Software Architecture

Lecture 5
Last time

- We discussed several case studies on how to architect specific systems
- We also observed architectural qualities for the styles involved
Today

• We capture *architectural styles* in a precise manner
  • *Formalisms*
  • *What is this?*

• We start our study of the *Attribute Driven Design* method (ADD method)
Formal foundation for software architecture?

• What does this mean?

• Why needed?
  • Architectural paradigms are often understood in an idiomatic way
  • And applied in an ad hoc fashion
Formalisms

• Formal models and techniques are cornerstones of a mature engineering discipline
• Engineering disciplines used models and techniques in different ways
  • Provide precise, abstract models
  • Provide analytical techniques based on models
  • Provide design notations
  • Provide basis for simulations …
Architectural formalism?

- *Architecture of a specific system*
  - Allow the architect to plan a specific system
  - Becomes part of the specification of the system
    - Augments the informal characteristics of the SA
    - Permits specific analyses of the system
Architectural formalism? 2

• **Architectural style**
  • Describe architectural abstractions for families of systems
  • Purposes:
    • Make common idioms, patterns and reference architectures precise
    • Show precisely how different architectural representations can be treated as specializations of some common abstraction
Architectural formalism? 3

- *Theory of software architecture*
  - Clarify the meaning of generic architectural concepts
    - Architectural connection, hierarchical architectural representation, architectural style
  - Provide deductive basis for analyzing systems at an architectural level
    - Might provide rules for determining when an architectural description is well formed
    - Compositionality
Architectural formalism? 4

- *Formal semantics of ADL:s*
  - Architectural description is a language issue
  - Apply traditional techniques for representing semantics of languages
Architectural formalism? 5

- *But*:
  - How to formalize?
  - How to compare their relative benefits?
  - Today three (3) examples
    - Use the specification language Z
Formal SA of specific systems

• Many software systems start informally
  • There may be no other ways for this level
  • Modularization facilities of programming languages are often inadequate
    • Require designer to translate architectural abstractions to low-level primitives of programming languages

• E.g. OO architectures appropriate for some architectural decomposition
  • Still too low level of abstraction
Formal SA of specific systems 2

- Use a formal specification language to describe the architecture of a system
  - High level of abstraction

- Purpose:
  - Provide a precise characterization of the system-level functions

- The example: *Oscilloscope* (see lecture on 29.1)
Oscilloscope: SA formalization

• Graph of transformations
  • Pipe-and-filter style
  • Analog signals enter the system
  • Pass through a network of transformations
  • Emerge as pictures and measurements displayed front panel of instrument

• More
  • Each transformer has interface
    • User can tune transformation by configuring parameters
Oscilloscope: what to specify

• Each of the component transformations
  • Filters
• How they are interconnected
• What data is communicated between them
Data streams of oscilloscope

- To formalize the functions of the oscilloscope, begin by characterizing the data
  - Signals $S$, waveforms $W$, traces $T$ as functions over time, volts, and screen coordinates
- In $\mathbb{Z}$
  - $Signal == \text{AbsTime} \rightarrow \text{Volts}$
  - $Waveform == \text{AbsTime} \rightarrow \text{Volts}$
  - $Trace == \text{Horiz} \rightarrow \text{Vert}$
Functions of oscilloscope

• Provide a formal description to each component
  • Explains the configuration parameters
  • What function is computed by the transformation
    • For each configuration

• Treat each component as a higher-order function
  • When applied to its configuration parameters it produces a new function representing the results of the transformation
Oscilloscope function: Couple

- The component *Couple*
  - Used to subtract a DC offset from a signal
  - User has three (3) parameters to choose from
    - DC, AC, GND (Ground)
  - DC leaves the signal unchanged
  - AC subtracts the appropriate DC offset
  - GND produces a signal whose value is 0 volts at all times
Oscilloscope function: Couple 2

- **Coupling** will be the type of the first parameter of the higher-order function **Couple**

  \[ \text{Coupling ::= DC | AC | GND} \]

  determines the resulting function, of type **Signal** --> **Signal**:

  \[ \text{Couple: Coupling --> Signal --> Signal} \]
Oscilloscope function: Acquire

- A *Waveform* $W$ is obtained from a *Signal* $S$ by extracting a time slice
  - Waveform identical to the signal except that defined only over a bounded interval
  - Interval determined by
    - Two time values, *delay* and *duration*
    - A reference time, *trigger event*
  - Duration determines length of interval
  - Delay determines when the interval is sampled relative to trigger event

  $\text{TriggerEvent} == \text{RelTime}$
Oscilloscope: connectors

- Putting things together by interpreting connectors of the architecture as establishing input/output relationships between components
  - Collect the individual components and compose them into a single subsystem
  - Package the parameters of the components
  - Subsystem is a functional composition of the individual transformers
Oscilloscope: putting things together

- Package individual components parameters as single *data structure*

- *ChannelParameters*
  - `c: Coupling`
  - `delay, duration: RelTime`
  - `scaleH: RelTime`
  - `scaleV: Volts`
  - `posnV: vert`
  - `posnH: Horiz`
Oscilloscope: putting things together

• Subsystem

\textit{ChannelConfiguration} : ChannelParameters --> TriggerEvent --> Signal --> Trace

\textit{ChannelConfiguration} = (\lambda \text{trig:TriggerEvent} \cdot \\
\text{Clip} \circ \\
\text{WaveformToTrace}(p.\text{scaleH},p.\text{scaleV},p.\text{posnH},p.\text{posnV}) \circ \\
\text{Acquire}(p.\text{delay},p.\text{duration}) \ \text{trig} \circ \\
\text{Couple} \ p.c)
Oscilloscope results

• What good is this?
  • We have a precise characterization of the system
  • The architecture has been exposed as a configuration of components (parameterized data transformers) connected functionally by inputs and outputs
    • Without translating into some specific programming language
    • Makes precise certain architectural assumptions
      • Components share data only via their connections
      • External parameters need to be evaluated before the components can perform their primary function
Formalizing an Architectural Style

- Problems with the previous specification:
  - The underlying architectural style is not made explicit
    - Must be inferred from the description of a particular system
    - How to elaborate the design?
    - E.g.: absence of cycles: in this example or essential feature?
  - Avoids design issues due to high abstraction level
    - How is data transmitted?
    - Is there a fair scheduling between filters?
  - Architectural connection is implicit via functional composition
    - Cannot reason about topological properties independently
Formalizing *pipe-and-filter* style: *components*

- Components:
  - Filters which transform streams of data
- Each filter has *input ports* for reading data and *output ports* for writing results
- A filter performs its computations incrementally and locally
  - Filters operate concurrently
Formalizing *pipe-and-filter style: connectors*

- Connectors
  - *Pipes* that control the flow of data
- Each pipe links an output port to an input port
  - Indicates how data flows
  - Carries out the transmission
Formalizing *pipe-and-filter style*: computational step

- A **computational step** is either:
  - An *incremental transformation* of data by a **filter** or
  - A *communication of data* between ports by a **pipe**
Formalizing *pipe-and-filter* style

- A tree step approach
  - Define components: filters
  - Define connectors: pipes
  - Show how pipes and filters are combined
- For each aspect characterize its
  - Static and dynamic properties
  - Here via system state
Formalizing *pipe-and-filter* style: 

*data*

- Given

  FILTER, PORT, FSTATE, DATA

  \( \text{Port\_State} == \text{PORT} \rightarrow \text{seq DATA} \)

  \( \text{Partial\_Port\_State} == \text{PORT} \rightarrow \text{seq DATA} \)
Formalizing *pipe-and-filter* style

**Filter**

- **Formal filter**
  - Defined by name, ports and program
  - **Ports** defined as a set of names
    - Directional: *input* ports and *output* ports
  - **Typed ports**
    - The type represents the kind of data the filter is prepared to process on that port
    - The types are subsets of DATA, the alphabet of the port
Formalizing *pipe-and-filter style*

**filter 2**

- Filter’s program in three parts
  - Set of legal program states
  - A starting state
  - A mapping from inputs to output
    - With a possible state change as a side effect
- Gives a state machine view of a filter
- Invariant includes consistency checks:
  - Respecting port types
  - No illegal states
Formalizing *pipe-and-filter style*

State of the filter is composed of:
- the current program state: `internal_state`
- the data in the input ports not yet read
- data written on output ports not yet delivered
Formalizing *pipe-and-filter* style

**filter 4**

- A computational step of a filter
  - Reading from the inputs and writing to the outputs
  - Relation based on inputs, internal state and program
- Filter operates incrementally and locally
  - Output depends on what is *actually* consumed, not on data yet to be consumed
  - Is allowed to depend on historical data, not on anything outside the filter or on previous output
Formalizing *pipe-and-filter style*

**pipe**

- Formal pipe
  - typed connection between two ports, one output of a filter and the other an input to a filter
- State divided into two parts:
  - Data already delivered to the sink
  - Data yet to be delivered
- Pipes are here self-contained entities and
  - One can reason about them independently of filters
Formalizing *pipe-and-filter style*

- A consequence:
  - The same data appears in two places, at the ports of the filters and at the ends of the pipes
  - Not a problem in mathematics
  - Need to be combined when building a system
Formalizing *pipe-and-filter* style

- A computational step
  - Pipe delivers data from its source to its sink
- Several aspects of pipes made formal:
  - Data is not altered during transmission
  - The order of transmitted data is not changed
  - A pipe connects exactly two ports
  - The amount of data is not specified
    - Allows several different implementations and data transmission policies
Formalizing *pipe-and-filter* style *pipe-and-filter* system

- Pipe-and-filter system composed as a collection of filters and a collection of pipes
- Consistency guaranteed by
  - Each filter has a unique name
  - No “dangling” pipes
    - Requirement for defining a system w/o reference to other systems
    - Pipes create a context within which filters operate
      - Pipe defines and is defined by the ports it connects
  - Ports connect to no more than one pipe
    - Distinction between filters and pipes
Formalizing *pipe-and-filter style*

*pipe-and-filter system 2*

- Not every port of a filter must be connected to some pipe
  - Allows us to model systems that are later connected to other systems
- Open pipe-and-filter systems
- Allows hierarchical treatment of pipe-and-filter systems
  - Any pipe-and-filter system is equivalent to a high-level filter
Formalizing *pipe-and-filter* style

*pipe-and-filter* system 3

- The state of a system defined by states of its components
  - We identify the states of ports and pipes
Formalizing *pipe-and-filter* style

*pipe-and-filter* system

- A computational step is either
  - A computation of a filter or
  - A transmission of a pipe
- Non-deterministic execution of a single filter, leaving the rest of the system unchanged
- Non-deterministically chosen transmitting pipe leaving everything else unchanged
Formalizing *pipe-and-filter style* pipe-and-filter system

- System computation is a sequence of steps beginning with a start state and continuing via legal computation steps
  - Every filter is in its start state
  - Every pipe is empty
  - Every output port contains no data
    - Unconnected ports are not required to be empty as treated as system input/output
Formalizing *pipe-and-filter style*

*pipe-and-filter system final*

• Provides a precise, mathematical description of a family of systems
  • Expose essential characteristics, hiding unnecessary details

• Allows us to analyze properties of systems designed in this style
  • E.g. subnets can be encapsulated as new filters

• Specializations of the style possible
  • A pipeline
Formalizing an Architectural Design Space

• **Problem**: different designers may interpret an architectural idiom in different ways
  • Client-server might mean different things to different designers

• **Related problem**: several systems may be designed with similar architectural structure, but designers do not recognize this
  • Missed opportunities to share experience
Formalizing an Architectural Design Space 2

• An architectural formalism can make relationships between architectures precise

• Example:
  • Relate systems built around the implicit invocation architectural style
  • Implicit invocation: components announce events
  • Components register to receive events
  • When an event is announced, all the procedures associated with it will be automatically invoked
Implicit invocation

- What is the vocabulary of events?
- Can events announcements carry data?
- Concurrency in handling events?
- ...
- Different answers lead to architectures with different properties
Implicit invocation 2

• How to formalize the above?
  • Start with a simple architectural abstraction and show how specific systems refine the abstraction
• Assume a basic set of events, methods, and component names
  [EVENT, METHOD, CNAME]
• An architectural component is an entity that has a name and an interface consisting of a set of methods and a set of events
Implicit invocation 3

• An event or a method has a component name and the event or method itself
  Events == CNAME × EVENT
  Methods == CNAME × METHOD

• An event system EventSystem consists of a set of components and an event manager EM

• EM associates events with methods

• The invariant asserts that
  • Components have unique names
  • Event manager relates actual events to actual methods
Implicit invocation 4

• EM is very general
  • Allows one event to be associated with many different methods
  • Even within the same component
  • Some events might be associated with no methods
• Open issues
  • Which components can announce events?
  • Any restrictions on methods that can be associated to events?

• The issues above need to be resolved for a more concrete architectural style
MVC paradigm

• Specializing the model:
  • Model-View-Controller (MVC) paradigm
    • Any object can register as a dependent of any other object
    • An announcement of the *changed* event results in the implicit invocation of the *update* method in each its dependents
    • MVC provides hence a fixed, predetermined set of events (*changed*) and associated methods (*update*)
MVC paradigm 2

- Formally: declare *changed* event and *update* method as elements of EVENT and METHOD, respectively
  
  *changed*: EVENT
  
  *update*: METHOD
MVC paradigm 3

- Model dependencies between objects as a relation between components
  - Via EM relation
    - Events restricted to \{changed\}
    - Pair changed event with update methods
- A consequence: each dependent must have update as one of its methods
Tool integration

• Another application: tool integration
  • Tools communicate by broadcasting events
  • Other tools register patterns indicating which events they are interested in and which methods should be called when an event matches a pattern
  • A pattern matcher checks registered patterns when events are announced
  • Pattern matcher invokes associated methods
Tool integration 2

• Formalize this in terms of the basic model
  • Add a new type of basic entity, PATTERN
    [PATTERN]
  • Associate a pattern matcher (\textit{match}) with \textit{EventSystem}
  • \textit{register} relation associates patterns with methods
    • Determines EM
  • Invariant guarantees that the Event/Method pairs are those with registered patterns
Formalizing an Architectural Design Space final

• So what?
  • We have a common architectural style shared by many systems
  • Exposes the similarities in two different systems by showing how they are instances of the same basic formal architecture
  • We have a template to use when similar comparisons are carried out
  • It is possible to provide a computational model associated with the style
  • Provides a general framework for specifying styles
Theory of Software Architectures

• Goal of research: *to clarify the basic nature of software architecture*

• What is a component?

• What is a connector?

• What is a well-formed architecture?

• What are reasonable rules for architectural decomposition?
  • Component/connector represented by sub architectures
Designing the architecture

- Architecture in the life cycle
- Designing the architecture
- Forming the team structure
  - And its relationship to the architecture
- Creating a skeletal system
Architecture in the life cycle

• Evolutionary Delivery Life Cycle
  • Get user and customer feedback
  • Iterate through several releases before final release
  • Adding of functionality with each iteration
  • Delivery of a limited version
    • When sufficient set of features has been developed
Requirements

- Architecture is shaped by requirements
  - Functional, quality, and business requirements
  - Called architectural drivers
- Identifying drivers
  - Determine highest priority business goals (few!)
  - Turn these business goals into quality scenarios
  - Choose the ones with most impact on architecture: these are the architectural drivers (less than 10)
ADD method

• Attribute-Driven Design
  • Method to design architecture so that both functional and quality requirements are met
  • Input: the set of quality scenarios for drivers
    • Key drivers may change during design, due to
      • Better understanding of requirements
      • Changing requirements
  • Employs knowledge about relation between quality attribute achievement and architecture
ADD

- Approach to defining SA by decomposing based on the quality attributes
- Recursive decomposition process
  - At each stage tactics and architectural patterns are chosen to satisfy some qualities
  - Then functionality is added to instantiate the module types provided by the pattern
ADD output

• First several levels of a module decomposition view of an architecture
  • Possibly other views too
• Not all details of the views result from applying ADD
• System described as
  • a set of containers for functionality
  • the interactions among the containers
ADD output 2

- Necessarily coarse grained
- Critical for achieving desired qualities
- Provides framework for achieving the functionality
- Difference between ADD output and an architecture ready for implementation
  - Detailed design decisions postponed
  - Flexibility
Case study

• **Garage door opener**
  • Responsible for raising and lowering the garage door, via
    • Switch
    • Remote control
    • Home information system
  • It is possible to diagnose problems of opener from the home information system (**HIF**)
• **Product line architecture! (**PLA**)**
Sample input

• ADD assumes **functional requirements** and **constraints** as input

• ADD also needs the **quality requirements**
  • Set of **system-specific quality scenarios**
  • These provide a checklist to be used for the development of **system-specific scenarios**
  • Only the necessary detail
Case study: quality scenarios

• Device and controls for opening and closing the door are different for the different products in the product line
  • May include controls from within the HIF
  • Product architecture for a specific set of controls should be directly derivable from the PLA
• Processor used in different products will differ
  • Product architecture for each specific processor should be directly derivable from the PLA
Case study: quality scenarios 2

- If an obstacle (person/object) is detected by garage door during descent, it must halt or re-open within 0.1 second.
- Garage door opener should be accessible for diagnosis and administration from within the HMI.
  - Using a product-specific diagnosis protocol.
  - Should be possible to directly produce an architecture that reflects this protocol.
ADD Steps

• Choose the module (initially whole system) to decompose
  • Required input available for that module
• Refine the module according to following steps
  • Choose architectural drivers
  • Choose architectural pattern that satisfies the drivers
  • Instantiate modules and allocate functionality
  • Define interfaces of child modules
  • Verify and refine use cases and quality scenarios and make them constraints for child modules
• Repeat steps above for every module that needs further decomposition