

Quality & Productivity in Component-based Development

M.R.V. Chaudron

www.win.tue.nl/~mchaudro

Dept. of Mathematics and Computing Science
Eindhoven University of Technology

With contributions from Prof. Ivica Crnkovic, Malardalen, Sweden

Examples of bad interface design?

Agenda

- Quality
 - On testing
 - Predicting Quality Properties of component compositions
- Productivity
 - Reuse
 - Economics of Reuse
- In 2 weeks: demonstration of the assignments

Quality

What is Quality of a system?

- Absence of defects
- Degree to which requirements are met
- Degree to which customer is satisfied

References

- Testing and Quality Assurance for Component-based Software, J. Gao, H.-S. Jacob Tsao, Y. Wu, Artech House, 2003

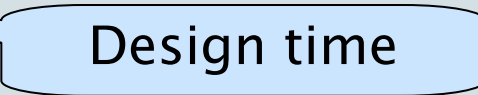

Component Testing

Component Testing

the testing activities that uncover software errors and validate the quality of software components at the unit level

Testing can not establish the absence of errors,
only their presence (E.W. Dijkstra)

Dependability Terminology

- Error: human mistake that results in incorrect software 
- Fault: state/condition of a system that causes a system to fail
- Failure: the occurrence of a fault 

Stages of Software Testing

Software Systems should go through at least the following stages of testing:

- Unit testing:
 - testing of individual components
- Integration testing:
 - subsystems formed by integrating individually tested components are tested
- System testing:
 - The system formed from tested subsystems is tested

Two perspectives on Testing

Vendor

- Does the component deliver what is stated in the specification?
 - Including component model & standards

Buyer/User

- Is this component right for my system?
 - Does the component provide what I need?
 - Are we using/configuring/adapting it correctly?
 - Is it reusable? Interoperable? Customizable?
 - Is it economically feasibly?

Vendor testing

- i. Functional & Extra-functional
performance etc
- ii. Packaging & customization
- iii. Deployment
 - does it install properly?

Buyer testing

‘Reverse’ order

i. Deployment

- does it install properly?

ii. Customization

- can I tailor it to my system?

iii. Functional & Extra-functional

CBD testing problems – Buyer

- Lack of access to source code & design documents makes testing more difficult because:
 - More difficult to understand what happens/can happen
 - What is its state-space?
 - Access to source or documentation is desirable for selecting test-scenarios
 - Consider a set implemented as an array or as a list
 - Test adequacy
 - how much testing is enough? No coverage known!
- Lack of knowledge of component creator

CBD testing problems – Buyer

- Input domain may be too large to cover a significant portion
- If a component provides functionality you are not planning to use, it is probably wise to also test that.
- There may be more assumptions on the working of other components than becomes clear from the specification & interface. Where to look for these?

Testability

determined by (o.a.) the following factors

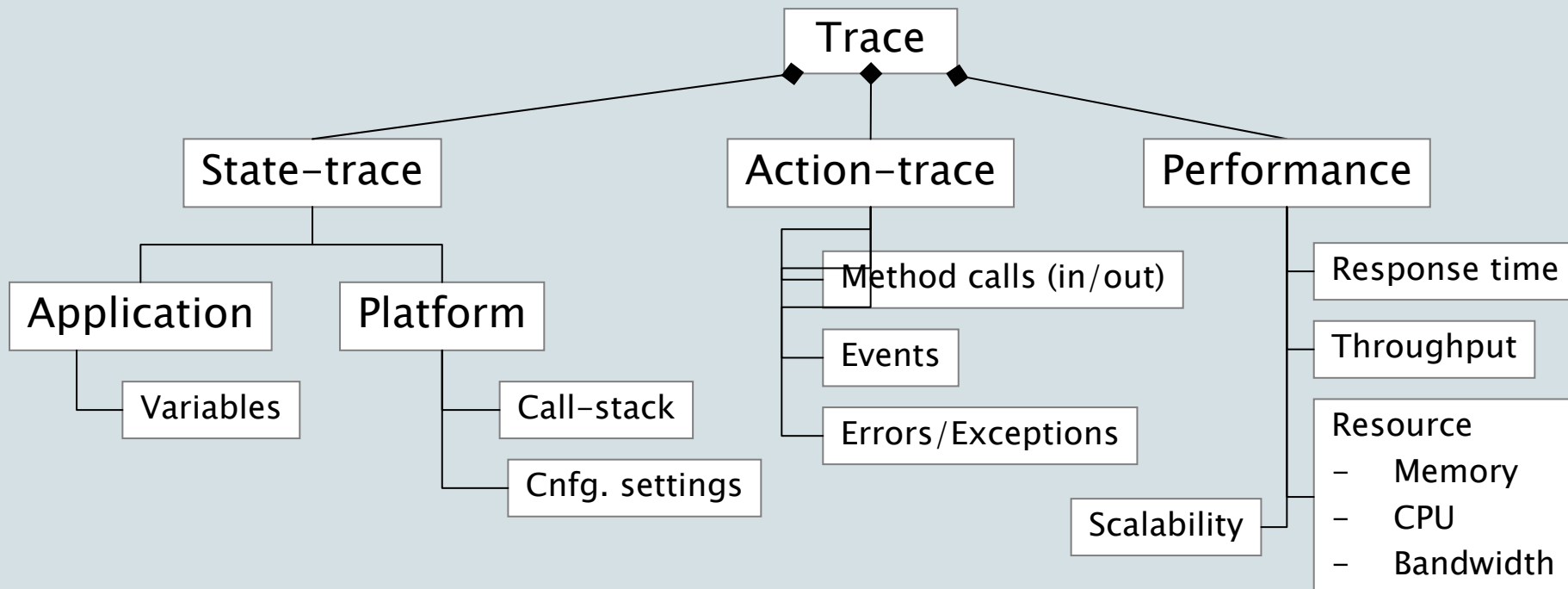
- Understandability
- Observability
- Traceability
- Controllability
- Test support capability

Testability – Understandability

- Availability of component information
 - User documentation
 - Development documentation
 - Testing documentation
 - Test plan
 - Test criteria
 - Test scripts/drivers/stubs
 - Problem reports (?)
 - Coverage data
- Understandability of available information

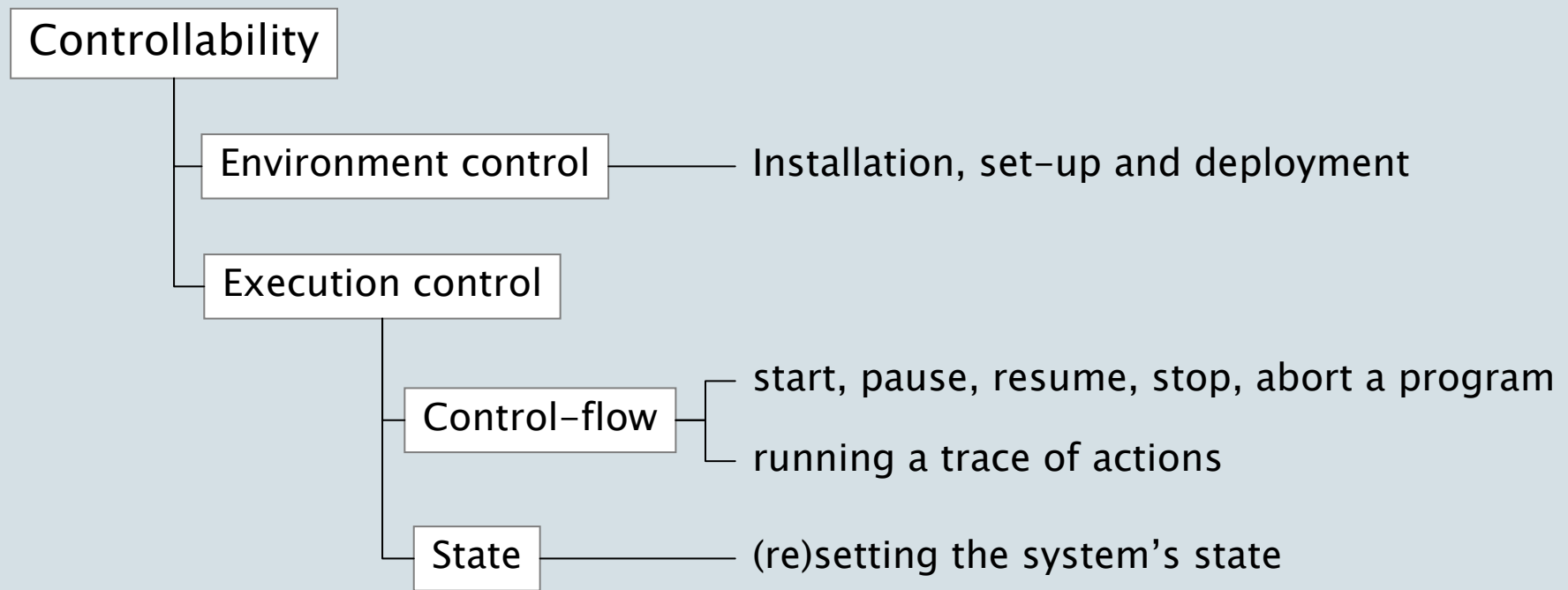
Testability – Observability/Traceability

- How easy it is to observe/monitor the response of a component on stimuli
 - Different inputs lead to different outputs
- Traceability: capabilities that enhance observability



Testability – Controllability

- How easy is it to put a component into a specific state and make it perform a certain action



Predictable Assembly

What are the extra-functional properties of a composition of components?

When Predictable Assembly?

- Design Time
- Run-time change in components

Predictable Assembly & Specification Guidelines

- How can properties of a composition be derived from the properties of its parts?
 - Functionality
 - Extra-Functional
- Which information needs to be specified per component in order to construct a model of the composition?

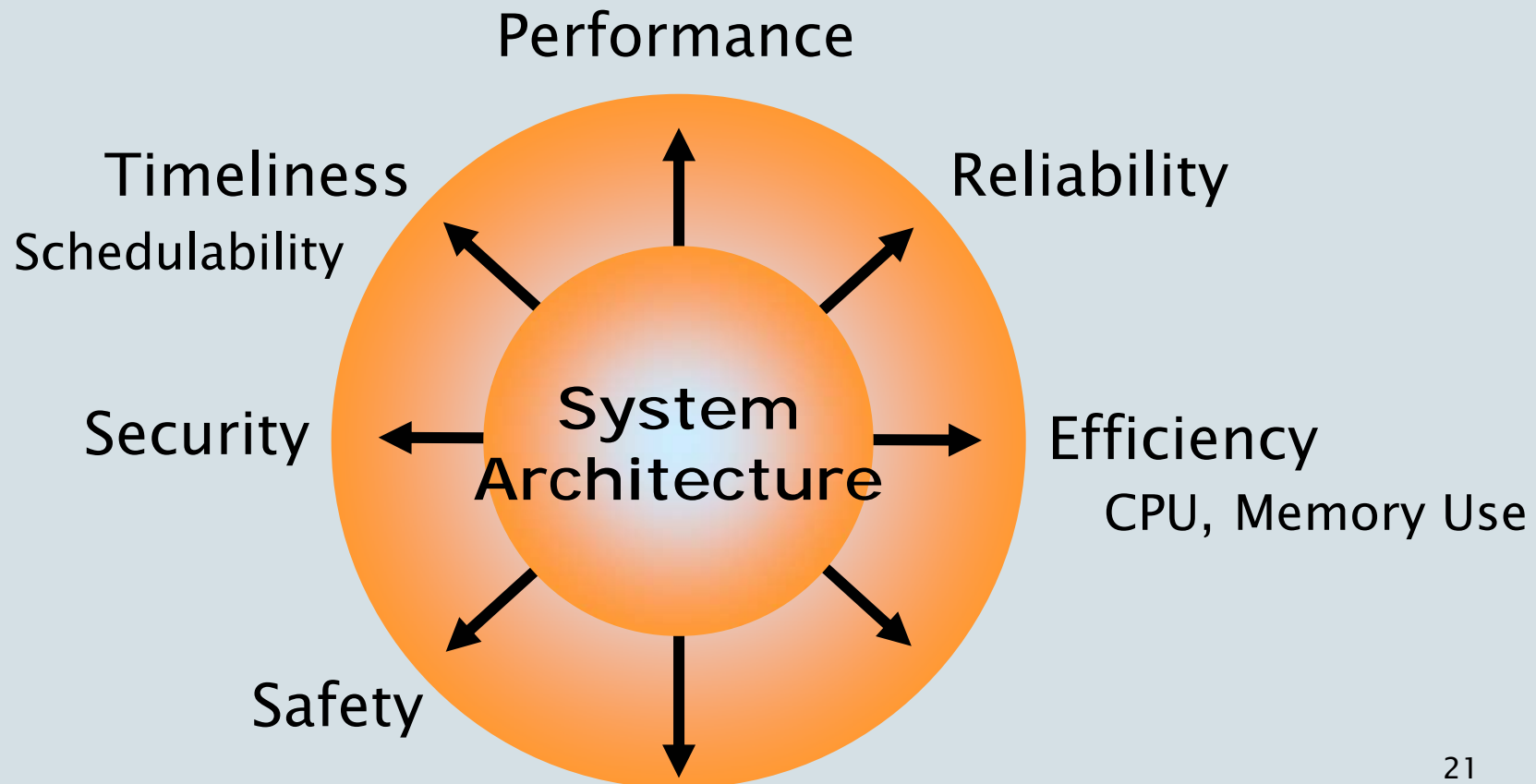
Composition of properties

Hypothesis: properties of the system are a function of properties of the components



Extra-Functional Properties

An architecture needs to balance the extra-functional system properties



What are properties?

Examples:

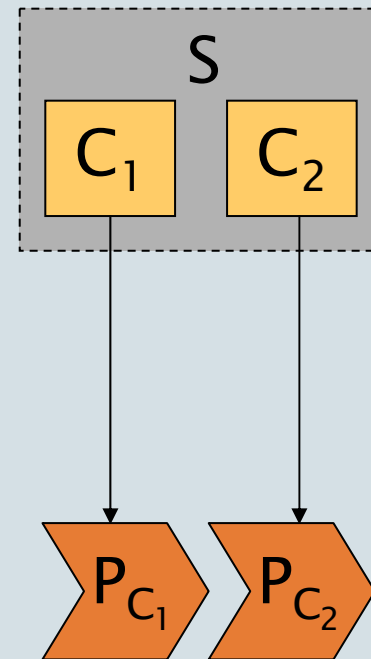
- Timeliness, Responsiveness, Schedulability, CPU utilization, Memory utilization, Latency, Throughput, Concurrency, Footprint
- Availability, Reliability, Safety, Security,
- Cost
- ...

Problems

- There are different system/components properties
 - Different abilities to (formally) specify them
 - Different abilities to measure them
 - Different complexity (relations to other attributes...)
 - Different ways (and efforts) to achieve them
- Different properties are of interest in different domains

Compositional Reasoning (Semantics)

- Calculating properties of a system by combining properties of its constituents.
- If $S = C_1 \oplus C_2$
- Then $P(S) = P(C_1) \otimes P(C_2)$
- ‘Traditionally’ $P(C_i)$ denotes the functional meaning of C_i



Predictable Assembly: Functionality

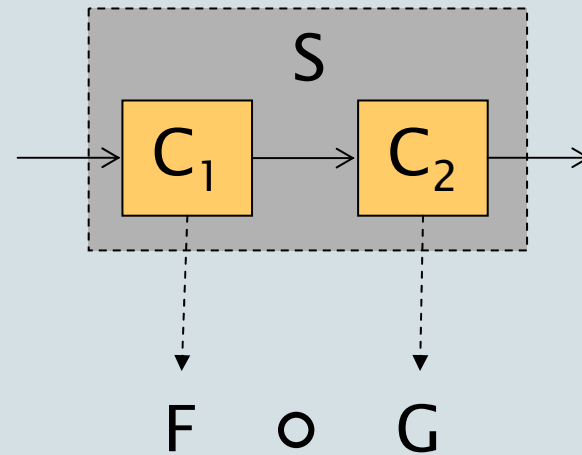
- Can be fairly adequately specified using
 - predicate logic (pre- and post-conditions)
 - using functions (lambda expressions)
- However, the **behaviour** of components is typically not sufficiently well specified;
typically because this is not part of the program text.
- Extra-functional properties?

Is a component autonomous or reactive?

- If autonomous, then when does it take action?
- What happens if two reactive components are composed?

Compositional Reasoning: Functionality

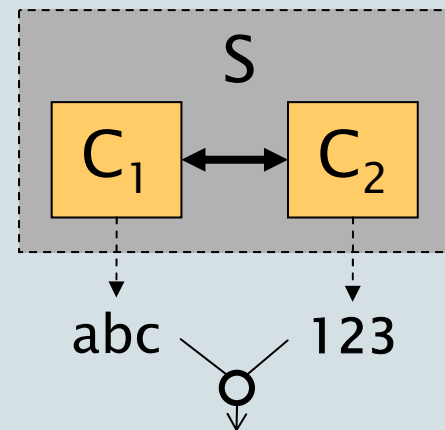
- Meaning $M(C)$ of program C can be a function from inputs to outputs; e.g. $M(C) = \lambda x.x^2$
- Then composition is nicely modelled by function composition



Compositional Reasoning: Behaviour

e.g. Traces

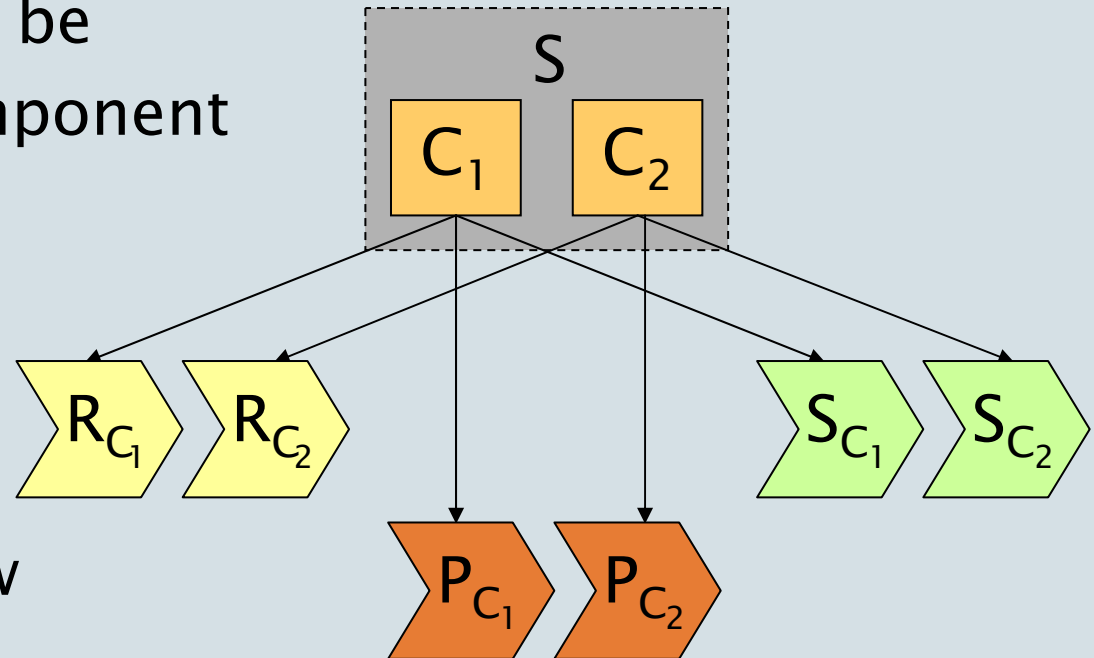
- Meaning $P(C)$ of program C can be the (set of) traces of actions that C can perform
- then sequential composition can be modelled by concatenation
- parallel composition can be modelled by interleaving
- complications arise if components synchronize



abc123, ab1c23, a1bc23, ..., 123abc₂₇

Compositional Reasoning

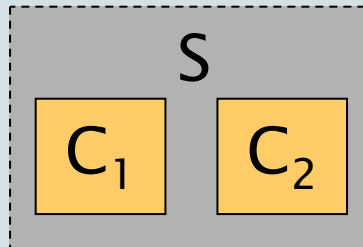
Different models may be associated with a component



some examples follow

Predictable Assembly

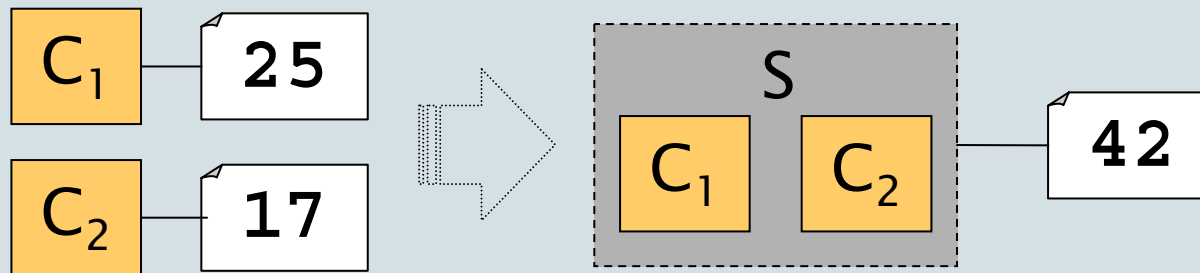
- Same question, now for extra-functional properties.
- Static properties vs. Dynamic properties
- Let's consider dynamic memory-use
- Given the dynamic memory-use of C_1 and C_2 .
- Now what is the dynamic memory use of S ?



?

Problem Instance: Cost

Derive cost of a system from cost of its parts.



Other example: static memory use

Let's move onto real-time / timeliness properties

Example

- “Physical characteristics”
 - Static memory

$$M(C) = \sum_{i=1}^n M(c_i)$$

M = memory size

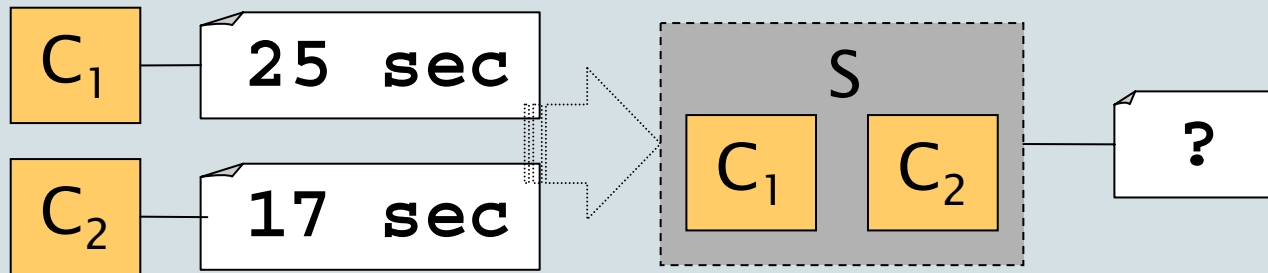
C = composition

c_i = components

$M(c_i)$ is known for all c_i

Problem Instance: Timeliness

Derive timing of a system from timing of its parts.



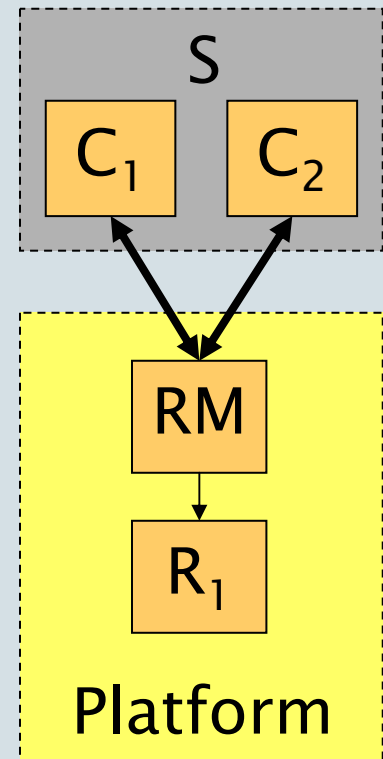
There is interaction via shared resources.

Discussion

- Take 4 minutes to think about a solution
- What are the complications

Complicating Factors in Compositional Reasoning about Extra-Functional Properties

- The property is not determined by the application software only
- But also by the platform
 - platform may be OS + run-time environment
 - in particular the resource management
 - Scheduling
 - Memory management
- The information supplied by C_1 and C_2 is not sufficient to reason about the composition of their extra-functional properties.



A Scenario-based Approach to Predictable Assembly

Based on research with

Johan Muskens & Egor Bondarev

as part of the ITEA projects Robocop and Space4U

Robocop Component

Robocop Component

Resource Model

Simulation Model

Documentation

Functionality Model

Source Code

...

Executable Component

A Robocop Component is a set of Models

- Model can be machine and/or human readable / executable
- Typically: one of those is executable on a target system
 - Called the (executable) component

Component behaviour model

```

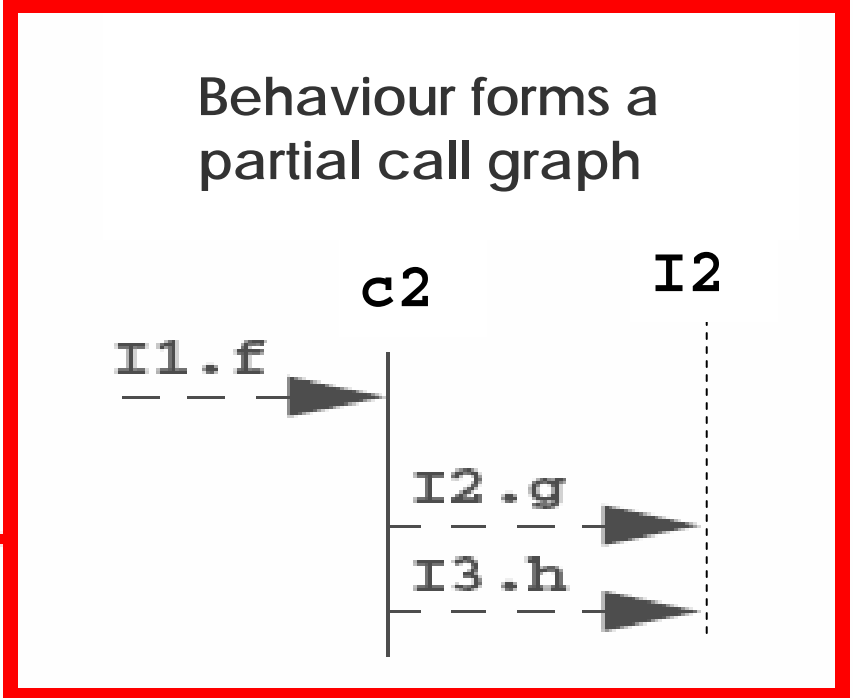
component c2
  requires I2
  requires I3
  provides I1{
    operation f
      uses I2.g
      uses I3.h
      behaviour
        operation f calls:
          I2.g*
          I3.h
  }
  }
  
```

Component

Behaviour

Variables

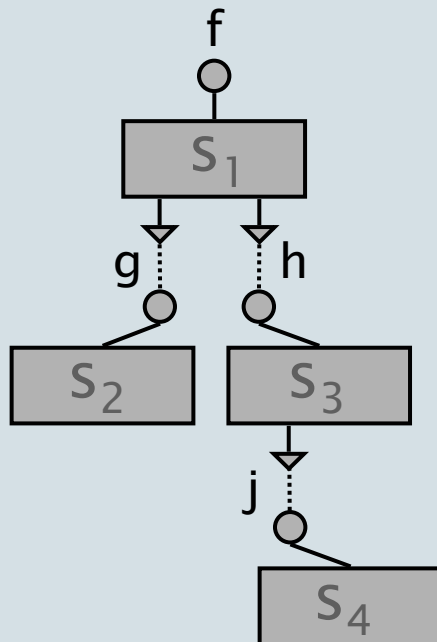
Resource claims / releases are modeled explicitly



Operation f resource use:
 claim 100
 release 100

Model Assembly: Phase 1: Structure

- At bind-time, the decision is made which services are composed to provide the implementation.
- Press button 'f'



- f is provided by s_1
- s_1 needs g and h
 - g is provided by s_2
 - s_2 is done
 - h is provided by s_3
 - s_3 needs j
 - j is provided by s_4
 - s_4 is done

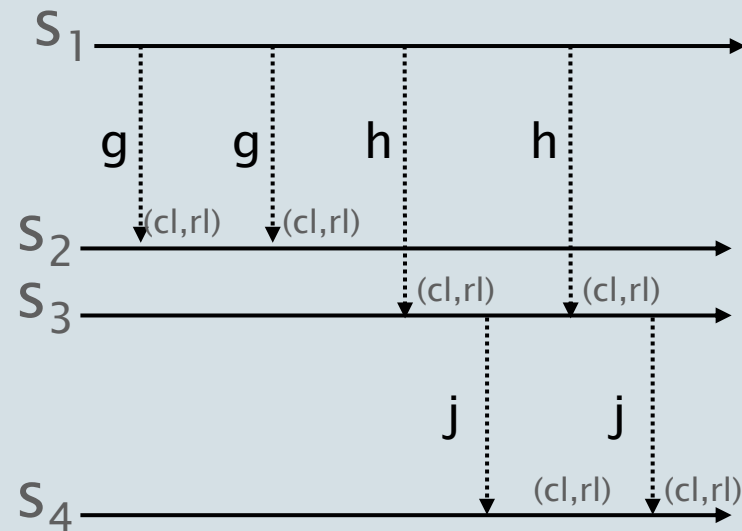
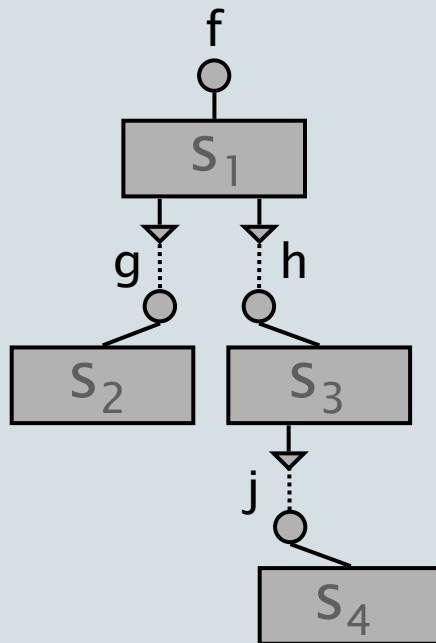
Model Assembly: Phase 2: Logical Behaviour

Combine the behaviour models of the operations used

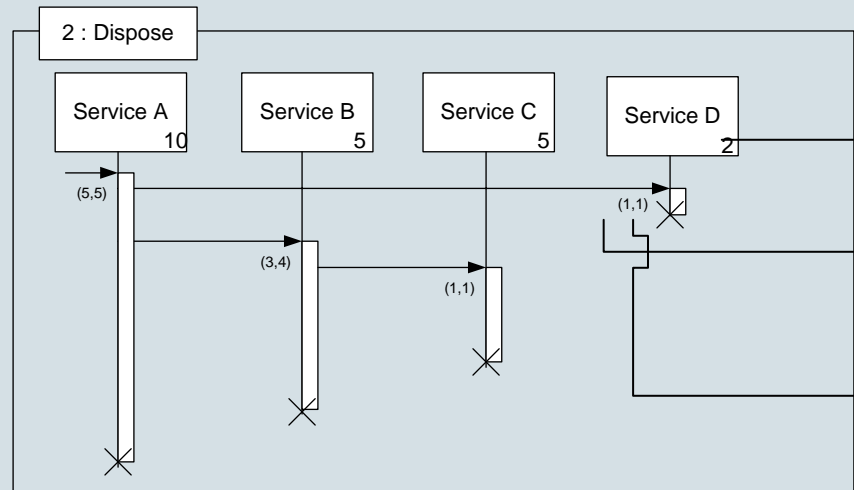
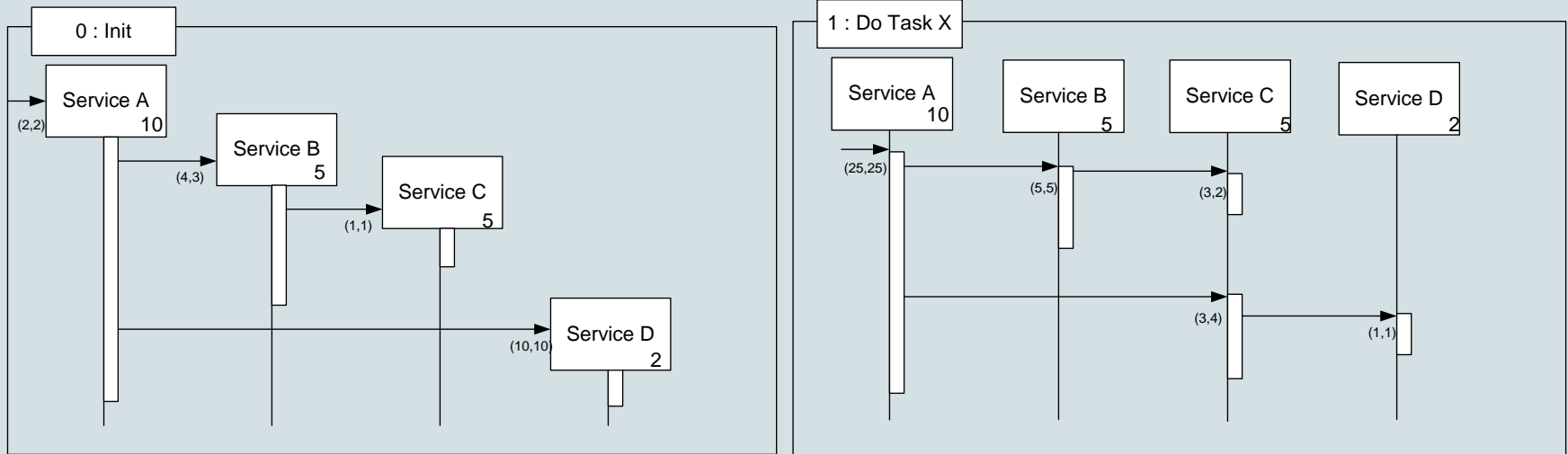
Press button 'f'

operation f calls:

S2.g; S2.g; S3.h; S3.h



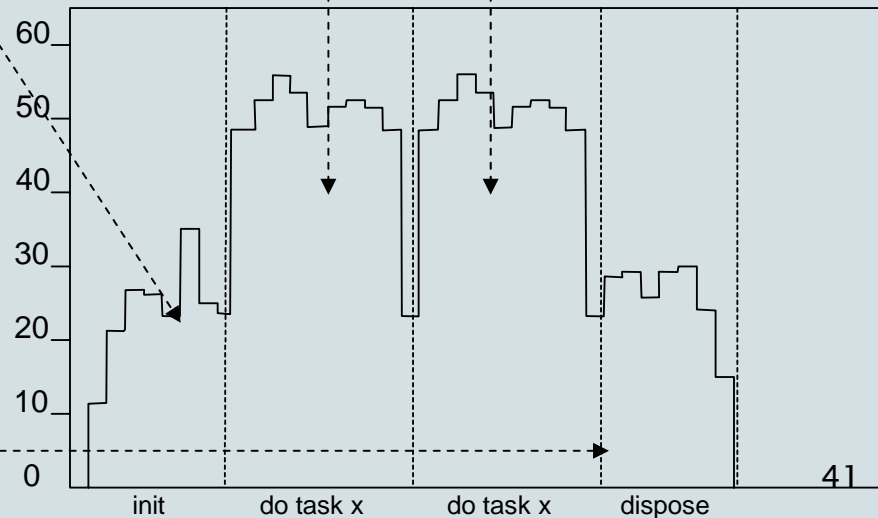
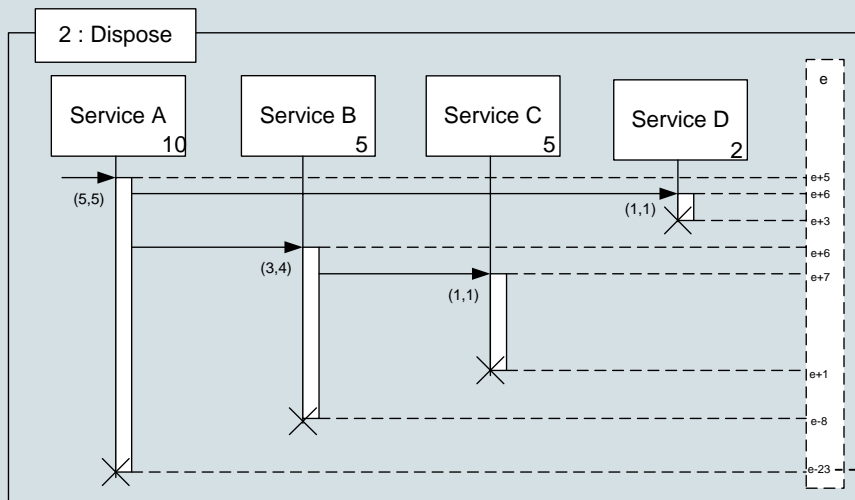
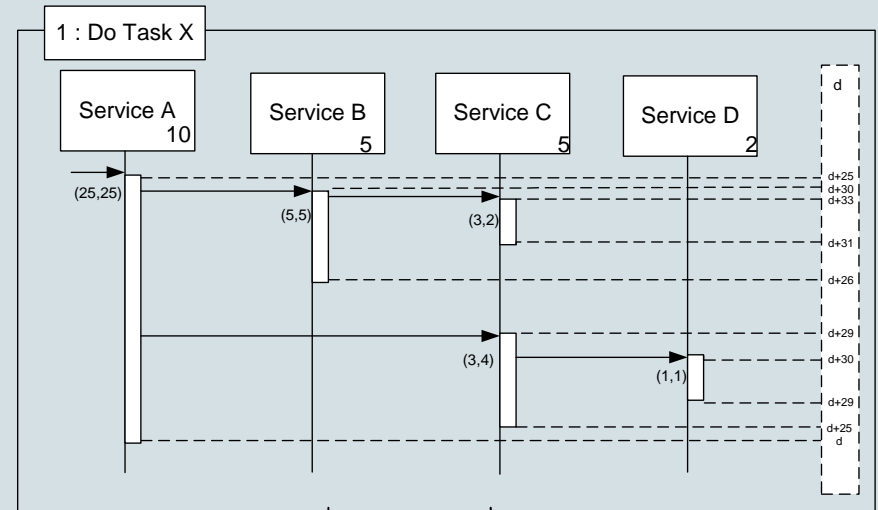
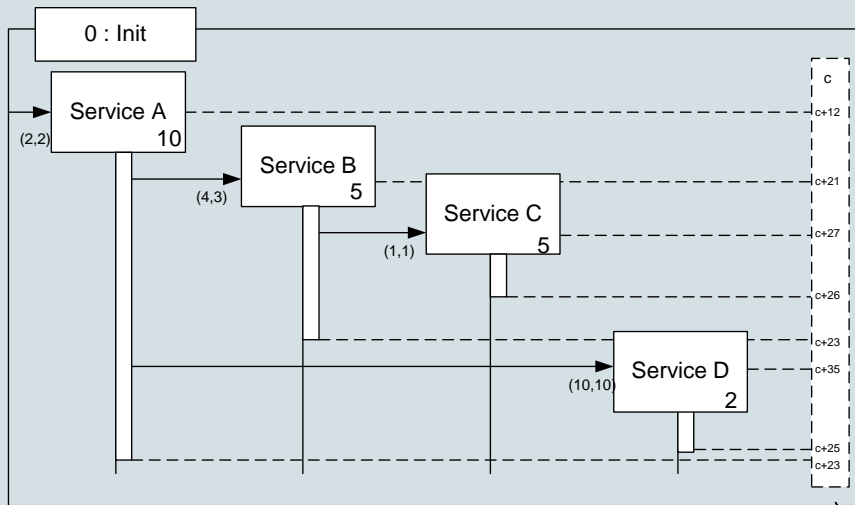
Annotate Scenario using Resource Use



Annotations:

- Nr of Resources needed for creation of instance
- Nr of Resources that are claimed before / during operation execution
- Nr of Resources that are released after / during operation execution

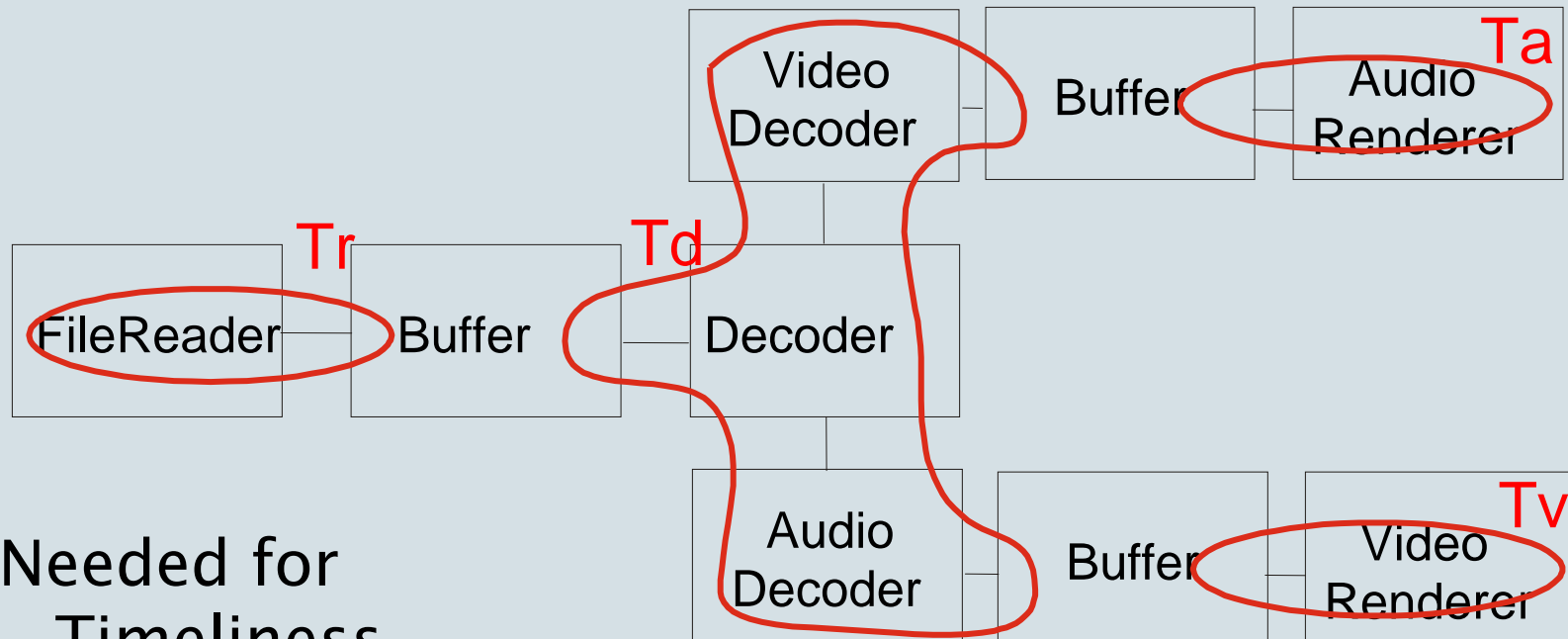
Concatenate Scenarios



What if multiple scenario's are executing concurrently?

Dealing with Concurrent Tasks

- Task based accumulation of resource consumption
- Accumulation of per task resource consumption based on scheduling policy

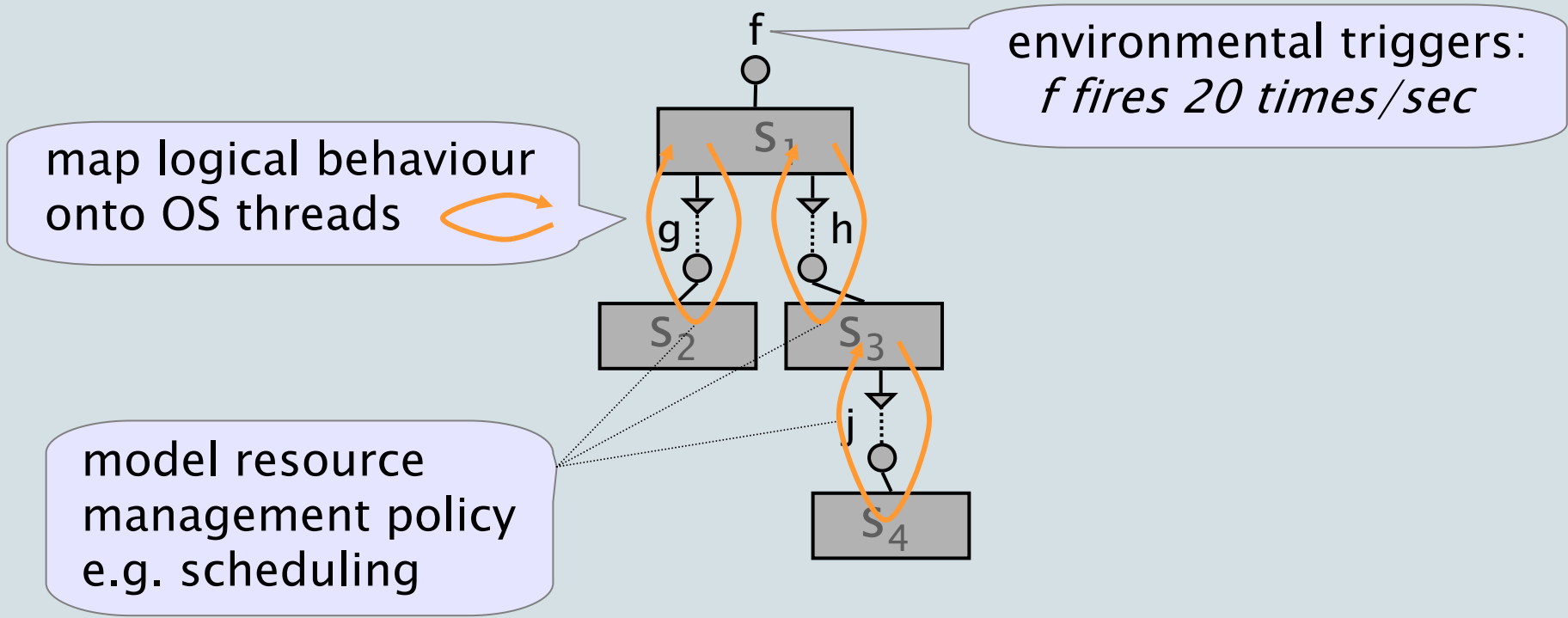


Needed for

- Timeliness
- CPU utilization

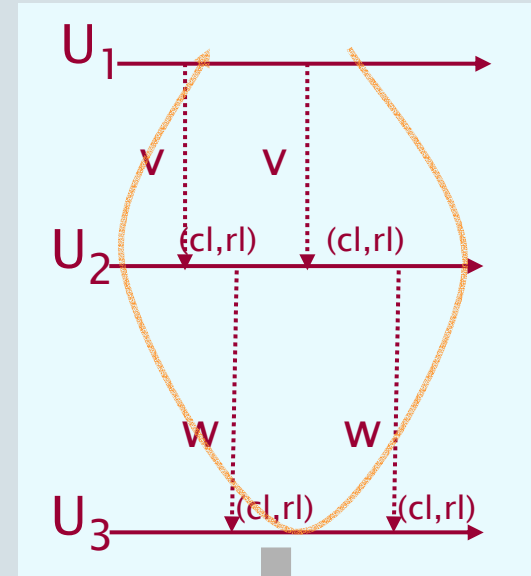
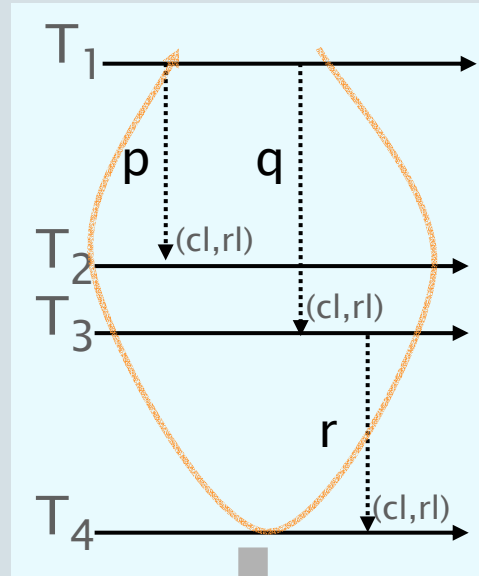
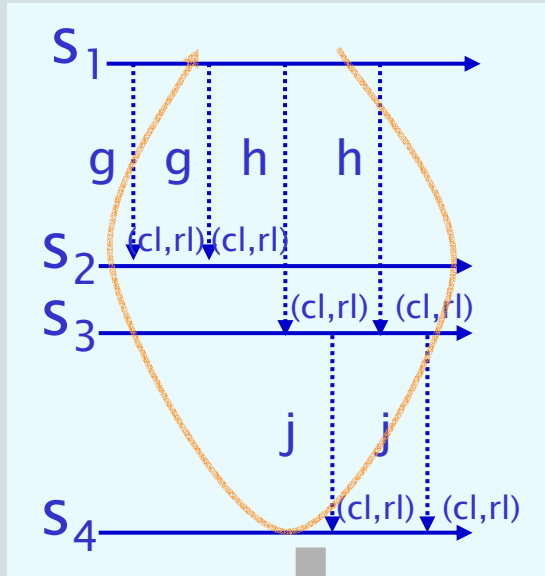
Composition of Execution Behaviour

- Map 'logical' behaviour onto concurrent tasks
- model triggers from the environment (statistically)



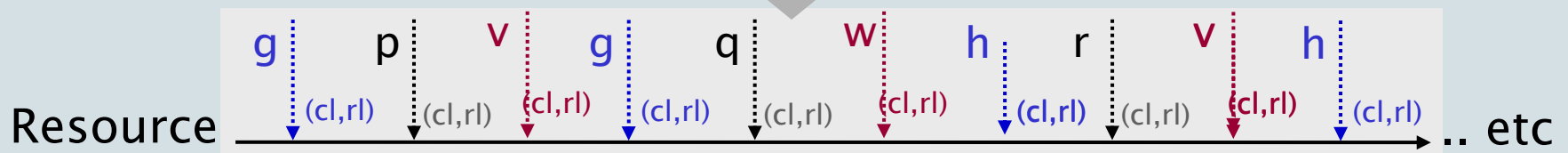
- additional complexity: synchronization between tasks 44

Phase 3: Execution Behaviour



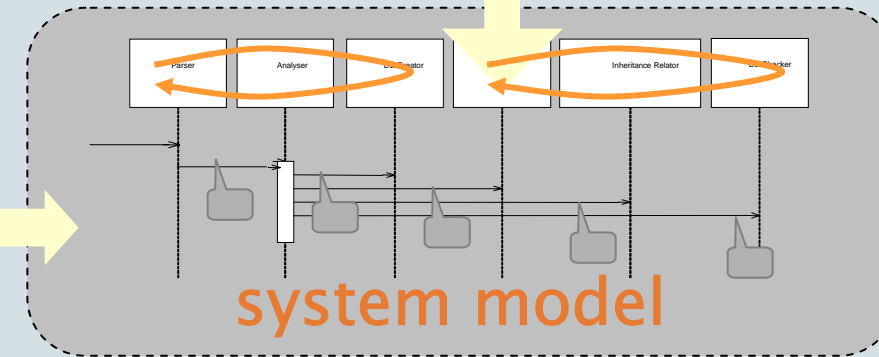
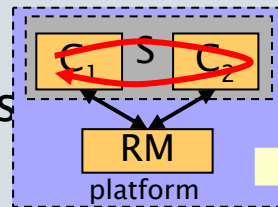
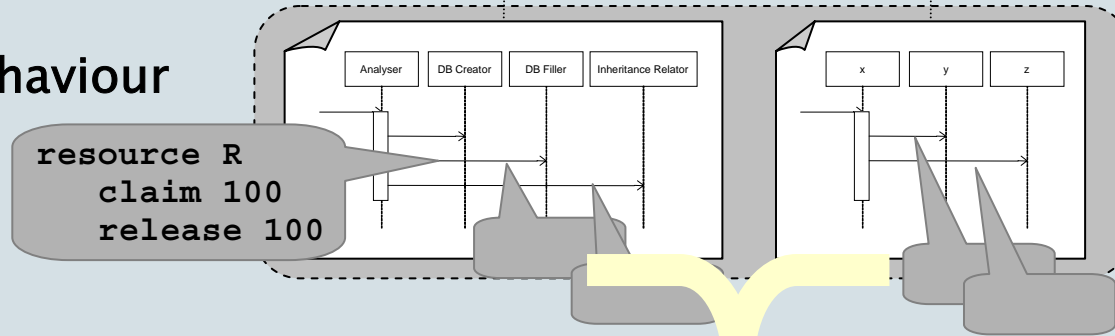
Resource Management Policy
e.g. scheduler

e.g. Round Robin,
EDF, ...

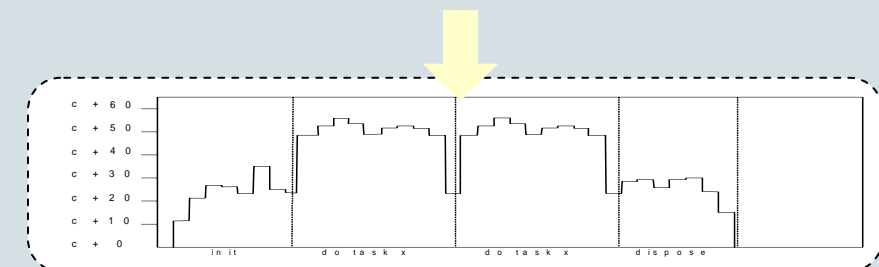


Recipy for Predictable Assembly

1. Composition of System Structure
2. Composition of Logical Behaviour of whole system
3. Composition of Execution Behaviour
 - threads
 - resource mng policies



4. Analyse



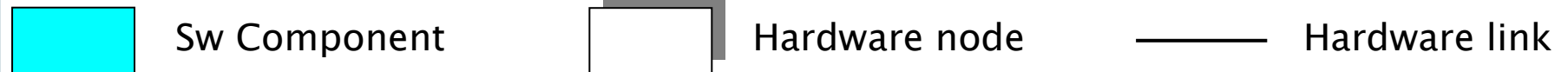
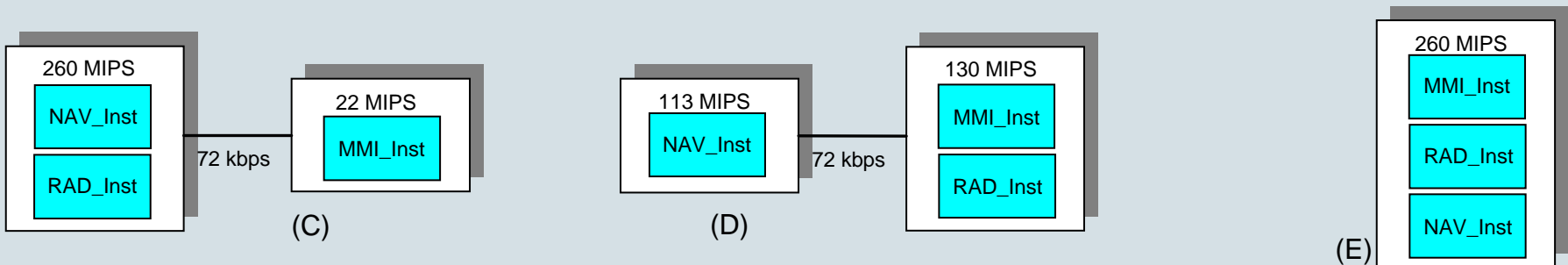
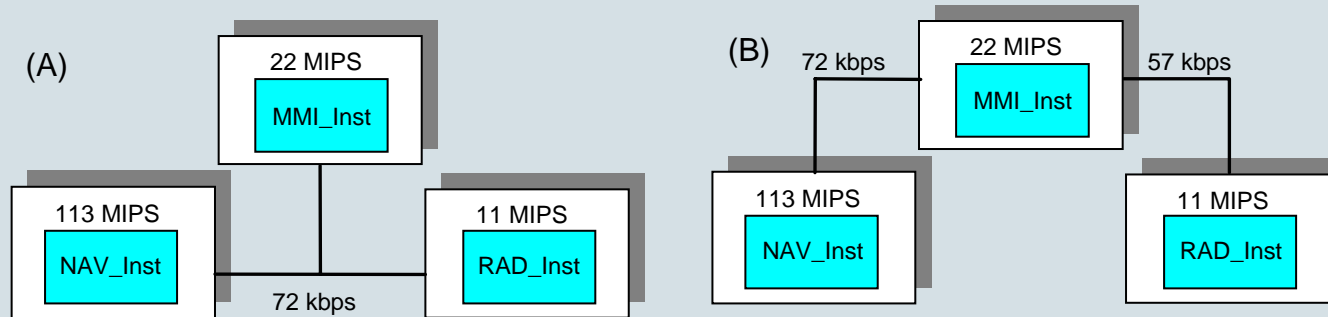
Tooling

The screenshot displays the Eclipse Platform interface for editing a scenario. The main window is titled "Resource - CarNav_Arch5_Scen1.scenario - Eclipse Platform". The interface is divided into several sections:

- Navigator:** Shows a project structure with folders like "ScenarioDemo" and "UseCase", and files like ".project", ".classpath", and ".uml".
- Toolbar:** Contains various icons for editing and simulation, such as "Select", "Marquee", "Edge", and "TriggerLink".
- Main Editing Area:** Displays a diagram titled "Edit the whole scenario". The diagram shows three blue rectangular components connected by lines. The connections are labeled "IPParameters" and "IRDSDecoder".
- Properties View:** Located at the bottom, it shows the properties of the selected scenario. The "Name" property is set to "CarNav_Arch5_Scen1".

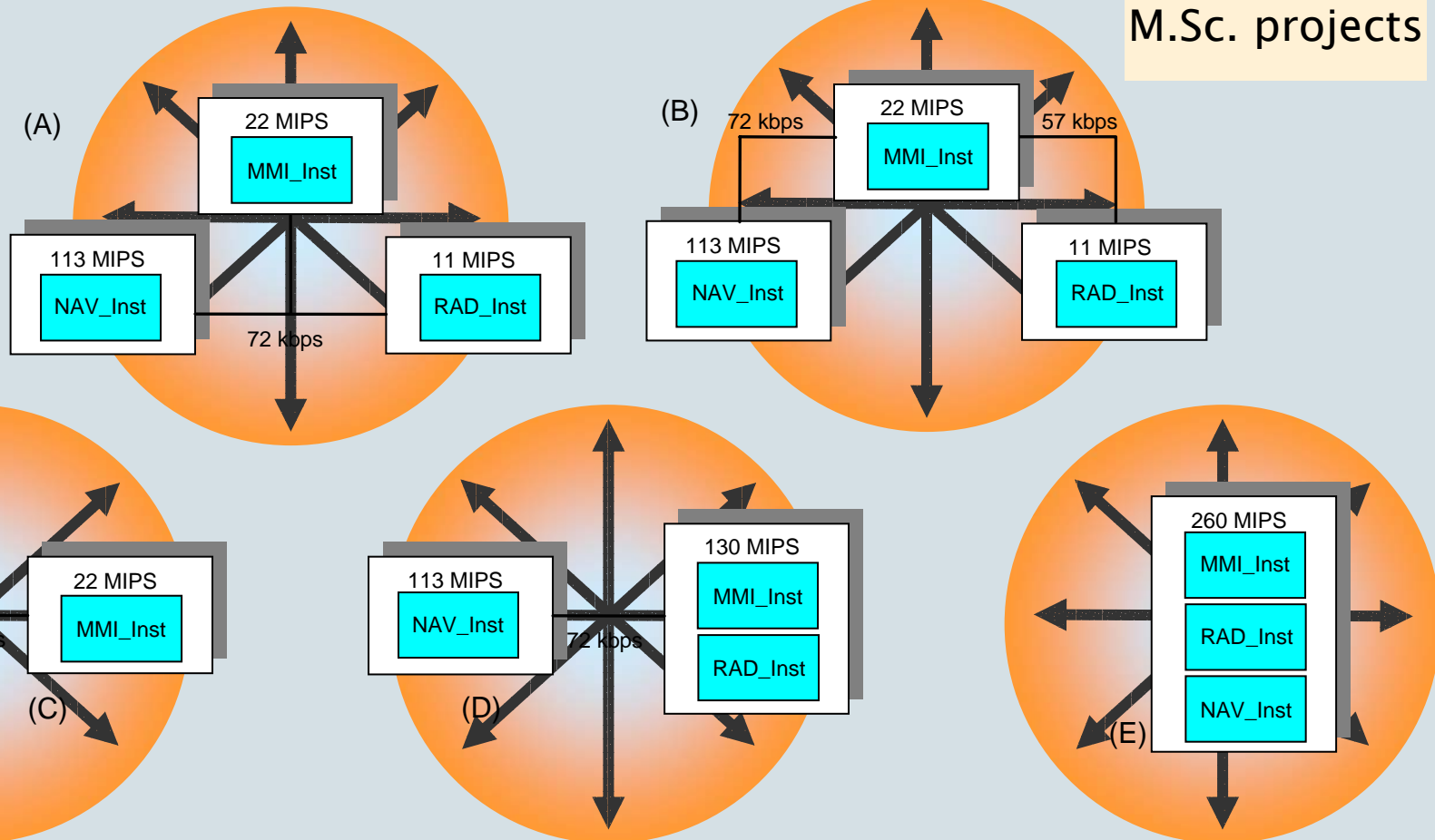
Property	Value
id	w11012943433160
Name	CarNav_Arch5_Scen1

Evaluating Architectural Alternatives



Evaluating Architectural Alternatives

M.Sc. projects



Sw Component

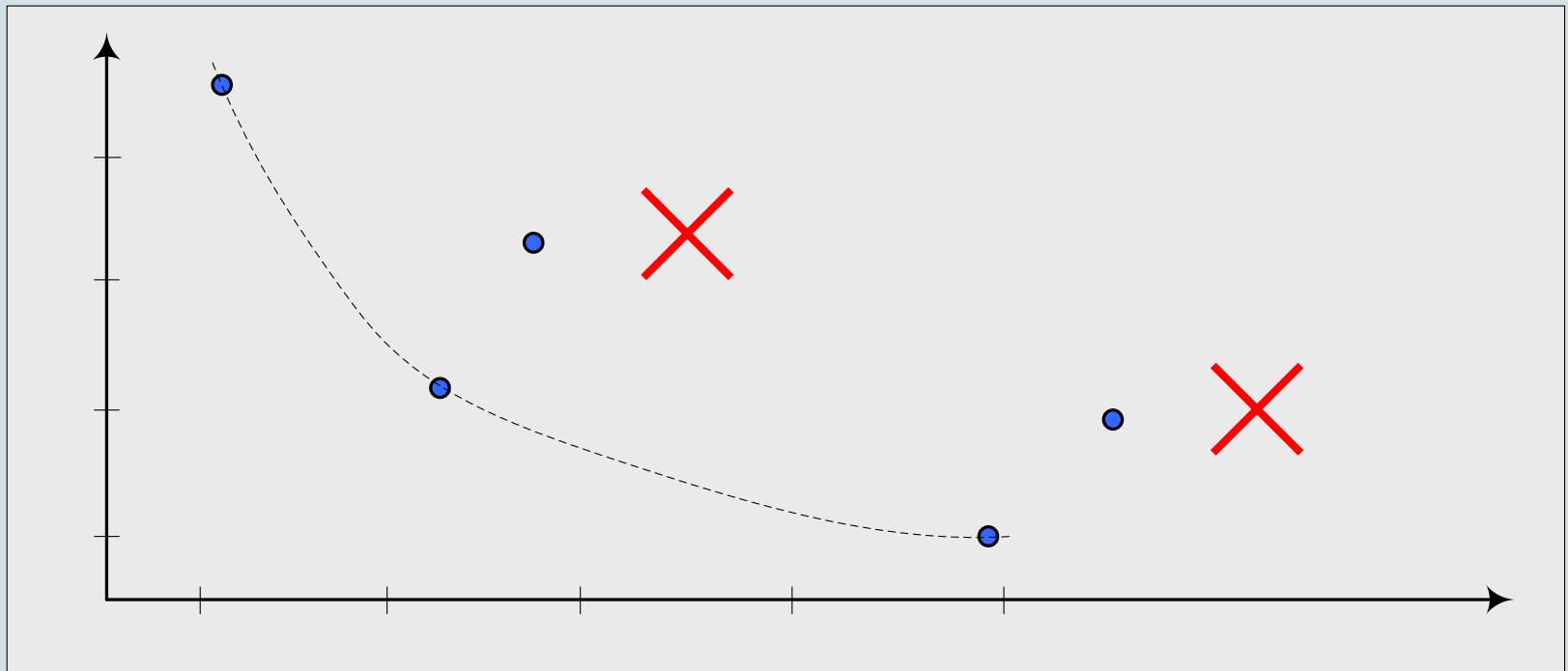


Hardware node



Hardware link

Pareto-analysis of Alternatives



Evaluation of the prediction method

- Compositional (supports third-party binding)
- Can be applied to different types of resources
- Support different types of analyses (Worst, Best, Avg, ...)
- Can be used throughout development cycle (design / implementation / deployment).
- Scenario's
 - Do not give 100% guarantees as usual in formal methods,
 - Focus on what is important
 - Allow incremental accuracy & trade-off with effort invested
 - Can be used in 'lightweight' manner

Composability Property Classification

1. *Directly composable properties.* A property of an assembly which is a function of, and only of the same property of the components involved. E.g. cost.
2. *Architecture-related properties.* A property of an assembly which is a function of the same property of the components and of the software architecture.
3. *Derived (emerging) properties.* A property of an assembly which depends on several different properties of the components.
4. *Usage-dependent properties.* A property of an assembly which is determined by its usage profile. E.g. CPU load.
5. *System context properties.* A property which is determined by other properties and by the state of the system environment.
E.g. safety.

Concluding Remarks

- Predictable Assembly
 - requires
 - the composition of independently developed models
 - the explicit modeling of the composition logic and the composition framework.
 - enables
 - design exploration

Software Reuse

References

- Reuse-Based Software Engineering: Techniques, Organization and Controls, H. Mili, A. Mili, S. Yacoub, E. Addy, Wiley & Sons, 2002
- Measuring Software Reuse: Principles, Practices and Economic Models, J.S. Poulin, Addison-Wesley, 1997

also see :

http://home.stny.rr.com/jeffreypoulin/html/reucalc_basic.html



Reuse

Reuse is like a savings account. Before you collect any interest, you have to make a deposit, and the more you put in, the greater the dividend.

attributed to Ted Biggerstaff, 1983 ITT Reuse workshop

What is software reuse?

Software reuse is the process whereby an organisation defines a set of systematic operating procedures to specify, produce, classify, retrieve, and adapt software artifacts for the purpose of using them in its development activities.

Mili et.al. 2002

Why Reuse?

Reuse is not a goal in itself (neither are CBSE, SPI, ...)

Reuse is driven by business goals:

- Increased productivity
 - Reduced development effort
 - Reduced development time
 - Reduced development cost
- Increased quality
 - better interoperability
- Easier maintenance

What to reuse? Asset

- Requirements
 - Typically are the result of a labour intensive process involving analysts and domain experts. They codify important domain knowledge.
- Architectures
 - Description of the principal solutions for achieving a set of functional and non-functional requirements. They capture important design decisions and their rationale.

What to reuse?

- Design
 - Detailed specifications for subsystems; e.g. design-patterns
- Code
 - Machine processable function. Embodies knowledge of encoding a solution in implementation language.
Also has non-functional properties.
 - Many possible forms:
 - Executable, source, macro's, template's, libraries, ...
- Data
 - Standards, formats, graphics, ...

What to reuse?

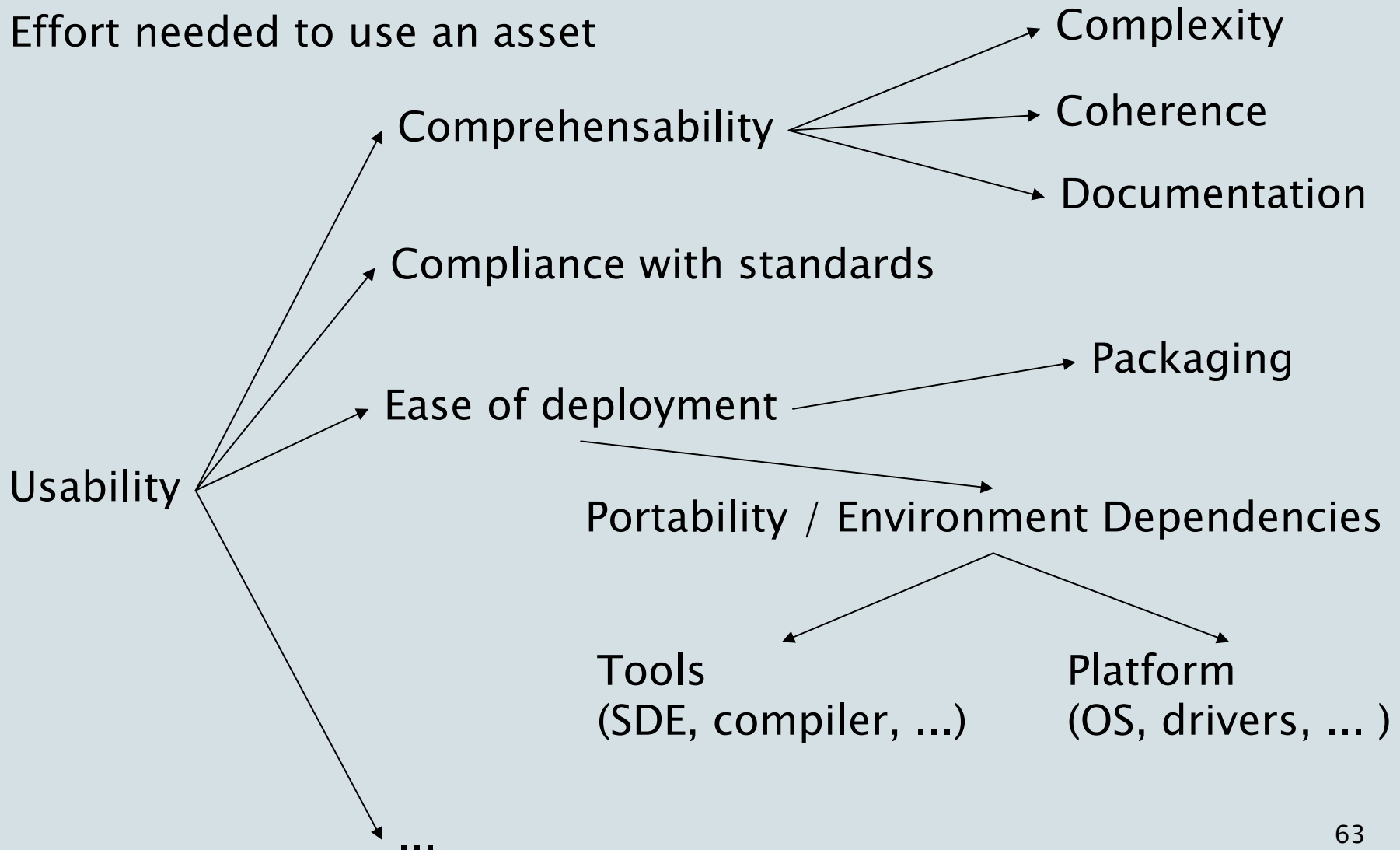
- Test data
 - Reuse test scenario's e.g. after a modification/extension.
Test scenario's capture knowledge about typical faults.
- Documentation
 - Explanation that accompanies some development document (req/arch/design/implement)

Not Considered Reuse

- Use of OS, Tools (e.g. Compilers)
- New versions of a system
- COTS
 - spreadsheet, database, ...
 - (domain independent) libraries
- Generated code
- Modified assets (!)
- Copy & Paste (code scavenging)

Usability

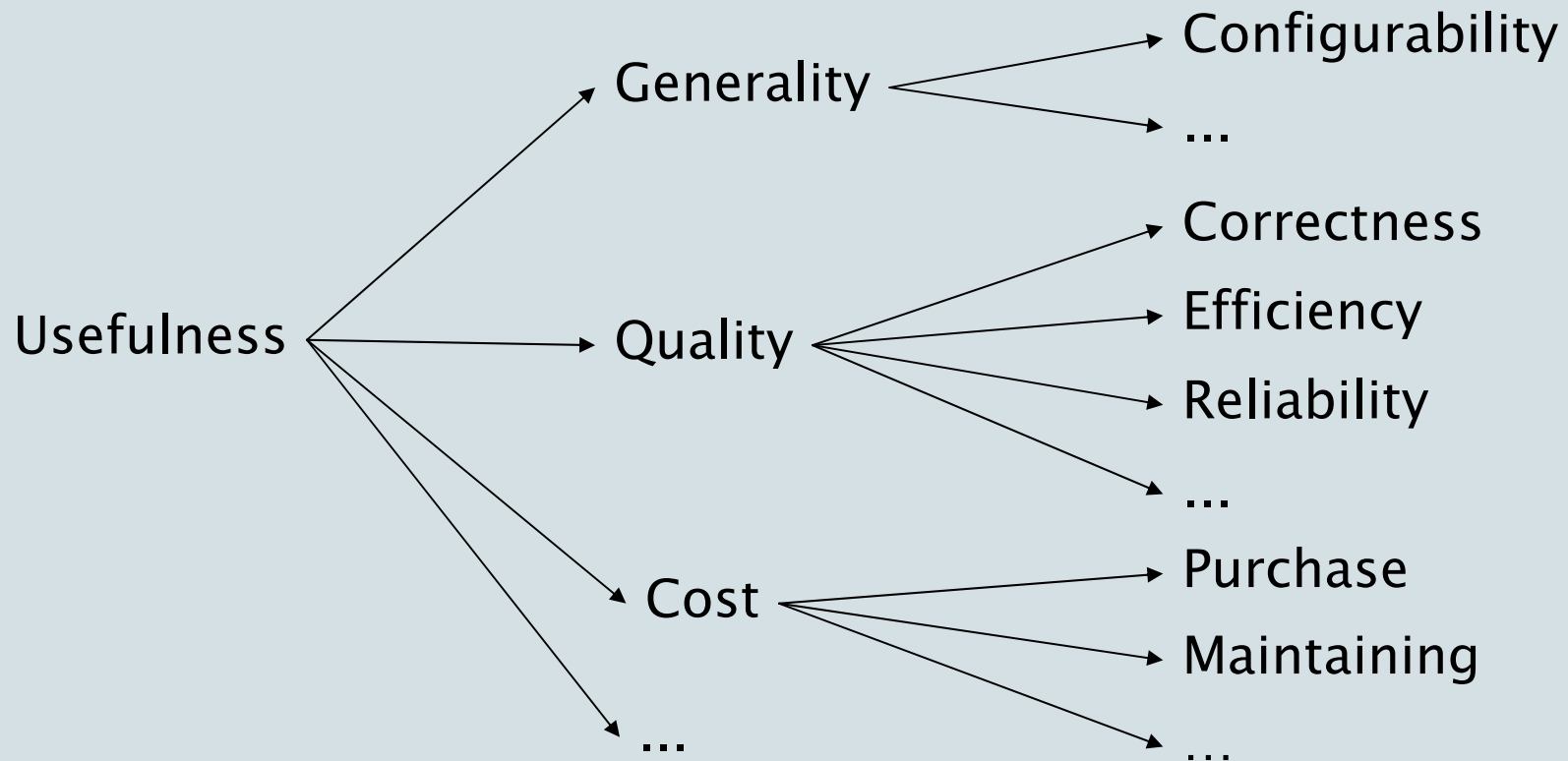
Effort needed to use an asset



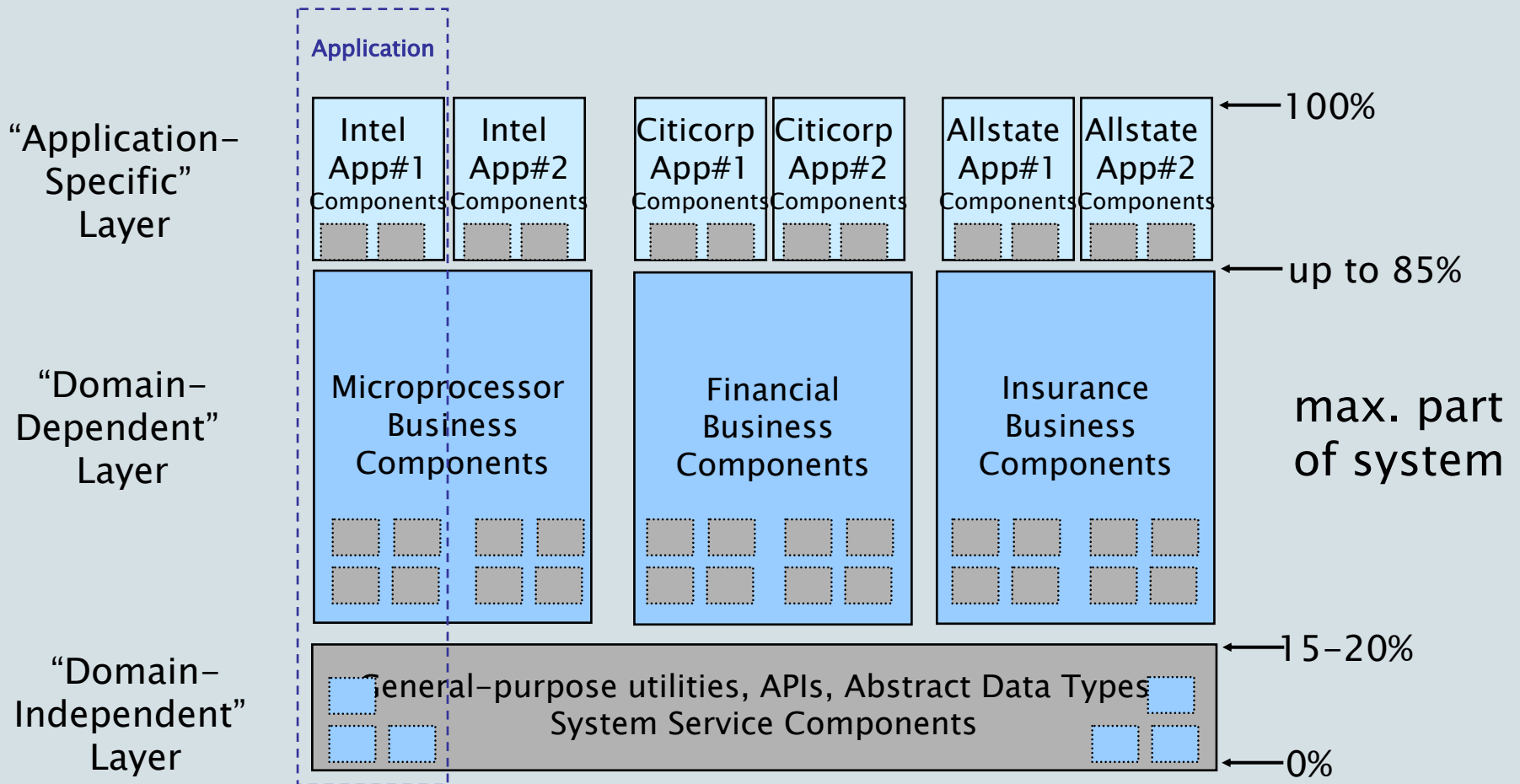
Usefulness

How well does it fit in a context?

‘frequency’ of suitability for use



The Three Classes of Software



This data indicates potential for reuse

Scope of Reuse

- Domain independent
 - ADT, GUI, Math-lib.
- Domain dependent
 - aircraft, finance, medical, ...
- Application specific / custom

typically no more than 20%
of any application

contributes the most
to reuse reported up to 85%

at least 15% of
any application

Effect of Scale of Reuse on Cost

Scale of Reuse affects potential ROI

Low	Nominal	High	Very High	XL High
No reuse	Across project	Across program	Across product-line	Across multiple product lines
1.00	1.15	1.33	1.53	1.75

Success factors in Reuse Management

- Focus on a narrow domain that has considerable commonality
 - Gather components based on domain analysis rather than ad-hoc
- A large volume of software must be developed
- The reuse program must be planned over a long period of time in order to amortize cost of new procedures and to populate the reuse repository
- provide tools for easing reuse

Success factors in Reuse Management

- Apply strict quality criteria for accepting assets in the reuse library
- Training of staff:
 - How to do reuse
 - Familiarity with available assets
- Management responsibilities:
 - Collect reuse metrics in order to manage reuse
 - Introduce reuse in an incremental, manner rather than as big-bang
 - Senior management commitment is required
 - Provide incentive

Economics of Software Reuse

How much does 1 line of code cost?

Outline

- What to measure?
- Reuse Cost Avoidance
- Reuse Cost Benefit analysis

References

- Measuring Software Reuse: Principles, Practices and Economic Models, Jeffrey Paulin, Addison Wesley, 1997

Asset Engineering Economics

It is widely accepted that developing a reusable asset costs more than developing an equivalent custom tailored asset.

Developing something reusable requires extra technical and process activities:

- Extra effort in generalizing interface
 - Adding customization / variability options
- Asset must satisfy more requirements
- More attention to documentation
- Reuse provides higher risk, hence more thorough testing is needed

Economic Models for Reuse

- Cost Avoidance
 - Emphasizes not spending resources
- Cost–Benefit Analysis
 - Determining the cost and benefits *of the reuse–activities*
- Return–on–Investment Models
 - Will the benefits exceed the investment (on the long term)?
 - Considers software development as a whole

Reuse leverage

$$\text{Reuse \%} = \frac{\text{Reused Software}}{\text{Total Software}} \times 100\%$$

Reuse is not a goal (& often too vague)

$$\text{Reuse Leverage for } \textit{Productivity} = \frac{\text{Productivity with Reuse}}{\text{Productivity without Reuse}} \times 100\%$$

Example: mis-use!

- A project reported 11k LOC of reuse
- 5120 lines came from one 10-line reusable macro in the same module

The original code:¹

```
Do i := 1 to 512
  MACRO (i);
```

- 1 Should be counted as 2 source instructions and 10 reused instructions.
- 2 Was reported as 512 source instructions and 5120 reused instructions.

“Unrolled” to yield:²

```
MACRO ( 1 );
MACRO ( 2 );
MACRO ( 3 );
MACRO ( 4 );
MACRO ( 5 );
MACRO ( 6 );
.
.
.
MACRO ( 509 );
MACRO ( 510 );
MACRO ( 511 );
MACRO ( 512 );
```

Reuse Metrics

Determining what to measure as reuse is emotional:
Organisations/teams want 'to look good'.

Cost – Benefit Analysis

Look at what cost are incurred by reuse (and by whom)

Look at the benefits generated by reuse (and by whom)

Cost–Benefit Analysis

C–B analysis requires assigning values to all known costs and benefits, including intangible items (e.g., quality).

$$\text{Benefits} = \sum_{i=1}^n B_i$$

$$\text{Costs} = \sum_{i=1}^n C_i$$

Consumer Benefits of Reusing Software

Reduced cost of

- design
- document
- implement
- testing
 - designing, documenting, implementing & executing tests
- maintenance
- tools
- equipment

Added revenue due to

- delivering product early
- improved sales

Consumer Costs of Reusing Software

Cost for

- performing domain analysis
- performing Cost–Benefit analysis
- procuring parts
 - search, negotiate, retrieve, buy
- assessing reusable assets
- configuring & integrating assets
- use & maintenance contracts
- modification
 - testing modified assets!
- reselling fees

Problem

- Difficult to get all the right data needed for doing Cost–Benefit analysis
- Data not available beforehand
- Perform a project with and without reuse?

➔ Let's look at a more coarse grained approach

Economics of Reuse: RCR

Relative Cost of Reuse (RCR)

The portion of the effort that it takes to reuse a component without modification versus writing it new.

The RCR can vary depending on several factors and range from about .03 up to about 0.4.

See also http://home.stny.rr.com/jeffreypoulin/html/reucalc_adv.html

Default value for RCR & Its assumptions

Poulin recommends an $RCR = 0.2$

This means that it takes about 20% of the effort to reuse software as it takes to write it new.

Assumptions:

- Programmer understands both source and target system
- Asset does not contain very complicated abstractions
- Asset consists of relatively small amounts of code
- Asset came with good documentation

Relative Cost of Reuse RCR

RCR is correlated with the complexity of the asset.

Higher complexity → Higher RCR

Modifying a component leads to a much higher RCR.

modified code < 25% → RCR of about 0.4

modified code > 25% → RCR of about 0.9

Economics of Reuse: RCWR

Relative Cost of Writing for Reuse (RCWR)

The ratio of the effort that it takes to write a reusable component versus writing it for one-time use only.

$$\text{RCWR} = \frac{\text{Effort of Development for Reuse}}{\text{Effort of Development without Reuse}}$$

The RCWR can vary depending on several factors and can range from about 1.0 up to about 2.2.

Empirical Data on RCWR

Relative contributions to RCWR (as reported by Tracz)

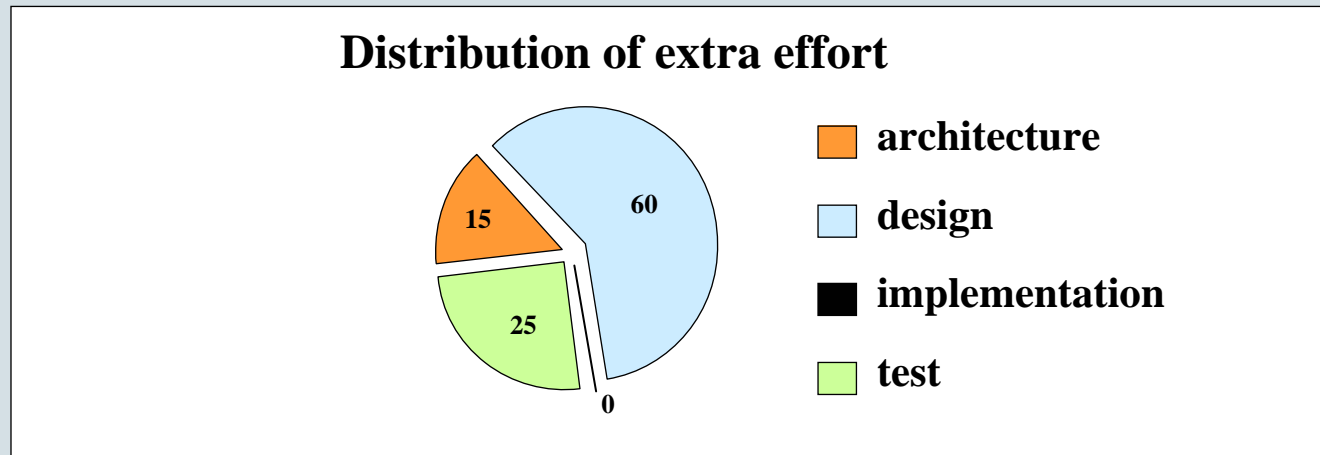
- 25% for generalisation
 - 15% for documentation
 - 15% for additional testing
 - 5% for library support and maintenance
- } Design & Specification

Total: 60%

Where does the extra effort (RCWR) go to?

Extra effort goes mainly to design & specification / documentation.

Cost of integrating components is about 30% of total development



Empirical data reported by Margano & Lindsey (1991)

Default Value for RCWR

Poulin recommends an $RCWR = 1.5$

This means that it takes about 50% additional effort to write reusable software.

Higher complexity → Higher RCWR

Measuring Software Reuse: Principles, Practices and Economic Models, Jeffrey Poulin, Addison Wesley, 1997

Using RCWR & RCR an approximate analysis can be made of the effects of reuse.

Concluding Remarks CBSE

- CBSE is a paradigm for design and construction of software
- CBSE is aimed at improving development–productivity (reduce time and effort)
- Components are building blocks that are part of a component model. May or may not be OO.
- If you have components, you can reuse them
- CBSE requires changes in development processes and development techniques
 - Separate system development from component development
 - Validation & certification are open problems
- There are different approaches for composition of software
 - Techniques for reasoning about composition are being developed

Discussion / Questions

Next week: Mark van de Brand

?

Concluding Remarks

- What have you learned?
- What would you like to see improved?
 - Book/notes/slides/...