Today’s Lecture

- Software Architecture
  - Specification
  - Examples
    - Chemical Abstract Machine
    - C2

Architecture Specification

- Design Elements
- Form
  - Relationships among elements
- Rationale
  - Justification or arguments for choices of elements and form
- Constraints
  - Properties and weights

Design Elements

- Processing Elements
  - Components that transform data elements
- Data Elements
  - Information within a system
- Connectors
  - “Glue” that holds an architecture together
- A Useful Metaphor
  - Consider Polo, Water Polo, and Soccer: Similar in processors and data, but differ in connectors
Formal Specification

- Structure (Form)
  - How is the system organized?
- Function
  - What does the system compute?
- Compatibility
  - When is a system properly composed?
- Specializations
  - How are generic systems constrained?

Benefit of Formal Specs?
Analysis

- Consistency of Style Constraints
- Satisfaction of Style by Architecture
- Satisfaction of Requirements by Architecture and of Architecture by Implementation
- Consistency of Structure and of Behavior
- Effects of Changes

Chemical Abstract Machine: CHAM

- A Convenient Metaphor
  - Components are like molecules
  - Systems are like solutions
  - Molecules interact (i.e., react)
  - Rules govern interaction
  - State of system is like state of solution
- Mathematical Foundation
  - Term rewriting

CHAM Background

- Developed by Berry and Boudol in 1992
  - Used as a generalized computation framework
  - Has also been applied to parallel programming
- Applied to Software Architectures in 1995
  - by Paola Inverardi and Alex Wolf
  - extended to detect architectural mismatch; 1999
  - extended to static checking of system behaviors
    - to appear in ACM TOSEM
CHAM Terminology

• A CHAM is specified by
  – defining *molecules* m1, m2, …
  – and *solutions* s0, s1, … of molecules
    • think of a “chemical solution”
• Molecules are basic elements of a system
• Solutions represent states
  – and are represented by multisets of molecules

CHAM Terminology, continued

• A solution is denoted as a comma separated list of molecules enclosed in braces
  – { m1, m2, … }
  – A solution can contain sub-solutions
• CHAMs evolve via *transformation rules*
  – t1, t2, …
  – Transformations occur on solutions, thus moving a CHAM from state to state

Transformation Rules

• A transformation rule can be applied to a solution if it matches the rule’s condition
  – A condition is specified as a *premise* of the rule
• Rules are enabled if their condition is met
  – If multiple rules are enabled for a single solution, one of the enabled rules is selected non-deterministically to transform the solution
• *Inert* solution: no enabled rules

Specifying Software Architectures

• Using a CHAM to specify a software arch.
  – Molecules define a system’s components
  – Initial state of a system is defined by a solution
  – Transformation rules define system behavior
• In addition, a set of solutions can be specified to represent “legal” final states of a system
Example: Client-Server System

- Details
  - Consists of single server and single client
  - Server provides a single piece of data and the client requests that piece of data
- Later
  - we will extend the example to two clients

Example: Define syntax

- Syntax
  - $M ::= P \mid C \mid D \mid M \diamond M$
  - $P ::= \text{Server} \mid \text{Client1}$
  - $C ::= \text{serve}(D) \mid \text{request}(D)$
  - $D ::= \text{data}$

- Operator $\diamond$ indicates status of client/server
  - $\text{serve}(\text{data}) \diamond \text{Server}$
    - denotes that the server is ready to serve a client
  - $\text{Server} \diamond \text{serve}(\text{data})$
    - denotes that the server is unable to serve a client

Example: Define Initial Solution

- $s_0$
  - $\{ \text{serve}(\text{data}) \diamond \text{Server}, \text{request}(\text{data}) \diamond \text{Client1} \}$
- Server ready to serve data
- Client ready to request data

- Now we need transformation rules

Example: Define Rules

- $T_1$
  - $\text{serve}(\text{d}) \diamond p_1, \text{request}(\text{d}) \diamond p_2 \rightarrow$
  - $p_1 \diamond \text{serve}(\text{d}), p_2 \diamond \text{request}(\text{d})$

- $T_2$
  - $p \diamond c \rightarrow c \diamond p$
Example: Execution

- $s_0$
  - $\{\text{serve}(\text{data}) \diamond \text{Server}, \text{request}(\text{data}) \diamond \text{Client1}\}$
- Apply $t_1$ to $s_0$: end in $s_1$
  - $\{\text{Server} \diamond \text{serve}(\text{data}), \text{Client1} \diamond \text{request}(\text{data})\}$
- Apply $t_2$ to $s_1$: end in $s_2$
  - $\{\text{serve}(\text{data}) \diamond \text{Server}, \text{Client1} \diamond \text{request}(\text{data})\}$
- And so on...

Example: Add a client

- Modify Syntax
  - $P ::= \text{Server} | \text{Client1} | \text{Client2}$
- New $s_0$
  - $\{\text{serve}(\text{data}) \diamond \text{Server}, \text{request}(\text{data}) \diamond \text{Client1}, \text{request}(\text{data}) \diamond \text{Client2}\}$
- With new client, we now have an element of non-determinism

Example: Add new rule

- $t_3$
  - $p \diamond c \rightarrow p$
- And add a “final state” $s_N$
  - $\{\text{serve}(\text{data}) \diamond \text{Server}, \text{Client1}, \text{Client2}\}$
- We can now start to ask questions:
  - Can the system reach its final state?
  - Are there any inert states?
  - etc.

Example: C2 Architectural Style

- Evolved from the Chiron User-Interface Development System
- Components and Connectors
  - each potentially with their own thread of control
- Constraint
  - Components can “see” “up” an architecture not “down”
- Benefit: Subsystems are Substitutable
- Research being conducted on C2 today...