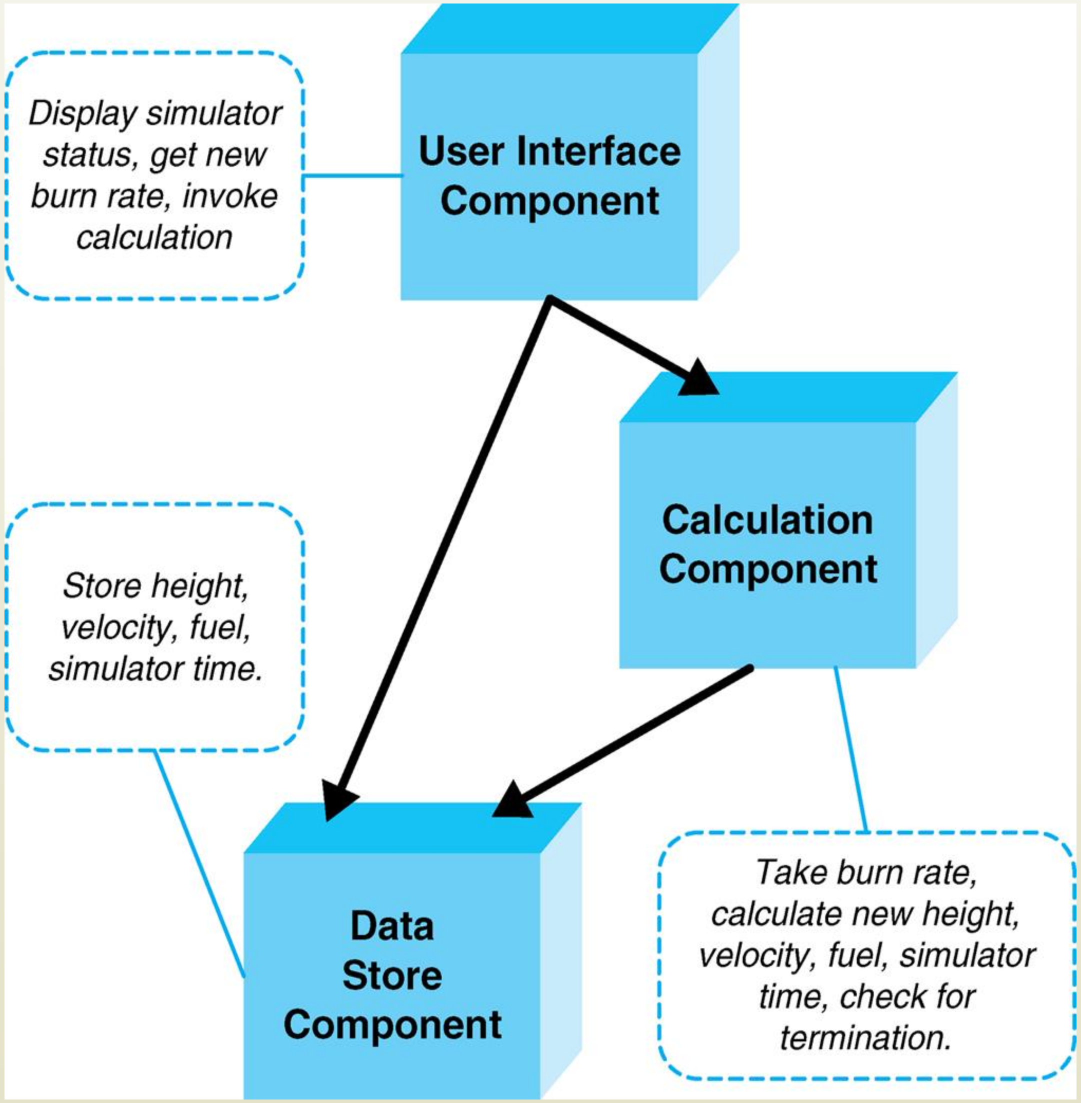
Analysis (of Software Architectures)

Software Architecture
Chapter 8

What Is Architectural Analysis?

- Architectural analysis is the activity of discovering important system properties using the system's architectural models
  - Early, useful answers about relevant architectural aspects
  - Available prior to system's construction
- Important to know
  1. Which questions to ask
  2. Why to ask them
  3. How to ask them
  4. How to ensure that they can be answered

Informal Architectural Models


Display simulator status, get new burn rate, invoke calculation
User Interface Component
Calculation Component
Store height, velocity, fuel, simulator time.
Data Store Component
Take burn rate, calculate new height, velocity, fuel, simulator time, check for termination.

- Helps architects get clarification from system customers (and vice versa)
- Helps managers ensure project scope is appropriate
- Not as useful to developers (no information e.g., about component interaction)

Formal Architectural Models and Analysis

```
Component UserInterface
  Port getValues
  Port calculate
  Computation
Connector Call
  Role Caller =
  Role Callee =
  Glue =
Configuration LunarLander
  Instances
    DS : DataStore
    C : Calculation
    UI : UserInterface
    CtoUIgetValues, CtoUIstoreValues, UItoC, UItoDS : Call
  Attachments
    C.getValues as CtoUIgetValues.Caller
    DS.getValues as CtoUIgetValues.Callee
    C.storeValues as CtoUIstoreValues.Caller
    DS.storeValues as CtoUIstoreValues.Callee
    UI.calculate as UItoC.Caller
    C.calulate as UItoC.Callee
    UI.getValues as UItoDS.Caller
    DS.getValues as UItoDS.Callee
End LunarLander.
```

- Helps architects determine component composability
- Helps developers with implementation-level decisions
- Helps with locating and selecting appropriate OTS components
- Helps with automated code generation
- Not as useful for discussions with non-technical stakeholders

Concerns Relevant to Architectural Analysis

- Goals of analysis
- Scope of analysis
- Primary architectural concern being analyzed
- Level of formality of associated architectural models
- Type of analysis
- Level of automation
- System stakeholders interested in analysis
- Applicable analysis techniques

Architectural Analysis Goals

- The four "C"s
  - Completeness
  - Consistency
  - Compatibility
  - Correctness

Architectural Analysis Goals – Completeness

- Completeness is both an external and an internal goal
- It is *external* with respect to system requirements
  - Does it adequately capture all of a system’s key functional and non-functional requirements?
  - Challenged by the complexity of large systems’ requirements and architectures
  - Challenged by the many notations used to capture complex requirements as well as architectures
- It is *internal* with respect to the architectural intent and modeling notation
  - Have all elements been fully modeled in the notation?
  - Have all design decisions been properly captured?
- In principle, internal completeness is easier to assess than external completeness, and is amenable to automation

A Partial, Formal Model of LL in Rapide

```
begin
         (start or UpdateStatusDisplay) where \
             ($TurnsRemaining > 0) => \
             if ( $TurnsRemaining > 0 ) then \
                 TurnsRemaining := $TurnsRemaining - 1; \
                 DoSetBurnRate(); \
             end if;;
         NotifyNewValues => UpdateStatusDisplay();;
         UpdateStatusDisplay where $TurnsRemaining == 0 \
             => Done();;
end UserInterface;

architecture lander() is
  P1, P2 : Player;
  C : Calculation;
  D : DataStore;
connect
  P1.DoSetBurnRate to C.SetBurnRate;
  P2.DoSetBurnRate to C.SetBurnRate;
  C.DoSetValues to D.SetValues;
  D.NotifyNewValues to P1.NotifyNewValues();
  D.NotifyNewValues to P2.NotifyNewValues();
end LunarLander;
```

- The component instances in the `architecture` portion of the model must be attached to one another (see the `connect` statement)
- A component's `out action` must be connected to another component's `in action`

Architectural Analysis Goals – Consistency

- Consistency is an internal property of an architectural model
- Ensures that different model elements do not contradict one another
- Dimensions of architectural consistency
  - Name
  - Interface
  - Behavior
  - Interaction
  - Refinement

Name Consistency

- Component and connector names
- Component service names
- May be non-trivial to establish at the architectural level
  - Multiple system elements/services with identical names
  - Loose coupling via publish-subscribe or asynchronous event broadcast
    - Contrast this with attempting, say in Java, to access a non-existent class or method which will result in a compile-time error
  - Dynamically adaptable architectures
    - A component or service referred to in the architecture initially may be unavailable but will be added later

Interface Consistency

- Encompasses name consistency
- Also involves parameter lists in component services
- A rich spectrum of choices at the architectural level
- Example: the interface of a required service in a simple QueueClient component, specified in some ADL may look like this:

```
ReqInt:   getSubQ(Natural first, Natural last, Boolean remove)
          returns FIFOQueue;
```

- The above interface is intended to access a service that returns the subset of a `FIFOQueue` between the specified `first` and `last` indices
- Depending on the value of `remove`, the original queue may remain intact or the specified subqueue may be extracted from it

Interface Consistency (cont'd)

- The QueueServer component providing this service may export two `getSubQ` interfaces as follows:

```
ProvInt1: getSubQ(Index first, Index last)
          returns FIFOQueue;

ProvInt2: getSubQ(Natural first, Natural last, Boolean remove)
          returns Queue;
```

- The three interfaces have no name inconsistency but the interfaces' parameter lists and return types are not identical
  - The types of the `first` and `last` parameters in `ReqInt` and `ProvInt1` are different
  - `ReqInt` introduces a Boolean `remove` parameter, which does not exist in `ProvInt1`
  - The return types of `ProvInt2` and `ReqInt` are different

Interface Consistency (cont'd)

- Whether these differences result in actual interface inconsistencies will depend on several factors
- If QueueClient and QueueServer were objects in, say, Java, and their interfaces denoted method invocations, the system might not even compile
- But modeling in software architectures is more flexible
  - If `Natural` is defined to be a subtype of `Index`, no type mismatch will occur between `ReqInt` and `ProvInt1`
  - If the connector between QueueClient and QueueServer is a direct procedure call, then the additional parameter in `ReqInt` will create an inconsistency with `ProvInt1`; however, if the connector is an event one, `remove` will be ignored
  - If `Queue` is not declared to be a subset of `FIFOQueue`, `ProvInt2` and `ReqInt` cannot interact with each other

Behavioral Consistency

- Names and interfaces of interacting components requesting or providing services may match, but behaviors need not
- Example: subtraction

```
subtract(Integer x, Integer y) returns Integer;
```

  - Can we be sure what the *subtract* operation does?
  - We assume it subtracts two integer numbers but what if it provides a calendar subtraction operation?
  - In this case, the result of `subtract(427,27)` will not be 400 but 331 (subtraction of 27 days from April 27 gives March 31)

Behavioral Consistency (cont'd)

- Another example is related to when required and provided interfaces have 'preconditions' which must hold true before the functionality exported via the interface is accessed and 'postconditions' which must hold true after the functionality is exercised
  - Assume that QueueClient requires a `front` operation, whose purpose is to return the first element of the queue (without deleting it from the queue)
  - Assume also that QueueServer provides this operation, and that the two corresponding interfaces match; QueueClient's required service behavior is:

```
precondition  q.size ≥ 0;
postcondition ~q.size = q.size;
```

  - ~ denotes the value of `q` after the operation has been executed

Behavioral Consistency (cont’d)

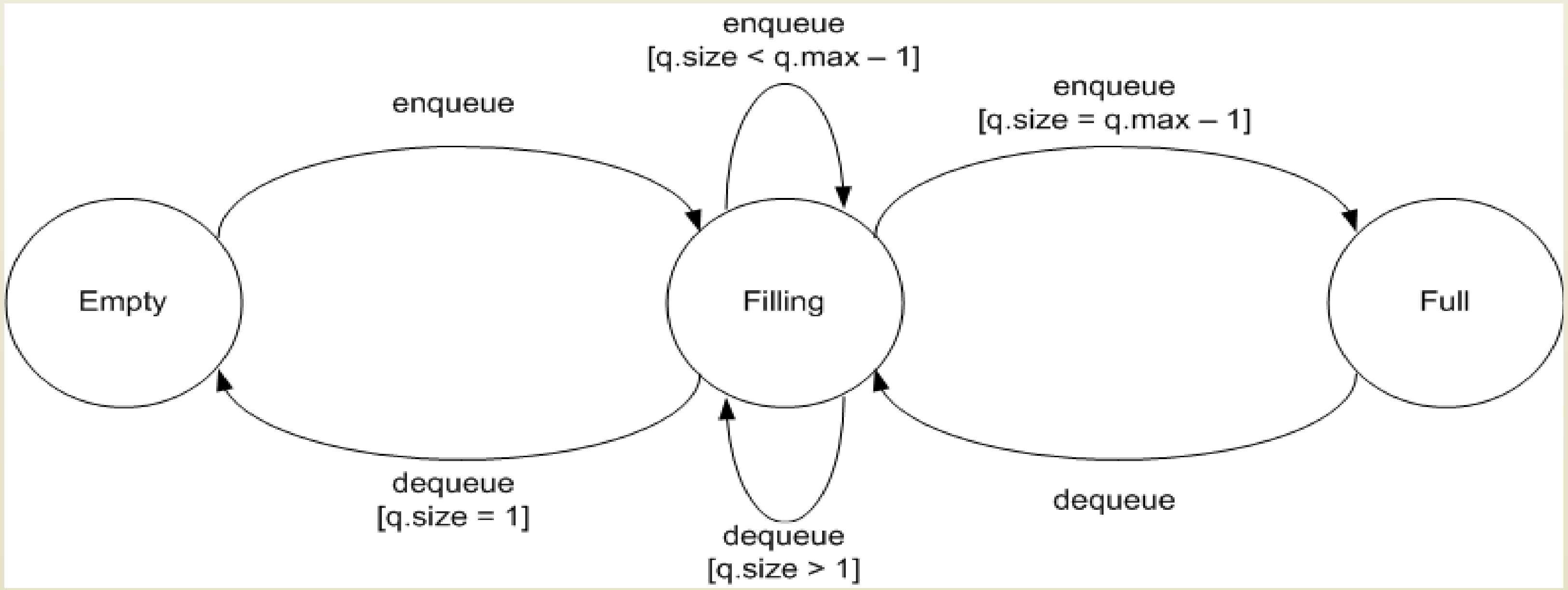
- Therefore, QueueClient assumes that the queue may be empty, and the front operation does not alter the queue
- Now, QueueServer’s provided service behavior for the `front` operation is defined as follows

```
precondition  q.size ≥ 1;
postcondition ~q.size = q.size - 1;
```

  - Here, the precondition asserts that the queue will not be empty and the postcondition specifies that `front` will alter the size of the queue
- Therefore, we have behavioral inconsistency between QueueClient and QueueServer

Interaction Consistency

- Names, interfaces, and behaviors of interacting components may match, yet they may still be unable to interact properly
- Example: QueueClient and QueueServer components



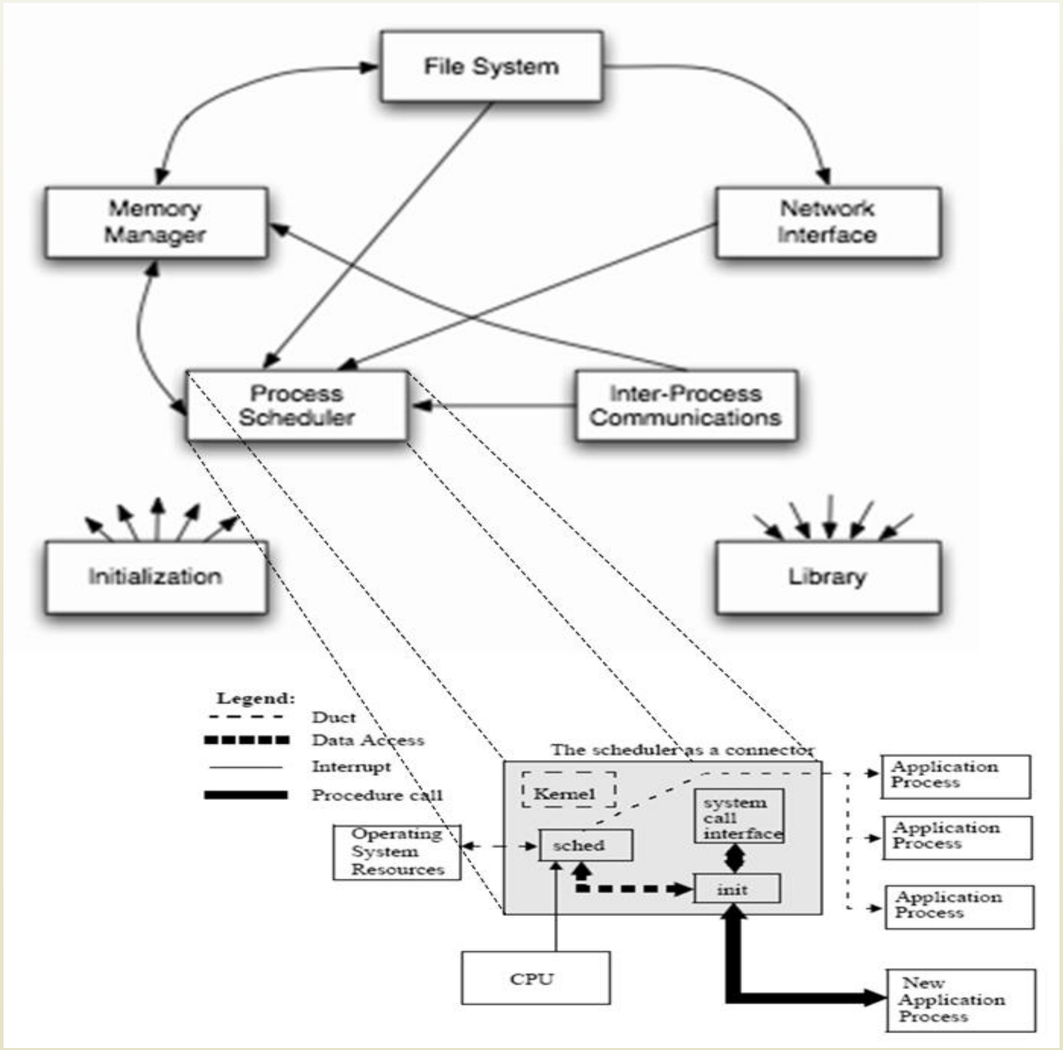
Interaction Consistency (cont'd)

- Interaction inconsistencies occur when a component's provided operations are accessed in a manner that violates certain interaction constraints, such as the order in which the component's operations are to be accessed (the so-called interaction *protocol*)
- The previous diagram specified the interaction protocol of QueueServer; here it is assumed that:
  - At least one element always be enqueued before an attempt to dequeue an element can be made
  - No attempts to enqueue elements onto a full queue will be made
- A QueueClient component that does not adhere to these constraints will cause an interaction inconsistency

Refinement Consistency

- Architectural models are refined during the design process
- A relationship must be maintained between higher- and lower-level models
  - All elements of the higher-level model are preserved in the lower-level one; i.e., no elements have been lost in the refinement process
  - All design decisions and key properties at the higher-level model are preserved in the lower-level one and have not been omitted, changed or violated
  - No new design decisions at the lower-level model violate existing design decisions

Refinement Consistency (cont'd)



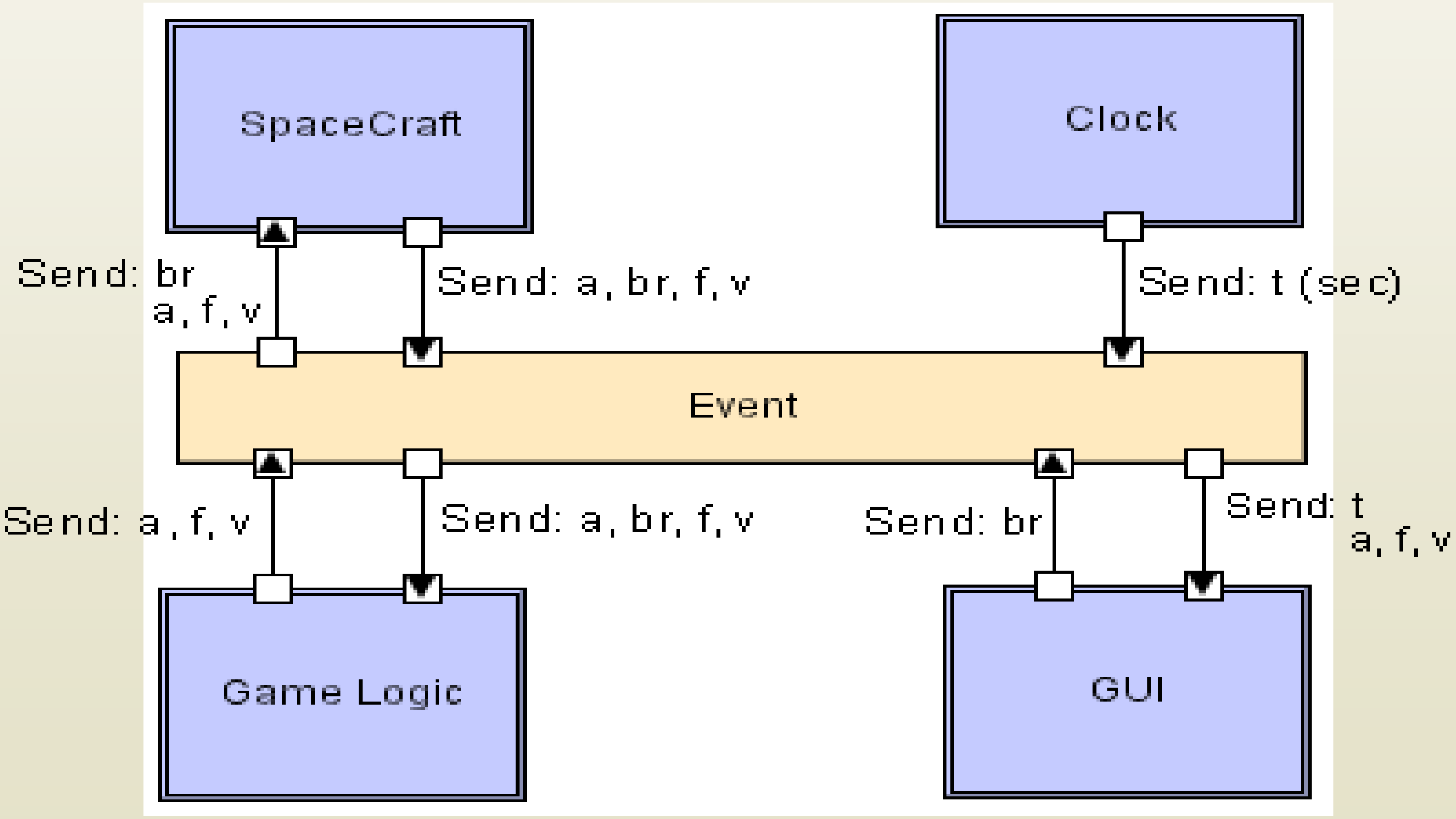
Refinement Consistency (cont'd)

- The previous figure shows at the top the high-level architecture of Linux and at the bottom the architecture of the process scheduler subsystem
- In the Linux architecture, Process Scheduler is modeled as a component while the more elaborate description of it at the bottom models it as a composite connector
- So, Process Scheduler is maintained as a separate entity between the two refinement levels
- However, further analysis and additional information is needed before it is determined whether the lower-level representation of Process Scheduler as a connector rather than a component is a refinement inconsistency

Architectural Analysis Goals – Compatibility

- Compatibility is an external property of an architectural model
- Ensures that the architectural model adheres to guidelines and constraints of
  - A style
  - A reference architecture
  - An architectural standard
- Reference architectures can be specified in ADLs, so compatibility may be a precise and automatable process
- Sometimes though it is not certain which particular style definition is used

Architectural Analysis Goals – Compatibility (cont'd)



Architectural Analysis Goals – Compatibility (cont'd)

- The previous figure depicts the Lunar Lander architecture using the event-based style
- However, a number of aspects of this modeling are not clear:
    - The configuration may also adhere to C2's principles, such as substrate independence
    - `SpaceCraft` may in fact be a blackboard, in which case the visual layout may be misleading
- In such case, more information is needed to make sure that compatibility is preserved

Architectural Analysis Goals – Correctness

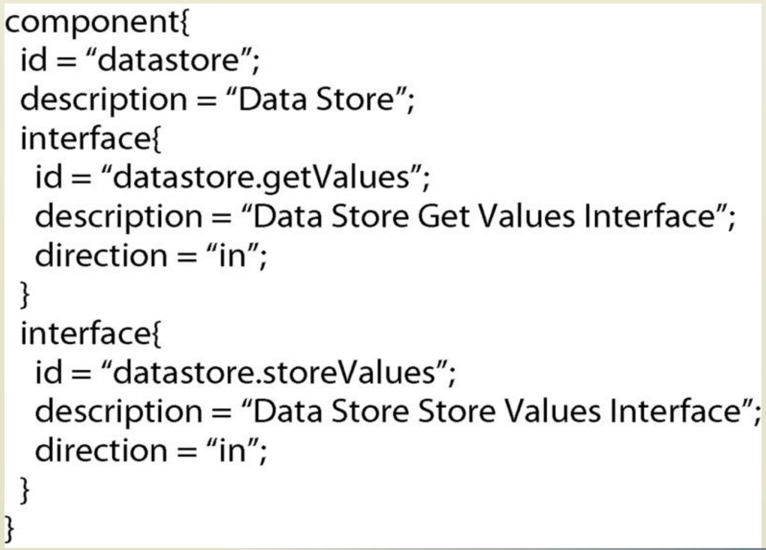
- Correctness is an external property of an architectural model
- Ensures that
  1. The architectural model fully realizes a system specification
  2. The system's implementation fully realizes the architecture
- Inclusion of OTS elements impacts correctness
  - Systems may include structural elements, functionality, and non-functional properties that are not part of the architecture
  - The notion of *fulfillment* is key to ensuring architectural correctness and based on the relative notion of correctness

Scope of Architectural Analysis

- Component- and connector-level
- Subsystem- and system-level
- Data exchanged in a system or subsystem
  - Data structure
  - Data flow
  - Properties of data exchange
- Architectures at different abstraction levels
- Comparison of two or more architectures
  - Processing
  - Data
  - Interaction
  - Configuration
  - Non-functional properties

Component- and Connector-Level Analysis

- The simplest type of component- and connector-level analysis ensures that the given connector or component provides the services expected of it
  - E.g., a component's or connector's interface can be inspected to make sure that no expected services are missing; the code below is part of modeling `Data Store` in xADLite and can be established that it provides both the expected services `getValues` and `storeValues`



```
component{
 id = "datastore";
 description = "Data Store";
 interface{
  id = "datastore.getValues";
  description = "Data Store Get Values Interface";
  direction = "in";
 }
 interface{
  id = "datastore.storeValues";
  description = "Data Store Store Values Interface";
  direction = "in";
 }
}
```

Component- and Connector-Level Analysis (cont'd)

- Checking simply name compatibility to ensure that a component or connector provides an expected service is not enough; e.g., this service may still be modeled with incorrect interfaces
- Also, semantics may have to be checked; e.g., `getValues` may not be modeled such that it accesses `Data Store` to obtain the needed values, but instead may request these values from the user, which (although legitimate in principle) is not the intended functionality of this service
- Also, a connector may provide interaction services with semantics that are different from the expected ones
    - E.g., a connector is expected to support asynchronous invocation, but it has been modeled to support synchronous invocation

Subsystem- and System-Level Analysis

- Even if individual components and connectors have desired properties, it is not necessarily the case that their composition will result in a system that will behave as expected
- Beware of the "honey-baked ham" syndrome: honey is fat free, ham is sugar free, but a honey-baked ham is not fat and sugar free
- In certain cases, it is obvious that a composition of components with some properties each will lead to a system with combined properties from all components
  - E.g., combining a data encryption component with a data compression one, results in a component which is both secure and efficient
- More often is the case where the interplay among components results in interference of their properties
  - In some cases this may be desirable (e.g., sacrificing efficiency for security), in other cases it is not

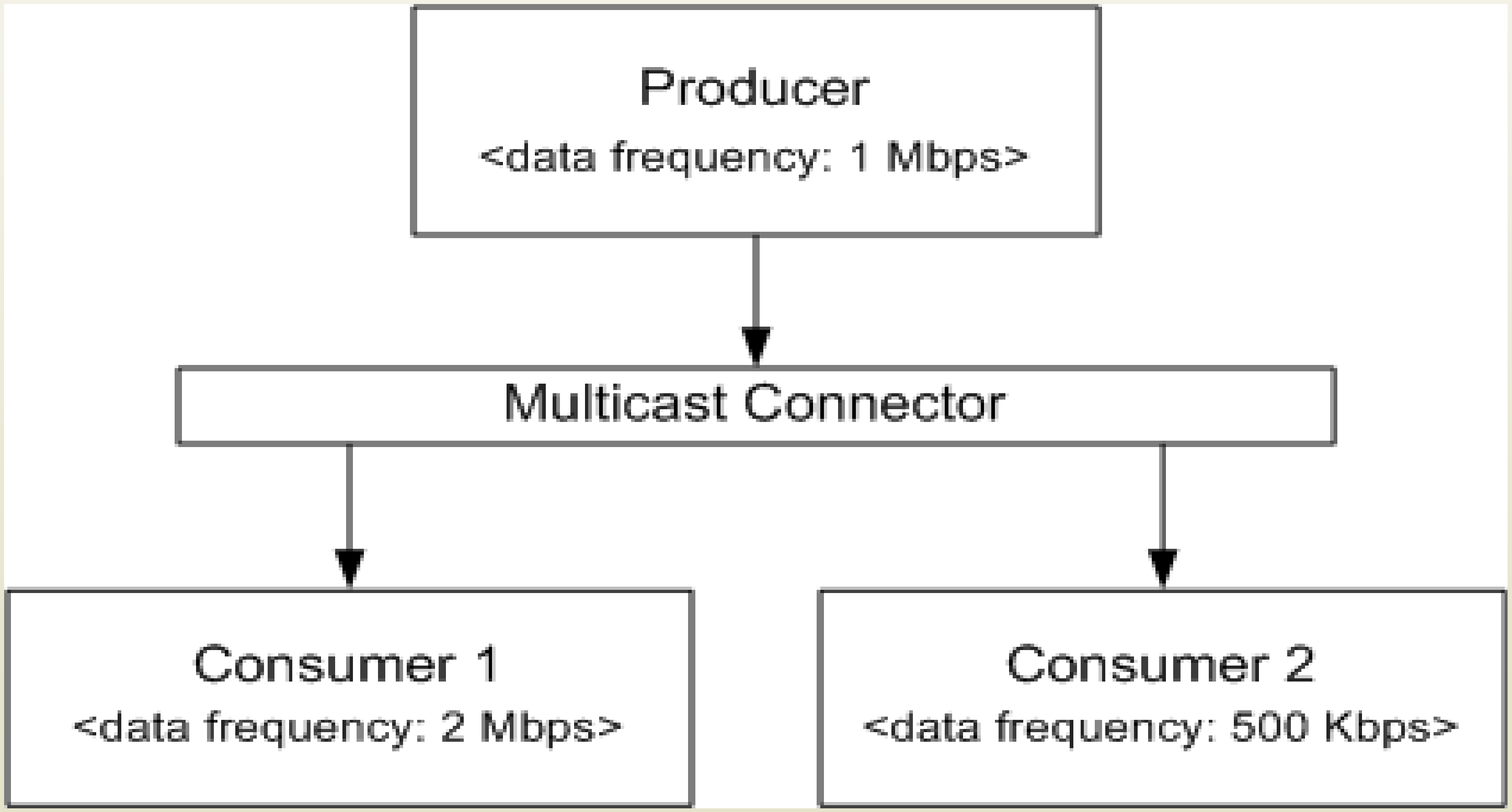
Data Exchanged in the System or Subsystem

- In many large, distributed software systems large amounts of data are processed, exchanged, and stored
- In such systems, it is important to ensure that the system's data is properly modeled, implemented, and exchanged among the structural elements (components, connectors, etc.)
- This involves assessing:
  - *The structure of the data*, such as typed versus untyped or discrete versus streamed
  - *The flow of the data through the system*, such as point-to-point versus broadcast
  - *The properties of data exchange*, such as consistency, security and latency

Data Exchanged in the System or Subsystem (cont'd)

- Consider a system consisting of a data-producer component and two data-consumer components
- The producer sends 1Mps and the consumers are able to receive 2 Mbs and 500Kbs respectively
- While the first consumer may wait idly 50% of its time to retrieve additional data from the producer, the second consumer may lose up to 50% of the produced data, as it can process data at a rate that is only half of that of the producer
- This problem may be mitigated if the three components are connected by means of a Multicast Connector which supports buffering and possibly also additional processing logic that will preserve the temporal order of the data being received by the consumers; other functionality may be included for security or efficiency

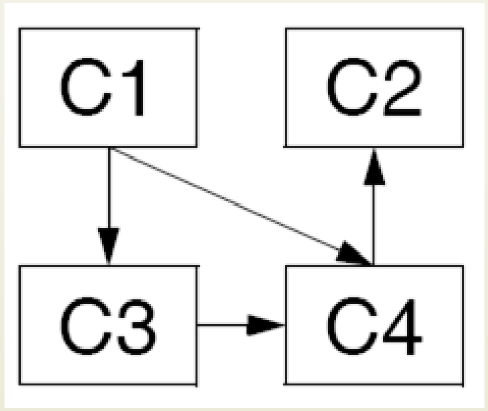
Data Exchanged in the System or Subsystem (cont'd)

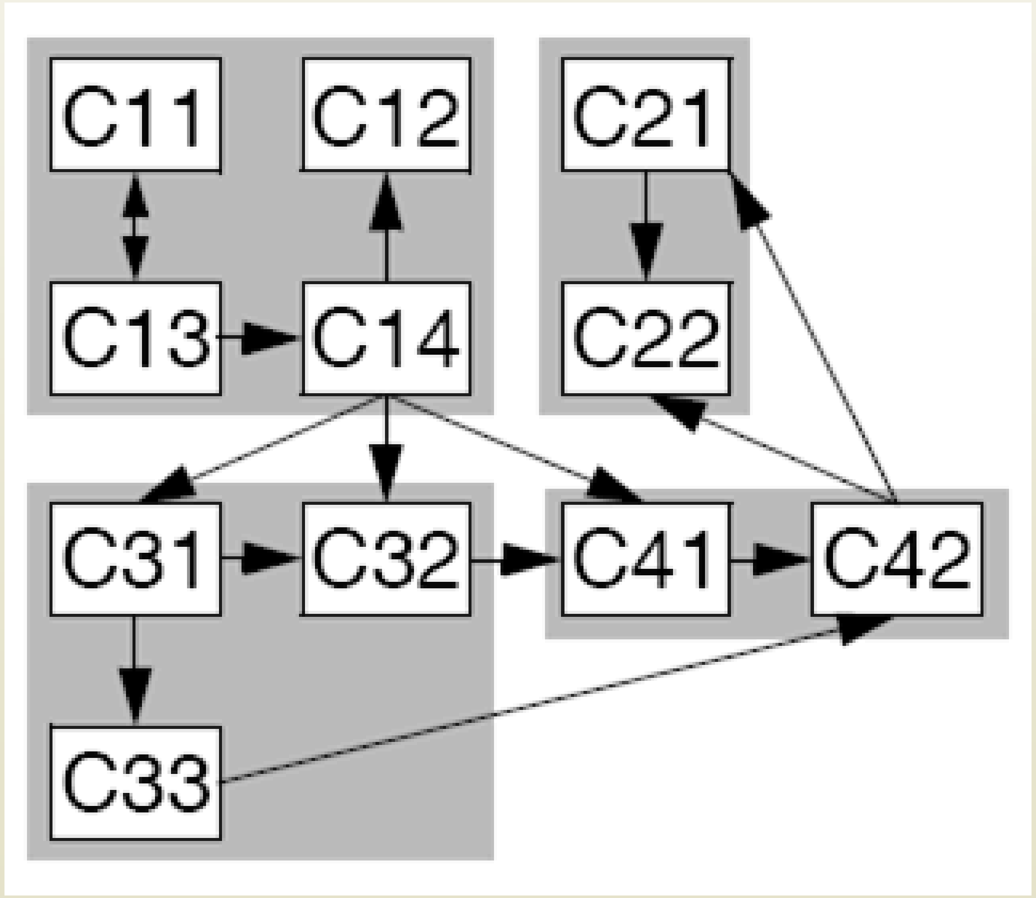


Architectures at Different Abstraction Levels

- Architects frequently address the critical system requirements first, and then both introduce additional elements into the architecture and refine the architecture to include additional details
- The next example shows on the left a high-level architectural breakdown of a system and on the right a refinement of the high-level architecture
- The refinement shows the constituent components comprising each one of the four high-level components as well as a more detailed presentation of connector involvement: e.g., only `C1`'s subcomponent `C14` is engaged in interactions with subcomponents of `C3` and `C4`

Architectures at Different Abstraction Levels (cont'd)





Architectures at Different Abstraction Levels (cont'd)

- As we said before, the process of refinement may introduce inconsistencies and incompatibilities
- E.g., the architecture in the previous example may need to abide by an architectural constrain stating that interactions crossing the boundaries of the system's original four components must have a single target and a single destination
- This constrain is not satisfied in the previous example, both in the case of `C1`'s interactions with `C3`, and `C4`'s interactions with `C2`
    - `C14` interacts with three `C3` subcomponents
    - `C42` interacts with two `C2` subcomponents

Comparison of Two or More Architectures

- In some cases it is important for architects to understand the relationship between the architecture they are interested in and a baseline architecture with known properties
  - E.g., ensuring the compliance of a given system's architecture with a reference architecture
- Such comparisons involve comparing, among others
  - Data storage capabilities provided by the components
  - Interactions as embodied in the connectors
  - Characteristics of the data exchange
  - Components' and connectors' compositions into the system configuration
  - Sources of the non-functional properties exhibited by the system

Architectural Concern Being Analyzed

- Architectural analysis techniques are directed at different facets of a given architecture
  - *Structural characteristics* of an architecture
  - *Behavioral characteristics* of the architectural elements
  - *Interaction characteristics* of the architectural elements
  - *Non-functional characteristics* exhibited by the architecture
- In practice, a given analysis technique, or suite of techniques, will address more than one architectural concern at a time

Structural Characteristics

- These concerns can help to determine whether an architecture is well formed, and include
  - Connectivity among an architecture’s components and connectors
  - Containment of lower-level architectural elements into composite higher-level elements
  - Possible points of network distribution
  - Candidate physical architectures on which a given system can be deployed
- Structural concerns can reveal disconnections of components or connectors from the rest of the system, missing or not intended interactions, etc.

Behavioral Characteristics

- Consider the internal behaviors of individual components
- Consider the architectural structure to assess composite behaviors
- If a system includes third-party components (e.g., OTS), such a behavioral analysis may have to be restricted to those components' public interfaces

Interaction Characteristics

- May include the numbers and types of distinct software connectors, and their values for different connector dimensions
- Interaction characteristics can help to establish whether the architecture will actually be able to fulfil some of its requirements
- Analysis of interaction characteristics may also encompass the interaction protocols for different system components and internal behaviors specified for different system connectors (see previous relevant examples)
- Such analysis may reveal problems related to illegal accessing of a component or possibility for deadlock

Non-Functional Characteristics

- They form a critical dimension of almost all software systems
- Typically cut across multiple components and connectors
- Are often not properly understood, are qualitative in nature, with definitions which are partial or informal
- Their analysis is a formidable challenge to software architects and architectural analysis techniques focusing on these characteristics are scarce

Level of Formality of Architectural Models

- Informal models
  - Typically captured in boxes-and-lines diagrams
  - Amenable to informal and manual analyses
  - More useful to non-technical stakeholders; e.g., a manager can determine a project's staffing needs
- Semi-formal models (e.g., UML)
  - Try to strike a balance between precision and formality on the one hand, with expressiveness and understandability on the other hand
  - Amenable to both manual and automated analysis
  - Useful to both technical and non-technical stakeholders
- Formal models (e.g., Wright)
  - They have both a formal notation as well as a formal semantics
  - Inherently amenable to formal, automated analysis
  - Typically intended for a system's technical stakeholders
  - Steep learning curve and scalability problems

Type of Analysis

- Static analysis
  - Involves inferring the properties of a software system from one or more of its models, without actually executing those models (e.g., syntactic analysis)
  - Can be automated (e.g., compilation) or manual (by inspection)
  - All architectural modeling notations are amenable to static analysis
- Dynamic analysis
  - Involves actually executing or simulating the execution of a model of a software system (e.g., state-transition diagrams)
- Scenario-based analysis
  - For large and complex software systems, where it is often infeasible to assert that a given property is valid for the entire system over all possible states or executions, it is preferable to examine specific use cases that represent the most important or common scenarios
  - Can be both static and dynamic

Level of Automation

- Manual
  - Requires significant human involvement and thus it is expensive
  - But it can be performed on models of varying levels of detail, rigor, formality, and completeness or when multiple, potentially clashing properties must be ensured in tandem
  - Architectural rationale can be taken into account
  - Analysis results are typically qualitative
- Partially automated
  - Most ADLs are amenable to a partial automated analysis, e.g., syntactic and/or semantic correctness related, say, to interconnectivity (e.g., xADL) or deadlock (e.g., Wright)
  - But other properties, such as availability, dependability, latency or reliability cannot be analyzed automatically
- Fully automated
  - Effectively combining partially automated techniques with human intervention

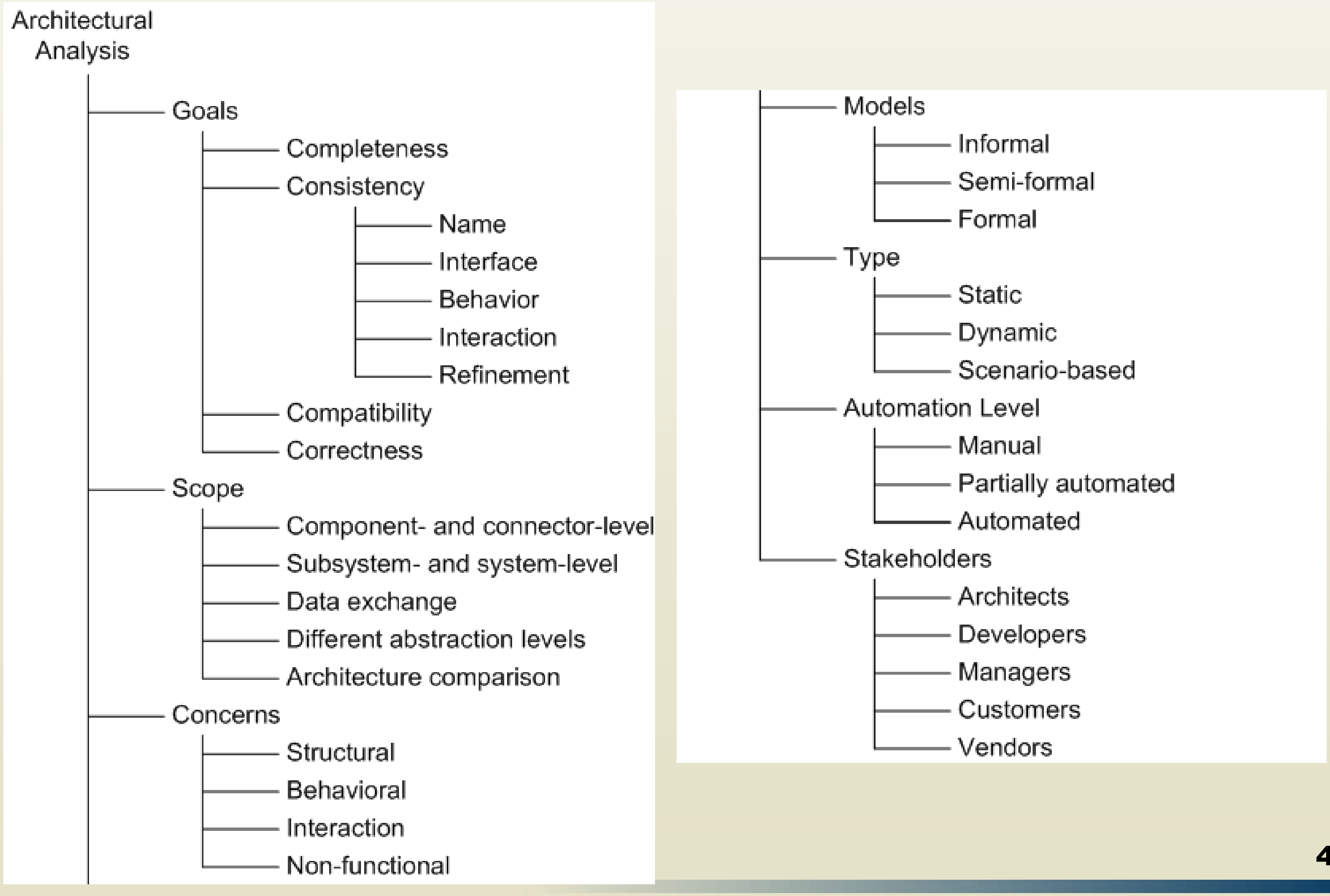
System (Analysis) Stakeholders

- Architects
  - Take a global view of the architecture and are interested in establishing all four Cs
  - May need to rely on all levels of architectural models at all levels of scope and formality
  - Frequently they really on both manual and semi-automated techniques
- Developers
  - Often take a more limited view of the architecture (modules or subsystems for which they are responsible)
  - Primarily interested in establishing consistency of their modules with other parts of the system and they need not worry about the architecture's completeness
  - Typically, they prefer formal modeling paradigms

System (Analysis) Stakeholders (cont'd)

- Managers
  - Typically interested in architectural completeness and correctness
- Customers
  - Is the development organization building the right system?
  - Is the development organization building the system right?
  - Typically favor understandability over formality
  - Interested in overall models and the system's properties
- Vendors
  - Typically, they sell technology (individual components or connectors) rather than architecture
  - As such, they are interested primarily in composability of those components and connectors as well as their compatibility with certain standards and widely used reference architectures

Architectural Analysis in a Nutshell



- Architectural Analysis
  - Goals
    - Completeness
    - Consistency
      - Name
      - Interface
      - Behavior
      - Interaction
      - Refinement
    - Compatibility
    - Correctness
  - Scope
    - Component- and connector-level
    - Subsystem- and system-level
    - Data exchange
    - Different abstraction levels
    - Architecture comparison
  - Concerns
    - Structural
    - Behavioral
    - Interaction
    - Non-functional
  - Models
    - Informal
    - Semi-formal
    - Formal
  - Type
    - Static
    - Dynamic
    - Scenario-based
  - Automation Level
    - Manual
    - Partially automated
    - Automated
  - Stakeholders
    - Architects
    - Developers
    - Managers
    - Customers
    - Vendors

Analysis Techniques

- Inspection- and review-based
- Model-based
- Simulation-based

Architectural Inspections and Reviews

- Architectural models studied by human stakeholders for specific properties
- The stakeholders define analysis objective
- Manual techniques
  - Can be expensive
- Useful in the case of informal architectural descriptions
- Useful in establishing “soft” system properties
  - E.g., scalability or adaptability
- Able to consider multiple stakeholders’ objectives and multiple architectural properties

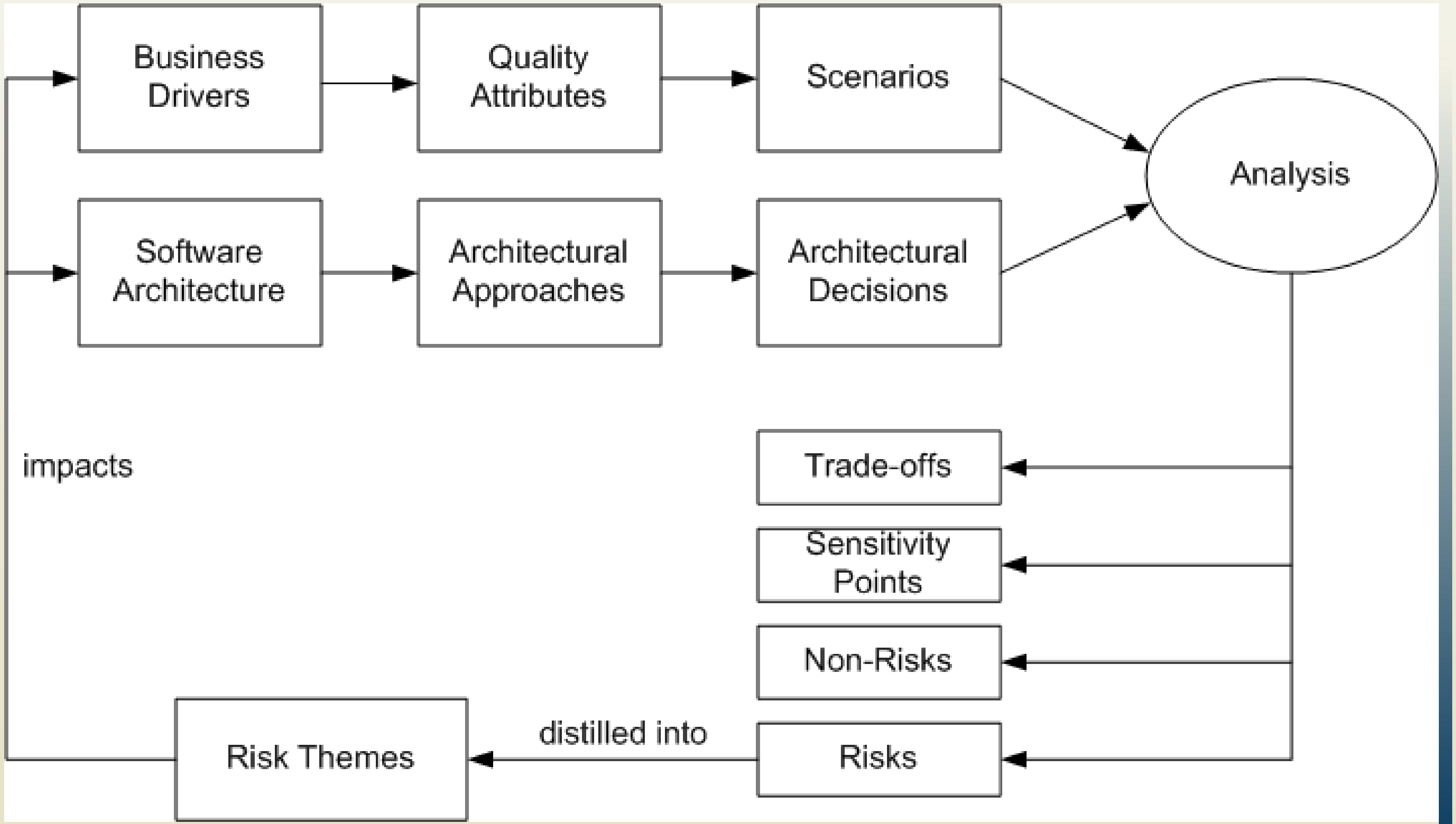
Inspections and Reviews in a Nutshell

- *Analysis Goals* – any
- *Analysis Scope* – any
- *Analysis Concern* – any, but particularly suited for non-functional properties
- *Architectural Models* – any, but must be geared to stakeholder needs and analysis objectives
- *Analysis Types* – mostly static and scenario-based
- *Automation Level* – manual, human intensive
- *Stakeholders* – any, except perhaps component vendors

Example – ATAM

- Stands for Architectural Trade-off Analysis Method
- Human-centric process for identifying risks early on in software design
- Focuses specifically on four quality attributes (NFPs)
    - Modifiability
    - Security
    - Performance
    - Reliability
- Reveals how well an architecture satisfies quality goals and how those goals trade-off

ATAM Process



ATAM Business Drivers

- The system's critical functionality
- Any technical, managerial, economic, or political constraints
- The project's business goals and context
- The major stakeholders
- The principal quality attribute (NFP) goals that impact and shape the architecture
- The quality attributes become a basis of eliciting a set of representative scenarios that will help ensure the system's satisfaction of those attributes
- There are three such scenario categories

ATAM Scenarios

- Use-case scenarios
  - Describe how the system is envisioned by the stakeholders to be used
- Growth scenarios
  - Describe planned and envisioned modifications to the architecture
- Exploratory scenarios
  - Try to establish the limits of architecture’s adaptability by postulating major changes to
    - System’s functionality
    - Operational profiles
    - Underlying execution platforms
  - Scenarios are prioritized based on importance to stakeholders

ATAM Architecture

- Technical constraints
  - Required hardware platforms, OS, middleware, programming languages, and OTS functionality
- Any other systems with which the system must interact
- *Architectural approaches* that have been used to meet the quality requirements
  - Sets of architectural design decisions employed to solve a problem
  - Typically, architectural patterns and styles
  - The architectural approaches are used to elaborate the architectural design decisions made for the system

ATAM Analysis

- Key step in ATAM
- Objective is to establish relationship between architectural approaches and quality attributes
- For each architectural approach a set of analysis questions are formulated
  - Targeted at the approach and quality attributes in question
- System architects and ATAM evaluation team work together to answer these questions and identify
  - Risks → these are distilled into risk *themes*
  - Non-Risks
  - Sensitivity points
  - Trade-off points
- Based on answers, further analysis may be performed

ATAM in a Nutshell

Goals	Completeness Consistency Compatibility Correctness`
Scope	Subsystem- and system-level Data exchange
Concern	Non-functional
Models	Informal Semi-formal
Type	Scenario-driven
Automation Level	Manual
Stakeholders	Architects Developers Managers Customers

Model-Based (Architectural) Analysis

- Analysis techniques that manipulate architectural description to discover architectural properties
- Tool-driven, hence potentially less costly
- Typically, useful for establishing “hard” architectural properties only
  - Unable to capture design intent and rationale
- Usually focus on a single architectural aspect
  - E.g., syntactic correctness, deadlock freedom, adherence to a style
- Scalability may be an issue
- Techniques typically used in tandem to provide more complete answers

Model-Based Analysis in a Nutshell

- *Analysis Goals* – consistency, compatibility, internal correctness
- *Analysis Scope* – any
- *Analysis Concern* – structural, behavioral, interaction, and possibly non-functional properties
- *Architectural Models* – semi-formal and formal
- *Analysis Types* – static
- *Automation Level* – partially and fully automated
- *Stakeholders* – mostly architects and developers

Model-Based Analysis in ADLs

- Wright – uses CSP to analyze for deadlocks
- Aesop – ensures style-specific constraints
- MetaH and UniCon – support schedulability analysis via NFPs such as component criticality and priority
- ADL parsers and compilers – ensure syntactic and semantic correctness
  - E.g., Rapide's generation of executable architectural simulations
- Architectural constraint enforcement
  - E.g., Armani or UML's OCL
- Architectural refinement
  - E.g., SADL and Rapide

ADLs’ Analysis Foci in a Nutshell

Goals	Consistency Compatibility Completeness (internal)
Scope	Component- and connector-level Subsystem- and system-level Data exchange Different abstraction levels Architecture comparison
Concern	Structural Behavioral Interaction Non-functional
Models	Semi-formal Formal
Type	Static
Automation Level	Partially automated Automated
Stakeholders	Architects Developers Managers Customers

(Architectural) Reliability Analysis

- *Reliability* is the probability that the system will perform its intended functionality under specified design limits, without failure
- A *failure* is the occurrence of an incorrect output as a result of an input value that is received, with respect to the specification
- An *error* is a mental mistake made by the designer or programmer
- A *fault* or a *defect* is the manifestation of that error in the system
  - An abnormal condition that may cause a reduction in, or loss of, the capability of a component to perform a required function
  - A requirements, design, or implementation flaw or deviation from a desired or intended state

Reliability Metrics

- Time to failure
  - Mean time until a system fails after its last restoration
- Time to repair
  - Mean time until a system is repaired after its last failure
- Time between failures
  - Mean time between two system failures

Assessing Reliability at Architectural Level

- Challenged by unknowns
  - Operational profile
  - Failure and recovery history
- Challenged by uncertainties
  - Multiple development scenarios
  - Varying granularity of architectural models
  - Different information sources about system usage
- Architectural reliability values must be qualified by assumptions made to deal with the above uncertainties
- Reliability modeling techniques are needed that deal effectively with uncertainties
  - E.g., Hidden Markov Models (HMMs)

Architectural Reliability Analysis in a Nutshell

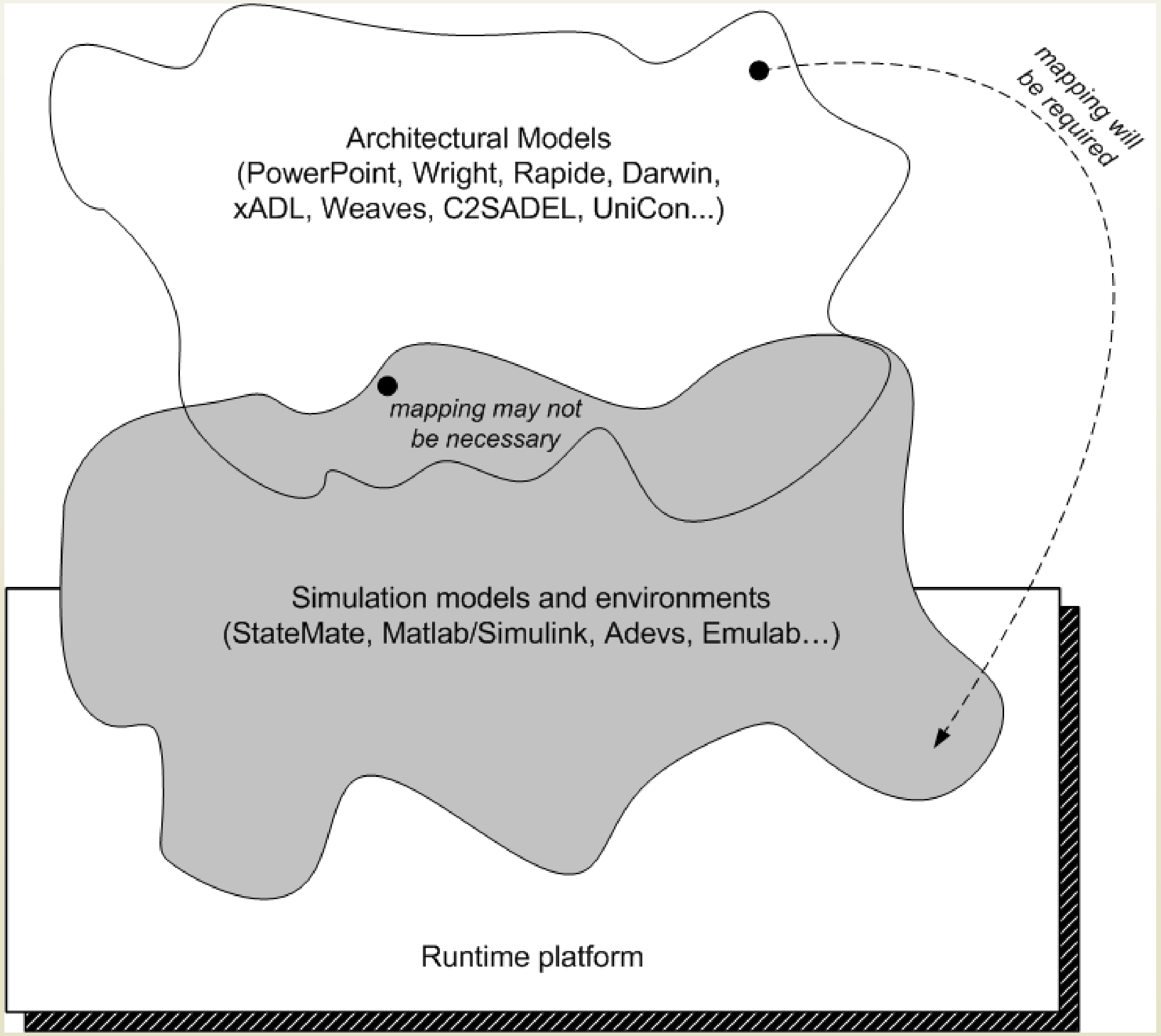
Goals	Consistency Compatibility Correctness
Scope	Component- and connector-level Subsystem- and system-level
Concern	Non-functional
Models	Formal
Type	Static Scenario-based
Automation Level	Partially automated
Stakeholders	Architects Managers Customers Vendors

Simulation-Based Analysis

- Requires producing an executable system model
- Simulation need not exhibit identical behavior to system implementation
  - Many low-level system parameters may be unavailable
- It needs to be precise and not necessarily accurate
- Some architectural models may not be amenable to simulation
  - Typically require translation to a simulatable language

Architectural and Simulation Models

- Some models such as UML or Rapide can be simulated directly; others need to be mapped to the simulation substrate


Architectural Models
(PowerPoint, Wright, Rapide, Darwin,
xADL, Weaves, C2SADEL, UniCon...)
mapping will be required
mapping may not be necessary
Simulation models and environments
(StateMate, Matlab/Simulink, Adevs, Emulab...)
Runtime platform

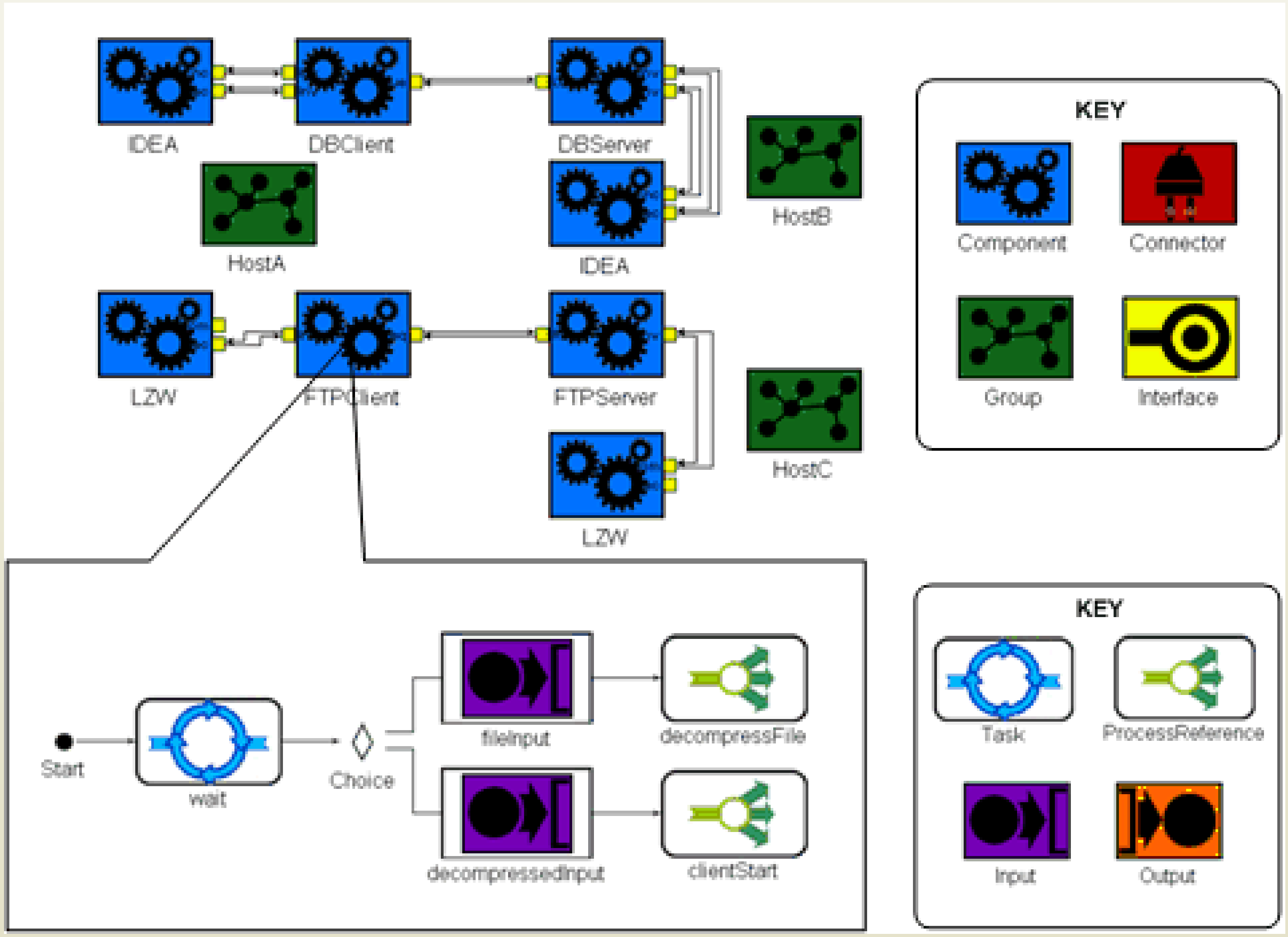
Simulation-Based Analysis in a Nutshell

- *Analysis Goals* – any
- *Analysis Scope* – any
- *Analysis Concern* –behavioral, interaction, and non-functional properties
- *Architectural Models* – formal
- *Analysis Types* – dynamic and scenario-based
- *Automation Level* – fully automated; model mapping may be manual
- *Stakeholders* – any

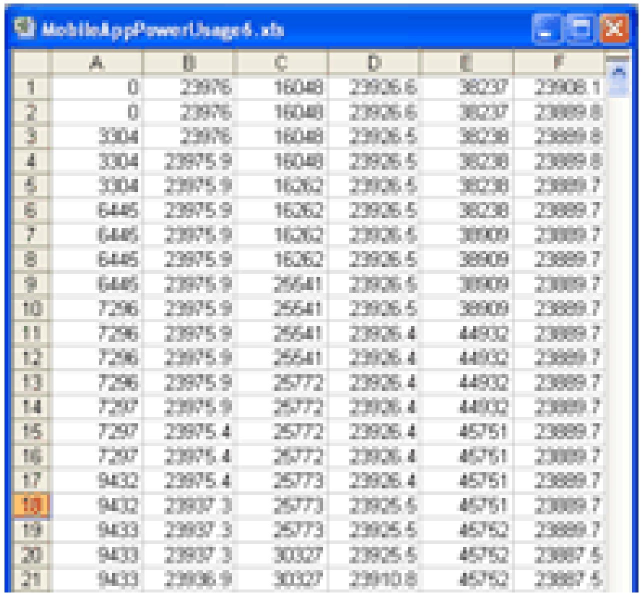
Example – XTEAM

- eXtensible Tool-chain for Evaluation of Architectural Models
- Targeted at mobile and resource-constrained systems
- Combines two underlying ADLs
  - xADL and FSP (Finite State Processes)
- Maps architectural description to Adevs (a discrete event simulator)
  - An OTS event simulation engine
- Implements different analyses via ADL extensions and a model interpreter
  - Latency, memory utilization, reliability, energy consumption

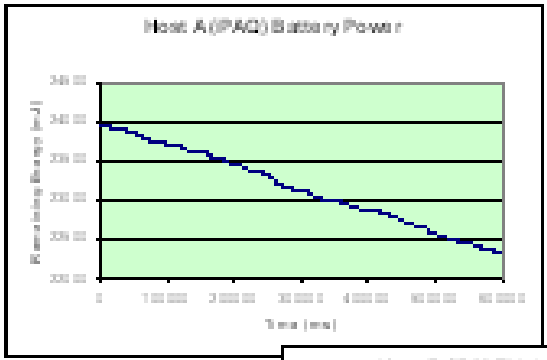
Example XTEAM Model

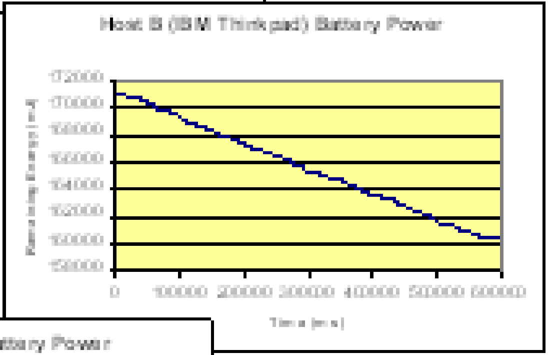


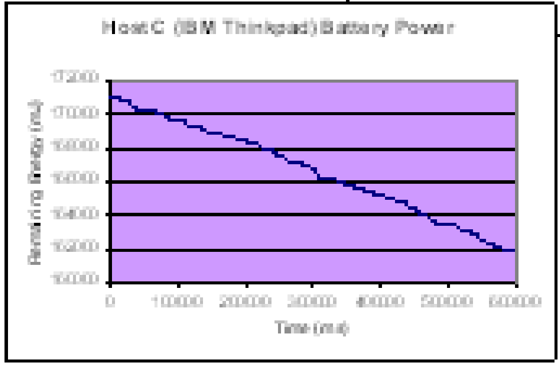
Example XTEAM Analysis



	A	B	C	D	E	F
1	0	23976	16048	23926.6	38237	23908.1
2	0	23976	16048	23926.6	38237	23889.8
3	3304	23976	16048	23926.5	38238	23889.8
4	3304	23975.9	16048	23926.5	38238	23889.8
5	3304	23975.9	16262	23926.5	38238	23889.7
6	6445	23975.9	16262	23926.5	38238	23889.7
7	6445	23975.9	16262	23926.5	38909	23889.7
8	6445	23975.9	16262	23926.5	38909	23889.7
9	6445	23975.9	25541	23926.5	38909	23889.7
10	7296	23975.9	25541	23926.5	38909	23889.7
11	7296	23975.9	25541	23926.4	44932	23889.7
12	7296	23975.9	25541	23926.4	44932	23889.7
13	7296	23975.9	25772	23926.4	44932	23889.7
14	7297	23975.9	25772	23926.4	44932	23889.7
15	7297	23975.4	25772	23926.4	45751	23889.7
16	7297	23975.4	25772	23926.4	45751	23889.7
17	9432	23975.4	25773	23926.4	45751	23889.7
18	9432	23937.3	25773	23925.5	45751	23889.7
19	9433	23937.3	25773	23925.5	45752	23889.7
20	9433	23937.3	30327	23925.5	45752	23887.5
21	9433	23936.9	30327	23910.8	45752	23887.5







XTEAM in a Nutshell

Goals	Consistency Compatibility Correctness
Scope	Component- and connector-level Subsystem- and system-level Data exchange
Concern	Structural Behavioral Interaction Non-functional
Models	Formal
Type	Dynamic Scenario-based
Automation Level	Automated
Stakeholders	Architects Developers Managers Customers Vendors

Summary

- Architectural analysis is neither easy nor cheap
- The benefits typically far outweigh the drawbacks
- Early information about the system's key characteristics is indispensable
- Multiple analysis techniques often should be used in concert
- "How much analyses?"
  - This is the key facet of an architect's job
  - Too many will expend resources unnecessarily
  - Too few will carry the risk of propagating defects into the final system
  - Wrong analyses will have both drawbacks