Managing Unanticipated Evolution of Software Architectures

Kim Mens[*], Tom Mens, Bart Wouters[*] and Roel Wuyts[†]
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
{ kimmens | tommens | bwouters | rwuyts }@vub.ac.be

June 8, 1999

Abstract

Few existing approaches towards architectural evolution deal with *unanticipated* evolution. This is an important restriction, since a lot of architectural changes are very difficult to anticipate. The reuse contract formalism has been designed specifically to deal with unanticipated software evolution, and has already proven its practical use in different domains. We claim that the reuse contract approach can be applied to the domain of software architectures, to manage unanticipated evolution of software architectures.

1 Introduction

A lot of current-day software is difficult to understand, maintain or adapt, hard to reuse and difficult to evolve [GAO95, JGJ97, Pan95]. This is partly due to the ever increasing size and complexity of software systems. As engineers searched for better ways to understand their software and new ways to build larger, more complex software systems, software architectures emerged as a natural evolution of design abstractions [SG96].

The *software architecture* of a system is the overall structure of a system in terms of its constituent *components* and their interconnections. Components represent architecturally relevant units of computation and data storage. Their interconnections are usually modeled by *connectors*. These connectors act as sophisticated mediators that can be attached to the components using *ports*. A component may have multiple ports, each of them defining a logically separable point of interaction with its environment. In some sense, these ports can be regarded as an *interface* of the component.

Because software architectures provide an insight into the overall structure and design of a software system, they facilitate software understanding and maintenance, and thus offer some interesting benefits with respect to software evolution as well. However, they do not solve the problems related to software evolution entirely. One could even argue that some of the problems are merely shifted to a higher level of abstraction. Indeed, constantly changing requirements and concerns sometimes force the software architecture itself to be revised [Pou96, Bos98].

It is essential to evolve the architecture of a successful software system continually, because when the quality of the architecture degrades, software modifications become more difficult. This is because design decisions at the architectural level have far reaching consequences on the resultant code [Jak98]. These problems are often referred to as *software aging*, *architectural erosion* and *architectural drift*.

Architects may try to anticipate possible future modifications to the architecture, and design the ar-

[*]Research funded by the Brussels Capital Region (Belgium)
[†]Research conducted on a doctoral grant from the Instituut voor Wetenschap en Technologie (Flanders, Belgium)

chitecture in such a way that it can be adapted to take these modifications into account. However, more often than not, architects are not able to predict adequately where and which changes to the architecture may possibly occur [OOP98]. So the problem remains what to do when unanticipated changes to the software architecture are required. Therefore, it is important to investigate how to deal with *unanticipated* evolution of software architectures.

2 Current Approaches to Architectural Evolution

Current approaches tend to focus mainly on how to deal with *anticipated* evolution of software architectures. A brief overview of current research trends in the area of evolution of software architectures is presented below.

2.1 Overview

In the research literature [Wer98], an important distinction is made between two kinds of architectural evolution. With *design-time* evolution, changes are made to the architecture during design time, i.e., before execution. With *run-time* evolution (also called *dynamic* evolution), the architecture is dynamically modified while the software is running, without compromising application integrity. Two kinds of run-time evolution can be distinguished, depending on whether the changes are triggered by the current state or topology of the system (*programmed* evolution) or if they are given by the reuser and thus completely unpredictable (*ad-hoc reconfiguration*).

2.2 Design-time evolution

For design-time evolution, the ability to manage unanticipated evolution of software architectures is particularly important, because at design time, changes are almost always unpredictable. Even when architects have provided hooks for future evolution, these hooks are seldom what is needed when the system needs to change. Very few architects have sufficient foresight to anticipate where these changes are going to come from [OOP98]. Unfortunately, very little research has been done on the topic of design-time evolution of software architectures.

2.3 Run-time evolution

Approaches for run-time evolution can be divided into three groups depending on how they manage unanticipated evolution. The first and easiest solution is to disallow unanticipated evolution entirely. Other approaches allow only a very restricted form of unanticipated evolution [KM98, OT98, Wer98]. (Often so restricted that it is almost the same as anticipated evolution.) Finally, some approaches do not restrict or prohibit unanticipated evolution, but provide no support for it (dynamic link libraries, COM, DCOM). The software architect is allowed to make unanticipated changes, but has to face the possible consequences of these modifications. We did not find an example of an approach where unanticipated evolution is allowed *and* fully supported.

3 Unanticipated Evolution

To summarize, nearly all current approaches we know of focus essentially on anticipated evolution of software architectures, and largely neglect the issue of unanticipated evolution. Therefore, we propose to investigate the problem of unanticipated evolution of software architectures. To simplify the problem, we will focus on design-time evolution first, but we have good hope that our approach will be sufficiently general so that it can be applied (or extended) to run-time evolution as well. Our approach has no intention of replacing current approaches dealing with run-time evolution but should instead be seen as a complementary approach focusing essentially on the aspect of unanticipated evolution.

3.1 Reuse Contracts

The idea of our approach is to use the *reuse contract* model [SLMD96, Luc97, Hon98] for dealing with architectural evolution and, more particularly, detecting incompatibilities, inconsistencies and conflicts during unanticipated evolution. Because reuse contracts have already proven useful to deal with this kind of evolution at the implementation level [SLMD96] and design level [Luc97, MLS98a, MLS98b], and because the underlying ideas are sufficiently general to be applied to other levels as well [Men99], we propose to apply them to the area of software architectures.

Essentially, a reuse contract consists of two *contract clauses* (the *provider clause* and the *reuser clause*) that are related to each other by means of a *contract type*. The provider of an evolvable software artifact has the contractual obligation to specify what properties can be relied on by dependent artifacts, while the evolver has the obligation to specify the modifications that are made to these properties. The *contract type* specifies the exact kind of modification that takes place. The basic contract types are *extension* and *cancellation*, which are used to add or remove any kind of element, and *refinement* and *coarsening* which are used to add or remove any kind of relationship between elements. These primitive contract types can be scaled up to *composite contract types* that correspond to predefined combinations of primitive contract types.

By comparing contract types, *potential evolution conflicts* can be detected when merging independent evolutions of the same software artifact. These evolution conflicts correspond to inconsistencies that arise when the same part of the software is modified in different ways.

Because design-time evolution is inherently unanticipated, it is not always as easy to know whether a conflict occurs or not. Therefore, reuse contracts take a "worst case" scenario by generating conflict warnings for every potentially undesired interaction. The more information that is known about the considered domain (e.g., software architectures) and the particular evolution, the better the approximation of the evolution conflicts will be.

3.2 Applying Reuse Contracts to Software Architectures

In [Men99] a unified formalism for reuse contracts[1] is being developed based on labelled typed nested graphs and conditional graph rewriting. This formalism can be customized to specific areas, such as evolution of software architectures.

Typical approaches towards (run-time) evolution of software architectures distinguish four fundamental ways in which a software architecture can be modified: by adding new components, by removing existing components, by adding new connectors between components and by removing existing connectors between components. Sometimes this set of modification operations is called an *Architectural Modification Language* (AML), as opposed to an *Architectural Definition Language* (ADL) which only describes the static architecture.

The four modification operators mentioned above strongly remind us of the primitive reuse contract types *extension*, *cancellation*, *refinement* and *coarsening*, strengthening our belief that reuse contracts are a good formalism to reason about evolution of architectures. While the contract types can be regarded as an AML, contract clauses form the equivalent of an ADL.

In order to customize the reuse contract formalism to the domain of software architectures, we first need to agree upon the kind of entities and relationships that are used in this domain, as well as the constraints that hold between them. Instead of distinguishing the notions of *component* and *connector* [PW92, Gar95] we take the more general approach of considering everything as an architectural *element*. Elements can have external *gates* that are used to *link* them to other elements. Elements can be *primitive* or *composite*. In the latter case, they constitute an entire *architecture* themselves. *Bindings* can be used to connect the external gates of a composite element to the gates of elements in the architecture defined by the composite element.

These general architectural notions can be customised further to define specific *architectural styles*.

[1]This formalism is an extension, formalization and generalization of the reuse contract formalism presented in [Luc97].

To model the *C3-style*, *components* and *connectors* can be defined as special kinds of elements with extra restrictions. *Ports* are used instead of gates. This style can be refined further to a *pipe-and-filter style*, by defining *pipes* as a special kind of connectors, and *filters* as a special kind of components. An even further refinement in the *pipeline style*, where *stages* are used as filters that have only one input and output port.

As a second step, the domain-independent contract types need to be customized to the specific case of software architectures. *Extension* and *cancellation* will correspond to the addition or removal of an architectural element or gate. *Refinement* and *coarsening* can be used to add or remove links or bindings between elements. These domain-specific variants of our primitive contract types can also be combined into frequently occurring composite contract types which have a more intuitive meaning.

Once these customizations have been made, it remains to be seen which specific *architectural evolution conflicts* can be detected. These conflicts will not only occur when the same architectural part is modified in different ways by different evolvers, but can also be used to check compliance between an architecture and its underlying implementation. If the implementation evolves in ways not supported by the architecture, a conflict will be detected. In this way, the problem of *architectural drift* can be tackled.

4 Conclusion and Future Work

Little research has been done on *unanticipated* evolution of software architectures, although it is an important research topic. The reuse contract approach has been conceived specifically to reason about unanticipated evolution of software artifacts, and has already proven its use for dealing with software evolution during design and implementation. Recently, a general domain-independent formal framework for reuse contracts has been developed [Men99], and a prototype PROLOG implementation of this framework exists. The framework has already been customized to the domain of class diagrams, and is currently being customized to the area of software architectures. This will allow us to support conflict detection and impact analysis during unanticipated architectural evolution.

An interesting alternative could be Oreizy and Taylor's Architecture Evolution Manager (AEM) to enforce compliance between the architectural and the implementation level [OT98]. External Analysis Modules can be plugged in to customize the basic functionality of the AEM. Reuse contracts may be implemented as such an external analysis module.

This research is part of the larger research effort of providing integrated support for unanticipated evolution during the entire software life-cycle, i.e., in and between all different phases, ranging from software requirements to implementation.

(Research conducted together with our industrial partner Wang Global and funded by the Brussels Capital Region.)

References

[ADG98] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of 1998 Conference on Fundamental Approaches to Software Engineering*, 1998. Lisbon, Portugal.

[Bos98] J. Bosch. Evolution and composition of reusable assets in product-line architectures: A case study. Technical report, University of Karlskrona/Ronneby, 1998.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, November 1995.

[Gar95] D. Garlan. First international workshop on architectures of software systems — workshop summary. *ACM SIGSOFT, Software Engineering Notes*, 20(3):84–89, 1995.

[Hon98] K. De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems.* PhD thesis, Dept. of Computer Science, Vrije Universiteit Brussel, Belgium, 1998.

[Jak98] C. B. Jaktman. A maintenance check for evolving a product-line architecture by determining the indicators of erosion, 1998. Workshop on Empirical Studies of Software Maintenance (WESS98), Bethesda, Maryland, November 16.

[JGJ97] I. Jacobson, M.L. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addisson-Wesley, 1997.

[KM98] J. Kramer and J. Magee. Analysing dynamic change in software architectures, 1998. In [ADG98].

[Luc97] C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 1997.

[Men99] T. Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1999. In preparation.

[MLS98a] T. Mens, C. Lucas, and P. Steyaert. Giving precise semantics to reuse in UML. In *Proceedings of ICSE'98 Workshop on Precise Semantics for Software Modeling Techniques, Kyoyo, Japan*, pages 73–89, April 1998. Technical Report TUM-I9803, Technische Universität Munchen.

[MLS98b] T. Mens, C. Lucas, and P. Steyaert. Supporting reuse and evolution of UML models. In *Proceedings of UML'98 International Workshop, Mulhouse, France*, June 1998.

[OOP98] How software architectures learn: What happens after they are built?, 1998. Workshop 7, Sunday, October 18th, OOPSLA'98 Conference, Vancouver, Canada.

[OT98] P. Oreizy and R. N. Taylor. On the role of software architectures in runtime system reconfiguration. 1998.

[Pan95] C. Pancake. Object roundtable, the promise and the cost of object technology: A five-year forecast. *Communications of the ACM*, 38(10):32–49, October 1995.

[Pou96] J. S. Poulin. Evolution of a software architecture for management information systems. In *Proceedings of the Second International Software Architecture Workshop (ISAW2)*, pages 134–137, 1996. San Francisco, California, USA, 14-15 October.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[SG96] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SLMD96] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of the OOPSLA'96 Conference on Object-Oriented Programming, Systems, Languages and Applications*, number 31(10) in ACM SIGPLAN Notices, pages 268–285. ACM Press, 1996.

[Wer98] M. Wermelinger. Software architecture and the chemical abstract machines, 1998. In [ADG98].