, naturally, programmers need to work on implementation details of those new components. Note 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 Chapter 5, 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. For example, all the changes (e.g. add interface, edit description, etc.) that were made to a newly added component were simply merged to the "*Add Component*" change that was previously recorded into the architecture change model. When mapping to code, no specific actions were taken for these afterwards changes and only the "*Add Component*" change was explicitly handled. This was based on our change filtering logic discussed in Section 5.2.3. Similarly, all the update changes made to a component that was later removed (e.g. *Trace Criteria View*) were also discarded and were seen as automatically handled.

Compared with architecture changes, code changes were tricky to evaluate, because the ACTS tool as an extension of ArchStudio was also built upon the *myx.fw* framework. The framework reduced the circumstances 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 addressed. Sixteen such errors ended up being found in the implementation of the eleven 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.

During the evaluation, we also found some other mismatches between the architecture and code that we believe were caused by programmers' work in user-defined code. One mismatch that we found, for example, was that the *Trace Publish Extract Links View* component actually did not use its *out* interface at all. In other words, it did not use any resource (e.g.

making any method call) through that interface. This was an error that 1.x-way mapping cannot address at this point, although we believe it was relatively easy to handle with some additional work done (e.g. static program definition/usage analysis) in the future. This kind of *negating* errors, as well as the *inducing* errors discussed in Section 4.4.4, are two kinds of problems that further study will address.

Another problem found through code examination was that the code of the *Trace Preference View* component actually accessed the code of the *Tracelink Controller* component, even though they were not directly connected in the architecture as shown in Figure 6-5. This was a typical *inducing* error mentioned above. What was special in this case is that the reference to the *Tracelink Controller* component was actually passed from *Tracelink View* to *Trace Preference View* in the form of method parameter. Again, this kind of problem can be completely avoided if we require that the architecture-prescribed code of a component can only be accessed by the architecture-prescribed code of other components, as we specifically discussed in Section 4.4.4.

6.3.3 Threats to Validity

There are two primary risks to the validity of this evaluation. One is that we simulated, instead of using a real a software development scenario, by recovering and replaying development changes in a project. The other risk is again related to the application of an architecture framework. Both are discussed and justified below.

Frist, some development changes could not be recovered from examining the commit history of the project repository, and thus failed to be replayed during the evaluation. For example, a single checkin in the project repository could involve the addition of more than one architecture component. As discussed earlier, we could not recover the exact order of how these components were added. Even for a single component, there were also some editing operations that were lost since the repository can only record *snapshots* of the project development. In essence, we recovered the development history based on a series of development *snapshots*. Thus, it was almost impossible to recover a complete history of the project development.

The use of the *myx.fw* framework 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. An architecture framework facilitates the architecture-implementation mapping by providing well understood implementations, which assist developers in implementing systems that conform to the prescriptions and constraints of the architecture. From this perspective, it was inevitable that the discovered code changes that invalidate the architecture would be different from those without using an architecture framework.

We believe our collective results are sustainable despite these risks 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 software frameworks is quite popular in complex software development, not to mention that additional measures were already taken in this regard during our evaluation. Of course, it would require a long-term study of 1.x-way mapping in real software development to completely address these two concerns.

6.3.4 Conclusion

Evaluation II provides results regarding how 1.x-way mapping can be used in the mapping of structural architecture changes to the code. All the recovered architecture changes were replayed and most were automatically mapped to the code using the 1.x-way mapping tool. They all ended up being correctly recorded in the architecture change model. This was verified by a manual examination of the model. For those changes that could not be mapped to code in a completely automatic manner, they were due to missing application logic. At this point, 1.x-way mapping also played a positive role in the sense that it automatically updated the architecture-prescribed code and sent notifications to user-defined code, so that we were able to solely focus on implementation details. During this process, change analysis and refinement greatly facilitated the map-to-code process given that many duplicate or unrelated changes were simply filtered out. It acted as a bridge between the architecture change model and the mapping tool. Significantly, the architecture remained consistent with the code structure after the map-to-code process was done, verified by an ArchStudio tool called *AIM Launcher*. Based on these results, an initial conclusion can be safely drawn that the change management mechanisms of 1.x-way mapping work as designed, and specific kinds of architecture changes can be automatically mapped to code in specific ways. As mentioned earlier, we intend to further validate this through a long-term study with a real software project.

With respect to code changes, the focus of 1.x-way mapping is on the prevention of mistaken changes to architecture-prescribed code. At this point, a number of errors were discovered during our replay of the development history. However, most of these errors were framework specific in the sense that the code simply did not follow the *myx.fw* framework programming rules (e.g. getting the reference though *getFirstRequiredServiceObject*). Only a few errors were direct results of accidentally changing the architecture-prescribed code. A most common case discovered was that the variable name of an interface in the code was not same as what is defined in the architecture. Overall, the accidental changes of architecture-prescribed code in the evaluation were not as many as we expected. We believe this was primarily due to:

(1) the original developers of ACTS were graduate students that were in the architecture area. (2) the versions uploaded to the repository were generally those that were well validated.

As noted earlier, a number of other programmer-induced negative changes (e.g. *negating*, *inducing, etc.*) were also discovered. At this point, all we can say is that 1.x-way mapping makes the discovery of these errors easier since the architecture is accessed in an explicit manner. What can be further done to avoid these errors is as what is suggested in Section 4.4.4, aided with static program analysis. This is one of our future tasks, and we believe 1.x-way mapping has potentials to completely address these issues.

6.4 Evaluation III: Replaying Changes of 1.x-Way Mapping

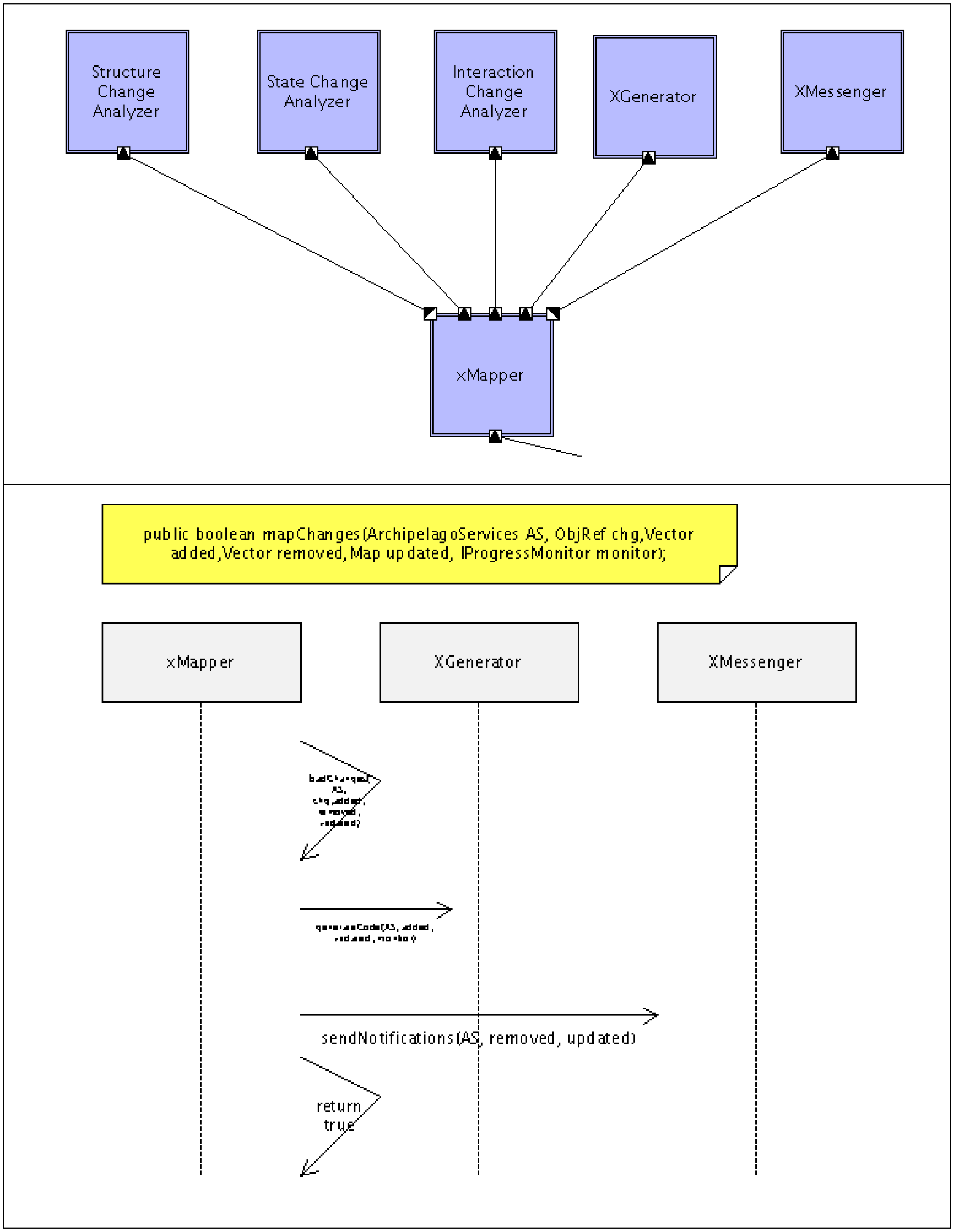
To evaluate how the automation of change mapping would vary when behavioral changes were involved, we remade changes done to ArchStudio in the development of the 1.x-way mapping tool itself with the developed tool. All the changes made to the ArchStudio architecture in previous projects (including the project in Evaluation II) were structure oriented, since ArchStudio itself only had a structural model and support for behavioral modeling (e.g. a fully-featured state diagram editor) was also limited. Here, behavioral changes are also considered.

6.4.1 Evaluation Method

We applied the same evaluation method here as described above with ACTS. Again, the replaying process was based on Eclipse's development and runtime workbench shown in Figure 6-2. The set up in the Eclipse development workbench was exactly same, with ArchStudio installed as plug-in and the developed 1.x-way mapping tool in the workspace. The difference in the runtime workbench was that its workspace now contained the refactored ArchStudio code plus the 1.x-way mapping tool under development. This was also the place where we replayed the development history.

As in Evaluation II, recovering development history in this evaluation involved the usage of the *Subclipse* and *Trac* systems. A separate branch was created in the ArchStudio *Subversion* repository for our 1.x-way mapping when the project began. A difference was that there were detailed notes available in this case since I am the original developer. As a result, a more complete history was recovered - starting from March 2011 when the branch was created to September 2011 when the implementation was completed and a demo video was created. In total, there were sixty commits during this period.

Replaying recovered changes was also similar to what we did with ACTS: we manually made all the recovered architecture changes, mapped them to code with the 1.x-way mapping tool, and again manually made necessary changes in user-defined code. Only in this case a number of adapted behavioral diagrams were involved. Figure 6-6 shows the architecture diagrams of our 1.x-way mapping tool. The sequence diagram in the figure enforced how the mapping tool interacted with the code generator and notifier exactly as described at the beginning of Section 4.2. The code was directly generated from these diagrams and they were kept consistent.


Structure Change Analyzer
State Change Analyzer
Interaction Change Analyzer
XGenerator
XMessenger
xMapper
public boolean mapChanges(ArchipelagoServices AS, ObjRef chg,Vector added,Vector removed,Map updated, IProgressMonitor monitor);
xMapper
XGenerator
XMessenger
loadChanges(AS, chg,added, removed, updated)
generateCode(AS, added, updated, monitor)
sendNotifications(AS, removed, updated)
return true

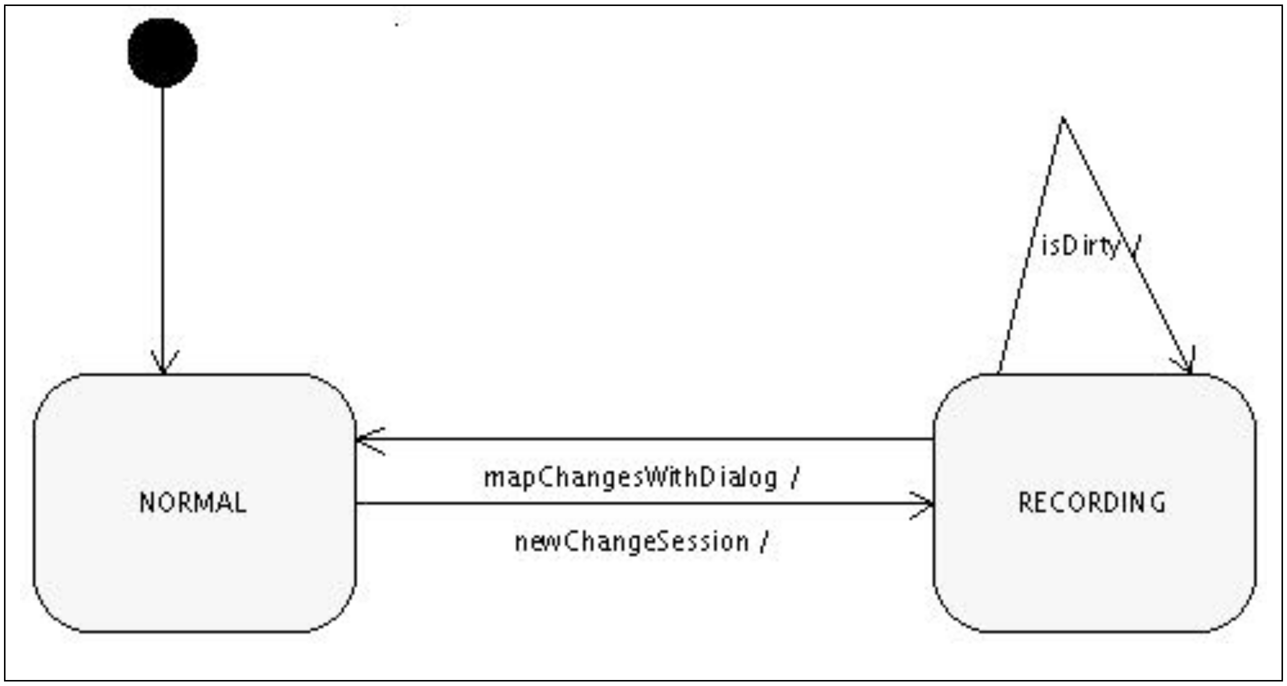


Figure 6-6: Architecture diagrams of 1.x-way mapping.

6.4.2 Results

In the evaluation, 118 architecture changes were recorded. 90 of these changes (about 76 percent) were automatically mapped to code, with the rest semi-automatically handled. Notice that the automation rate was even higher than what occurred when replaying the ACTS project. This was because most behavioral changes actually do not require manual intervention. There was one case in the evaluation that we had to manually respond to a behavioral change when a sequence diagram was removed. Correspondingly, we had to manually implement the specified operation in the user-defined code. In some other cases, we had to manually add Java exception handling statements to the generated code since our diagrams do not support this, and these changes were also classified as semi-automatically handled.

	Auto	Semi-auto	Total
Link Changes	15	_	15
Component Changes	37	22	59
Behavior Changes	38	6	44
Total	90	28	118

Table 6-3: Results of Evaluation III.

Change analysis and refinement as discussed in Section 5.2 are applicable to the behavioral changes as well. Not only recorded behavioral changes are refined, but also the refined behavioral changes are further refined against the component changes. For example, all the behavior changes that are made on a newly added component will be discarded and only an *Add Component* change will be processed in the end. This design also contributed to the high automation rate of behavioral changes during our evaluation.

6.4.3 Threats to Validity

A main threat to the validity of this evaluation is that we replayed the development history of our own project. On the one hand, this self-demonstration further proved the effectiveness of our approach since the same kind of approach can be used to develop itself. On the other hand, however, biases may also be induced. First of all, I was fully aware of the system as its original designer and developer, and chances were relatively low that I would accidentally change architecture-prescribed code. This is also an important reason that code changes were not studied in this evaluation, just to make the whole process more fair. Moreover, I knew the development history already at the beginning of the evaluation. This may not have a direct impact on the evaluation results since all I did was following the commit record and replaying changes. However, it is still possible that at some point I may subconsciously take some actions favorable to the following development.

Another threat concerns that this was still a replaying process. As discussed earlier, it was inevitable that some development activities failed to be recovered, even though it was done with the help of detailed notes in this evaluation. Overall, the missing changes may to some extent affect the specific automation rate, but we do not expect essentially different results to be reached. This was based on the design of 1.x-way mapping and the fact that all types of changes shown in Figure 4-7 were covered in our evaluation. For example, if there were situations where repetitive changes had to be made to the generated code, what could be done was to change our code generator to make it more customizable as what we did in Evaluation I. Finally, note that scalability is not specifically addressed in our evaluation. In other words, it is still to be further validated how 1.x-way mapping performs in practice when the number of models or model changes increases.

6.4.4 Conclusion

Automatic mapping of behavioral architecture and its changes to the code is an important feature of 1.x-way mapping. In particular, we noticed in this evaluation that most behavioral changes actually were mapped to code in a complete manner and no further manual work was required. Change notifications, however, were still sent to user-defined code for the purpose of precaution. For those behavioral changes where users' manual work was involved, most of them were to complete the generated code, instead of addressing consistency issues. As discussed earlier, this was primarily due to the limitations or incompleteness of current behavioral models (e.g. sequence diagrams), not the design of 1.x-way mapping. It is important to note that the conclusion drawn here so far is only applicable to the behavioral models described in Chapter 4 of the dissertation. It is our future work to explore how 1.x-way mapping can be used to support other behavioral models, especially those formal models that are often based on process algebras, such as CSP, Pi-calculus, and so on.

6.5 *Discussion: Generalization of Results*

The evaluations presented in this chapter validate the hypothesis of this dissertation. 1.x-way mapping can be applied in the development of a realistic system to prevent its architecture-prescribed code from being changed by programmers, and support automatic mapping of structural and behavioral architecture changes to code. Note that this was only done in the context of ArchStudio. In other words, what has been validated is that the hypothesis about 1.x-way mapping is true with the ArchStudio system and its derivative systems, ACTS and xMapper. Below we discuss why we believe the collected results can be generalized to other real applications.

First of all, most technologies used in the development of the ArchStudio system are well understood and have been widely used in other existing systems, such as the Java programming language, the component-and-connector architecture model, the Eclipse platform, and the XML technologies. Compared with them, the *myx.fw* framework that ArchStudio was built upon may not be common. However, specific concepts and patterns that the framework includes (e.g. class inheritance, callback lifecycle methods, etc.) are widely adopted as well. In particular, the use of frameworks, middleware, or domain-specific architectures is becoming increasingly important in today's software development. From this perspective, we believe the conclusions made in the evaluations, such as how deep separation can facilitate the use of a framework in Evaluation I, and automatic mapping of architecture changes in Evaluation II and III, are not limited to the ArchStudio system.

Second, complexity, changeability, conformity, and invisibility are identified as four essential properties of modern software systems [16]. We believe ArchStudio is also representative in terms of these properties. As mentioned earlier, ArchStudio consists of around 85KSLOC and forty architecture components. Its implementation involves the activities of File I/O, GUIs, code generation, and dynamic instantiation. In addition, ArchStudio has been extended in several research projects, where significant changes were made to its architecture and code. Significantly, the development and evolutions of ArchStudio were committed independently (e.g. by different people and for different purposes). ArchStudio is currently used in a couple of universities and companies. Thus, it is also under constant pressure of conforming to new needs of ArchStudio users. Invisibility is an inherent property of all software systems, and ArchStudio is not the exception.

Finally, the previous research experience with ArchStudio in our group reveals the effectiveness of ArchStudio both as a development platform and as a case study example in software architecture research. A number of important research results have been generated and validated based on the ArchStudio system, including software traceability [8], dynamic adaptation [59], product line architectures [68], and software security [121]. On the one hand, ArchStudio plays an important role in fostering and validating these research results; on the other hand, the success of the corresponding research further proves the generalization of ArchStudio-based research results, especially in the area of software architecture.

7 Conclusion

In this study, a new architecture-implementation mapping approach, 1.x-way mapping, is developed to maintain conformance between software architecture and code during development. The focus of the work is regulating the implementation of software architecture with a code separation mechanism, explicit modeling of architecture changes, automatic regeneration of architecture-prescribed code, and sending architecture change notifications to user-defined code. This approach has a number of desirable properties, including suppression of mistaken changes of architecture-prescribed code, automatic mapping of specific kinds of architecture changes to code in specific ways, and support for the mapping of behavioral architecture specifications to code.

7.1 Summary

This research tackles the issue of maintaining architecture-implementation conformance during software development. This is essential to architecture-centric software development, but fails to be addressed by existing approaches. Current architecture-implementation mapping approaches are deficient in that change mapping between architecture and code is weakly supported and most approaches are structure-oriented only. This is partially due to architecture being implemented in ad hoc ways, and architecture-prescribed code is mixed with implementation details. In particular, no explicit change management mechanism is provided to either regulate or analyze changes that are made to the two types of artifacts. As a result, significant overhead is incurred in architecture-centric development to manually maintain architecture-implementation conformance – overhead that few are willing to bear.

1.x-way mapping addresses the issues of change management and behavioral mapping to maintain architecture-implementation conformance. It regulates the implementation of software architecture by separating architecture-prescribed code and user-defined details of each architecture component into two independent program elements. This is called deep (linguistic) separation. Based on it, manual changes made by programmers are limited to user-defined code, and cannot contaminate architecture-prescribed code. In this way, architecture-implementation conformance is enhanced. The complexity of reverse engineering and roundtrip engineering is also avoided. In addition, deep separation enables support for behavioral architecture-implementation mapping with system dynamics modeled as UML-like sequence diagrams and state diagrams, from which code can be automatically generated in a way that maintains code separation.

With regard to architecture changes that break architecture-implementation conformance, an architecture change model is maintained in 1.x-way mapping to automatically record and classify various architecture changes. The recorded changes are organized into different change sessions, each of which comprises a list of specific changes and is mapped to code as a unit. All the changes in a change session are automatically mapped to code by completely regenerating architecture-prescribed code of modified components, with the code of other components not affected. For architecture changes that may affect user-defined code, architecture change notifications are also generated and delivered to corresponding user-defined code. In particular, change analysis and refinement is enforced during this process so that changes that together have no impact on the code can be automatically filtered away. By this means, not only unnecessary mappings are avoided, but also mappings that should be made are automated in specific ways.

1.x-way mapping is implemented and integrated as a tool called xMapper in ArchStudio 4, an Eclipse-based architecture development environment. The modularity and extensibility of ArchStudio and Eclipse played a positive role during the implementation. These two tools not only are easy to extend with new capabilities, but also provide a number of technologies, such as the JET code generation engine, Eclipse Markers, and the BNA framework, that were directly exploited in our implementation. To validate the utility of 1.x-way mapping, we applied it to the code and evolutions of ArchStudio 4. Specifically, we refactored the code of ArchStudio and replayed changes that had been made to ArchStudio in two research projects by redoing them with xMapper. The results show that (1) deep separation of 1.x-way mapping is applicable to the implementation of a program of significant complexity; (2) most architecture changes can be mapped to code in a completely automatic manner with the help of 1.x-way mapping, and the rest is semi-automatically handled; (3) the extensive application of 1.x-way mapping in complex software development, however, requires further development of modeling and code generation technologies.

7.2 Future Work

Architecture-implementation mapping directly determines the degree to which software architecture can be used in development to improve software productivity and quality. We believe that the 1.x-way mapping approach developed in this study can bring new power to some architecture-centric development activities, including architecture-based dynamic adaptation, product line architectures, and advanced architecture change management. Below some future enhancements and potential applications of 1.x-way mapping are identified.

7.2.1 Remaining Challenges of Architecture-Implementation Mapping

1.x-way mapping focuses on protecting architecture-prescribed code and mapping architecture changes to code to maintain architecture-implementation conformance. One remaining challenge, however, is how to prevent programmers' work in user-defined code from invalidating the architecture. As discussed earlier, user-defined code may invalidate the architecture by inducing new negative properties or negating existing architecture elements without using them. For example, the user-defined code of a component may reference another component that this component is not connected to, or the code may not use services provided by a connected component at all. Either way, it is hard to map this back to the architecture during software development, especially considering that the code-to-architecture mapping actually conflicts with the principle of architecture-centrality. We believe 1.x-way mapping has the potential to completely address this problem. As discussed in Section 4.4.4, what can be done is to enforce that user-defined code of a component be only accessed by its architecture-prescribed code based on the deep separation mechanism. In this way, illegal accesses of the user-defined code from other components are avoided.

Another potential enhancement to 1.x-way mapping is to send change requests, instead of change notifications, when user-defined code has to be modified in response to certain architecture changes. As discussed in Section 4.4.3, a change request describes what needs to be changed in user-defined code and provides more information regarding what to modify to accomplish a change than a simple change notification. Sending a change request is also feasible because of the way architecture resources are used in user-defined code of 1.x-way mapping: all accesses go through a single reference to the architecture. Static program analysis can be applied in this case.

7.2.2 Architecture-based Dynamic Adaptation

Dynamic adaptation refers to the capability of a software system that can modify its own behavior in response to changes in its operating environment (e.g. end-user input, external sensors, etc.) [116]. Architecture-based adaptation brings promising results in this regard. This is an approach where changes are first formulated in, and reasoned over, an explicit architectural model when the environment changes. Changes to the architectural model (usually at the level of components and connectors) are reflected in modifications to the application's implementation, while ensuring that the model and the implementation are consistent with one another. It is at this point that 1.x-way mapping has the potential to play a significant role by dynamically mapping architecture changes to code.

This can be performed in two specific ways. First, the architecture-prescribed code of an involved component can be regenerated, recompiled, and reloaded at runtime, without requiring user-defined code to be changed. Given that architecture-prescribed code may include the implementation of system dynamics, which is elaborated in Section 4.5, it becomes possible to dynamically associate a certain behavior with a component or several components. This is a significant improvement over existing architecture-based adaptation mechanisms, which usually can only support structural adaptations (adding components, links, etc.). Second, switches can be made at runtime between alternatives of user-defined implementations (e.g. using different system libraries) for an involved component. This time architecture-prescribed code remains stable, and its requests are dynamically redirected to different user-defined implementations based on the code integration framework presented in Section 4.3.2. By this means, a finer degree of granularity is enabled for dynamic adaptation.

Another potential benefit that 1.x-way mapping brings to architecture-based adaptation is its architecture change model. It provides a decent form to organize descriptions of formulated architecture changes. Specific issues that need to be addressed for dynamic adaptation include protecting integrity of adapted systems and identifying quiescent states when adaptation can safely occur. 1.x-way mapping induces additional difficulties at this point with one more layer of indirection in the implementation of each architecture component.

7.2.3 Implementation of Product Line Architectures

Keeping the cost of software changes low is another requirement in software evolution. This is particularly emphasized when making changes that are anticipated before system development starts. Anticipated changes usually occur when developing a family of software products, or a product line. For example, producing a new software product simply by extending a related existing product (e.g. adding an optional capability or customizing for different platforms). At this point, being able to reuse existing code that encapsulates domain, business, and technology information as much as possible becomes important. The use of product lines has become a principled form of software reuse over the past decade [149]. This is partially due to the application of product line architectures (PLAs), an architecture-centric approach to product lines. A PLA explicitly specifies variation points (e.g. optional and alternative elements) inside the reference architecture of an entire product line to differentiate products. Implementing a PLA is also a mapping problem, except that multiple products composed of core elements and variation points are involved. During this process, it is important that separation of concerns can be achieved among different component implementations as it is in the architecture. Otherwise extensive changes have to be made to the code of existing components to introduce variations, and software reusability is compromised. However, separating concerns in the implementation artifacts along preferred boundaries involves significant challenges.

We believe 1.x-way mapping can help to address the implementation problem of PLAs. On the one hand, architecture-prescribed code in 1.x-way mapping does not include details (e.g. platform specifics, domain knowledge, algorithm, system library usage, etc.) regarding how a programmer implements an operation, and thus, is sustainable to variations of these implementation specific concerns. Libraries of architecture implementations can be constructed from the same set of operations provided by user-defined code. On the other hand, user-defined code is relatively independent of architecture-prescribed code as well. It does not contain knowledge about architecture topology and message exchange among components, which are encapsulated in architecture-prescribed code. As a result, a separation of decision space is achieved within the implementation of each component, and both parts can vary independently. This has two implications to current PLA implementations. First, more variations are enabled for a PLA. Traditionally, variation points in a PLA are expressed at the level of components and connections. With 1.x-way mapping, they can be refined into individual components. For example, the usage of a different signal-processing algorithm can be specified as a variation of a component, which corresponds to different user-defined code. Or, a component can be customized with an optional interface. At this point, architecture-prescribed code may be regenerated and used without necessarily changing user-defined code. Second, the code reusability of each architecture component also increases. Since it is hard to completely separate concerns among component implementations as discussed above, 1.x-way mapping makes it possible to reuse a portion of an existing component's code (e.g. architecture-prescribed code or user-defined code), instead of recoding the whole component to introduce variations.

7.2.4 Advanced Architecture Change Management

The architecture change model developed in 1.x-way mapping provides information for architecture-based code regeneration and architecture change notifications, both of which are specific to architecture-implementation mapping. We believe more advanced change management activities are enabled based on the architecture change model, including parallel change sessions, change visualization, and change replay.

Parallel change session means multiple (*unmapped* or *open*) change sessions can co-exist in the architecture change model. Each session contains changes that modify different portions of the architecture, made by different people for different purposes, and be mapped to the code independently. Moreover, users can even switch among different change sessions (e.g. for different tasks). The architecture changes they made will correspondingly be recorded into different change sessions, and concurrent modification of the architecture model is potentially supported. One challenge to be addressed, however, is managing the relationships (e.g. mutual exclusion, dependency) between concurrent change sessions.

Another possible application based on architecture change model is to visualize the changes of each change session. For example, selecting a change session on the left panel of Archipelago displays all the architecture elements that are changed in that session in the editor panel of Archipelago. Some additional information related with the changes may also be shown, such as change time, author, and comments made. Users can then further select what to do with each specific change session, such as redo/undo. This feature is especially useful in an architecture-centric development environment, where architecture plays a central role in the development lifecycle.

In essence, the architecture change model makes architecture changes an independent artifact. From this perspective, an architecture change model can be shared, analyzed, or reused just like some other software artifacts (e.g. requirements specification). For example, an architecture change model can be analyzed to see if a specific change session in it induces any errors. Or we can reuse a change session by replaying all the included changes to an architecture. In this way, people can work concurrently on the same architecture simply by exchanging or sharing the associated architecture change model. All these represent potential applications of the architecture change model in 1.x-way mapping.

8 Reference

[1] Agha, G. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, 1986.

[2] Aizenbud-Reshef, N., Nolan, B.T., et al. Model Traceability. IBM Systems Journal. 45(3), p. 515-26, 2006.

[3] Aldrich, J., Chambers, C., et al. ArchJava: Connecting Software Architecture to Implementation. In Proceedings of the 24th International Conference on Software Engineering. p. 187-197, ACM. Orlando, FL, 2002.

[4] Allen, R. A Formal Approach to Software Architecture. Ph.D. Thesis. Carnegie Mellon University, p. 248, 1997. http://reports-archive.adm.cs.cmu.edu/anon/1997/CMU-CS-97-144.pdf.

[5] Arnold, R.S. Software Change Impact Analysis. 376 pgs., IEEE Computer Society Press, 1996.

[6] Aßmann, U. Automatic roundtrip engineering. Electron Notes Theor Comput Sci (ENTCS). 82(5), p. 1–9, 2003.

[7] Asuncion, H., François, F., et al. An End-To-End Industrial Software Traceability Tool. In Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering (ESEC/FSE). p. 115-124, Dubrovnik,Croatia, Sept 3-7, 2007.

[8] Asuncion, H.U., Asuncion, A.U., et al. Software traceability with topic modeling. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. p. 95-104, ACM: Cape Town, South Africa, 2010.

[9] Atkinson, C. and Kuhne, T. Model-Driven Development: A Metamodeling Foundation. IEEE Software. 20(5), 2003.

[10] Balzer, R. A 15 Year Perspective on Automatic Programming. IEEE Trans. Softw. Eng. 11(11), 1985.

[11] Bass, L. and Kazman, R. Architecture-Based Development. Carnegie Mellon University, Technical Report Report CMU/SEI-99-TR-007, April 1999, 1999.

[12] Batory, D. Scaling Step-Wise Refinement. In the 25th International Conference on Software Engineering. p. 187 - 197: Portland, Oregon 2003.

[13] Bernstein, P.A. Middleware: a model for distributed system services. Commun. ACM. 39(2), p. 86-98, 1996.

[14] Boocock, P. The Jamda Project. http://jamda.sourceforge.net/.

[15] Bowman, I.T., Holt, R.C., et al. Linux as a Case Study: Its Extracted Software Architecutre. In Proceedings of the 21st International Conference on Software Engineering (ICSE'99). Los Angeles, CA, May 16-22, 1999.

[16] Brooks, F.P. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer. April, 1987.

[17] Brown, A. An introduction to Model Driven Architecture. http://www.ibm.com/developerworks/rational/library/3100.html, 2004.

[18] Budinsky, F., Finnie, M., et al. Automatic Code Generation from Design Patterns. IBM Systems Journal. 35(2), p. 151--171, 1996.

[19] Carzaniga, A., Rosenblum, D.S., et al. Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems. 9(3), p. 332-383, August, 2001.
[20] Cassou, D., Balland, E., et al. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In Proceeding of the 33rd International Conference on Software Engineering. p. 431-440, ACM: Waikiki, Honolulu, HI, USA, 2011.
[21] Chalabine, M. and Kessler, C. A Formal Framework for Automated Round-Trip Software Engineering in Static Aspect Weaving and Transformations. In Proceedings of the 29th international conference on Software EngineeringIEEE Computer Society, 2007.
[22] Chikofsky, E.J. and Cross II, J.H. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software. 7(1), p. 13-17, January/February, 1990.
[23] Clayberg, E. and Rubel, D. Eclipse Plug-ins. 3 ed. Addison-Wesley Professional, 2008.
[24] Cleaveland, J.C. Building Application Generators. IEEE Software. 5(4), p. 25 - 33, 1988.
[25] Cleaveland, J.C. Program Generators with XML and Java. 448 pgs., Prentice-Hall, 2001.
[26] Clements, P., Bachmann, F., et al. Documenting Software Architectures: Views and Beyond. Addison Wesley, 2002.
[27] CollabNet. Subclipse. http://subclipse.tigris.org/, HTML, 2004.
[28] Collins-Sussman, B., Fitzpatrick, B.W., et al. Version Control with Subversion. http://svnbook.red-bean.com/, HTML, 2004.
[29] Cook, S. Domain-Specific Modeling and Model Driven Architecture. MDA Journal. January 2004, 2004.
[30] Czarnecki, K. and Eisenecker, U.W. Components and generative programming (invited paper). ACM SIGSOFT Software Engineering Notes. 24(6), p. 2 - 19, 1999.
[31] Czarnecki, K. and Helsen, S. Classification of Model Transformation Approaches. In Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture: Anaheim, California, USA., 2003.
[32] Dashofy, E., van der Hoek, A., et al. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. ACM Transactions on Software Engineering and Methodology (TOSEM). 14(2), p. 199-245, April, 2005.
[33] Dashofy, E. The Myx Architectural Style. University of California, Irvine, Whitepaper Report, 2006.
[34] Dashofy, E. Myx and myx.fw. http://www.isr.uci.edu/projects/archstudio/myx.html, 2008.
[35] Dashofy, E.M., Medvidovic, N., et al. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In Proceedings of the 21st International Conference on Software Engineering (ICSE'99). p. 3-12, Los Angeles, CA, May 16-22, 1999.
[36] Dashofy, E.M., van der Hoek, A., et al. A Highly-Extensible, XML-Based Architecture Description Language. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). Amsterdam, The Netherlands, August 28-31, 2001.
[37] Dashofy, E.M., van der Hoek, A., et al. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In Proceedings of the 24th International Conference on Software Engineering (ICSE 2002). p. 266-276, ACM. Orlando, Florida, May, 2002.

[38] Dashofy, E.M. Supporting Stakeholder-Driven, Multi-View Software Architecture Modeling. Ph.D. Thesis. Information and Computer Science, University of California, Irvine, p. 294, 2007.

[39] Dashofy, E.M., Asuncion, H., et al. ArchStudio 4: An Architecture-Based Meta-Modeling Environment. In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007). Informal Research Demonstrations, Companion Volume, p. 67-68, Minneapolis, MN, May 20-26, 2007.

[40] Denno, P., Steves, M.P., et al. Model-Driven Integration Using Existing Models. IEEE Software. 20(5), 2003.

[41] DeRemer, F. and Kron, H.H. Programming-in-the-Large versus Programming-in-the-Small. IEEE Transactions on Software Engineering. 2(2), p. 80-86, June, 1976.

[42] Diaz-Pace, J.A., Carlino, J.P., et al. Assisting the synchronization of UCM-based architectural documentation with implementation. In Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009., 2009.

[43] Doug Brown, J.L., Tony Mason. lex & yacc. O'Reilly Media, 1992.

[44] Eclipse. Eclipse JET Project. http://www.eclipse.org/modeling/m2t/?project=jet.

[45] Eclipse. ATL - a model transformation technology. http://www.eclipse.org/atl/.

[46] Engels, G., Hücking, R., et al. UML collaboration diagrams and their transformation to java. In Proceedings of the 2nd international conference on The unified modeling language: beyond the standard. p. 473-488, Springer-Verlag: Fort Collins, CO, USA, 1999.

[47] Feiler, P.H., Gluch, D.P., et al. The Architecture Analysis & Design Language (AADL): An Introduction. CMU/SEI-2006-TN-011, Report, 2006.

[48] Fickas, S.F. Automating the Transformational Development of Software. IEEE Trans. Softw. Eng. 11(11), 1985.

[49] Fielding, R.T. and Taylor, R.N. Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology (TOIT). 2(2), p. 115-150, May, 2002.

[50] Fowler, M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. Addison Wesley: Reading, MA, 2003.

[51] Fowler, M. UmlMode. http://martinfowler.com/bliki/UmlMode.html, 2003.

[52] France, R. and Rumpe, B. Model-driven Development of Complex Systems: A Research Roadmap. In Future of Software Engineering 2007, Briand, L. and Wolf, A. eds. IEEE-CS Press, 2007.

[53] Gamma, E., Helm, R., et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley Professional: Reading, MA, 1995.

[54] Garlan, D. and Shaw, M. An Introduction to Software Architecture. In Advances in Software Engineering and Knowledge Engineering, Ambriola, V. and Tortora, G. eds. p. 1-39, World Scientific Publishing Company: Singapore, 1993.

[55] Garlan, D., Allen, R., et al. Architectural Mismatch: Why Reuse Is So Hard. IEEE Software. 12(6), p. 17-26, November, 1995.

[56] Garlan, D. Style-Based Refinement for Software Architecture. In Proceedings of the Second International Software Architecture Workshop (ISAW-2). p. 72-75, San Francisco, CA, October, 1996.

[57] Garlan, D., Monroe, R.T., et al. ACME: An Architecture Description Interchange Language. In Proceedings of the CASCON '97. p. 169-183, IBM Center for Advanced Studies. Toronto, Ontario, Canada, November, 1997.
[58] Georgas, J.C., Dashofy, E.M., et al. Architecture-Centric Development: A Different Approach to Software Engineering. ACM Crossroads, issue on Software Engineering. 12(4), Summer, 2006.
[59] Georgas, J.C. and Taylor, R.N. Policy-Based Self-Adaptive Architectures: A Feasibility Study in the Robotics Domain. In Proceedings of the ACM/IEEE International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008), held in conjunction with ICSE 2008. Leipzig, Germany, May 12-13, 2008.
[60] GNU. Bison - GNU parser generator. http://www.gnu.org/software/bison/.
[61] Greenfield, J., Short, K., et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley; 1st edition, 2004.
[62] Gruber, O., Hargrave, B.J., et al. The Eclipse 3.0 platform: adopting OSGi technology. IBM Systems Journal. 44(2), p. 289-299, July, 2005.
[63] Hailpern, B. and Tarr, P. Model-driven development: The good, the bad, and the ugly. IBM Systems Journal. 45(3), 2006.
[64] Hayes-Roth, B., Pfleger, K., et al. A Domain-Specific Software Architecture for Adaptive Intelligent Systems. IEEE Transactions on Software Engineering. 21(4), p. 288-301, April, 1995.
[65] Hazeline, U.A. Architecture-Centric Traceability for Stakeholders (ACTS). http://tps.ics.uci.edu/svn/projects/archstudio4/branches/traceability/.
[66] Hazeline, U.A. Architecture-Centric Traceability for Stakeholders (ACTS). Thesis. Information and Computer Science, University of Califorina, Irvine, 2009.
[67] Heineman, G.T. and Councill, W.T. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley: Reading, Massachusetts, 2001.
[68] Hendrickson, S.A. and van der Hoek, A. Modeling Product Line Architectures through Change Sets and Relationships. In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007). p. 189-198, Minneapolis, MN, May 20-26, 2007.
[69] Herrington, J. Code Generation in Action. Manning Publications Co., 2003.
[70] Hoare, C.A.R. Communicating sequential processes. viii+256 pgs., Prentice-Hall, 1985.
[71] Holzmann, G.J. The Model Checker SPIN. IEEE Transactions on Software Engineering. 23(5), p. 279-295, May, 1997.
[72] Hunt, A. and Thomas, D. The pragmatic programmer: from journeyman to master. 321 pgs., Addison-Wesley Longman Publishing Co., Inc., 1999.
[73] IBM. IBM Rational Software Development Platform. http://www-01.ibm.com/software/info/developer/faq.jsp.
[74] Jackson, D. and Damon, C.A. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. IEEE Transactions on Software Engineering. 22(7), p. 484-495, 1996.
[75] Johnson, R. Frameworks = Components + Patterns. Communications of the ACM. 40(10), p. 39 - 42, October, 1997.
[76] Kadia, R. Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project. In Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments. p. 169-180, ACM Press. Tyson's Corner, Virginia, United States, December 9-11, 1992.

[77] Kang, K.C., Kim, S., et al. FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering. 5, p. 143-168, 1998.
[78] Karsai, G., Sztipanovits, J., et al. Model-Integrated Development of Embedded Software. Proceedings of the IEEE. 91(1), p. 145-164, 2003.
[79] Kelly, S. and Tolvanen, J.-P. Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press., 2008.
[80] Kiczales, G., Lamping, J., et al. Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97). p. 220-242, Jyväskylä, Finland, June 9-13, 1997.
[81] Kleppe, A., Warmer, J., et al. MDA Explained: The Model Driven Architecture: Practice and Promise. 192 pgs., Addison-Wesley Professional, 2003.
[82] Krueger, C.W. Software Reuse. ACM Computing Surveys. 24(2), p. 131-183, 1992.
[83] Ledeczi, A., Balogh, G., et al. Model Integrated Computing in the Large. In Aerospace Conference. p. 1-8, 2005.
[84] Lopez, N., Casallas, R., et al. Issues in mapping change-based product line architectures to configuration management systems. In Proceedings of the 13th International Software Product Line Conference. p. 21-30, Carnegie Mellon University: San Francisco, California, 2009.
[85] Luckham, D.C., Kenney, J.J., et al. Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering. 21(4), p. 336-355, April, 1995.
[86] Magee, J., Dulay, N., et al. Specifying Distributed Software Architectures. In Proceedings of the 5th European Software Engineering Conference (ESEC 95). 989, p. 137-153, Springer-Verlag, Berlin. 1995.
[87] Medvidovic, N. and Taylor, R.N. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering. 26(1), p. 70-93, January, 2000. Reprinted in Rational Developer Network: Seminal Papers on Software Architecture. Rational Software Corporation, <http://www.rational.net/%3E, 2001.
[88] Medvidovic, N., Mehta, N.R., et al. A Family of Software Architecture Implementation Frameworks. In Proceedings of the 3rd IFIP Working International Conference on Software Architectures. Montreal, Canada, August, 2002.
[89] Medvidovic, N., Rosenblum, D.S., et al. Modeling Software Architectures in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology (TOSEM). 11(1), p. 2-57, January, 2002.
[90] Medvidovic, N., Gruenbacher, P., et al. Bridging Models across the Software Lifecycle. Journal of Systems and Software. 68(3), 2003.
[91] Mehta, N.R., Medvidovic, N., et al. Towards a Taxonomy of Software Connectors. In Proceedings of the 2000 International Conference on Software Engineering. p. 178-187, ACM Press. Limerick, Ireland, 4-11 June, 2000.
[92] Mellor, S.J. and Balcer, M.J. Executable UML: A Foundation for Model Driven Architecture. 1st ed. 416 pgs., Addison-Wesley Professional, 2002.
[93] Mens, T. and Tourwe, T. A survey of software refactoring. IEEE Transactions on Software Engineering. 30(2), p. 126-139, 2004.

[94] Mezini, M. and Lieberherr, K. Adaptive plug-and-play components for evolutionary software development. ACM SIGPLAN Notices. 33(10), p. 97 - 116, October 1998, 1998.
[95] Milner, R. A Calculus of Communicating Systems. 92, Springer-Verlag, 1980.
[96] Milner, R. Communication and Concurrency. Prentice Hall International Series in Computer Science. Hoare, C.A.R. ed. Prentice Hall International: Hemel Hempstead, Hertfordshire, UK, 1989.
[97] Milner, R., Parrow, J., et al. A calculus of mobile processes, I. Inf. Comput. 100(1), p. 1-40, 1992.
[98] Moriconi, M., Qian, X., et al. Correct Architecture Refinement. IEEE Transactions on Software Engineering. 21(4), p. 356-372, 1995.
[99] MSDN. Partial Class Definitions. http://msdn.microsoft.com/en-us/library/wa80x488(v=vs.80).aspx, 2005.
[100] Muccini, H., Dias, M., et al. Software Architecture-based Regression Testing. Journal of Systems and Software. 79(10), p. 1379-1396, October, 2006.
[101] Murata, T. Petri nets: Properties, analysis and applications. Proceedings of the IEEE. 77(4), p. 541-580, April, 1989.
[102] Murphy, G.C., Lai, A., et al. Separating features in source code: an exploratory study. In Proceedings of the 23rd International Conference on Software EngineeringIEEE Computer Society: Toronto, Ontario, Canada, 2001.
[103] Murphy, G.C., Notkin, D., et al. Software Reflexion Models: Bridging the Gap Between Design and Implementation. IEEE Transactions on Software Engineering. 27(4), p. 364-380, April, 2001.
[104] Murta, L.G.P., van der Hoek, A., et al. ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links. In Proceedings of the Twenty-first IEEE/ACM International Conference on Automated Software Engineering (ASE 2006). p. 135–144, Tokyo, Japan, September, 2006.
[105] Murta, L.P.G., van der Hoek, A., et al. Continuous and Automated Evolution of Architecture-to-Implementation Traceability Links. Automated Software Engineering, Special Issue on Selected Papers from the 21st International Conference on Automated Software Engineering (ASE'2006). 15(1), p. 75-107, 2008.
[106] Niaz, I.A. and Tanaka, J. Code Generation From Uml Statecharts. In Proc. 7 th IASTED International Conf. on Software Engineering and Application (SEA 2003). p. 315 - 321: Marina Del Rey, 2003.
[107] Nickel, U., J\, et al. The FUJABA environment. In Proceedings of the 22nd international conference on Software engineering. p. 742-745, ACM: Limerick, Ireland, 2000.
[108] Nistor, E., Erenkrantz, J.R., et al. ArchEvol: Versioning Architectural-Implementation Relationships. In Proceedings of the 12th International Workshop on Software Configuration Management. p. 99-111, Lisbon, Portugal, September 5-6, 2005.
[109] Nistor, E.C. Towards Maintaining Consistency between Architectural Models and Code. UC Irvine, Report, 2005.
[110] Nistor, E.C. and van der Hoek, A. Explicit Concern-Driven Development with ArchEvol. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software EngineeringIEEE Computer Society, 2009.
[111] Ommering, R.v., Linden, F.v.d., et al. The Koala Component Model for Consumer Electronics Software. IEEE Computer. 33(3), p. 78-85, March, 2000.

[112] Oreizy, P., Medvidovic, N., et al. Architecture-Based Runtime Software Evolution. In Proceedings of the 20th International Conference on Software Engineering (ICSE '98). p. 177-186, IEEE Computer Society. Kyoto, Japan, April, 1998.
[113] Oreizy, P., Gorlick, M.M., et al. An Architecture-based Approach to Self-Adaptive Software. IEEE Intelligent Systems. 14(3), p. 54-62, May-June, 1999.
[114] Oreizy, P. Open Architecture Software: A Flexible Approach to Decentralized Software Evolution. Thesis (Ph. D., Information and Computer Science) Thesis. Information and Computer Science, University of California, 2000. http://www.ics.uci.edu/~peymano/papers/thesis.pdf.
[115] Oreizy, P., Medvidovic, N., et al. Runtime software adaptation: framework, approaches, and styles. In Companion of the 30th international conference on Software engineeringACM: Leipzig, Germany, 2008.
[116] Oreizy, P., Medvidovic, N., et al. Runtime software adaptation: Framework, approaches, and styles. In Companion of the 30th International Conference on Software Engineering, Leipzig. p. 899-910, ACM: New York, NY, 2008.
[117] Parnas, D. Software Aging. In Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italy, 1994.
[118] Parnas, D.L. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM. 15(12), p. 1053-1058, 1972.
[119] Parnas, D.L. Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering. 5(2), p. 128-137, 1979.
[120] Perry, D.E. and Wolf, A.L. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes. 17(4), p. 40-52, October, 1992.
[121] Ren, J. A Connector-Centric Approach to Architectural Access Control. Thesis. Information and Computer Science, University of California, Irvine, p. 222, 2006. http://www.ics.uci.edu/~jie/Thesis.pdf.
[122] Robbins, J.E., Hilbert, D.M., et al. Extending Design Environments to Software Architecture Design. Automated Software Engineering. 5(3), p. 261-290, July, 1998.
[123] Sefika, M., Sane, A., et al. Monitoring Compliance of a Software System with Its High-Level Design Models. In Proceedings of the 18th international conference on Software engineering. p. 387 - 396: Berlin, Germany 1996.
[124] Seidewitz, E. What Models Mean. IEEE Software. 20(5), 2003.
[125] Selic, B. The Pragmatics of Model-Driven Development. IEEE Software. 20(5), 2003.
[126] Sendall, S. and Kozaczynski, W. Model Transformation: The Heart and Soul of Model-Driven Software Dvelopment. IEEE Software. 20(5), 2003.
[127] Sendall, S. and Küster, J. Taming model round-trip engineering. In Proceedings of Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications): Vancouver, Canada, 2004.
[128] Shaw, M., DeLine, R., et al. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering. 21(4), p. 314-335, April, 1995.
[129] Shaw, M. and Clements, P. The Golden Age of Software Architecture. IEEE Softw. 23(2), p. 31-39, 2006.
[130] Silva, I.A.d., Chen, P.H., et al. Lighthouse: coordination through emerging design. In Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchangeACM: Portland, Oregon, 2006.

[131] Software. Argo/UML. http://argouml.tigris.org/.
[132] Software, E. The Trac Open Source Project. http://trac.edgewall.org/.
[133] Spanoudakis, G. and Zisman, A. Software Traceability: A Roadmap Advances in Software Engineering and Knowledge Engineering. Chang, S.K. ed. 3, World Scientific Publishing, 2005.
[134] Spivey, J.M. The Z Notation: A Reference Manual. Prentice-Hall International Series In Computer Science. 155 pgs., Prentice-Hall International: Englewood Cliffs, N.J., 1989.
[135] Steinberg, D., Budinsky, F., et al. EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Professional, 2008.
[136] Sztipanovits, J. and Karsai, G. Model-Integrated Computing. Computer. 30(4), p. 110-111, 1997.
[137] Sztipanovits, J. and Karsai, G. Generative Programming for Embedded Systems. In 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering, 2002.
[138] Szyperski, C. Component Software - Beyond Object-Oriented Programming. 2nd ed. Addison-Wesley, 2002.
[139] Takeo, N. Supporting Architecture-centric Software Development Through Code Generation. Thesis. Information and Computer Science, University of California, Irvine, 2009. http://tps.ics.uci.edu/svn/projects/archstudio4/branches/nobu-3.4/doc/doc/NobuTakeo_Masters_Thesis_2009.pdf.
[140] Tarr, P., Ossher, H., et al. N Degrees of Separation: Multi-dimensional Separation of Concerns. In Proceedings of the 21st International Conference on Software Engineering (ICSE '99). p. 107-119, Los Angeles, California, United States, May 16-22, 1999.
[141] Taylor, R. and van der Hoek, A. Software Design and Architecture: The Once and Future Focus of Software Engineering In Future of Software Engineering 2007, Briand, L.C. and Wolf, A.L. eds. p. 226-243, IEEE-CS Press, 2007.
[142] Taylor, R.N., Medvidovic, N., et al. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering. 22(6), p. 390-406, 1996.
[143] Taylor, R.N., Medvidovic, N., et al. Software Architecture: Foundations, Theory, and Practice. 736 pgs., John Wiley & Sons, 2010.
[144] Teitelman, W. and Masinter, L. The Interlisp Programming Environment. Computer. 14(4), p. 25-34, April, 1981.
[145] Thomas, I. and Nejmeh, B.A. Definitions of Tool Integration for Environments. IEEE Software. 9(2), p. 29-35, March, 1992.
[146] Tracz, W. DSSA (Domain-Specific Software Architecture): Pedagogical Example. ACM SIGSOFT Software Engineering Notes. 20(3), July, 1995.
[147] Trujillo, S., Batory, D., et al. Feature Oriented Model Driven Development: A Case Study for Portlets. In Proceedings of the 29th international conference on Software EngineeringIEEE Computer Society, 2007.
[148] Ubayashi, N., Nomura, J., et al. 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. p. 75-84, ACM: Cape Town, South Africa, 2010.
[149] van Ommering, R. Building Product Populations with Software Components. In Proceedings of the 24th International Conference on Software Engineering (ICSE 2002). p. 255-265, Orlando, Florida, May 19-25, 2002.

[150] van Vliet, H. Software Engineering: Principles and Practice. Second ed. 726 pgs., John Wiley & Sons, LTD, 2000.
[151] W3C. Extensible Markup Language (XML). http://www.w3.org/XML/.
[152] W3C. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, W3C, 2005.
[153] Yan, H., Garlan, D., et al. DiscoTect: A System for Discovering Architectures from Running Systems. In Proceedings of the International Conference on Software Engineering. Edinburgh, Scotland, May, 2004.
[154] Zhang, J. and Cheng, B.H.C. Model-based development of dynamically adaptive software. In Proceedings of the 28th international conference on Software engineering. p. 371-380, ACM: Shanghai, China, 2006.
[155] Zheng, Y. and Taylor, R.N. Taming Changes With 1.x-Way Architecture-Implementation Mapping. In 26th IEEE/ACM International Conference On Automated Software Engineering. p. 396 - 399: Lawrence, Kansas, 2011.
[156] Zheng, Y. and Taylor, R.N. A Rationalization of Confusion, Challenges, and Techniques in Model-Based Software Development. Institute for Software Research, UC Irvine, Report, 2011.