JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2002; **14**:317–333 (DOI: 10.1002/smr.258)

Research

Change impact analysis to support architectural evolution

Jianjun Zhao[1,*,†], Hongji Yang[2], Liming Xiang[3] and Baowen Xu[4]

[1]*Department of Computer Science and Engineering, Fukuoka Institute of Technology, 3-30-1 Wajiro-Higashi, Fukuoka 811-0295, Japan*
[2]*Computer Science Department, De Montfort University, The Gateway, Leicester LE1 9BH, U.K.*
[3]*Department of Information Science, Kyushu Sangyo University, 2-3-1 Matsukadai, Fukuoka 813-8503, Japan*
[4]*Department of Computer Science, Southeast University, Nanjing 210096, People's Republic of China*

SUMMARY

Change impact analysis is a useful technique in software maintenance 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 supporting change impact analysis at the architectural level of software systems based on an architectural slicing and chopping 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 can be automated completely. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: architectural description language; architectural evolution; architectural slicing and chopping; change impact analysis; software maintenance

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, modifies the system if the requirements were not

*Correspondence to: Dr Jianjun Zhao, Department of Computer Science and Engineering, Fukuoka Institute of Technology, 3-30-1 Wajiro-Higashi, Fukuoka 811-0295, Japan.
†E-mail: zhao@cs.fit.ac.jp

Contract/grant sponsor: Japan Society for the Promotion of Science (JSPS) under Grand-in-Aid for Encouragement for Young Scientists; contract/grant number: 11870241
Contract/grant sponsor: National Natural Science Foundation of China (NSFC); contract/grant number: 60073012
Contract/grant sponsor: Natural Science Foundation of Jiangsu, China; contract/grant number: BK2001004
Contract/grant sponsor: Opening Foundation of State Key Laboratory of Software Engineering in Wuhan University
Contract/grant sponsor: Foundation of State Key Laboratory for Novel Software Technology in Nanjing University

Received 31 January 2002
Revised 1 August 2002

correctly implemented, or moves the system into a new operational environment. The mini-cycle of change as described in [1] is composed of several phases: *request for change*, *planning phase* consisting of program comprehension and change impact analysis, *change implementation* including restructuring for change and change propagation, *verification and validation*, and *re-documentation*. Among these phases, in this paper we focus our attentions on the issue of the planning phase, in particular on change impact analysis, to support architectural evolution.

Change impact analysis [2,3] is the task through which 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 provides 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.

Separation of concerns [4] is a foundation principle of software engineering. Separating the aspects of systems that perform different roles may have many benefits for software development, such as simplifying the code of software systems, making systems easier to understand, maintain, and evolve, and also less prone to bugs. From the viewpoint of separation of concerns, change impact analysis may have different dimensions (regarding the levels of abstraction of a system) according to which the separation of concerns can be made. For example, change impact analysis can be performed at the code level to obtain the detailed information regarding the effect of changes at the code level of the system. Also, change impact analysis can be performed at the architectural level of a system to reduce the complexity of code-level change impact analysis and to allow a maintenance programmer to assess the effect of changes of the system at the architectural level so that software evolution actions can be made earlier [5]. The separation of change impact analysis from different levels is therefore a necessary step to reduce the cost of change impact analysis during software evolution. However, most existing work on change impact analysis has been focused on the code level of software systems which is derived solely from the source code of a program [6–8], and the study of architectural-level change impact analysis has received little attention.

In order to develop a change impact analysis technique at the architectural level to support architectural evolution during software design, formal modeling of software architectures is required. 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 [9]. 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 [10], Rapide [11], and UniCon [12] 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 the development of techniques to support change impact analysis at the architectural level 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 to supporting change impact analysis at the architectural level of software systems based on an *architectural slicing and chopping* 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 can be automated completely.

Traditional program slicing, originally introduced by Weiser [13], is a decomposition technique that 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*. As shown in [2], program slicing is an essential technique to support change impact analysis of software systems at the source code level. Jackson and Rollins [14] introduced a similar notion, called *program chopping*, which can be regarded as the generalization of program slicing. Given two elements u and v in a program, chopping answers questions of the form 'What are all the program elements that serve to transmit effects from a given source elements u to a given target element v?' Similarly to program slicing, program chopping has similar applications in change impact analysis. In contrast to traditional program slicing and chopping, architectural slicing and chopping are designed to operate on a formal architectural specification of a software system, rather than the source code of a conventional program. Architectural slicing and chopping provide knowledge about the high-level structure of a software system, rather than the low-level implementation details of a conventional program.

Using architectural slicing and chopping to support change impact analysis of software architectures promises benefits for architectural evolution. When a maintenance programmer wants to modify a component in a software architecture in order to satisfy new requirements, the programmer must first investigate which components will affect the modified component and which components will be affected by the modified component. By using a slicing or chopping tool, the programmer can extract the parts of a software architecture containing those components that might affect, or be affected by, the modified component. A tool which provides such change impact information can assist the programmer greatly.

The primary idea of architectural slicing has been presented in [15,16], and this paper can be regarded as an outgrowth of our previous work to apply architectural slicing for change impact analysis of software architectures. To this end, we present some new architectural slicing notions which are different from the original architectural slicing introduced in [15,16] and a new form of technique called architectural chopping. Our slicing and chopping notions are designed specially to support change impact analysis. Moreover, the goal of this paper is to provide a sound and formal basis for performing architectural-level change impact analysis before applying it to real software architecture design.

The rest of the paper is organized as follows. Section 2 briefly introduces how to represent a software architecture using WRIGHT. Section 3 defines some notions about architectural slicing and chopping. Section 4 shows how to perform change impact analysis of software architectures. Section 5 discusses some related work. Concluding remarks are given in Section 6.

2. MODELING SOFTWARE ARCHITECTURES USING WRIGHT

We assume that readers are familiar with the basic concepts of software architecture and an architectural description language. In this paper we choose the WRIGHT architectural description language [10] as our target language for formally representing software architectures. This selection is based on the fact that components, connectors, and configurations in WRIGHT are clearly specified and WRIGHT has already been well studied.

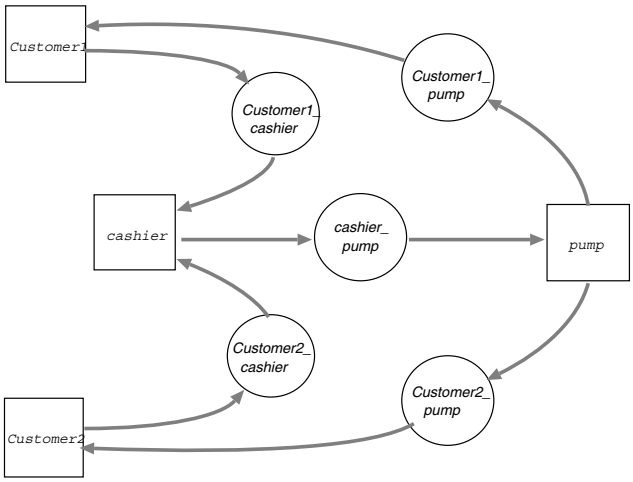


Figure 1. The architecture of the Gas Station system.

Below, we use a WRIGHT architectural specification taken from [17] as a sample to briefly introduce how to use WRIGHT to represent a software architecture. The specification shown in Figure 2 models the system architecture of a Gas Station system [18] shown in Figure 1.

2.1. Architectural structure modeling

WRIGHT uses a *configuration* to describe the architectural structure as a 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.

Configuration GasStation
  Component Customer
    Port Pay = $\overline{pay!x}$ → Pay
    Port Gas = $\overline{take}$ → pump?x → Gas
    Computation = $\overline{Pay.pay!x}$ → $\overline{Gas.take}$ →
      Gas.pump?x → Computation
  Component Cashier
    Port Customer1 = pay?x → Customer1
    Port Customer2 = pay?x → Customer2
    Port Topump = $\overline{pump!x}$ → Topump
    Computation = Customer1.pay?x →
      $\overline{Topump.pump!x}$ → Computation []
      Customer2.pay?x → $\overline{Topump.pump!x}$ →
      Computation
  Component Pump
    Port Oil1 = take → $\overline{pump!x}$ → Oil1
    Port Oil2 = take → $\overline{pump!x}$ → Oil2
    Port Fromcashier = pump?x → Fromcashier
    Computation = Fromcashier.pump?x →
    (Oil1.take → $\overline{Oil1.pump!x}$ → Computation)
      [] (Oil2.take → $\overline{Oil2.pump!x}$ → Computation)
  Connector Customer_Cashier
    Role Givemoney = $\overline{pay!x}$ → Givemoney
    Role Getmoney = pay?x → Getmoney
    Glue = Givemoney.pay?x → $\overline{Getmoney.pay!x}$
      → Glue
  Connector Customer_Pump
    Role Getoil = $\overline{take}$ → pump?x → Getoil
    Role Giveoil = take → $\overline{pump!x}$ → Giveoil
    Glue = Getoil.take → $\overline{Giveoil.take}$ →
      Giveoil.pump?x → $\overline{Getoil.pump!x}$
      → Glue
  Connector Cashier_Pump
    Role Tell = $\overline{pump!x}$ → Tell
    Role Know = pump?x → Know
    Glue = Tell.pump?x → $\overline{Know.pump!x}$ → 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

(a)

Configuration GasStation
  Component Customer
    Port Pay = $\overline{pay!x}$ → Pay

    Computation = $\overline{Pay.pay!x}$ → $\overline{Gas.take}$ →
      Gas.pump?x → Computation
  Component Cashier
    Port Customer1 = pay?x → Customer1
    Port Customer2 = pay?x → Customer2
    Port Topump = $\overline{pump!x}$ → Topump
    Computation = Customer1.pay?x →
      $\overline{Topump.pump!x}$ → Computation []
      Customer2.pay?x → $\overline{Topump.pump!x}$ →
      Computation
  Connector Customer_Cashier
    Role Givemoney = $\overline{pay!x}$ → Givemoney
    Role Getmoney = pay?x → Getmoney
    Glue = Givemoney.pay?x → $\overline{Getmoney.pay!x}$
      → 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

(b)

Figure 2. (a) An architectural specification in WRIGHT and (b) a backward slice of it.

2.2. Architectural behavior modeling

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 [19]. 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 ports. Moreover, WRIGHT uses an overbar to distinguish initiated events from observed events. For example, the `Customer` initiates `Pay` action (i.e. $\overline{pay}!x$) 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 not 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` port `Gas` and the `Customer_Pump` role `Getoil` are identical. 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 ($\overline{Know.pump}!x$).

Based on the formal WRIGHT architectural specification of a software architecture, we can infer the information flows within a component/connector by analyzing its computation/glue specification, and the information flows between a component and a connector by analyzing both port and role specifications belonging to the component and connector. For example, from the computation specification of component `Cashier`, we can find that information flows may transfer from either port `Customer1` or port `Customer2` to port `Topump` within the component because the `Cashier`, upon receiving the payment (i.e. `Customer1.pay?x`), turns the pump on (i.e. `Topump.pump!x`). We can also know, from the glue specification of connector `Cashier_Pump`, that information flows may transfer from role `Tell` to role `Know` within the connector because a `Cashier_Pump` tell (i.e. `Tell.pump?x`) must be transmitted to the `Cashier_Pump` know (i.e. $\overline{Know.pump}!x$). Moreover, we can further find that information flows may transfer from the port `Pay` of component `Customer` to the role `Givemoney` of connector `Customer_Cashier` because the `Customer` initiates `Pay` action (i.e. `pay!x`) while the `Cashier` observes it (i.e. `pay?x`) through the ports `Givemoney` (i.e. `Givemoney.pay?x`) and `Getmoney` (i.e. `Getmoney.pay!x`) of connector `Customer_Cashier`. As we will show in Section 4, such kinds of information can be used to construct the architectural flow graph of a software architecture for computing an architectural slice and chop.

In this paper we assume that a software architecture is 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.

3. ARCHITECTURAL SLICING AND CHOPPING

While program slicing and chopping is useful in change impact analysis, existing slicing and chopping techniques for conventional programming languages cannot be applied to architectural specifications straightforwardly due to the following reasons. Generally, the traditional definition of slicing or chopping is concerned with slicing or chopping programs written in conventional programming languages which primarily consist of variables and statements, and a slice or chop consists of only variables and statements. However, in a software architecture, the basic elements are components and their interconnections, but neither variables nor statements as in conventional programming languages. Therefore, to perform slicing and chopping at the architectural level, new slicing and chopping notions must be defined.

In this section, we propose some new notions for architectural slicing and chopping and show how an architectural slice or chop can be contributed to a change impact analysis problem. In the following, we first introduce some change impact analysis problems we are interested in, and then define some types of architectural slices and chops to fit these problems. Finally, we discuss the relations between architectural slices (or chops) and change impact analysis.

We first introduce the concepts of a reduced component, connector, and configuration that are useful for defining architectural slice and chop.

- Let $P = (C_m, C_n, C_g)$ be an architectural specification and $c_m \in C_m$, $c_n \in C_n$, and C_g be a component, connector, and configuration of P respectively. A *reduced component* of c_m is a component c'_m that is derived from c_m by removing zero or more elements from c_m. A *reduced connector* of c_n is a connector c'_n that is derived from c_n by removing zero or more elements from c_n. A *reduced configuration* of C_g is a configuration C'_g that is derived from C_g by removing zero or more elements from C_g.

The above definition shows that a reduced component, connector, or configuration of a component, connector, or configuration may equal itself in the case that none of its elements has been removed, or an *empty* component, connector, or configuration in the case that all its elements have been removed.

3.1. Architectural slicing

In a WRIGHT architectural specification, for example, a component's interface is defined to be a set of ports that identify the form of the component interacting with its environment, and a connector's interface is defined to be a set of roles that identify the form of the connector interacting with its environment. To understand how a component interacts with other components and connectors for 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, for example, 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) $c \in C_m$ is a component and E is a set of ports of c, or (2) $c \in C_n$ is a connector and E is a set of roles 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, and E may be the set of ports or just a subset of ports of the component. If the users are interested in examining a connector, they may use slicing criterion 2, and 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: (1) a *backward architectural slice* S_{bp} of P on a given slicing criterion (c, E) consists of all reduced components, connectors, and the configuration in P that might directly or indirectly affect the behavior of c through elements in E; (2) a *forward architectural slice* S_{fp} of P on a given slicing criterion (c, E) consists of all reduced components, connectors, and configuration that might be directly or indirectly affected by the behavior of c through elements in E; (3) a *unified architectural slice* S_{up} of P on a given slicing criterion (c, E) consists of all reduced components, connectors, and the configuration that might affect or be affected by, directly or indirectly, the behavior of c through elements in E.

Intuitively, an architectural slice may be viewed as a subset of the behavior of a software architecture, similarly 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. Moreover, according to the slice definitions above, we have the following property.

- Let $P = (C_m, C_n, C_g)$ be an architectural specification and let $\delta = (c, E)$ be a slicing criterion of P. A unified architectural slice S_{up} on δ is the combination of the backward architectural slice S_{bp} and forward architectural slice S_{fp} on δ, i.e. $S_{up} = S_{bp} \cup S_{fp}$.

3.2. Architectural chopping

In addition to some architectural slices described above, we also present a new slicing-like technique called *architectural chopping*.

- Let $P = (C_m, C_n, C_g)$ be an architectural specification. A *chopping criterion* for P is a pair (S, T) such that: (1) S is a set of ports of a component and T is a set of ports of another component; (2) S is a set of ports of a component and T is a set of roles of a connector; (3) S is a set of roles of a connector and T is a set of ports of a component; (4) S is a set of roles of a connector and T is a set of roles of another connector.

Note that the selection of a chopping criterion also depends on users' interests. If they want to examine the effect transmitted from one component to another, they may use chopping criterion 1 and the sets S and T may be the set of ports or just a subset of ports of these components. If they want to examine the effect transmitted from one component to a connector, they may use chopping criterion 2, and the set S may be the set of ports or just a subset of ports of the component and the set T 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. An *architectural chop* S_{cp} of P on a given chopping criterion (S, T) consists of all reduced components, connectors, and the

configuration in P that serve to transmit effects from a given source element in S to the given target element in T.

The elements in an architectural chop with respect to the chopping criterion (S, T) are a subset of both elements in the architectural forward slice with respect to slicing criterion (c, S) and the elements in the architectural backward slice with respect to the slicing criterion (c, T). In most cases, the size of an architectural chop is substantially smaller than the size of either the architectural forward or backward slice. Furthermore, the architectural chop with respect to (S, T) is, in general, a proper subset of the intersection of the forward architectural slice from S with the backward architectural slice from T.

3.3. Relating architectural slices and chops to change impact analysis

The architectural slices and chop defined above form the basis for change impact analysis of a software architecture. During change impact analysis of a software architecture, we are interested in some types of questions such as:

(1) If a change is made to a component c, what other components might be affected by c?
(2) What components might potentially affect a component c?
(3) If a component c is dynamically replaced, what are the potential effects of the replacement related to c?
(4) What are all the components that serve to transmit effects from a given source component s to a given target component?

In terms of our slice and chop definitions above, a backward architectural slice would contribute to answer the first question, a forward architectural slice would contribute to answer the second question, and a unified architectural slice would contribute to answer the third question. Moreover, an architectural chop would contribute to answer the fourth question.

4. PERFORMING CHANGE IMPACT ANALYSIS

Roughly speaking, the process of change impact analysis of an architectural specification is how to find some backward, forward, and unified architectural slices as well as some architectural chops of the specification by starting from a component and/or a connector of interest. In the following we demonstrate the process of performing change impact analysis at the architectural level. The process consists of three phases which include building the architectural flow graph, finding slices and chops over the graph, and mapping the slices and chops to the actual code of an architectural specification.

4.1. A scenario for change impact analysis

We first examine a scenario of our approach on change impact analysis of software architectures via architectural slicing and chopping.

Consider the Gas Station system whose WRIGHT specification is shown in Figure 2. Suppose a maintenance programmer wants to make a change on the component `cashier`, or more particularly,

the port `Topump` of `cashier` in the architectural specification in order to satisfy some new design requirements. The first thing the programmer has to do is to investigate which components and connectors directly or indirectly interact with component `cashier` through its port `Topump`. Without a change impact analysis tool, the programmer has to manually check the actual code of the specification to find such information. However, it is time consuming and error-prone even for a small size specification because there may exist complex dependence relations between components and connectors in the specification. If the programmer has a change impact analysis tool at hand, the work may probably be simplified and automated without the disadvantages mentioned above. In such a scenario, the impact analysis tool is invoked, which takes as input: (1) a complete architectural specification of the system; and (2) a port of the component `cashier`, i.e. `Topump` (this is actually an *architectural slicing criterion*). The tool then computes a backward, forward, or unified architectural slice with respect to the criterion to answer questions such as 'which components and connectors might affect, or be affected by, directly or indirectly, component `cashier` through port `Topump`?', and output such information to the programmer for his change management. 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 programmer can thus examine only the contents included in these slices to assess the effect of changes. In the remainder of Section 4 we explain the process of change impact analysis in more detail.

4.2. Building architectural flow graphs

The first phase of our change impact analysis is to build a flow graph model called the architectural flow graph for software architectures on which architectural slices can be computed efficiently.

- The *architectural flow graph* (AFG) of an architectural specification P is an arc-classified digraph $(V_{com}, V_{con}, Com, Con, Int)$, where V_{com} is the set of port vertices of P, V_{con} is the set of role vertices of P, Com is the set of component–connector flow arcs, Con is the set of connector–component flow arcs, and Int is the set of internal flow arcs.

In an AFG, vertices represent the ports of components and the roles of the connectors in an architectural specification, and arcs represent possible information flows between ports of components and/or roles of connectors in the specification. There are three types of information flow arcs in the AFG. *Component–connector flow arcs* represent information flows between a port of a component and a role of a connector in an architectural specification. Informally, if there is an information flow from a port of a component to a role of a connector in the specification, then there is a component–connector flow arc in the AFG which connects the corresponding port vertex to the corresponding role vertex. *Connector–component flow arcs* represent information flows between a role of a connector and a port of a component in an architectural specification. Informally, if there is an information flow from a role of a connector to a port of a component in the specification, then there is a connector–component flow arc in the AFG which connects the corresponding role vertex to the corresponding port vertex. *Internal flow arcs* represent internal information flows within a component or connector in an architectural specification. Informally, for a component, there is an internal flow from an input port to an output port, and for a connector, there is an internal flow from an input role to an output role.

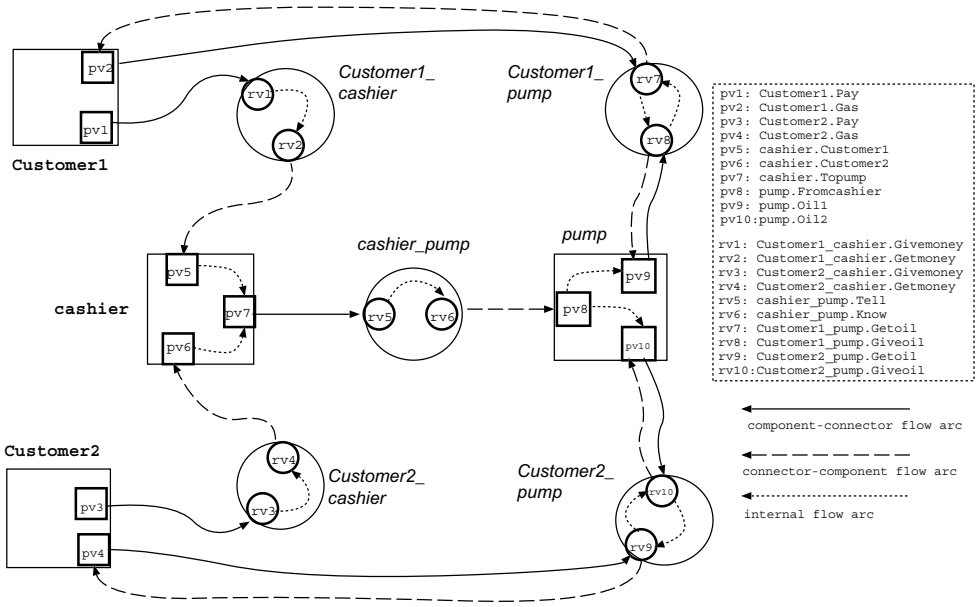


Figure 3. The AFG for the architectural specification in Figure 2.

As we introduced in Section 2, the information flow in a WRIGHT specification can be inferred statically. As a result, by using a static analysis tool which takes an architectural specification as its input, we can construct the AFG of a WRIGHT architectural specification automatically.

Figure 3 shows the AFG of the architectural specification in Figure 2. In the AFG, large squares represent components in the specification, and small squares represent the ports of each component. Each port vertex has a name described by *component_name.port_name*. For example, $pv5$ (`cashier.Customer1`) is a port vertex that represents the port `Customer1` of the component `cashier`. Large circles represent connectors in the specification, and small circles represent the roles of each connector. Each role vertex has a name described by *connector_name.role_name*. For example, $rv5$ (`cashier_pump.Tell`) is a role vertex that represents the role `Tell` of the connector `cashier_pump`. The complete specification of each vertex is shown on the right side of the figure.

4.3. Computing architectural slices and chops

The second and third phases of our change impact analysis is to compute some architectural slices and chops for recording change impact information. However, the slicing and chopping notions defined in Section 3 give us only a general view of an architectural slice or chop, and do not tell us how to compute it. In this section we present a two-phase algorithm to find an architectural slice or chop of an architectural specification based on its architectural flow graph. Our algorithm contains two phases:

(1) computing a slice S_g or a chop S_{cg} over the architectural flow graph of an architectural specification, and (2) constructing an architectural slice S_p from S_g or an architectural chop S_{cp} from S_{cg}.

4.3.1. Finding slices and chops over the AFG

To compute a slice or chop over the graph G, we refine the slicing and chopping notions defined in Section 3. Let $P = (C_m, C_n, C_g)$ be an architectural specification and $G = (V_{com}, V_{con}, Com, Con, Int)$ be the AFG of P. We first refine some slicing notions over the AFG.

- A *slicing criterion* for G is a pair (c, V_c) such that $c \in C_m$ and V_c is a set of port vertices corresponding to the ports of c, or $c \in C_n$ and V_c is a set of role vertices corresponding to the roles of c. A *backward slice* S_{bg} of G on a given slicing criterion (c, V_c) is a subset of vertices of G such that, for any vertex v of G, $v \in S_{bg}(c, V_c)$ iff there exists a path from v to $v' \in V_c$ in the AFG. A *forward slice* S_{fg} of G on a given slicing criterion (c, V_c) is a subset of vertices of G such that, for any vertex v of G, $v \in S_{fg}(c, V_c)$ iff there exists a path from $v' \in V_c$ to v in the AFG. A *unified slice* S_{ug} of G on a given slicing criterion (c, V_c) is a subset of vertices of G such that, for any vertex v of G, $v \in S_{ug}(c, V_c)$ iff there exists a path from v to $v' \in V_c$ or $v' \in V_c$ to v in the AFG.

We then refine some chopping notions over the AFG.

- A *chopping criterion* for G is a pair $\{(c, V_s), (c', V_t)\}$ such that: (1) $c \in C_m$ and V_s is a set of port vertices corresponding to the ports of c, or $c \in C_n$ and V_s is a set of role vertices corresponding to the roles of c; (2) $c' \in C_m$ and V_t is a set of port vertices corresponding to the ports of c', or $c' \in C_n$ and V_s is a set of role vertices corresponding to the roles of c'; and (3) c is different from c'. A *chop* S_{cg} of G on a given chopping criterion $\{(c, V_s), (c', V_t)\}$ is a subset of vertices of G such that, for any vertex v of G, $v \in S_{cg}$ iff there exists a path from $v' \in V_s$ to v and v to $v'' \in V_t$ in the AFG.

According to the above descriptions, the computation of a slice or chop over the AFG can be solved by using an usual depth-first or breadth-first graph traversal algorithm to traverse the graph by taking some port or role vertices of interest as the start point of interest. Figure 4 shows a backward slice over the AFG with respect to the slicing criterion $(\texttt{cashier}, V_c)$ where $V_c = \{pv7\}$. The slice can be computed by finding all vertices on the AFG that can reach to $pv7$ through a path. Therefore, we can obtain the slice as $\{pv1, pv3, pv5, pv6, pv7, rv1, rv2, rv3, rv4\}$ from two paths $(pv3 \rightarrow rv3 \rightarrow rv4 \rightarrow pv6 \rightarrow pv7)$ and $(pv1 \rightarrow rv1 \rightarrow rv2 \rightarrow pv5 \rightarrow pv7)$.

4.3.2. Deriving architectural slices and chops

A slice or chop computed above is only a slice or chop over the AFG of an architectural specification, which is a set of vertices of the AFG. Therefore we should map each element in the slice or chop to the actual code of the specification. Let $P = (C_m, C_n, C_g)$ be an architectural specification, $G = (V_{com}, V_{con}, Com, Con, Int)$ be the AFG of P, and $S_g \in \{S_{bg}, S_{fg}, S_{ug}, S_{cg}\}$. By using the concepts of a reduced component, connector, and configuration introduced in Section 3, a slice or chop of an architectural specification P can be constructed as follows.

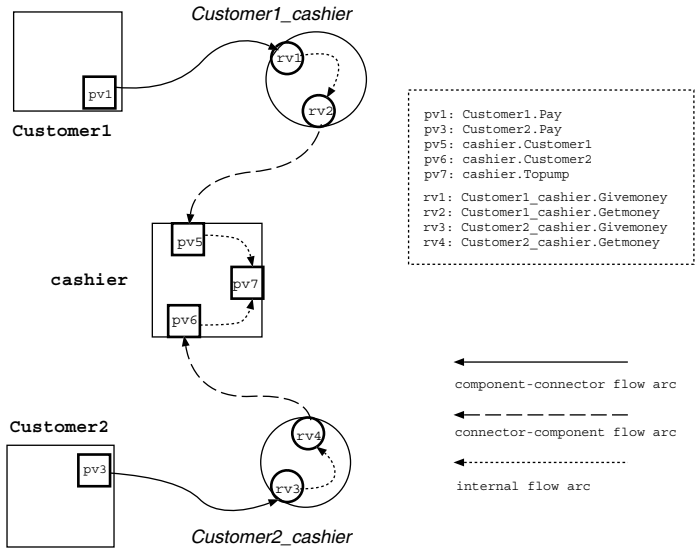


Figure 4. A backward slice over the AFG of the architectural specification in Figure 2.

1. Construct a reduced component c'_m from a component c_m by removing all ports such that their corresponding port vertices in G have not been included in S_g and removing unnecessary elements in the computation from c_m. For example, the following shows component `Customer` and its reduced component (with * mark) in which the port `Gas` has been removed:

Component Customer
Port Pay $= \overline{pay!x} \rightarrow$ Pay
Port Gas $= \overline{take} \rightarrow$ pump?x $\rightarrow$ Gas
Computation $= \overline{Pay.pay!x} \rightarrow \overline{Gas.take}$
$\rightarrow$ Gas.pump?x $\rightarrow$ Computation

*** Component** Customer
Port Pay $= \overline{pay!x} \rightarrow$ Pay
Computation $= \overline{Pay.pay!x} \rightarrow \overline{Gas.take}$
$\rightarrow$ Gas.pump?x $\rightarrow$ Computation

2. Construct a reduced connector c'_n from a connector c_n by removing all roles such that their corresponding role vertices in G have not been included in S_g and removing unnecessary elements in the glue from c_n. For example, the following shows connector `Customer_Cashier` and its reduced connector (with * mark) in which role `Givemoney` has been removed:

Connector Customer_Cashier
Role Givemoney $= \overline{pay!x} \rightarrow$ Givemoney
Role Getmoney $=$ pay?x $\rightarrow$ Getmoney
Glue $=$ Givemoney.pay?x $\rightarrow \overline{Getmoney.pay!x}$
$\rightarrow$ Glue

*** Connector** Customer_Cashier
Role Getmoney $=$ pay?x $\rightarrow$ Getmoney
Glue $=$ Givemoney.pay?x $\rightarrow \overline{Getmoney.pay!x}$
$\rightarrow$ Glue

3. Construct the reduced configuration C'_g from the configuration C_g by the following steps: (1) remove all component and connector instances from C_g that are not included in C'_m and

C'_n; and (2) remove all attachments from C_g such that there exist no two vertices v_1 and v_2 where $v_1, v_2 \in S_g$ and `v1 as v2` represents an attachment. For example, the following shows the configuration and its reduced configuration (with * mark) in which some instances and attachments have been removed:

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

*** 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

Figure 2(b) shows a backward architectural slice of the WRIGHT specification with respect to the slicing criterion (`cashier`, E) where E = {`Topump`} is a port of component `cashier`. The slice is obtained from a slice over the AFG in Figure 4 according to the mapping process described above, and consists of some reduced components and connectors as well as the reduced configuration of the original specification.

5. RELATED WORK

Much research has been carried out to support change impact analysis of software systems at the code level. Bohner and Arnold [2] edited a book which is a collection of many papers related to change impact analysis of software systems at the code level. Among these papers, program slicing is regarded as one of the basic operations for assessing the effect of changes for a software system (for detailed information on program slicing, refer to [20]). 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 to ours is that presented by Stafford *et al.* [21], 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 dependencies that can be followed during analysis. They proposed three types of chains, namely, the affected-by chain, the affects chain, and the related chain, which are similar to our forward slice, backward slice, and unified slice. However, in contrast to our flow graph-based approach to computing slices, they used a table-based approach to

identify the chain sets. Moreover, they did not consider the problem of architectural chopping which is also a useful technique to support change impact analysis. The concept of static architectural slicing was introduced by the first author [16] who computed an architectural slice as a reduced architectural specification that can preserve the partial semantics of the original architectural specification. Our work is different from his architectural slice notions in several ways: (1) in contrast to a reduced architectural specification our architectural slices only consist of components and connectors and do not require one to preserve the semantics of the original specification; (2) in addition to backward and forward slicing, we also consider the computation of unified architectural slices and architectural chops.

6. CONCLUDING REMARKS

In this paper, we have presented an approach to supporting change impact analysis of software architectures based on an *architectural slicing and chopping* 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 can be automated completely.

Our approach has shown that separation of change impact analysis of a system from the code level to the architectural level is an efficient step to reduce the cost of change impact analysis during software evolution, because this allows a maintenance programmer to assess the effect of changes of the system at the architectural level so that many implementation details of the system need not be considered. This is especially useful for large-scale software systems which consist of numerous components and connectors.

In architectural description languages, in addition to providing 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 environment as an essential means to support software architecture development and evolution.

To demonstrate the usefulness of our change impact analysis approach, we are developing an impact analysis tool, called `Ciasa`, for WRIGHT architectural specifications. The `Ciasa` is a prototype implementation of the techniques described in this paper and is written in C and working on UNIX. Although at present `Ciasa` can only handle WRIGHT architectural specifications, we are also considering the use of `Ciasa` to handle other ADLs such as Rapide. We also plan to perform some experiments using `Ciasa` to show the effectiveness of our approach to support architectural-level change impact analysis.

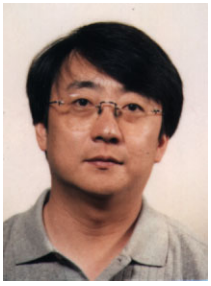
ACKNOWLEDGEMENTS

The authors would like to thank our guest editors, Tom Mens and Michel Wermelinger and the anonymous referees for their valuable suggestions and comments on earlier drafts of the paper.

REFERENCES

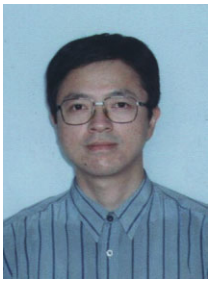
1. Yau SS, Collofello JS, MacGregor T. Ripple effect analysis of software maintenance. *Proceedings of COMPSAC'78*. IEEE Computer Society Press: Los Alamitos CA, 1978; 60–65.
2. Bohner SA, Arnold RS. *Software Change Impact Analysis*. IEEE Computer Society Press: Los Alamitos CA, 1996.
3. Lindvall M, Sandahl K. How well do experienced software developers predict software changes? *Journal of Systems and Software* 1998; **43**(1):19–27.
4. Parnas DS. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 1972; **15**(12):1053–1058.
5. Rombach HD. Design measurement: Some lessons learned. *IEEE Software* 1990; **7**(2):17–25.
6. Gallagher KB, Lyle JR. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering* 1991; **17**(8):751–761.
7. Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Language and System* 1990; **12**(1):26–60.
8. Loyall JP, Mathisen SA. Using dependence analysis to support the software maintenance process. *Proceedings Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1993; 282–291.
9. Shaw M, Garlan D. *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall: Englewood Cliffs NJ, 1996.
10. Allen R. A formal approach to software architecture. *Doctoral Dissertation*, Department of Computer Science, Carnegie Mellon University, Pittsburgh PA, 1997.
11. Luckham DC, Augustin LM, Kenney JJ, Veera J, Bryan D, Mann W. Specification analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering* 1995; **21**(4):336–355.
12. Shaw M, DeLine R, Klein DV, Ross TL, Young DM, Zelesnik G. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering* 1995; **21**(4):314–335.
13. Weiser M. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. *Doctoral Dissertation*, University of Michigan, Ann Arbor MI, 1979.
14. Jackson D, Rollins EJ. A new model of program dependences for reverse engineering. *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press: New York NY, 1994; 2–10.
15. Zhao J. Using dependence analysis to support software architecture understanding. *New Technologies on Computer Software*, Li M (ed.). International Academic Publishers: Beijing, 1997; 135–142.
16. Zhao J. Applying slicing technique to software architectures. *Proceedings 4th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society Press: Los Alamitos CA, 1998; 87–98.
17. Naumovich G, Avrunin GS, Clarke LA, Osterweil LJ. Applying static analysis to software architectures. *Proceedings of the Sixth European Software Engineering Conference*. ACM Press: New York NY, 1997; 77–93.
18. Helmbold D, Luckham D. Debugging Ada tasking programs. *IEEE Software* 1985; **2**(2):47–57.
19. Hoare CAR. *Communicating Sequential Processes*. Prentice-Hall: Englewood Cliffs NJ, 1985.
20. Tip F. A survey of program slicing techniques. *Journal of Programming Languages* 1995; **3**(3):121–189.
21. Stafford JA, Richardson DJ, Wolf AL. Aladdin: A tool for architecture-level dependence analysis of software systems. *Technical Report CU-CS-858-98*, Department of Computer Science, University of Colorado, Boulder CO, 1998.

AUTHORS' BIOGRAPHIES



Jianjun Zhao is an associate professor of Computer Science at Fukuoka Institute of Technology, Japan. He received BS and PhD degrees in Computer Science from Tsinghua University, China in 1987 and Kyushu University, Japan in 1997. His research interests include software architecture analysis, ubiquitous computing environment analysis, and aspect-oriented software development. He is a member of ACM and IEEE.

Hongji Yang is a Reader at the Software Technology Research Laboratory, School of Computing, De Montfort University, England and leads the Software Evolution and Re-engineering Group. His general research interests include software engineering and distributed computing. He served as a Program Co-Chair at IEEE International Conference on Software Maintenance '1999 (ICSM '99) and is serving as the Program Chair at IEEE Computer Software and Application Conference '2002 (COMPSAC 2002).



Limin Xiang received a BS, a ME, and a PhD in Computer Science from Nanjing University in 1982, the University of Electronic Science and Technology of China in 1988, and Kyushu University in 1999, respectively. He is currently an associate professor of Information Science at Kyushu Sangyo University. His research interests include algorithm design and analysis and parallel computation. He has published more then ten papers in international journals such as the Computer Journal and IEEE Transactions on Parallel and Distributed Systems. He was listed in Who's Who in Science and Engineering (2002–2003) and several other biographical publications.

Baowen Xu received MS and PhD degrees in computer science in 1984 and 2002. He is a professor in the Computer Science & Engineering Department of Southeast University, China. His current research interests include programming language, program analysis, program understanding, metrics, testing, Web search engine, and other topics related to software engineering. He has published more than 200 technical papers and 10 books. He is a PC Member of IEEE COMPSAC'2001 and COMPSAC'2002.