Software Architecture: Past, Present, and Future

David Garlan
Carnegie Mellon University
University of Texas, Dallas
October 14, 2005

Examples of Architecture Descriptions

Figure 1. The NIST/ECMA reference model.
This Talk

- The Challenge for Software Architecture
  - Today’s practice
  - What is needed
- Research Themes (Part 1)
  - Notations and tools for software architecture
  - Architecture-based analysis
- Research Themes (Part 2)
  - Architecture-based dynamic adaptation
Joint Work

- Staff
  - Bradley Schmerl
- Graduate Students
  - Robert Allen
  - Owen Cheng
  - George Fairbanks
  - Vahe Poladian
  - Bob Monroe
  - Bridget Spitznagel

Issues Addressed by an Architectural Design

- Decomposition of a system into interacting components
  - typically hierarchical
  - using rich abstractions for component interaction or system “glue”

- Emergent system properties
  - performance, throughput, latencies
  - reliability, security, fault tolerance, evolvability

- Rationale and assignment of function to components
  - relates requirements and implementations

- Envelope of allowed change
  - “load-bearing walls”, limits of scalability and adaptation
  - design idioms and styles
The Challenge

How can we establish intellectual control over this world?

- Express architectural descriptions precisely and intuitively
- Provide soundness criteria & tools to check them
- Analyze architectural designs to determine key properties
- Exploit patterns and styles
- Guarantee conformance between architecture and implementation

Software Architecture in Context
Today’s Practice

- **Growing recognition of role of sw architecture**
  - Architect as distinct job title
  - Architectural design reviews part of sw devel processes
  - Investment in product lines and frameworks
  - Courses, textbooks, certificates, conferences

- **Standard notations and techniques**
  - UML 2.0
    - supporting object-oriented arch modeling
  - “Model-driven architecture,”
    - addressing platform independence
  - Middleware and integration standards
    - enabling component composition

But …

- **Notations are largely informal**
  - Meager analytical capability
  - No way to check/enforce compatibility with implementation
  - Hard to maintain architectural integrity over time

- **There are few tools for the architect**
  - Supporting scalability
  - Tailorable to domain and product family
  - Allowing flexible tool integration and analysis
  - Enabling code generation and conformance checking
Research Themes (Part 1)

- Formal representation of software architecture
  - Precise definition of high-level system designs
    - Identify design flaws early in lifecycle
    - Specify rules for domain-specific architectural frameworks
  - Architecture-based analyses
    - Reliability, performance, framework conformance,…

- Tools to support software architects
  - Graphical and textual interfaces for creating and maintaining architectures
  - Integration platform for architecture-based analyses and code generation for frameworks

Architectural Views

- There are many possible “views” of software architecture
  - Implementation structures
    - Modules, packages, work units
    - Uses, contains, specializes relations
  - Run-time structures
    - Components, connectors
    - Interactions, quality attributes
  - Deployment structures
    - Hardware, processes, networks

- We focus on Component & Connector (C&C) Views
Representing C&C Views

```
System simple-cs = {
    Component client = { port call-rpc; };
    Component server = { port rpc-request; };
    Connector rpc = {
        role client-side;
        role server-side;
    };
    Attachments = {
        client.call-rpc to rpc.client-side;
        server.rpc-request to rpc.server-side;
    }
}
```

Modeling Structure
Representations

- Provide hierarchical element abstractions
- Can represent (multiple) sub-architectures

Beyond Structure

- Annotate structure with properties
  - Quality attributes (e.g., performance, reliability)
  - Behavior (e.g., protocols of interaction)
  - Interface details (e.g., required and provided services)
- Properties can then be analyzed by tools
  - Schedulability analysis
  - Reliability analysis
  - Deadlock and race condition detection
Properties

```plaintext
System simple-cs = {
    ... 
    Component server = {
        port rpc-request = {
            Property sync-requests : boolean = true;
            Property max-transactions-per-sec : int = 5;
            Property max-clients-supported : int = 100;
        },
        Connector rpc = {
            Property protocol : string = "aix-rpc";
        };
    };
    ... 
}
```

Schedulability Analysis
Modeling Architecture Behavior

- **Key idea:** represent behavior as protocols
  - For connectors define separate protocols for each role and for the “glue” that binds them together
  - For components define protocols for ports and for the overall component behavior
- **Can then check (using model checkers)**
  - Consistency freedom of connectors
  - Compatibility of component interface to connector interaction protocol
  - Consistency of a component behavior to its interfaces

Representing Behavior

- Which is the reading/writing end of the pipe?
- Is writing synchronous?
- What if $F_2$ tries to read and the pipe is empty?
- Can $F_1$ choose to stop writing?
- Can $F_2$ choose to stop reading without consuming all of the data?
- If $F_1$ closes the pipe, can it start writing again?
- If $F_2$ never reads, can $F_1$ write indefinitely?
Specifying Connector Behavior

Wright: a variant of CSP (Hoare 85)

- **Events:** e, request, read?y, write!5
- **Processes:** P, Reader, Writer, Client, §
  - **Sequence:** e → P, P ; Q
  - **Choice:** P [Q, P Q]
  - **Composition:** P || Q

Example: A Pipe Connector

**Connector Pipe**

- **Role** Writer = (write!x → Writer) [ (close → §) ]
- **Role** Reader = Read [ Exit
  - **where** Read = (read?x → Reader) [] (eof → Exit)
    - Exit = close → §
- **Glue** = Writer.write?x → Glue []
  - Reader.read!y → Glue []
  - Writer.close → ReadOnly []
  - Reader.close → WriteOnly
  - **where** ...
Architectural Styles

- Architectural styles represent families of systems
  - Vocabulary of component and connector types (clients&servers, pipes&filters, …)
  - Properties of interest and shared analyses
  - Constraints on topology and properties

- Most systems are instances of styles
  - Sometimes generic (3-tired client-server, …)
  - Often domain-specific (power-train controllers, …)

Representing Styles

- Augment notation with
  - Component, connector, and property types
  - Constraints

- Constraints
  - First-order predicates over architecture structure and properties
  - Augmented with architectural primitives to simplify expressions
Styles/Families

**Family** PipeFilterFam = {
  **Component Type** filterT = {
    **Ports** {In,Out};
    ...
  }

  **Connector Type** pipeT = {
    **Role** Reader = {Property datatype = ...};
    **Role** Writer = {Property datatype = ...};
    **Invariant** self.Reader.datatype = self.Writer.datatype;
    ...
  }

  **System** myPF-System : PipeFilterFam = {...}
}

Example: MDS

- MDS defines an architectural framework for a family of NASA space systems
  - System of architectural component types
  - Rules on how they can be connected
  - Run-time infrastructure for executing MDS systems
  - Reusable code base
- Checking/ensuring conformance to MDS is an important and hard problem
  - Many rules, many components, complex topology
  - Mapping between architectural design and code is non-trivial
Formal Modeling of MDS

- Acme used to specify the MDS style
  - 8 Component types (sensor, actuator, estimator ...)
  - 12 Connector types (measurement query, command submit, state update)
- MDS rules defined using Acme constraints
  - Ten “rules” from MDS designers become 38 checkable predicates
- AcmeStudio for tool support
  - Eclipse-based graphical editor, constraint checker, tool plug-ins
Temperature Control System

The MDS Style

©David Garlan 2005
MDS Rules

- As specified by MDS designers:
  
  “For any given Sensor, the number of Measurement Notification ports must be equal to the number of Measurement Query ports (rule R5A).”

- Acme rule (associated with the sensor component type)

  numberOfPorts (self, MeasurementNotifReqrPortT) == numberOfPorts (self, MeasurementQueryProvPortT)
More MDS Rules

- Rule 4: “Every estimator requires 0 or more Measurement Query ports. It can be 0 if estimator does not need/use measurements to make estimates, as in the case of estimation based solely on commands submitted and/or other states. Every sensor provides one or more Measurement Query ports. *It can be more than one if the sensor has separate sub-sensors and there is a desire to manage the measurement histories separately.* For each sensor provided port there can be zero or more estimators connected to it. It can be zero if the measurement is simply raw data to be transported such as a science image. It can be more than one if the measurements are informative in the estimation of more than one state variable."

More MDS Rules

- As specified by MDS designers:
  
  “...*It can be more than one if the sensor has separate sub-sensors and there is a desire to manage the measurement histories separately....*”

- Acme rule (associated with the sensor component type):
  
  (numberOfPorts(self, MeasurementQueryPort) > 1) →
  
  self.manageHistoriesSeparately AND hasCommandableSubunits(self);
  
  *where* hasCommandableSubunits =  ...
On-going Work

- Scaling up to realistic systems
  - Thousands of components
- Tools to refine architectures to code
  - Ensure implementation conforms to architecture
  - Reuse large body of framework code
- Analyses
  - Schedulability, power consumption, footprint
  - Requirements coverage
Example: Distributed Simulation

- Distributed simulation
  - simulation is a multi-billion $ industry
  - critical problem for DoD (and others) is multi-vendor interoperability
  - envision ~1000 cooperating simulations
- The “High-Level Architecture” (HLA)
  - Defense Modeling and Simulation Office (DMSO)
  - standard -- about 250 pages
  - http://www.dmso.mil/docslib/ hla
  - each page defines 1 API call

Classification of Findings

- Ambiguity/imprecise wording 28
  - critical reading, Wright, other
- Inadequate pre-/post-conditions 12
  - critical reading
- Missing information 20
  - critical reading, Wright, FDR
- Race conditions 5
  - FDR, Wright
- Errors (invariant violation, unexpected conseqs) 11
  - critical reading, Wright, other
- Misc (typos, impl warnings, docn inconsistencies) 11
  - critical reading, Wright, FDR

87 issues
Example: Ford Model-based Design

- Worked with Ford Motor Company to develop tools for design of automotive control systems
- Two layered model
  - abstract, platform-independent
  - concrete, component model
- Tools to map between them
  - Component selection
  - Automatic “hook-up”
  - Creation of composite Simulink models
- Estimated savings
  - “what used to take 6 months now takes a week”
Beyond Static Analysis

- We are making great progress in design-time techniques for improving traditional systems
- But … increasingly, systems
  - are composed of parts built by many organizations
  - must run continuously
  - operate in environments where resources change frequently
- For such systems, traditional methods break down
  - Exhaustive verification and testing not possible
  - Manual reconfiguration does not scale
  - Off-line repair and enhancement is not an option

Research Themes (Part 2)

- Goal: systems automatically and optimally adapt to handle
  - changes in user needs
  - variable resources
  - faults
  - mobility

But how?

Answer: Move from open-loop to closed-loop systems

©David Garlan 2005
Example: University Grade System

- Students using University web
  - University aims to provide timely and ubiquitous access
  - One student tries to hack in and change her grades

- Possible (escalating) responses:
  - Turn on auditing
  - Switch authentication scheme
  - Sandboxing
  - Move grades data
  - Close off connections
  - Partition network
  - Turn off services

Many Things Can Go Wrong

- Resource variability
- Changing environments
- Shifting user needs and intents
- System faults

**The system should dynamically adapt to these problems.**
Traditional, Internal Mechanisms

Limitations
- Detection limited to localized view of system
- Outcome difficult to reason about
- Costly or infeasible to modify existing system
- Difficult to reuse logic for new system

- Exception handling
- Network time-outs
- Signal and interrupt
- Memory management

Data formatting in DB causes exception
Network failure causes time-outs
One application failure causes sig-HUP on the socket of another
Garbage collection

External Adaptation

- Global system perspective
- Important system-level behaviors and properties adapt
- Explicit system integrity constraints
- Proven trade-off analysis techniques

Architectural model & Adaptation mechanism

Running System

©David Garlan 2005
Rainbow Illustrated – Intrusion Detection

False! Find the right tactic

True?: intrusion_prob <= max_prob

Client2.isolate() / Grades.audit()

Client2.intrusion_prob = 75%

Change link / Add Auditing Possible grade change

©David Garlan 2005
Research Challenges

- One size does not fit all
- Ideally the approach should
  - apply to many architecture and implementation styles
    - Generality
  - facilitate adding self-adaptation capabilities to existing software systems at low cost
    - Cost-effectiveness
  - support run-time trade-off between multiple adaptation goals
    - Composability

Our *Rainbow* Approach (2)
Rainbow as a Tailorable Framework

- General framework with
  - Reusable infrastructure + tailorable mechanisms

  Specialized to targeted
  - system + adaptation goals

- Main components
  - Monitoring mechanisms
  - Model manager
  - Architectural evaluator
  - Adaptation engine
  - Effector mechanisms
  - Translation infrastructure

<table>
<thead>
<tr>
<th>What's tailored</th>
<th>Properties, probes &amp; gauges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vocabulary of model</td>
</tr>
<tr>
<td></td>
<td>Architectural constraints</td>
</tr>
<tr>
<td></td>
<td>Strategies &amp; tactics</td>
</tr>
<tr>
<td></td>
<td>System change operators</td>
</tr>
<tr>
<td></td>
<td>Arch-system mappings</td>
</tr>
</tbody>
</table>

Progress To Date

- Rainbow prototype
  - Developed and integrated mechanisms
  - Tested control cycle
  - Demonstrated usefulness for specific adaptation scenarios

- Case studies
  - Three styles of system
    - Client-server, service-coalition, data repository
  - Three kinds of adaptation goals
    - Performance + security + cost

- Adaptation language under development
Some Research Challenges

- **Modeling**
  - Architectural “recovery” at run time
  - Environment modeling and scoping
  - Handling multiple models and dimensions of concern
- **Capabilities of the adaptation infrastructure**
  - Efficient, scalable constraint evaluation
  - Timing issues (non-deterministic arrival of system observations, change latencies)
  - Avoiding thrashing
- **Advanced features**
  - Reasoning about the correctness of adaptation
  - Adapting the adaptation strategies

Other Software Architecture Research

- **Architectures for emerging systems**
  - *Pervasive computing* – thousands of heterogeneous computing elements
  - *Service oriented computing* – highly dynamic, highly distributed
- **Architecture conformance and discovery**
  - How can we ensure that a system has its advertised architecture?
- **Methods and processes**
  - Architecture-centric development
The END

Software architecture has come a long way. There remain many challenges. We examined two research threads

- Modeling architectures:
  - Representation and Analysis
  - Practical tools
- Run-time adaptation:
  - The reusable, tailorble Rainbow framework

Many more exist!

David Garlan
garlan@cs.cmu.edu