Verifying the Architecture

István Majzik, Zoltán Micskei

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group

Main topics of the course

- Overview (1)
  - V&V techniques, Critical systems
- Static techniques (2)
  - Verifying specifications
  - Verifying source code
- Dynamic techniques: Testing (7)
  - Developer testing, Test design techniques
  - Testing process and levels, Test generation, Automation
- **System-level verification (3)**
  - **Verifying the architecture**, Dependability analysis
  - Runtime verification

Table of Contents

- Introduction
  - Architecture design and languages
  - What is determined by the architecture?
  - What kind of verification methods can be used?
- Requirements based architecture analysis
  - ATAM: Architecture Trade-off Analysis
- Systematic analysis methods
  - Interface analysis
  - Rule based checking
  - Fault effects analysis
- Model based evaluation
  - Performance evaluation

Learning outcomes

- Explain the activities and tasks in the typical architecture verification process (K2)
- List what system level properties are determined by the architecture (K1)
- Recall the analysis process in ATAM (K1)
- Perform fault effect analysis with fault trees and event tree analysis (K3)
- Identify how models can be used for performance evaluation (K1)

INTRODUCTION

Architecture design and languages

What is determined by the architecture?

What kind of verification methods can be used?

Architecture design

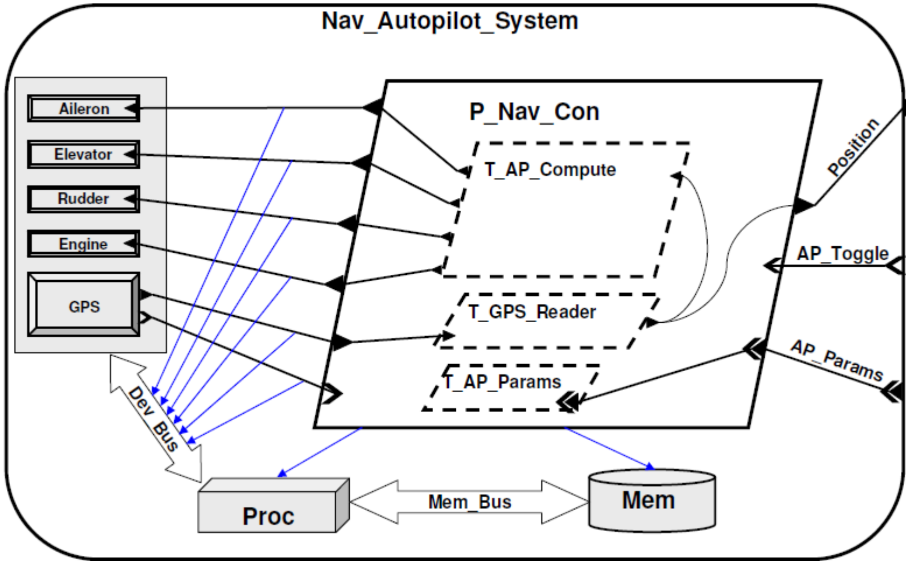
- What is the architecture?
  - Components (with properties)
  - Relations among them (use of service, deployment, …)
- Design decisions
  - Selecting components and specifying their relations
    - System functions implemented by interactions of components
    - Hardware-software interactions
  - Specifying properties of components
    - Influences performance, reliability, testability, ...
  - Using architecture design patterns
    - E.g., MVC, N-tier, …
    - Supports maintainability
  - Re-use (off-the-shelf and available components)

Typical languages for architecture design

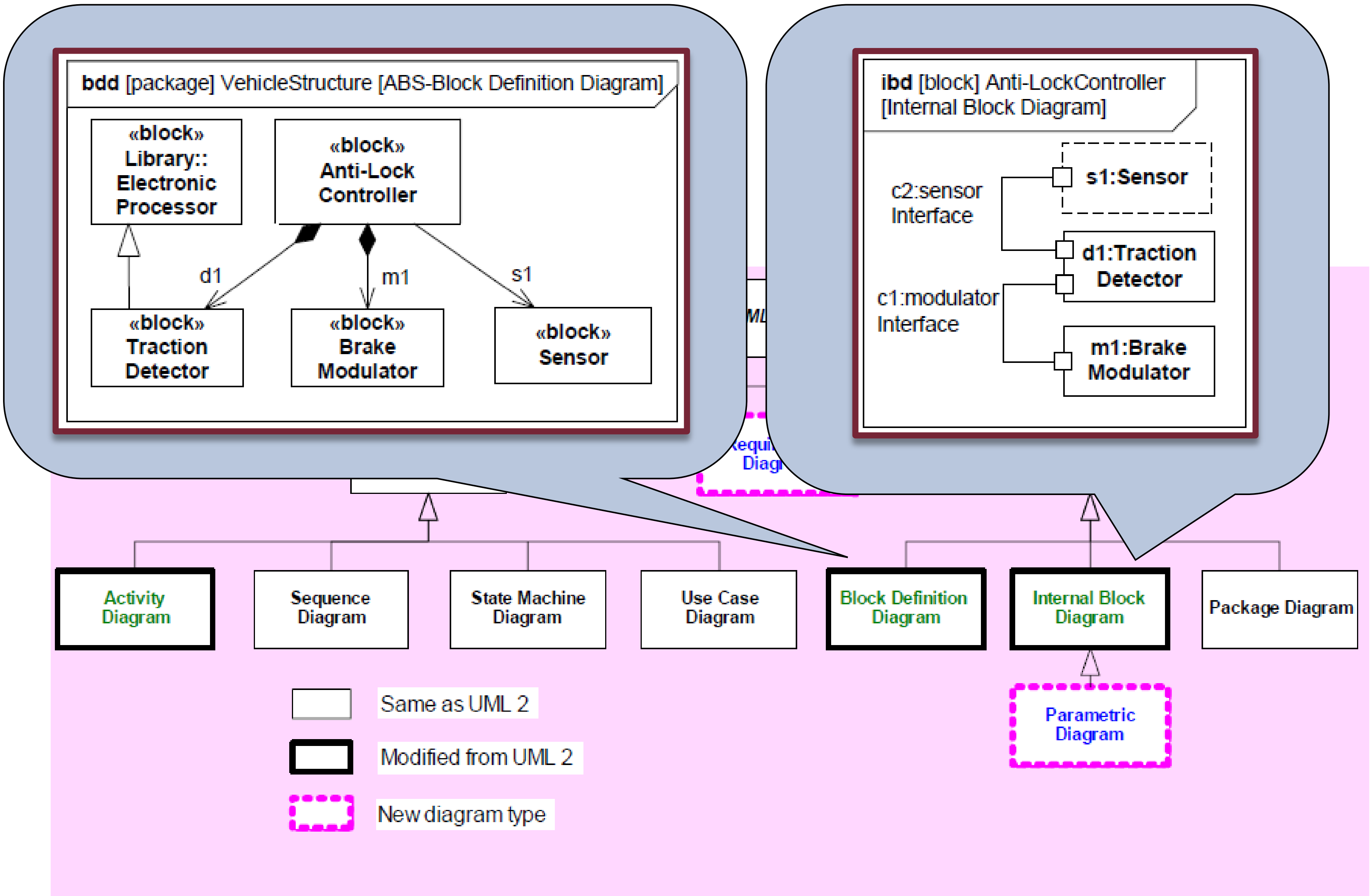
- UML
- SysML (e.g., Block diagram)
- AADL: Architecture Analysis and Design Language
  - Components
  - Relations: Data/event interchange on ports
  - Mapping to hardware
  - Properties for analysis

```
thread implementation CoinPublisher.impl
         calls(u: subprogram updateTotal;);
     properties

         Compute_Execution_Time => 30ms .. 40ms;
         Dispatch_Protocol => ( Sporadic );
         annex behavior {**
              compute(5ms);
              compute(10ms);
              compute(15ms);
              raise(availableContent);
         **};
end CoinPublisher.impl;
```



Typical languages for architecture design: SysML

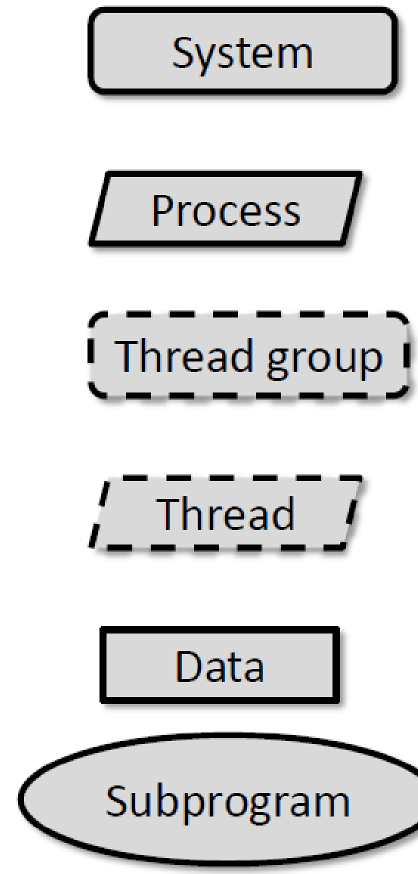


Typical languages for architecture design: AADL

AADL: Architecture Analysis and Design Language (v2: 2009)

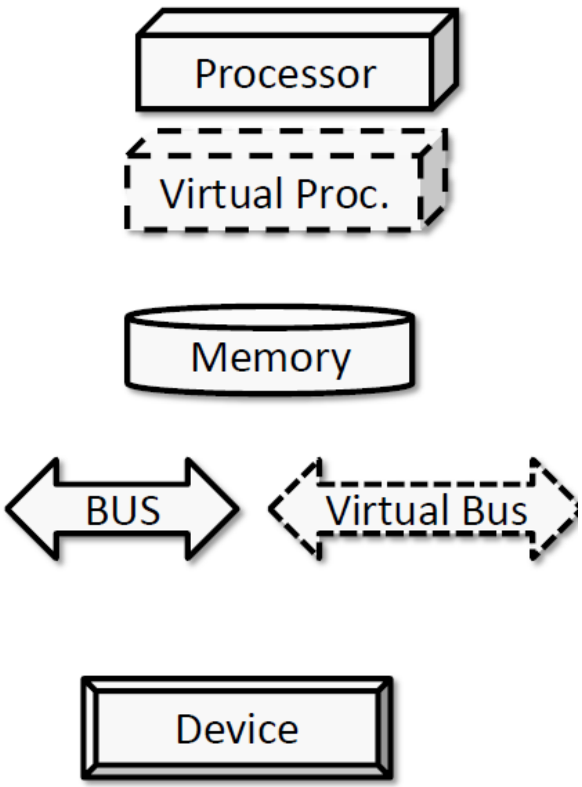
- For embedded systems (SAE)

- **Software components**
  - System: Hierarchic structure of components
  - Process: Protected address range
  - Thread group: Logic group of threads
  - Thread: Concurrently schedulable execution unit
  - Data: Sharable data
  - Subprogram: Sequential, callable code unit



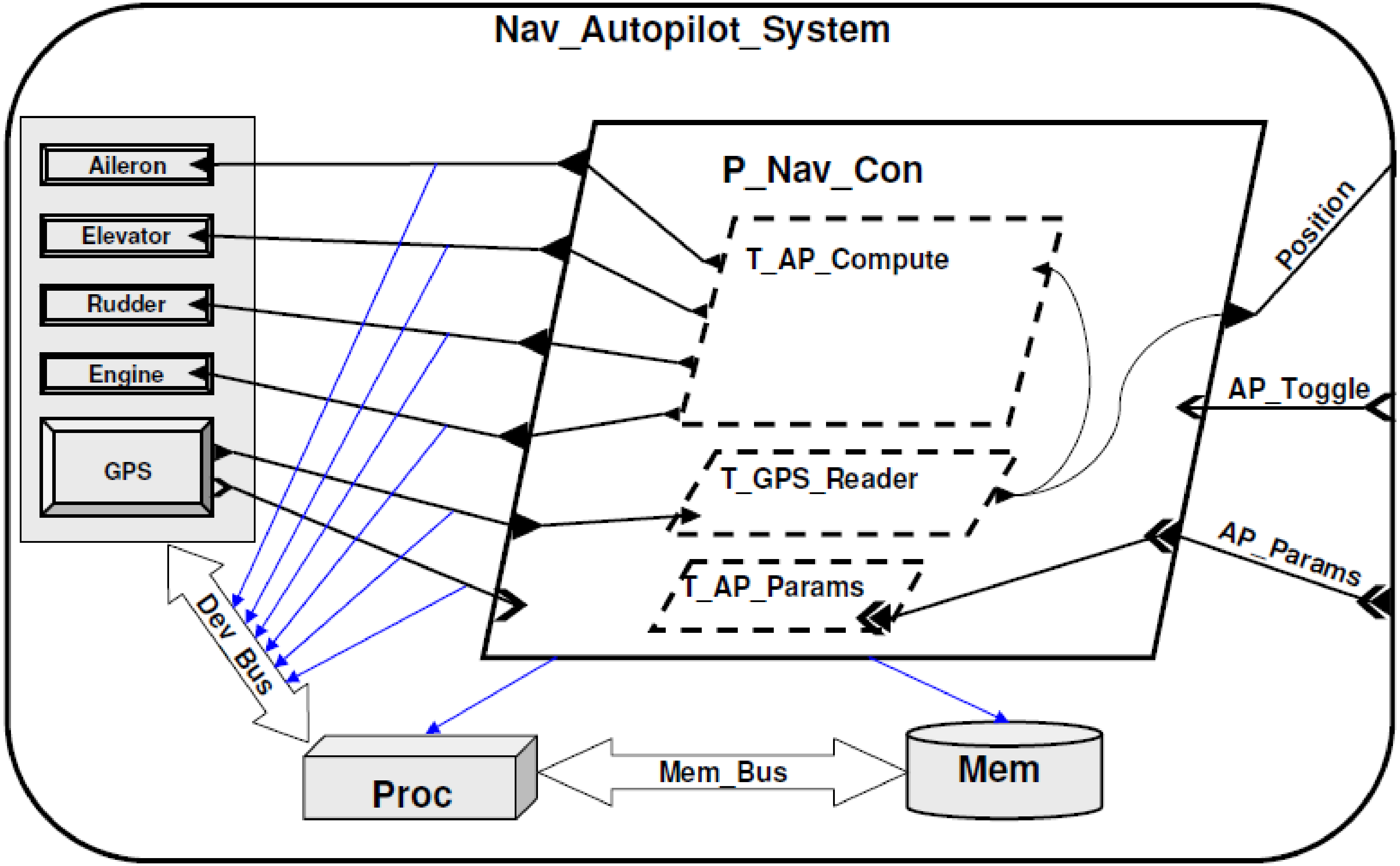
Typical languages for architecture design: AADL

- **Hardware components**
  - Processor, Virtual Processor: Platform for scheduling of threads/processes
  - Memory: Storage for data and executable code
  - Bus, Virtual Bus: Physical or logical unit of connection
  - Device: Interface to/from external environment
- **Mapping**
  - Between software and hardware
  - Between logical (virtual) and physical components



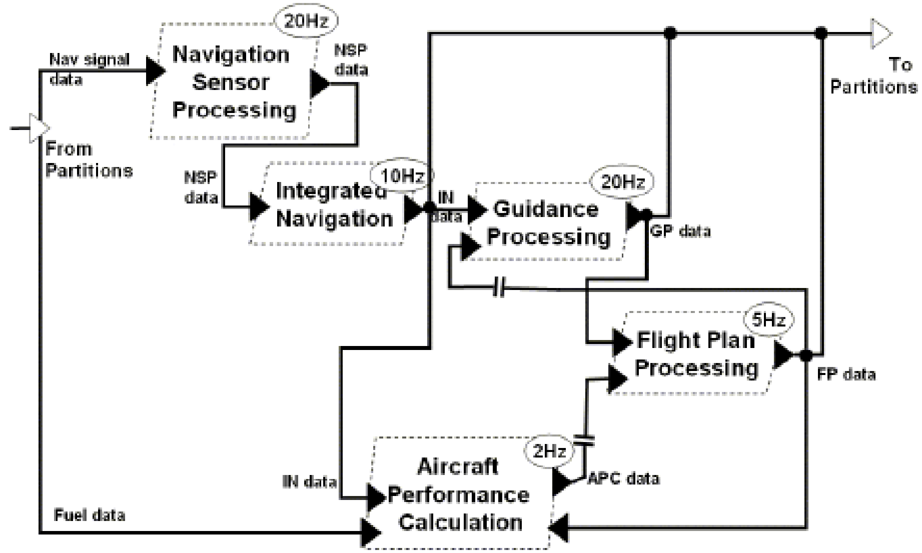
Typical languages for architecture design: AADL

- Example: Mapping between components



Typical languages for architecture design: AADL

- Relations
  - Data and event flow on ports
- Property specification for analysis
  - Timing
  - Scheduling
  - Error propagation (using an extension of AADL)
- Models in graphical, textual, XML formats




```
thread implementation CoinPublisher.impl
            calls(u: subprogram updateTotal;);
      properties

            Compute_Execution_Time => 30ms .. 40ms;
            Dispatch_Protocol => ( Sporadic );
            annex behavior {**
                  compute(5ms);
                  compute(10ms);
                  compute(15ms);
                  raise(availableContent);
            **};
end CoinPublisher.impl;
```

What is determined by the architecture? 1/2

- **Performance**
  - Resource assignment: Parallel processing, queuing policy, deployment of critical services
  - Resource management: Scheduling of resources, dynamic assignment, load balancing
- **Dependability**
  - Error detection: Push/pull monitoring, exception handling
  - Fault tolerance: Static redundancy, dynamic redundancy
  - Fault handling: Reconfiguration, graceful degradation
- **Security**
  - Protection of sensitive data: Components for authentication, authorization, data hiding
  - Detection of intrusion: Confinement of illegal changes
  - Recovery after intrusion: Maintenance of data integrity

What is determined by the architecture? 2/2

- **Maintainability**
  - Encapsulation: Semantic coherence
  - Avoiding domino effects of changes: Information hiding, error confinement, usage of proxies
  - Late binding: Runtime registration, configuration descriptors, polymorphism
- **Testability**
  - Assuring controllability and observability
  - Separation of interfaces and implementation
  - Recording and replaying interactions
- **Usability**
  - Separation of user interface
  - Maintenance of internal models (user model, task model, environment model) in runtime

Example: Architecture for software safety (EN 50128)

- Highly recommended techniques for SIL 3 and SIL 4
  - Diverse programming
  - Fault detection and diagnostics
  - Failure assertion programming
  - Defensive programming
  - Storing executed cases
  - (Software fault effect analysis)

  –> Software, information and time redundancy

Combination of techniques is allowed

Reference for error detection

- Not recommended techniques
  - Forward and backward recovery
  - Artificial intelligence based fault handling
  - Dynamic software reconfiguration

Operation is hard to predict in design time

Summary: System properties and the design space

System property	Architectural decisions (examples)
Performance	Resource assignment, resource management
Dependability	Error detection and confinement, fault tolerance, fault handling
Security	Protection against illegal access, detection of intrusion, maintenance
Maintainability	Localizing, avoiding domino effect, late binding
Testability	Controllability, observability, separation of interfaces
Usability	Separation of UI, maintenance of user, task and environment model

Overview: What are the verification techniques?

- Review: Requirement based architecture analysis
  - Architecture trade-off analysis (ATAM)
- Static analysis: Systematic checking of the architecture
  - Interface analysis
    - Conformance of required and offered interfaces
  - Rule based checking of the architecture
    - Dependencies, containment, inheritance etc.
  - Fault effect analysis by combinational techniques
    - Component level faults ↔ System level effects
- Quantitative analysis: Model based evaluation
  - Evaluation of extra-functional properties by constructing and solving an analysis model
    - Computing system level properties by solving the analysis model

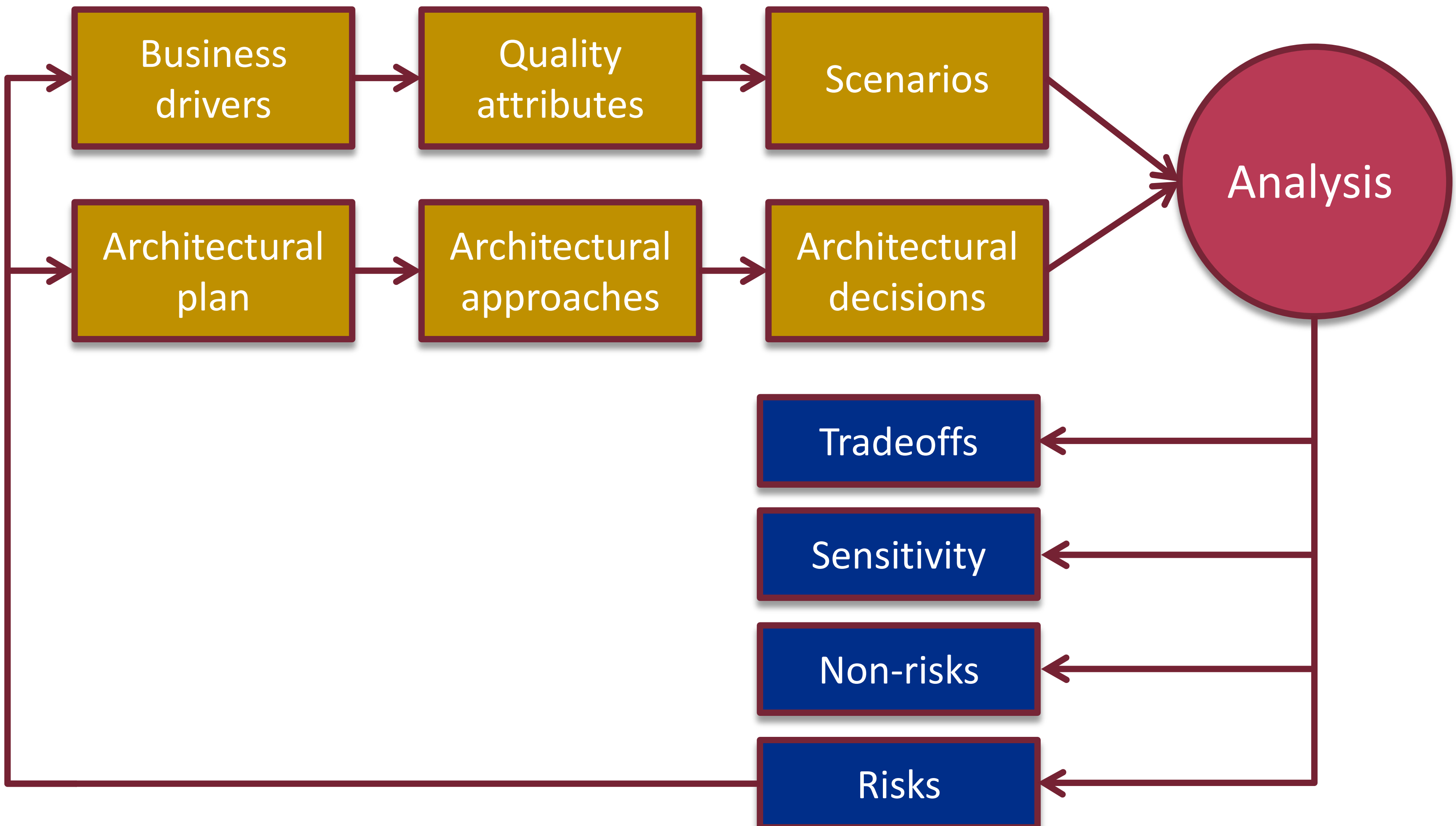
REQUIREMENTS BASED ARCHITECTURE ANALYSIS

Architecture Trade-off Analysis Method (ATAM)

Requirements based architecture analysis

- Architecture Tradeoff Analysis Method (ATAM)
  - What are the quality objectives and their attributes?
    - What are the relations and priorities of the quality objectives?
  - How does the architecture satisfy the quality objectives?
    - Do the architecture level design decisions support the quality objectives and their priorities? What are the risks?
- Basic ideas
  - Systematic collection of quality objectives and attributes: Utility tree with priorities
  - Capturing and understanding the objectives: Scenarios (that exemplify the role of the quality attribute)
  - Architecture evaluation: What was the design decision, what are the related sensitivity points, tradeoffs, risks?

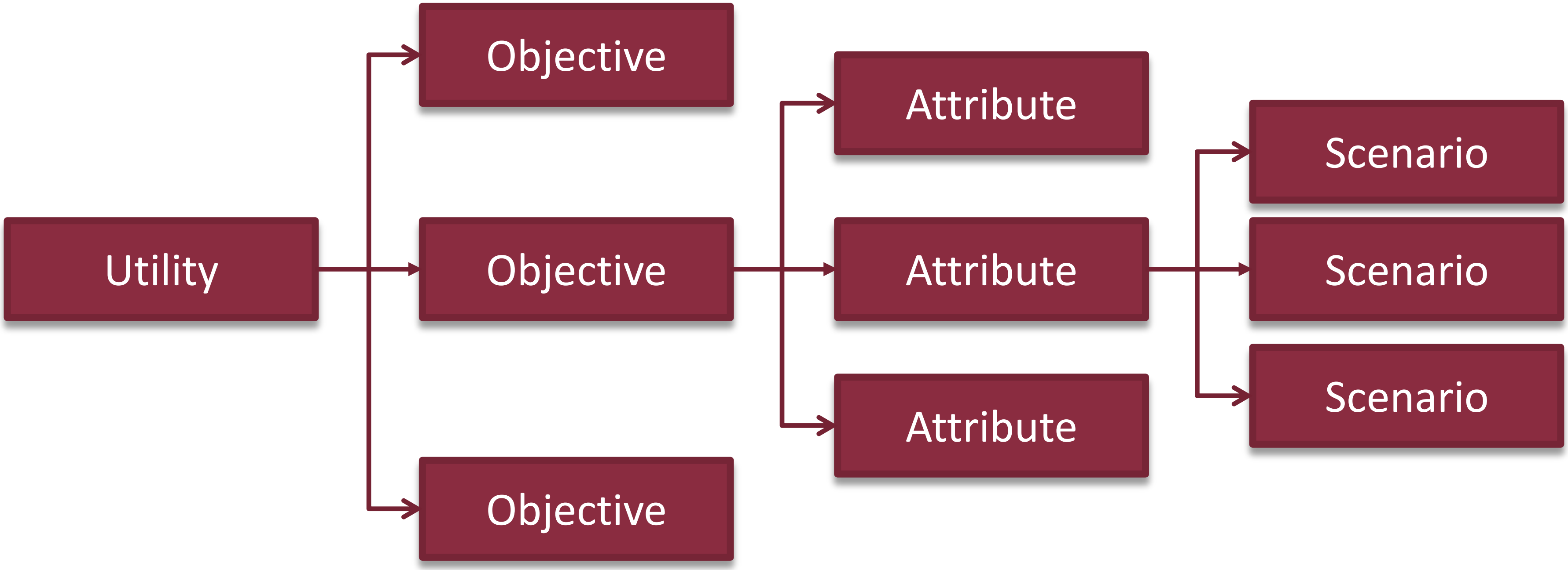
ATAM conceptual analysis process



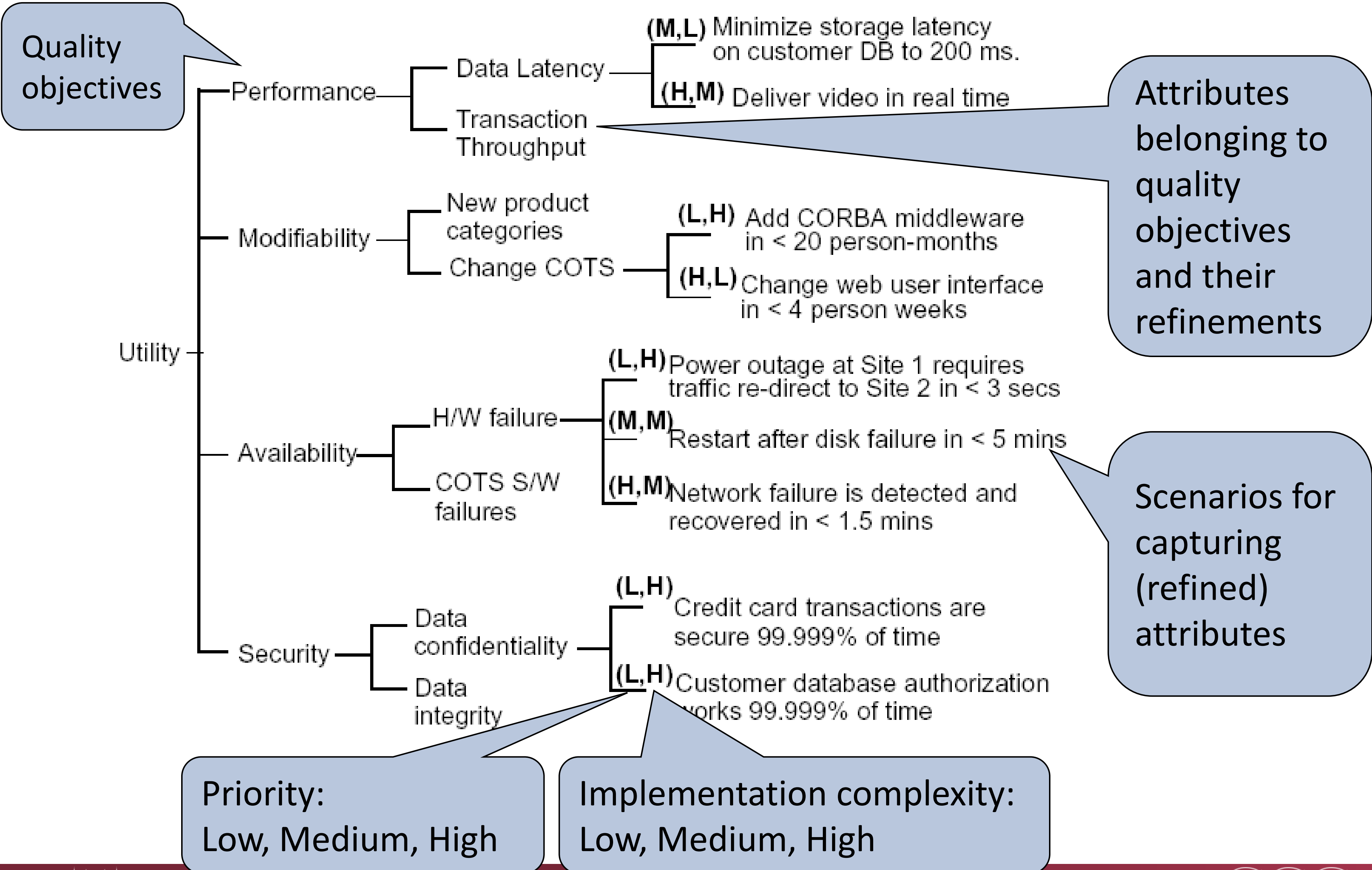
http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm

Collection of quality objectives: Utility tree structure

- Utility divided to quality objectives
- Quality objectives are characterized by attributes
- Attributes are exemplified by scenarios



Collection of quality objectives: Utility tree



Steps of the analysis (with examples)

1. Analysis of the architectural support for the scenarios
   - Scenario: Recovery in case of disk failure shall be performed in < 5 min
   - Reaction as design decision: Replica database is used
2. Analysis of sensitivity points
   - The use of replica database influences availability
   - The use of replica database influences also performance
     - Synchronous updating of the replica database: Slow
     - Asynchronous updating of the replica database: Faster, but potential data loss
3. Analysis and optimization of the tradeoffs
   - The use of replica database influences both availability and performance – influence depends on the updating strategy
     - Tradeoff (architecture decision): Asynchronous updating of the replica database
4. Analysis of the risks of tradeoffs
   - Replica database with asynchronous updating (as an architecture design decision) is a risk, if the cost of data loss is high
     - The decision is optimal only in case of given needs and cost constraints

The process of ATAM 1/2

1. Presentation of the method <- evaluation leader
2. Presentation of business drivers <- development leader
   - Functions, quality objectives, stakeholders
   - Constraints: technical, economical, management
3. Presentation of the architecture <- designers
4. Identification of the design decisions <- designers
5. Construction of the utility tree <- designers, verifiers
   - Refinement of quality objectives
   - Assignment of scenarios to capture objectives:
     - Inputs, effects that are relevant to the quality objective
     - Environment (e.g., design-time or run-time)
     - Expected reaction (support) from the architecture
   - Assignment of priorities to the scenarios (objectives)

The process of ATAM 2/2

6. Analysis of the architecture <- verifiers
   - Architectural support
   - Sensitivity points
   - Tradeoffs
   - Risks
7. Extending the scenarios <- stakeholders
   - Contribution of testers, users, etc.
   - Brainstorming: Aspects of testability, maintenance, ergonomics, etc.
   - Assignment of priorities
8. Continuing the architecture analysis <- verifiers
   - In case of scenarios with priorities that are high enough
9. Presentation of results <- verifiers
   - Preparation of a summary document

Advantages of ATAM

- Explicit and clarified quality objectives
  - Refinement of objectives, assignment of scenarios
  - Assignment of priorities
- Early identification of risks
  - Explicit analysis of the effects of architecture design decisions (model based analysis may be used)
  - Investigation of tradeoffs
- Stakeholders are involved
  - Designer, tester, user, verifier
  - Communication among the stakeholders
- Documenting architecture related decisions and risks

INTERFACE ANALYSIS

Checking conformance of interfaces

Interface analysis

- Goals
  - Checking the conformance of component interfaces
  - Completeness: Systematic coverage of relations and interfaces
- Syntactic analysis
  - Checking function signatures (number and types of parameters)
- Semantic analysis
  - Based on the description of the functionality of the components
  - Analysis of contracts (contract based specifications)
- Behavioral analysis
  - Based on the behavior specification of components
  - Behavioral conformance is checked (e.g., in case of protocols)
  - Precise behavioral equivalence relations are defined (e.g., bisimulation), also timing can be checked

Example: Specification of contracts

- ”Contract based” specification of component functionality: JML

```
public class Purse {
    final int MAX_BALANCE;
    int balance;
      /*@ invariant pin != null && pin.length == 4  @*/
    byte[] pin;
      /*@ requires amount >= 0;
        @ assignable balance;
        @ ensures balance == \old(balance) – amount
                  && \result == balance;
        @ signals (PurseException) balance == \old(balance);
        @*/
    int debit(int amount) throws PurseException {
      if (amount <= balance) {
        balance -= amount;
        System.out.println("Debit placed"); return balance; }
      else {
        throw new PurseException("overdrawn by " + amount); }}
```

- Matching interfaces on the basis of contacts (requires – ensures)

RULE BASED CHECKING OF THE ARCHITECTURE

Checking dependencies, containment, inheritance

Checking architecture related rules

- Goals
  - Verifying the architecture using models or code
  - Checking rules for correct architecture
- Examples of rules
  - Allowed dependencies between packages and classes
  - Avoiding cyclic dependencies
  - Access constraints between layers in the architecture
- Tool example: ArchUnit
  - Focus: Automatically test architecture and coding rules using any plain Java unit testing framework (e.g. JUnit)

Example: Using rules in ArchUnit

- Importing application classes to check

```
JavaClasses classes =
    new ClassFileImporter().importPackages("com.mycompany.myapp");
```

- Definition of rules using abstract DSL-like fluent API
  - Example: Services should only be accessed by Controllers

```
ArchRule myRule = classes()
  .that().resideInAPackage("..service..")
  .should().onlyBeAccessed().byAnyPackage("..controller..", "..service..");
```

- Evaluation of the rule

```
myRule.check(classes);
```

- Checking cyclic dependency

```
slices().matching("com.mycompany.myapp.(*)..").should().beFreeOfCycles()
```

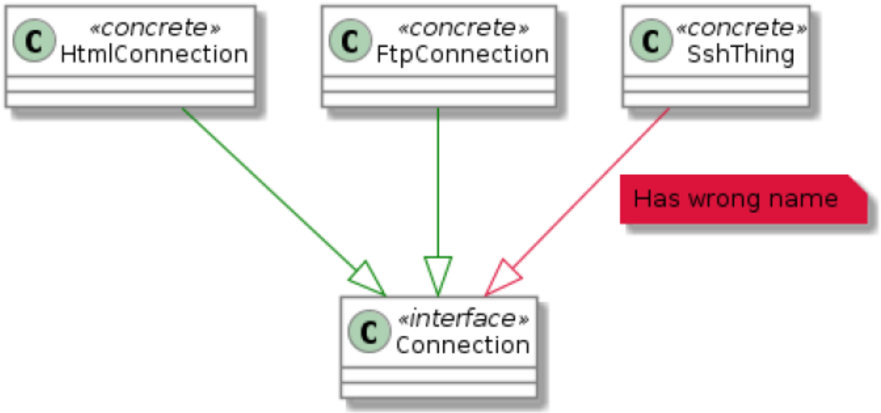
Source: ArchUnit User Guide, https://www.archunit.org/userguide/html/000_Index.html

Example architecture rules

- Package and class dependency check:


```
noClasses().that().resideInAPackage("..source..")
    .should().dependOnClassesThat().resideInAPackage("..foo..")
```

- Inheritance check:




```
classes().that().implement(Connection.class)
    .should().haveSimpleNameEndingWith("Connection")
```

Source: ArchUnit User Guide, https://www.archunit.org/userguide/html/000_Index.html

FAULT EFFECTS ANALYSIS

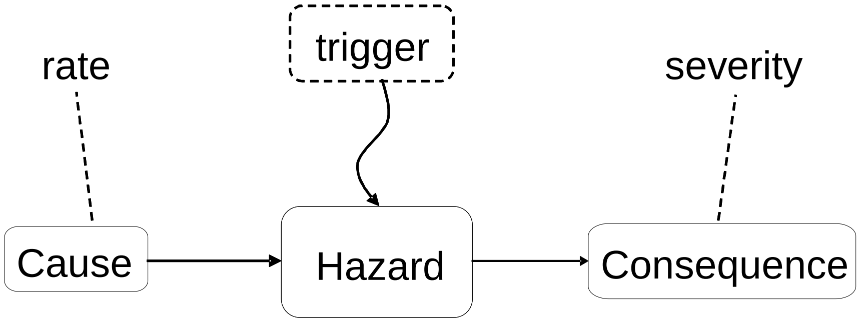
Fault Tree, Event Tree, Failure Modes and Effects Analysis

(see also: IT System Design course)

Analysis of fault effects

- Goal: Analysis of the fault effects and the evolution of hazards on the basis of the architecture
  - What are the causes for a hazard?
  - What are the effects of a component fault?
- Results:
  - Hazard catalogue
  - Categorization of hazards
    - Rate of occurrence
    - Severity of consequences

    → Risk matrix
  - These results form the basis for risk reduction



Categorization of the techniques

- Cause-consequence view:
  - Forward (inductive): Analysis of the effects of faults and events
  - Backward (deductive): Analysis of the causes of hazards
- System hierarchy view:
  - Bottom-up: From the components to subsystems / system level
  - Top-down: From the system level down to the components

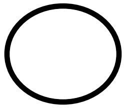
Systematic techniques are needed

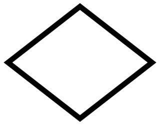
Fault tree analysis

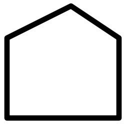
- Analysis of the causes of system level hazards
  - Top-down analysis
  - Identifying the combinations of component level faults and events that may lead to hazard
- Construction of the fault tree
  1. Identification of the foreseen system level hazard: on the basis of environment risks, standards, etc.
  2. Identification of intermediate events (pseudo-events): Boolean (AND, OR) combinations of lower level events that may cause upper level events
  3. Identification of primary (basic) events: no further refinement is needed/possible

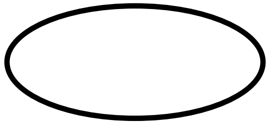
Set of elements in a fault tree

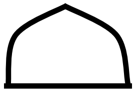

Top level or intermediate event

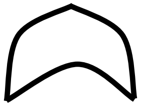

Primary (basic) event


Event without further analysis

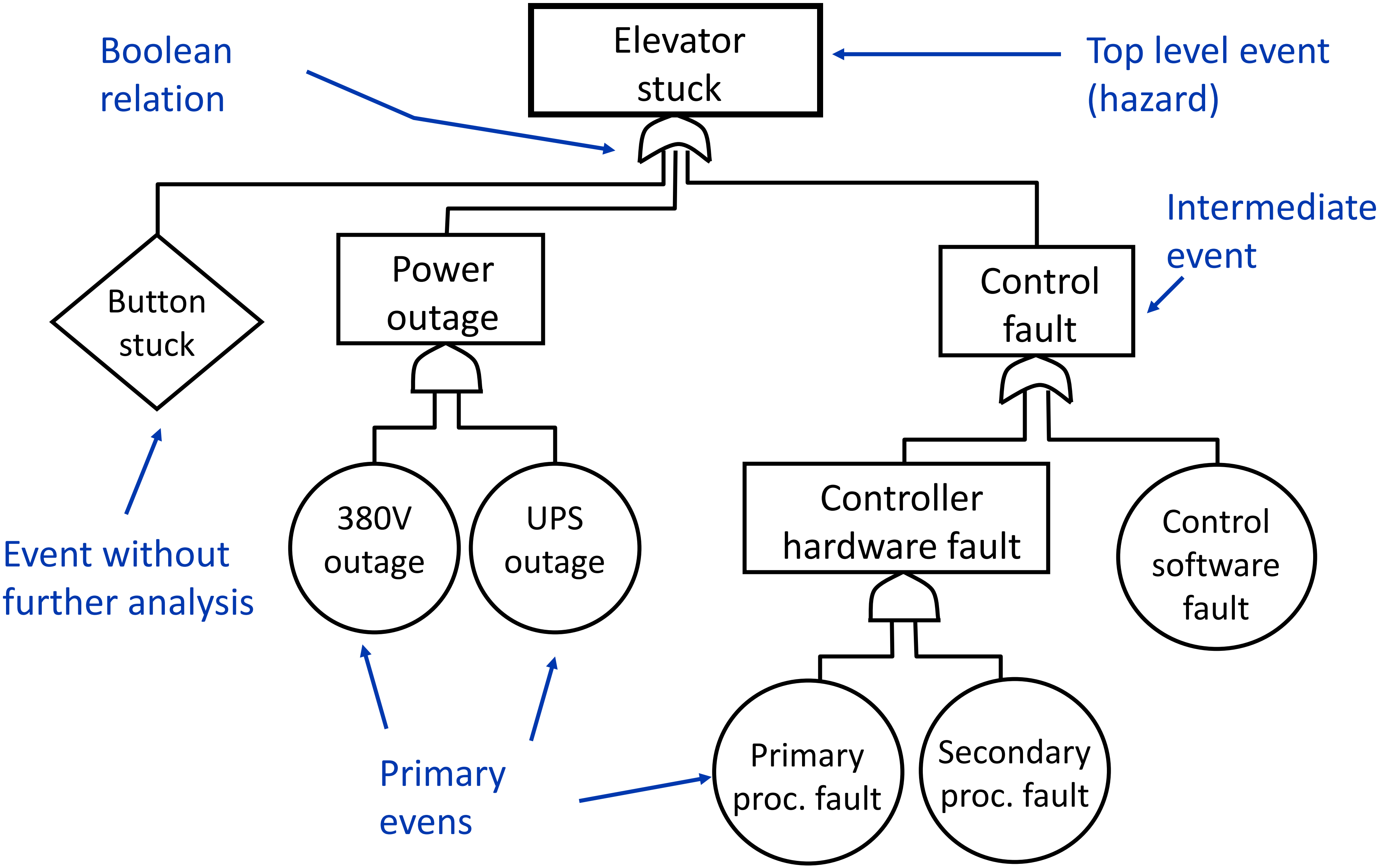

Normal event (i.e., not a fault)


Conditional event


AND combination of events


OR combination of events

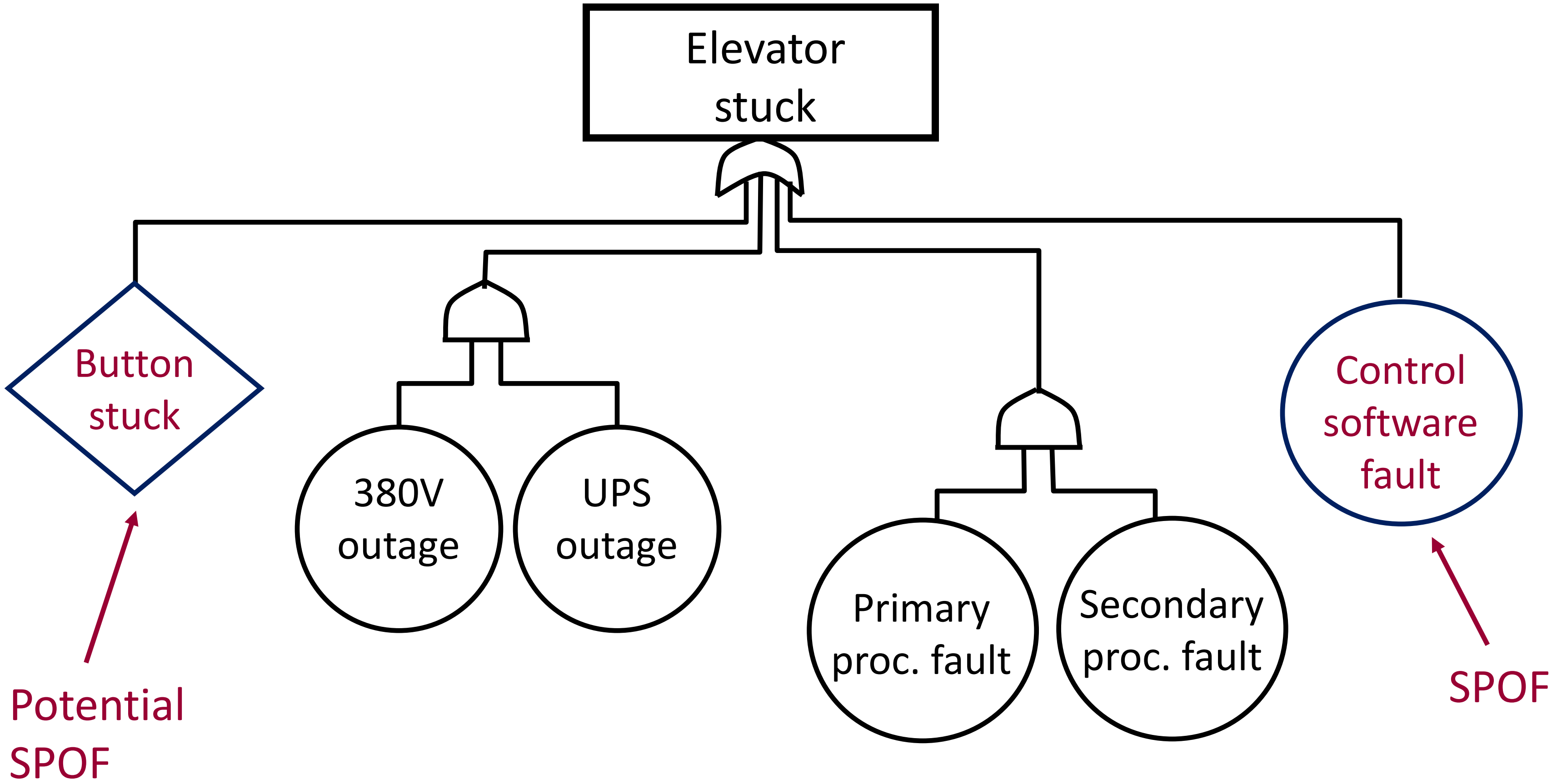
Fault tree example: Elevator



Qualitative analysis of the fault tree

- Fault tree reduction: Resolving intermediate events/pseudo-events using primary events $\rightarrow$ disjunctive normal form (OR on the top of the tree)
- Cut of the fault tree:
  AND combination of primary events
- Minimal cut set: No further reduction is possible
  - There is no cut that is a subset of another
- Outputs of the analysis of the reduced fault tree:
  - Single point of failure (SPOF)
  - Events that appear in several cuts

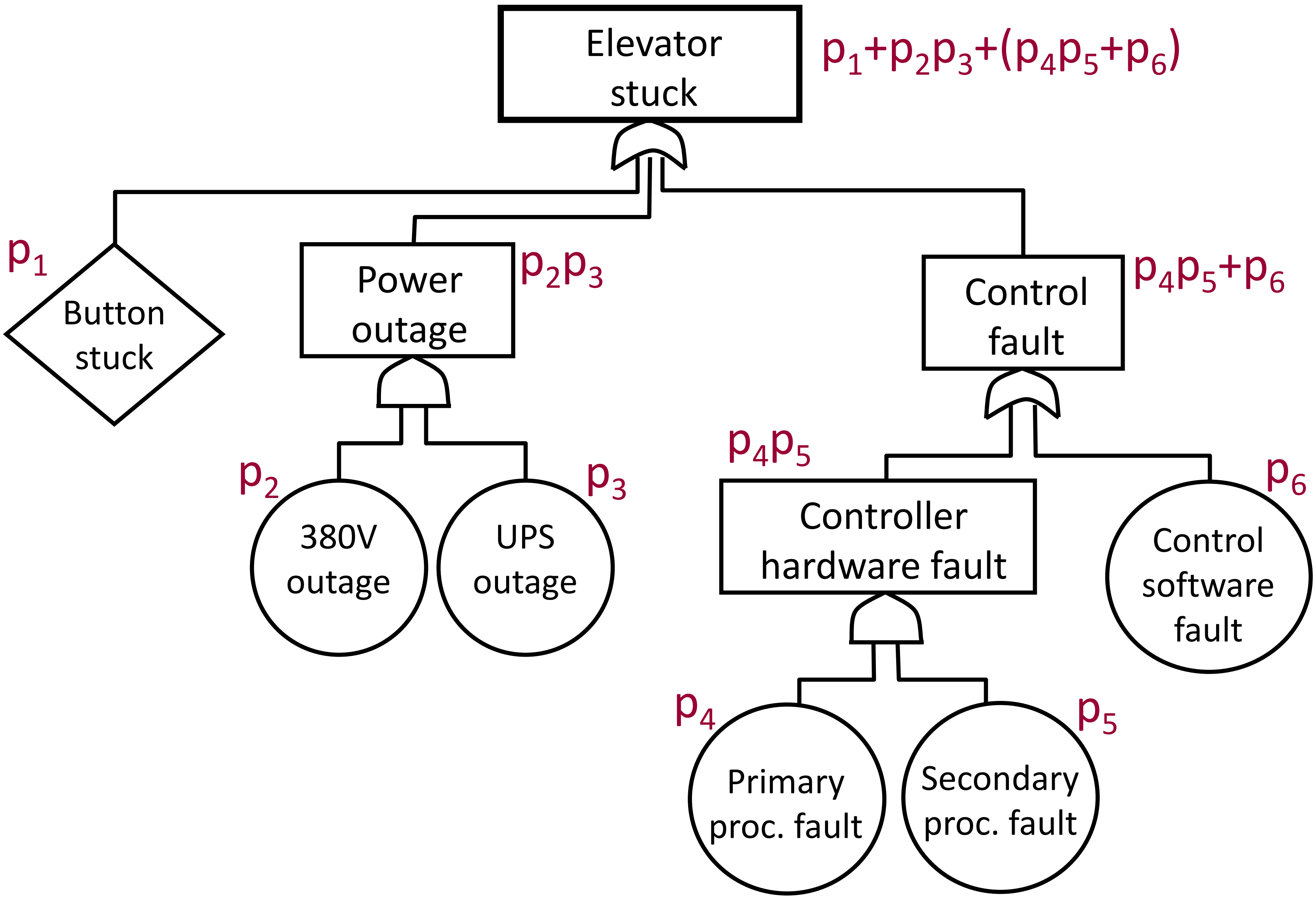
Reduced fault tree of the elevator example



Quantitative analysis of the fault tree

- Basis: Probabilities of the primary events
  - Component level data, experience, or estimation
- Result: Probability of the system level hazard
  - Computing probability on the basis of the probabilities of the primary events, depending on their combinations
  - AND gate: Product (if the events are independent)
    - Exact calculation: P{A and B} = P{A} · P{B|A}
  - OR gate: Sum (worst case estimation)
    - Exactly: P{A or B} = P{A} + P{B} - P{A and B} <= P{A} + P{B}
- Limitations of the analysis
  - Correlated faults (not independent)
  - Representation of fault sequences

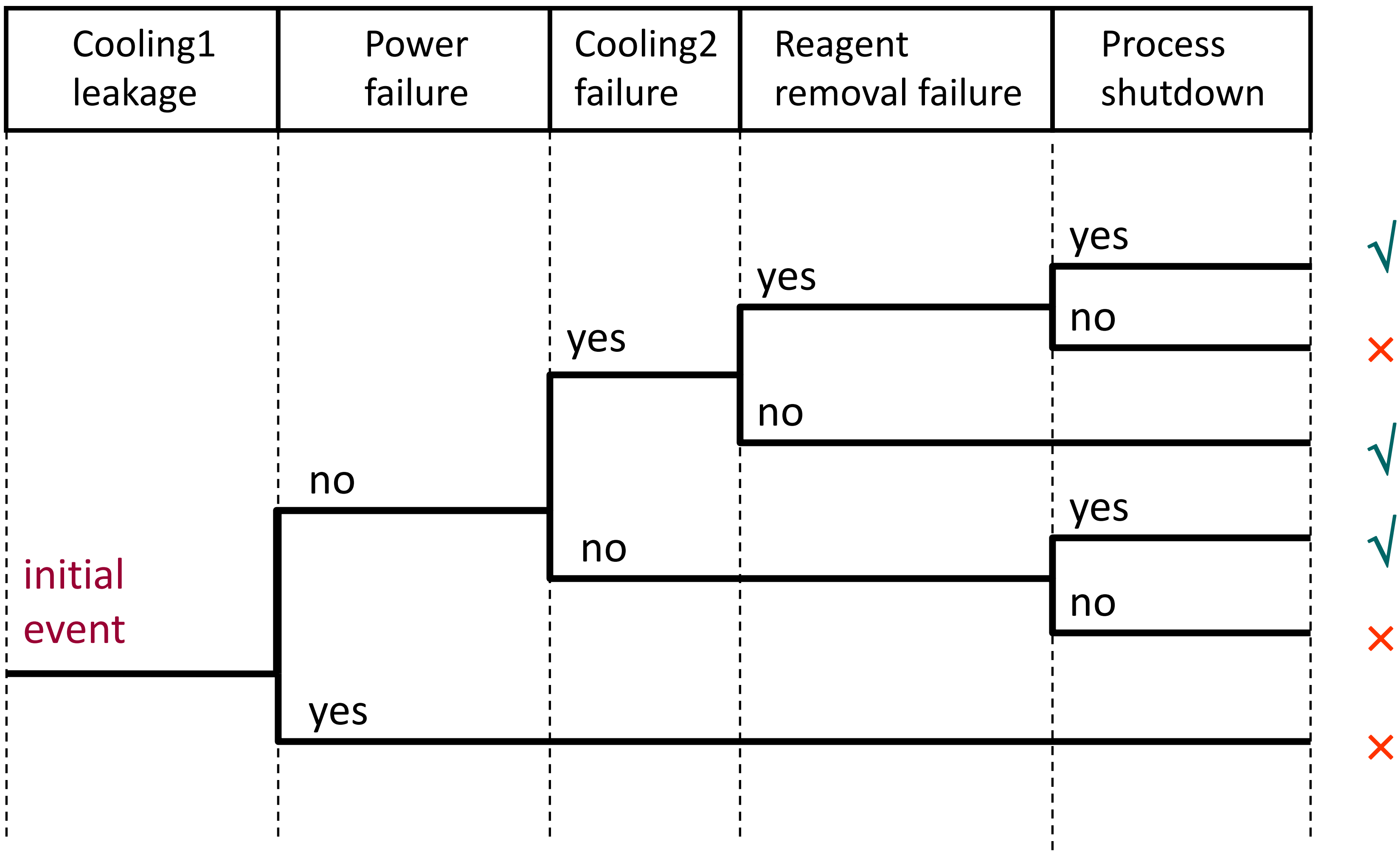
Fault tree of the elevator with probabilities



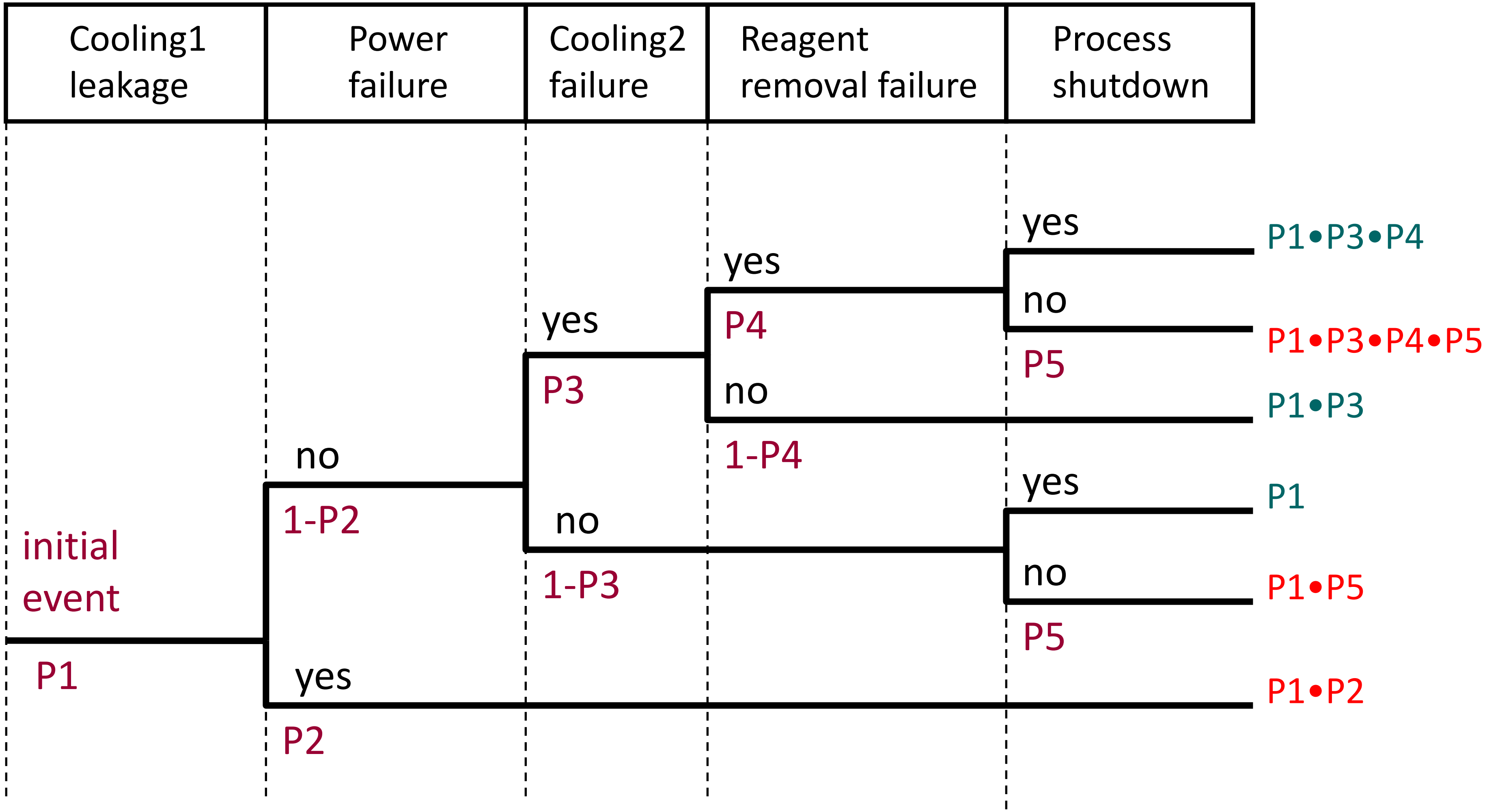
Event tree analysis

- Forward (inductive) analysis:
  Investigates the effects of an initial event (trigger)
  - Initial event: component level fault/event
  - Related events: faults/events of other components
  - Ordering: causality, timing
  - Branches: depend on the occurrence of events
- Investigation of hazard occurrence „scenarios”
  - Path probabilities (on the basis of branch probabilities)
- Advantages: Investigation of event sequences
  - Example: Checking protection systems (protection levels)
- Limitations of the analysis
  - Complexity, multiplicity of events

Event tree example: Reactor cooling



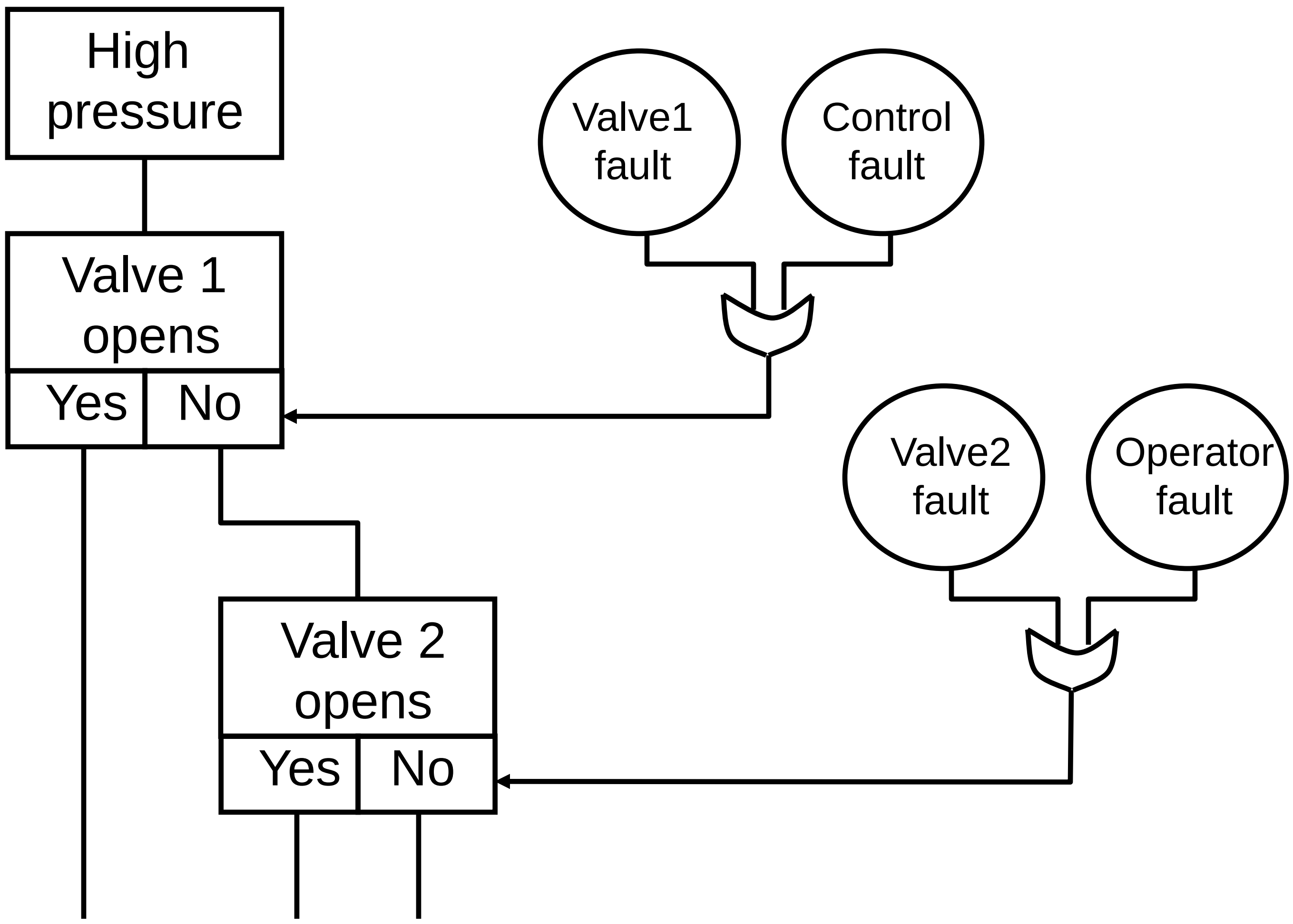
Event tree example: Reactor cooling



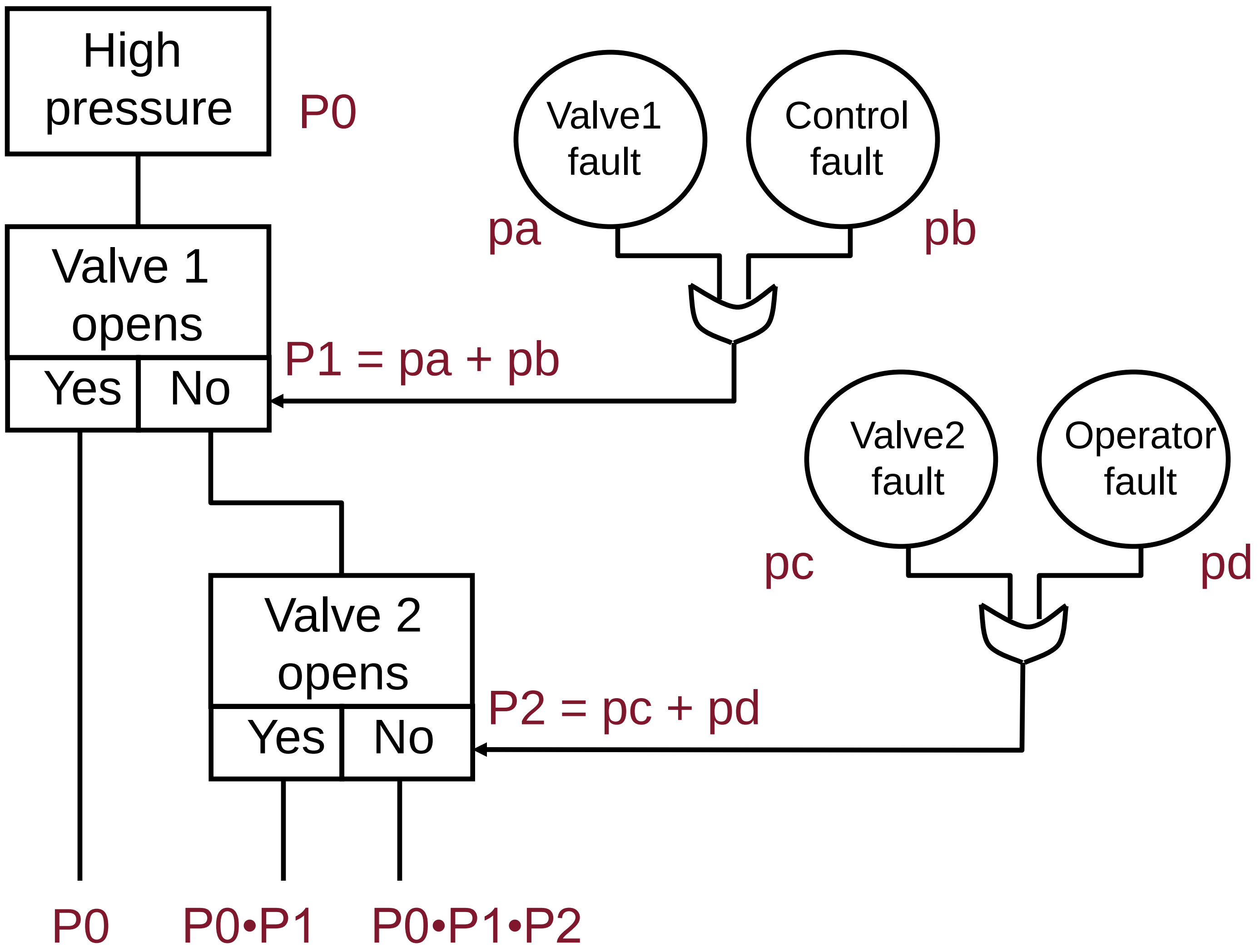
Cause-consequence analysis

- Connecting event tree with fault trees
  - Event tree: Scenarios (sequence of events)
  - Connected fault trees: Analysis of event occurrence, computing the probability of occurrence
- Advantages:
  - Sequence of events (forward analysis) together with analysis of event causes (backward analysis)
- Disadvantages:
  - Complexity: Separate diagrams are needed for all initial events

Example for cause-consequence analysis



Example for cause-consequence analysis



Failure Modes and Effects Analysis (FMEA)

- Tabular representation and analysis of components, failure modes, probabilities (occurrence rates) and effects
- Advantages:
  - Systematic listing of components and failure modes
  - Analysis of redundancy
- Limitations of the analysis
  - Complexity of determining the fault effects (using simulators, analysis models, symbolic execution etc.)

Component	Failure mode	Probability	Effect
Temperature limit L detector function	> L not detected ≤ L detected	65% 35%	Over-heating Process is stopped
…	…	…	…

MODEL BASED QUANTITATIVE EVALUATION

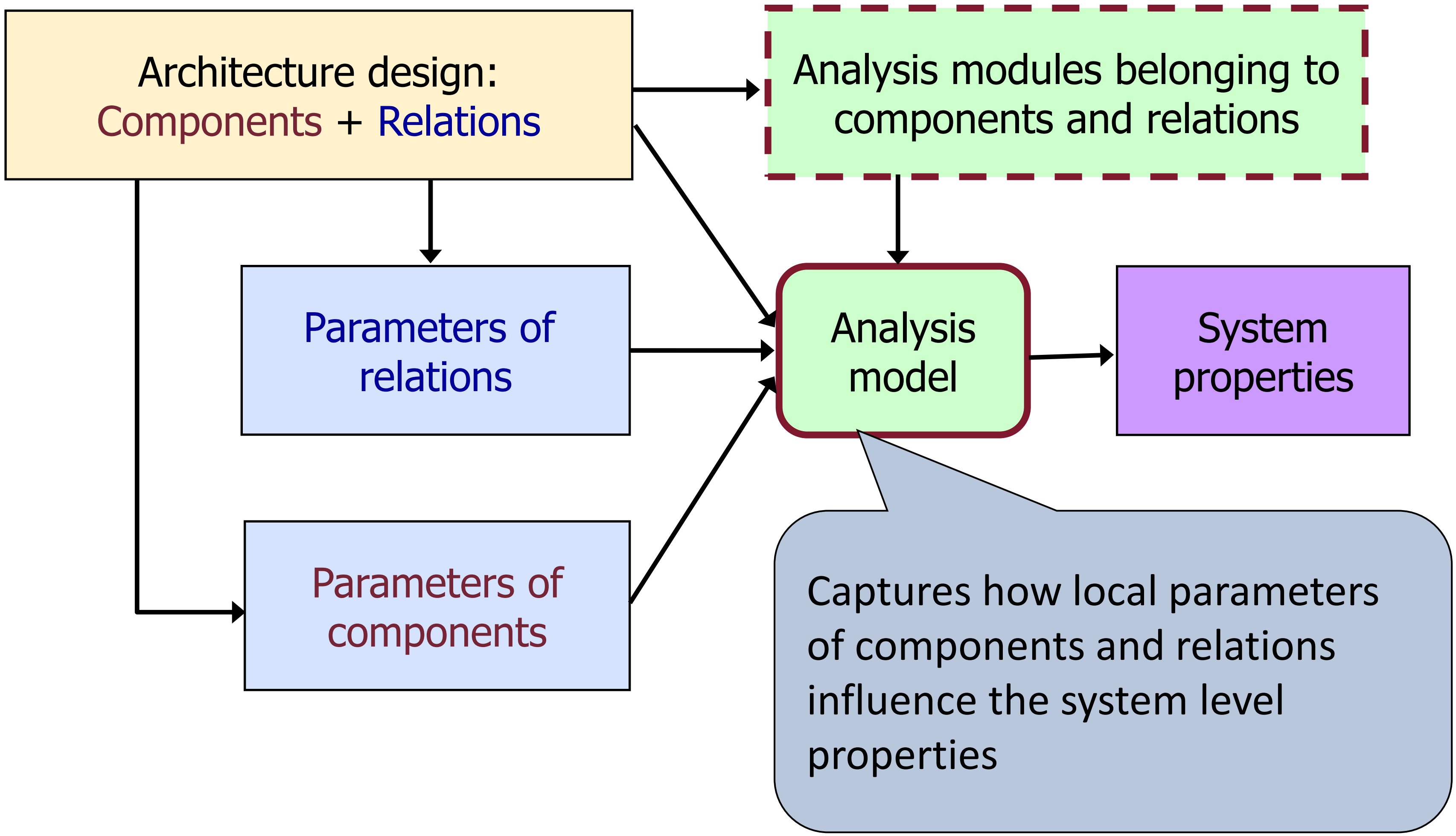
Model based performance evaluation

Model based quantitative evaluation

Goal: Evaluation of architecture solutions

- Analysis models are constructed and solved on the basis of the architecture model, e.g.
  - Performance model
  - Dependability model
  - Safety analysis model
- Analysis models are mathematical models
  - Capture how local parameters of components and relations influence system level properties
  - The solution of the model (= computation of selected model characteristics) provide system level properties
- Modular construction of analysis models (possibly automated)
  - Architecture: Component and relations
  - Analysis model: Submodels (modules) for components and relations

General approach for model based evaluation



Typical analysis models

	Performance model	Dependability model
Component parameters	Local execution time of functions, priorities, scheduling	Fault occurrence rate, error delay, repair rate, error detection coverage, …
Relation parameters	Call forwarding rate, call synchronization	Error propagation probability, conditions or error propagation, repair strategy
Model	Queuing network	Markov-chain, Petri-net
System properties (computed)	Request handling time, throughput, processor utilization	Reliability, availability, MTTF, MTTR, MTBF

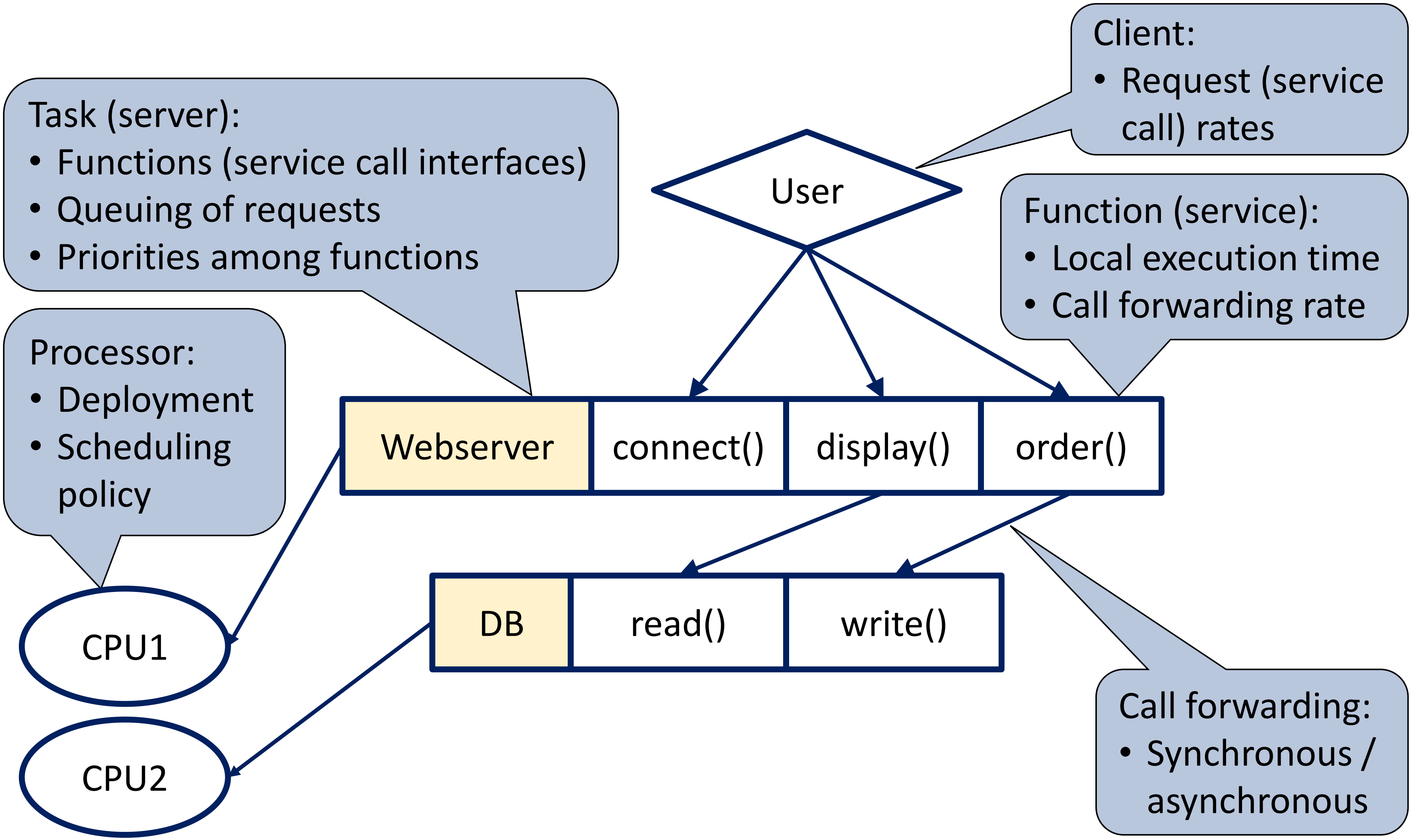
Typical analysis models

	Performance model	Dependability model
Component parameters	Local execution time of functions, priorities, scheduling	Fault occurrence rate, error delay, repair rate, error detection coverage, …
Relation parameters	Call forwarding rate, call synchronization	Error propagation probability, conditions or error propagation, repair strategy
Model	Queuing network	Markov-chain, Petri-net
System properties (computed)	Request handling time, throughput, processor utilization	Reliability, availability, MTTF, MTTR, MTBF

Performance modeling

- Typical formalisms: Queuing networks
- Example: Layered Queuing Network (LQN)
  - Suitable for distributed client-server applications
- Model elements
  - Client submitting requests to (remote) servers
  - Servers (called “tasks” by convention)
    - Queuing of incoming requests
    - Entry points for service threads (called “functions”) with priorities
    - Forwarding function calls to other servers
  - Hosts (called “processors”)

Example: Elements of an LQN model



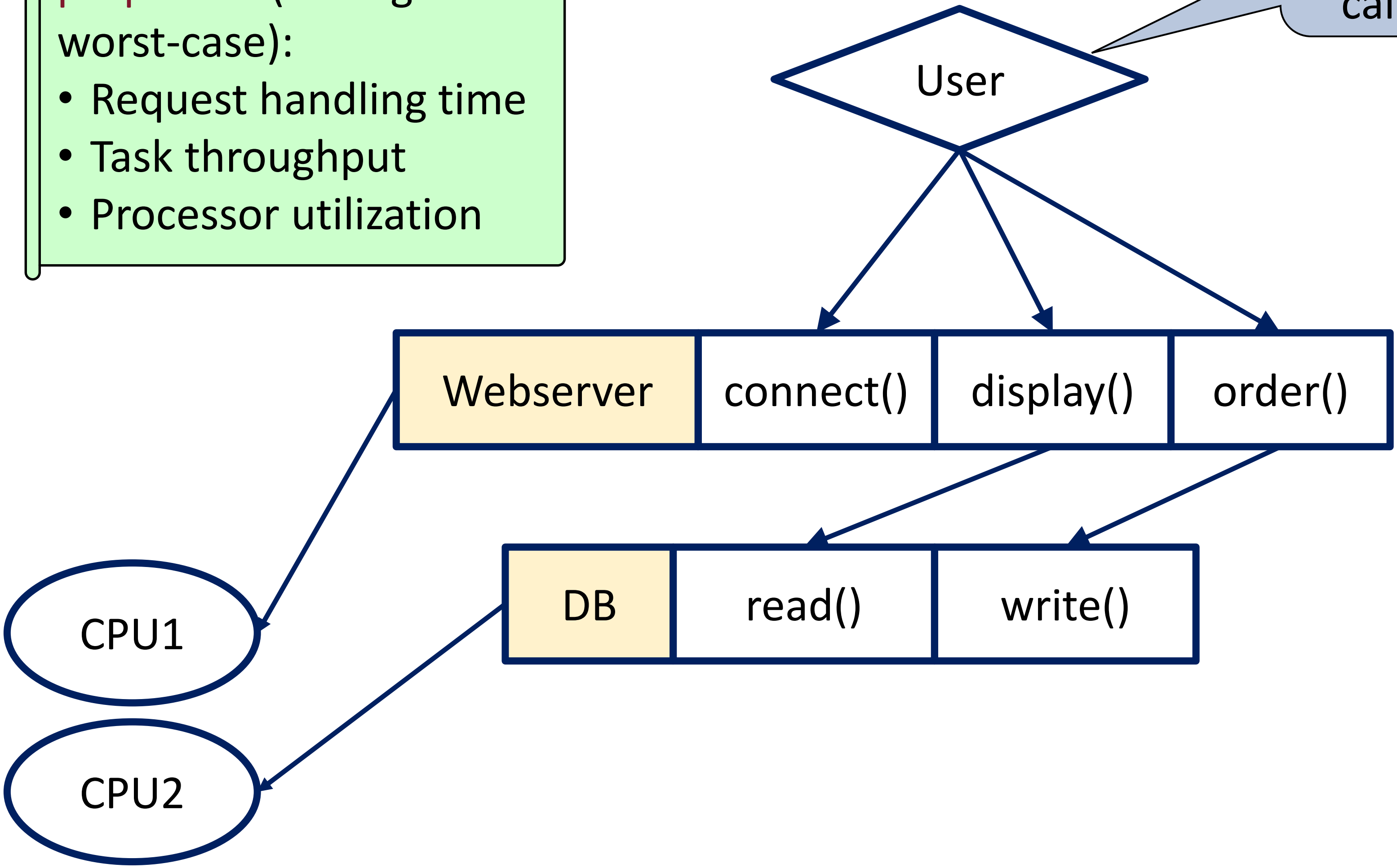
Example: Results of the analysis of an LQN model

Computed system level properties (average and worst-case):

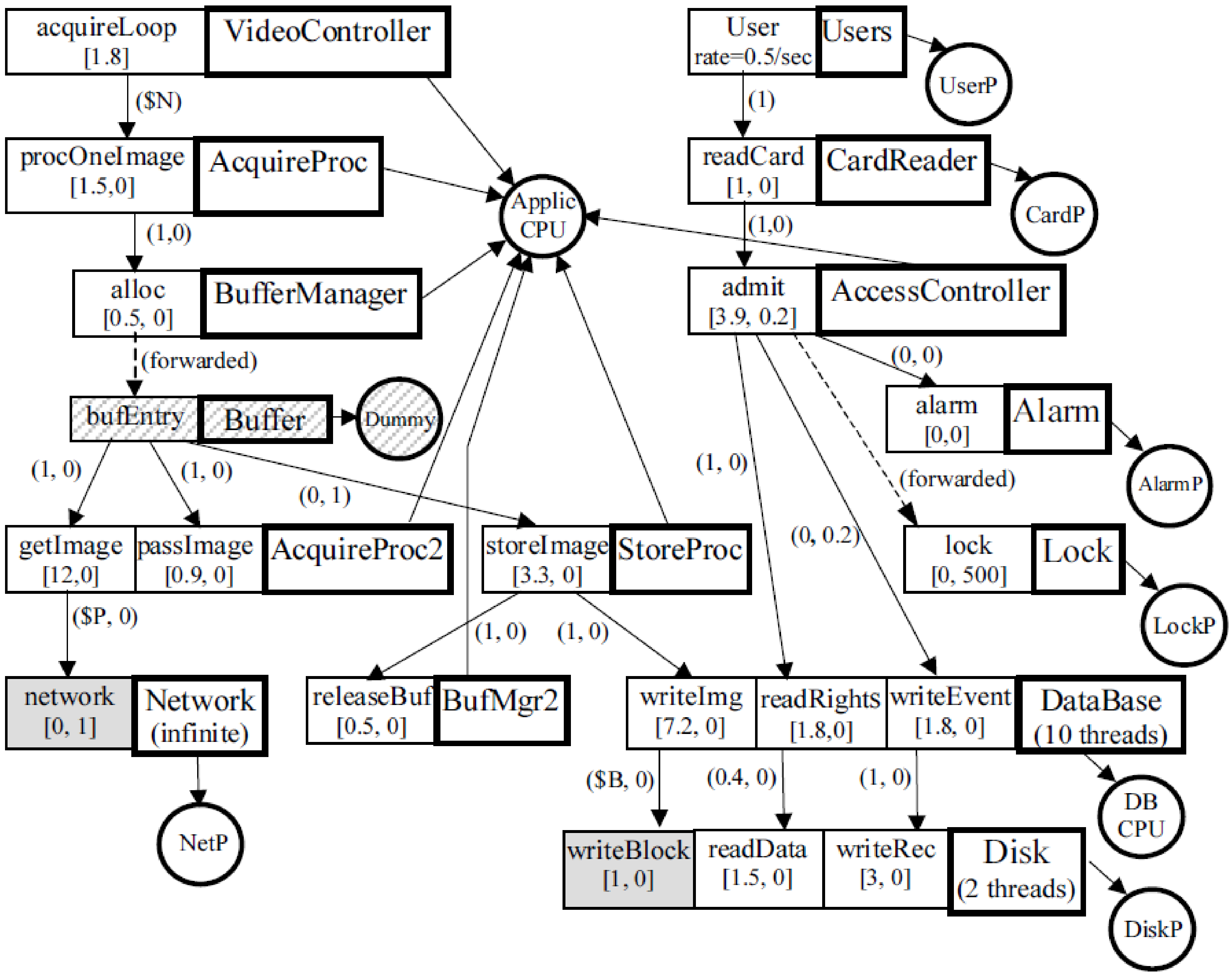
- Request handling time
- Task throughput
- Processor utilization

Client:

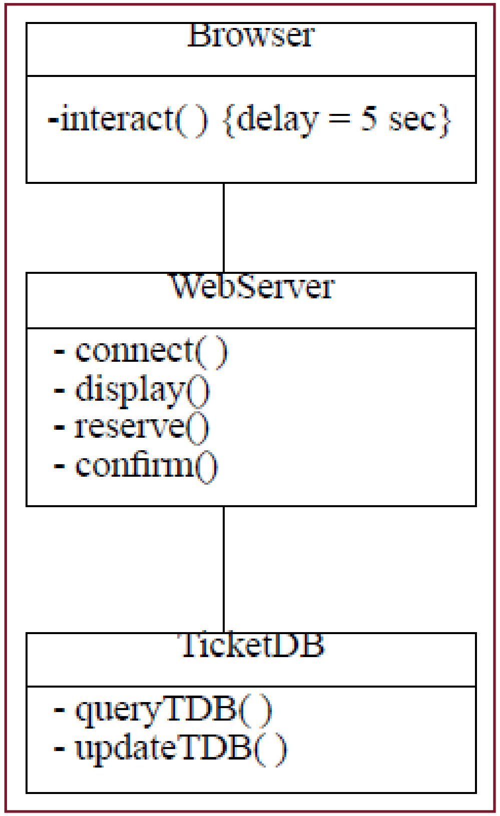
- Request (service call) rates



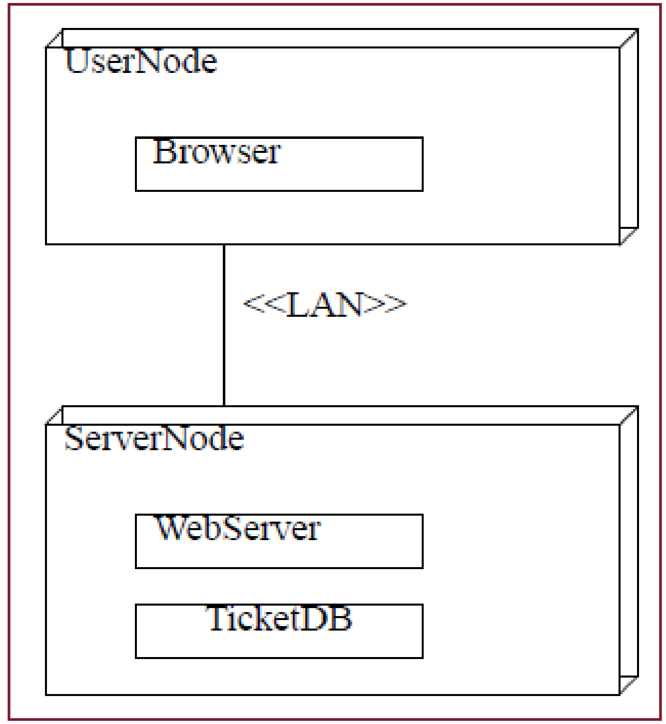
Example: Layers in complex LQN models



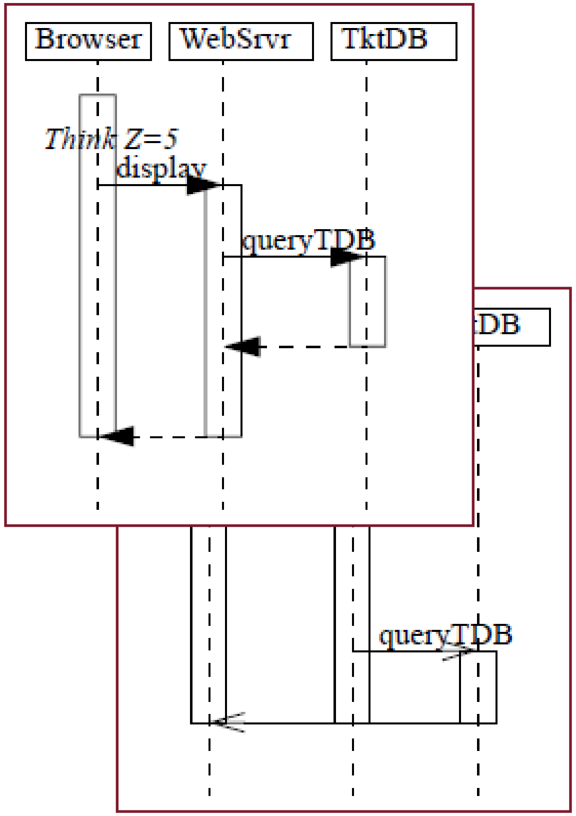
Example: Mapping architecture model to analysis model



Classes and objects
with local parameters

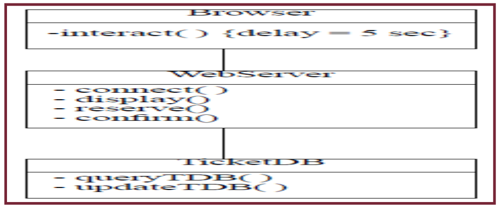


Servers and
deployment

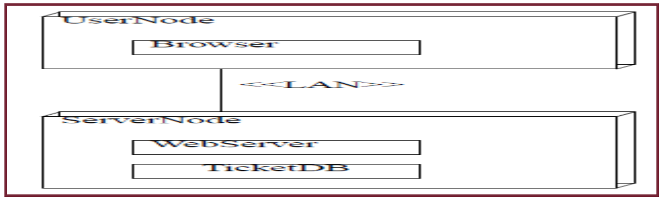


Interactions
(calls)

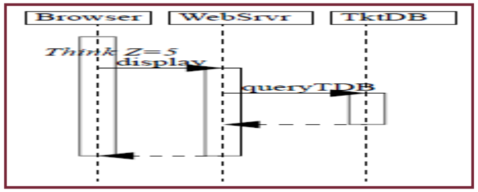
Example: Mapping architecture model to analysis model



Classes (objects)

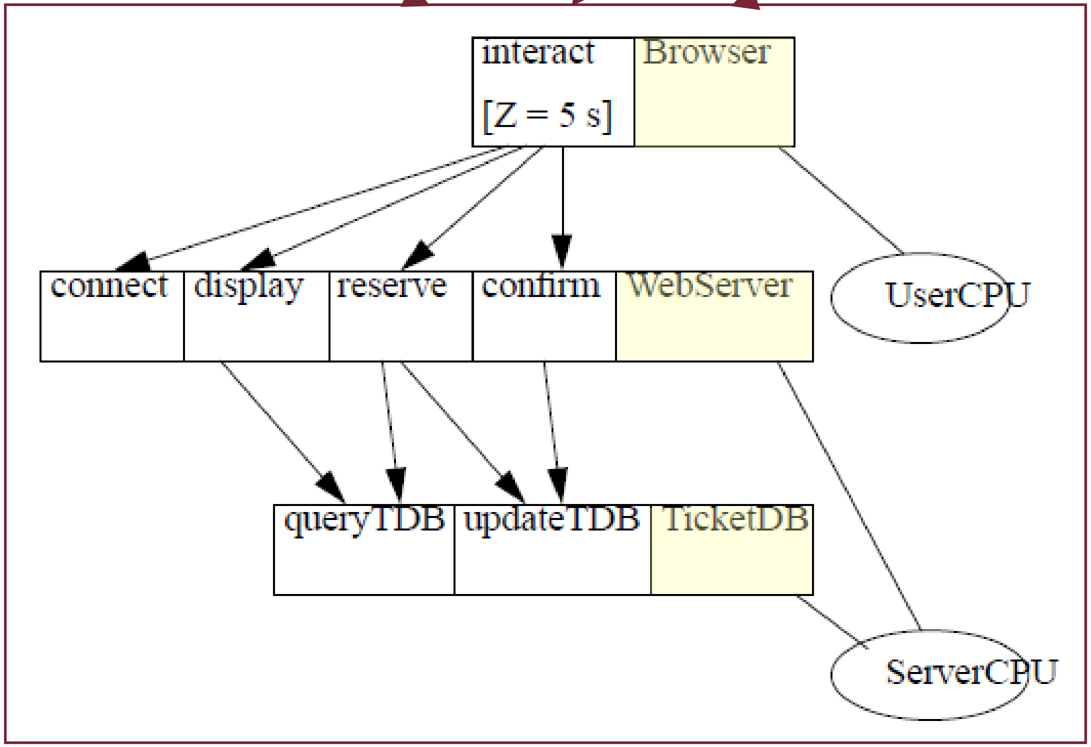


Deployment



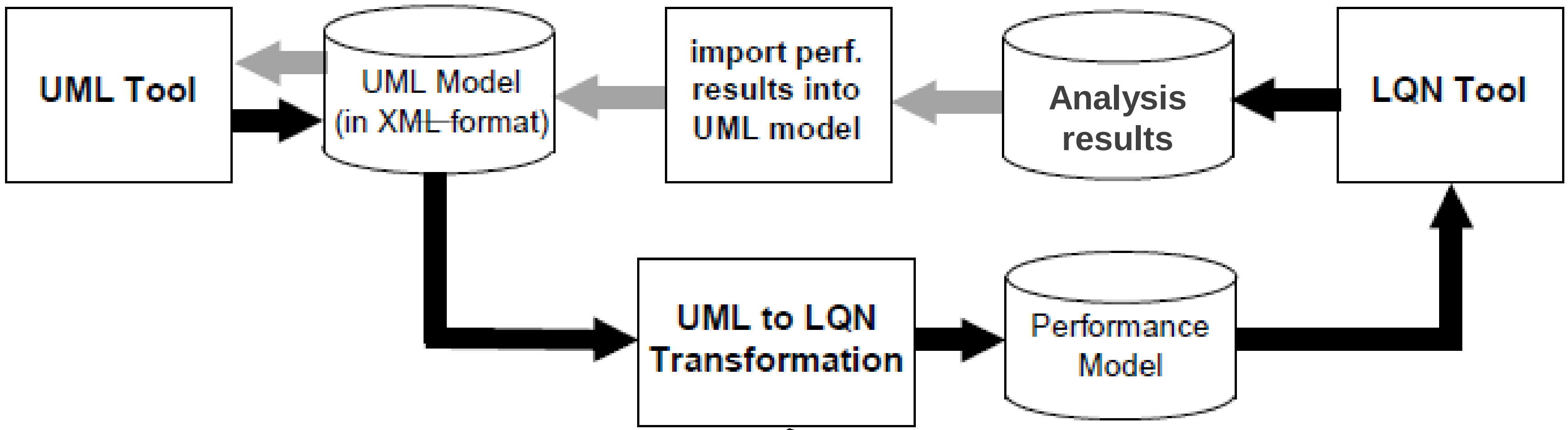
Interactions

Model transformation

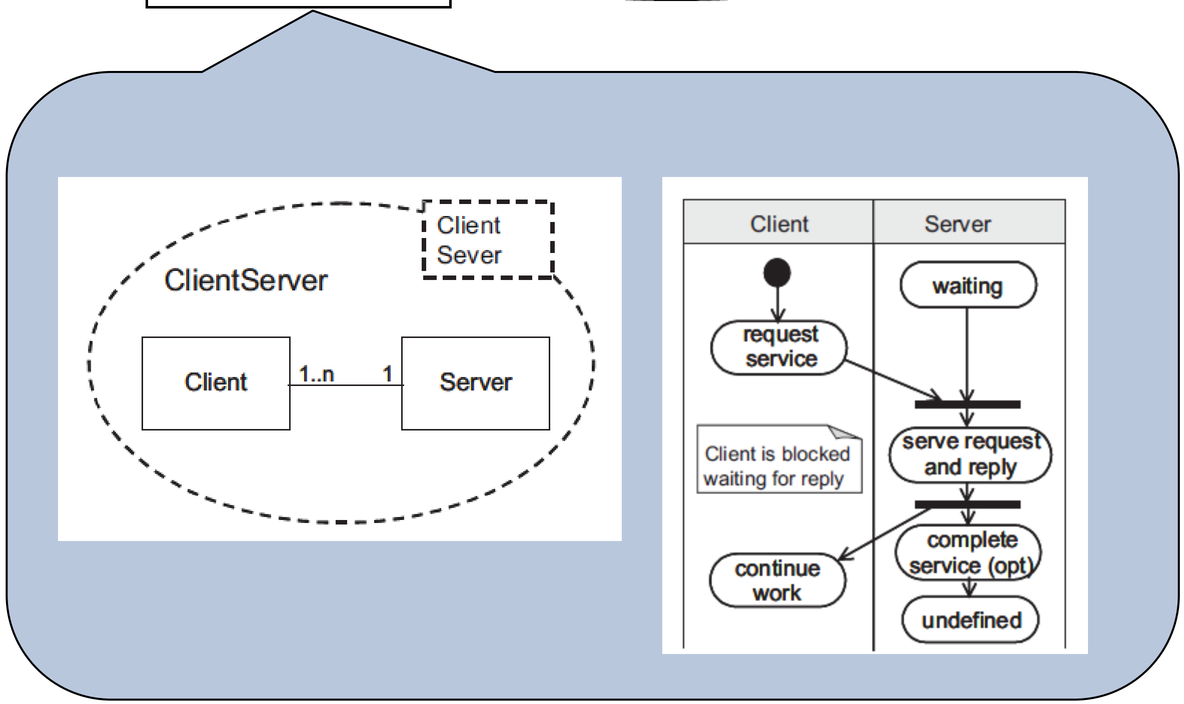


LQN performance model

Example: Mapping architecture model to analysis model



Architecture design patterns can be identified to assign analysis modules



Summary

- Motivation
  - What is determined by the architecture?
  - What kind of verification methods can be used?
- Requirements based architecture analysis
  - ATAM: Architecture Trade-off Analysis
- Systematic analysis methods
  - Interface analysis
  - Fault effects analysis
- Model based evaluation
  - Performance evaluation

  → Next lecture: Dependability modeling