Change Impact Analysis for Architectural Evolution *

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0214, Japan
Email: zhao@cs.fit.ac.jp

Abstract

Chang impact analysis is useful in software maintnenance and evolution. Many techniques have been proposed to support change impact analysis at the code level of software systems, but little effort has been made for change impact analysis at the architectural level. In this paper, we present an approach to support change impact analysis of software architectures based on architectural slicing technique. The main feature of our approach is to assess the effect of changes in a software architecture by analyzing its formal architectural specification, and therefore, the process of change impact analysis at the architectural-level can be automated completely.

1 Introduction

Software change is an essential operation for software evolution. The change is a process that either introduces new requirements into an existing system, or modifies the system if the requirement were not correctly implemented, or moves the system into a new operation environment. The mini-cycle of change as described in [15] is composed of the several phases: request for change, planning phase which consists of program comprehension and change impact analysis, change implementation including restructuring for change and change propagation, verification and validation, and redocumentation. Among these phases, in this paper we will focus our attentions on the issue of planing phase, in particularly, change impact analysis.

Change impact analysis is the task that through which the programmers can assess the extent of the change, i.e., the software component that will impact the change, or be impacted by the change. Change impact analysis provide techniques to address the problem by identifying the likely ripple-effect of software changes and using this information to re-engineer the software system design.

Most work on software change impact analysis focused on code level of software systems which are derived solely from source code of a program [3, 6, 7], and the study of architectural-level change impact analysis has received little attention. However, as software systems become large and complex, it is necessary to perform architectural-level impact analysis because it allows you to capture the information of change effect of the a system's architecture earlier in the system life cycle so you can perform software evolution actions earlier [10].

However, the study of architectural-level impact analysis has received little attention in comparison with code-level impact analysis. One important reason is while the code level for software systems is now well understood, the architectural level is currently understood mostly at the level of intuition, anecdote, and folklore [12]. Existing representations that a system architect uses to represent the architecture of a software system are usually informal and *ad hoc*, and therefore can not capture enough useful information of the system's architecture. Moreover, with such an informal and *ad hoc* manner, it is difficult to develop analysis tools to automatically support the change impact analysis at the architectural level of software systems. In order to develop architectural-level change impact analysis tool to support architectural evolution during software design, formal modeling of software architectures is strongly required.

Recently, as the size and complexity of software systems increases, the design and specification of the overall software architecture of a system is receiving increasingly attention. The software architecture of a system defines its high-level structure, exposing its gross organization as a collection of interacting components. A well-defined architecture allows an engineer to reason about system properties at a high level of abstraction [12]. Architecture description languages (ADLs) are formal languages that can be used to represent the architecture of a software system. They focus on the high-level structure of the overall application rather than the implementation details of any specific source module. In order to support formal representation and reasoning of software architecture, a number of ADLs such as Wright [1], Rapide [8], and UniCon [11] have been proposed. By using an ADL, a system architect can formally represent various general attributes of a software system's architecture. This provides a promising solution to develop techniques to support change impact analysis for software architectures because formal language support for software architecture provides a useful platform on which automated support tools for architectural-level impact analysis can be developed.

In this paper, we present an approach for change impact analysis of software architectures based on *architectural slicing* technique. The main feature of our ap-

*This work is partly supported by The Ministry of Education, Science, Sports and Culture of Japan under Grand-in-Aid for Encouragement for Young Scientists (No.11780241) and by a grant from the Computer Science Laboratory of Fukuoka Institute of Technology.

proach is to assess the effect of changes in a software architecture by analyzing its formal architectural specification, and therefore, the process of change impact analysis at the architectural-level can be automated completely.

Traditional program slicing, originally introduced by Weiser [14], is a decomposition technique which extracts program elements related to a particular computation. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion*. The task to compute program slices is called *program slicing*.

In contrast to traditional program slicing, architectural slicing is designed to operate on a formal architectural specification of a software system, rather than the source code of a conventional program. Architectural slicing provides knowledge about the high-level structure of a software system, rather than the low-level implementation details of a conventional program. Our purpose for development of architectural slicing is to support architectural-level impact analysis, maintenance, reengineering, and reverse engineering of large-scale software systems.

Applying slicing technique to change impact analysis of software architectures promises benefit for software architecture understanding and maintenance. When a maintainer wants to modify a component in a software architecture in order to satisfy new design requirements, the maintainer must first investigate which components will affect the modified component and which components will be affected by the modified component. This process is usually called *impact analysis*. By slicing a software architecture, the maintainer can extract the parts of a software architecture containing those components that might affect, or be affected by, the modified component. The slicing tool which provides such information can assist the maintainer greatly.

The primary idea of architectural slicing has been presented in [16, 17, 18], and this article can be regarded as an outgrowth of applying architectural slicing to support impact analysis of software architectures.

The rest of the paper is organized as follows. Section 2 briefly introduces how to represent a software architecture using Wright: an architectural description language. Section 3 shows a motivation example. Section 4 describes some notions about architectural slicing. Section 5 discusses some related work. Concluding remarks are given in Section 6.

2 Software Architectural Specification in Wright

We assume that readers are familiar with the basic concepts of software architecture and architectural description language, and in this paper, we use Wright architectural description language [1] as our target language for formally representing software architectures. The selection of Wright is based on that it supports to represent not only the architectural structure but also the architectural behavior of a software architecture.

Below, we use a simple Wright architectural specification taken from [9] as a sample to briefly introduce how to use Wright to represent a software architecture.

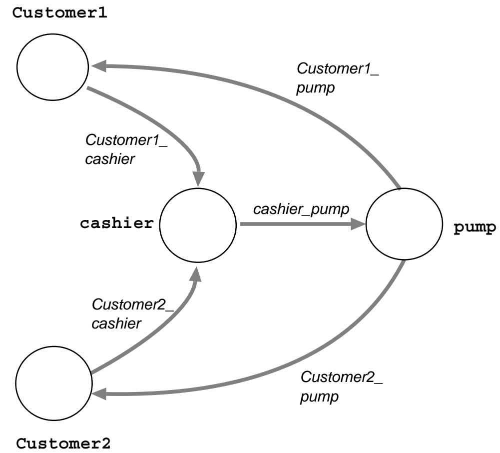

Figure 1: The architecture of the Gas Station system.

The specification is showed in Figure 2 which models the system architecture of a Gas Station system [4].

2.1 Representing Architectural Structure

Wright uses a *configuration* to describe architectural structure as graph of components and connectors.

Components are computation units in the system. In Wright, each component has an *interface* defined by a set of *ports*. Each port identifies a point of interaction between the component and its environment.

Connectors are patterns of interaction between components. In Wright, each connector has an *interface* defined by a set of *roles*. Each role defines a participant of the interaction represented by the connector.

A Wright architectural specification of a system is defined by a set of component and connector type definitions, a set of instantiations of specific objects of these types, and a set of *attachments*. Attachments specify which components are linked to which connectors.

For example, in Figure 2 there are three component type definitions, `Customer`, `Cashier` and `Pump`, and three connector type definitions, `Customer_Cashier`, `Customer_Pump` and `Cashier_Pump`. The configuration is composed of a set of instances and a set of attachments to specify the architectural structure of the system.

2.2 Representing Architectural Behavior

Wright models architectural behavior according to the significant events that take place in the computation of components, and the interactions between components as described by the connectors. The notation for specifying event-based behavior is adapted from CSP [5]. Each CSP process defines an alphabet of events and the permitted patterns of events that the process may exhibit. These processes synchronize on common events (i.e., interact) when composed in parallel. Wright uses such process descriptions to describe the behavior of ports, roles, computations and glues.

A *computation* specification specifies a component's behavior: the way in which it accepts certain events on certain *ports* and produces new events on those or other

```
Configuration GasStation
    Component Customer
        Port Pay = pay!x → Pay
        Port Gas = take → pump?x → Gas
        Computation = Pay.pay!x → Gas.take → Gas.pump?x → Computation
    Component Cashier
        Port Customer1 = pay?x → Customer1
        Port Customer2 = pay?x → Customer2
        Port Topump = pump!x → Topump
        Computation = Customer1.pay?x → Topump.pump!x → Computation
            [] Customer2.pay?x → Topump.pump!x → Computation
    Component Pump
        Port Oil1 = take → pump!x → Oil1
        Port Oil2 = take → pump!x → Oil2
        Port Fromcashier = pump?x → Fromcashier
        Computation = Fromcashier.pump?x →
            (Oil1.take → Oil1.pump!x → Computation)
            [] (Oil2.take → Oil2.pump!x → Computation)
    Connector Customer_Cashier
        Role Givemoney = pay!x → Givemoney
        Role Getmoney = pay?x → Getmoney
        Glue = Givemoney.pay?x → Getmoney.pay!x → Glue
    Connector Customer_Pump
        Role Getoil = take → pump?x → Getoil
        Role Giveoil = take → pump!x → Giveoil
        Glue = Getoil.take → Giveoil.take → Giveoil.pump?x → Getoil.pump!x → Glue
    Connector Cashier_Pump
        Role Tell = pump!x → Tell
        Role Know = pump?x → Know
        Glue = Tell.pump?x → Know.pump!x → Glue
    Instances
        Customer1: Customer
        Customer2: Customer
        cashier: Cashier
        pump: Pump
        Customer1_cashier: Customer_Cashier
        Customer2_cashier: Customer_Cashier
        Customer1_pump: Customer_Pump
        Customer2_pump: Customer_Pump
        cashier_pump: Cashier_Pump
    Attachments
        Customer1.Pay as Customer1_cashier.Givemoney
        Customer1.Gas as Customer1_pump.Getoil
        Customer2.Pay as Customer2_cashier.Givemoney
        Customer2.Gas as Customer2_pump.Getoil
        casier.Customer1 as Customer1_cashier.Getmoney
        casier.Customer2 as Customer2_cashier.Getmoney
        cashier.Topump as cashier_pump.Tell
        pump.Fromcashier as cashier_pump.Know
        pump.Oil1 as Customer1_pump.Giveoil
        pump.Oil2 as Customer2_pump.Giveoil
End GasStation.
```

Figure 2: An architectural specification in WRIGHT.

ports. Moreover, WRIGHT uses an overbar to distinguish initiated events from observed events *. For example, the `Customer` initiates `Pay` action (i.e., `pay!x`) while the `Cashier` observes it (i.e., `pay?x`).

A *port* specification specifies the local protocol with which the component interacts with its environment through that port.

A *role* specification specifies the protocol that must be satisfied by any port that is attached to that role. Generally, a port need no have the same behavior as the role that it fills, but may choose to use only a subset of the connector capabilities. For example, the `Customer` role `Gas` and the `Customer_Pump` port `Getoil` are identical.

*In this paper, we use an underbar to represent an initiated event instead of an overbar that used in the original WRIGHT language definition [1].

A *glue* specification specifies how the roles of a connector interact with each other. For example, a `Cashier_Pump` tell (Tell.pump?x) must be transmitted to the `Cashier_Pump` know (`Know.pump!x`).

As a result, based on formal WRIGHT architectural specifications, we can infer which ports of a component are input ports and which are output ports. Also, we can infer which roles are input roles and which are output roles. Moreover, the direction in which the information transfers between ports and/or roles can also be inferred based on the formal specification. Such kinds of information can be used to construct the architectural flow graph of a software architecture for computing an architectural slice efficiently.

In this paper we assume that a software architecture be represented by a formal architectural specification which contains three basic types of design enti-

ties, namely, *components* whose interfaces are defined by a set of elements called *ports*, *connectors* whose interfaces are defined by a set of elements called *roles* and the *configuration* whose topology is declared by a set of elements called *instances* and *attachments*. Moreover, each component has a special element called *computation* and each connector has a special element called *glue* as we described above. In the rest of the paper, we assume that an architectural specification P be denoted by (C_m, C_n, c_g) where C_m is the set of components in P, C_n is the set of connectors in P, and c_g is the configuration of P.

3 Motivation Example

We present a simple example to explain our approach on change impact analysis for software architectures via architectural slicing.

Consider the Gas Station system whose architectural representation is shown in Figure 1, and WRIGHT specification is shown in Figure 2. Suppose a maintainer needs to modify the component `cashier` in the architectural specification in order to satisfy some new design requirements. The first thing the maintainer has to do is to investigate which components and connectors interact with component `cashier` through its ports `Customer1`, `Customer2`, and `Topump`. A common way is to manually check the source code of the specification to find such information. However, it is very time-consuming and error-prone even for a small size specification because there may be complex dependence relations between components in the specification. If the maintainer has an architectural slicer at hand, the work may probably be simplified and automated without the disadvantages mentioned above. In such a scenario, an architectural slicer is invoked, which takes as input: (1) a complete architectural specification of the system, and (2) a set of ports of the component `cashier`, i.e., `Customer1`, `Customer2` and `Topump` (this is an *architectural slicing criterion*). The slicer then computes a backward and forward architectural slice respectively with respect to the criterion and outputs them to the maintainer. A backward architectural slice is a partial specification of the original one which includes those components and connectors that might affect the component `cashier` through the ports in the criterion, and a forward architectural slice is a partial specification of the original one which includes those components and connectors that might be affected by the component `cashier` through the ports in the criterion. The other parts of the specification that might not affect or be affected by the component `cashier` will be removed, i.e., sliced away from the original specification. The maintainer can thus examine only the contents included in a slice to investigate the impact of modification.

4 Architectural Slicing

In this paper we assume that a software architecture be represented by a formal architectural specification which contains three basic types of design entities, namely, *components* whose interfaces are defined by a set of elements called *ports*, *connectors* whose interfaces are defined by a set of elements called *roles* and the *configuration* whose topology is declared by a set of elements called *instances* and *attachments*. Moreover, each component has a special element called *computation* and each connector has a special element called *glue* as we described above. In the rest of the paper, we assume that an architectural specification P be denoted by (C_m, C_n, c_g) where C_m is the set of components in P, C_n is the set of connectors in P, and c_g is the configuration of P.

Intuitively, an *architectural slice* may be viewed as a subset of the behavior of a software architecture, similar to the original notion of the traditional static slice. However, while a traditional slice intends to isolate the behavior of a specified set of program variables, an architectural slice intends to isolate the behavior of a specified set of a component or connector's elements. Given an architectural specification $P = (C_m, C_n, c_g)$, our goal is to compute an architectural slice $S_p = (C'_m, C'_n, c'_g)$ which consists of those components and connectors of P that preserve partially the semantics of P. we can give some notions of architectural slicing as follows.

In a WRIGHT architectural specification, for example, a component's interface is defined to be a set of ports which identify the form of the component interacting with its environment, and a connector's interface is defined to be a set of roles which identify the form of the connector interacting with its environment. To understand how a component interacts with other components and connectors to making changes, a maintainer must examine each port of the component of interest. Moreover, it has been frequently emphasized that connectors are as important as components for architectural design, and a maintainer may also want to modify a connector during the maintenance. To satisfy these requirements, we can define a slicing criterion for a WRIGHT architectural specification as a set of ports of a component or a set of roles of a connector of interest.

Let $P = (C_m, C_n, c_g)$ be an architectural specification. A *slicing criterion* for P is a pair (c, E) such that: (1) $c \in C_m$ and E is a set of elements of c, or (2) $c \in C_n$ and E is a set of elements of c.

Note that the selection of a slicing criterion depends on users' interests on what they want to examine. If they are interested in examining a component in an architectural specification, they may use slicing criterion 1. If they are interested in examining a connector, they may use slicing criterion 2. Moreover, the determination of the set E also depends on users' interests on what they want to examine. If they want to examine a component, then E may be the set of ports or just a subset of ports of the component. If they want to examine a connector, then E may be the set of roles or just a subset of roles of the connector.

Let $P = (C_m, C_n, c_g)$ be an architectural specification. A *backward architectural slice* S_{bp} of P on a given slicing criterion (c, E) is a set of those reduced components, connectors, and configuration that might directly or indirectly affect the behavior of c through elements in E. A *forward architectural slice* S_{fp} of P on a given slicing criterion (c, E) is a set of those reduced components, connectors, and configuration that might be directly or

```
Configuration GasStation
    Component Customer
        Port Pay = pay!x → Pay

        Computation = Pay.pay!x → Gas.take → Gas.pump?x → Computation
    Component Cashier
        Port Customer1 = pay?x → Customer1
        Port Customer2 = pay?x → Customer2
        Port Topump = pump!x → Topump
        Computation = Customer1.pay?x → Topump.pump!x → Computation
            [] Customer2.pay?x → Topump.pump!x → Computation

    Connector Customer_Cashier
        Role Givemoney = pay!x → Givemoney
        Role Getmoney = pay?x → Getmoney
        Glue = Givemoney.pay?x → Getmoney.pay!x → Glue

    Instances
        Customer1: Customer
        Customer2: Customer
        cashier: Cashier

        Customer1_cashier: Customer_Cashier
        Customer2_cashier: Customer_Cashier

    Attachments
        Customer1.Pay as Customer1_cashier.Givemoney

        Customer2.Pay as Customer2_cashier.Givemoney

        casier.Customer1 as Customer1_cashier.Getmoney
        casier.Customer2 as Customer2_cashier.Getmoney

    End GasStation.
```

Figure 3: A backward slice of the architectural specification in Figure 2.

indirectly affected by the behavior of c through elements in E.

The view of an architectural slice defined above contains enough information for a maintainer to facilitate the modification.

The slicing notions defined here give us only a general view of an architectural slice, and do not tell us how to compute it. In [17, 18] we presented a two-phase algorithm to compute a slice of an architectural specification based on its information flow graph. Our algorithm contains two phases: (1) Computing a slice S_g over the information flow graph of an architectural specification, and (2) Constructing an architectural slice S_p from S_g.

Figure 3 shows a backward slice of the WRIGHT specification in Figure 2 with respect to the slicing criterion (`cashier`, E) such that E={`Customer1`, `Customer2`, `Topump`} is a set of ports of component `cashier`.

5 Related Work

Many researches have been done to support change impact analysis of software systems at the code level. Bohner and Arnold [2] recently edited a book which is a collection of many papers and articles related to change impact analysis of software systems at the code level. However, in comparison with code-level change impact analysis, the study of architectural-level change impact analysis of software systems has received little attention. To the best of our knowledge, the only work that is similar with ours is that presented by Stafford, Richardson and Wolf [13], who introduced a software architecture dependence analysis technique, called *chaining* to support software architecture development such as debugging and testing. In chaining, links represent the dependence relationships that exist in an architectural specification. Links connect elements of the specification that are directly related, producing a chain of dependences that can be followed during analysis. However, their technique is mainly focused on handling Rapide architectural description language in which connectors are not explicitly modeled.

6 Concluding Remarks

In this paper, we presented an approach for change impact analysis of software architectures based on *architectural slicing* technique. The main feature of our approach is to assess the effect of changes of a software architecture by analyzing its formal architectural specification, and therefore, the process of change impact analysis at the architectural-level can be automated completely.

In architectural description languages, in addition to provide both a conceptual framework and a concrete syntax for characterizing software architectures, they also provide tools for parsing, displaying, compiling, analyzing, or simulating architectural specifications written in their associated language. However, existing language environments provide no tools to support architectural-level change impact analysis from an engineering viewpoint. We believe that such a tool should be provided by any ADL as an essential means to support software architecture development and evolution.

To demonstrate the usefulness of our impact analysis approach, we plan to implement an impact analysis tool for WRIGHT architectural descriptions to support architectural-level understanding and evolution.

References

[1] R. Allen, "A Formal Approach to Software Architecture," PhD thesis, Department of Computer Science, Carnegie Mellon University, 1997.

[2] S. A. Bohner and R. S. Arnold, "Software Change Impact Analysis," IEEE Computer Society Press, 1996.

[3] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transaction on Software Engineering*, Vol.17, No.8, pp.751-761, 1991.

[4] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, Vol.2, No.2, pp.47-57, 1985.

[5] C.A.R. Hoare, "Communicating Sequential Processes," Prentice Hall, 1985.

[6] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.

[7] J. P. Loyall and S. A. Mathisen, "Using Dependence Analysis to Support the Software Maintenance Process," *Proceedings of the International Conference on Software Maintenance*, 1993.

[8] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Bryan, and W. Mann, "Specification Analysis of System Architecture Using Rapide," *IEEE Transaction on Software Engineering*, Vol.21, No.4, pp.336-355, April 1995.

[9] G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J.Osterweil, "Applying Static Analysis to Software Architectures," *Proceedings of the Sixth European Software Engineering Conference*, LNCS, Vol.1301, pp.77-93, Springer-Verlag, 1997.

[10] H. D. Rombach, "Design Measurement: Some Lessons Learned," *IEEE Software*, pp.17-25, March 1990.

[11] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transaction on Software Engineering*, Vol.21, No.4, pp.314-335, April 1995.

[12] M. Shaw and D. Garlan, "Software Architecture: Perspective on an Emerging Discipline," Prentice Hall, 1996.

[13] J. A. Stafford and A. L. Wolf, "Architecture-level Dependence Analysis for Software Systems," Technical Report CU-CS-913-00, Department of Computer Science, University of Colorado, December 2000.

[14] M. Weiser, "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method," PhD thesis, University of Michigan, Ann Arbor, 1979.

[15] S. S. Yau, J. S. Collofello, and T. MacGregor, "Ripple Effect Analysis of Software Maintenance," *Proc. of the COMPSAC'78*, pp.60-65, IEEE Computer Society Press, 1978.

[16] J. Zhao, "Using Dependence Analysis to Support Software Architecture Understanding," in M. Li (Ed.), *New Technologies on Computer Software*, pp.135-142, International Academic Publishers, September 1997.

[17] J. Zhao, " Software Architecture Slicing," *Proceedings of JSSST'97*, pp.49-52, September 1997

[18] J. Zhao, " Applying Slicing Technique to Software Architectures," *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems*, pp.87-98, Monterey, USA, August 1998.