Perspective-based Architectural Approach for Dependable Systems

Sheldon X. Liang, J. Puett, Luqi

Naval Postgraduate School
May 2003
Perspective-based Architectural Approach for Dependable Systems

- Overview
- Perspective-based Architecting
- Dependable Compositional Patterns
- Conclusion
Objectives

- to develop a method known as PBA\(^1\)
- to incorporate rapid system prototyping (RSP)
- to build a synthesizing approach that enables
  - explicitly architecting HDSIS\(^2\)
  - consistently engineering HDSIS

---

\(^1\) PBA: Perspective-Based Architectural Approach

\(^2\) HDSIS: Highly Dependable Software-Intensive Systems
Barriers

- Perspective confusion problem
- Model construction problem
- Constraint localization problem
- Software tool support problem
They are not always coincident concerns for all stakeholders, sometimes they are even contradictory for customer, architect and implemen ter, respectively.
Model Construction

Modeling a system against different perspectives should reflect different stakeholder’s concerns, and it is required that these models are compatible.

A transitional process can be applied to change one into the other with a dependability-conserved transformation.
How to localize these constraints becomes key issue because it is not easy to find the crucial formal argument on which constraints are localized.

Dependable properties of HDSIS, such as availability, reliability, integrity, security, maintainability, are generally translated into quantitative constraints.
A well-formulated description provides the mechanism for reasoning and manipulation. Intellectual models are to be represented as semantic formulas that is suitable for reasoning and manipulation by CASE tools and this will be the main challenge.
Solutions

- Modeling HDSIS via multiple perspectives
- Explicit architecture via compositional patterns
- Property formulation via localized constraints
- System evolution via generic framework
PBA: Perspective-Based Architectural Approach

- **User’s Informal Needs**
  - Prototyping (Model)
  - System Architecting
  - Compositional Architecture
  - Behavioral Activities
  - Real-time Constraints
  - Architectural Properties
  - Design Inspection

- **Operational Concept**
  - Constructions (Prototype)
  - Component Evolving
  - Derivational Transformation
  - Generating (Framework)
  - Components and Connectivity

- **Highly Dependable Systems**
  - Refining (Coding)
  - Prototyping
  - Refining (Coding)
  - Evolving (Prototype)
  - Constructing (Prototype)
  - Prototyping (Model)

**Software Engineering Automation Center**

ICSE 2003 Workshop on Software Architecture for Dependable Systems

Naval Postgraduate School, 833 Dyer Road, Monterey, CA 93943-5118
Tel: (831) 656-3195
Email: seac@nps.navy.mil
http://seac.nps.navy.mil/
Fax: (831) 656-3225

May 19, 2003
Perspective-Based Architectures

- Computational Activity
- Compositional Architecture
- Derivational Implementation
- Transitional Procedure
Computational activity accounts for the customer perspective concerns of computation and interconnection.

\[
P_{\text{computation}} = [C_c, I, C_t (C_c, I)]
\]

Computational activity is used to capture the activities and information flows that will accomplish the operational concept.
Compositional Architecture

Compositional architecture accounts for the architect's perspectives of explicit treatment of system composition and architecture with constraints localized on compositional patterns.

\[ P_{\text{composition}} = [C_c \Rightarrow R, R_o \xrightarrow{S/P} R_p, \text{Ct} (R, S, P)] \]

Compositional architecture provides a set of rules (patterns) that governs the interactions among components.
Derivational Implementation

Derivational implementation accounts for the implementer's perspectives of component derivation and connectivity

\[ P_{\text{derivation}} = \{ R \supset C_p, (C_p \rightarrow^r R_0) \sim_{/P} (R_i \leftarrow C_p), \text{ Ct } (C_p S, P) \} \]

Derivational implementation identifies physical components and connectivity that will be instantiated to carry out the computational activity.
Transitional Procedure

\[ P \text{ computation} \quad \begin{array}{|c|c|c|c|} \hline P_{\text{composition}} & P_{\text{composition}} & P_{\text{derivation}} \\ \hline C_c & C_c \Rightarrow R & R \Rightarrow C_p \\ \hline I & R_c \Leftrightarrow R_i & (C_p \Rightarrow R) \Leftrightarrow (R \Rightarrow C_p) \\ \hline Ct(COM, INT) & Ct(R, S, P) & Ct(C_p, S, P) \\ \hline \end{array} \]

Explicit Architecting via Compositional patterns

Physical Derivation via PBA composers

Prototyping Analyzer

Pattern Selector

Framework Generator

Component Evolver
Dependable Compositional Patterns

Conceptual Model
Formal Representation
Substantiated Interconnections
Dependability vs Constraints
Compositional patterns provide a set of rules that govern the interactions among components with localized constraints.

Characterized as the interactions between two interactive roles via the architectural styles while complying with the communicatory protocols.
**Formal Representation**

composer Pipeline is generalized
  type Data is private;
  Size : Integer := 100;
  style as <#pipe-filter#>;
  protocol as <#dataflow-stream#>;
wrapper
role Outflow is
  port
    procedure Output(d: Data);
    procedure Produce(d: Data) is abstract;
  computation
    Produce (d);
    *[ Output (d) \rightarrow Produce (d) \diamond met(100) \rightarrow exception; ]
  end Outflow;
role Inflow is
  port
    procedure Input(d: Data);
    procedure Consume(d: Data) is abstract;
  computation
    *[ Input (d) \rightarrow Consume (d) \diamond mrt(100) \rightarrow exception; ]
  end Inflow;
collaboration (P : Outflow; C : Inflow)
  P•Produce(d);
  *[ P•Output(d) \rightarrow P•Produce(d) \setminus C•Input(d) \rightarrow C•Consume (d) ]
end Pipeline;
Substantiated Interconnections

Substantiating the interconnections among components deals with following four aspects:

- **Dependable composers** to promote interactions
- **Heterogeneous forms** to establish communication
- **Topological connectivity** to guide configuration
- **Constraint localization** to govern interconnections
Topological Connectivity

Topological connectivity simplifies the interconnection among components and comes in the following forms:

- **Fork (1~N):** single producer to multiple consumers
- **Merge (N~1):** multiple producers to single consumer
- **Unique (1~1):** single producer to single consumer
- **Hierarchy:** external producer to interact with the internal consumer, and vice versa
Topological Connectivity -- FORK
Dependability vs Constraints

This deals with the abstraction of dependability, its translation to quantitative constraints, and the handling of these constraints applied in the design, construction, and evolution of a software-intensive system.

<table>
<thead>
<tr>
<th>Dependability</th>
<th>Translation</th>
<th>Constraints</th>
<th>Localization</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Availability</td>
<td>• Consistency</td>
<td>• Consistency</td>
<td></td>
<td>• Role</td>
</tr>
<tr>
<td>• Reliability</td>
<td>• Compatibility</td>
<td>• Compatibility</td>
<td></td>
<td>• Style</td>
</tr>
<tr>
<td>• Security</td>
<td>• Granularity</td>
<td>• Granularity</td>
<td></td>
<td>• Protocol</td>
</tr>
<tr>
<td>• Integrity</td>
<td>• Heterogeneity</td>
<td>• Heterogeneity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Flexibility</td>
<td>• Real time</td>
<td>• Real time</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Synchronization</td>
<td>• Synchronization</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of Localized Constraints

**composer** Pipeline is generalized

... role Outflow is

**port**

**procedure** Output(d: Data);

**procedure** Produce(d: Data) is abstract;

**computation**

Produce (d);

* [Output (d) ◊ \( latency(60) \) \( \rightarrow \) Produce (d) ◊ \( met(100) \)

<table>
<thead>
<tr>
<th>latency-signaled</th>
<th>LAT-EXCEPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>met-signaled</td>
<td>MET-EXCEPTION</td>
</tr>
</tbody>
</table>

end Outflow;

... ...

end Pipeline;

---

**Latency**: the upper bound of communicating delay

**MET**: Maximum Execution Time of computation

Dynamical design inspection to monitor system execution
Conclusion

- Explicitly defined architectures promise: faster, better, cheaper systems
- PBA uncovers perspective concerns customer, architect, implementer
- PBA incorporates requirements validation prototyping / requirement adjustment
- PBA quantifies invariant architecture heterogeneity, granularity, compatibility
Thank you very much!

That is all

Questions?