A Slicing-Based Approach to Extracting Reusable Software Architectures

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0295, Japan
Email: zhao@cs.fit.ac.jp

Abstract

An alternative approach to developing reusable components from scratch is to recover them from existing systems. Although numerous techniques have been proposed to recover reusable components from existing systems, most have focused on implementation code, rather than software architecture. In this paper, we apply architectural slicing to extract reusable architectural elements (i.e., components and connectors) from the existing architectural specification of a software system. Unlike traditional program slicing, which operates on the source code of a program to provide the low-level implementation details of the program, architectural slicing directly operates on the architectural specification of a software system, and therefore can provide useful knowledge about the high-level architecture of the system.

1 Introduction

Software reusability is widely accepted as the key to improving both the quality of software and the productivity of software engineers. Software reuse can take many forms – reuse of specification, designs, architecture, and code. However, although software reuse is a simple idea, it has not been accepted by software industry as a standard software engineering practice. One reason of that is the initial building of reusable software is a costly task. This leads to the reluctance of companies to adopt software reuse as an established practice in developing software. To make software reuse more acceptable, we must try to reduce its cost.

Instead of building reusable software from the beginning, we can recover new reusable components from existing software systems. Such a approach has a great potential because numerous software has already been written by programmers. Project managers do not expect the past knowledge and experience embodied in their software portfolio to be thrown away.

As reuse can take many forms as mentioned above, recovering new reusable components from existing software systems can also be performed at different level. However, previous work on this topic has mainly focused on the code level, that is, to recover new reusable components from the source code of a software system. In this paper, we consider the problem of recovering new reusable software components at the architectural level, that is, to recover new reusable architectural elements (i.e., components and connectors) from the existing architectural specification of a software system. While reuse of code is important, in order to make truly large gains in productivity and quality for software development, reuse of software architectures and patterns may offer the greater potential for return on investment. To extract reusable architectural elements from a formal architectural specification, we need a decomposition method which is able to group generally sets of architectural elements.

We believe that one way to support such work is slicing technique. Program slicing, originally introduced by Weiser [16], 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*. Program slicing has been studied primarily in the context of conventional programming languages. In such languages, slicing is typically performed by using a control flow graph or a dependence graph [5, 14, 17]. In addition to software reuse, program slicing has many other applications in software engineering activities including program understanding [4], debugging, testing, reverse engineering [2], and maintenance [6].

Using program slicing to support software reuse has been studied in [3, 11, 12] (for more detailed information, see related work section). However, previous slicing-based approaches to extracting reusable components have mainly focused on the code level, rather than the architectural level.

In this paper, we apply architectural slicing to extract reusable architectural elements from existing architectural specifications. Abstractly, our architectural slicing algorithm takes as input a formal architectural specification of a software system, then it removes from the specification those components and connectors which are not interested by the architect. The rest of the specification, i.e., its architectural slice, can thus be reused by the architect in a new system architecture design. This benefit allows one to rapidly reuse existing architecture design resources when performed architectural-level design.

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

The primary idea of architectural slicing has been presented in [17, 18], and this article can be regarded as an outgrowth of our previous work for applying this technique to architectural reuse. Moreover, the goal of this paper is to provide a sound and formal basis for our slicing-based architectural extraction approach 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: an architectural description language. Section 3 shows a motivation example. Section 4 defines some notions about extraction criteria. Section 5 shows how to extract reusable architectural elements. Section 6 discusses some related work. Concluding remarks are given in Section 7.

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 [13] as a sample to briefly introduce how to use WRIGHT to represent a software architecture. The specification is showed in Figure 2 which models the system architecture of a Gas Station system [8].

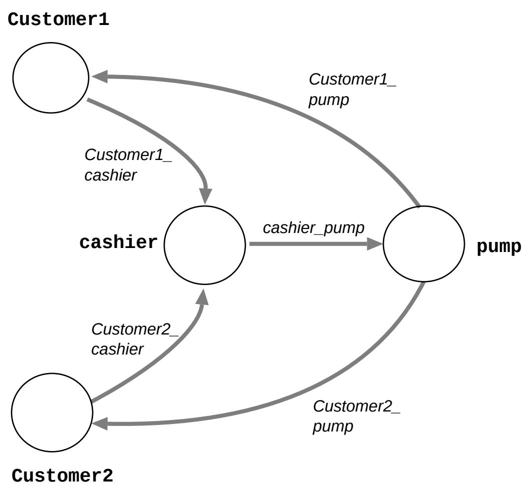

Figure 1: The architecture of the Gas Station system.

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 [7]. 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., $\underline{\texttt{pay!x}}$) 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.

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 ($\underline{\texttt{Know.pump!x}}$).

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. As we will show in Section 5, such kinds of information can be used to construct the architecture information 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 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 Motivation Example

We present a simple example to explain our approach on how to apply architectural slicing to extract reusable architectural functions from an existing architectural specification.

Consider the Gas Station system whose architectural representation is shown in Figure 1, and WRIGHT specification is shown in Figure 2. During the design process, suppose a system architect wants to use existing design resources to design a new system' architecture. Suppose the architect has the source code of architectural specification of the LAS system, the architect wants to reuse

[*]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].

the source. However, instead of reusing the whole specification, the architect wishes to use only a partial specification, that is, a functionality which is concerned with the component `cashier`. 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 architect 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 architect. 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 architect can thus reuse the partial architectural specification in the new system's architecture design.

4 Extraction Criteria

In this section, we give some informal definitions of an architectural slice. The formal definition of an architectural slice can be found in [18]

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 find an architectural slice $S_p = (C'_m, C'_n, c'_g)$ which should be a "sub-architecture" of P and preserve partially the semantics of P. To define the meanings of the word "sub-architecture," we introduce the concepts of a reduced component, connector and configuration.

Definition 4.1 *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 showed 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.

Having the definitions of a reduced component, connector and configuration, we can define the meaning of the word "sub-architecture".

Definition 4.2 *Let $P = (C_m, C_n, c_g)$ and $P' = (C'_m, C'_n, c'_g)$ be two architectural specifications. Then P' is a* reduced architectural specification *of P if:*

- *$C'_m = \{c'_{m_1}, c'_{m_2}, \ldots, c'_{m_k}\}$ is a "subset" of $C_m = \{c_{m_1}, c_{m_2}, \ldots, c_{m_k}\}$ such that for $i = 1, 2, \ldots, k$, c'_{m_i} is a reduced component of c_{m_i},*
- *$C'_n = \{c'_{n_1}, c'_{n_2}, \ldots, c'_{n_k}\}$ is a "subset" of $C_n = \{c_{n_1}, c_{n_2}, \ldots, c_{n_k}\}$ such that for $i = 1, 2, \ldots, k$, c'_{n_i} is a reduced connector of c_{n_i},*
- *c'_g is a reduced configuration of c_g,*

Having the definition of a reduced architectural specification, we can define some notions about slicing software architectures.

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 extract reusable architectural elements from an existing architectural specification, an architect needs to examine each port of the component of interest and each role of the connector. To satisfy these requirements, we regard 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.

Definition 4.3 *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 the start points from which architects start their extractions. For example, if they want to extract a partial specification starting from a component of an existing architectural specification, they can use the slicing criterion 1. If they want to extract a partial specification from a connector of an existing architectural specification, they can use the slicing criterion 2. Moreover, the determination of the set E also depends on architects' interests. If they want to start their extractions from a component, then E should be the set of ports or just a subset of ports of the component. If they want to start their extractions from a connector, then E should be the set of roles or just a subset of roles of the connector.

Definition 4.4 *Let $P = (C_m, C_n, c_g)$ be an architectural specification.*

- *A* backward architectural slice $S_{bp} = (C'_m, C'_n, C'_g)$ *of P on a given slicing criterion (c, E) is a reduced architectural specification of P which contains only those reduced components, connectors, and configuration that might directly or indirectly affect the behavior of c through elements in E.*
- Backward-slicing *an architectural specification P on a given slicing criterion is to find the backward architectural slice of P with respect to the criterion.*

Definition 4.5 *Let $P = (C_m, C_n, c_g)$ be an architectural specification.*

- *A* forward architectural slice $S_{fp} = (C'_m, C'_n, C'_g)$ *of P on a given slicing criterion (c, E) is a reduced architectural specification of P which contains only those reduced components, connectors, and configuration that might be directly or indirectly affected by the behavior of c through elements in E.*
- Forward-slicing *an architectural specification P on a given slicing criterion is to find the forward architectural slice of P with respect to the criterion.*

From Definitions 4.4 and 4.5, it is obviously that there is at least one backward slice and at least one forward slice of an architectural specification that is the specification itself. Moreover, the architecture represented by S_{bp} or S_{fp} should be a "sub-architecture" of the architecture represented by P.

Notice that in contrast to define an architectural slice as a set of components, here we define an architectural slice as a reduced architectural specification of the original one that consists of either components or connectors. Such a definition is particularly useful for extracting reusable architectural elements from the specification. By using an architectural slicer and giving some slicing criterion appropriately, an architect can automatically decompose an existing architectural specification into some small specifications each having its own functionality which can be reused into new architectural-level designs.

5 Extracting Reusable Architectures

Roughly speaking, the process of extracting reusable architectural functions is how to find some architectural slices. However, the slicing notions described in Section 4 give us only a general view of an architectural slice, and do not tell us how to compute it. In [18] we presented a two-phase algorithm to find an architectural slice based on the architecture information flow graph. Our algorithm contains two phases: (1) Finding a slice S_g over the architecture information flow graph of an architectural specification, and (2) Constructing an architectural slice S_p from S_g. In this section we first introduce the architecture information flow graph of software architectures briefly, and then describe the slicing algorithm.

The architecture information flow graph is an arc-classified digraph whose vertices represent the ports of components and the roles of the connectors in an architectural specification, and arcs represent possible information flows between components and/or connectors in the specification.

Definition 5.1 *The* Architecture Information Flow Graph *(AIFG) 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;*
- *Int is the set of internal flow arcs.*

There are three types of information flow arcs in the AIFG of a architectural specification, namely, *component-connector flow arcs*, *connector-component flow arcs*, and *internal flow arcs*, which represents information flows between a port of a component and a role of a connector, a role of a connector and a port of a component, and within a component or connector in an architectural specification.

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

Figure 3 shows the AIFG of the architectural specification in Figure 2. In the figure, 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, $rv5$

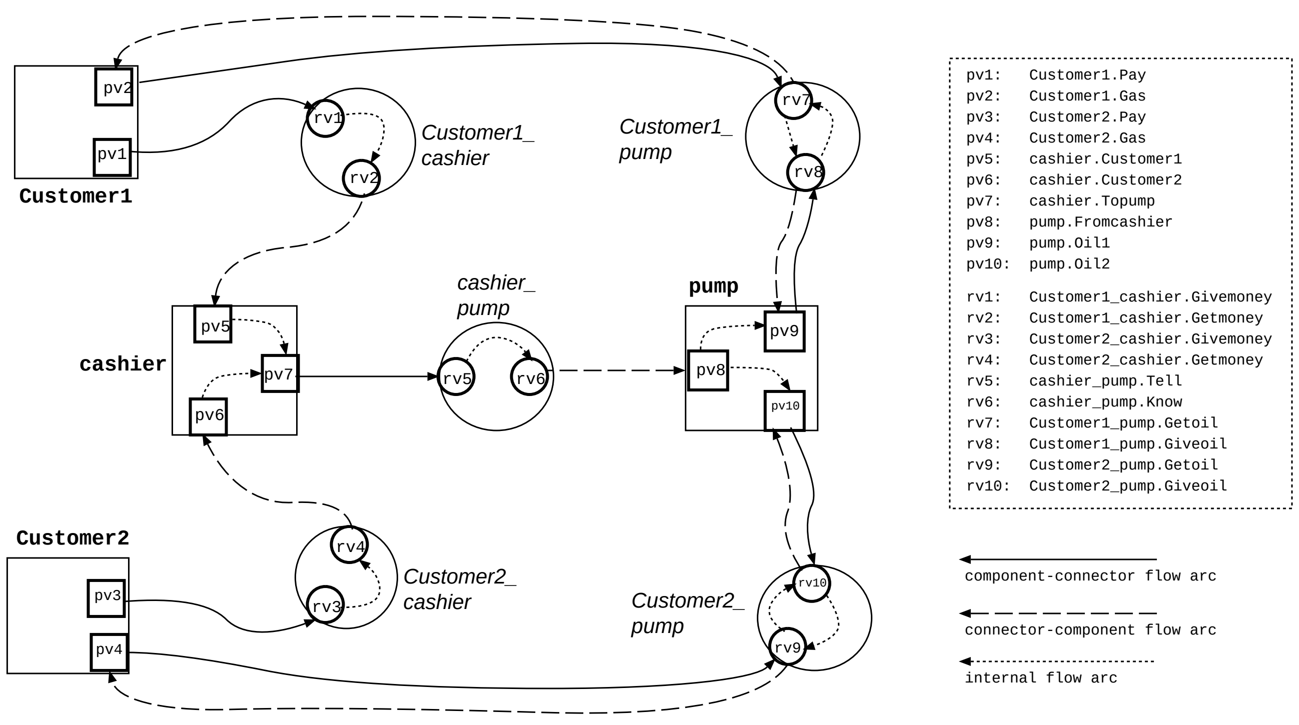

Figure 3: The information flow graph of the architectural specification in Figure 2.

(`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.

Solid arcs represent component-connector flow arcs that connect a port of a component to a role of a connector. Dashed arcs represent connector-component flow arcs that connect a role of a connector to a port of a component. Dotted arcs represent internal flow arcs that connect two ports within a component (from an input port to an output port), or two roles within a connector (from an input role to an output role). For example, $(rv2, pv5)$ and $(rv6, pv8)$ are connector-component flow arcs. $(pv7, rv5)$ and $(pv9, rv8)$ are component-connector flow arcs. $(rv1, rv2)$ and $(pv8, pv10)$ are internal flow arcs.

5.1 Finding a Slice over the AIFG

Let $P = (C_m, C_n, c_g)$ be an architectural specification and $G = (V_{com}, V_{con}, Com, Con, Int)$ be the AIFG of P. To find a slice over the G, we refine the slicing notions defined in Section 4 as follows:

- *A slicing criterion for G is a pair (c, V_c) such that: (1) $c \in C_m$ and V_c is a set of port vertices corresponding to the ports of c, or (2) $c \in C_n$ and V_c is a set of role vertices corresponding to roles of c.*

- *The backward slice $S_{bg}(c, V_c)$ 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 AIFG.*

- *The forward slice $S_{fg}(c, V_c)$ 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 AIFG.*

According to the above descriptions, the finding of a slice over the AIFG can be solved by using an usual depth-first or breath-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 slice over the AIFG with respect to the slicing criterion $(\texttt{cashier}, V_c)$ such that $V_c = \{pv5, pv6, pv7\}$.

5.2 Constructing an Architectural Slice

The slice S_{bg} or S_{fg} computed above is only a slice over the AIFG of an architectural specification, which is a set of vertices of the AIFG. Therefore we should map each element in S_{bg} or S_{fg} to the source code of the specification. Let $P = (C_m, C_n, c_g)$ be an architectural specification and $G = (V_{com}, V_{con}, Com, Con, Int)$ be the AIFG of P. By using the concepts of a reduced component, connector, and configuration introduced in Section 4, an architectural slice $S_p = (C'_m, C'_n, c'_g)$ of an architectural specification P can be constructed in the following steps:

1. Constructing 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_{bg} or S_{fg} and unnecessary elements in

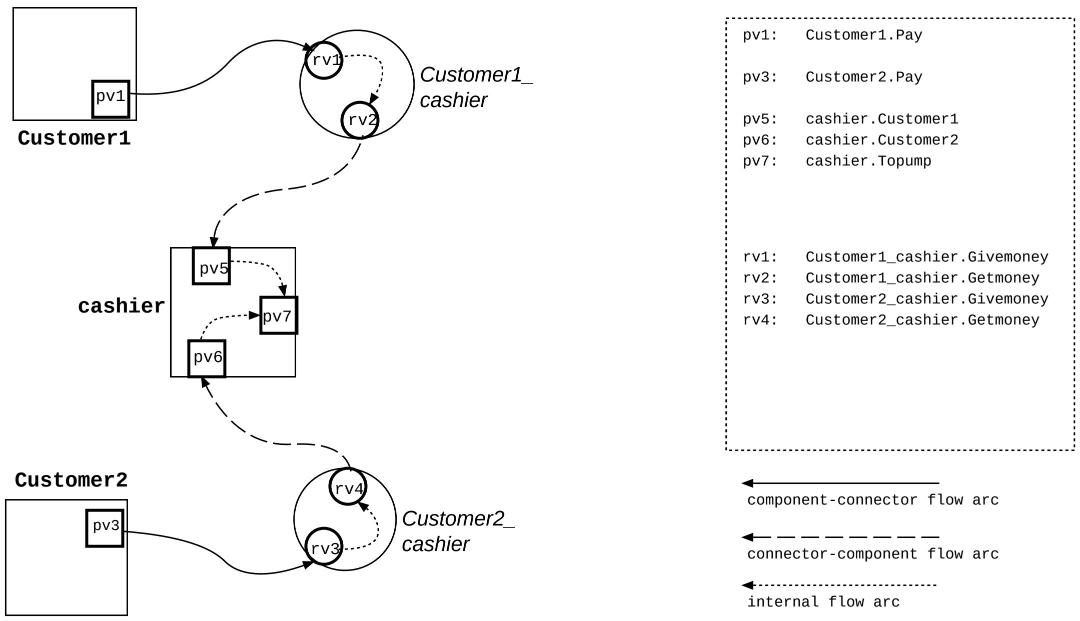

Figure 4: A slice over the AIFG of the architectural specification in Figure 2.

the computation from c_m. The reduced components C'_m in S_p have the same relative order as the components C_m in P.

2. Constructing 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_{bg} or S_{fg} and unnecessary elements in the glue from c_n. The reduced connectors C'_n in S_p have the same relative order as their corresponding connectors in P.

3. Constructing the reduced configuration c'_g from the configuration c_g by the following steps:

   – Removing all component and connector instances from c_g that are not included in C'_m and C'_n.

   – Removing all attachments from c_g such that there exists no two vertices v_1 and v_2 where $v_1, v_2 \in S_{bg}$ or $v_1, v_2 \in S_{fg}$ and `v1 as v2` represents an attachment.

   – The instances and attachments in the reduced configuration in S_p have the same relative order as their corresponding instances and attachments in P.

Figure 5 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`. The small rectangles represent the parts of specification that have been removed, i.e., sliced away from the original specification. The slice is obtained from a slice over the AIFG in Figure 4 according to the mapping process described above.

6 Related Work

Software reuse has been studied widely by researchers over the past decade. Although reuse of code has been widely studied, reuse of software architectures and patterns has not received as much as attention in comparing with reuse of code (for a detailed survey, see [10]). In this section, we review some related work on reusing software architectures as well as applying slicing technique to support software reuse. To the best of our knowledge, the work presented in this paper is the first time to apply slicing technique to extract reusable architectural elements from existing architectural specifications of software systems.

Monroe and Garlan [12] proposed a approach to support the reuse of software architectures. In [12] they discussed software reuse at the architectural level of design, and presented the concept of architectural style which is useful in supporting the classification, storage, and retrieval of reusable architectural design elements. While our focus is on extracting reusable architectural elements from an existing software system, they focused on the reuse problem on how to exploit the basic elements of architectural design (large-scale components and their interactions).

Lanubile and Visaggio [11] applied traditional slicing techniques to the problem of extracting reusable functions from ill-structured programs. They extended the definition of traditional slicing to the transform slicing which can find slices including statements that contribute directly or indirectly to transform a set of input variables into a set of output variables. Unlike tradi-

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

tional slicing, these statements do not include either the statements necessary to get input data or the statements which test the binding conditions of the function. Transform slicing presupposes the knowledge that a function is performed in the code and its partial specification, only in terms of input and output data. Also, they discussed that how to use domain knowledge to formulate expectations of the functions implemented in the code. Our work is rather inspired by their work of applying slicing technique to extract reusable functions from existing software systems. While they focused on the code level of a system, we focused on the architectural level of the system. However, we believe that the combination of two techniques is more promise for providing a powerful tool to support extract reusable components from an existing system not only at the code level but also at the architectural level.

Beck and Eichmann [2] introduced another slicing notion called *interface slicing*. Their interest in slicing is in the contexts of repositories and reusability. Because long-lived components frequently accrete much functionality over their lifetimes, making the comprehension required for modification or reabstraction increasingly difficult. Interface slicing could provide knowledge about a component to support its modification and eliminate the need for some manual reverse engineering efforts. However, interface slicing does not and can not be applied to slicing formal architectural specifications because in such a specification, source code concerning implementation details is usually not available.

7 Concluding Remarks

In this paper, we applied architectural slicing to extract reusable architectural elements from existing architectural specifications. Abstractly, our architectural slicing algorithm takes as input a formal architectural specification of a software system, then it removes from the specification those components and connectors which are not interested by the architect. The rest of the specification, i.e., its architectural slice, can thus be reused by the architect in a new system architecture design. This benefit allows one to rapidly reuse existing architecture design resources when performed architectural-level design.

The next step for us is to implement a tool based on the technique presented in this paper to support architectural-level reuse of software systems.

Acknowledgements

The author would like to thank the anonymous referees for their valuable suggestions and comments on earlier drafts of the paper.

References

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

[2] J. Beck and D. Eichmann, "Program and Interface Slicing for Reverse Engineering," *Proceeding of the 15th International Conference on Software Engineering*, pp.509-518, Baltimore, Maryland, IEEE Computer Society Press, 1993.

[3] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca, "Software Salvaging Based on Conditions," *Proceedings of the International Conference on Software Maintenance*, pp.424-433, Victoria, Canada, September 1994.

[4] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing," *Proceedings of the Fourth Workshop on Program Comprehension*, Berlin, Germany, March 1996.

[5] J.Ferrante, K.J.Ottenstein, and J.D.Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349, 1987.

[6] 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.

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

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

[9] 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.

[10] C. W. Krueger, "Software Reuse," *ACM Computing Surveys*, Vol.24, No.2, pp.131-183, June 1992.

[11] F. Lanubile and G. Visaggio, "Extracting Reusable Functions By Flow Graph-Based Program Slicing," *IEEE Transaction on Software Engineering*, Vol.23, No.4, pp.246-259, April 1997.

[12] R. T. Monroe and D. Garlan, "Style-Based Reuse for Software Architectures," *Proc. 4th International Conference on Software Reuse*, pp., 1996.

[13] 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.

[14] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.

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

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

[17] 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.

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