

The Value of Software Architecture

GSAW98

John Salasin
DARPA / ITO



What is it?

- **A precise representation of a system's functional decomposition and control, communication and (gross) data structures**
 - Shows component interactions (in time and space) and relationships (e.g., derivation)
- **A specification of global constraints over the composition of the system, and the interaction of its components**
- **A formal synthesis of informal practice by system architects of sketching systems using boxes and arrows**



What is it?

Analogous to “types” in programming languages

- Provide checking and generation
- Simplification through specialization

Data Types

- Abstracts complex data types (strong typing), e.g.
 - $X := \text{list of apples}$
 - $Y := \text{array of oranges}$
- Defines legal operations, e.g.
 - Apples + Apples OK
 - Apples + Oranges ~~OK~~
- Generates code to implement logical operator specialization
 - “+” for matrix, vector, boolean
 - “sort” for integer, real, character

Architecture (styles)

- Abstracts component interactions
 - Pipe and filter
 - Transaction processing
- Defines legal connections / interactions
 - Pipe \Rightarrow Filter
 - Pipe \Rightarrow ~~Transaction~~
- Generates “glue” code to implement component interaction rules /constraints
 - Temporal / control relationships for pipe/filter vs. transaction processing
 - Triggers to control (dynamic) typology of components



Why is it useful?

- Enables automatic analysis and early detection of errors (correctness)
- Enables reuse and product line development
- Supports incrementality
- Supports optimization (non-functional attributes)
- Provides basis for Software Process Improvement (SPI)



Why is it useful?

- Enables automatic analysis and early detection of errors, e.g.:
 - Given sequence of events can (or cannot) occur
 - Deadlock, livelock conditions
 - Sequence of processing steps in distributed applications (e.g., authorization completed before data is accessed)
 - Components can (or cannot) be composed with predictable properties, e.g.:
 - Timing
 - Resource use, starvation
 - Control of dynamic interaction (reconfiguration behavior) to ensure, e.g.:
 - Components exist before invocation
 - Links don't exist to non-existent components



Why is it useful?

- Enables reuse and product line development
 - Provides a transferable, reusable abstraction of a system and unambiguous specification of architectural standards (e.g., for HLA, ATIS, ...)
 - Provides infrastructure for very high level domain-specific notations / languages (and generating code)
 - “Style” provides vocabulary (component and connector “types”) and grammar (rules for legal interactions)
 - “General purpose” Architecture Description Languages (ADLs) allow disciplined design of systems that mix multiple styles
 - Support to defining families of systems -- foundation of software developed as a product line
- Provides assurance that implementation is a valid instantiation of architecture



Why is it useful?

- Supports Incrementality
 - Assurances that properties can be relied upon while the system evolves
 - Assuring properties at architecture, rather than code, level
 - Automated code development / evolution
 - Architecture modification => code modification
 - Automated support to test and analysis
 - Basis for specifying / deriving test and analysis plans (modifications)
 - Dynamic (run-time) modification
 - Specification and control of change mechanisms



Why is it useful?

- Supports optimization of component interaction wrt, e.g.:
 - Performance
 - Fault tolerance
 - Security/safety concerns



Why is it useful?

- Provides basis for Software Process Improvement (SPI)
 - Early, up-front predictive analysis of tradeoffs
 - Medium for communication between engineers and management
 - Supports cognitive design processes by modeling architectures from multiple perspectives
 - Delivers design guidance in a timely and understandable fashion
 - Supports planning (resource estimation) and managing system development.



Some Evaluation Criteria

- Utility Measures
 - Ability to assure that various (types of) constraints are satisfied without testing -- based, e.g., on guarantees built into the representation or software development process
 - Ability to ensure plug-and-play replacement of components with minimal impact
 - Cost savings from analysis and optimization at the architectural level, relative to prototype and test approaches (i.e., savings from not having to implement as much code that is thrown away).
 - Ability to use same (family of) representation(s):
 - to analyze/optimize with respect to performance, fault tolerance,...
 - to perform tradeoff analyses involving various qualities (e.g., performance, reliability, etc.)



Some Evaluation Criteria (cont)

- Utility Measures (cont)
 - Support for automatic system generation, reuse and product line development
 - Understandability -- architecture descriptions serve as useful documentation (as perceived by users of the standard)
 - Support to Incrementality
 - Correlation (linear relationship) between size of change and effort required [human or machine]
 - Percent of reduction in effort for testing code due to:
 - Increased number of analyses conducted at architecture level
 - Automated generation of test plans/data
 - Avoidance of retest for unchanged portions of system
 - Reduction in code (and programming effort) to specify and implement dynamic architecture



Some Evaluation Criteria

- Adaptability / Usability Measures
 - Kinds of properties detected (vs. standard design notations/tools)
 - Extent to which “style” vocabulary and grammar supports domain specific language definition
 - Extensibility -- ease of adding domain-specific constructs to existing styles
 - Scalability -- ability to represent and analyze large scale standards/frameworks like HLA