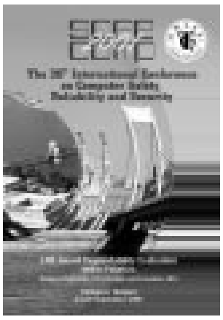
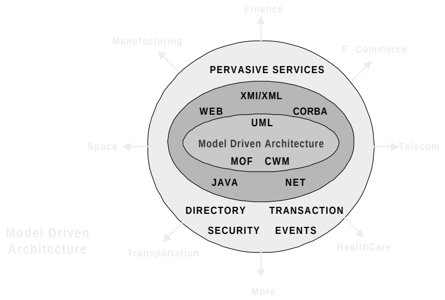


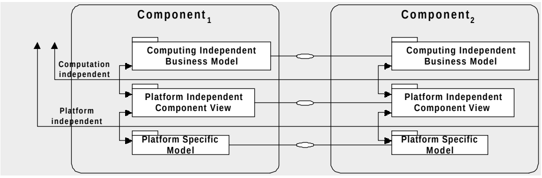
OMG Model Driven Architecture

Document: ormsc/2001-07-01
Architecture Board ORMSC 1
July 9, 2001

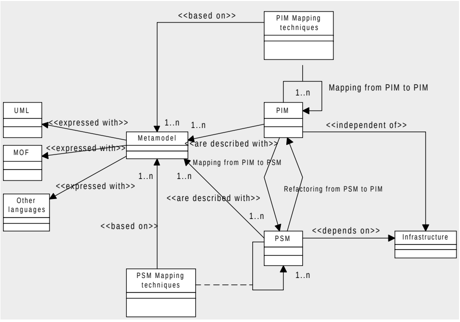
OMG's vision on architecture



Modeling views

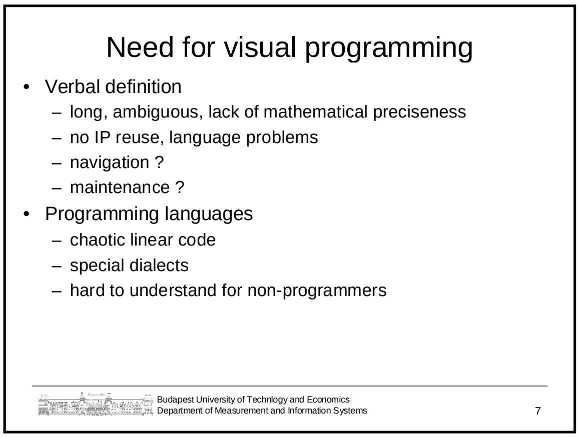

Component1
Component2
Computation independent
Platform independent
Computing Independent Business Model
Platform Independent Component View
Platform Specific Model
Computing Independent Business Model
Platform Independent Component View
Platform Specific Model

Metamodell



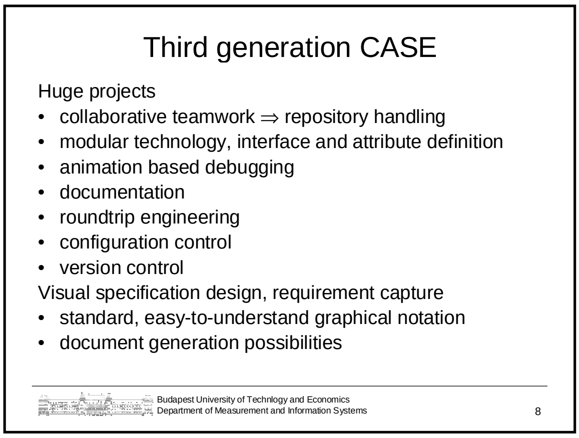
UML promises

UML 1.4 specification
www.omg.org



Need for visual programming

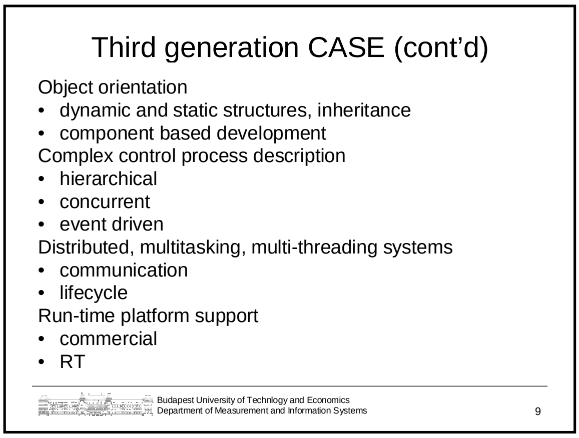
- Verbal definition
  - long, ambiguous, lack of mathematical preciseness
  - no IP reuse, language problems
  - navigation ?
  - maintenance ?
- Programming languages
  - chaotic linear code
  - special dialects
  - hard to understand for non-programmers



Third generation CASE

Huge projects
- collaborative teamwork ⇒ repository handling
- modular technology, interface and attribute definition
- animation based debugging
- documentation
- roundtrip engineering
- configuration control
- version control

Visual specification design, requirement capture
- standard, easy-to-understand graphical notation
- document generation possibilities



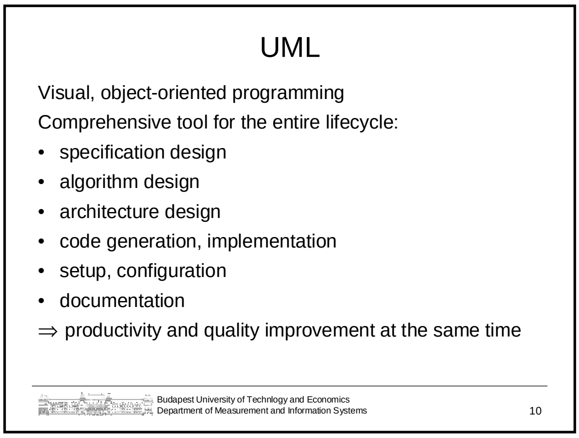
Third generation CASE (cont’d)

Object orientation
- dynamic and static structures, inheritance
- component based development

Complex control process description
- hierarchical
- concurrent
- event driven

Distributed, multitasking, multi-threading systems
- communication
- lifecycle

Run-time platform support
- commercial
- RT



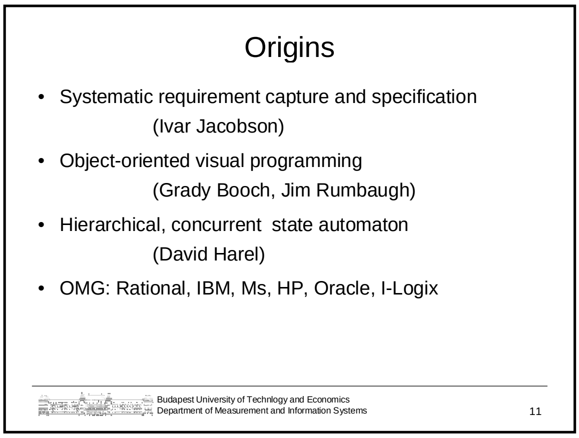
UML

Visual, object-oriented programming

Comprehensive tool for the entire lifecycle:

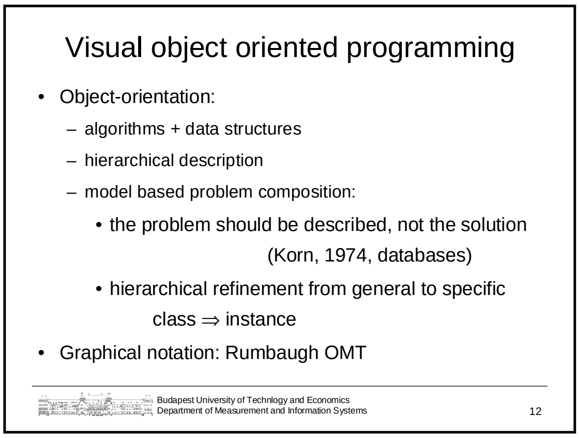
- specification design
- algorithm design
- architecture design
- code generation, implementation
- setup, configuration
- documentation

⇒ productivity and quality improvement at the same time



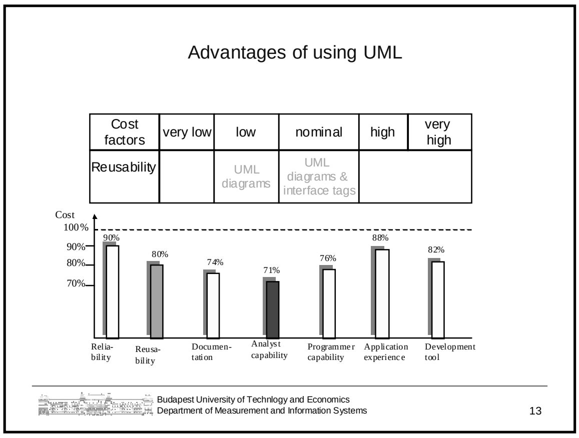
Origins

- Systematic requirement capture and specification
  (Ivar Jacobson)
- Object-oriented visual programming
  (Grady Booch, Jim Rumbaugh)
- Hierarchical, concurrent state automaton
  (David Harel)
- OMG: Rational, IBM, Ms, HP, Oracle, I-Logix



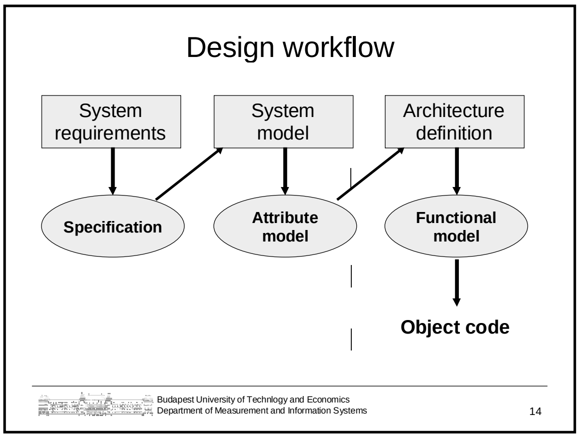
Visual object oriented programming

- Object-orientation:
  - algorithms + data structures
  - hierarchical description
  - model based problem composition:
    - the problem should be described, not the solution
      (Korn, 1974, databases)
    - hierarchical refinement from general to specific
      class ⇒ instance
- Graphical notation: Rumbaugh OMT



Advantages of using UML

Cost factors	very low	low	nominal	high	very high
Reusability		UML diagrams	UML diagrams & interface tags		



Design workflow

System requirements

System model

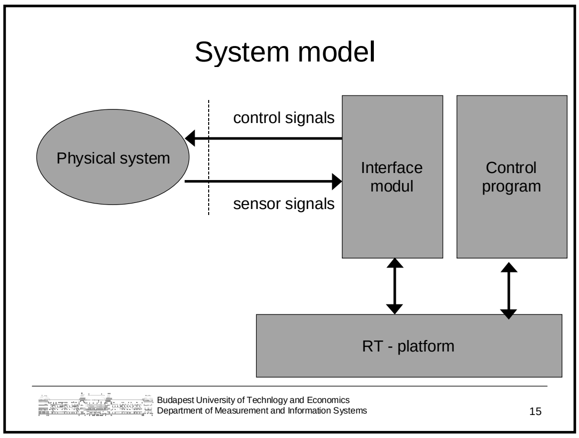
Architecture definition

Specification

Attribute model

Functional model

Object code



System model

Physical system

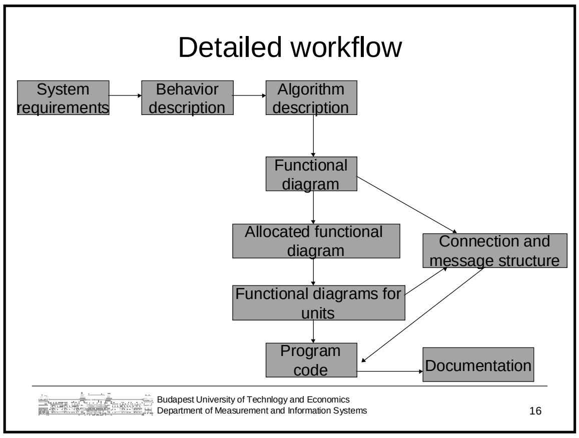
control signals

sensor signals

Interface modul

Control program

RT - platform



Detailed workflow

System requirements → Behavior description → Algorithm description

Algorithm description → Functional diagram

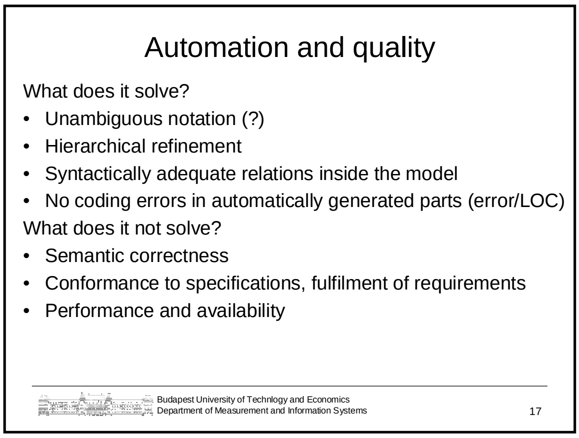
Functional diagram → Allocated functional diagram; Functional diagram → Connection and message structure

Allocated functional diagram → Functional diagrams for units

Functional diagrams for units → Program code; Functional diagrams for units → Connection and message structure

Connection and message structure → Program code

Program code → Documentation



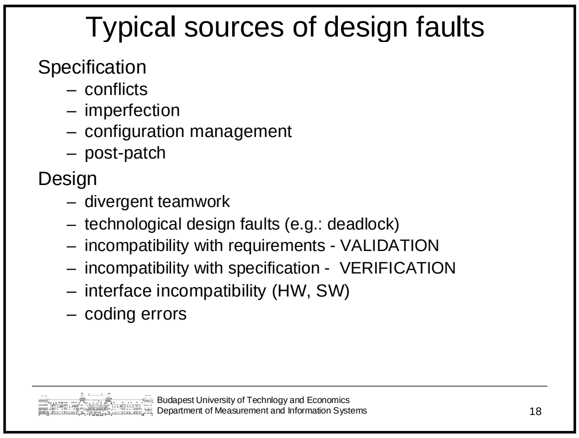
Automation and quality

What does it solve?

- Unambiguous notation (?)
- Hierarchical refinement
- Syntactically adequate relations inside the model
- No coding errors in automatically generated parts (error/LOC)

What does it not solve?

- Semantic correctness
- Conformance to specifications, fulfilment of requirements
- Performance and availability



Typical sources of design faults

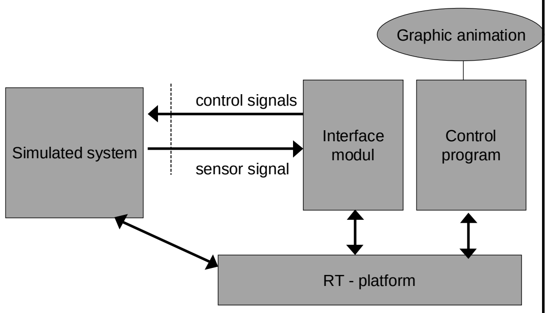
Specification

- conflicts
- imperfection
- configuration management
- post-patch

Design

- divergent teamwork
- technological design faults (e.g.: deadlock)
- incompatibility with requirements - VALIDATION
- incompatibility with specification - VERIFICATION
- interface incompatibility (HW, SW)
- coding errors

Simulation and animation based verification



Analysis of system dependability

A great number of accidents due to specification problems:
- imperfect,
- inadequate prerequisites,
- uncovered cases,

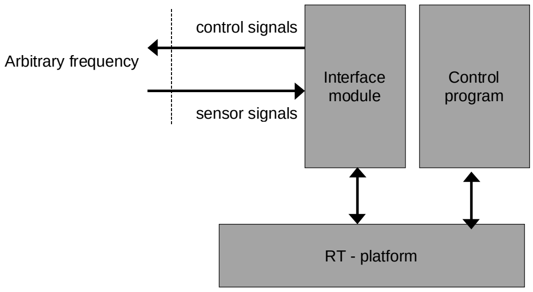
specification integrity checking (top-down)

Specification components:
- Functional requirements,
- Security criteria,
- **Operation requisites:** limitations to design space
- Ranked quality requirements
  (priority: performance or security?)

Completness

- **The specification:**
  – distinction from undesired behavior,
  ✓ avoid misunderstandings
- **Automatic support of specification completeness:**
  – phrase constraints
  (e.g. validity time interval rules for input)
  – verification (e.g. reachability analysis)
- **Further analysis of software model:**
  – controller (state machine model for behavioral description),
  – sensors, controllers, controlled environment

Criteria of completeness



Inputs and outputs

Completeness of input and output variables

- reaction must be defined for all possible inputs from sensors (even if NOP)
- unused, but possible output values must be verified (e.g. valve always opened, no closing procedure)
- credibility check required upon security-critical output values

Trigger events

Expected input values:

- prerequisites relative to environment
- acceptable domains

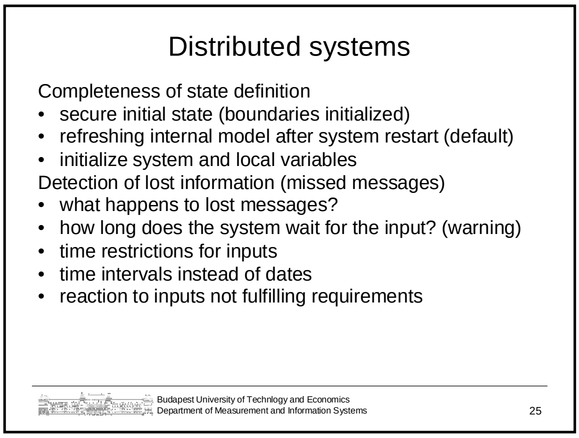
„Unexpected” input values:

- potential error
- pre-defined reaction
- logging

From all states, for all events (event combinations)

- Specified behavior
- Even if there is no event for a certain period

Input check required



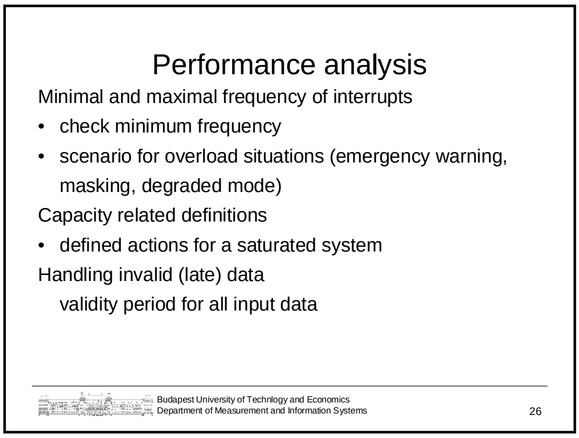
Distributed systems

Completeness of state definition

- secure initial state (boundaries initialized)
- refreshing internal model after system restart (default)
- initialize system and local variables

Detection of lost information (missed messages)

- what happens to lost messages?
- how long does the system wait for the input? (warning)
- time restrictions for inputs
- time intervals instead of dates
- reaction to inputs not fulfilling requirements



Performance analysis

Minimal and maximal frequency of interrupts

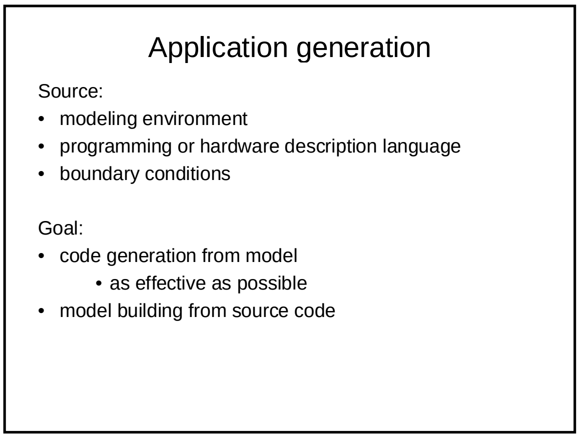
- check minimum frequency
- scenario for overload situations (emergency warning, masking, degraded mode)

Capacity related definitions

- defined actions for a saturated system

Handling invalid (late) data

validity period for all input data



Application generation

Source:

- modeling environment
- programming or hardware description language
- boundary conditions

Goal:

- code generation from model
    - as effective as possible
- model building from source code

Code generation levels

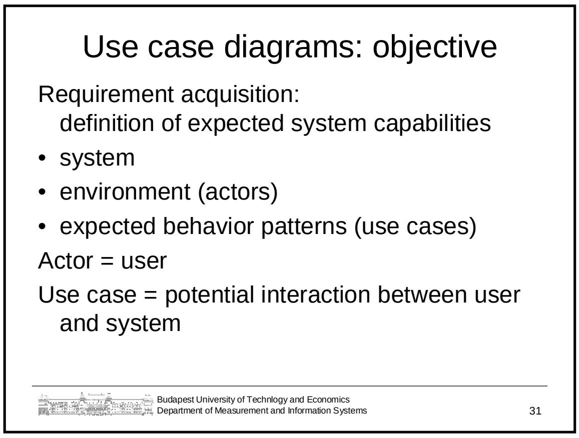
- From the entire model
- From certain parts of the model
  - class
    - declaration
    - declaration + behaviour
  - package
  - ...

Requirement modelling and architecture design

Use case, class and package diagrams

Content

- Use case diagrams
  - actors, use cases
  - scenarios, event sequences
- Class and object diagrams
  - classes, objects
  - relations
- Package diagram



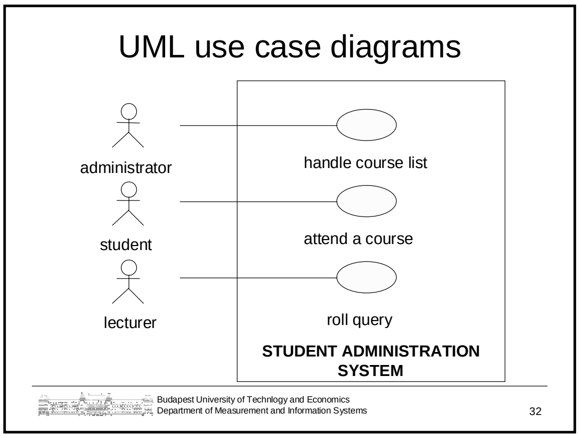
Use case diagrams: objective

Requirement acquisition:
definition of expected system capabilities

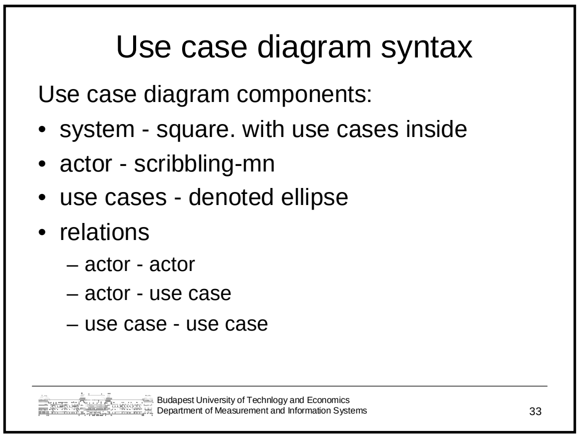
- system
- environment (actors)
- expected behavior patterns (use cases)

Actor = user

Use case = potential interaction between user and system



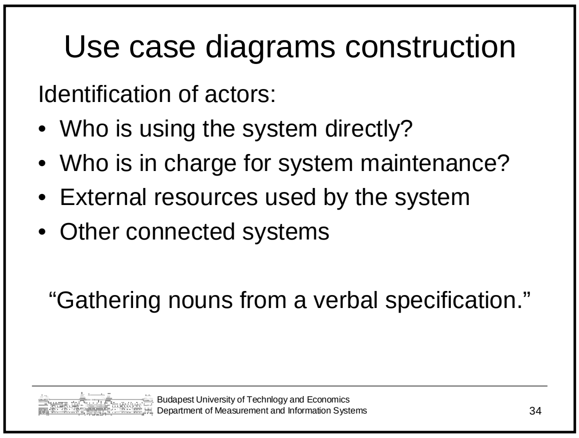
UML use case diagrams



Use case diagram syntax

Use case diagram components:

- system - square. with use cases inside
- actor - scribbling-mn
- use cases - denoted ellipse
- relations
  - actor - actor
  - actor - use case
  - use case - use case

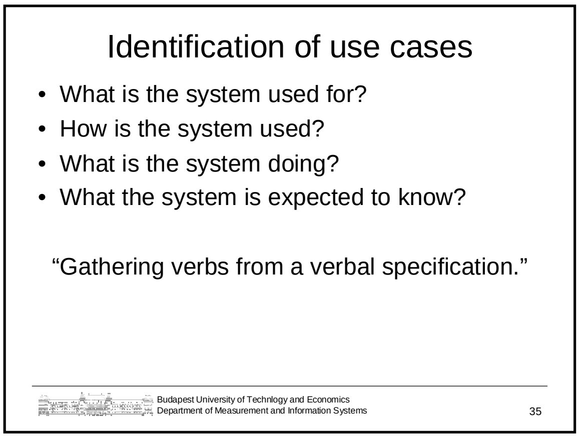


Use case diagrams construction

Identification of actors:

- Who is using the system directly?
- Who is in charge for system maintenance?
- External resources used by the system
- Other connected systems

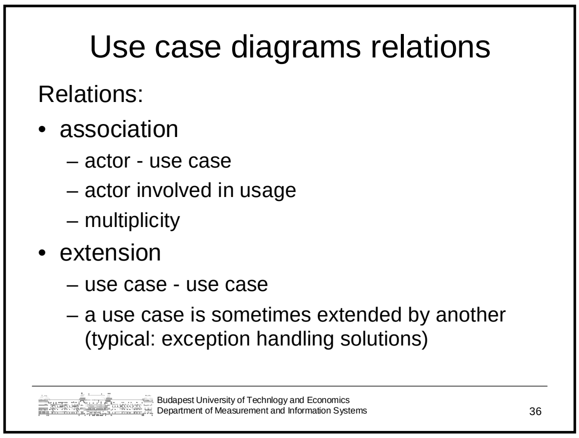
“Gathering nouns from a verbal specification.”



Identification of use cases

- What is the system used for?
- How is the system used?
- What is the system doing?
- What the system is expected to know?

“Gathering verbs from a verbal specification.”



Use case diagrams relations

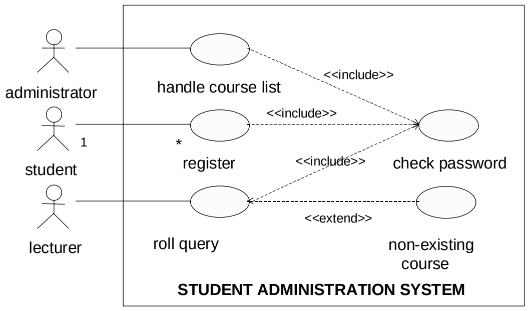
Relations:

- association
  - actor - use case
  - actor involved in usage
  - multiplicity
- extension
  - use case - use case
  - a use case is sometimes extended by another (typical: exception handling solutions)

Use case diagrams relations

- generalization
  - actor - actor
  - use case - use case
  - a use case or an actor is a special occurrence of another use case, or actor
- include
  - use case - use case
  - a use case always includes another

Use case diagrams example

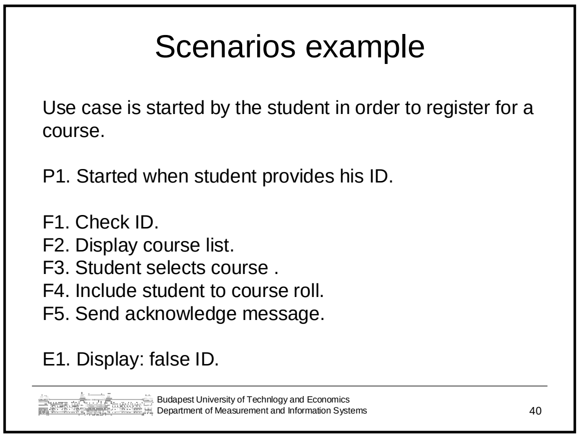


Scenarios

Scenario: what does the system do from the actors point of view (for all use cases)

- brief description
- prerequisites, initial event
- series of events
- alternative series of events
- final event, after-effects

NOT UML!



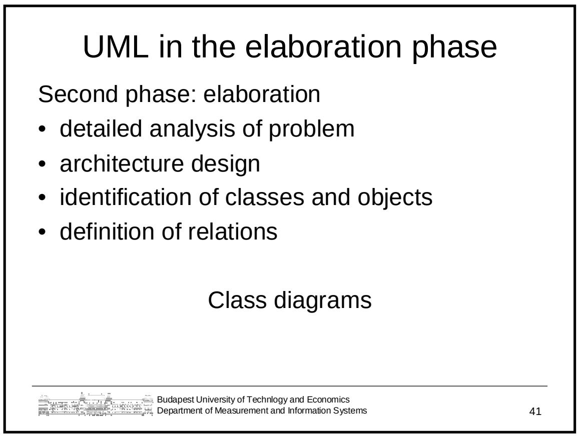
Scenarios example

Use case is started by the student in order to register for a course.

P1. Started when student provides his ID.

F1. Check ID.
F2. Display course list.
F3. Student selects course .
F4. Include student to course roll.
F5. Send acknowledge message.

E1. Display: false ID.

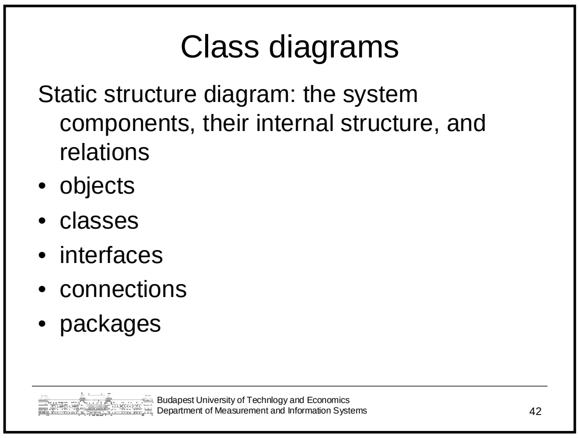


UML in the elaboration phase

Second phase: elaboration

- detailed analysis of problem
- architecture design
- identification of classes and objects
- definition of relations

Class diagrams

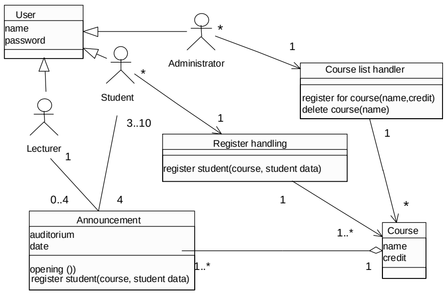


Class diagrams

Static structure diagram: the system components, their internal structure, and relations

- objects
- classes
- interfaces
- connections
- packages

Class diagrams example



User
name
password

Administrator

Student

Lecturer

Course list handler
register for course(name,credit)
delete course(name)

Register handling
register student(course, student data)

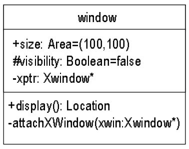
Announcement
auditorium
date
opening ())
register student(course, student data)

Course
name
credit

Class diagram elements

Class

- name
- attributes
  - – visibility
  - – type
  - – initial value
- methods
  - – visibility
  - – type



window

+size: Area=(100,100)
#visibility: Boolean=false
-xptr: Xwindow*

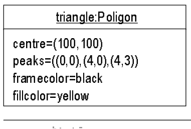
+display(): Location
-attachXWindow(xwin:Xwindow*)

Refinement

Object:

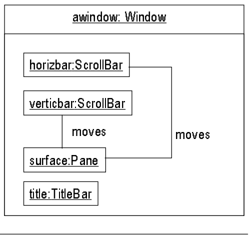
- name
- attributes

Composite object



triangle:Poligon

centre=(100,100)
peaks=((0,0),(4,0),(4,3))
framecolor=black
fillcolor=yellow



awindow: Window

horizbar:ScrollBar

verticbar:ScrollBar

surface:Pane

title:TitleBar

Association

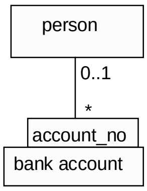
Association: relation of classes and objects

- name
- navigation properties, direction
- roles
- multiplicity
- type
- implementation
  - – attribute
  - – method

Structuring

Qualification -

- attribute enabling the grouping of objects,
- querying of group elements



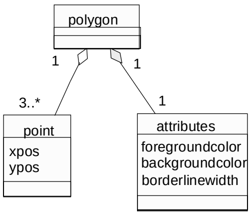
Hierarchy

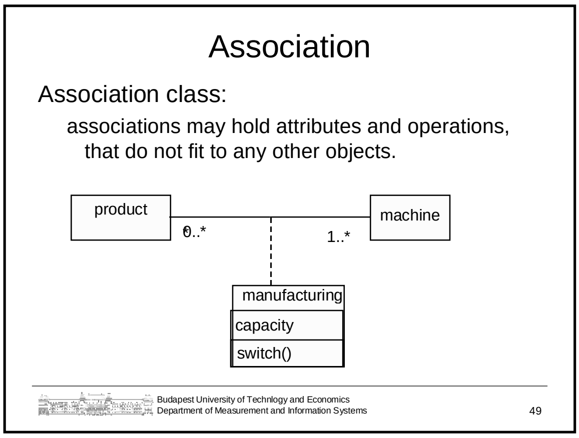
Content:

between container and its component objects

Composition:

Container physically holds its parts

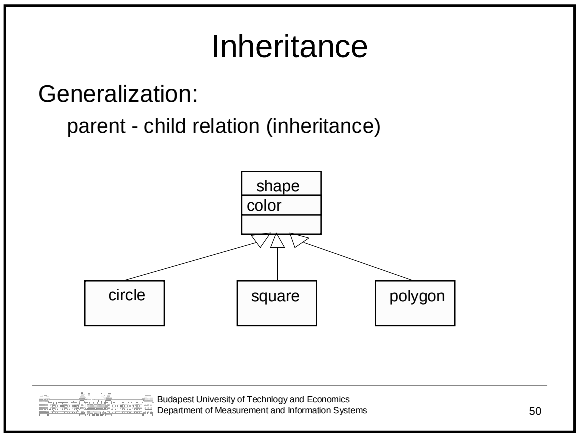




Association

Association class:

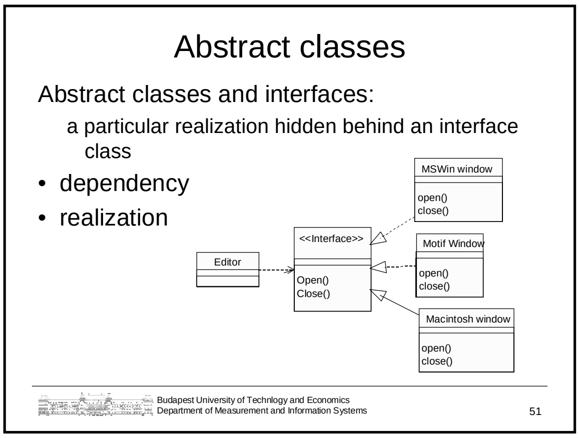
associations may hold attributes and operations, that do not fit to any other objects.



Inheritance

Generalization:

parent - child relation (inheritance)



Abstract classes

Abstract classes and interfaces:

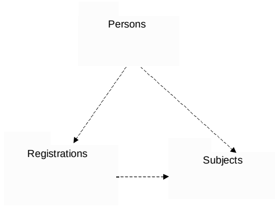
a particular realization hidden behind an interface class

- dependency
- realization

Package diagrams

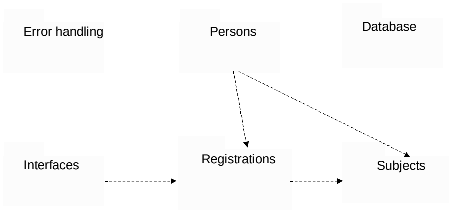
Package:

- functionally correlated model components
- package
- relation
  - dependency



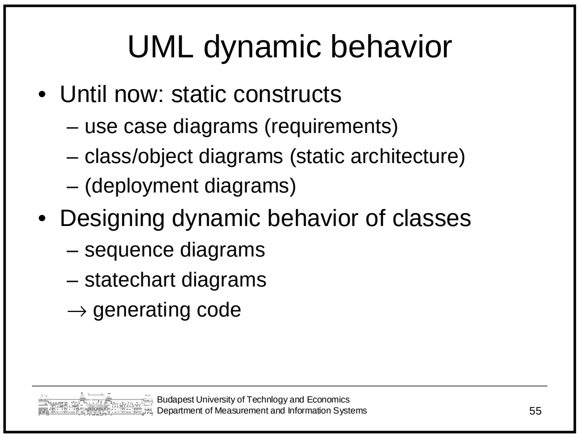
Package diagrams and SW architecture

Top level package diagram - SW architecture



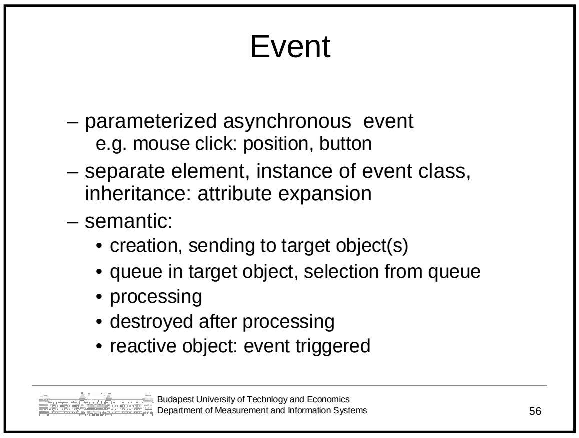
Control process design

- Basic components: events, operations
- Sequence diagrams
- Statechart



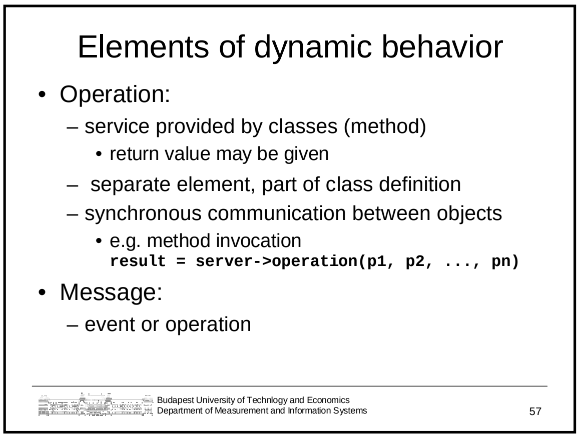
UML dynamic behavior

- Until now: static constructs
  - – use case diagrams (requirements)
  - – class/object diagrams (static architecture)
  - – (deployment diagrams)
- Designing dynamic behavior of classes
  - – sequence diagrams
  - – statechart diagrams
  - → generating code



Event

- – parameterized asynchronous event
  e.g. mouse click: position, button
- – separate element, instance of event class, inheritance: attribute expansion
- – semantic:
  - creation, sending to target object(s)
  - queue in target object, selection from queue
  - processing
  - destroyed after processing
  - reactive object: event triggered



Elements of dynamic behavior

- Operation:
  - – service provided by classes (method)
    - return value may be given
  - – separate element, part of class definition
  - – synchronous communication between objects
    - e.g. method invocation

```
result = server->operation(p1, p2, ..., pn)
```

- Message:
  - – event or operation

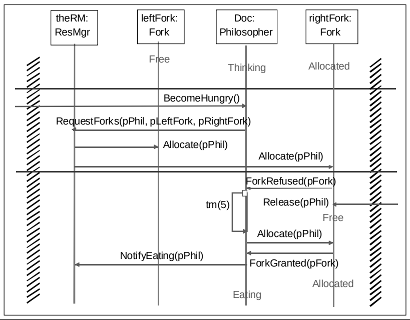
Elements of dynamic behavior II.

- State:
  - state of an object
  - determined by:
    - attribute values (e.g. x<3)
    - condition fulfillment (e.g. operation executable)
- State transition:
  - change of state
  - due to an incoming message (triggered) or autonomously (null trigger)
- Action: operation performed by the object

Sequence diagrams

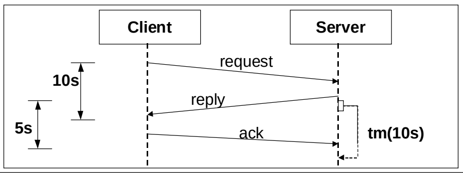
- Provides:
  - order of messages between objects
  - a typical case scenario
- Available for:
  - development of detailed behavior descriptions (typical cases for a statechart)
  - development of tests
  - check timing parameters

Sequence diagrams - overview



Timing on sequence diagrams

- Descriptive (defining requirements)
  - time interval
  - timeout

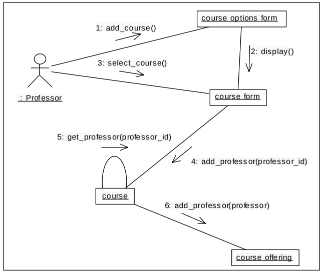


Sequence diagram components

- Object (class instance)
- Lifecycle
- Message: event or object
- Creating/deleting objects
- Condition/state, fork
- Timeout
- System boundaries
- Partitioning
- Time interval definition

Co-operation diagram

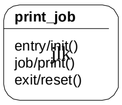
- Relations and messages between objects



State charts - introduction

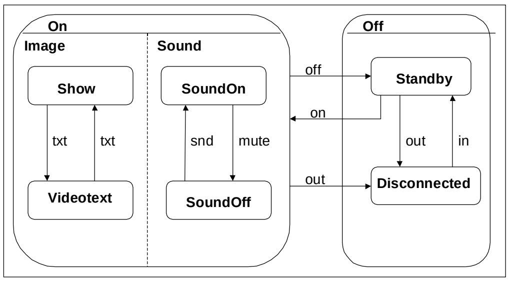
- Behavior of (reactive) objects
  - reaction to messages (event, operation): state transitions and actions
  - conventional: state diagram
- State chart: extension of state diagram
  - state hierarchy: refine states
  - concurrent behavior: parallel threads
  - history: last active state configuration

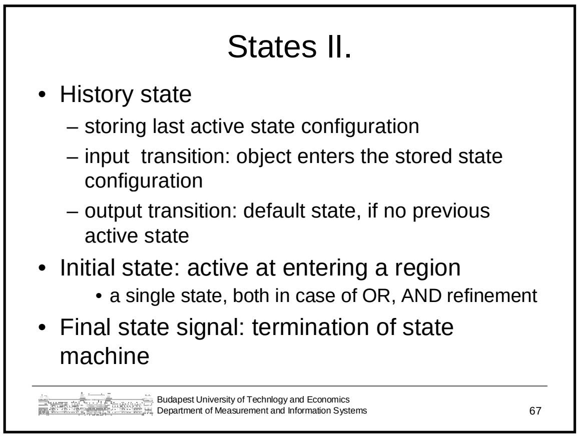
States I.



- Parameters:
  - entry action
  - exit action
  - static reaction
- State refinement:
  - simple state (no refinement)
  - OR refinement : inferior state machine, one state active at a time
  - AND refinement : concurrent regions (state machines), all regions simultaneously active

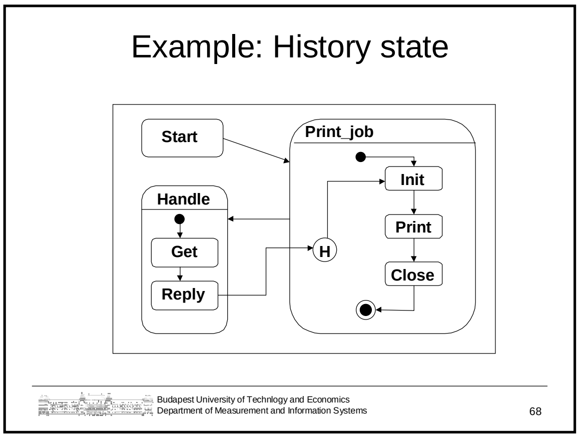
State refinement- example: TV remote control



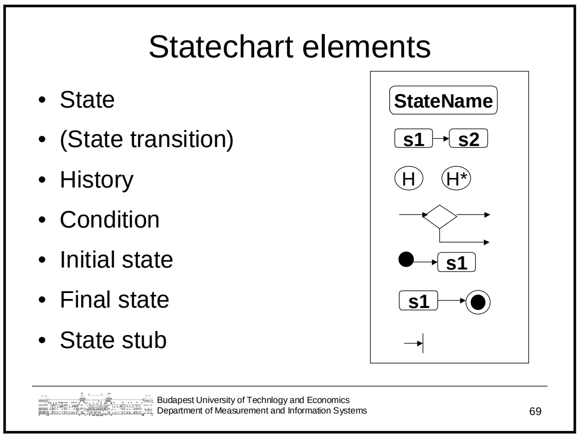


States II.

- History state
  - storing last active state configuration
  - input transition: object enters the stored state configuration
  - output transition: default state, if no previous active state
- Initial state: active at entering a region
  - a single state, both in case of OR, AND refinement
- Final state signal: termination of state machine

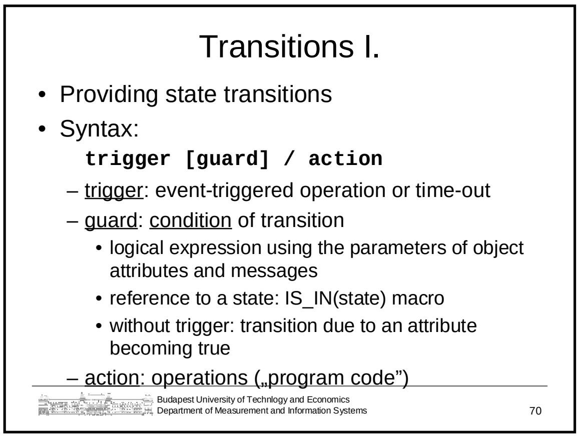


Example: History state



Statechart elements

- State
- (State transition)
- History
- Condition
- Initial state
- Final state
- State stub



Transitions I.

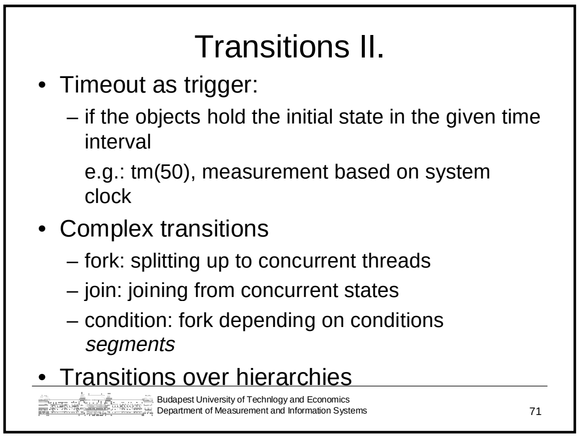
- Providing state transitions
- Syntax:

  `trigger [guard] / action`

  – trigger: event-triggered operation or time-out

  – guard: condition of transition
    - logical expression using the parameters of object attributes and messages
    - reference to a state: IS_IN(state) macro
    - without trigger: transition due to an attribute becoming true

  – action: operations („program code”)



Transitions II.

- Timeout as trigger:

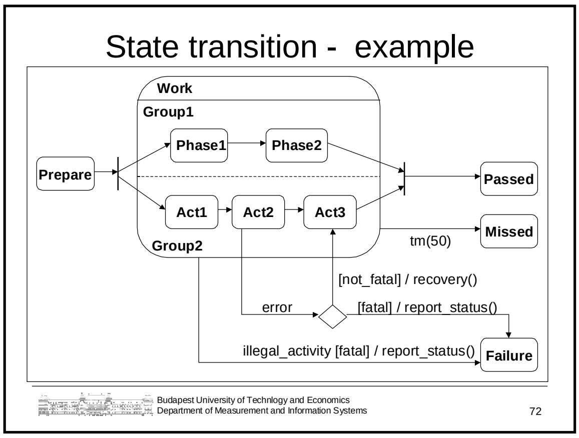
  – if the objects hold the initial state in the given time interval

  e.g.: tm(50), measurement based on system clock
- Complex transitions

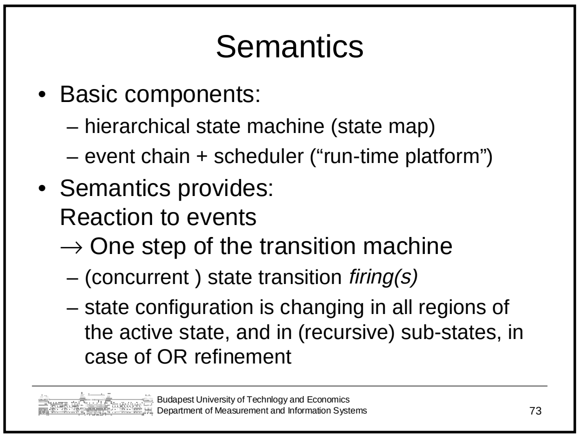
  – fork: splitting up to concurrent threads

  – join: joining from concurrent states

  – condition: fork depending on conditions *segments*
- Transitions over hierarchies



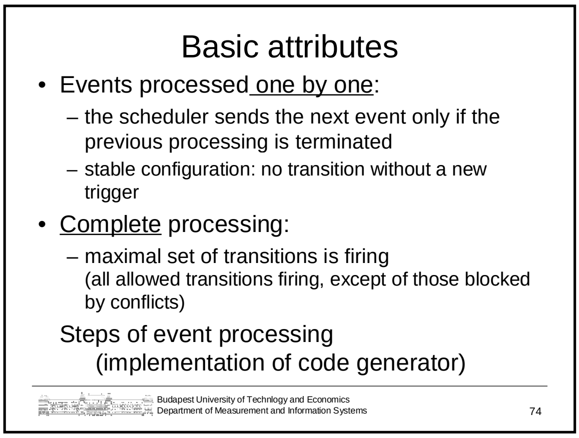
State transition - example



Semantics

- Basic components:
  - – hierarchical state machine (state map)
  - – event chain + scheduler (“run-time platform”)
- Semantics provides:
  Reaction to events
  → One step of the transition machine
  - – (concurrent) state transition *firing(s)*
  - – state configuration is changing in all regions of the active state, and in (recursive) sub-states, in case of OR refinement

Budapest University of Technlogy and Economics
Department of Measurement and Information Systems
73

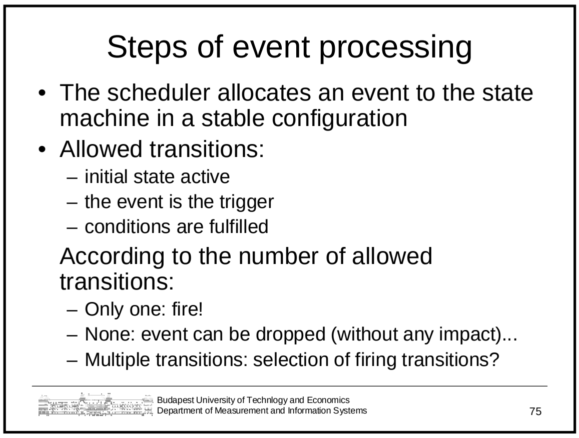


Basic attributes

- Events processed one by one:
  - – the scheduler sends the next event only if the previous processing is terminated
  - – stable configuration: no transition without a new trigger
- Complete processing:
  - – maximal set of transitions is firing
    (all allowed transitions firing, except of those blocked by conflicts)

Steps of event processing
(implementation of code generator)

Budapest University of Technlogy and Economics
Department of Measurement and Information Systems
74



Steps of event processing

- The scheduler allocates an event to the state machine in a stable configuration
- Allowed transitions:
  - – initial state active
  - – the event is the trigger
  - – conditions are fulfilled

According to the number of allowed transitions:

- – Only one: fire!
- – None: event can be dropped (without any impact)...
- – Multiple transitions: selection of firing transitions?

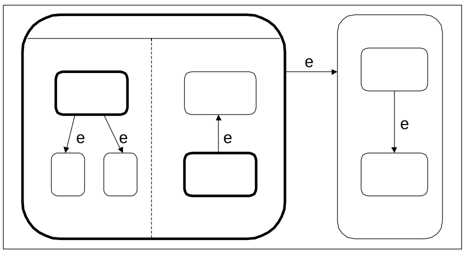
Budapest University of Technlogy and Economics
Department of Measurement and Information Systems
75

Steps of event processing

- Selecting firing transitions:
  - Maximal number of non-conflicting transitions simultaneous firing of concurrent transitions
  - Conflict: Leaving the same state (intersection of state sets is not empty)
    - → Solution:
    - <u>priority</u>: higher, if the initial state has <u>lower level</u> in hierarchy (refinement) (OO principle: refinement, re-definition)
    - <u>random choice</u>, if states are independent

Steps of event processing

- Conflict resolution - example:



Steps of event processing

- Selected fireable transitions: random order
- A single fireable transition:
  - leave initial state(s), first executing exit actions on lower level
  - execution of action(s)
  - enter target state(s) → new configuration first executing enter actions at the higher level

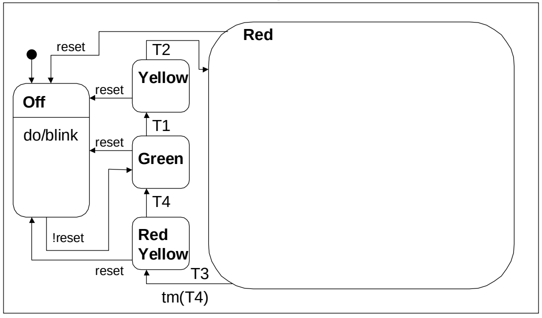
Steps of event processing

- Entering a new configuration:
  - simple target state: part of new configuration
  - non-concurrent superstate: an active substate with direct access or the initial state
  - concurrent target state: must have a reachable active sub-state or the initial state in all regions
  - history state: last active state
  - instable state: immediate transition (part of the step)

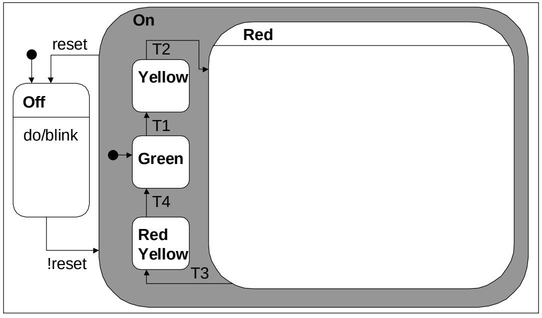
Example

- Intersection, traffic light controller
  - switch off (blinking yellow)
  - switch on: at the beginning green for main road
  - green, yellow, red etc. time intervals (scheduler)
  - three cars waiting on the main road: green light required independently of scheduler
  - snap a photo of irregular drivers
  - function activated/deactivated manually

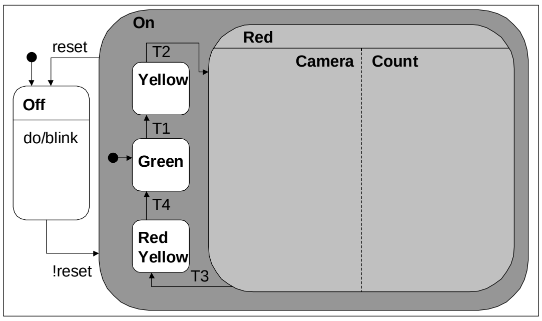
1. Change color



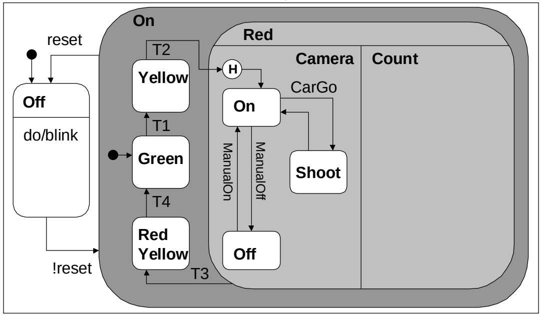
2. Hierarchy

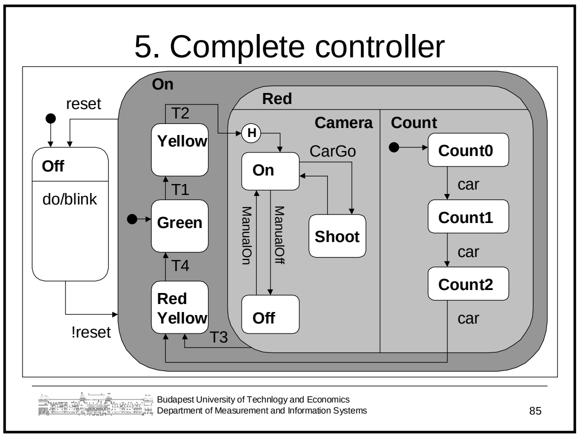


3. Concurrent sub-states

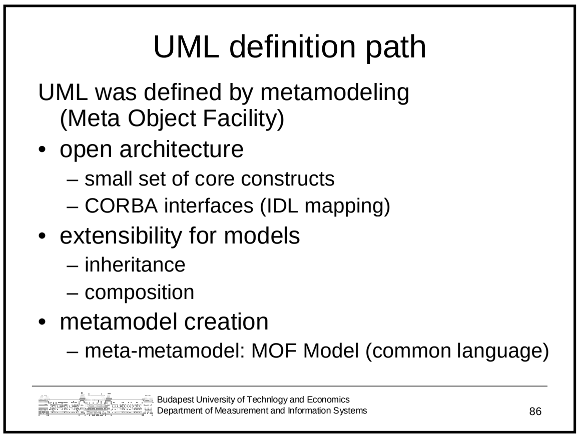


4. History state





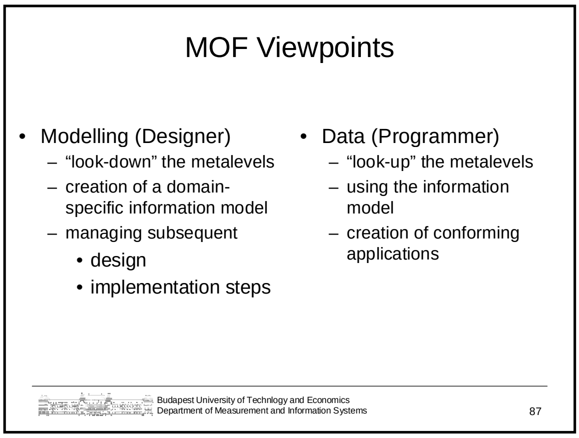
5. Complete controller



UML definition path

UML was defined by metamodeling
(Meta Object Facility)

- open architecture
  - – small set of core constructs
  - – CORBA interfaces (IDL mapping)
- extensibility for models
  - – inheritance
  - – composition
- metamodel creation
  - – meta-metamodel: MOF Model (common language)

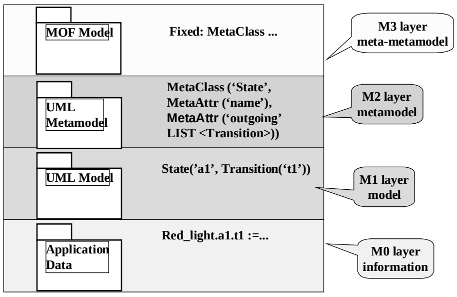


MOF Viewpoints

- Modelling (Designer)
  - – “look-down” the metalevels
  - – creation of a domain-specific information model
  - – managing subsequent
    - design
    - implementation steps

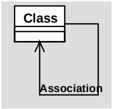
- Data (Programmer)
  - – “look-up” the metalevels
  - – using the information model
  - – creation of conforming applications

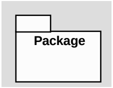
Four Layer Metadata Architecture



Basic MOF Constructs

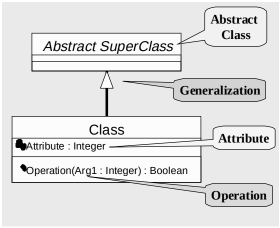
- *Class*:
  - describing meta-objects
- *Association*:
  - binary relation between *Class*es
- *DataTypes*:
  - modelling external data, primitive types
- *Package* :
  - modularizing the models
- *Constraint* (OCL):
  - semantic restrictions on elements

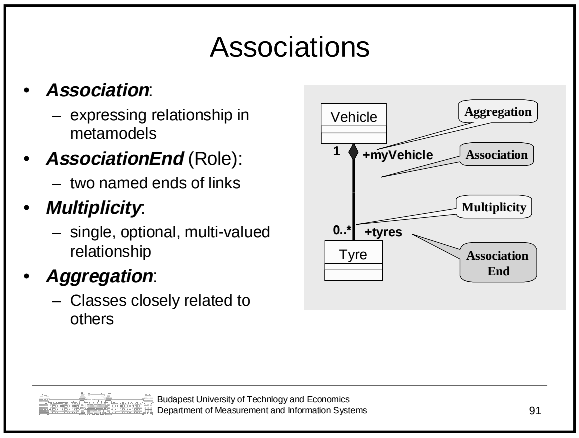




Classes

- ***Class**es*:
  - describing meta-objects
- ***Attribute**s*:
  - value holder in a Class instance
- ***Operation**s*:
  - name and type signature
- ***Generalization***:
  - inheritance
- ***Abstract Classes***:
  - no instance objects





Associations

- ***Association***:
  - expressing relationship in metamodels
- ***AssociationEnd*** (Role):
  - two named ends of links
- ***Multiplicity***:
  - single, optional, multi-valued relationship
- ***Aggregation***:
  - Classes closely related to others

Vehicle

1 +myVehicle

0..* +tyres

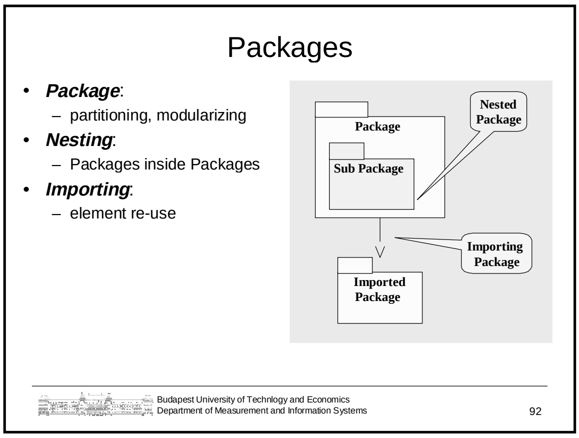
Tyre

Aggregation

Association

Multiplicity

Association End



Packages

- ***Package***:
  - partitioning, modularizing
- ***Nesting***:
  - Packages inside Packages
- ***Importing***:
  - element re-use

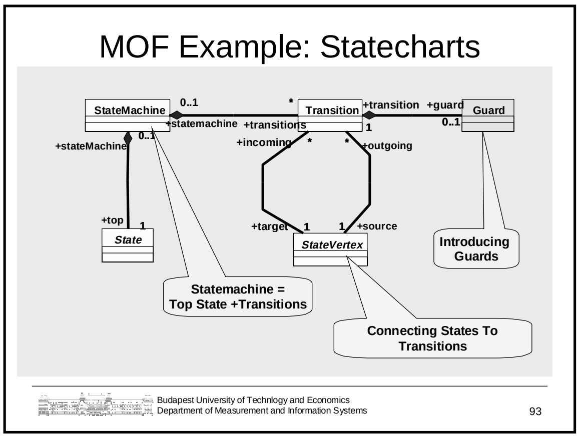
Package

Sub Package

Nested Package

Importing Package

Imported Package



MOF Example: Statecharts

StateMachine

0..1

+statemachine

+stateMachine

0..1

*

+transitions

Transition

+transition

1

+guard

0..1

Guard

+incoming

*

*

+outgoing

+top

1

State

+target

1

1

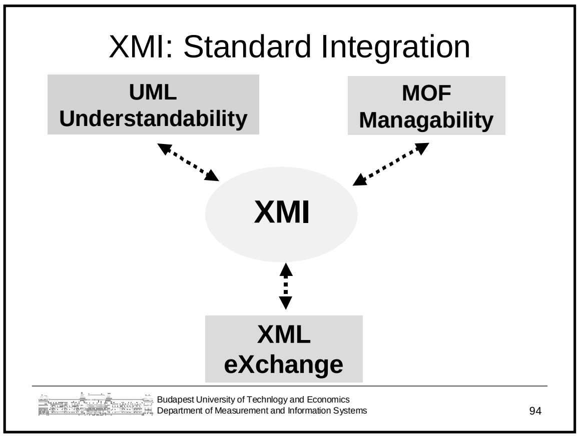
+source

StateVertex

Introducing Guards

Statemachine = Top State +Transitions

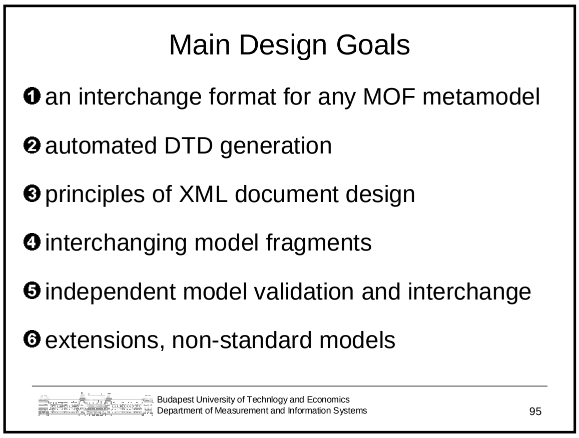
Connecting States To Transitions



XMI: Standard Integration

Budapest University of Technlogy and Economics
Department of Measurement and Information Systems

94



Main Design Goals

❶ an interchange format for any MOF metamodel

❷ automated DTD generation

❸ principles of XML document design

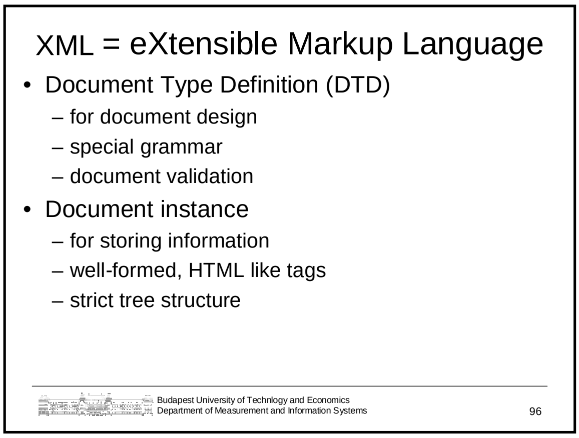
❹ interchanging model fragments

❺ independent model validation and interchange

❻ extensions, non-standard models

Budapest University of Technlogy and Economics
Department of Measurement and Information Systems

95

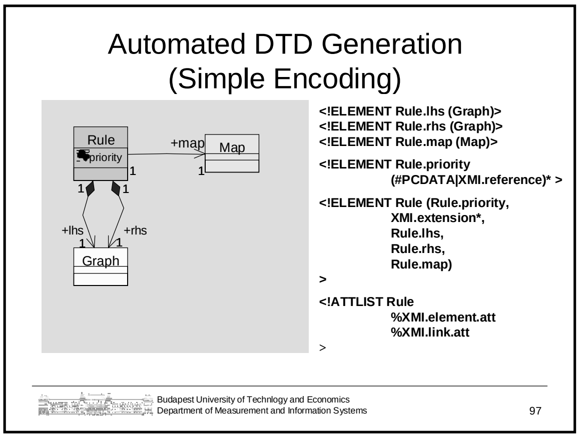


XML = eXtensible Markup Language

- Document Type Definition (DTD)
  - for document design
  - special grammar
  - document validation
- Document instance
  - for storing information
  - well-formed, HTML like tags
  - strict tree structure

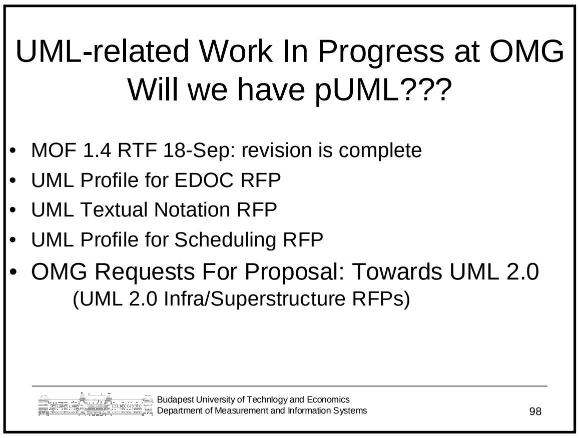
Budapest University of Technlogy and Economics
Department of Measurement and Information Systems

96



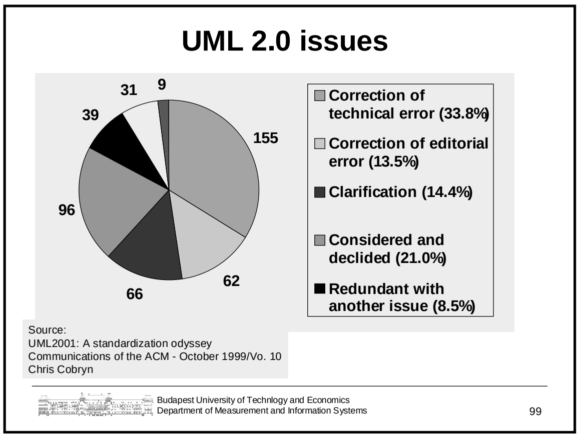
Automated DTD Generation (Simple Encoding)

```
<!ELEMENT Rule.lhs (Graph)>
<!ELEMENT Rule.rhs (Graph)>
<!ELEMENT Rule.map (Map)>
<!ELEMENT Rule.priority
          (#PCDATA|XMI.reference)* >
<!ELEMENT Rule (Rule.priority,
          XMI.extension*,
          Rule.lhs,
          Rule.rhs,
          Rule.map)
>
<!ATTLIST Rule
          %XMI.element.att
          %XMI.link.att
>
```



UML-related Work In Progress at OMG Will we have pUML???

- MOF 1.4 RTF 18-Sep: revision is complete
- UML Profile for EDOC RFP
- UML Textual Notation RFP
- UML Profile for Scheduling RFP
- OMG Requests For Proposal: Towards UML 2.0
  (UML 2.0 Infra/Superstructure RFPs)



UML 2.0 issues

Source:
UML2001: A standardization odyssey
Communications of the ACM - October 1999/Vo. 10
Chris Cobryn

UML 2.0
(major revision ⇒ 2/2002)

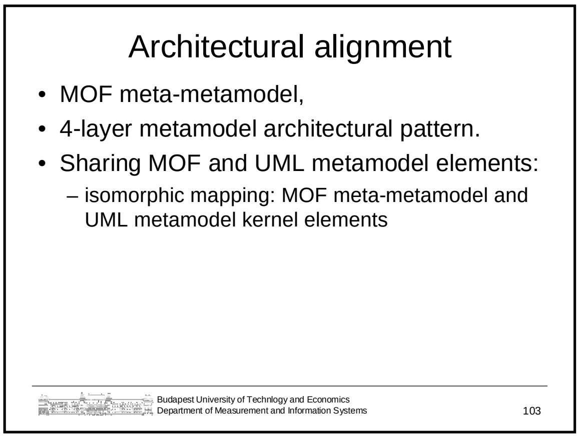
- **General requirements**
- **Infrastructure:** architectural alignment, restructuring and extension mechanism
  - UML 2.0 metamodel.
- **Superstructure** : refinement and extension of UML 1.x semantics and notation.

Infrastructure:
UML metamodel problems

- Compatibility with the MOF meta-metamodel?
- No strict conformance to 4-layer metamodel pattern!
- Deviation ⇒ implementation problems in other OMG standards (e.g. XMI)
  - Current UML: inclusion of a “physical metamodel” (additional detail for model transformations and interchange)

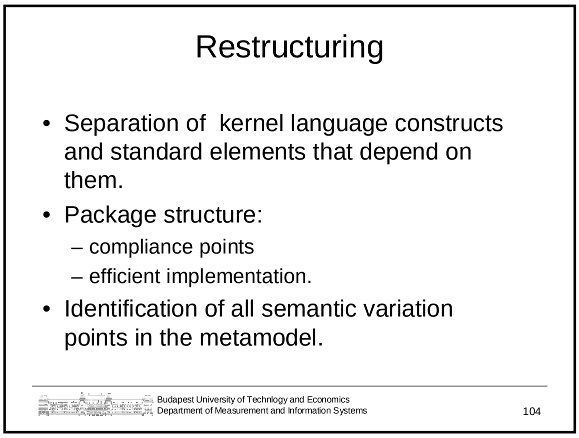
Need for restructuring

- Maintenance, implementation and extension
- Large metamodel
  - > 100 metaclasses, >70 standard elements
  - continuing expansion, behavioral semantics
  - inconsistencies.
- Uneven in the depth and quality of its semantics,
- Mixing abstract and implementation-specific constructs



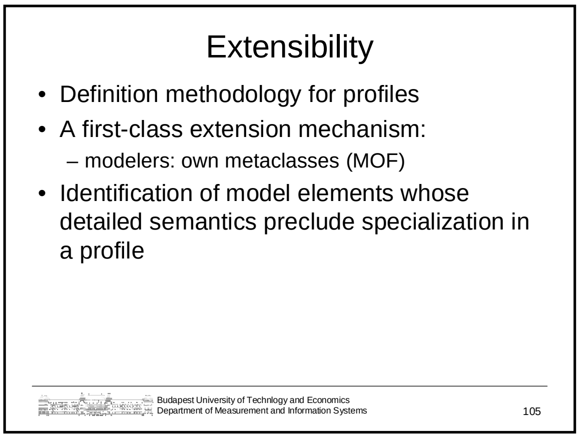
Architectural alignment

- MOF meta-metamodel,
- 4-layer metamodel architectural pattern.
- Sharing MOF and UML metamodel elements:
  - isomorphic mapping: MOF meta-metamodel and UML metamodel kernel elements



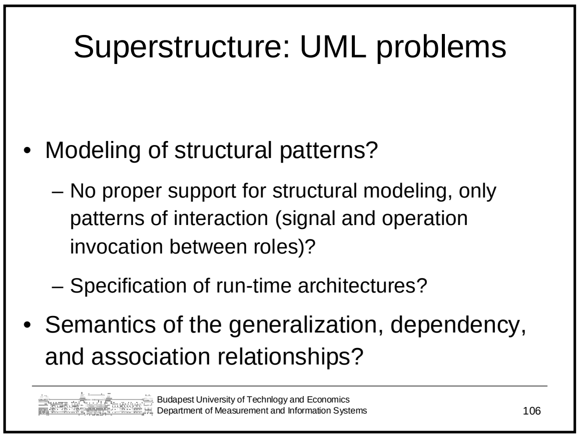
Restructuring

- Separation of kernel language constructs and standard elements that depend on them.
- Package structure:
  - compliance points
  - efficient implementation.
- Identification of all semantic variation points in the metamodel.



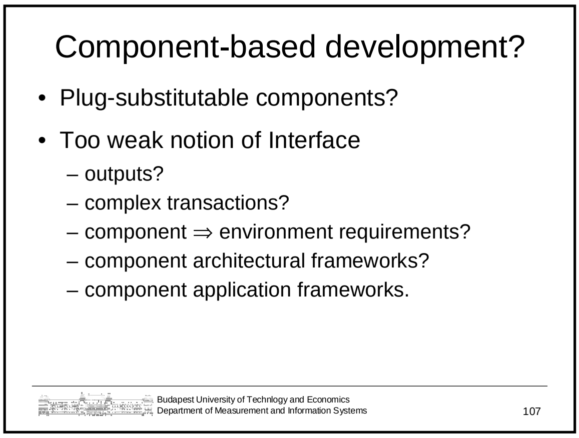
Extensibility

- Definition methodology for profiles
- A first-class extension mechanism:
  - modelers: own metaclasses (MOF)
- Identification of model elements whose detailed semantics preclude specialization in a profile



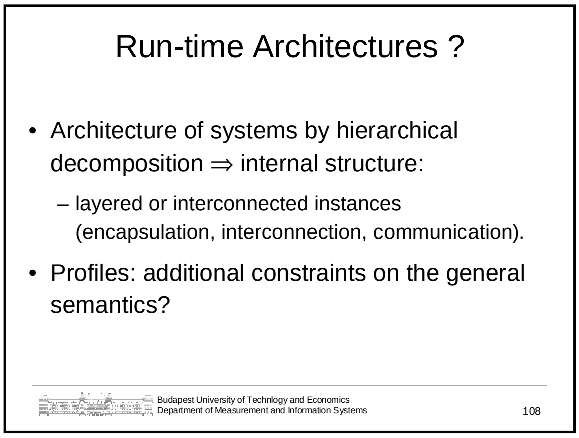
Superstructure: UML problems

- Modeling of structural patterns?
  - – No proper support for structural modeling, only patterns of interaction (signal and operation invocation between roles)?
  - – Specification of run-time architectures?
- Semantics of the generalization, dependency, and association relationships?



Component-based development?

- Plug-substitutable components?
- Too weak notion of Interface
  - – outputs?
  - – complex transactions?
  - – component ⇒ environment requirements?
  - – component architectural frameworks?
  - – component application frameworks.



Run-time Architectures ?

- Architecture of systems by hierarchical decomposition ⇒ internal structure:
  - – layered or interconnected instances (encapsulation, interconnection, communication).
- Profiles: additional constraints on the general semantics?

Inheritance mechanisms ?

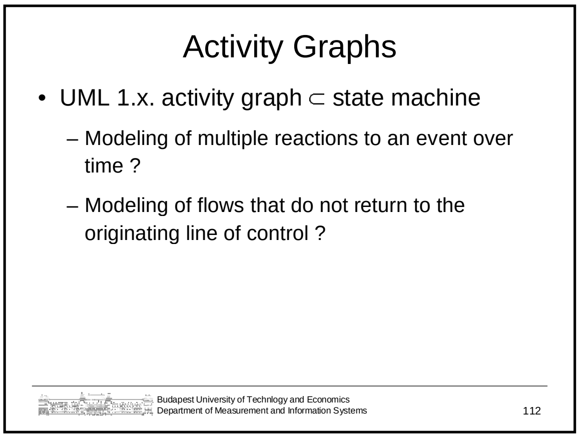
- Implementation languages ⇔
  UML generalization relationship.
- Elementwise scope of inheritance?
  What is inherited for each model element ?
- StateMachines cannot be generalized!

State machines

– Complexity of state machines cannot always be fully captured by hierarchical composition (Composite states). groups of states with identical behavior but where the same state participates in more than one such group.

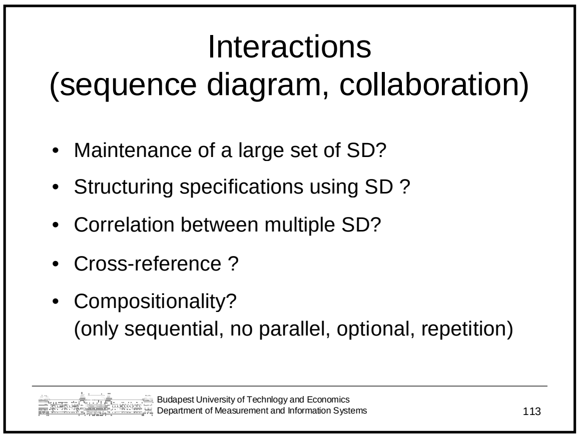
Data flow modeling?

- Removal of restrictions on activity graph modeling due to the mapping to state machines ⇒
  data flow modeling at a high level of granularity



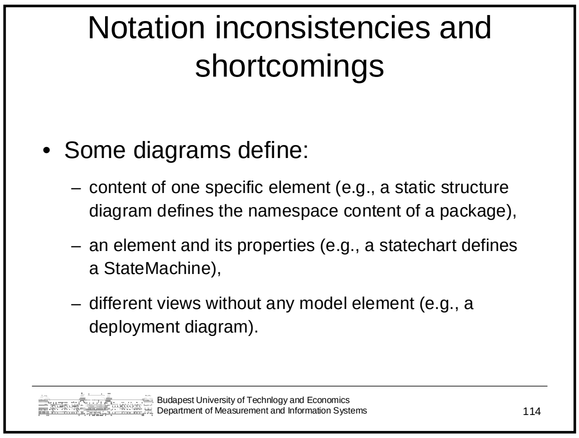
Activity Graphs

- UML 1.x. activity graph ⊂ state machine
  - Modeling of multiple reactions to an event over time ?
  - Modeling of flows that do not return to the originating line of control ?



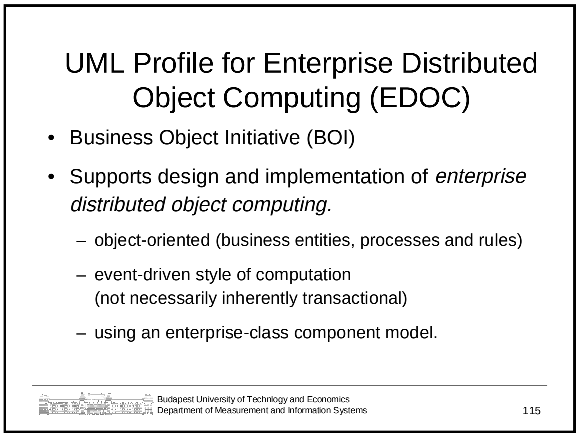
Interactions (sequence diagram, collaboration)

- Maintenance of a large set of SD?
- Structuring specifications using SD ?
- Correlation between multiple SD?
- Cross-reference ?
- Compositionality?
  (only sequential, no parallel, optional, repetition)



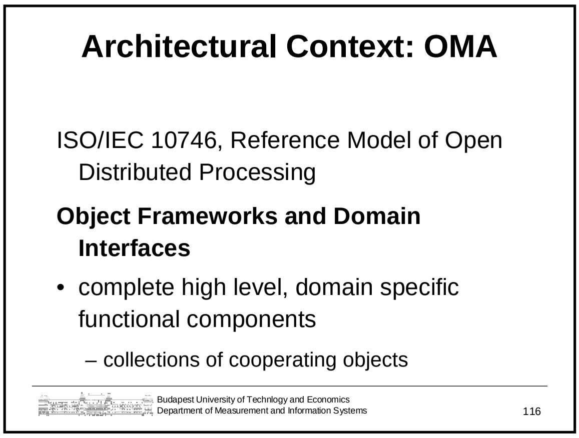
Notation inconsistencies and shortcomings

- Some diagrams define:
  - content of one specific element (e.g., a static structure diagram defines the namespace content of a package),
  - an element and its properties (e.g., a statechart defines a StateMachine),
  - different views without any model element (e.g., a deployment diagram).



UML Profile for Enterprise Distributed Object Computing (EDOC)

- Business Object Initiative (BOI)
- Supports design and implementation of *enterprise distributed object computing.*
  - object-oriented (business entities, processes and rules)
  - event-driven style of computation
    (not necessarily inherently transactional)
  - using an enterprise-class component model.

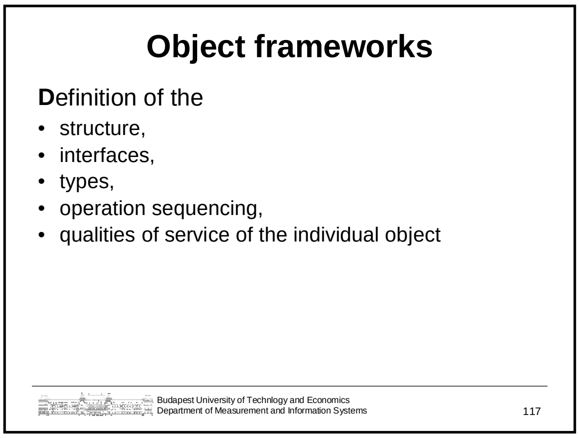


Architectural Context: OMA

ISO/IEC 10746, Reference Model of Open Distributed Processing

Object Frameworks and Domain Interfaces

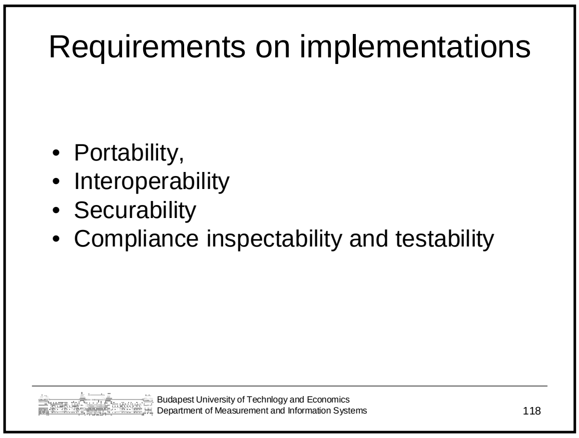
- complete high level, domain specific functional components
  - collections of cooperating objects



Object frameworks

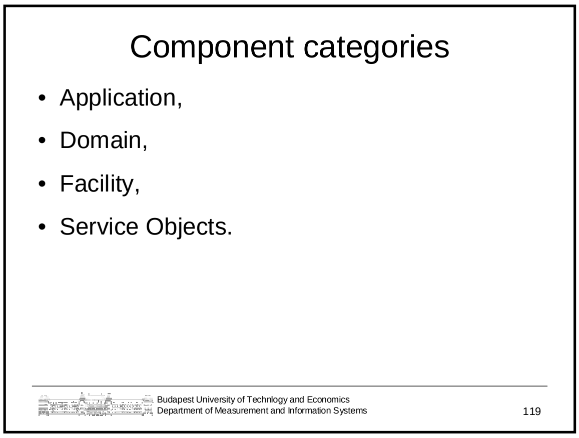
Definition of the

- structure,
- interfaces,
- types,
- operation sequencing,
- qualities of service of the individual object



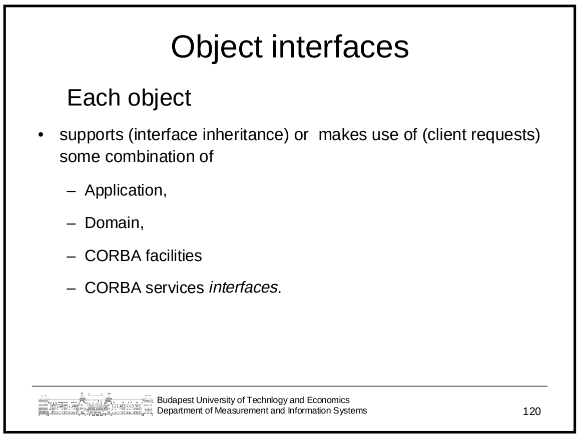
Requirements on implementations

- Portability,
- Interoperability
- Securability
- Compliance inspectability and testability



Component categories

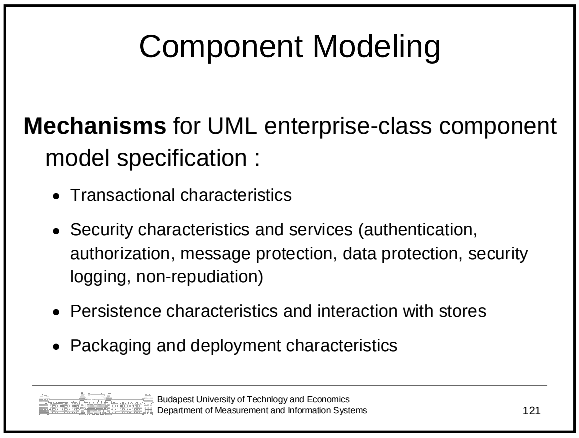
- Application,
- Domain,
- Facility,
- Service Objects.



Object interfaces

Each object

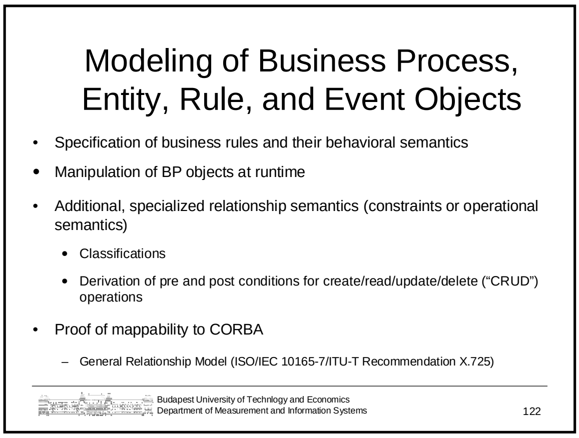
- supports (interface inheritance) or makes use of (client requests) some combination of
  - Application,
  - Domain,
  - CORBA facilities
  - CORBA services *interfaces*.



Component Modeling

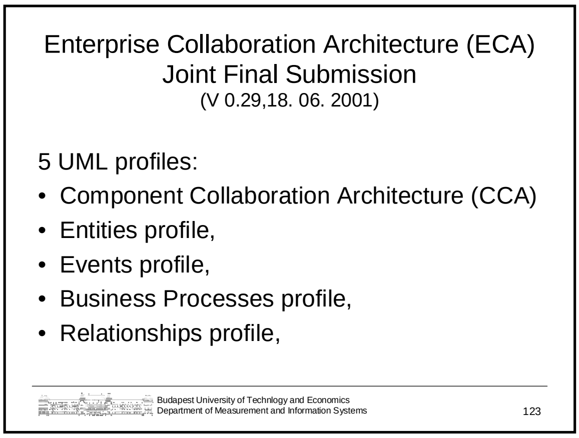
Mechanisms for UML enterprise-class component model specification :

- Transactional characteristics
- Security characteristics and services (authentication, authorization, message protection, data protection, security logging, non-repudiation)
- Persistence characteristics and interaction with stores
- Packaging and deployment characteristics



Modeling of Business Process, Entity, Rule, and Event Objects

- Specification of business rules and their behavioral semantics
- Manipulation of BP objects at runtime
- Additional, specialized relationship semantics (constraints or operational semantics)
  - Classifications
  - Derivation of pre and post conditions for create/read/update/delete (“CRUD”) operations
- Proof of mappability to CORBA
  - General Relationship Model (ISO/IEC 10165-7/ITU-T Recommendation X.725)



Enterprise Collaboration Architecture (ECA) Joint Final Submission

(V 0.29,18. 06. 2001)

5 UML profiles:

- Component Collaboration Architecture (CCA)
- Entities profile,
- Events profile,
- Business Processes profile,
- Relationships profile,

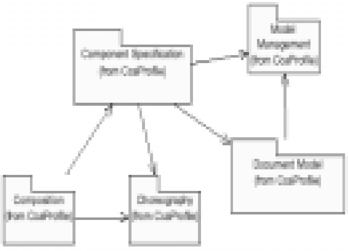
Component Collaboration Architecture (CCA)

- Structural and behavioral system modeling based on “Process Components”
- UML classes, collaborations and activity graphs,
- Interaction: through Ports,

Methodology

- Recursive decomposition of Process Components:
  - Composition (Collaboration) assembly of Process Components,
  - Choreography (Activity Graph) : flow of activities
- Varying and mixed levels of granularity, further specialization by profiles

Main functionalities



Entities profile

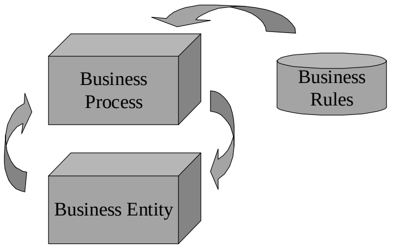
UML extensions

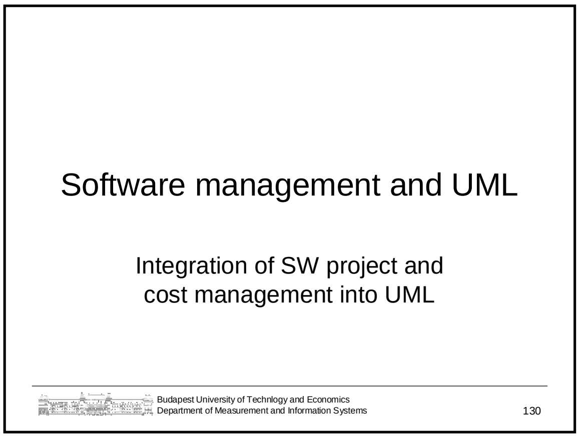
- modeling entity objects
- representations of concepts in the application problem domain
- definition as composable components;
  - attributes, relationships, operations, constraints, dependencies at a technology-independent level

Events profile

- Event driven systems: (business entity, business event, business process, business activity and business rule).
- Basic building blocks:
  - business process
  - business entity

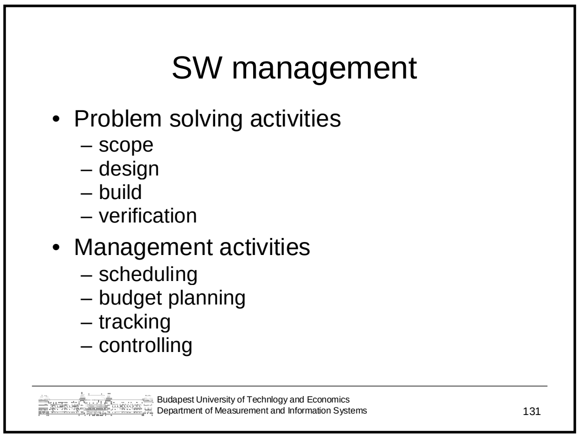
Interaction





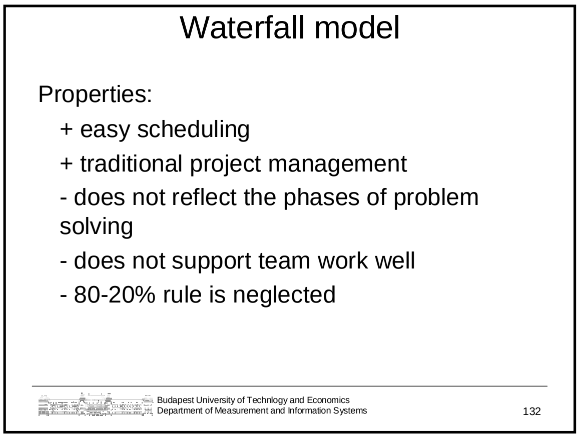
Software management and UML

Integration of SW project and
cost management into UML



SW management

- Problem solving activities
  - scope
  - design
  - build
  - verification
- Management activities
  - scheduling
  - budget planning
  - tracking
  - controlling



Waterfall model

Properties:

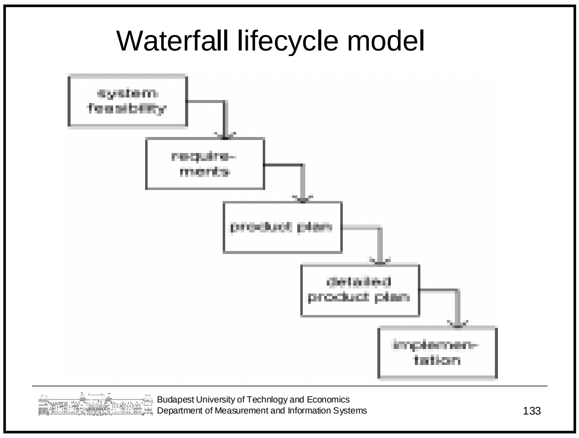
+ easy scheduling

+ traditional project management

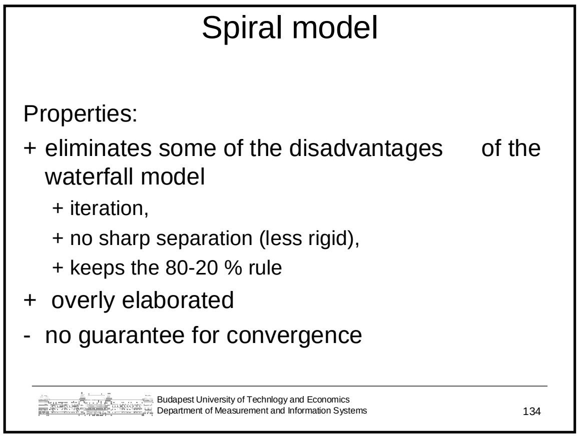
- does not reflect the phases of problem solving

- does not support team work well

- 80-20% rule is neglected



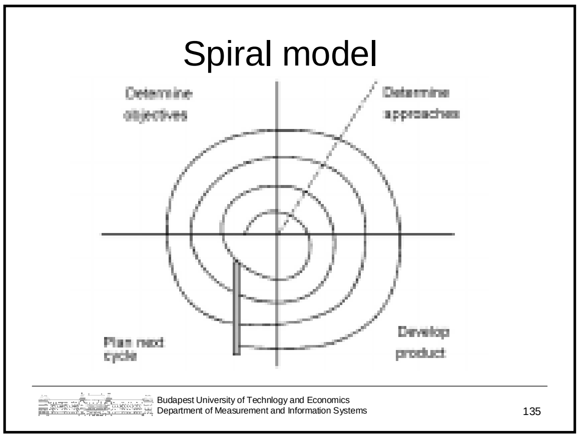
Waterfall lifecycle model



Spiral model

Properties:

+ eliminates some of the disadvantages of the waterfall model
  + iteration,
  + no sharp separation (less rigid),
  + keeps the 80-20 % rule
+ overly elaborated
- no guarantee for convergence



Spiral model

Controlled iteration

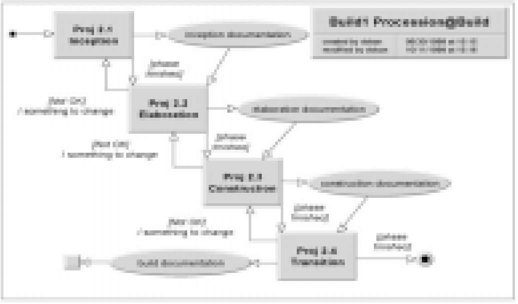
New lifecycle model for OO development:

- combination of how people work and
- proper management.

Basically, it is based on development phases, and activities serve the management during phases.

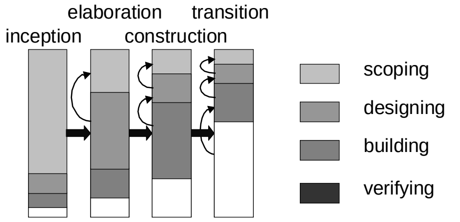
Previous models had serious limitations.

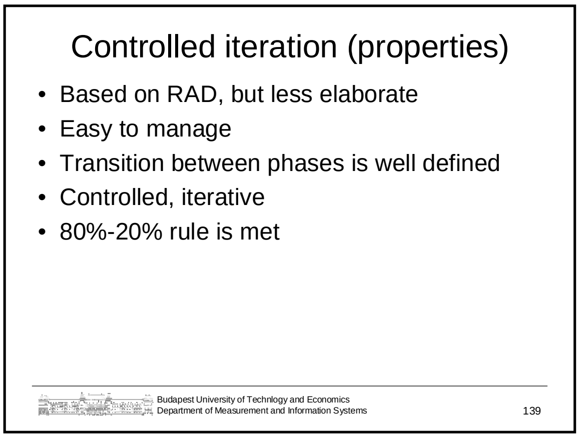
Controlled Iteration Model



Controlled iteration (basic idea)

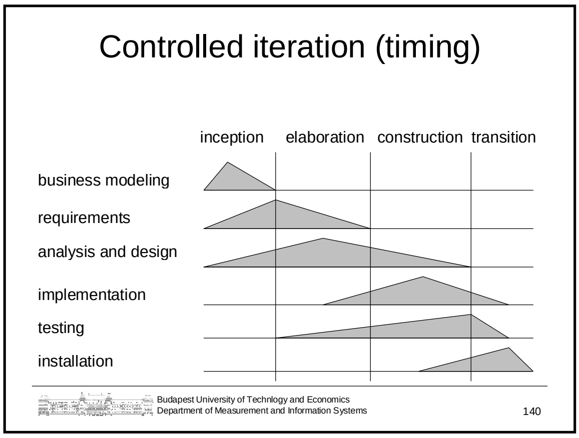
Inter-phase transitions based on maturity:



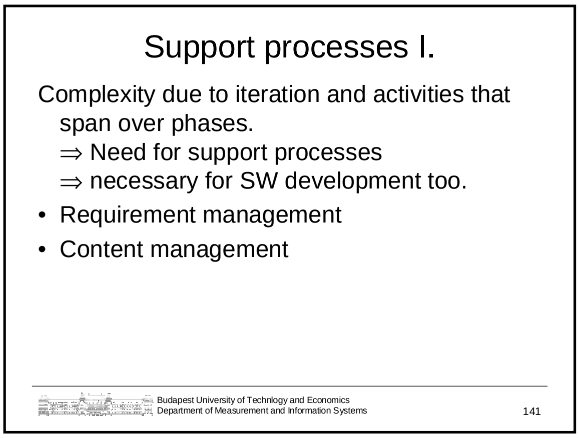


Controlled iteration (properties)

- Based on RAD, but less elaborate
- Easy to manage
- Transition between phases is well defined
- Controlled, iterative
- 80%-20% rule is met



Controlled iteration (timing)



Support processes I.

Complexity due to iteration and activities that span over phases.
⇒ Need for support processes
⇒ necessary for SW development too.

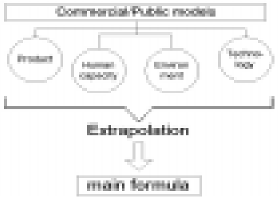
- Requirement management
- Content management

Support processes II.

- Design management
- Construction and integration
- Testing and quality assurance
  - – component (class, object)
  - – subsystem (package)
  - – system

Cost estimation

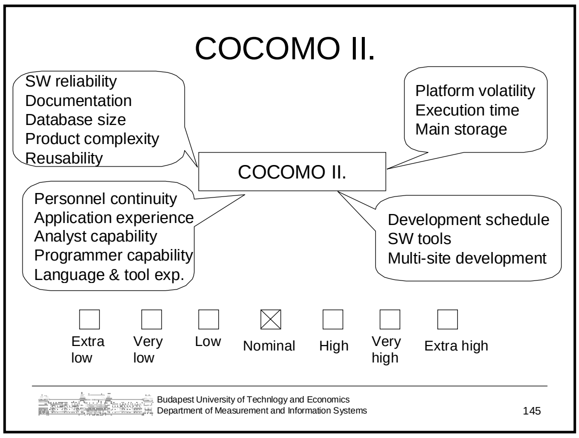
- Aim: prediction of development time
  - – different project phases
  - – different project type



COCOMO II.

$$PM=A\cdot(\prod^{17} Em_i)\cdot Size^B$$

- PM: Person Months
- A: invariant
- Em_i: Effort Multiplier
  - – product, human, technology, environment
- Size: source lines of code
- B: Scale factors
  - – development process



COCOMO II.

SW reliability
Documentation
Database size
Product complexity
Reusability

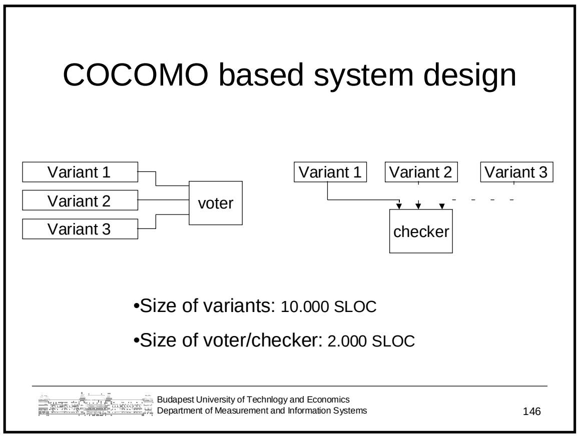
Platform volatility
Execution time
Main storage

COCOMO II.

Personnel continuity
Application experience
Analyst capability
Programmer capability
Language & tool exp.

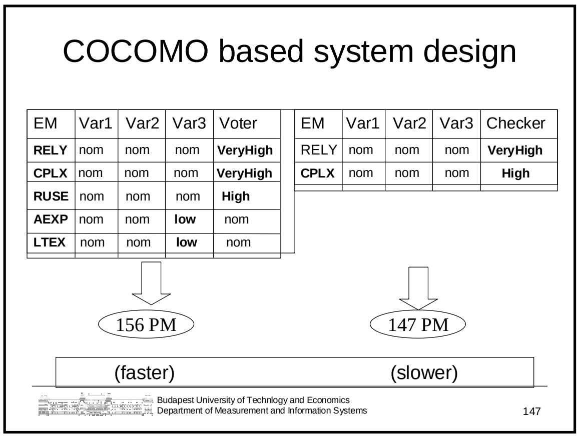
Development schedule
SW tools
Multi-site development

□ Extra low □ Very low □ Low ☒ Nominal □ High □ Very high □ Extra high



COCOMO based system design

- •Size of variants: 10.000 SLOC
- •Size of voter/checker: 2.000 SLOC



COCOMO based system design

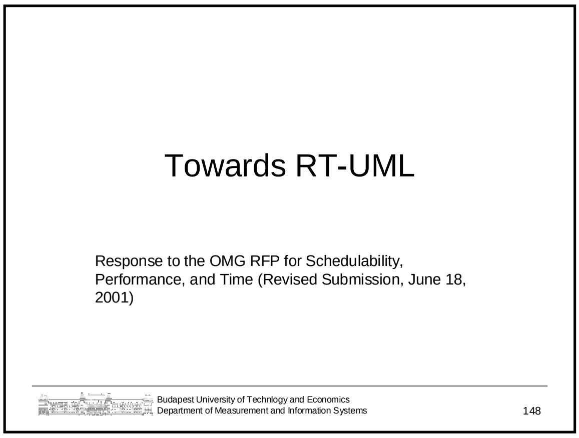
EM	Var1	Var2	Var3	Voter
RELY	nom	nom	nom	**VeryHigh**
CPLX	nom	nom	nom	**VeryHigh**
RUSE	nom	nom	nom	**High**
AEXP	nom	nom	**low**	nom
LTEX	nom	nom	**low**	nom

EM	Var1	Var2	Var3	Checker
RELY	nom	nom	nom	**VeryHigh**
CPLX	nom	nom	nom	**High**

156 PM

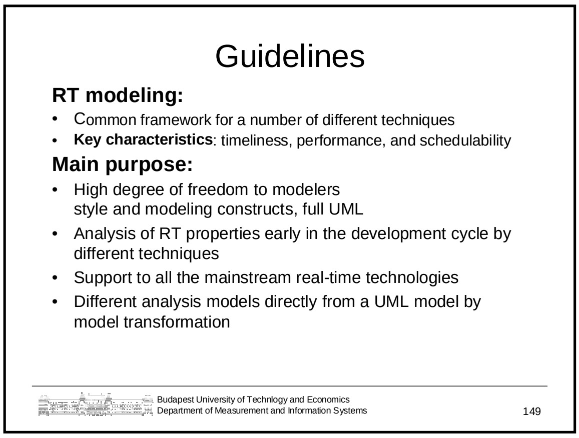
147 PM

(faster) (slower)



Towards RT-UML

Response to the OMG RFP for Schedulability, Performance, and Time (Revised Submission, June 18, 2001)



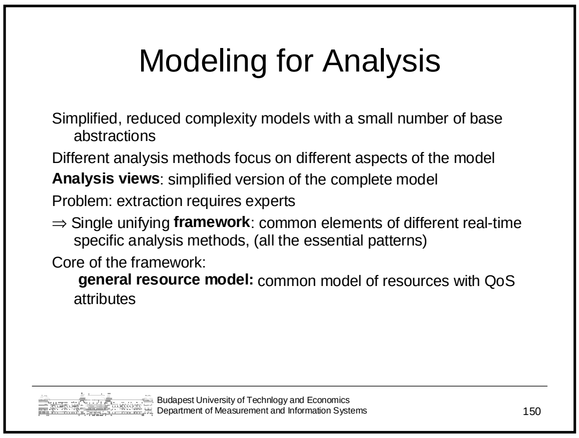
Guidelines

RT modeling:

- Common framework for a number of different techniques
- **Key characteristics**: timeliness, performance, and schedulability

Main purpose:

- High degree of freedom to modelers
  style and modeling constructs, full UML
- Analysis of RT properties early in the development cycle by different techniques
- Support to all the mainstream real-time technologies
- Different analysis models directly from a UML model by model transformation



Modeling for Analysis

Simplified, reduced complexity models with a small number of base abstractions

Different analysis methods focus on different aspects of the model

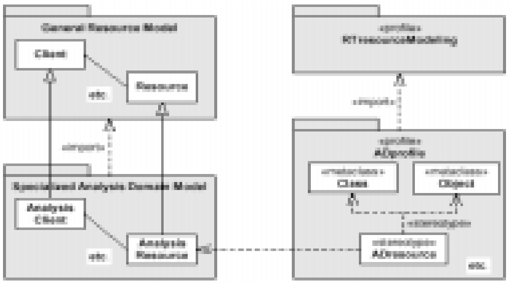
Analysis views: simplified version of the complete model

Problem: extraction requires experts

⇒ Single unifying **framework**: common elements of different real-time specific analysis methods, (all the essential patterns)

Core of the framework:
general resource model: common model of resources with QoS attributes

Framework



Modeling aspects

Resources:
- typical patterns present in many real-time analysis methods

Modeling time
- only metric time
- distinction between physical and simulated time

Scheduling ability
- focus on systems having hard timeliness requirements
- support for RMA, DMA, EDA, and other analysis methods

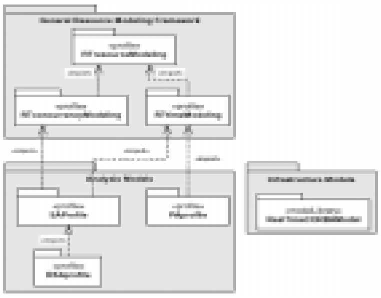
Performance
- stochastic modeling

Structure of the RT-Profile

Modularized

Suited for future extensions

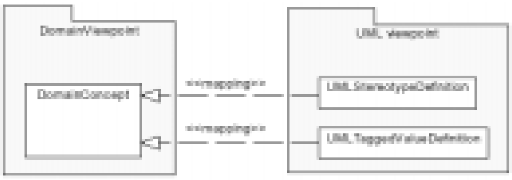
Unnecessary elements can be left out



General Resource Modeling (GRM)

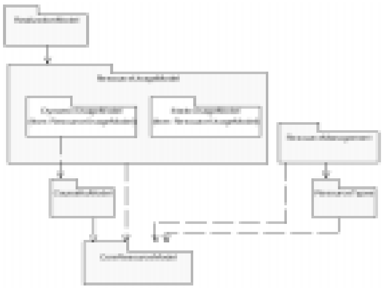
Essential framework for modeling real-time systems

Core: notion of QoS



GRM - Domain Viewpoint

Overall package structure described in UML:



GRM - UML Viewpoint

Domain concept ⇒ UML stereotype

- Modeling Realization Relationships
- UML Extensions
- Modeling Guidelines and Examples
- Required UML Metamodel Changes
- Proposed Notational Extensions

General Time Modeling (GTM)

General framework for representing time and time-related mechanisms

Cardinality of time

(delay, duration,clock time, ...) ⇒ **metric time** instead of logical time

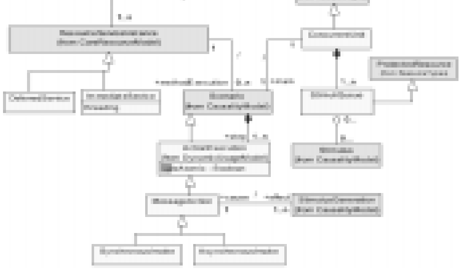
General Concurrency Modeling (GCM)

Primary purposes:

- It enables modelers to describe a rich enough **domain model of concurrently executing and communicating objects** that can serve as a base for more concrete analysis models.
- It enables providers of real-time system infrastructures (e.g., operating systems) **to describe the concurrency and communication mechanisms** of their system.

GCM - Domain Viewpoint

General concurrency modeling concepts:



Schedulability Modeling (SM)

Component of the proposed profile that is intended specifically for scheduling ability analysis

Minimal set of common scheduling annotations for very basic scheduling ability analysis

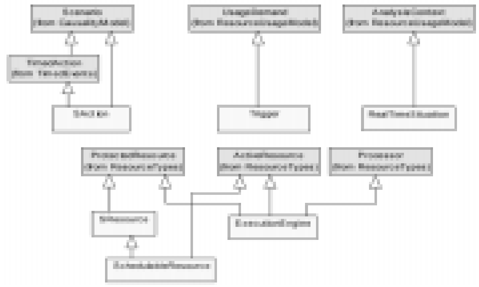
Designer’s aspect: analyze the system under several scenarios using different parameter values

2 important functions of typical tools:

- Calculate the scheduling ability of the system
- Assistance with determining how the system can be improved

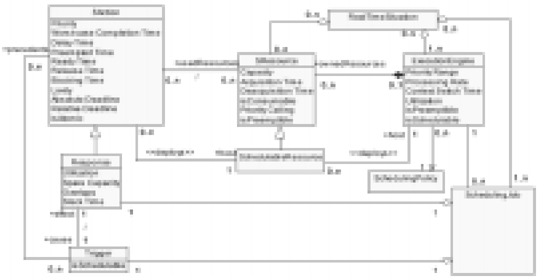
SM - Domain Viewpoint

The scheduling ability model derived form GRM:


Scenario (from CausalityModel)
UsageDemand (from ResourceUsageModel)
AnalysisContext (from ResourceUsageModel)
TimedAction (from TimedEvents)
SAction
Trigger
RealTimeSituation
ProtectedResource (from ResourceTypes)
ActiveResource (from ResourceTypes)
Processor (from ResourceTypes)
SResource
ExecutionEngine
SchedulableResource

SM - Domain Viewpoint

The core scheduling ability model:



Performance Modeling (PM)

A component of the profile that is intended for general performance analysis of UML models. Minimal set of concepts to support the central ideas of perf. analysis

Facilities for:

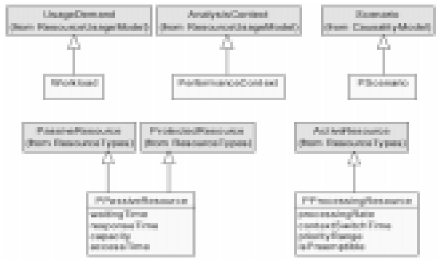
- capturing performance requirements
- associating performance-related QoS characteristics with selected elements of a UML model
- specifying execution parameters
- presenting performance results

Two important functions of typical tools:

- estimate the performance of a system instance
- assistance with determining how the system can be improved

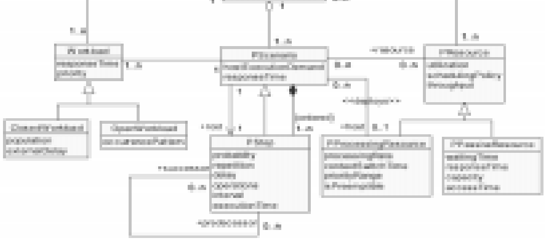
PM - Domain Viewpoint

Relationship between the performance concepts and the general resource model



PM - Domain Viewpoint

The performance analysis domain model



Real-time properties and OCL

S. Flake, W. Mueller. An OCL Extension for Real-Time Constraints.

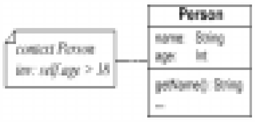
In: T. Clark, J. Warmer (eds.), Advances in Object Modelling with the OCL, (Springer, 2001. to appear)

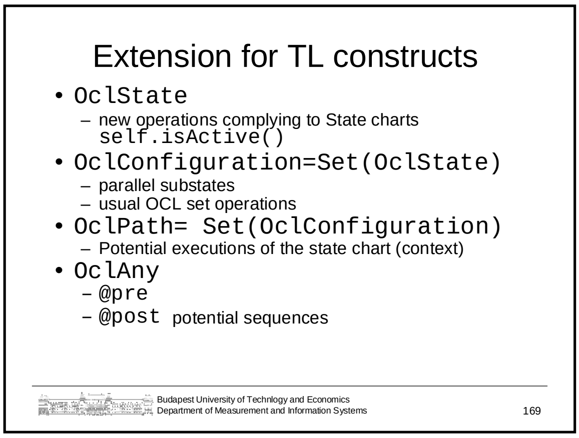
Problems in the practical use of temporal logic

- Hard to understand
  - additional temporal properties
  - not the entire expressive power is needed
- Approaches
  - graphic notations (sequence charts)
  - design patterns for requirement analysis
  - natural language (or artificial language = standards)
  - OCL

OCL

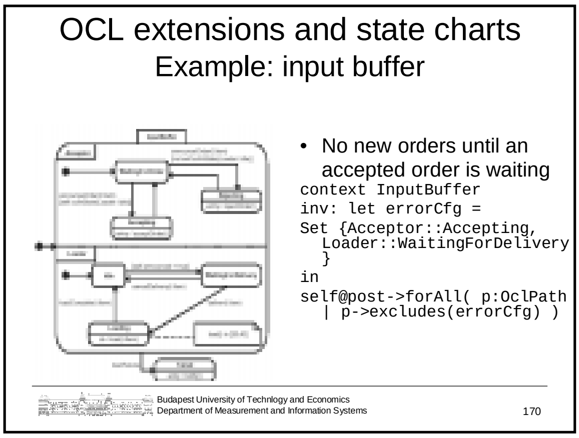
- Textual, programming language-like
- Defined for expressing pre-/post invariants of classes
- Sets, sequences, multi-sets
- Navigation
- No statements on states
- No temporal expressions





Extension for TL constructs

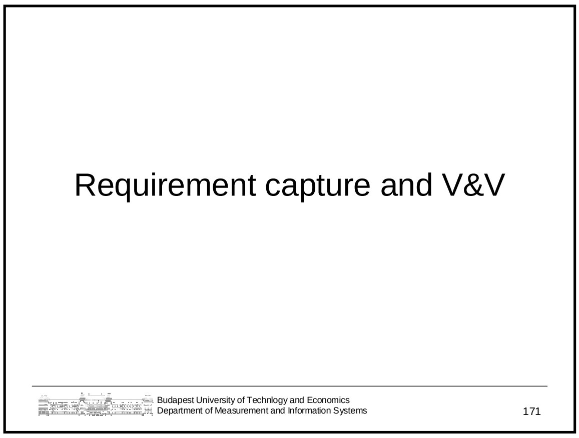
- `OclState`
  - new operations complying to State charts `self.isActive()`
- `OclConfiguration=Set(OclState)`
  - parallel substates
  - usual OCL set operations
- `OclPath= Set(OclConfiguration)`
  - Potential executions of the state chart (context)
- `OclAny`
  - `@pre`
  - `@post` potential sequences



OCL extensions and state charts
Example: input buffer

- No new orders until an accepted order is waiting

```
context InputBuffer
inv: let errorCfg =
Set {Acceptor::Accepting,
   Loader::WaitingForDelivery
   }
in
self@post->forAll( p:OclPath
   | p->excludes(errorCfg) )
```

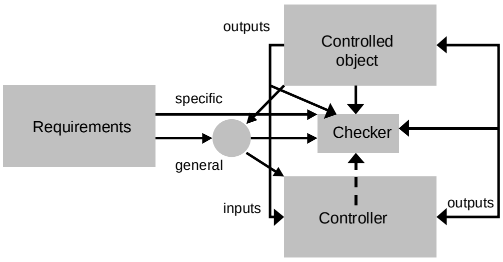


Requirement capture and V&V

Embedded systems

- Main problem: check the interaction of the system and its environment
- Needed:
  - Description of the environment (“controlled object”)
  - Description of the system under design (“controller”)
  - Requirements
  - Checking methods

V&V oriented system model



Basic objects

Environment

- Complete model - closed loop (abstraction? constraints by the environment !)
- Input/output behavior - half-open loop (scenario)
- Arbitrary - open loop (most pessimistic)

System under design

- Different UML models in different phases
  - black box
  - grey box
  - white box

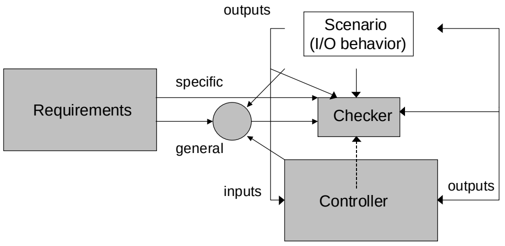
Requirements specification

Syntactic: completeness (static/dynamic)

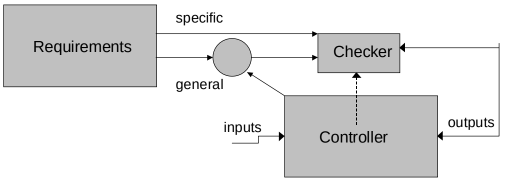
Semantic

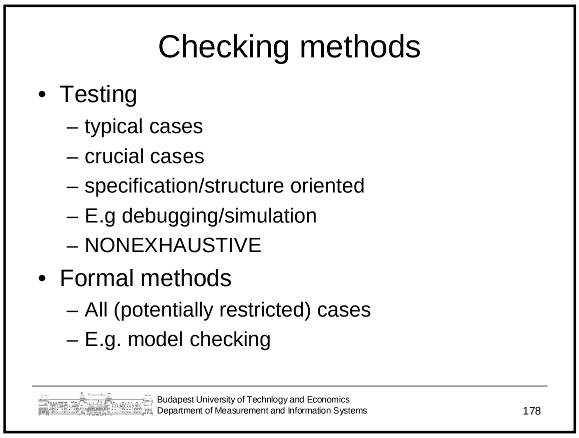
- General (no deadlocks)
  - – metamodel level definition
  - – OCL
  - – external: temporal logic
- Application-specific (e.g. safety constraints)
  - – external expression (e.g temporal logic)
  - – co-modeling (“observer”)

Requirement models



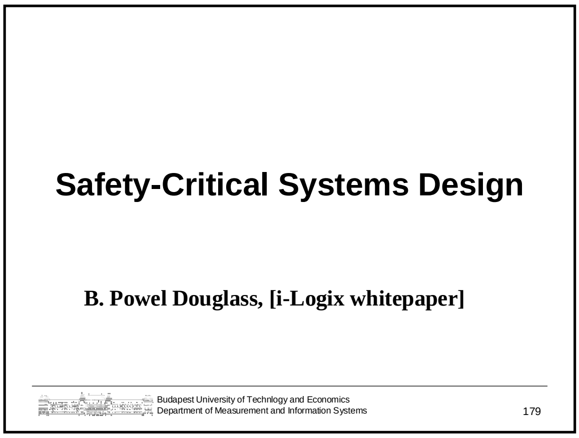
Open loop





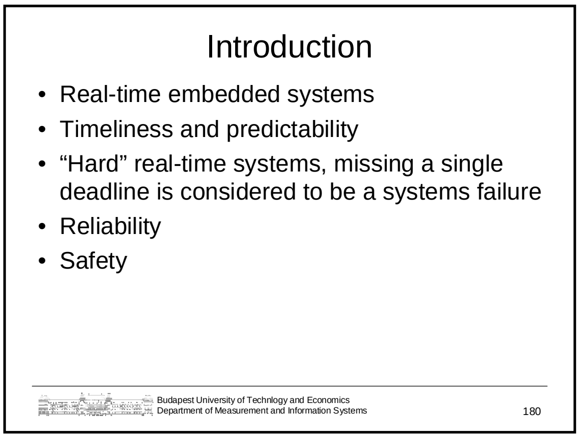
Checking methods

- Testing
  - typical cases
  - crucial cases
  - specification/structure oriented
  - E.g debugging/simulation
  - NONEXHAUSTIVE
- Formal methods
  - All (potentially restricted) cases
  - E.g. model checking



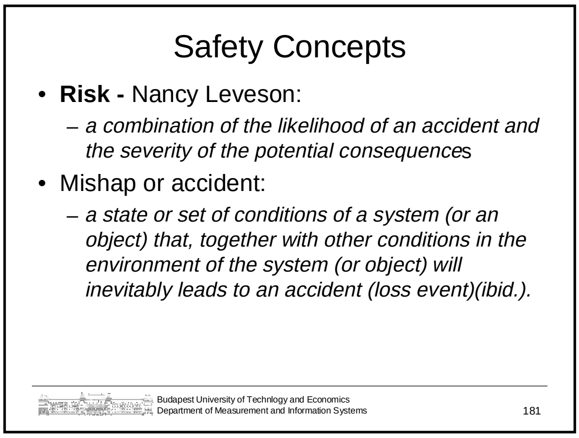
Safety-Critical Systems Design

B. Powel Douglass, [i-Logix whitepaper]



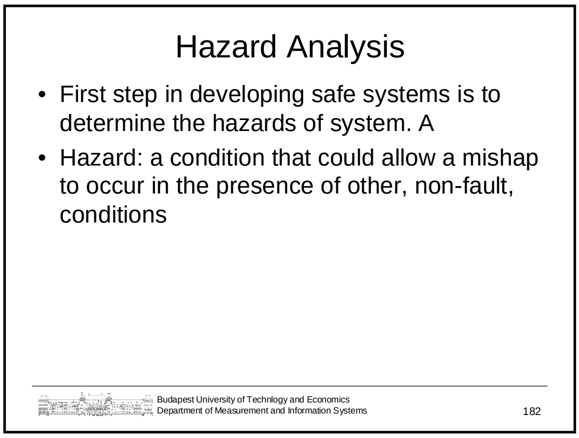
Introduction

- Real-time embedded systems
- Timeliness and predictability
- “Hard” real-time systems, missing a single deadline is considered to be a systems failure
- Reliability
- Safety



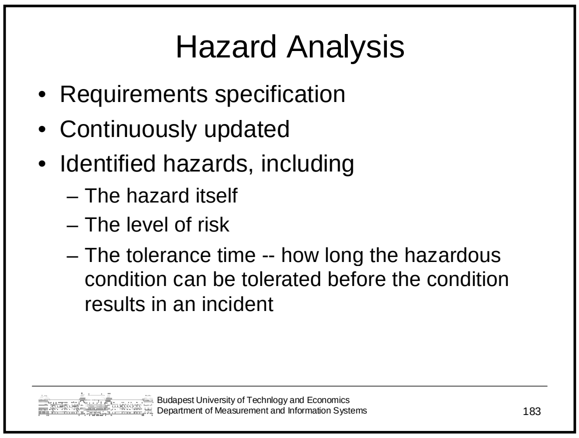
Safety Concepts

- **Risk -** Nancy Leveson:
  - *a combination of the likelihood of an accident and the severity of the potential consequence*s
- Mishap or accident:
  - *a state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object) will inevitably leads to an accident (loss event)(ibid.).*



Hazard Analysis

- First step in developing safe systems is to determine the hazards of system. A
- Hazard: a condition that could allow a mishap to occur in the presence of other, non-fault, conditions



Hazard Analysis

- Requirements specification
- Continuously updated
- Identified hazards, including
  - The hazard itself
  - The level of risk
  - The tolerance time -- how long the hazardous condition can be tolerated before the condition results in an incident

Hazard Analysis

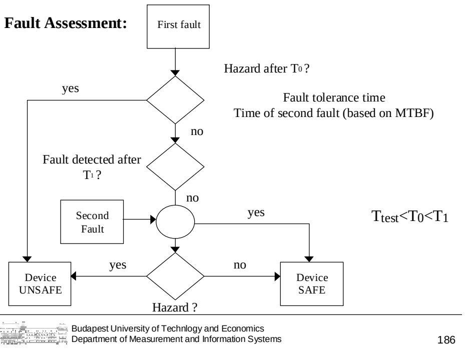
- Means by which the hazards can arise
  - – The fault leading to the hazard
  - – Likelihood of fault
  - – The fault detection time
- The means by which the hazards are handled
  - – The means
  - – The fault reaction (exposure) time

Hazard Handling

- Obviation
- Education
- Alarming
- Interlocks
- Internal checking
- Additional safety equipment
- Restricting access to potential hazards
- Labeling

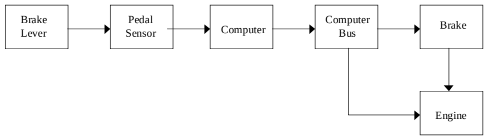
Single Point Failures - TUV

TUV Single Fault Assessment:



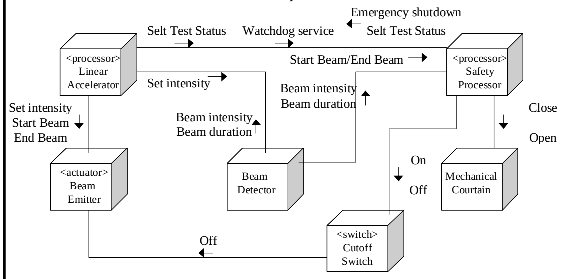
Safe Designs

Single Channel Protected Designs (SCPD)



Safe Designs

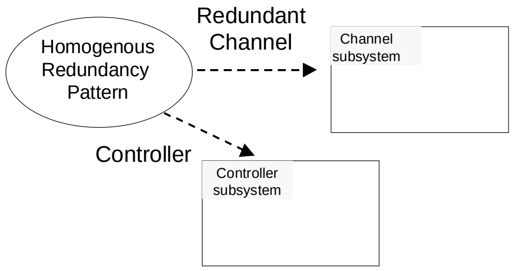
Dual Channel Designs (DCD)



Design Patterns for Reliability and Safety

- Homogeneous Redundancy Pattern
- Diverse Redundancy Pattern
- Monitor-Actuator Pattern
- Safety Executive Pattern

Homogeneous Redundancy Pattern (HRP)



Homogenous Redundancy Pattern

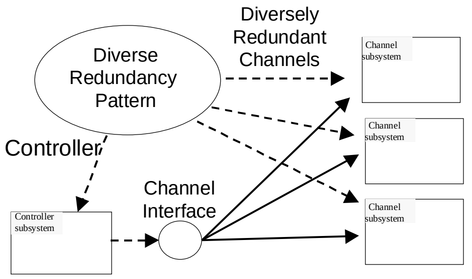
Redundant Channel

Channel subsystem

Controller

Controller subsystem

Diverse Redundancy Pattern (DRP)



Diverse Redundancy Pattern

Diversely Redundant Channels

Channel subsystem

Channel subsystem

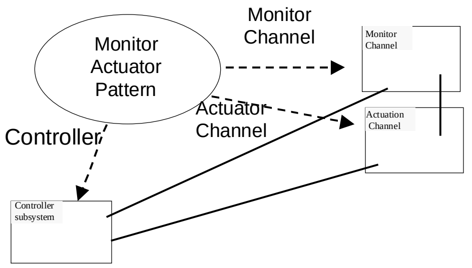
Channel subsystem

Controller

Channel Interface

Controller subsystem

Monitor-Actuator Pattern (MAP)



Monitor Actuator Pattern

Monitor Channel

Monitor Channel

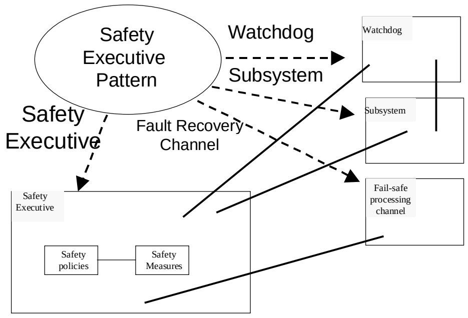
Actuator Channel

Actuation Channel

Controller

Controller subsystem

Safety Executive Pattern (SEP)

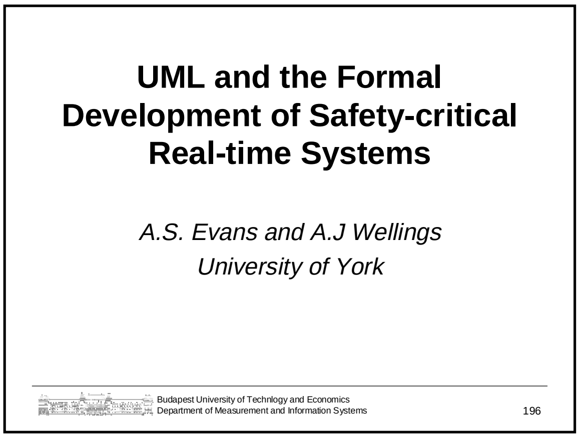


Implementing Designs Safely

- Language choice
  - – Compile-time checking (C vs. Ada)
  - – Run-time checking (C vs. Ada)
- Exceptions vs. error codes
- Use “safe” language subsets (e.g. avoiding void*)

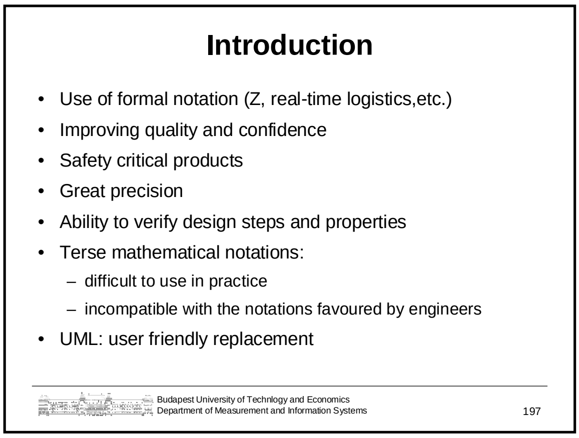
Testing For Safety

- Fault seeding:
  - – inducing (or simulating) faults impacting the safety of the system to ensure that the system acts in the safe, correct manner in the presence of those faults



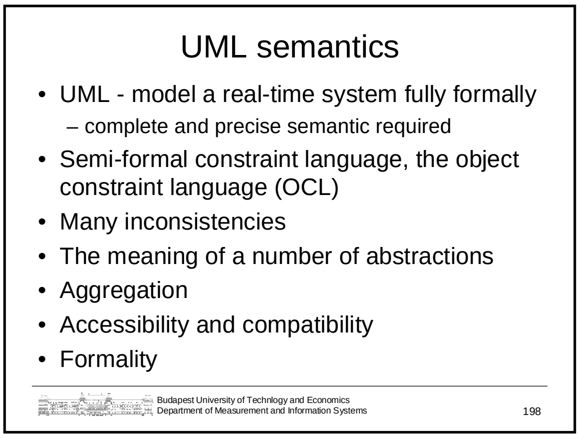
UML and the Formal Development of Safety-critical Real-time Systems

A.S. Evans and A.J Wellings
University of York



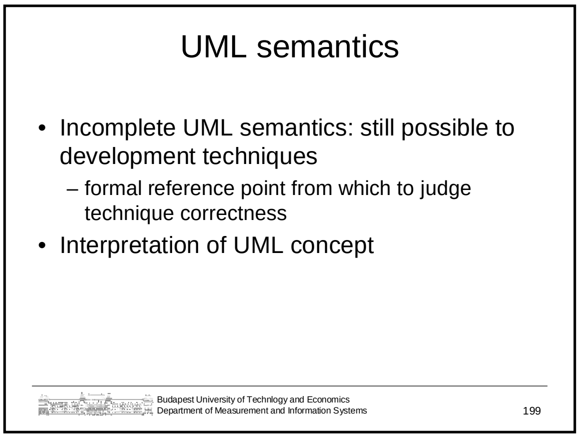
Introduction

- Use of formal notation (Z, real-time logistics,etc.)
- Improving quality and confidence
- Safety critical products
- Great precision
- Ability to verify design steps and properties
- Terse mathematical notations:
  - difficult to use in practice
  - incompatible with the notations favoured by engineers
- UML: user friendly replacement



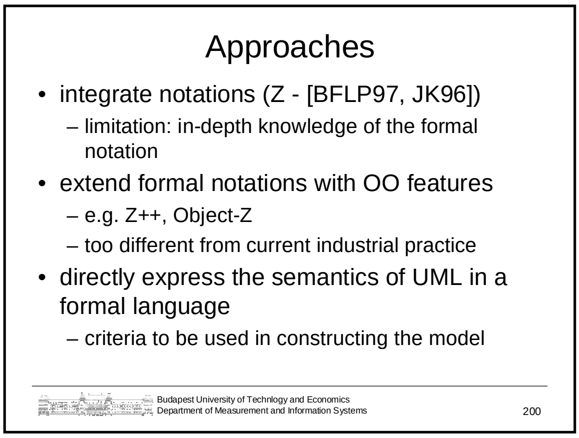
UML semantics

- UML - model a real-time system fully formally
  - complete and precise semantic required
- Semi-formal constraint language, the object constraint language (OCL)
- Many inconsistencies
- The meaning of a number of abstractions
- Aggregation
- Accessibility and compatibility
- Formality



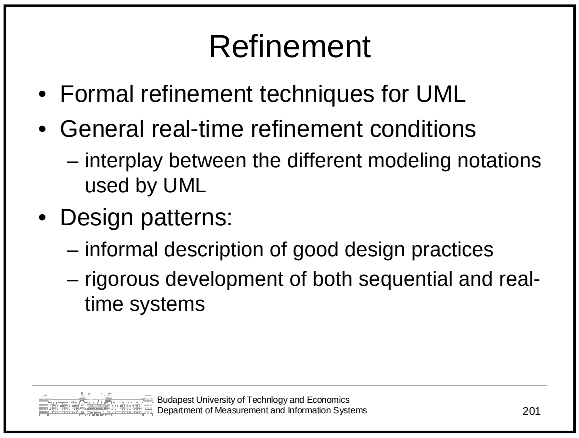
UML semantics

- Incomplete UML semantics: still possible to development techniques
  - formal reference point from which to judge technique correctness
- Interpretation of UML concept



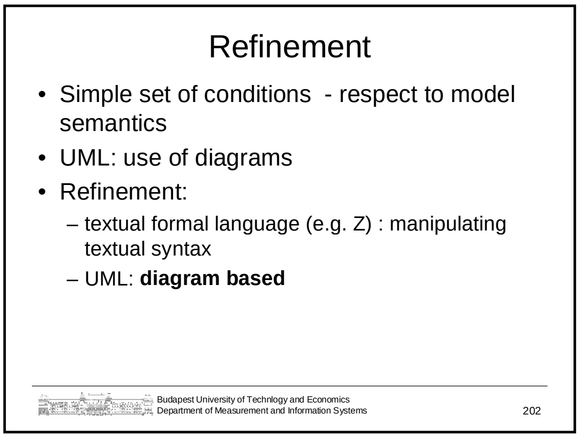
Approaches

- integrate notations (Z - [BFLP97, JK96])
  - limitation: in-depth knowledge of the formal notation
- extend formal notations with OO features
  - e.g. Z++, Object-Z
  - too different from current industrial practice
- directly express the semantics of UML in a formal language
  - criteria to be used in constructing the model



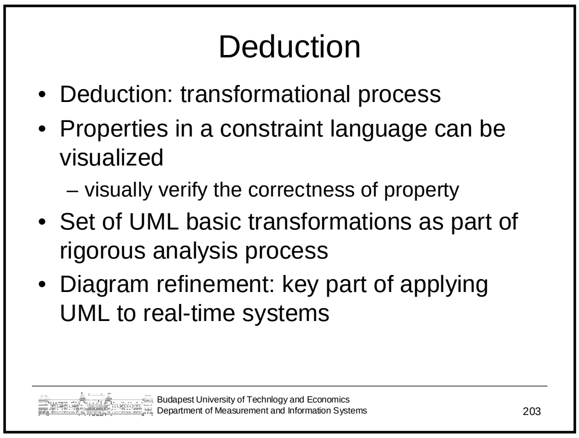
Refinement

- Formal refinement techniques for UML
- General real-time refinement conditions
  - interplay between the different modeling notations used by UML
- Design patterns:
  - informal description of good design practices
  - rigorous development of both sequential and real-time systems



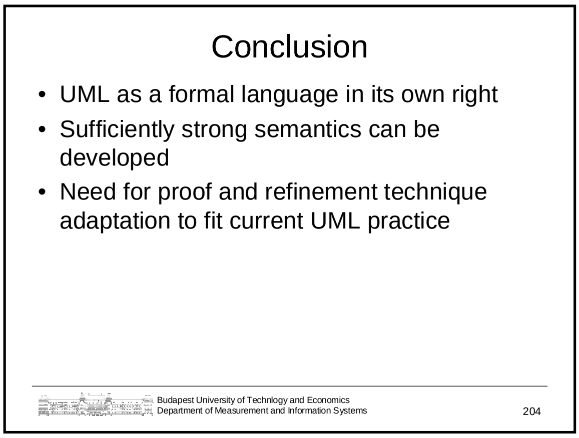
Refinement

- Simple set of conditions - respect to model semantics
- UML: use of diagrams
- Refinement:
  - textual formal language (e.g. Z) : manipulating textual syntax
  - UML: **diagram based**



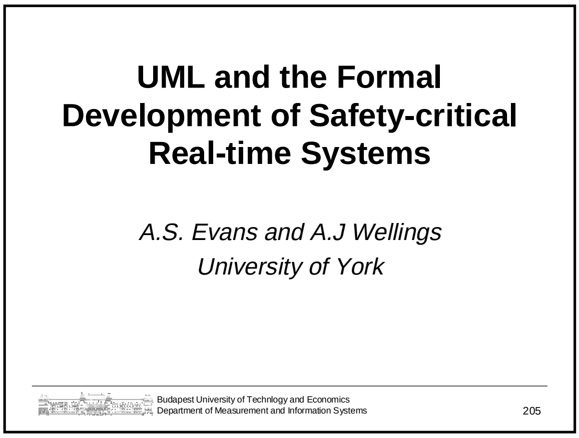
Deduction

- Deduction: transformational process
- Properties in a constraint language can be visualized
  - visually verify the correctness of property
- Set of UML basic transformations as part of rigorous analysis process
- Diagram refinement: key part of applying UML to real-time systems



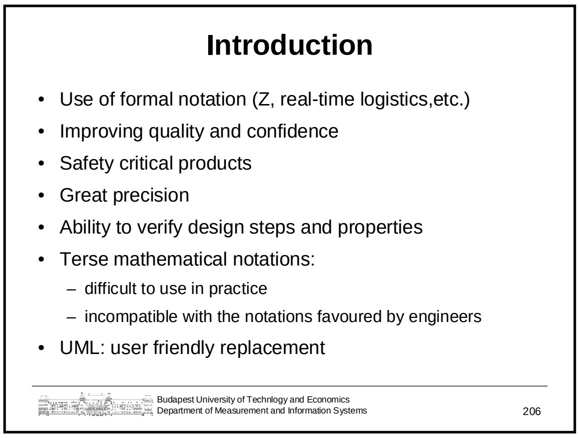
Conclusion

- UML as a formal language in its own right
- Sufficiently strong semantics can be developed
- Need for proof and refinement technique adaptation to fit current UML practice



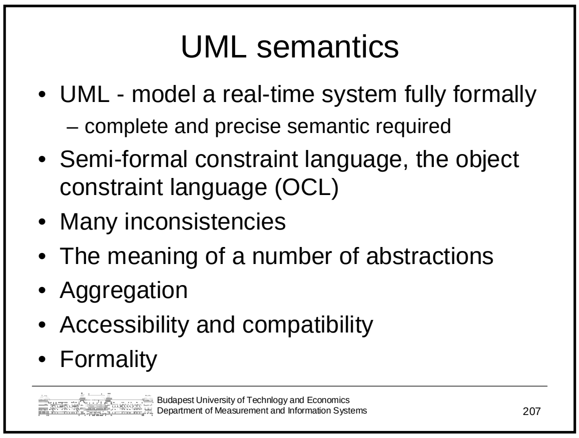
UML and the Formal Development of Safety-critical Real-time Systems

A.S. Evans and A.J Wellings
University of York



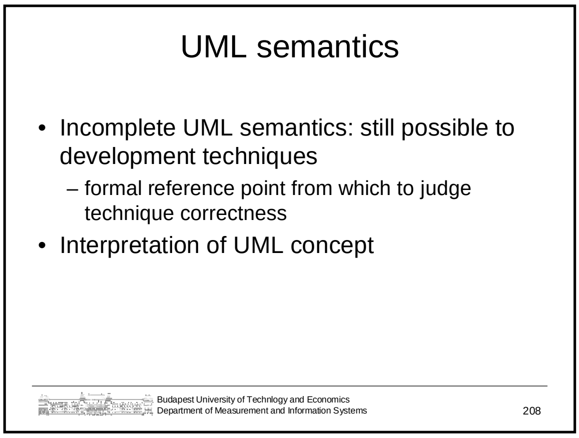
Introduction

- Use of formal notation (Z, real-time logistics,etc.)
- Improving quality and confidence
- Safety critical products
- Great precision
- Ability to verify design steps and properties
- Terse mathematical notations:
  - difficult to use in practice
  - incompatible with the notations favoured by engineers
- UML: user friendly replacement



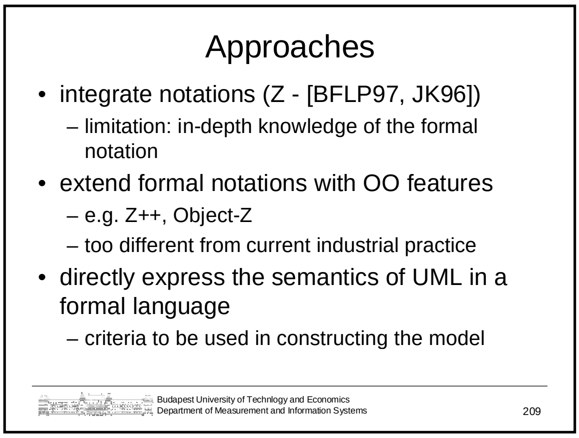
UML semantics

- UML - model a real-time system fully formally
  - complete and precise semantic required
- Semi-formal constraint language, the object constraint language (OCL)
- Many inconsistencies
- The meaning of a number of abstractions
- Aggregation
- Accessibility and compatibility
- Formality



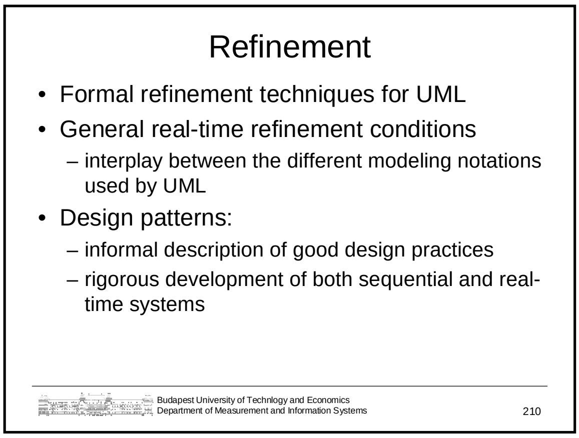
UML semantics

- Incomplete UML semantics: still possible to development techniques
  - formal reference point from which to judge technique correctness
- Interpretation of UML concept



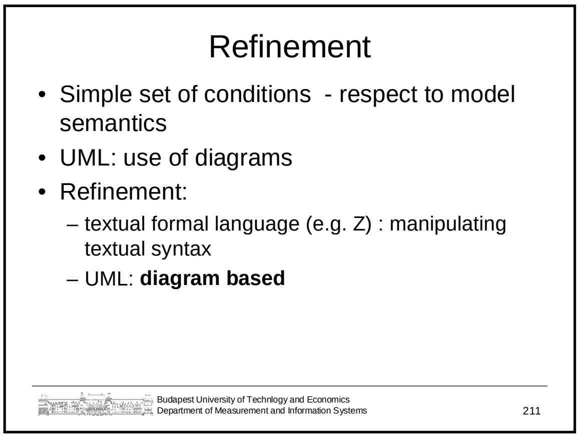
Approaches

- integrate notations (Z - [BFLP97, JK96])
  - limitation: in-depth knowledge of the formal notation
- extend formal notations with OO features
  - e.g. Z++, Object-Z
  - too different from current industrial practice
- directly express the semantics of UML in a formal language
  - criteria to be used in constructing the model



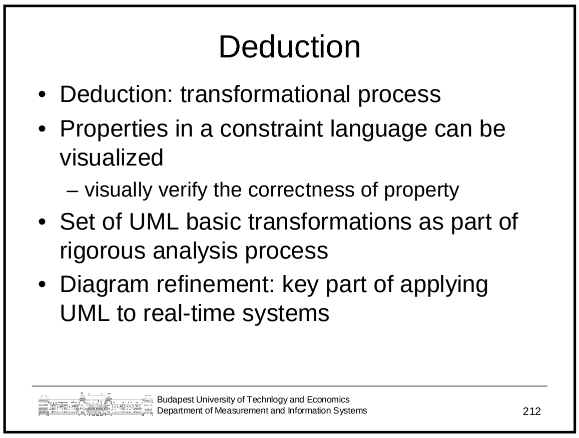
Refinement

- Formal refinement techniques for UML
- General real-time refinement conditions
  - interplay between the different modeling notations used by UML
- Design patterns:
  - informal description of good design practices
  - rigorous development of both sequential and real-time systems



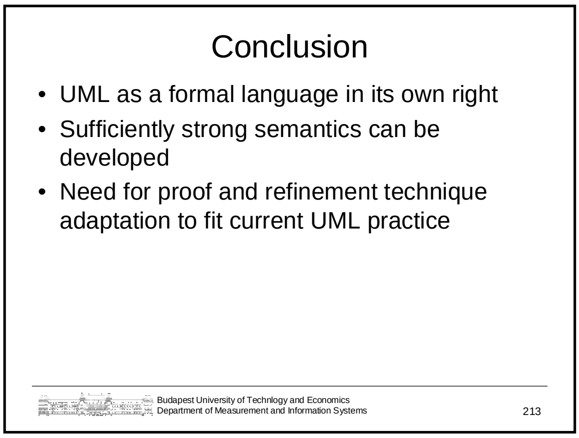
Refinement

- Simple set of conditions - respect to model semantics
- UML: use of diagrams
- Refinement:
  - textual formal language (e.g. Z) : manipulating textual syntax
  - UML: **diagram based**



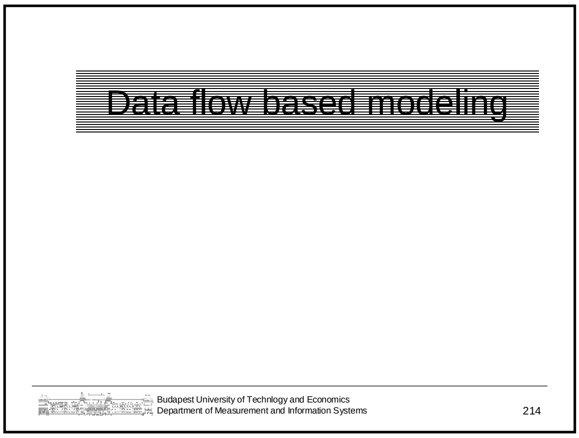
Deduction

- Deduction: transformational process
- Properties in a constraint language can be visualized
  - visually verify the correctness of property
- Set of UML basic transformations as part of rigorous analysis process
- Diagram refinement: key part of applying UML to real-time systems

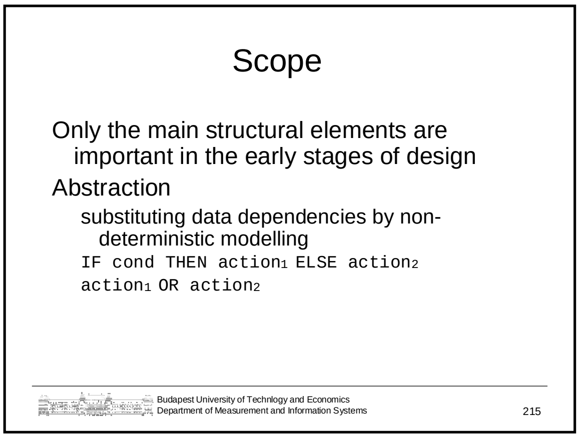


Conclusion

- UML as a formal language in its own right
- Sufficiently strong semantics can be developed
- Need for proof and refinement technique adaptation to fit current UML practice



Data flow based modeling

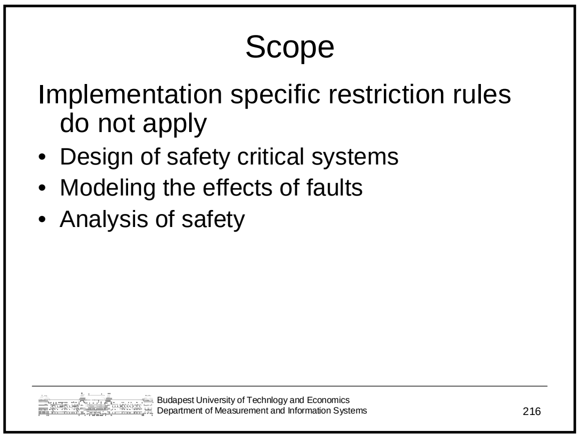


Scope

Only the main structural elements are important in the early stages of design

Abstraction

- substituting data dependencies by non-deterministic modelling
- `IF cond THEN action`$_1$ `ELSE action`$_2$
- `action`$_1$ `OR action`$_2$

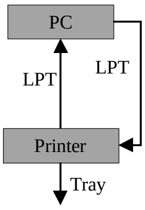


Scope

Implementation specific restriction rules do not apply

- Design of safety critical systems
- Modeling the effects of faults
- Analysis of safety

Theory of data flow networks



Node: representation of a HW/SW component

Channel: directed FIFO connection between nodes

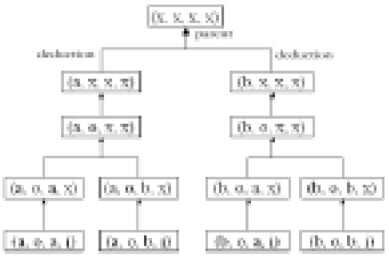
Token: representation of data

Node functionality: firing rules

Systematic generation of firing rules

- Example: Printer
  (Ready, LPT→doc, Busy, Tray→paper, 0)
- A large set of firing rules can not be handled efficiently, algorithmization is needed
- Deduction tree: ordered structure of the firing rules built by narrowing
- DON’T CARE component: x

Deduction tree



Examining the completeness of the firing rule set

During the design process warning about omitted firings

ROBDD: graph based structure, which is capable of efficiently generating the normal form of Boolean expressions

Steps of the algorithm:

- Read of the missing firings from the nodes
- Store them in a ROBDD structure
- Read the normal form from the ROBDD

Fault modeling with data flow networks

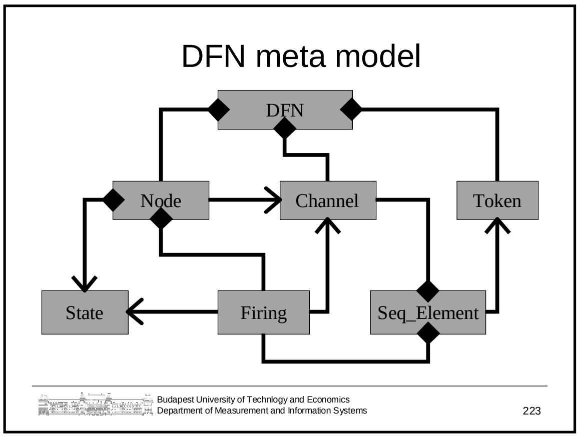
- Qualitative modelling
- System behavior is described by firing rules
- Fault modeling in separate phase without any restriction

Example: fault propagating behavior of the Printer

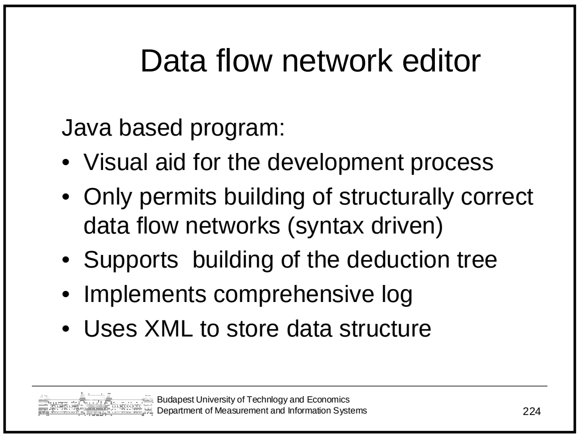
(Ready, LPT→fty_doc, Busy, Tray→fty_paper, 0)

Meta modeling

- Design and documentation is essential, therefore required
- Meta modeling is capable of modeling data as well as program structure
- Effective way of visual development
- Example: Dataflow UML model, DFN metamodel



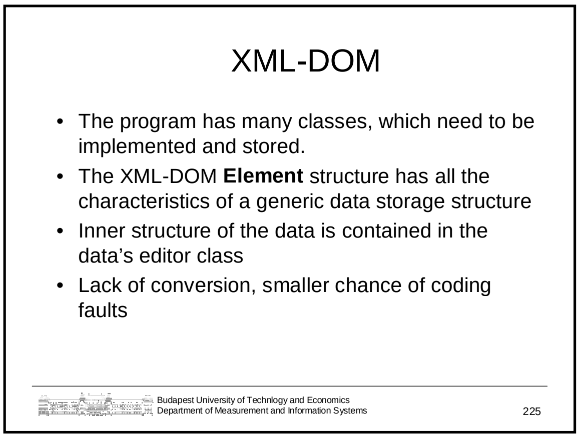
DFN meta model



Data flow network editor

Java based program:

- Visual aid for the development process
- Only permits building of structurally correct data flow networks (syntax driven)
- Supports building of the deduction tree
- Implements comprehensive log
- Uses XML to store data structure



XML-DOM

- The program has many classes, which need to be implemented and stored.
- The XML-DOM **Element** structure has all the characteristics of a generic data storage structure
- Inner structure of the data is contained in the data’s editor class
- Lack of conversion, smaller chance of coding faults

Dataflow.java

- 32 days, 5361 lines, 177 Kbyte
- Structured design: tested modules, well defined connections
- Benefit of meta modeling : easy implementation of extensions
  - Implementation of “DON’T CARE”

Testing of UML designs

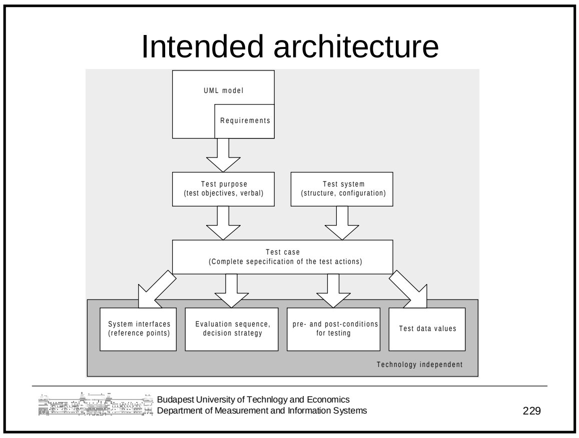
OMG UML Testing Profile RFP

Issued: July 13, 2001

Deadline: June 3, 2002

Objective

- Computational UML models ⇒ conformance requirements (functional black-box test cases)
  - typically use-case driven
  - functional requirements (testing and certification)
- Test specific concepts basis :
  - UML metamodel or MOF-based meta-model.
- Exchange of test specifications between tools: XMI



Intended architecture

UML model

Requirements

Test purpose
(test objectives, verbal)

Test system
(structure, configuration)

Test case
(Complete sepecification of the test actions)

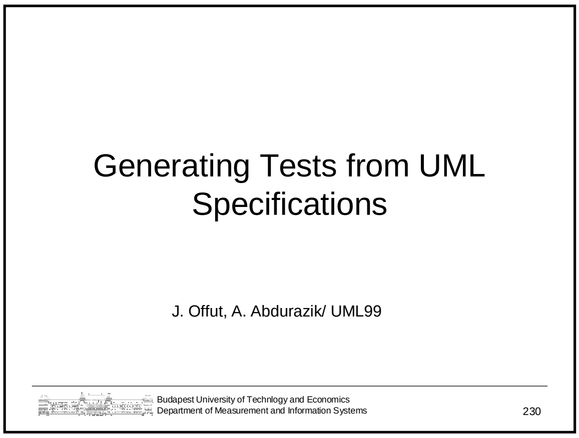
System interfaces
(reference points)

Evaluation sequence,
decision strategy

pre- and post-conditions
for testing

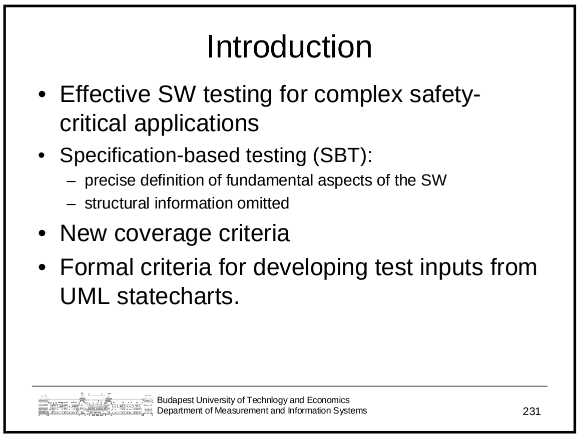
Test data values

Technology independent



Generating Tests from UML Specifications

J. Offut, A. Abdurazik/ UML99



Introduction

- Effective SW testing for complex safety-critical applications
- Specification-based testing (SBT):
  - precise definition of fundamental aspects of the SW
  - structural information omitted
- New coverage criteria
- Formal criteria for developing test inputs from UML statecharts.

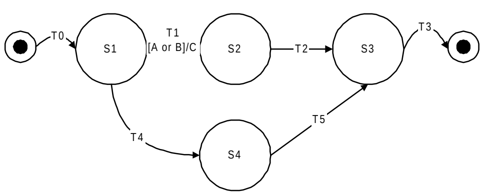
UML Based Test Data Generation

- UML categorizes transitions into five types
  - Only interested in enabled transitions
  - Similar to transitions based on predicate satisfaction
- Four kinds of events
  - Change events as predicates, (basis for generating tests)
    - (associated predicate← true) ⇒ Raised implicitly
    - Evaluated continuously until it becomes true

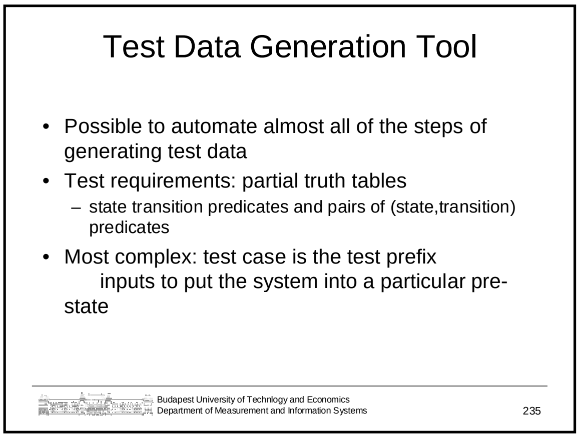
Levels of testing

- Coverage levels for change event enabled transitions:
  - transition (“branch”)
  - full predicate (“branch+selector”)
  - transition-pair (“2 step path”)
  - complete sequence (“path”)
- Choose a level: cost/benefit tradeoff

Statecharts

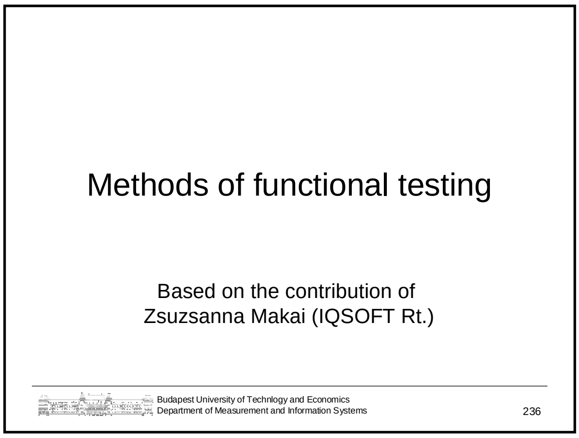


Transition: {<T0>; <T1>; <T2>; <T3>; <T4>; <T5>}
Predicate: {<T0>; <T1|A and not B >; <T1|not A and B > ; <T2>; <T3>; <T4>; <T5>}
Transition pair: {<T0,T1>; <T1,T2>; <T2,T3>; <T0, T4>; <T4,T5>; <T5,T3>}
Complete sequence:{<T0,T1,T2,T3>; <T0, T4,T5,T3>}



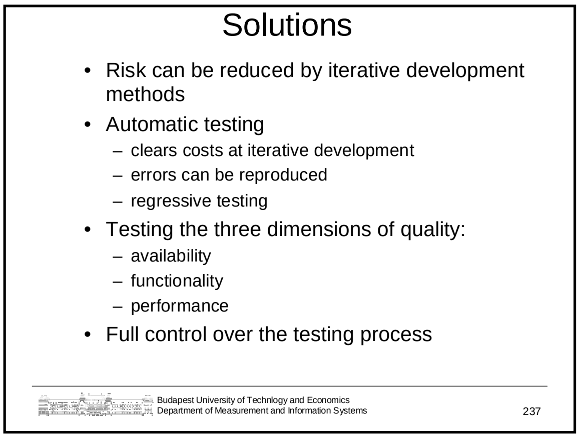
Test Data Generation Tool

- Possible to automate almost all of the steps of generating test data
- Test requirements: partial truth tables
  - state transition predicates and pairs of (state,transition) predicates
- Most complex: test case is the test prefix
  inputs to put the system into a particular pre-state



Methods of functional testing

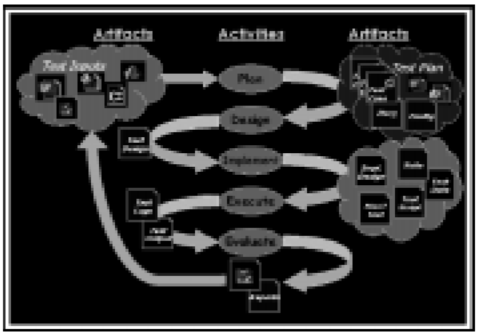
Based on the contribution of
Zsuzsanna Makai (IQSOFT Rt.)



Solutions

- Risk can be reduced by iterative development methods
- Automatic testing
  - clears costs at iterative development
  - errors can be reproduced
  - regressive testing
- Testing the three dimensions of quality:
  - availability
  - functionality
  - performance
- Full control over the testing process

Workflow of testing



Test inputs

- Any component that may affect testing
- Define WHAT to test

Definition of own inputs:

Excel table

Demand of change

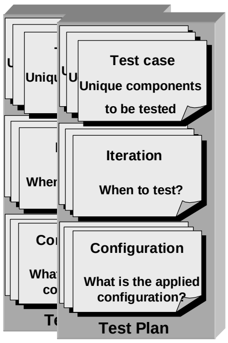
? "Custom input"

Built-in inputs:

ReqPro requirement

Rose model components

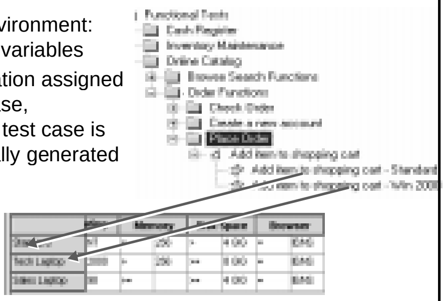
Test schemes


Test case
Unique components to be tested
Iteration
When to test?
Configuration
What is the applied configuration?
Test Plan

- Test scheme
  - a testing task
- Test case
  - related to test input
  - defines components to be tested
- Iteration
  - development phases
  - defines testing date
- Configuration
  - e.g.. operating system, browser, etc.

Configuration

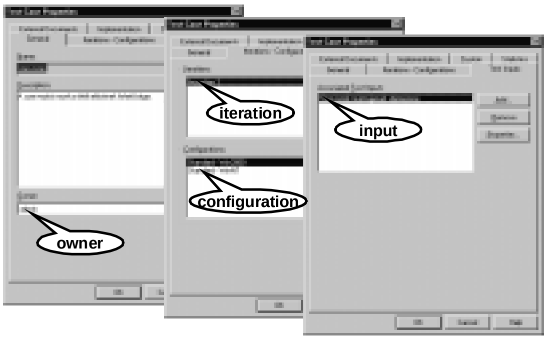
- Testing environment: defined by variables
- If configuration assigned to a test case, configured test case is automatically generated



Functional Tests
Cash Register
Inventory Maintenance
Online Catalog
Browse Search Functions
Order Functions
Check Order
Create a new account
Place Order
Add item to shopping cart
Add item to shopping cart - Standard
Add item to shopping cart - Win 2000

	[illegible]	Memory	[illegible] Space	Browser
Standard	[illegible]	[illegible]	[illegible]	[illegible]
Tech Laptop	[illegible]	[illegible]	[illegible]	[illegible]
[illegible] Laptop	[illegible]	[illegible]	[illegible]	[illegible]

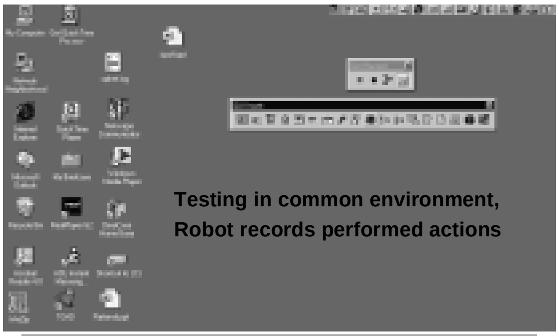
Test case properties



Testing scenarios

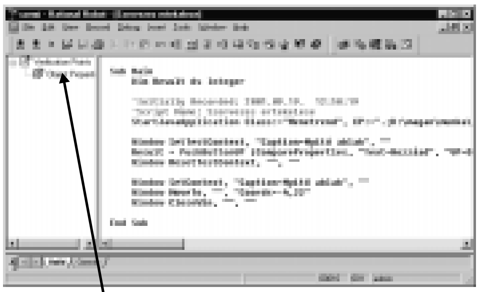
- Steps to be executed
- Verification points
- Pre- and after-conditions
- Acceptance criteria
- Implementation of testing
  - Manual testing
  - Script definition with Robot
  - Unique implementation of test scripts

Script record



Testing in common environment, Robot records performed actions

Recorded script



```
Sub Main
    Dim Result As Integer

    [illegible]
    [illegible]
    [illegible]

    [illegible]
    [illegible]
    [illegible]

    [illegible]
    [illegible]
    [illegible]

End Sub
```

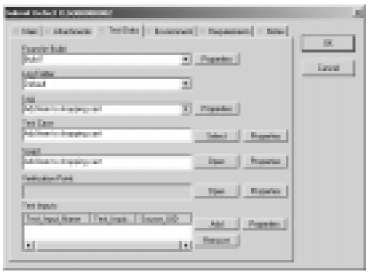
Checkpoints

Test execution

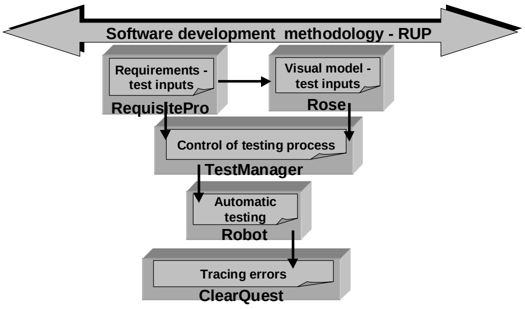
- Executable: implemented test cases, scripts, suites
- Execution of suites including several script types available (manual, GUI, VU etc.)
- Arbitrary scripts may be executed by creating adapters

Tracing reported errors

- TestManager assigns actual products to reported errors:
  - Log
  - Test Case
  - Test Script
  - Checkpoint
  - Test Input



Integrated tools


Software development methodology - RUP
Requirements - test inputs
RequisitePro
Visual model - test inputs
Rose
Control of testing process
TestManager
Automatic testing
Robot
Tracing errors
ClearQuest

Testing Embedded Systems

V. Encontre (Rational whitepaper 6/29/2001)

Introduction

- Testing:
  - disciplined process
  - behavior, performance and robustness match expected criteria
  - main criteria: as defect-free as possible
  - formally described and measurable
  - debugging only part of the testing process

RT systems

- Correctness: logical + temporal correctness (safety-critical system)
- Separation between application development and execution platforms
  - A large variety of execution platforms ⇒ cross-development environments
  - Coexistence of various implementation paradigms

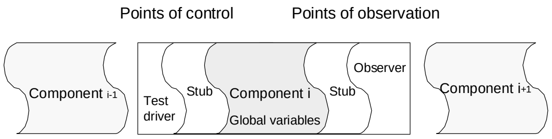
Testability and measurability

- 50%+ of embedded systems development projects are months behind schedule
- 44% of designs are within 20% of feature and performance expectations
- 50%+ of total development effort is spent in testing

Generic test iteration steps

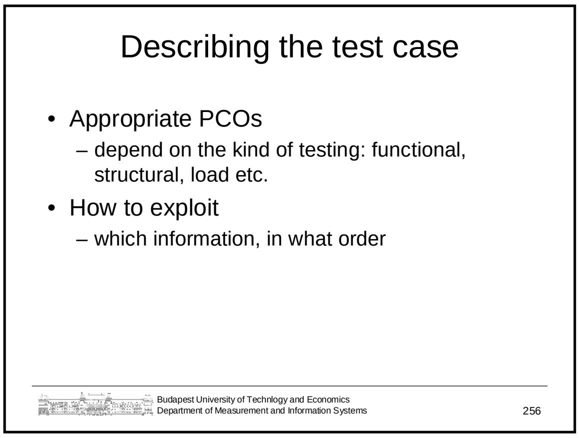
- Identify the granule to be tested
- Root for granularity
- Transform it to a testable granule or granule under test (GuT):
  - isolating the granule from its environment: *stubs, adapters*
  - stub: piece of code that simulates 2-way access between the granule and the rest of the application
  - *test drive*r: measures output, then compares it

Test harness environment



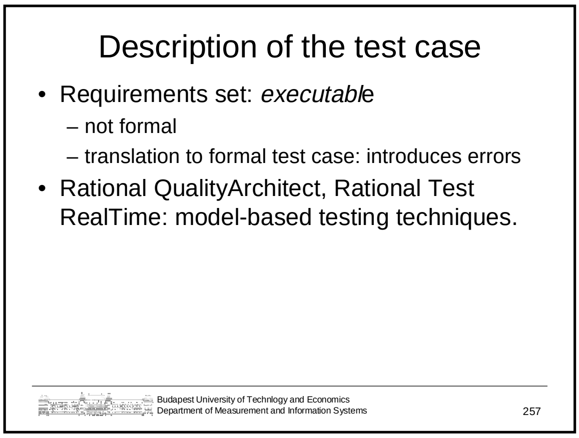
GuT paths

- *Points of Control and Observation* (PCOs)
  - at the border of, or inside the GuT
- PO inside the granule: coverage of a specific line of code in the function
- PO at the border of the granule: parameter values returned by the function
- PC inside the granule: change of a local variable
- PC at the border of the granule: the function call with actual parameters



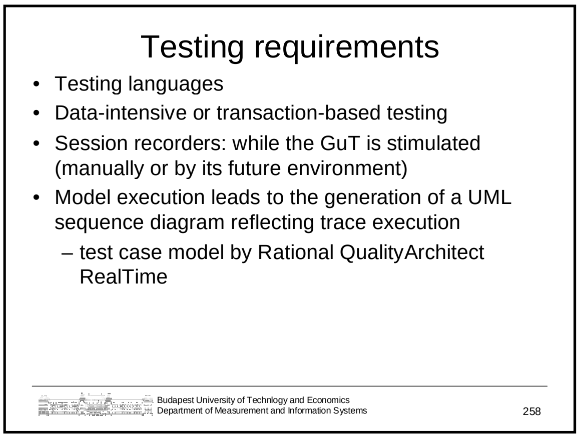
Describing the test case

- Appropriate PCOs
  - – depend on the kind of testing: functional, structural, load etc.
- How to exploit
  - – which information, in what order



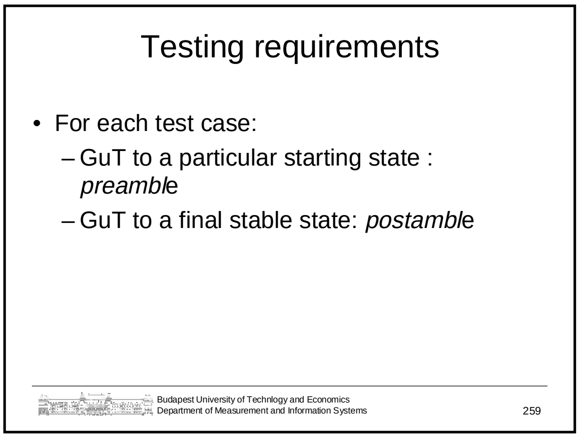
Description of the test case

- Requirements set: *executabl*e
  - – not formal
  - – translation to formal test case: introduces errors
- Rational QualityArchitect, Rational Test RealTime: model-based testing techniques.



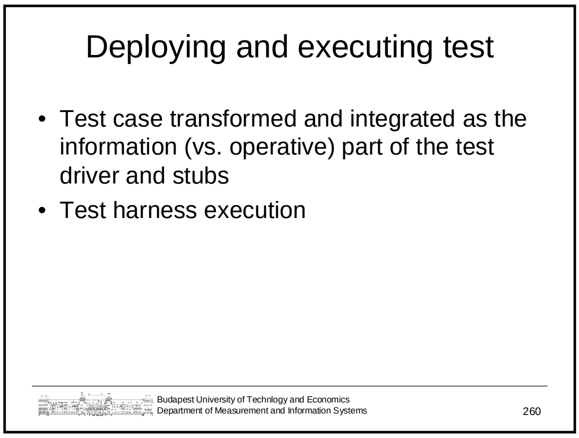
Testing requirements

- Testing languages
- Data-intensive or transaction-based testing
- Session recorders: while the GuT is stimulated (manually or by its future environment)
- Model execution leads to the generation of a UML sequence diagram reflecting trace execution
  - – test case model by Rational QualityArchitect RealTime



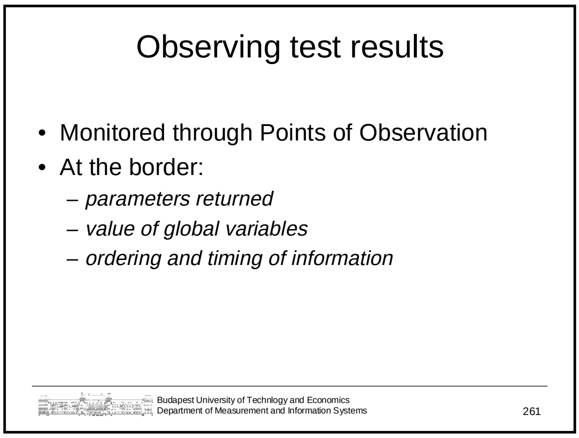
Testing requirements

- For each test case:
  - GuT to a particular starting state : *preambl*e
  - GuT to a final stable state: *postambl*e



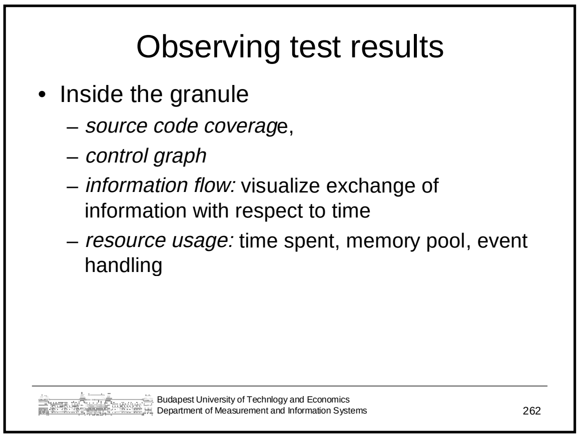
Deploying and executing test

- Test case transformed and integrated as the information (vs. operative) part of the test driver and stubs
- Test harness execution



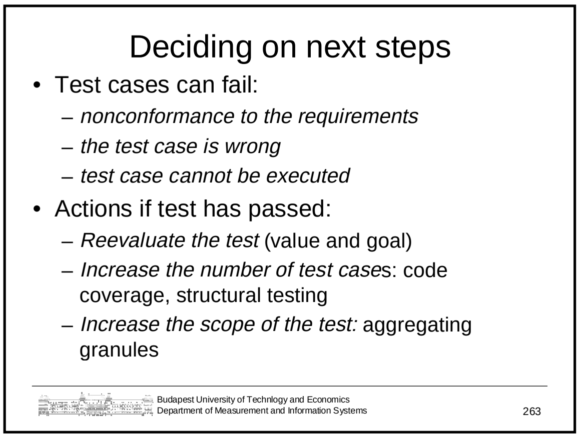
Observing test results

- Monitored through Points of Observation
- At the border:
  - *parameters returned*
  - *value of global variables*
  - *ordering and timing of information*



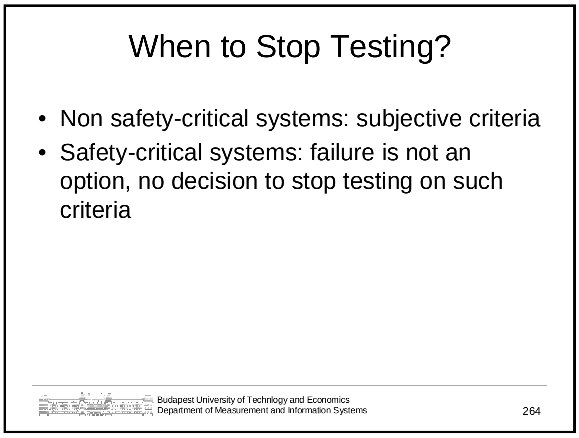
Observing test results

- Inside the granule
  - *source code coverag*e,
  - *control graph*
  - *information flow:* visualize exchange of information with respect to time
  - *resource usage:* time spent, memory pool, event handling



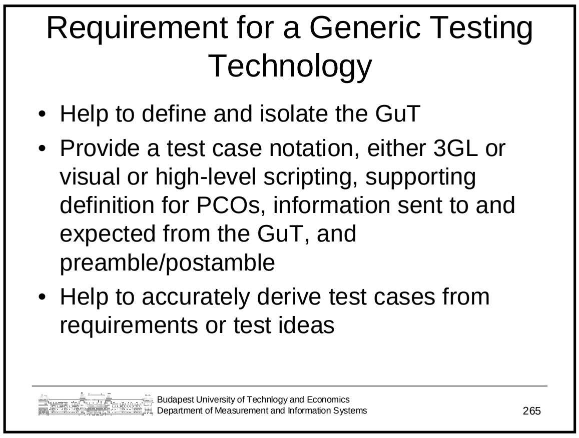
Deciding on next steps

- Test cases can fail:
  - *nonconformance to the requirements*
  - *the test case is wrong*
  - *test case cannot be executed*
- Actions if test has passed:
  - *Reevaluate the test* (value and goal)
  - *Increase the number of test case*s: code coverage, structural testing
  - *Increase the scope of the test:* aggregating granules



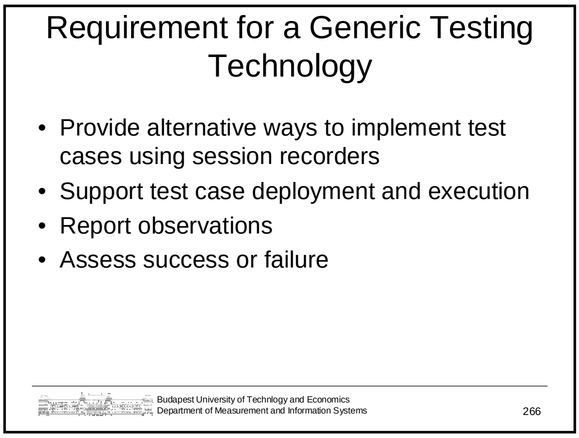
When to Stop Testing?

- Non safety-critical systems: subjective criteria
- Safety-critical systems: failure is not an option, no decision to stop testing on such criteria



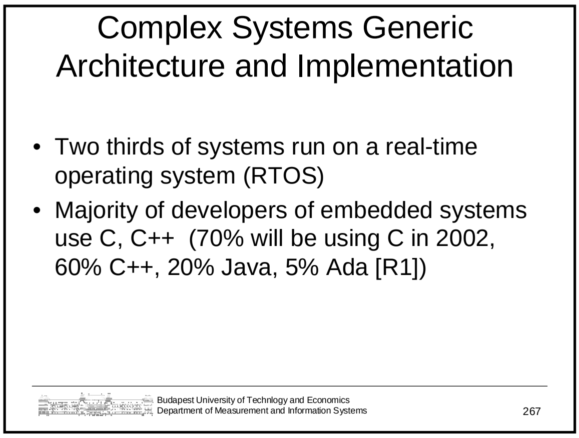
Requirement for a Generic Testing Technology

- Help to define and isolate the GuT
- Provide a test case notation, either 3GL or visual or high-level scripting, supporting definition for PCOs, information sent to and expected from the GuT, and preamble/postamble
- Help to accurately derive test cases from requirements or test ideas



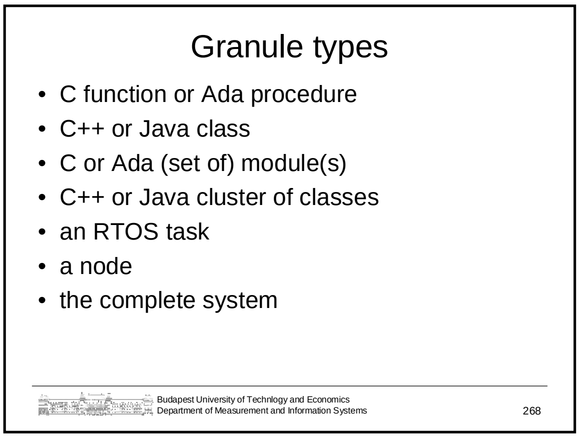
Requirement for a Generic Testing Technology

- Provide alternative ways to implement test cases using session recorders
- Support test case deployment and execution
- Report observations
- Assess success or failure



Complex Systems Generic Architecture and Implementation

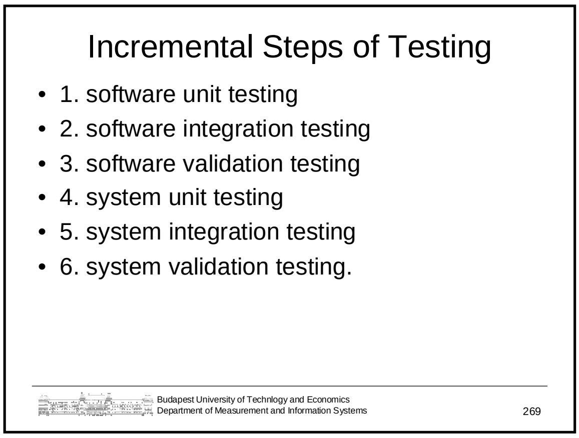
- Two thirds of systems run on a real-time operating system (RTOS)
- Majority of developers of embedded systems use C, C++ (70% will be using C in 2002, 60% C++, 20% Java, 5% Ada [R1])



Granule types

- C function or Ada procedure
- C++ or Java class
- C or Ada (set of) module(s)
- C++ or Java cluster of classes
- an RTOS task
- a node
- the complete system

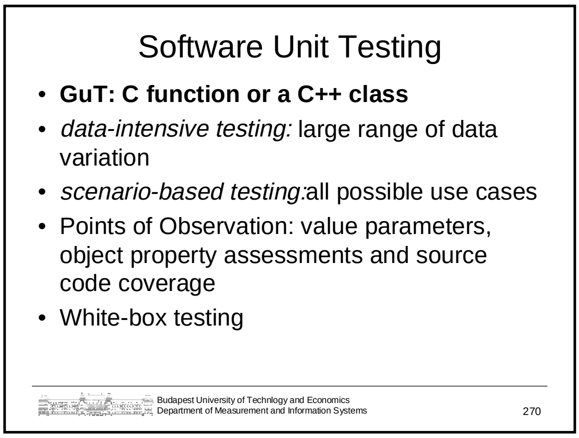
Budapest University of Technlogy and Economics
Department of Measurement and Information Systems
268



Incremental Steps of Testing

- 1. software unit testing
- 2. software integration testing
- 3. software validation testing
- 4. system unit testing
- 5. system integration testing
- 6. system validation testing.

Budapest University of Technlogy and Economics
Department of Measurement and Information Systems
269



Software Unit Testing

- **GuT: C function or a C++ class**
- *data-intensive testing:* large range of data variation
- *scenario-based testing*:all possible use cases
- Points of Observation: value parameters, object property assessments and source code coverage
- White-box testing

Budapest University of Technlogy and Economics
Department of Measurement and Information Systems
270

Software Integration Testing

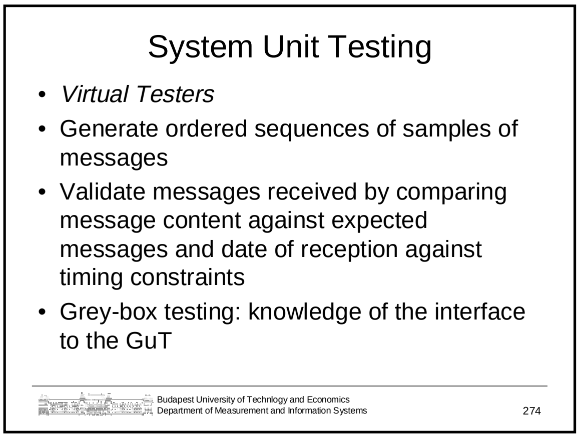
- **GuT: set of functions or a cluster of classes**
- Validation of the interface
- Points of Control: data-intensive main function call or method-invocation sequences
- Points of Observation: interactions
- GuT starts to be meaningful
  - end-to-end test scenario
  - performance tests

Software Validation Testing

- **GuT: all the user code inside a component**

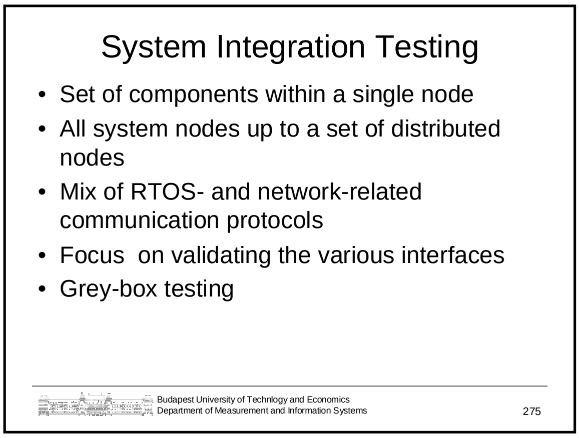
System Unit Testing

- **GuT: full system componen**t
- user code, RTOS- and platform-related pieces:
  - tasking mechanisms, communications, interrupts, etc.
- Point of Control protocol: message sent/received using the RTOS message queues



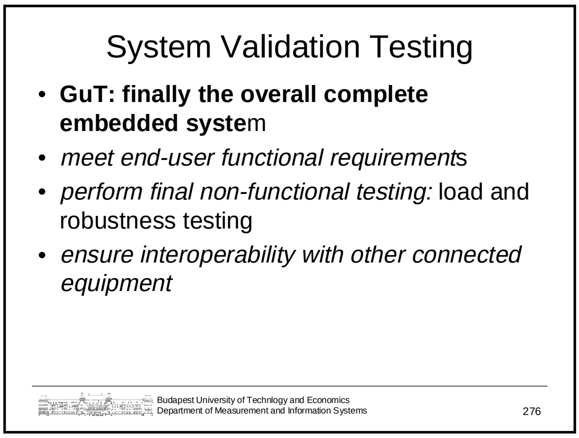
System Unit Testing

- *Virtual Testers*
- Generate ordered sequences of samples of messages
- Validate messages received by comparing message content against expected messages and date of reception against timing constraints
- Grey-box testing: knowledge of the interface to the GuT



System Integration Testing

- Set of components within a single node
- All system nodes up to a set of distributed nodes
- Mix of RTOS- and network-related communication protocols
- Focus on validating the various interfaces
- Grey-box testing



System Validation Testing

- **GuT: finally the overall complete embedded syste**m
- *meet end-user functional requirement*s
- *perform final non-functional testing:* load and robustness testing
- *ensure interoperability with other connected equipment*

Test execution sequence

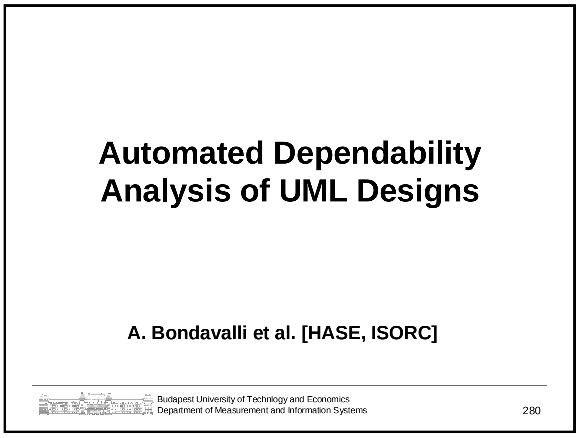
- Whether all these steps applies to the system
- Exhaustive/partial tests
- Test order

Additional Requirements

- Testing technology must add the following capabilities:
  - – Manage multiple types of Points of Control in order to stimulate the GuT
  - – Offer of a wide variety of Points of Observation

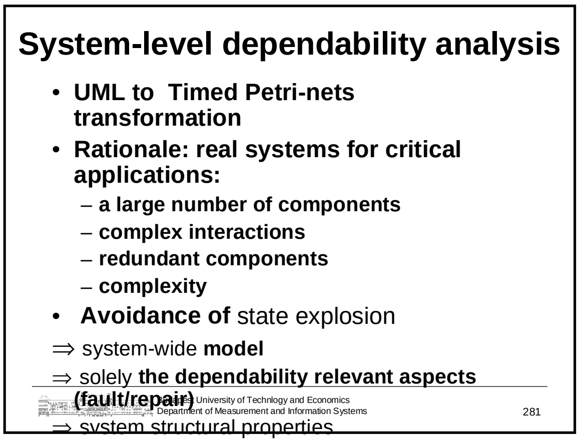
Application Development and Execution Platforms

- The technology used by Rational Test RealTime is embedding the test harness onto the target system
- Compiling test data previously translated into the application programming language (C, C++ or Ada)
- All mandatory DO-178B test requirements up to and including level-A equipment



Automated Dependability Analysis of UML Designs

A. Bondavalli et al. [HASE, ISORC]



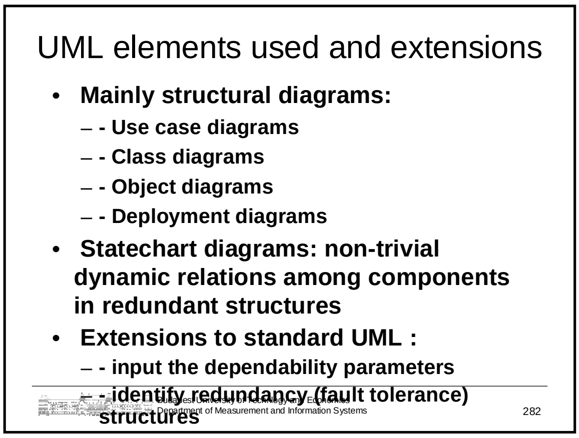
System-level dependability analysis

- **UML to Timed Petri-nets transformation**
- **Rationale: real systems for critical applications:**
  - – **a large number of components**
  - – **complex interactions**
  - – **redundant components**
  - – **complexity**
- **Avoidance of** state explosion

⇒ system-wide **model**

⇒ solely **the dependability relevant aspects (fault/repair)**

⇒ system structural properties



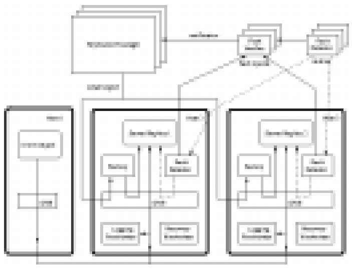
UML elements used and extensions

- **Mainly structural diagrams:**
  - – **- Use case diagrams**
  - – **- Class diagrams**
  - – **- Object diagrams**
  - – **- Deployment diagrams**
- **Statechart diagrams: non-trivial dynamic relations among components in redundant structures**
- **Extensions to standard UML :**
  - – **- input the dependability parameters**
  - – **- identify redundancy (fault tolerance) structures**

Constraints on the UML designer

- **- redundancy:**
  - **class-based approach**
- **- redundant structure: objects from stereotyped classes:**
  - **- redundancy manager,**
  - **- variant,**
  - **- adjudicator, refined by subtypes, e.g. tester, voter, or comparator**
- **- this constraint allows:**
  - **- easily identifying redundancy in the UML design**
  - **automated derivation of relations among**

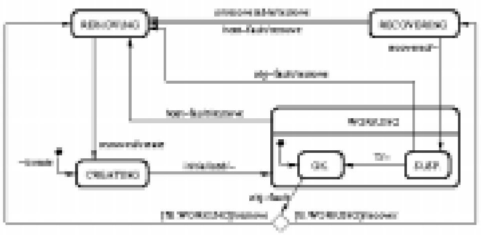
Redundancy Management in Distributed OO Systems



Modelling of Redundancy Structures

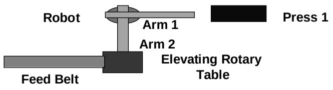
- Dependability model available in the early phases of the system design
- Decision about redundancy scheme (replication, recovery and repair strategies)
- Abstract from details of consistency, checkpoints, recovery
- Dependability model based on the object model
  - (UML: class, object and deployment diagrams)

Statechart of the Redundancy Manager

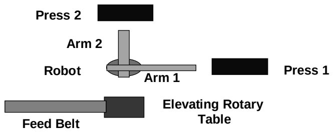


Production Cell example

Basic version of the system

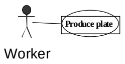


Redundant version with two presses

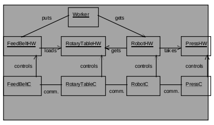


UML Structural views of the Production Cell

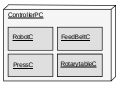
Use case diagram



Object diagram



Deployment diagram



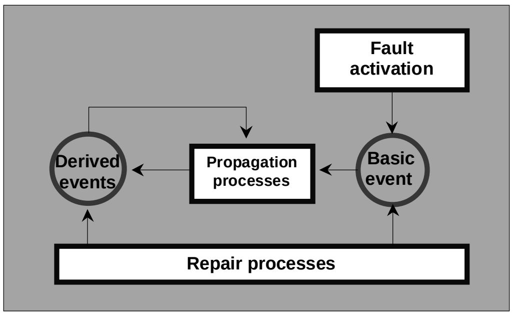
Sketch of the transformation

❶ Extraction of the relevant dependability information

❶UML description⇒ Intermediate model**, fixing:**

- –**-** fault activation processes **resulting in** basic failure events
- –**-** propagation processes**,** **consequences of basic events and** derived failure events
- –**-** repair processes
- –**The Intermediate model is a** hypergraph
  - nodes: **entities from the UML structural diagrams**
  - hyperarcs: **relations between elements**

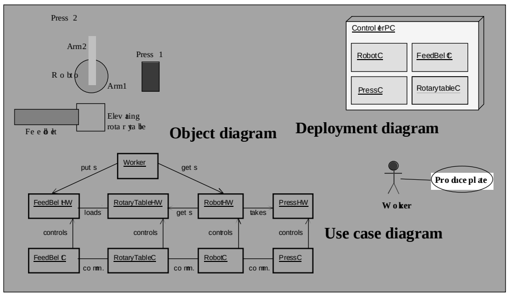
Sketch of the transformation-1



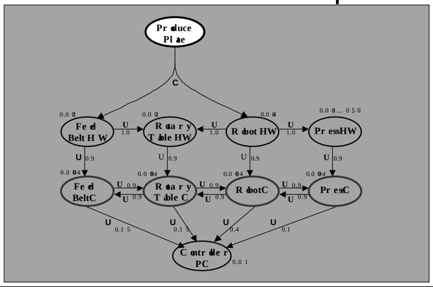
Sketch of the transformation-2

- **❷ IM ⇒ TPN (generality:** postponement **of the selection of the automatic analysis tool)**
- Timed Petri Nets:
  - general **class of Petri Nets, which encompasses GSPN, SAN, SRN.**
  - elements**: places, transitions, I/O arcs,** subnets **(**modular **modeling)**
  - **easy translation into less expressive and powerful PN classes**

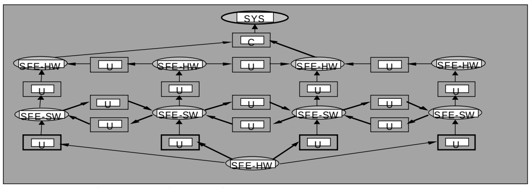
UML diagrams of the Production Cell example



Intermediate model of the Production Cell example



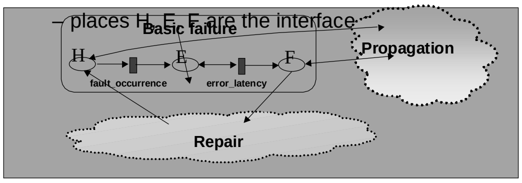
TPN model



- - Ovals: Basic Subnets
- - Rectangles: Propagation Subnets
- - Basic/Propagation subnets refined ⇒ more accurate representation of the crucial parts

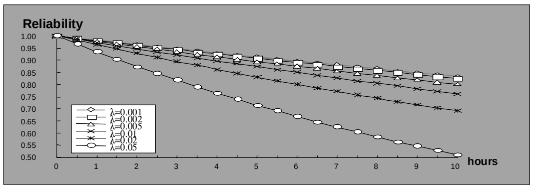
Model refinement

- Subnets of the TPN include well-specified
- interface elements, the sole points at which subnets can be linked to each other
- Basic failure subnet:
  - places H, E, F are the interface



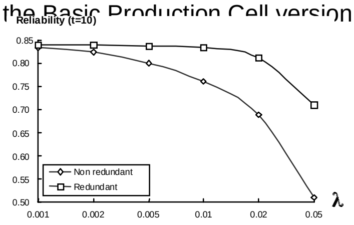
Example of back-annotated results

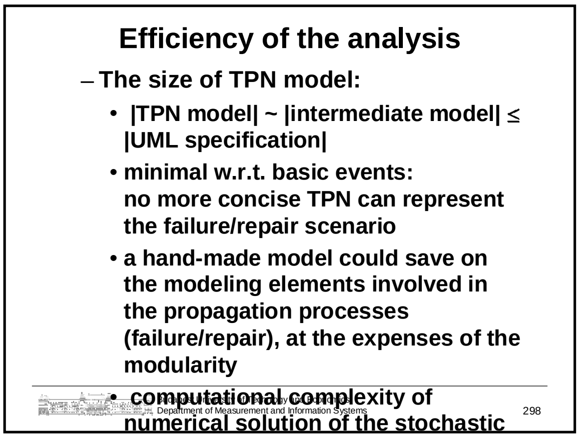
- Reliability of the Basic Production Cell
  - values of the press failure rate l ranging within the interval [0.001,0.05]



Possible design improvement and subsequent re-evaluation

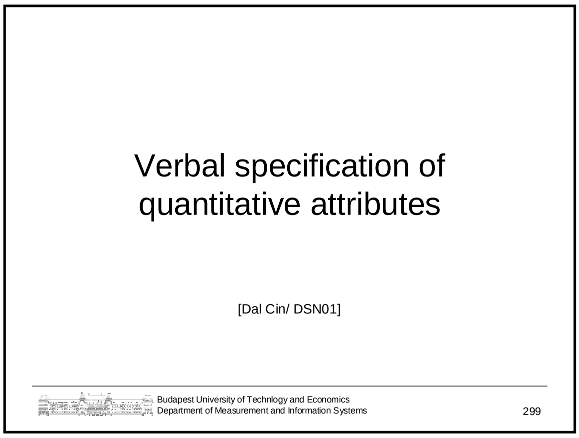
- Two redundant presses: comparison with the Basic Production Cell version





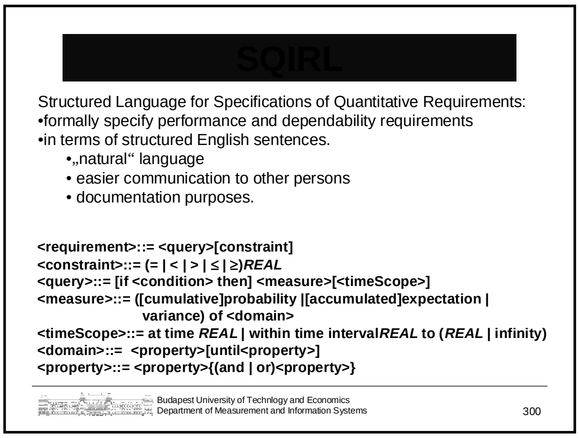
Efficiency of the analysis

- The size of TPN model:
  - |TPN model| ~ |intermediate model| ≤ |UML specification|
  - minimal w.r.t. basic events: no more concise TPN can represent the failure/repair scenario
  - a hand-made model could save on the modeling elements involved in the propagation processes (failure/repair), at the expenses of the modularity
  - computational complexity of numerical solution of the stochastic



Verbal specification of quantitative attributes

[Dal Cin/ DSN01]



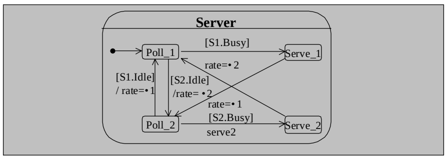
SQIRL

Structured Language for Specifications of Quantitative Requirements:
•formally specify performance and dependability requirements
•in terms of structured English sentences.
- •„natural“ language
- easier communication to other persons
- documentation purposes.

<requirement>::= <query>[constraint]
<constraint>::= (= | < | > | ≤ | ≥)*REAL*
<query>::= [if <condition> then] <measure>[<timeScope>]
<measure>::= ([cumulative]probability |[accumulated]expectation | variance) of <domain>
<timeScope>::= at time *REAL* | within time interval*REAL* to (*REAL* | infinity)
<domain>::= <property>[until<property>]
<property>::= <property>{(and | or)<property>}

Semantics

For a quantitative analysis, the non-terminal symbols **condition** and **property** have to be specified in the specific model context.
•context of UML Statecharts.



probability of in S1.Busy and not in Server.Serve_1 <0.5
probability of the station S1 waiting to be served should be less then 0.5.

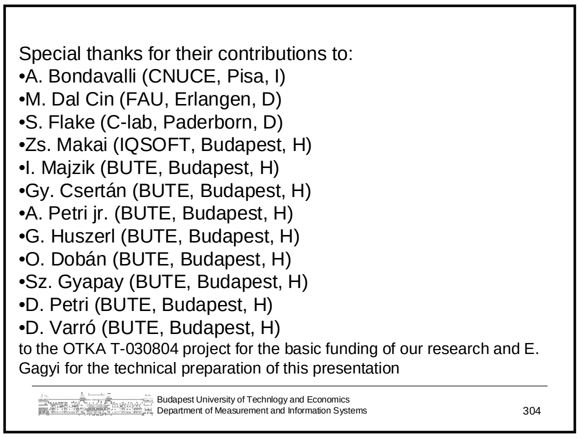
Requirements

The use of SQIRL follows a three-step approach.

- The non-functional requirements: in general terms using SQIRL.
- in the context of the system model used, for example, UML-State chart-models.
- Translation of refined requirements to a notation that can be interpreted by an analysis tool.

Analysis steps

A: Validation of the requirements: SQIRL parser for UML Statechart Models
Syntactical correctness check, validation against the system model
B: Validation of the system model (design): system model and validated requirements as an input for an analysis tool.
Stochastic Reward Nets (SRN) - an extension of Generalized Stochastic Petri-Nets (GSPN) - analysis tool **PANDA**.



Special thanks for their contributions to:

- A. Bondavalli (CNUCE, Pisa, I)
- M. Dal Cin (FAU, Erlangen, D)
- S. Flake (C-lab, Paderborn, D)
- Zs. Makai (IQSOFT, Budapest, H)
- I. Majzik (BUTE, Budapest, H)
- Gy. Csertán (BUTE, Budapest, H)
- A. Petri jr. (BUTE, Budapest, H)
- G. Huszerl (BUTE, Budapest, H)
- O. Dobán (BUTE, Budapest, H)
- Sz. Gyapay (BUTE, Budapest, H)
- D. Petri (BUTE, Budapest, H)
- D. Varró (BUTE, Budapest, H)

to the OTKA T-030804 project for the basic funding of our research and E. Gagyi for the technical preparation of this presentation

Budapest University of Technlogy and Economics
Department of Measurement and Information Systems 304