Analyzing Software Architecture Descriptions to Generate System-level Tests[1]

Aynur Abdurazik[†], **Zhenyi Jin**[‡], **A. Jefferson Offutt**[†], and **Elizabeth L. White**[§]

[†]ISE Department
Software Engineering Research Lab
George Mason University
Fairfax, VA 22030-4444 USA
+1-703-993-1654
{aynur,ofut}@ise.gmu.edu

[‡]ITT Industries
Advanced Engineering and Sciences
1761 Business Center Drive
Reston, VA 22090
jjin@sed.stel.com

[§]CS Department
Software Engineering Research Lab
George Mason University
Fairfax, VA 22030-4444 USA
+1-703-993-1586
white@cs.gmu.edu

1 Introduction

As the size and complexity of software systems increase, problems stemming from the design and integration of overall system structure become more significant than problems stemming from the choice of algorithms and data structures. This is due not only to the increased amount of code, but also to the need to distribute the parts of the application and have them interact in complex and potentially novel ways. Future software systems can be expected to continue to grow in size and complexity, which will greatly strain our already questionable ability to develop software that is both functional and reliable. At least for the foreseeable future, software architecture and object-oriented design will continue to be used to facilitate this growth. Software architecture allows 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. One product of software architecture research has been a set of formal and semi-formal languages that provide behavioral descriptions of the components and connectors. These architecture description languages (ADLs) represent a significant opportunity for dealing with the issue of scale with respect to testing and analysis of software systems.

Other researchers have looked into aspects of analysis and testing of software architectures, including developing integration test plans by using test criteria based on CHAM [3], dependence analysis techniques called chaining [6], and applying dependence analysis to problems in software maintenance [5]. We are currently developing criteria, techniques, and automated tools for the specification and generation of system level tests from architectural descriptions. This is an early life cycle technique that can be used to generate tests before design is completed. Although our focus is primarily on dynamic methods, we are also looking at static analysis techniques. This position paper presents initial observations about what properties should be tested at the architectural level. These properties are being used to develop formal system-level test criteria, which will then be used to generate test cases and evaluate externally-defined test cases.

2 Issues in Software Architecture-based Testing

Although there are differences in how the term "software architecture" is defined, there is agreement that the emphasis is on the structure and relationships between the elements of a system, rather than on implementation issues such as data structures and algorithms [4]. 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. The interfaces of a connector are typically associated with a given 'role' (for example client versus server) and the behavior of a connector "glues" these interfaces into an interaction. Components, interfaces and connectors are combined to form *configurations*.

[1] This work is supported in part by the National Science Foundation under grants CCR-98-04111 and CCR-96-25202.

There are many questions that can be asked about a software architecture and its corresponding implementations, including: (1) Are the individual connectors implemented correctly? (2) Does an individual component use its multiple interfaces correctly? (3) Is the configuration adequate to solve the problem? (4) Does the system as a whole (i.e. components and connections) work correctly?

We need to know what to test at the software architecture level so that test requirements can be defined. Lower level testing techniques use the software structure and data and control interactions among software methods and classes to define test adequacy criteria. 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 major focus when testing will be on the interactions among components.

3 General Properties To Be Tested at the Software Architectural Level

An initial step in developing new testing methods is to enumerate the kinds of problems that can exist. We have developed some preliminary architectural testing properties. It should be emphasized that this list is tentative and work is ongoing to refine the set of properties to test for at this level. In the list of properties, a *conflict* occurs when rules, constraints or semantics cannot both be satisfied at the same time. In general, *deadlock* implies that a process does not participate in any events, but has not yet terminated successfully. A process is *deadlock free* if it can never go into a deadlock state.

1. **Component Consistency Requirements**
   Semantics, constraints and interfaces can be associated with components. They should be consistent with respect to each other and this consistency needs to be considered at the architecture level. Interfaces have types as well as data and control constraints.

   - Component constraints and semantics should have no conflicts.
   - Component constraints and semantics should be deadlock free.
   - Component constraints and semantics should have no conflicts with the component interfaces constraints.

2. **Connector Consistency Requirements**
   A connector also contains interfaces, semantics, and constraints that need to be consistent. Interfaces have types as well as data and control constraints.

   - Connector constraints and semantics should have no conflicts.
   - Connector constraints and semantics should be deadlock free.
   - Connector constraints and semantics should have no conflicts with the associated connector interfaces constraints.

3. **Component-Connector Compatibility Requirements**
   Component interfaces are associated with connector interfaces to enable interactions. Informally, *compatibility* means that a component interface behaves in a manner that is consistent with assumptions made by the connector.

   - Component interfaces should be compatible with the associated connector interfaces.
   - For some compatibility requirements, it must be determined whether the component/connector relationship is deadlock free.

4. **Configuration Requirements**
   The configuration of a software architecture should be tested against several test requirements. 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*. This transfer of execution could be between states in a component, through connectors, or across components.

- 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.
- 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 could indicate potential problems.
- Interactions that in isolation are deadlock free can interact in such a way as to cause a deadlock situation. It should be the case that the system is deadlock free.

5. **Style Restriction Requirement**

   - The architecture style being used imposes some constraints on the software configuration. The system being used must satisfy those constraints.

4 Testing Criteria

System-level tests derived from architectures can validate that the software implements the architecture correctly and help to verify the architecture. If the architecture is sufficiently descriptive, then the tests should be effective at finding problems in the implementation. We are currently defining test criteria that are based on the properties in Section 3. These criteria are based on previous work in specification-based testing [1, 2], and are defined in terms of models of the architecture that are based on graphs and Petri nets.

References

[1] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. IEEE Computer Society Press.

[2] Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, Las Vegas, NV, October 1999. IEEE Computer Society Press.

[3] Debra J. Richardson and Alexander L. Wolf. Software Testing at Architecture Level. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 68–71, San Francisco, CA, October 1996.

[4] M. Shaw and D. Garlan. **Software Architecture: Perspectives on an Emerging Discipline**. Prentice Hall, 1996.

[5] Judith Stafford and Alexander L. Wolf. Architecture-Level Dependence Analysis in Support of Software Maintenance. In *Proceedings of the Third International Software Architecture Workshop (ISAW-2)*, pages 129–132, Orlando, FL, November 1998.

[6] Judith Stafford, Alexander L. Wolf, and Debra J. Richardson. Architecture Level Dependence Analysis for Software Systems. In *Proceedings of International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, Marsala, Sicily, Italy, July 1998.