Overview – Software Architecture

- High level of abstraction
- Components, connections, and configuration
- Focus on interactions
- Architecture Description Languages (ADLs)
Motivation

• **Software Architecture Research**
  – System complexity increases
  – Better understanding and handling of larger systems
  – Reuse potential

• **Lack of Testing Techniques at Architecture Level**
  – Early faults
  – No formal definition of test criteria
  – No general techniques

Issues in Software Architecture-Based Testing

• What general properties are important for testing at this level?
• Based on these general properties, what test requirements can be defined?
• Can general testing criteria be defined at this level?
• If so, what criteria?
Research Project Overview

1) Define general architecture-level test properties

2) Develop graphical representation

2) Define testing criteria

3A) Apply to ADL Wright

3B) Apply to other ADLs …

4) Develop prototype tool

Testing Properties

Component

semantics
constraints

Connector

semantics
constraints

Component

semantics
constraints
Testing Properties

- Component Internal Relation
  - \textit{Component\_Internal\_Relation} (N_1.interf_1, N_1.interf_2)

- Component Internal Transfer Relation
  - \textit{Component\_Internal\_Transfer\_Relation} (N.interf_1, N.interf_2)

- Component Internal Ordering Relation
  - \textit{Component\_Internal\_Ordering\_Relation} (N.interf_1, N.interf_2)

- Connector Internal Relation
  - \textit{Connector\_Internal\_Relation} (C.interf_1, C.interf_2)

- Connector Internal Transfer Relation
  - \textit{Connector\_Internal\_Transfer\_Relation} (C.interf_1, C.interf_2)

Testing Properties (2)

- Connector Internal Ordering Relation
  - \textit{Connector\_Internal\_Ordering\_Relation} (C.interf_1, C.interf_2)

- Component and Connector Relation
  - \textit{N\_C\_Relation} (N.interf_1, C.interf_1)

- Connector and Component Relation
  - \textit{C\_N\_Relation} (C.interf_1, N.interf_1)

- Direct Component Relation
  - \textit{Direct\_Component\_Relation} (N_1.interf_1, C_1.interf_1, C_1.interf_2, N_2.interf_1)

- Indirect Component Relation
  - \textit{Indirect\_Component\_Relation} (N_1.interf_1, C_1.interf_1, C_1.interf_2, N_2.interf_1, C_2.interf_1, C_2.interf_2, N_3.interf_1)
Graphical Representation

Interface Connectivity Graph

Testing Criteria

1. Individual component interface coverage
2. Individual connector interface coverage
3. All direct component-to-component coverage
4. All indirect component-to-component coverage
5. All connected components coverage
Testing Criteria Levels

Increasing levels of abstraction of concern

Application to ADL Wright

Component Client
Port Service = ClientPullT
Computation = Service.open ; UseOrExit
  where UseOrExit = UserService | Exit
  UseService = Service.request → Service.result?y → UseOrExit
  Exit = Service.close → §

Component Server
Port Provide = ServerPushT
Computation = WaitForClient □ Exit §
  where WaitForClient = □ Provide.open → Provide.receive-request →
  Provide.result?y
  → WaitForClient
  Exit = Provide.close → §

Interface Type ClientPullT = open → Operate □ §
  where Operate = request → result?x → Operate □ Close
  Close = close → §

Interface Type ServerPushT = open → Operate □ §
  where Operate = request → result!x → Operate □ Close
  Close = close → §
Application to ADL Wright (2)

Connector C-S Connector
Role Client = ClientPullT
Role Server = ServerPushT
Glue = Client.open → Server.open → Glue
      ∘ Client.close → Server.close → Glue
      ∘ Server.result?x → Client.result!x → Glue

Instances
  c: Client
  s: Server
  cs: C-S Connector
Attachments:
  c provides as cs.c
  s provides as cs.S

Behavior Graph (BG)

• The architectural description is modeled in a graphical representation called the behavior graph
• The BG uses a Petri net with extensions to model behavior
• It incorporates four kinds of paths:
  1. BG component behavior path (B-path)
  2. BG component connection path (C-path)
  3. BG interface interaction path (I-path)
  4. BG component indirect connection path (Indirect C-path)
Behavior Graph Example

Server Component

Client Component

Mapping Wright to BG

Rule 1: Events e are translated as $\text{Transf}[e]$

Rule 2: Events e?x are translated as $\text{Transf}[e?x]$

Rule 3: Events e!x are translated as $\text{Transf}[e!x]$

Rule 4: Process definitions $P = e \rightarrow P$ are translated as $\text{Transf}[P = e \rightarrow P]$

Rule 5: Event successful § is translated as $\text{Transf}[$§$]$

Rule 6: Sequential Composition $\circ$: $\text{Transf}[P_1 \rightarrow P_2] = \text{Transf}[P_1] \circ \text{Transf}[P_2]$

Rule 7: Non-deterministic (Internal Choice) Composition $+: \text{Transf}[P_1 \ box P_2] = \text{Transf}[P_1] + \text{Transf}[P_2]$

\[\vdots\]
Mapping Wright to BG (2)


Rule 11: Naming Composition "where": Transf[f → P where P = P₁] = Transf[f] ♦

Rule 12: Transf[∀x : S [ P(x)]] = Transf[P(x₁)] + Transf[P(x₂)] + … + Transf[P(xₙ)]

Rule 13: Transf[∃x : S Π P(x)] = Transf[P(x₁)] ♦ Transf[P(x₂)] ♦ … ♦ Transf[P(xₙ)]

Rule 14: Transf[∀x : S ; P(x)] = Transf[P(S)]

Rule 15: State Variables

Proof-of-Concept Tool
Validation – Procedure

For each architecture description $a$ and test adequacy criterion $c$:

1. Generate $c$-adequate test data set $T(a, c)$
2. Define fault set $F(a)$ for $a$
3. For each $f \in F(a)$ define the fault seeded architecture $A(f)$ by seeding $a$ with faults, yielding a fault-seeded architecture $a(f)$ where each $a(f) \in A(f)$
4. For each $t \in T(a, c)$, if it detects some faults, increase $\text{Num}(a, c)$ -- the number of faults detected by test data set $T(a, c)$
5. Determine the fault detection rate $R(a, c)$, for test adequacy criterion $c$ with respect to architecture $a$, as: $R(a, c) = \frac{\text{Num}(a, c)}{|F(a, c)|}$
   • Determine the fault detection effectiveness $E(a, c)$, for test adequacy criterion $c$ with respect to architecture $a$, as: $E(a, c) = \frac{R(a, c)}{|T(a, c)|}$

Validation – Subject Program

• Industrial program furnished by first author's company
• Responsible for processing external messages
• 8 software components (Java, Perl, C)
• 3 data sources
• Connections through internet and LAN
Validation – Faults Seeded

• 16 types of faults

• Communication, messages, shared resources, …

• Details in the paper

Results: Faults Detected

<table>
<thead>
<tr>
<th></th>
<th>Architecture-based</th>
<th>Manual / Specification</th>
<th>Coupling-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of test cases</td>
<td>24</td>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td>Faults found</td>
<td>14</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Faults not found</td>
<td>2</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>% faults found</td>
<td>87.5%</td>
<td>62.5%</td>
<td>50.0%</td>
</tr>
<tr>
<td>Test Effectiveness</td>
<td>59.3%</td>
<td>47.6%</td>
<td>57.1%</td>
</tr>
</tbody>
</table>

(C) Copyright 2001, Offutt & Jin. All Rights Reserved.
Conclusions

• A new general purpose software architecture-based testing technique
• Properties to be evaluated at the architectural level
• Formally defined general architecture-based testing criteria
• Testing criteria instantiated to a specific ADL, Wright
• Prototype tool and algorithm defined for the technique
• A Petri net based architecture modeling technique

Future Work

• Application to other ADLs

• Mapping architecture names to implementation names

• Catalog of architecture mistakes
  – Faults
  – Potential ramifications

www.ise.gmu.edu/faculty/ofut/