

# Deriving Tests From Software Architectures\*

Zhenyi Jin  
ITT Industries  
Advanced Engineering and Sciences  
1761 Business Center Drive  
Reston, VA 22090 USA  
jenny.jin@itt.com

Jeff Offutt  
ISE Department  
Software Engineering Research Lab  
George Mason University  
Fairfax, VA 22030-4444 USA  
www.ise.gmu.edu/faculty/ofut/  
ofut@ise.gmu.edu

*The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01), pages 308–313, Hong Kong, PRC, November 2001.*

## Abstract

*Software architectures are intended to describe essential high level structural and behavioral characteristics of the system. Architecture Description Languages (ADLs) describe these characteristics in ways that can be analyzed and manipulated algorithmically. This provides a unique opportunity for deriving tests at the system level. This paper defines formal testing criteria based on architecture relations, which are paths that architectural components use to communicate. The criteria have been applied to a specific ADL. Results from a comparative empirical study on industrial software are presented.*

## 1. Introduction

Software architecture is described by Shaw and Garlan [16] as a necessary step in raising the level of abstraction at which software is conceived and developed. Software architectures are viewed as a set of components together with a description of the interactions (called connectors) between these components. Architectural styles are a family of systems that share repeating patterns of computation and interaction, together with rules for how these are used in specific configurations. Software architectures let developers to abstract away the details of the individual components of an application, allowing them to be viewed as sets of components with associated connectors that describe the interactions between these components. Architecture Descrip-

tion Languages (ADLs) have been proposed as modeling and design notations to support analysis and development of architecture-based software. Most of them use formal approaches to architecture representations. ADLs provide a significant opportunity for dealing with the issue of scale with respect to testing and analysis of software systems.

Medvidovic and Rosenblum [10] presented an ADL survey. This survey classifies and compares properties in components, connections, and configurations, and how they are represented in these ADLs. Allen [1] introduced Wright, an ADL based on the formal description of the abstract behavior of architectural components and connectors, and showed how Wright provides a practical way to describe and analyze software architectures and architectural styles. Richardson, Wolf and Stafford introduced architecture-based dependency analysis and testing in the context of the CHAM (Chemical Abstract Machine) [12]. The behavioral dependencies allow one to relate states or interactions to other states or interactions. These relations are recorded in a table for dependency analysis. Rosenblum initiated the development of a component-based software testing theory [14, 13]. This technique is at the integration level rather than at the architectural level and it is also based on the completeness of the implementation. Ghosh [5] investigated issues in testing distributed component-based applications and proposed interface mutation testing.

This paper presents our work in developing a new technique for testing software, which is based on software architecture descriptions. As software architecture focuses more directly on the intended interactions between the high-level computational components of a system [2], we distinguish architecture testing from traditional system testing. System testing addresses issues pertaining to whether the overall system satisfies system requirements, whereas architecture testing attempts to test interactions at the architecture description level and then these tests may be reused at the

---

\*This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111.

implementation level when implementation information is provided. We define general software architecture-based testing criteria. The characteristics of individual ADLs are used to instantiate the general criteria in terms of the ADLs. For specific software architectures, then, the test criteria are used to define test requirements, which are finally used to generate inputs to test software. These criteria have been instantiated to create test criteria for the ADL Wright [1], and have been empirically evaluated on an industrial software system.

## 2. General Software Architectural Level Test Issues

Generally, software testing can be carried out at different levels, for instance, specification testing, system testing, acceptance testing, integration testing, unit testing, etc.. Testing goals and activities are different at different levels. Unit and integration level testing techniques use the software structure and data and control interactions among software methods and classes to define test adequacy criteria. Adequacy criteria are defined for testers to decide whether software has been adequately tested for a specific testing criterion [3]. Software architectures focus on the abstract components and an implementation-independent view of the interactions among these components, so traditional implementation-based testing criteria may not be possible at the software architecture level. Our testing focus is on how components and connectors interact with each other.

Although there are differences in how the term “software architecture” is defined, there is agreement on emphasizing the structure and relationships among elements of a system, rather than on implementation issues such as data structures and algorithms [1, 16]. There are four elements that are generally accepted to be part of the high-level architecture of a system: **components**, **interfaces**, **connectors** and **configuration**. **Components** are the computational units of a system – they have semantic meaning, named interfaces through which they expect to be able to interact, and often have constraints on the use of these interfaces. Component **interfaces** have formal descriptions, including the local control and data assumed. **Connectors** define the interactions in the architecture. An interface is defined [1] as a typed object that is a logical point of interaction between a component/connector and its environment. **Configurations** define how components, interfaces and connectors are combined to form an architecture.

Based on traditional data and control flow testing criteria [3], we propose to test data flow and control flow properties at the architecture level. An *initiation state* is the “start

state”; the state that the system is initially in. There are explicit *data flows* through the architecture of the system; a data element is given a value (*defined*) in its *source component* and the value is *used* in a *target component*. There are also explicit *control flows*; each architecture element has one or more designated *next element* on its control flow. We focus on the following properties of software architectures.

- **Data flow reachability:** A data element should be able to reach its designated target component from its source component through the connectors. The data element should reach the target component without having its value modified.
- **Control flow reachability:** Every architecture element should be able to reach its designated next element on its control thread.
- **Connectivity:** A component or connector interface with either no next element or no previous element is said to be “dangling”. Dangling components and connector interfaces indicate potential problems because they may not be connected to other components or connectors, which causes a discontinuity in the data or control flow.
- **Concurrency:** Interactions that in isolation are deadlock free can interact in such a way as to cause a deadlock situation. The system should be deadlock free.

## 3. A General Architecture-based Testing Technique

This section defines six architecture relations among architecture units that are based on the possible bindings of data transfer, control transfer, and execution sequencing rules. These relations are key factors in a software architecture description. The relations are then used to define architecture testing paths, which are then used to define architecture level testing criteria.

The underlying premise of the architecture-based testing criteria is that to achieve confidence in the architecture, it must be ensured that all the existing architecture relations are covered. Because this technique is limited to the architecture descriptions, it is only concerned with relations at the architecture level. In the following context, where “component (connector)” is used,  $N\_Interf$  must be replaced by  $C\_Interf$ , and by  $N1$  or  $N2$  by  $C1$  or  $C2$  when it means connector.

1. Component(Connector)\_Internal\_Transfer\_Relation ( $N.interf_1, N.interf_2$ ). The component (connector) internal transfer relation exists when there is a data or control transfer from  $N.interf_1$  to  $N.interf_2$ .

2. Component(Connector)\_Internal\_Sequencing\_Relation ( $N.interf_1, N.interf_2$ ) A component (connector) internal

sequencing relation exists if the behavior of component (connector) interfaces  $N.interf_1$  and  $N.interf_2$  have to follow some execution rules. For example, if the corresponding components (connectors) have to run in parallel or sequence, then the component (connector) interfaces have a sequencing relation. If two or more interfaces can process in any order, then there is no specific sequencing relation between the interfaces.

The interface sequencing rules contain three types of sequences:

1.  $(p1 \mid p2)$ : interface p1 has a data or control transfer to interface p2
2.  $(p1 \parallel p2)$ : components corresponding to interface p1 and p2 run in parallel
3.  $(p1 \implies p2)$ : components corresponding to interface p2 executes after component corresponding to p1 finishes

3.Component (Connector) Internal Relation( $N1.interf_1, N1.interf_2$ ) For a component(connector), an internal relation exists if there is either an internal transfer or internal sequencing relation between its interfaces  $N1.interf_1$  and  $N1.interf_2$ . Note that sequencing rules in an architecture description may be represented as constraints or implicitly described as part of the component behavior. For instance, in Wright, concurrent processing of each interface can be described by constraints.

4.  $N\_C\_Relation(N.interf_1, C.interf_1)$  or  $C\_N\_Relation(C.interf_1, N.interf_1)$  There is a relation between a component and a connector if an interface of a component  $N.interf_1$  is associated with an interface of a connector  $C.interf_1$ .

5.Direct\_Component\_Relation( $N_1.interf_1, C_1.interf_1, C_1.interf_2, N_2.interf_2$ ) A direct relationship between components  $N1$  and  $N2$  exists if  $N\_C\_Relation(N_1.interf_1, C_1.interf_1)$  exists,  $C\_N\_Relation(C_1.interf_2, N_2.interf_2)$  exists, and  $Connector\_Internal\_Relation(C_1.interf_1, C_2.interf_2)$  exists.

6. Indirect Component Relation. Two components  $N1$  and  $N3$  have an indirect component relation if  $Direct\_Component\_Relation(N1.interf_1, C1.interf_1, C1.interf_2, N2.interf_1)$  exists, and components  $N2$  and  $N3$  have a direction relation  $Direct\_Component\_Relation(N2.interf_2, C2.interf_1, C2.interf_2, N3.interf_2)$ , and  $Component\_Internal\_Relation(N2.interf_1, N2.interf_1)$  exists.

### 3.1. Architecture-based Testing Requirements

Six types of architecture-based testing **paths** are defined.  $C_i$  are connectors,  $N_i$  are components, and  $Interf_i$  are interfaces. Where “component (connector)” is used,

$N\_Interf$  must be replaced by  $C\_Interf$ , and by  $N1$  or  $N2$  by  $C1$  or  $C2$ .

**1. Component (Connector) internal transfer path:** Given a component  $N$ , there is a component internal transfer path if there is a *Component Internal Transfer Relation* ( $N.interf_1, N.interf_2$ ) between the two interfaces.

**2. Component (Connector) internal sequencing path:** If any interface of component  $N$  has a sequencing relation, the relation must have a rule or set of rules.

**3. Component to connector (N\_C) path or Connector to component (C\_N) path :** If  $N$  and connector  $C$  have a *N\_C\_Relation* ( $N.interf_1, C.interf_1$ ), there is a path from  $N.interf_1$  to  $C.interf_1$ . Or if there is a *C\_N\_Relation* ( $C.interf_1, N.interf_1$ ), there is a path from  $C.interf_1$  to  $N.interf_1$ .

**4. Direct component to component path:** If there is an architecture relation *Direct\_Component\_Relation* ( $N_1.interf_1, C_1.interf_1$ ) and a *C\_N\_Relation* ( $C_1.interf_2, N_2.interf_1$ ), there is a path from  $N_1.interf_1$  to  $C_1.interf_1$  to  $C_1.interf_2$  to  $N_2.interf_1$ .

**5. Indirect component to component path:** If there are relations that connect  $N_1, N_2, N_3, C_1,$  and  $C_2$  together, then the resulting path is:  $N_1 - C_1 - N_2 - C_2 - N_3$ .

**6. All connected components path:** Given components  $N_1, N_2, \dots, N_s$  and connectors  $C_1, C_2, \dots, C_t$ , if relations that connect these components and connectors together exist, then the resulting path is:  $N_1 - C_1 - \dots - C_t - N_s$ .

### 3.2. Architecture-based Testing Criteria

Five software architecture-based test criteria have been defined. They specify test requirements in terms of identified properties and relations of the specification of the software architecture, so that a test set is adequate if all the identified architecture relations have been fully exercised. They provide an increasing amount of coverage at more cost and time, that is, each criterion is formally subsumed by the following criterion [8]. In the criteria,  $T$  is a set of tests, or inputs, to the software that is implemented to satisfy the architecture.

- **AC1: Individual component interface coverage** requires that the set of paths executed by  $T$  covers all *Component Internal Transfer Paths* and *Component Internal Sequencing Paths* for each component.
- **AC2: Individual connector interface coverage** requires that the set of paths executed by  $T$  covers all *Connector Internal Transfer Paths* and *Connector Internal Sequencing Paths* for each connector.
- **AC3: All direct component-to-component coverage** requires that the set of paths executed by  $T$

covers all  $N\_C$  paths, all  $C\_N$  paths and all *Direct\_Component\_to\_Component\_Paths*.

- **AC4: All indirect component-to-component coverage** requires that the set of paths executed by  $T$  covers all *Indirect\_Component\_to\_Component\_Paths*.
- **AC5: All connected components coverage** requires that the set of paths executed by  $T$  covers all possible *All\_Connected\_Components\_Paths* for all the components in the architecture.

#### 4. Testing Criteria for Wright

These criteria are designed to apply to any ADL. The initial validation used Wright, which was chosen because components, connectors, and configurations are clearly specified and Wright has already been well studied.

In a Wright description, a *component* may have multiple *ports* (interfaces) and optionally, a *computation* to define what the component does and how the component uses the interactions described by the ports. A *connector* may have multiple *roles* and the *glue*. A role defines what is expected of any component that will participate in the interaction. An *attachment* defines the configuration by defining which component interfaces participate in which connector roles. The behavior and coordination of components is specified using a notation based on CSP [1, 7]. Here are some of the representations in Wright:

$e$ :	Event.
$e?x$ :	Process receives data $x$ .
$e!x$ :	Process outputs data $x$ .
$\S$ :	Successful termination of system.
$e \rightarrow P$ :	The process first engages in the event $e$ and then behaves as $P$ .
$\square$ :	External choice. Choices made deterministically by the environment.
$\sqcap$ :	Internal choice. Choice made non-deterministically by the process itself.
$;$ :	Combines two processes in sequence.
<i>where</i> :	Behavior patterns that occur repetitively can be described by naming particular processes.

For example, a Wright description of two components in a Read-Write model is shown in Figure 1. The description has two components, C1 and C2. C1 has two ports: port *In* reads data from an other source, and port *Out* writes the data that was just read from port *In*. *Out* also sends the data that was read to C2 to *In*. C2 has only one port, *In*. The connector links the roles *In* and *Out* together.

With the given architecture behavioral descriptions of components, connectors and their interfaces, we bring in a graphical representation, the Behavior Graph (BG), as shown in Fig 2, to represent the behavior and the relations

of related components. BG is based on the Petri Net theory [11, 15], we added *external choices* (deterministic choices) and *internal choices* (non-deterministic choices) to the Petri Net notion. The purpose for using the BG is that we can provide a visual model that can be executed for system modeling and simulation, also, we can apply existing Petri Net analytical tools and algorithms for our architectures. For instance, to evaluate the consistency and compatibility of component or connector requirements, making sure that constraints and rules that are associated with the components and connectors can be satisfied at the same time.

In the graph, a *place* (circle) represents a resource, or a condition; a *transition* (rectangle) stands for a process, or an event. the arcs connect a place to a transition, or vice versa. Places may have tokens to represent either the availability of the resource or the fulfillment of the condition. A transition can *fire* when all its input places contain at least one token, at which time a token from each input place is removed and a token is added to each output place.

The BG corresponding to Figure 1 is shown in Figure 2. There are two subnets for components C1 and C2. Subnet C1 has two port subnets for port C1 . In and C1 . Out. Subnet C2 has one port subnet for C2 . In. Places (circles) p3, p4 and p5 and corresponding light-dash arrows connect the two port subnets within C1 (computation of a component). Places p101, p102 and corresponding dark-dash arrows connect components C1 and C2 (connector *glue*).

Testing paths (in terms of BG paths in BG) can be derived when the architecture-based testing criteria are applied. These tests can be used both at the BG level for simulation and testing, and can be applied later at the implementation levels when implementation information is provided.

Due to space limitation, conversion from a Wright description to a BG is not presented in this paper. Related details can be found in Jin's dissertation [8] and Goltz and Reisig's paper [6]. Petri Net analysis techniques such as consistency checks, deadlock checks, and etc. [11] can be applied to check the relations between roles and ports. This is beyond the scope of this paper.

#### 5. Empirical Validation

A proof-of-concept tool, ABATT, was built to generate test paths derived from the architecture-based test criteria. ABATT was written in Java 1.2 to meet two major objectives: (1) to demonstrate the feasibility and effectiveness of the software architecture-based testing technique, and (2) to generate test requirements (in terms of test paths).

An experiment was carried out to compare the fault detection effectiveness of the architecture-based testing method with the coupling-based integration testing method

---

```

Component C1
  Port In = read?x → in □ close → §
  Port Out = write!x → Out □ Close → §
  Computation = (In.read?x → Out.write!x → Computation)
                □ (In.close → Out.close → §)
Component C2
  Port In = read?x → In □ close → §
  Computation
Connector CC
  Role Input = read?x → Input □ close → § □ fail → §
  Role Output = write!x → Out □ Close → §
  Glue = Output.write!x → Input.read?x → Glue
        □ close → §
Instances
  component1: C1
  component2: C2
  connect: C1-C2 Connector
Attachments:
  component1 provides as C1-C2.C1
  component2 provides as C1-C2.C2
end

```

---

**Figure 1. A Wright Example.**

[9], and with a specification-based testing method. A small-scale industrial program was used and three software professionals each generated tests for each of the testing technique used. Faults were created based on the architectural mismatch classifications by Gacek [4]. The experiment results show that the architecture-based testing technique could be effectively applied, and performed better than the other two techniques in terms of faults detected, but not in terms of faults detected per test. The experiment was limited in size of the software application there is no guarantee that the seeded faults were representative, and only one ADL was used.

## 6. Conclusions and Future Work

This paper presented a formal technique to test software systems at the architectural level, particularly for software systems developed using software Architecture Description Languages (ADL). Our long term research goal is to define general criteria for testing at the architectural level. These criteria are meant to be formalizable and automatable. It is hoped that these general criteria can be adapted to derive specific test criteria for specific ADLs. To demonstrate this approach, the initial general criteria have been defined, and adapted to derive criteria for the ADL Wright. This paper presented the general criteria, the Wright criteria, and results from an empirical study to validate the criteria on an industrial software system.

To use the architectural-based testing criteria in prac-

tice, it must be possible to relate structures that appear in the architectural description to corresponding structures in the software. The simplest way is to assume that the same names will be used in both locations. but a more general approach to be to require a mapping function from architectural names to program names. This is a problem that shows up in a variety of testing techniques that are based on designs or specifications.

Architecture-level testing offers two broad advantages to software developers. The first is that they have the potential to help us create higher quality software. Less obviously is that, by providing a potentially beneficial side-effect of using architecture descriptions, they provide additional motivation for using ADLs.

## References

- [1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh PA, May 1997. Technical Report CMU-CS-97-144.
- [2] Robert Allen and David Garlan. Beyond definition/use: Architectural interconnection. In *Proceedings of ACM Workshop on Interface Definition Languages*, pages 35–45, Portland, Oregon, January 1994.
- [3] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

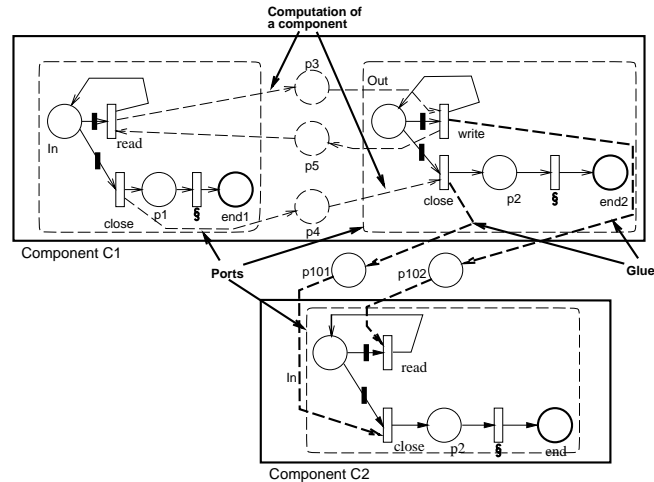


Figure 2. A BG Example.

- [4] C. Gacek and B. Boehm. Composing components: How does one detect potential architectural mismatches? In *Proceedings of the OMG-DARPA-MCC Workshop on Compositional Software Architectures*, pages 85–95, Los Angeles, CA, January 1998.
- [5] S. Ghosh. *Testing Component-Based Distributed Applications*. PhD thesis, Purdue University, West Lafayette IN, 2000.
- [6] U. Goltz and W. Reisig. Csp-programs as nets with individual tokens. *Lecture Notes in Computer Sciences*, 188:169–196, 1994.
- [7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [8] Z. Jin. *A Software Architecture-based Testing Technique*. PhD thesis, George Mason University, Fairfax VA, 2000. Technical report ISE-TR-00-??, <http://www.ise.gmu.edu/techrep>.
- [9] Zhenyi Jin and A. Jefferson Offutt. Coupling-based criteria for integration testing. *The Journal of Software Testing, Verification, and Reliability*, 8(3):133–154, September 1998.
- [10] Nenad Medvidovic and David S. Rosenblum. Domains of concern in software architectures and architecture description languages. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997. IEEE Computer Society.
- [11] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ, 1981.
- [12] Debra J. Richardson and Alexander L. Wolf. Software testing at the architecture level. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 68–71, San Francisco, CA, October 1996.
- [13] D. S. Rosenblum. Adequate testing of component-based software. Technical report 97-34, Department of Information and Computer Science, University of California, Irvine, CA, August 1997.
- [14] D. S. Rosenblum. Challenges in exploiting architectural models for software testing. In *Workshop on Evaluating Software Architectural Solutions – 2000*, Irvine, CA, May 2000. <http://www.isr.uci.edu/events/wesas2000/>.
- [15] G. Rozenberg and P. S. Thiagarajan. Petri nets: Basic notions, structure and behavior. *Lecture Notes in Computer Science*, 224:585–668, 1986.
- [16] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.