Using Dependence Analysis to Support Software Architecture Understanding

Jianjun Zhao

Department of Computer Science and Engineering

Fukuoka Institute of Technology

3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-02, Japan

zhao@cs.fit.ac.jp

Abstract

Software architecture is receiving increasingly attention as a critical design level for software systems. As software architecture design resources (in the form of architectural descriptions) are going to be accumulated, the development of techniques and tools to support architectural understanding, testing, reengineering, maintaining, and reusing will become an important issue. In this paper we introduce a new dependence analysis technique, named architectural dependence analysis to support software architecture development. In contrast to traditional dependence analysis, architectural dependence analysis is designed to operate on an architectural description of a software system, rather than the source code of a conventional program. Architectural dependence analysis provides knowledge of dependences for the high-level architecture of a software system, rather than the low-level implementation details of a conventional program.

1 Introduction

Software architecture is receiving increasingly attention as a critical design level for software systems [19]. 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. The importance of software architecture for practicing software engineers is highlighted by the ubiquitous use of architectural descriptions in system documentation.

Architectural 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. ADLs are intended to play an important role in the development of software by composing source modules rather than by composing individual statements written in conventional programming languages. Recently, a number of architectural description languages have been proposed such as ACME [8], Rapide [11], UniCon [18], and Wright [2] to support formally representation and reasoning of software architectures. As software architecture design resources (in the form of architectural descriptions) are going to

be accumulated, the development of techniques and tools to support understanding, testing, reengineering, maintaining, and reusing of software architectures will become an important issue.

One promising way to support software architecture development is to use dependence analysis technique. Program dependences are dependence relationships holding between program statements in a program that are determined by the control flows and data flows in the program. Usually, there are two types of program dependences in a conventional program, *control dependences* that represent the control conditions on which the execution of a statement or expression depends and *data dependences* that represent the flow of data between statements or expressions. The task to determine a program's dependences is called *program dependence analysis*. We refer to this kind of dependence analysis as *traditional dependence analysis* to distinguish it from a new form dependence analysis introduced later.

Traditional dependence analysis has been primarily studied in the context of conventional programming languages. In such languages, it is typically performed using *program dependence graphs* [4, 10, 15, 21, 22]. Traditional dependence analysis, though originally proposed for complier optimization, has also many applications in software engineering activities such as program slicing, understanding, debugging, testing, maintenance and complexity measurement [1, 3, 4, 15, 17, 21, 22].

Applying dependence analysis to software architectures promises benefit for software architecture development at least in two aspects. First, architectural understanding and maintenance should benefit from dependence analysis. To understand a software architecture to make changes during maintenance, a maintainer must take into account the many complex dependence relationships between components and/or connectors in the architecture. This makes dependence analysis an essential step to architectural level understanding and maintenance. Second, architectural reuse should benefit from dependence analysis. While reuse of code is important, reuse of software designs and patterns may offer the greater potential for return on investment in order to make truly large gains in productivity and quality. By analyzing dependences in an architectural description of a software

system, a system designer can extract reusable architectural descriptions from it, and reuse them into new system designs for which they are appropriate.

While dependence analysis is useful in software architecture development, existing dependence analysis techniques for conventional programming languages can not be applied to architectural descriptions straightforwardly due to the following reasons. The traditional definition of dependences only concerned with programs written in conventional programming languages which primarily consist of variables and statements as their basic language elements, and dependences are usually defined as dependence relationships between statements or variables. However, in an architectural description language, the basic language elements are primarily components and connectors, but neither variables nor statements as in conventional programming languages. Moreover, in addition to definition/use binding relationships, an architectural description language topically support more broad and complex relationships between components and/or connectors such as pipes, event broadcast, and client-server protocol. As a result, new types of dependence relationships in an architectural description must be studied based on components and connectors.

In this paper we introduce a new dependence analysis technique, named *architectural dependence analysis* to support software architecture development. In contrast to traditional dependence analysis, architectural dependence analysis is designed to operate on an architectural description of a software system, rather than the source code of a conventional program. Architectural dependence analysis provides knowledge of dependences for the high-level architecture of a software system, rather than the low-level implementation details of a conventional program.

The purpose of development of architectural dependence analysis is quite different from the purpose for development of traditional dependence analysis. While traditional dependence analysis was designed originally for supporting compiler optimization of a conventional program, architectural dependence analysis was primarily designed for supporting architectural understanding and reuse of a large-scale software system. However, just as traditional dependence analysis has many other applications in software engineering activities, we expect that architectural dependence analysis has also useful in other software architecture development activities including architectural testing, reverse engineering, reengineering, and complexity measurement.

The rest of the paper is organized as follows. Section 2 briefly introduces the ACME: an architectural description language. Section 3 presents a dependence model for software architectures. Section 4 discusses some applications of the model. Concluding remarks are given in Section 5.

2 Architectural Descriptions in ACME

We assume that readers are familiar with the basic concepts of architectural description languages, and in this paper, we use ACME architectural description language [8] as our target language to represent software architectures. The selection of the ACME is based on its potentially wide use because "it is being developed as a joint effort of the software architecture research community to provide a common intermediate representation for a wide variety of architecture tools." [8]

There are seven design elements in ACME that can be used to represent software architectures which include components, connectors, systems, ports, roles, representations, and bindings. Among them, the most basic elements of architectural description are *components*, *connectors*, and *systems*. Readers can refer [8] for more details of the language description, and we briefly introduce these design elements here.

Components are used to represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include clients, servers, filters, objects, and databases. Each component has its interface defined by a set of *ports*. A component may provide multiple interfaces by using different types of ports. Each port identifies a point of interaction between the component and its environment. A port can represent a simple interface such as procedure signature, or more complex interfaces, such as a collection of procedure calls that must be invoked in certain specified orders, or an event multi-cast interface point.

Connectors are used to represent interactions between components. Connectors mediate the communication and coordination activities between components. Intuitively, they correspond to the lines in box-and-line descriptions. connectors may represent simple forms of interaction, such as pipes, procedure calls, event broadcasts, and also more complex interactions, such as a client-server protocol or a SQL link between a database and an application. Each connector has its interface defined by a set of *roles*. Each role of a connector defines a participant of the interaction represented by the connector. Connectors may have two roles such as the *caller* and *callee* roles of an RPC connector, the *reading* and *writing* roles of a pipe, or the *sender* and *receiver* roles of a message passing connector, or more than two roles such as an even broadcast connector which might have a single *event-announcer* role and an arbitrary number of *event-receiver* roles.

Systems represent configurations of components and connectors.

Figure 5 (a) shows the ACME architectural description of a simple London Ambulance Service dispatch system (LAS system) which is taken from [14], and Figure 1 shows its architectural representation. The architectural representation contains five components which are connected by six connectors. For example, in the representation, the component `call_entry` and the component `incident_mgr` is connected by the connector `call_info_channel`. Each component is declared to have a set of ports, and each connector is declared to have a set of roles. For example, a component `incident_mgr` has four ports designed as `map_request`, `incident_info_request`, `send_incident_info`, and `receive_call_msg`, and a connector `call_info_channel` has two roles designed as `from` and `to`. The topology of the system is declared by a set of attachmentses. For example, an attachments `incedent_info_path` represents the con-
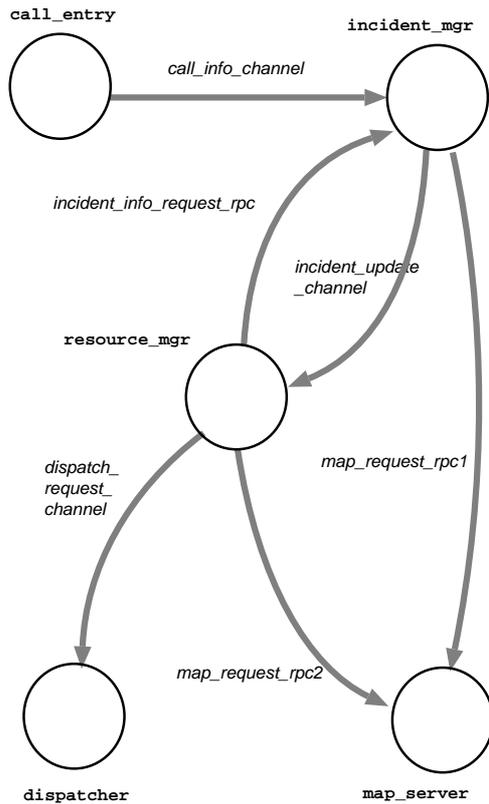
Figure 1: The architecture of the LAS system.

nections from calls to incident_manager, incident updates to resource manager, and dispatch requests to dispatcher.

In order to provide more information about architectural descriptions, ACME also supports annotation of architectural structure with lists of *properties*. Each property has a name, an optional type, and a value, and each ACME architectural design entity can be annotated. For example, in Figure 5, the connector **call_info_channel1** has a set of properties that state the connection type is massage passing channel and the message flow is from the role **from** to the role **to**.

In order to focus on the key idea of architectural dependence analysis, we assume that an ACME architectural description contains these basic elements including *component* whose interface is defined by a set of *ports*, *connector* whose interface is defined by a set of *roles* and *system* whose topology is declared by a set of *attachmentses* each including a set of attachments. *Representations* and *bindings* will not be considered here, and we will consider them in our future work. 5and A_m are the set of components, connectors, and attachmentses

3 A Dependence Model for Software Architectures

In this section we first introduce three types of dependences in an architectural description, then present a dependence graph for architectural descriptions.

3.1 Dependences in Architectural Descriptions

Traditional dependence analysis has been primarily studied in the context of conventional programming languages. In such languages, dependences are usually defined between statements or variables. However, in an architectural description language, the basic language elements are components and connectors, but neither statements nor variables. Moreover, in an architectural description languages, the interactions among components and/or connectors is through their interfaces that are usually defined to be a set of ports (for components) and a set of roles (for connectors). As a result, it is not enough to define dependences just between components and/or connectors in an architectural description. In this paper, we define dependences in an architectural description as dependence relationships between ports and/or roles of components and/or connectors. In the following, we present three types of dependences in an architectural description.

Component-Connector Dependences

The first type of dependence relationship in an architectural description is called *component-connector dependences* which can be used to represent dependence relationships between a port of a component and a role of a connector in the description. Informally, if there is an information flow from a port of a component to a role of a connector, then there exists a component-connector dependence between them. For example, in Figure 5 (a), there is a component-connector dependence between the port **receive_incident_info** of the component **resource_mgr** and the role **to** of the connector **incident_update_channel** since there is a message flow from the role **to** to the port **receive_incident_info**.

Connector-Component Dependences

The second type of dependence relationship in an architectural description is called *connector-component dependences* which can be used to represent dependence relationships between a role of a connector and a port of a component. Informally, if there is an information flow from a role of a connector to a port of a component, then there exists a connector-component dependence between them. For example, in Figure 5 (a), there is a connector-component dependence between the role **from** of the connector **call_info_channel** and the port **send_call_msg** of the component **call_entry** since there is a message flow from the port **send_call_msg** to the role **from**.

Additional Dependences

The third type of dependence relationships in an architectural description is called *additional dependences* which can be used to represent dependence relationships between two ports or roles within a component or connector. Informally, for a component or connector there are additional dependences from each
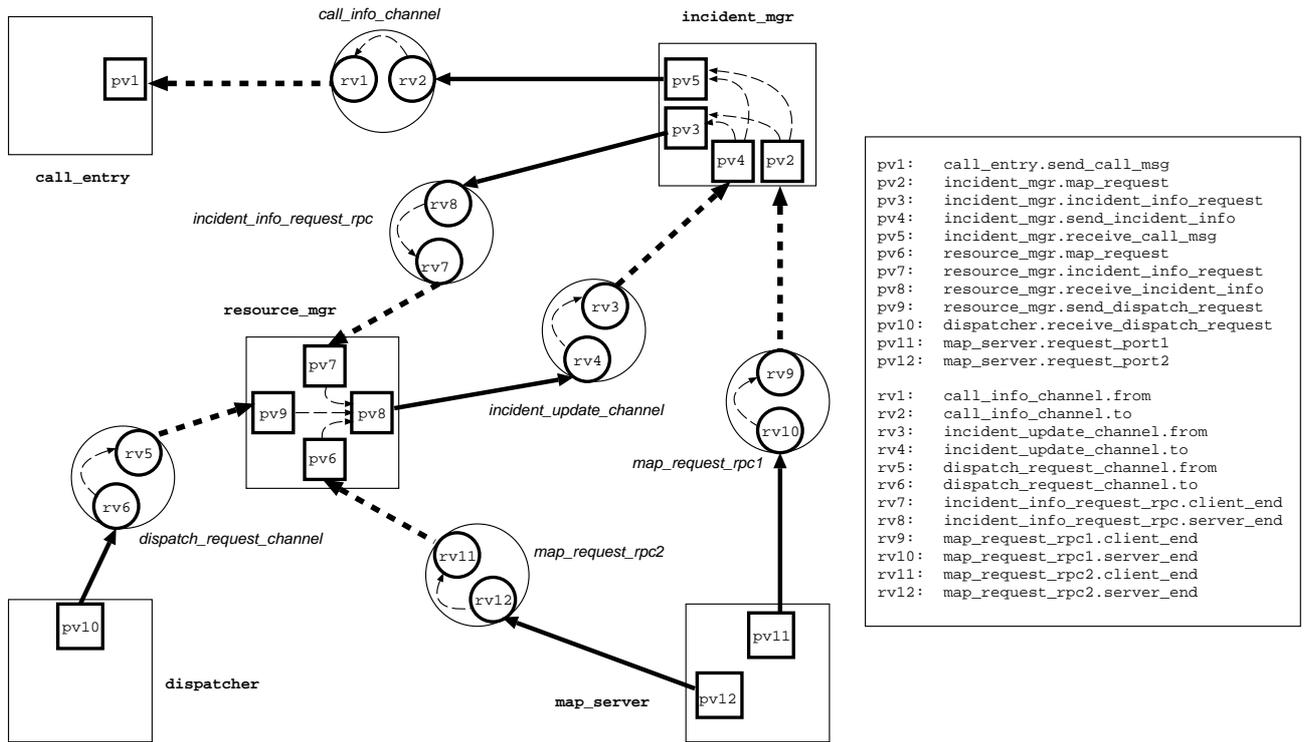
Figure 2: The dependence graph of the architectural description in Figure 5.

port or role as input to other ports or roles as output. For example, in Figure 5 (a), there is an additional dependence between the roles **client_end** and **server_end** of the connector **map_request_rpc2** and also an additional dependence between the ports **map_request** and **receive_incident_info** of the component **resource_msg**.

3.2 Software Architectural Dependence Graph

It has been shown that a dependence graph representation such as the *program dependence graph* (PDG) [6, 10] for programs written in conventional programming languages, has many application in software engineering activities since it provides a powerful framework for control flow and date flow analysis. This motivates us to present a similar representation to explicitly represent dependences in an architectural description. In this section, we present a dependence graph named *software architectural dependence graph* (SADG for short) for architectural descriptions to explicitly represent three types of dependences in an architectural description introduced above. The SADG of an architectural description is an arc-classified digraph whose vertices represent the ports of components and the roles of the connectors in the description, and arcs represent three types of dependence relationships in the description.

Figure 2 shows the SADG of the architectural description in Figure 5. In the figure, large

squares represent components in the description, and small squares represent the ports of each component. Each port vertex has its name described by *component_name.port_name*. For example, *pv8* (**resource_mgr.receive_incident_info**) is a port vertex that represents the port **receive_incident_info** of the component **resource_mgr**. Large circles represent connectors in the description, and small circles represent the roles of each connector. Each role vertex has its name described by *connector_name.role_name*. For example, *rv7* (**incident_info_request_rpc.client_end**) is a role vertex that represents the role **client_end** of the connector **incident_info request**. The complete description of each vertex is shown in the bottom of the figure.

Bold arcs represent component-connector dependence arcs that connect a port of a component to a role of a corresponding connector. Bold dashed arcs represent connector-component dependence arcs that connect a role of a connector and a port of a corresponding component. Thin dashed arcs represent additional dependence arcs that connect two ports or roles within a component or connector. For example, $(pv8, rv4)$ and $(pv3, rv8)$ are component-connector dependence arcs. $(rv5, pv9)$ and $(rv9, pv2)$ are connector-component dependence arcs. $(rv2, rv1)$ and $(rv6, rv5)$, and $(pv2, pv5)$ and $(pv7, pv8)$ are additional dependence arcs.

Note that there are some efficient algorithms to

```
pv1:    call_entry.send_call_msg

pv3:    incident_mgr.incident_info_request
pv4:    incident_mgr.send_incident_info
pv5:    incident_mgr.receive_call_msg
pv6:    resource_mgr.map_request
pv7:    resource_mgr.incident_info_request
pv8:    resource_mgr.receive_incident_info
pv9:    resource_mgr.send_dispatch_request

rv1:    call_info_channel.from
rv2:    call_info_channel.to
rv3:    incident_update_channel.from
rv4:    incident_update_channel.to


rv7:    incident_info_request_rpc.client_end
rv8:    incident_info_request_rpc.server_end
```
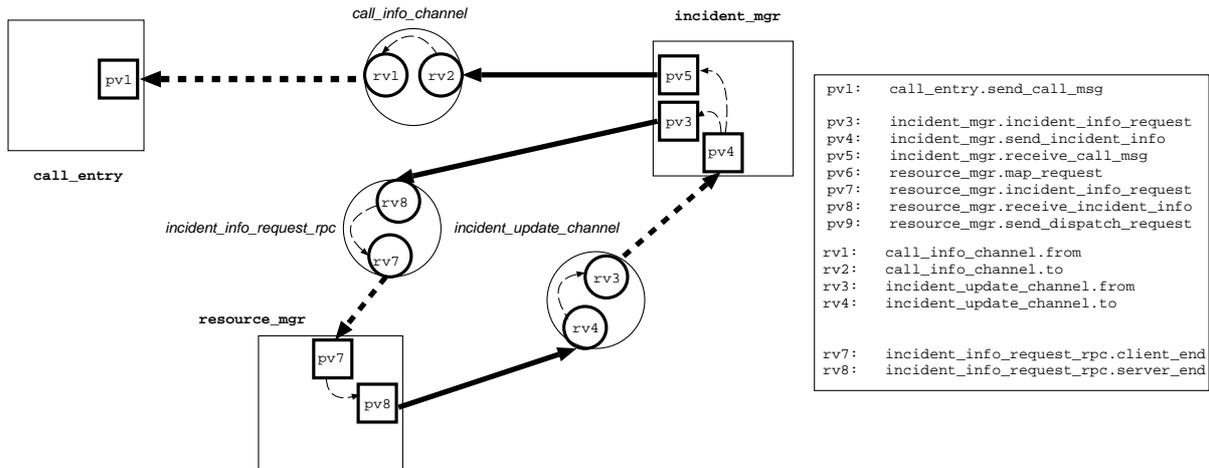
Figure 3: A slice over the ADDG of the architectural description in Figure 5.

compute program dependences and construct the dependence graph representations for programs written in conventional programming languages [9, 16]. These algorithms can easily be modified to compute dependences in an architectural description and construct the SADG representation as well.

4   Applications

As dependence graph representations for conventional programming languages have many applications in software engineering activities, the dependence model presented in this paper should have similar applications in practical development of software architectures.

4.1   Architectural Slicing and Understanding

Program slicing, originally introduced by Weiser [20], 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*. We refer to this kind of slicing as *traditional slicing*. Traditional slicing has been widely studied in the context of traditional programming languages and has many applications in software engineering activities such as program understanding [5], debugging [1], testing [3], maintenance [7] and complexity measurement [15].

Having SADG as a representation of architectural descriptions, we can apply traditional slicing technique to software architectures. We presented an entirely new form of slicing named *architectural slicing*, to slicing software architectures in order to support architectural understanding and reuse [23]. Architectural slicing is designed to operate on the architectural description of a software system and can provide

knowledge about the high-level architecture of a software system.

Intuitively, an *architectural slice* may be viewed as a subset of the behavior of a software architectural description, 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's ports or a connector's roles. Given an architectural description $P = (C_m, C_n, A_m)$ of a software system, our goal is to compute a slice $S_p = (C'_m, C'_n, A'_m)$ that should be a "subset" of P that preserves partially the semantics of P. In [23], We use a dependence graph based approach to compute an architectural slice, that is based on the SADG of the description. Our slicing algorithm contains two phases:

> **Step 1:** Computing a slice S_g over the SADG of an architectural description,

Figure 3 shows a slice over the ADDG in Figure 2. The slice was computed with respect to the slicing criterion (**resource_mgr**, V_c) such that $V_c = \{pv7, pv8\}$.

> **Step 2:** Constructing an architectural description slice S_p from S_g.

Figure 5 (b) shows a slice of the ACME description in Figure 5 (a) with respect to the slicing criterion (**resource_mgr**, E) such that E=\{incident_info_request, receive_incident_info\} is a set of ports of component **resource_mgr**. The small rectangles represent the parts of description that have been removed, i.e., sliced away from the original description. The slice is obtained from a slice over the ADDG in Figure 3 according to the mapping process described above. Figure 4 shows the architectural representation of the slice in Figure 5 (b).

In the following, we present a simple example to show how architectural slicing can be used to aid architectural understanding of a software system.
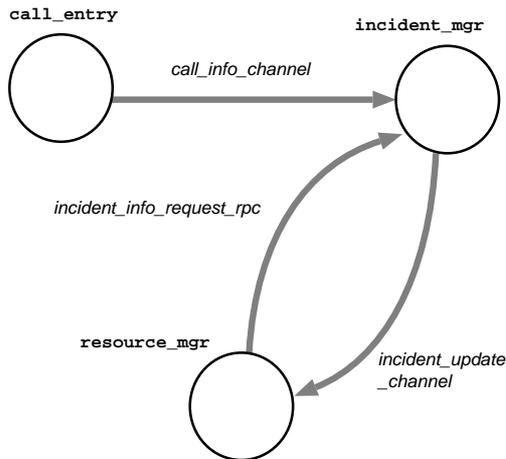
Figure 4: The architectural representation of the slice in Figure 5 (b).

Consider a simple London Ambulance Service dispatch system (LAS system) whose ACME description is shown in Figure 5 (a). This example is taken from [14]. Suppose a maintainer needs to modify two ports `incident_info_request` and `receive_incident_info` of the component `resource_mgr` in the architectural description in order to satisfy new design requirement, the first thing he/she has to do is to investigate which components and connectors interact with component `resource_mgr` through these two ports. A common way is to manually check the source code of the description to find such information. However, it is very time-consuming and error-prone even for a small size description because there may be complex dependence relations between components and/or connectors in the description. However, if the maintainer has an architectural slicer in hand, The work may probably be simplified and automated without the disadvantages mentioned above. In such a scenario, he/she only needs to invoke the slicer, which takes as input a complete architectural description of the system and the set of ports of the component `resource_mgr`, i.e., `incident_info_request`, `receive_incident_info` (this is an *architectural slicing criterion*). The slicer then computes an architectural slice with respect to the criterion and outputs it to the maintainer. Such a slice is a partial description of the original one which includes those components and connectors that might affect the component `resource_mgr` through ports in the criterion. The other parts of the description that might not affect the component `resource_mgr` have been removed, i.e., sliced away from the original description. The maintainer can thus focus his/her attention only on the contents included in the slice to investigate the impact of modification.

4.2 Architectural Reuse

While reuse of code is important, in order to make truly large gains in productivity and quality, reuse of software designs and patterns may offer the greater potential for return on investment. Although there are many researches have been proposed for reuse of code, little reuse method has been proposed for architectural reuse. By slicing an architectural description of a software system, a system designer can extract reusable architectural descriptions from it, and reuse them into new system designs for which they are appropriate.

5 Concluding Remarks

Software architecture is receiving increasingly attention as a critical design level for software systems. As software architecture design resources (in the form of architectural descriptions) are going to be accumulated, the development of techniques and tools to support architectural-level understanding, testing, reengineering, maintaining, and reusing will become an important issue. In this paper we introduce a new dependence analysis technique, named *architectural dependence analysis* to support software architecture development. In contrast to traditional dependence analysis, architectural dependence analysis is designed to operate on an architectural description of a software system, rather than the source code of a conventional program. Architectural dependence analysis provides knowledge of dependences for the high-level architecture of a software system, rather than the low-level implementation details of a conventional program. In order to perform architectural dependence analysis, we also presented the *software architectural dependence graph* to explicitly represent various types of dependences in an architectural description of a software system. While our initial exploration used ACME as the architectural description language, the concept and approach of architectural dependence analysis are language-independent. However, the implementation of an architectural dependence analysis tool may differ from one architecture description language to another because each language has its own structure and syntax which must be handled carefully.

In architectural description languages, in addition to provide both a conceptual framework and a concrete syntax for characterizing software architectures, they also provide tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions written in their associated language. However, existing language environments provide no tools to support architectural understanding, maintenance, testing, and reuse from an engineering viewpoint. We believe that a dependence analysis tool such as an architectural dependence analyzer introduced in this paper should be provided by any ADL as an essential means to support software architecture development activities.

As future work, we would like to extend our approach presented in this paper to handle other constructs in ACME language such as *templates* and *styles* which were not considered here, and also to extend our approach to handle other architecture description languages such as UniCon and Wright. More-

over, to demonstrate the usefulness of our dependence analysis approach, we are implementing an architectural dependence analyzer for ACME architectural descriptions to support architectural understanding and reuse. The next step for us is to perform some experiments to evaluate the usefulness of architectural dependence analysis in practical development of software architectures.

Acknowledgements

References

[1] H. Agrawal, R. Demillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software-Practice and Experience*, Vol.23, No.6, pp.589-616, 1993.

[2] R. Allen and D. Garlan, "The Wright Architectural Specification Language," Technical Report CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, 1996.

[3] S. Bates, S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pp.384-396, Charleston, South California, ACM Press, 1993.

[4] J. Cheng, "Dependence Analysis of Parallel and Distributed Programs and Its Applications," *Proceedings of 1997 IEEE-CS International Conference on Advances in Parallel and Distributed Computing*, pp.370-377, March 1997.

[5] 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.

[6] 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.

[7] 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.

[8] D. Garlan, R. Monroe, and D. Wile, "ACME: An Interchange Language for Software Architecture," 2nd edition, Tech. Report, Carnegie Mellon University, 1997.

[9] M. J. Harrold, B. Malloy, and G. Rothermel, "Efficient Construction of Program Dependence Graphs," *ACM International Symposium on Software Testing and Analysis*, pp.160-170, June 1993.

[10] 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.

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

[12] B. Korel, "Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol.24, pp.103-108, 1987.

[13] D. Kuck, R.Kuhn, B. Leasure, D. Padua, and M. Wolfe, "Dependence Graphs and Compiler and Optimizations," *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp.207-208, 1981.

[14] B. Monroe, D. Garlan, and D. Wile, "ACME BNF and Examples," *Microsoft Component-Based Software Development Workshop*, June 3-5, 1996.

[15] 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.

[16] K. J. Ottenstein and S. Ellcey, "Experience Compiling Fortan to Program Dependence Graphs," *Software - Practice and Experience*, Vol.22, No.1, pp.41-62, 1992.

[17] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transaction on Software Engineering*, Vol.16, No.9, pp.965-979, 1990.

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

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

[20] M. Weiser, "Program Slicing," *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357, 1984.

[21] J. Zhao, J. Cheng and K. Ushijima, "Program Dependence Analysis of Concurrent Logic Programs and Its Applications," *Proceedings of 1996 International Conference on Parallel and Distributed Systems*, pp.282-291, IEEE Computer Society Press, June 1996.

[22] J. Zhao, J. Cheng and K. Ushijima, "Static Slicing of Concurrent Object-Oriented Programs," *Proc. of the COMPSAC'96*, pp.312-320, IEEE Computer Society Press, August 1996.

[23] J. Zhao, "Software Architecture Slicing," *Proceedings of the 14th Annual Conference of Japan Society for Software Science and Technology*, September 1997 (to appear).

```
// Instance based example - simple LAS architecture:
System LAS_CAD = {
// system components
    call_entry = component {
        ports : { send_call_msg }
    }
    incident_mgr = component {
        ports : { map_request, incident_info_request,
                  send_incident_info, receive_call_msg }
    }
    resource_mgr = component {
        ports : { map_request, incident_info_request,
                  receive_incident_info, send_dispatch_request }
    }
    dispatcher = component {
        ports : { receive_dispatch_request }
    }
    map_server = component {
        ports : { request_port1, request_port2 }
    }

// system connectors
    // message passing connectors
    call_info_channel = connector {
        roles : { from, to }
        properties : { conn_type : string = message_pass_channel;
                  msg_flow : flow_direction = from -> to; }
    }
    incident_update_channel = connector {
        roles : { from, to }
        properties : { conn_type : string = message_pass_channel;
                  msg_flow : flow_direction = from -> to; }
    }
    dispatch_request_channel = connector {
        roles : {from, to }
        properties : { conn_type : string = message_pass_channel;
                  msg_flow : flow_direction = from -> to; }
    }
    // RPC connectors
    incident_info_request_rpc = connector {
        roles : { client_end, server_end }
        property : { conn_type : string = RPC; }
    }
    map_request_rpc1 = connector {
        roles : { client_end, server_end }
        property : { conn_type : string = RPC; }
    }
    map_request_rpc2 = connector {
        roles : {client_end, server_end }
        property : { conn_type : string = RPC; }
    }

// connect up the attachments
    incident_info_path = attachments : {
        // calls to incident_manager
        call_entry.send_call_msg to call_info_channel.to;

        // incident updates to resource manager
        incident_mgr.send_incident_info to
                  incident_update_channel.from;
        resource_mgr.receive_incident_info to
                  incident_update_channel.to;

        // dispatch requests to dispatcher
        resource_mgr.send_dispatch_request to
                  dispatch_request_channel.from;
        dispatcher.receive_dispatch_request to
                  dispatch_request_channel.to;
    }

    rpc_requests = attachments : {
        // calls to map server
        incident_mgr.map_request to map_request_rpc1.client_end;
        map_server.request_port1 to map_request_rpc1.server_end;
        resource_mgr.map_request to map_request_rpc2.client_end;
        map_server.request_port2 to map_request_rpc2.server_end;

        // incident info from incident_mgr
        resource_mgr.incident_info_request to
                  incident_info_request_rpc.client_end;
        incident_mgr.incident_info_request to
                  incident_info_request_rpc.server_end;
    }
}
                        (a)
```

```
// Instance based example - simple LAS architecture:
System LAS_CAD = {
// system components
    call_entry = component {
        ports : { send_call_msg }
    }
    incident_mgr = component {
        ports : { map_request, incident_info_request,
                  send_incident_info, receive_call_msg }
    }
    resource_mgr = component {
        ports : { map_request, incident_info_request,
                  receive_incident_info, send_dispatch_request }
    }

    ┌──────────────────────────────────────┐
    │                                      │
    └──────────────────────────────────────┘

    ┌──────────────────────────────────────┐
    │                                      │
    └──────────────────────────────────────┘

// system connectors
    // message passing connectors
    call_info_channel = connector {
        roles : { from, to }
        properties : { conn_type : string = message_pass_channel
                  msg_flow : flow_direction = from -> to; }
    }
    incident_update_channel = connector {
        roles : { from, to }
        properties : { conn_type : string = message_pass_channel
                  msg_flow : flow_direction = from -> to; }
    }

    ┌──────────────────────────────────────┐
    │                                      │
    └──────────────────────────────────────┘

    // RPC connectors
    incident_info_request_rpc = connector {
        roles : { client_end, server_end }
        property : { conn_type : string = RPC; }
    }

    ┌──────────────────────────────────────┐
    │                                      │
    └──────────────────────────────────────┘

    ┌──────────────────────────────────────┐
    │                                      │
    └──────────────────────────────────────┘

// connect up the attachments
    incident_info_path = attachments : {
        // calls to incident_manager
        call_entry.send_call_msg to call_info_channel.to;

        // incident updates to resource manager
        incident_mgr.send_incident_info to
                  incident_update_channel.from;
        resource_mgr.receive_incident_info to
                  incident_update_channel.to;

    ┌──────────────────────────────────────┐
    │                                      │
    │                                      │
    └──────────────────────────────────────┘

    rpc_requests = attachments : {
    ┌──────────────────────────────────────┐
    │                                      │
    └──────────────────────────────────────┘

        // incident info from incident_mgr
        resource_mgr.incident_info_request to
                  incident_info_request_rpc.client_end;
        incident_mgr.incident_info_request to
                  incident_info_request_rpc.server_end;
    }
}
                        (b)
```

Figure 5: An architectural description in ACME and a slice of it.