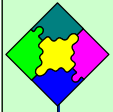


5. Describing Software Architectures: Approach, Issues, Concepts, Languages



Outline

- ⌘ What is software architecture?
- ⌘ What are essential architectural concepts?
- ⌘ Overview of Architecture Description Languages (ADLs)

Examples of Architecture Descriptions

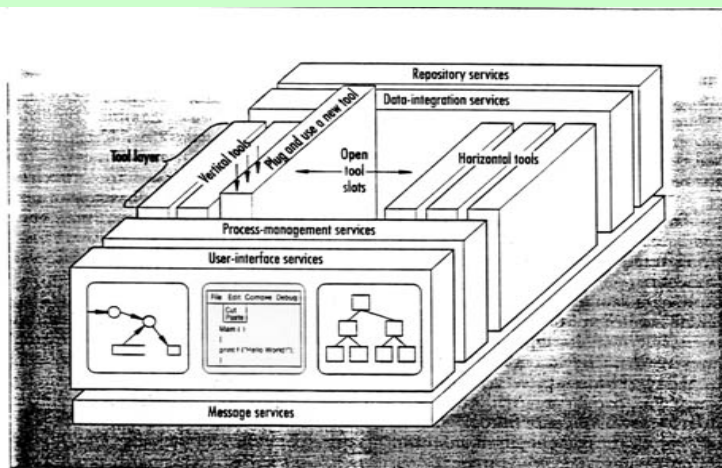


Figure 1. The NIST/ECMA reference model.

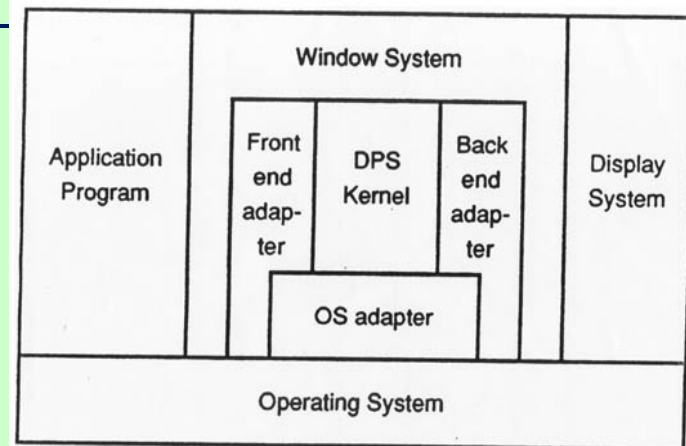
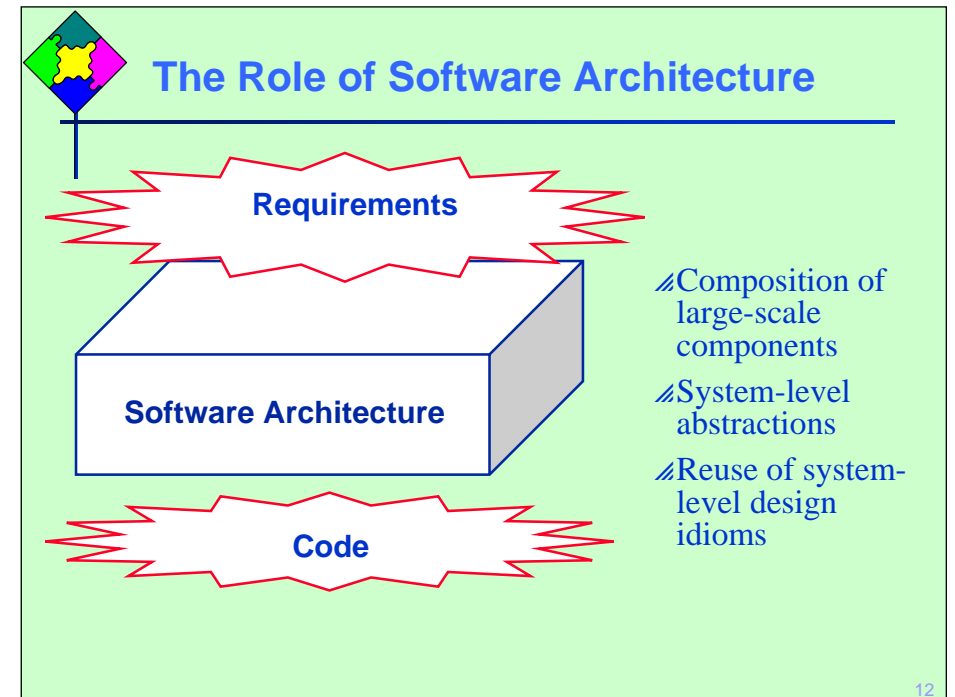
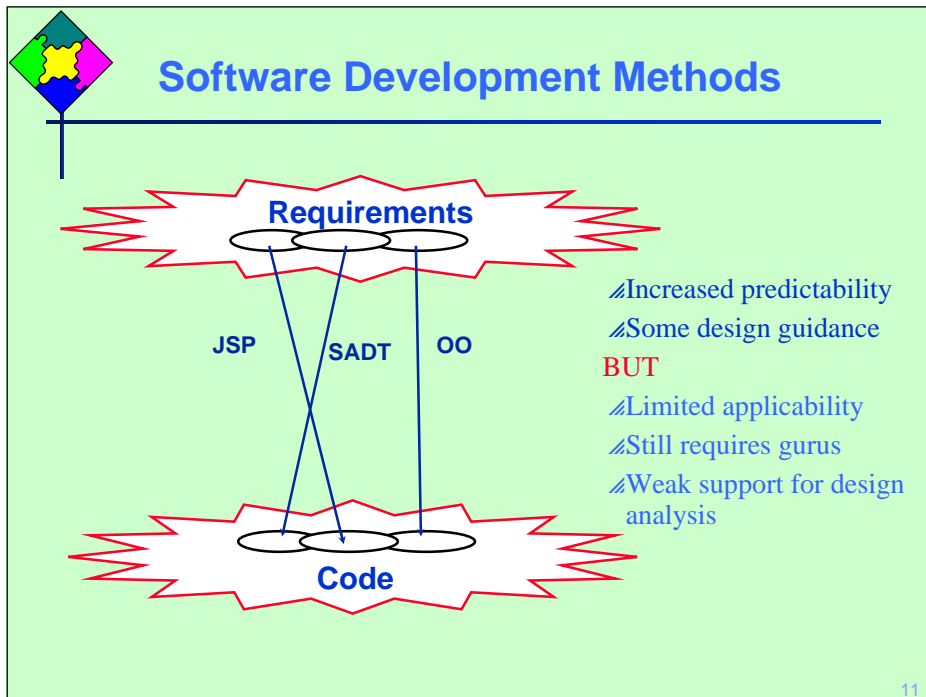
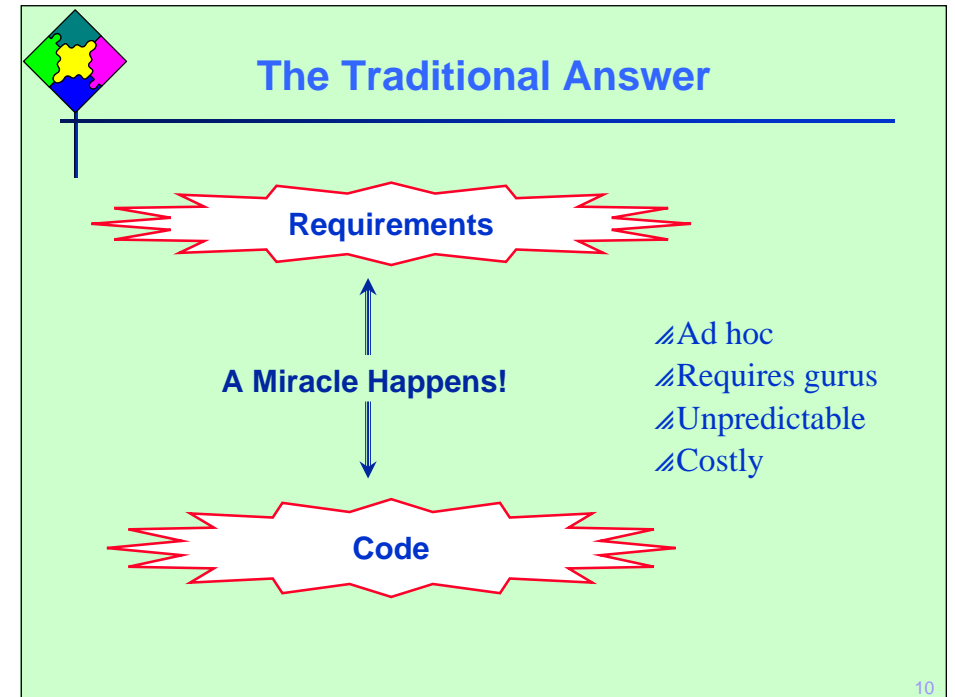
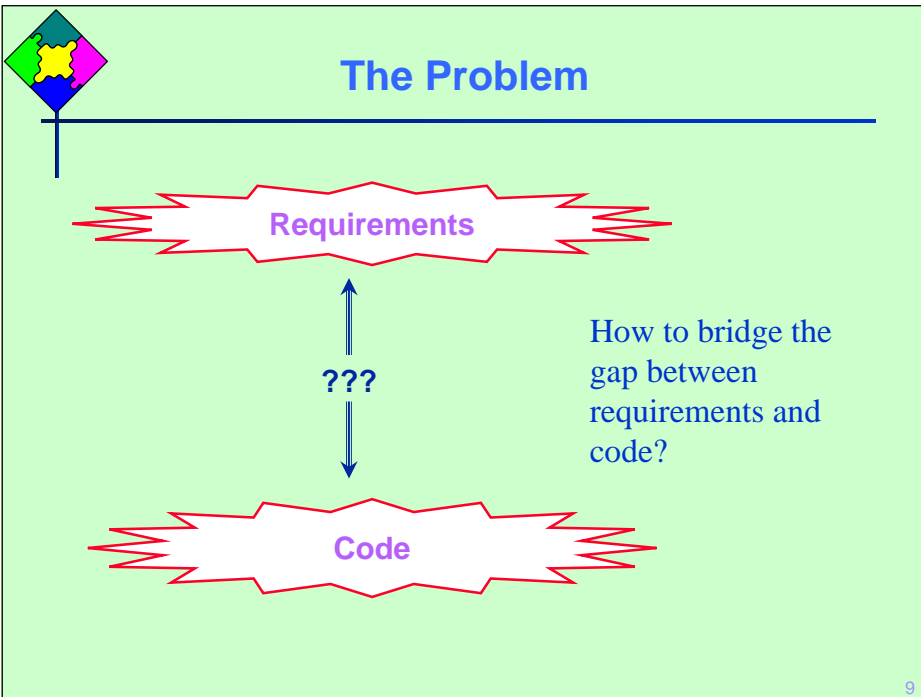
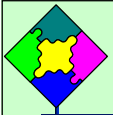


Figure 2. Display PostScript interpreter components.

An Overview of the DISPLAY POSTSCRIPT™ System, Adobe Systems Incorporated, March 16, 1988, P. 10





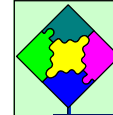
Definitions of Software Architecture

- ⌘ Many definitions
- ⌘ Here is a typical one

A software architecture for a system is the structure or structures of the system, which comprise elements, their externally-visible behavior, and the relationships among them.

- ⌘ But what structures? What is “behavior”? What kinds of relationships?

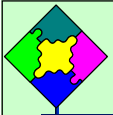
13



Issues Addressed by an Architectural Design

- ⌘ **Gross decomposition of a system into interacting components**
 - ❖ typically **hierarchical**
 - ❖ using **rich abstractions** for component interaction (or system “glue”)
 - ❖ often using common design **idioms/styles**
- ⌘ **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

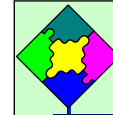
14



Many Views of Architecture

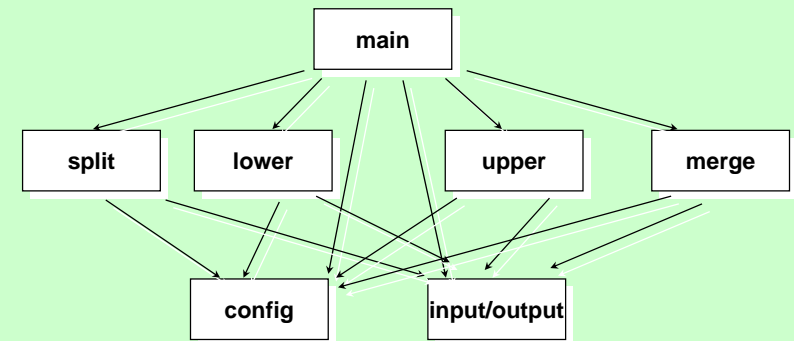
- ⌘ **There are many possible views of software systems**
 - ❖ **Module structures:** code/implementation structures
 - ☞ e.g., class diagrams, work breakdown structures, “def-use” graphs
 - ❖ **Deployment structures:** physical/resources structures
 - ☞ e.g., processors, networks
 - ❖ **Run-time structures:** system structure and behavior at run time
 - ☞ e.g., clients, servers, databases, instances of objects
- ⌘ **Each has its purpose in understanding the overall nature of a system**
 - ❖ Modules good for reasoning about maintenance
 - ❖ Deployment good for reasoning about resources
 - ❖ Run-time good for reasoning about system behavior
- ⌘ **In this lecture I will focus on run-time structures**

15



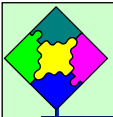
Example: Alternating Characters (Module View)

Produce alternating case of characters in a stream

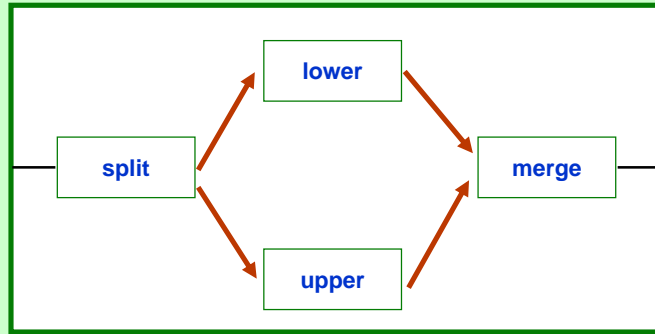


Definition/Use Modularization

16

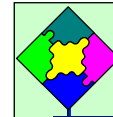


Example: Architectural Description (Component-and-Connector View)



Components and Connectors

17



Why Formalize Software Architecture?

Understanding

- ❖ Precise definition of system
- ❖ Clarification of requirements

Analysis

- ❖ System-level analysis of critical properties

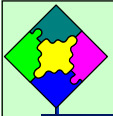
Construction

- ❖ Blueprint for construction; basis for code generation
- ❖ Identify opportunities for reuse

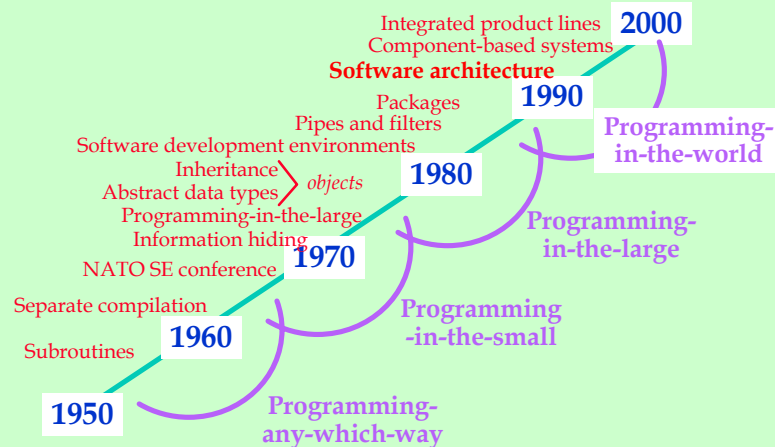
Evolution

- ❖ Define allowable envelope of change, ensure that system invariants are not violated

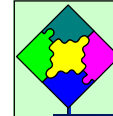
18



Software Architecture in Context



19



Evolution of the Field

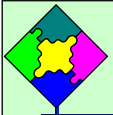
Today

- ❖ Recognition of the value of architects in software development organizations
- ❖ Processes that require architectural design reviews and explicit architectural documentation
- ❖ Emerging use of product line architectures, commercial architectural standards, component integration frameworks
- ❖ Codification of vocabulary, notations & tools for architectural design
- ❖ Books/courses on software architecture

Unfortunately

- ❖ Little use of formal models, tools, analyses
- ❖ Many gaps in our understanding about how to model and analyze key aspects of software architecture

20

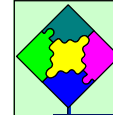


Elements of Architectural Descriptions: **Systems**

///The architecture of an individual **system** identifies its principle

- ❖ **Components:** define the locus of computation
 - ☞ Examples: filters, databases, objects, ADTs
- ❖ **Connectors:** mediate interactions of components
 - ☞ Examples: procedure call, pipes, event broadcast
- ❖ **Properties:** specify information for construction & analysis
 - ☞ Examples: signatures, pre/post conditions, RT specs, protocols, performance attributes

21



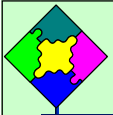
Elements of Architectural Descriptions: **Styles**

///An **architectural style** defines a *family* of architecture instances including

- ❖ **Component/connector types**
 - ☞ the vocabulary of architectural building blocks
- ❖ **Constraints** on how the building blocks can be used, including
 - ☞ **topological** rules
 - ☞ **interface** standards
 - ☞ **required** properties

Note: relationship between architectural styles and system instances is similar to that between types and instances

22



Styles: Questions to Address

///System Model

- ❖ What is the overall organizational pattern?

///Structure

- ❖ What are the basic components and connectors?
- ❖ What topologies are allowed?

///Computation

- ❖ What is the underlying computational model?
- ❖ How is control and data transferred between components?

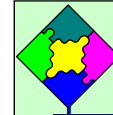
///Properties

- ❖ Why is this style useful?
- ❖ What kinds of properties are exposed?

///Analyses

- ❖ What kinds of analysis does the style support?

23



Taxonomy of Architectural Styles

Data Flow

- Batch sequential
- Dataflow network (pipes&filters)
 - acyclic, fanout, pipeline, Unix
- Closed loop control

Call-and-return

- Main program/subroutines
- Information hiding
 - ADT, object, naive client/server

Interacting processes

- Communicating processes
 - LW processes, distrib objects,
- Event systems
 - implicit invocation, pure events

Data-oriented repository

- Transactional databases
 - True client/server
- Blackboard
- Modern compiler

Data-sharing

- Compound documents
- Hypertext
- Fortran COMMON
- LW processes

Hierarchical

- Layered
- Interpreter

24

Data Flow: Pipes and Filters

The diagram illustrates a data flow pipeline. It starts with an input 'Signal' entering a box labeled 'Couple'. A purple arrow labeled 'Computation filter' points to this box. An arrow labeled 'Dataflow pipe' points to the connection between 'Couple' and the next box 'Acquire'. From 'Acquire', an arrow labeled 'Waveform' branches to a box labeled 'Measure', which outputs 'Measurement'. The main path continues from 'Acquire' to 'To-XY', which outputs 'Trace', and finally to 'Clip', which has an outgoing arrow.

25

Pipes and Filters

- /// Filter
 - ❖ Incrementally transform some amount of the data at inputs to data at outputs
 - ⊖ Stream-to-stream transformations
 - ❖ Use little local context in processing stream
 - ❖ Preserve no state between instantiations
- /// Pipe
 - ❖ Move data from a filter output to a filter input
 - ❖ Pipes form data transmission graphs
- /// Computational Model
 - ❖ Run pipes and filters (non-deterministically) until no more computations are possible.
- /// Analysis
 - ❖ Functional composition; additive latencies; etc.

26

Data Oriented Repository: Blackboard

The diagram shows a central box labeled 'Blackboard (shared data)'. Eight surrounding boxes, labeled 'ks1' through 'ks8', are connected to the blackboard by double-headed red arrows. A purple arrow labeled 'Computation' points to 'ks2'. A purple arrow labeled 'Memory' points to 'ks5'. Two teal arrows labeled 'Direct access' point to 'ks1' and 'ks8'.

27

The Blackboard Model

- /// Knowledge Sources
 - ❖ World and domain knowledge partitioned into separate, independent computations
 - ❖ Respond to changes in blackboard
- /// Blackboard Data Structure
 - ❖ Entire state of problem solution
 - ❖ Hierarchical, non-homogeneous
 - ❖ Only means by which knowledge sources interact to yield solution
- /// Computational Model
 - ❖ Changes to data in blackboard by one knowledge source trigger the actions of other knowledge sources to create new blackboard data

28

Call-Return: Object-Oriented

The diagram illustrates a network of objects (obj) connected by operations (op). A purple arrow labeled 'Manager ADT' points to a specific object. A green arrow labeled 'Proc call' points to an operation between two objects. Text at the bottom left states: 'obj is a manager' and 'op is an invocation'.

29

Call-Return Systems

- // Objects
 - ❖ Encapsulate representations
 - ❖ Provide interfaces to access services
- // Connectors
 - ❖ Call-return; service invocation
- // Computational Model
 - ❖ Services requested from known service provider; Requester blocks
- // Analysis
 - ❖ Correctness of a component depends on correctness of services it invokes

30

Loosely Coupled Components: Publish-Subscribe

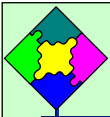
The diagram shows several oval-shaped components. Some have red exclamation marks (!) and question marks (?) pointing to them. A purple arrow labeled 'Object or Process' points from a component to another. A green arrow labeled 'Publish-Subscribe' points from a component to another.

31

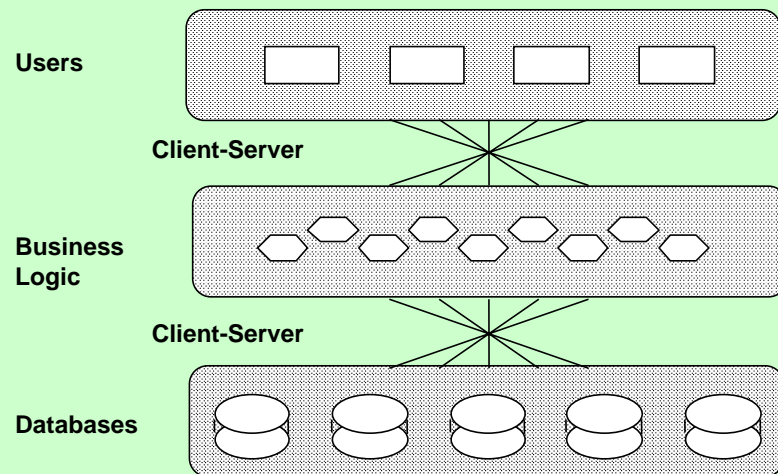
Publish-Subscribe (Implicit Invocation)

- // Components
 - ❖ Objects, processes
 - ❖ Have a set of methods
 - ❖ Announce (publish) events – via multicast
 - ❖ Subscribe to events by associating a procedure to call
- // Connectors
 - ❖ Event space (bus)
 - ❖ Typically implemented as a dispatcher
- // Computational model
 - ❖ When an event is announced invoke associated procedures (in any order)
 - ❖ Correctness of a component should not depend on correctness of components that subscribe to its announced events.

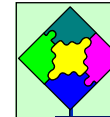
32



Tiers: 3-Tiered Client Server



33



Many Others

Often styles are used in combination

- ❖ Example: each layer might be different style internally
- ❖ Example: a component might have substructure defined in a different style than its surrounding

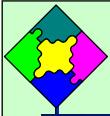
Many styles are closely tied to specific domains -- often by specializing a more generic style

- ❖ sometimes called **component integration frameworks**
- ❖ N-tiered MIS Systems
- ❖ OSI Protocol Stack
- ❖ Instrumentation Systems

In many cases these are specialized to a particular product family,

- ❖ sometimes called a **product line architecture**

34

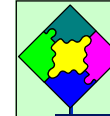


The Challenge for Architectural Description

Software Architecture frees us from the limitations of programming language abstractions

- ❖ new kinds of **components**
 - ☞ not just abstract data types, modules, and objects
- ❖ new kinds of **glue** (connectors)
 - ☞ not just procedure call, data sharing
- ❖ new patterns, idioms, **styles** for system structure
 - ☞ not just for algorithms and data structure
- ❖ many **properties** of interest
 - ☞ not just functional behavior

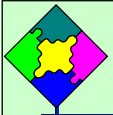
35



How can we establish intellectual control over this new world? Ideally we would like to:

- (1) express arch descriptions precisely and intuitively
 - ☞ both syntax and semantics
- (2) develop soundness criteria & tools to check them
 - ☞ what type checkers and linkers do for current systems
- (3) analyze architectures to determine key properties
 - ☞ such as performance, reliability, change impact, interoperability
- (4) exploit patterns and styles
 - ☞ without unnecessary proliferation and isolation of new design languages and tools

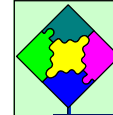
36



Some ADLs (Architecture Description Languages) at a Glance

- /// Acme: architectural tool integration
- /// Aesop: style-specific environments
- /// Armani: designing with constraints
- /// Darwin: distributed systems exo-structure
- /// Meta-H: real-time, fault-tolerant avionics
- /// Rapide: event patterns, architecture simulation
- /// SADL: refinement patterns
- /// UniCon: architectural compilation
- /// Wright: protocol analysis
- /// π -Space: dynamic systems
- /// ArchWare ADL: evolving, dynamic and mobile systems

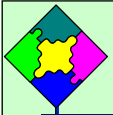
37



Dimensions of Variability

- /// Most Architecture Description Languages (ADLs) agree on use of structure
 - ❖ Essentially as outlined earlier
- /// But they differ considerably about
 - ❖ Kinds of properties that can be characterized and analyzed
 - ❖ Whether there is a fixed set of connector types
 - ❖ Emphasis on static versus dynamic analysis
 - ❖ Support for dynamism (runtime system evolution)
 - ❖ Executability
 - ❖ Openness (ability to incorporate external tools)
 - ❖ Domain specificity

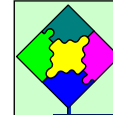
38



Meta-H (Honeywell, USA)

- /// Key Idea: Domain-specific ADL for real-time, fault-tolerant software
- /// Main features
 - ❖ Analysis capabilities
 - ☞ Real-time schedulability analysis
 - ☞ Reliability
 - ☞ Safety/security
 - ❖ Accepts external generators, libraries
 - ❖ Automated system builder
 - ☞ provides runtime communication/synch support
 - ❖ Integrates hardware and software components
 - ❖ Applied with success in avionics domain

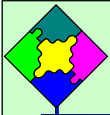
39



Rapide (Stanford Univ., USA)

- /// Key Idea: Define system behavior using event patterns that permit simulation and post facto analysis
- /// Main features
 - ❖ Behavioral modeling allows one to determine erroneous or missing causal information
 - ❖ Trace viewing capabilities
 - ❖ Architectural animation
 - ❖ Supports dynamic architectures
 - ❖ Applications include X/Open architecture, TRW applications

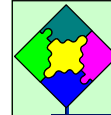
40



Darwin (Imperial College, UK)

- ⌘ Key idea: specify structure of distributed systems
- ⌘ Declarative binding language for defining dynamically-evolving, hierarchical compositions.
- ⌘ Component interfaces defined in terms of provided/required services
- ⌘ Fixed set of connector types
- ⌘ Semantics given in terms of π -Calculus
- ⌘ Behavior of components specified outside system

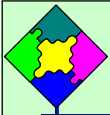
41



Acme (Carnegie Mellon Univ., USA)

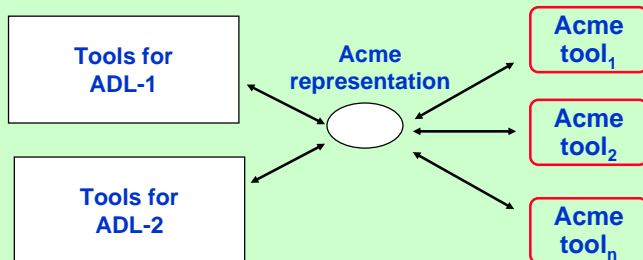
- ⌘ Key Idea: support ADL integration
- ⌘ Integrate diverse ADLs and toolsets
 - ❖ Leverage/integrate capabilities of different ADLs
 - ❖ Minimize ADL translation effort
- ⌘ Platform for ADL-independent tool development
 - ❖ Tool writers can build one tool for many ADLs
 - ❖ Tool developers don't need to invent new ADLs to do architectural analyses
 - ❖ Rich library for constructing ACME tools reduces tool development cost for ACME adopters
- ⌘ Extensible language base for new ADLs
 - ❖ Explore ADL ideas by extending ACME instead of building a new ADL from scratch

42

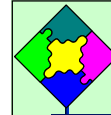


Acme

- ⌘ Tools written to manipulate Acme descriptions
- ⌘ External tools can treat Acme as another ADL toolset



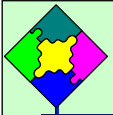
43



Armani (Carnegie Mellon Univ., USA)

- ⌘ Extends Acme with composable and reusable design constraints
 - ❖ Captures both invariant and heuristic constraints
 - ⌘ Constraints expressed in logic-based predicate language.
 - ⌘ Both types and system instances can specify rich constraint predicates
- ⌘ Emerging tool support for constraint checking
 - ❖ Text-based language processing infrastructure
 - ❖ GUI-based environment

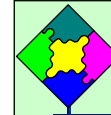
44



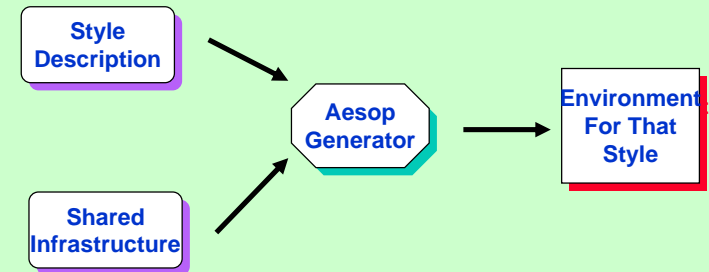
Aesop (Carnegie Mellon Univ., USA)

- ⚡ Key idea: support rapid development of custom, style-specific software architecture design environments.
- ⚡ Configurable architecture design environment can be specialized by loading style descriptions
 - ❖ Minimizes environment development cost
 - ❖ Exploits commonalities of families of systems
 - ❖ Supports domain-specific arch analysis and code generation
- ⚡ Permits integration with external analysis tools
 - ❖ Reuse legacy applications
 - ❖ Integration with existing CASE tools
 - ❖ Sample styles (and analysis):
 - ☞ Pipe and Filter (code generation)
 - ☞ Real-Time-Message Passing (schedulability)
 - ☞ Generic (completeness and consistency checks)

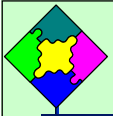
45



Generating Style-Specific Environments



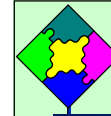
46



WRIGHT (Carnegie Mellon Univ., USA)

- ⚡ Support analysis of arch connection and style
- ⚡ Explicit connector types
 - ❖ Support reuse of:
 - ☞ Patterns of interaction
 - ☞ Components in multiple contexts of interaction
 - ❖ Capture high-level composition abstractions
- ⚡ Statically checkable behavior notation
 - ❖ Capture dynamic communication ordering, non-determinism, and locus of control
 - ❖ Architectural consistency and completeness checks (using model checking tools)
- ⚡ Explicit description of style as constraints
 - ❖ Leverage analysis over families of systems
 - ❖ Proofs exploit architectural structure
 - ❖ Check conformance of configuration to style

47



UniCon (Carnegie Mellon Univ., USA)

- ⚡ Open system supports incremental definition and compilation of heterogeneous architectures
- ⚡ Heterogeneous abstractions
 - ❖ Supports many types of connectors
 - ❖ Supports many types of components
 - ❖ Distinct modes of interaction, packaging
- ⚡ Open System
 - ❖ Accepts externally developed components, connector types and analysis tools
- ⚡ System construction emphasis
 - ❖ Separates structure from implementation
 - ❖ Compiles to std. connector implementations
 - ❖ Incremental development
 - ☞ Partial/incremental specifications
 - ☞ open-ended semantics via property lists

48



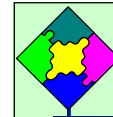
UniCon Component and Connector Types

Abstractions for components

Computation	pure function
SharedData	Fortran common +
SeqFile	unix file
Filter	unix filter
Process	unix process
SchedProcess	real-time process
Module	conventional compilation unit

Abstractions for connectors

Pipe	unix pipe
FileIO	unix ops between process & file
ProcedureCall	architectural use of proc
DataAccess	shared data within process
RemoteProcCall	RPC
RTScheduler	processes compete for time
PLBundler	set of procedure calls and data



π -Space (Univ. of Savoie, France)

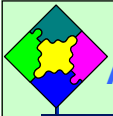
Key Idea: support description of dynamic architectures

Main features:

- ❖ Able to represent static and dynamic software architectures
- ❖ Based on a sound foundation: the π -Calculus
- ❖ Support for definition of architectural styles ($\sigma\pi$ -Space)
- ❖ Description of architectures from styles

Toolset:

- ❖ Visual description of software architectures using an architecture profile for UML
- ❖ Validation by simulation of architectures in PICT
- ❖ Automatic code generation from concrete architectures in ProcessWeb PML



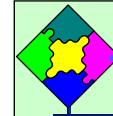
ArchWare π -ADL (Univ. of Savoie, France)

Key Idea: support evolving software architectures (active architectures)

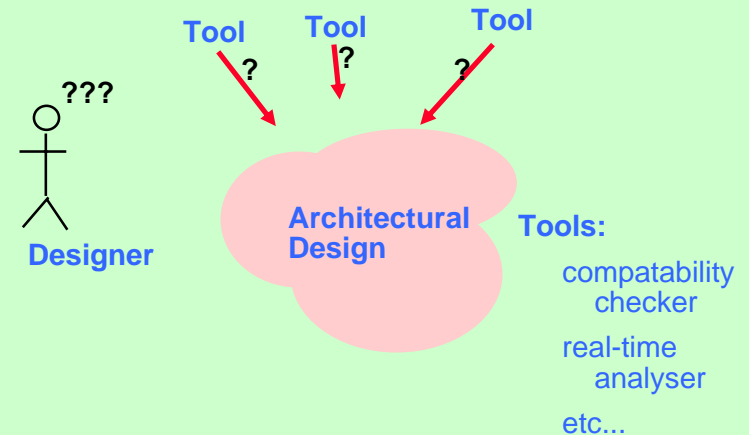
- ❖ Formal architecture description language
- ❖ Support a formal architecture refinement language
- ❖ Able to represent static, dynamic and mobile software architectures
- ❖ Based on a sound foundation: the Higher-Order π -Calculus
- ❖ Customizable architectural elements: components, connectors, configurations, ...
- ❖ Support for definition of architectural styles
- ❖ Description and generation of architectures from styles

Toolset:

- ❖ Visual description of software architectures using an architecture profile for UML
- ❖ Validation by animation/simulation of architectures
- ❖ Verification by theorem prover and model checker
- ❖ Automatic code generation from concrete architectures in different languages using a synthesizer (PBase, Java, ...)

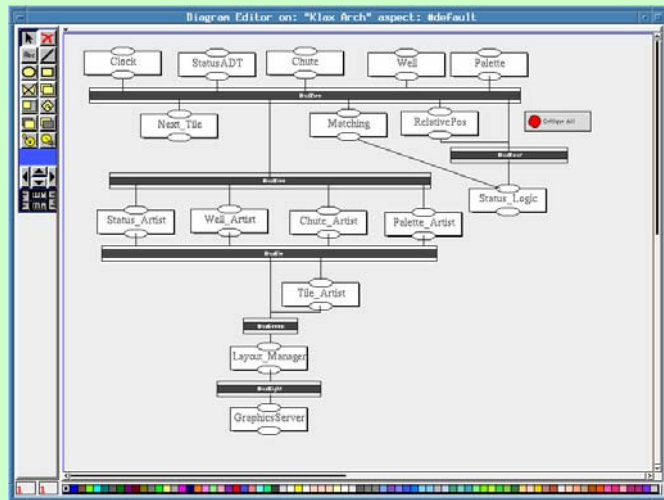


Tool Support for Architectural Design

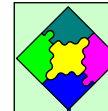




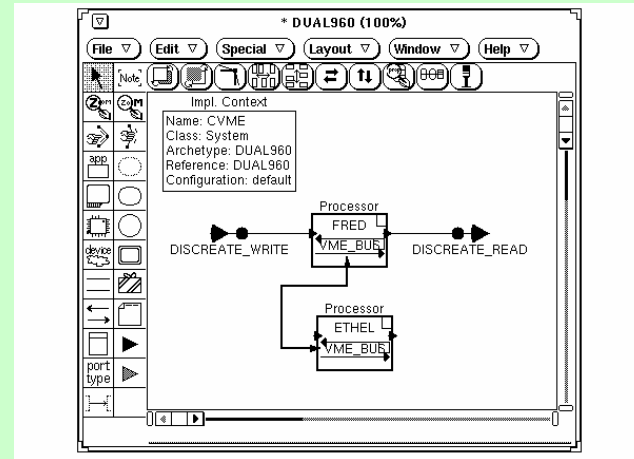
Example Environment: C2



53



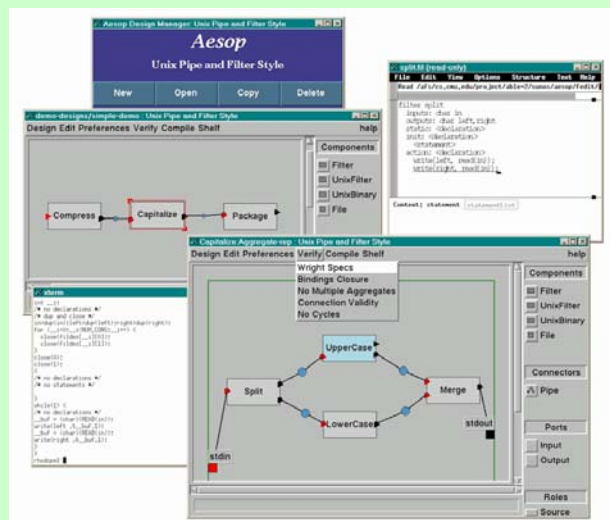
Example Environment: Meta-H



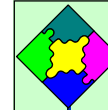
54



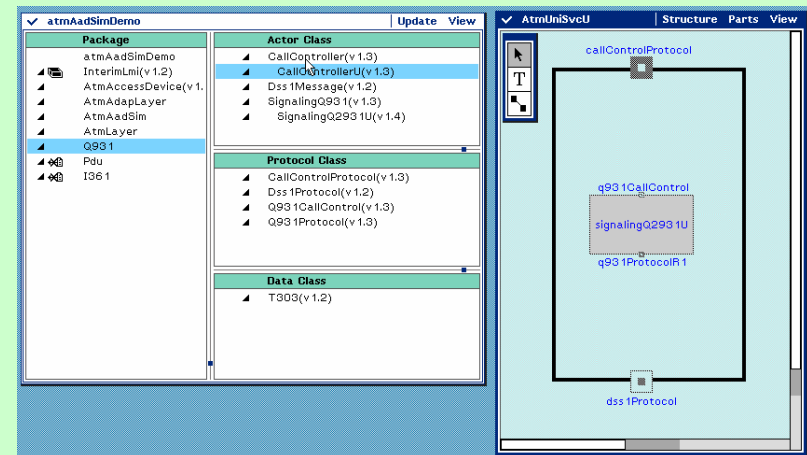
Example Environment: Aesop/PF



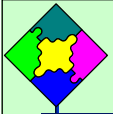
55



Example Environment: ObjecTime



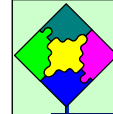
56



Analyzing Architectural Behavior

- ⌘ Much of the architectural research effort has been devoted to modeling and analysis of architectural behavior
 - ❖ Defines abstract events at the architectural level
 - ❖ Usually identifies protocols of interaction for connectors
- ⌘ Based on various modeling formalisms
 - ❖ Process algebras
 - ❖ Chemical Abstract Machine
 - ❖ Statecharts
 - ❖ Pre/post conditions
 - ❖ Category theory (ComUnity)
 - ❖ ... and many others

61



Features of Modern ADLs

- ⌘ System structure is defined separately from individual components
 - ❖ parts are “context independent”
 - ❖ supports hierarchical design
- ⌘ New kinds of connectors can often be defined
 - ❖ need not be realizable directly by a single primitive of an implementation language
 - ❖ have rich semantics
- ⌘ Can express/analyze extra-functional properties
 - ❖ performance, reliability, etc.
- ⌘ Support for architectural styles
 - ❖ reusable architectural patterns

62