Software-Workshop

Effiziente Entwicklung zuverlässiger Software und methodisches Instrumentarium

Veranstalter

Ada Deutschland, Gesellschaft für Informatik (GI), VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik (GMA), die Regionalgruppe Karlsruhe der GI und die Fachgruppe Ada der GI sowie der Fachausschuss Embedded Software der GMA, der VDI-Bezirksverein Karlsruhe und der Technologiepark Karlsruhe.

Organisation

Dr. Hubert B. Keller

Forschungszentrum Karlsruhe, Institut für Angewandte Informatik

Effiziente Entwicklung zuverlässiger Software und methodisches Instrumentarium

- Softwarebasierte Funktionen ermöglichen beispielsweise auf Basis des klassischen Maschinenbaus mit vertretbaren Kosten Funktionserweiterungen zu realisieren. Die Zuverlässigkeit softwarebasierter Funktionen und Systeme muss in gleicher Größe wie ohne Elektronik erhalten bleiben. Softwarebasierte Funktionen treten leider mehr in ihren Fehlern in Erscheinung als in den hierdurch neu eingebrachten Funktionalitäten.
- Die Herstellung und Einbettung von Software ist also der primäre Adressat zur Sicherung der Zuverlässigkeit von softwarebasierten Funktionen/Systemen, denn die verfügbare Hardware arbeitet weitestgehend stabil.
- Über 40% der im Betrieb auftretenden Fehler in der Software werden in der Analyse- und Spezifikationsphase eingeführt und ziehen eine Kostenexplosion bei der Wartung nach sich.
- Der Workshop will diese Probleme beleuchten und Lösungen und Wege in Form praktikabler konstruktiver und analytischer Methoden zu einer zuverlässigen Software aufzeigen.

- Dr. Hubert B. Keller
  Forschungszentrum Karlsruhe, Institut für Angewandte Informatik

"Efficient Development of Reliable Software and Related Methods"

- Software based functions make it possible to extend functions, on the basis of classic engineering approaches, and to do so at acceptable costs.
- The reliability of software-based functions should be in the same range as that of functions without electronics, but the new software-based functions seem to be fault-inducing rather than beneficial. Therefore, improvements in the development and operation of software are the primary objective for ensuring reliability, because the underlying hardware usually works well.
- More than 40% of the operational failures are induced during the requirements phase, resulting in overwhelming maintenance costs.
- The workshop addresses these problems and shows solutions to overcome them based on practicable analytical and constructive methods.

Programm

- Begrüßung

- Sitzung 1 (Leitung: Keller)
  - Erwin Reyzl, Siemens München: Formal Model Verification in the Industrial Software Engineering Practice
  - Frank Ortmeier, Uni Augsburg: Modell-basierte Sicherheits-analyse
  - Eberhard Kuhn, Andrena Objects Karlsruhe: ISO 9001 + ISIS: ein agiles Qualitätsmanagementsystem

… Programm

- Sitzung 2 (Leitung: Dencker)
  - Jose Ruiz, AdaCore France: Ada 2005 for real-time, embedded and high-integrity systems
  - Tom Grosman, AONIX: Hibachi - the Eclipse Ada Development Toolset (entfallen)
  - Christoph Diesch, T-Systems: Testkonzepte und Entwicklungsprozesse für sicherheitsrelevante Systeme/Software

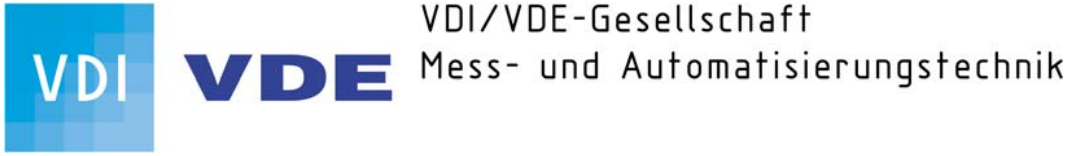
- Sitzung 3 (Leitung: Sommer)
  - Hubert B. Keller, FZK: Einführung modellbasierter Techno-logien im industriellen SWE Umfeld
  - Peter Dencker, Etas Stuttgart: Modellbasierte Entwicklung effizienter SW in der Automobilindustrie
  - Steffen Becker, FZI Karlsruhe: Software-Komponenten und -Architekturen für die ingenieurmäßige Software-Entwicklung

Sitzungen

- Sitzungen Fachgesellschaften
  - Gesellschaft für Informatik, Fachgruppe Ada
  - Förderverein Ada-Deutschland e.V.
  - Gesellschaft für Mess- und Automatisierungstechnik, Fachausschuss Embedded Software

Danksagung

Die Veranstalter und Organisatoren danken
für die freundliche Unterstützung.



(1) Die Kooperation von Forschungszentrum Karlsruhe GmbH und Universität Karlsruhe (TH)

Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft

SIEMENS

Corporate Technology

Formal Model Verification in the Industrial Software Engineering

Erwin Reyzl, Siemens AG, Corporate Technology
Vladimir Okulevich, Siemens Russia, Corporate Technology

Software Workhop
“Efficient Development of Reliable Software and Related Methods”
Karlsruhe, Germany
24th January 2008

Dependable Software and Siemens Products

Dependability of a computing system is the ability to **deliver services that can justifiably be trusted**.

Ref. : A. Avizieniz, J.-C. Laprie, B. Randell: Fundamental Concepts of Dependability

All Siemens Divisions develop and sell products that perform mission-critical operations.

Most of these products contain an ever **increasing software part**.

Dependability is decisive for our commercial success.

Dependability yields higher confidence and acceptance, and is pre-condition to market access.

Following engineering standards gives **evidence of** a product's quality and **trustworthiness**.

Dependability relies on an integral **management & engineering** approach.

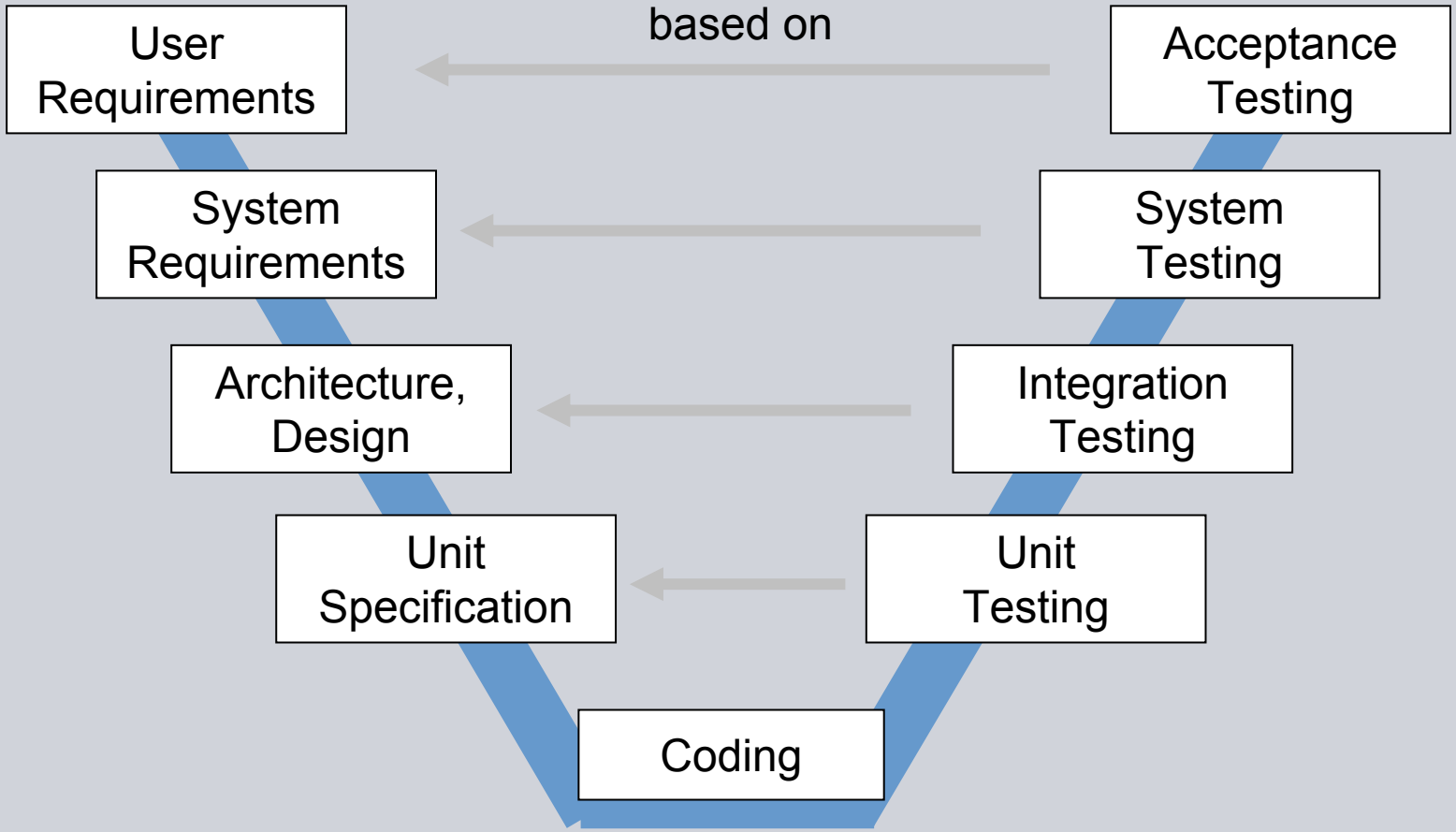
SIEMENS

Dependability Competence Team at Siemens Corporate Technology

- **Safety / RAMS Engineering**
- **RAMS Analysis & Assessment**
- **Requirement Engineering**
- **Model Driven Development**
- **Code Quality Management**
- **Software Testing & Verification**
- **Validation & Certification**
- **Software Architectures**
- **Real-time Technology**
- **Embedded Platforms**
- **Fault-Tolerance**
- **…**

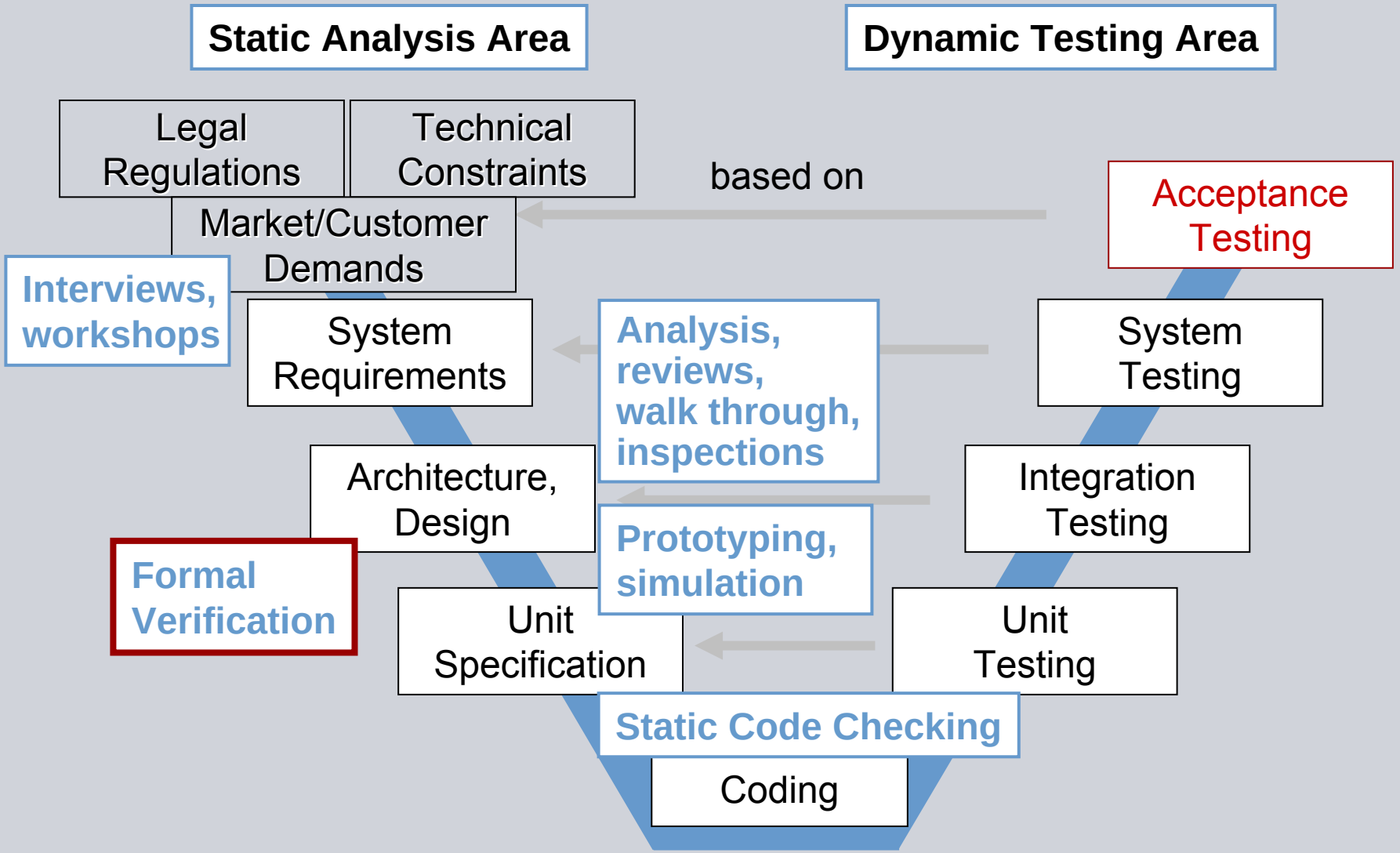
Engineering of high-quality software
Test levels – example V model

SIEMENS



Engineering of high-quality software
Test levels, Verification & Validation – A closer View

SIEMENS



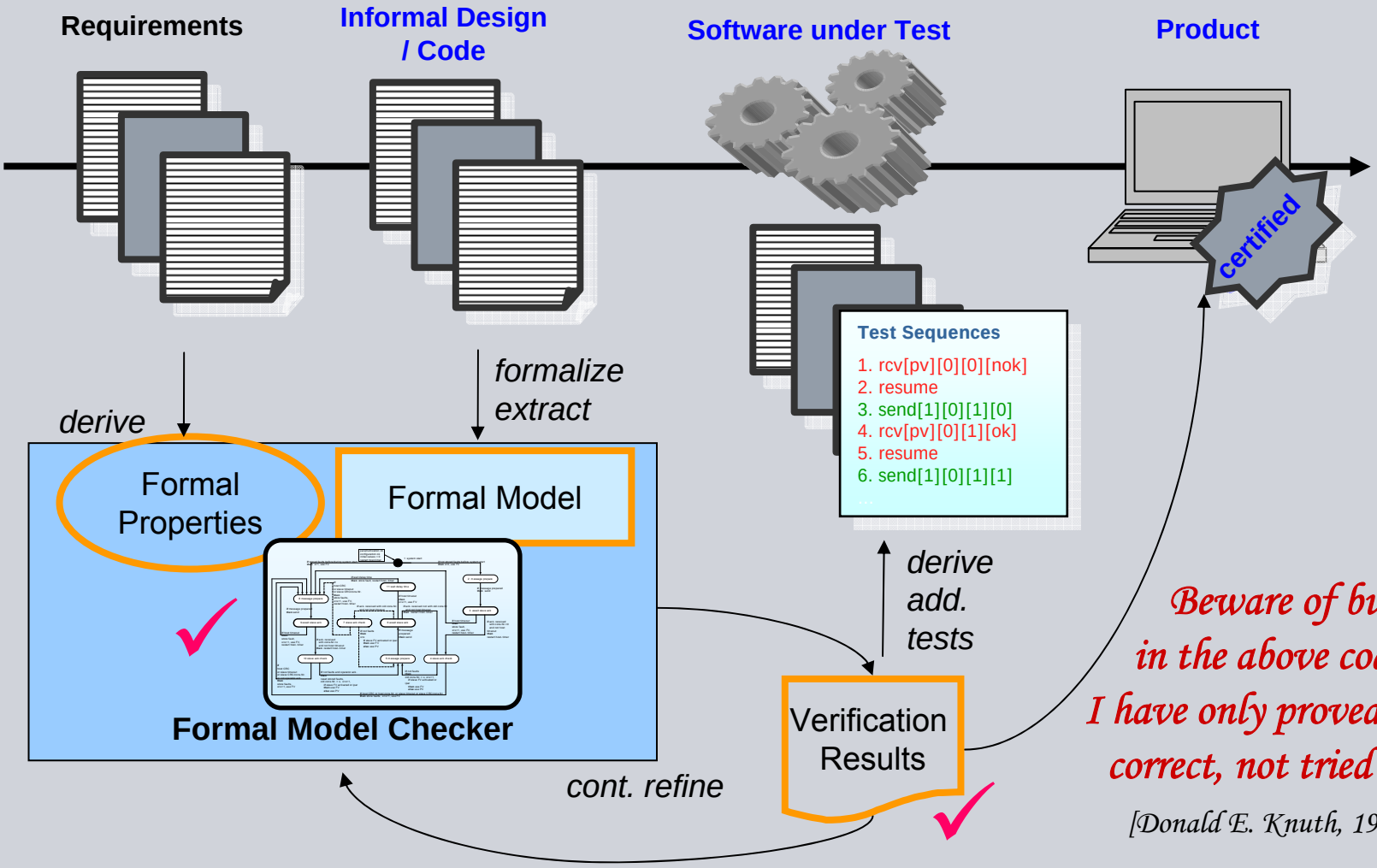
SIEMENS

Formal Verification and Related Terms

- **Software Verification, Software Safety Validation (IEC61508)**
  Verification & Validation, Static Analysis& Dynamic Testing
  - See http://en.wikipedia.org/wiki/Formal_verification :
- **Validation**: "Are we building the right product?",
  i.e., does the product do what the user/the application really requires?
- **Verification**: "Are we building the product right?",
  i.e., does the product conform to the specifications?
- **The verification process consists of static and dynamic parts.**
  E.g., for a software product one can inspect the source code (static) and run against specific test cases (dynamic). Validation usually can only be done dynamically.
- **Formal Method (IEC61508)**
  Formal Specification
- **Formal Proof (IEC61508)**
  Formal Verification, Model Checking/Theorem Proving
  - See http://en.wikipedia.org/wiki/Formal :
    formal methods in computer science, including:
    formal specification describes what a system should do
    formal verification proves correctness of a system

Embedding Formal Verification into Software Development Life-Cycle

SIEMENS

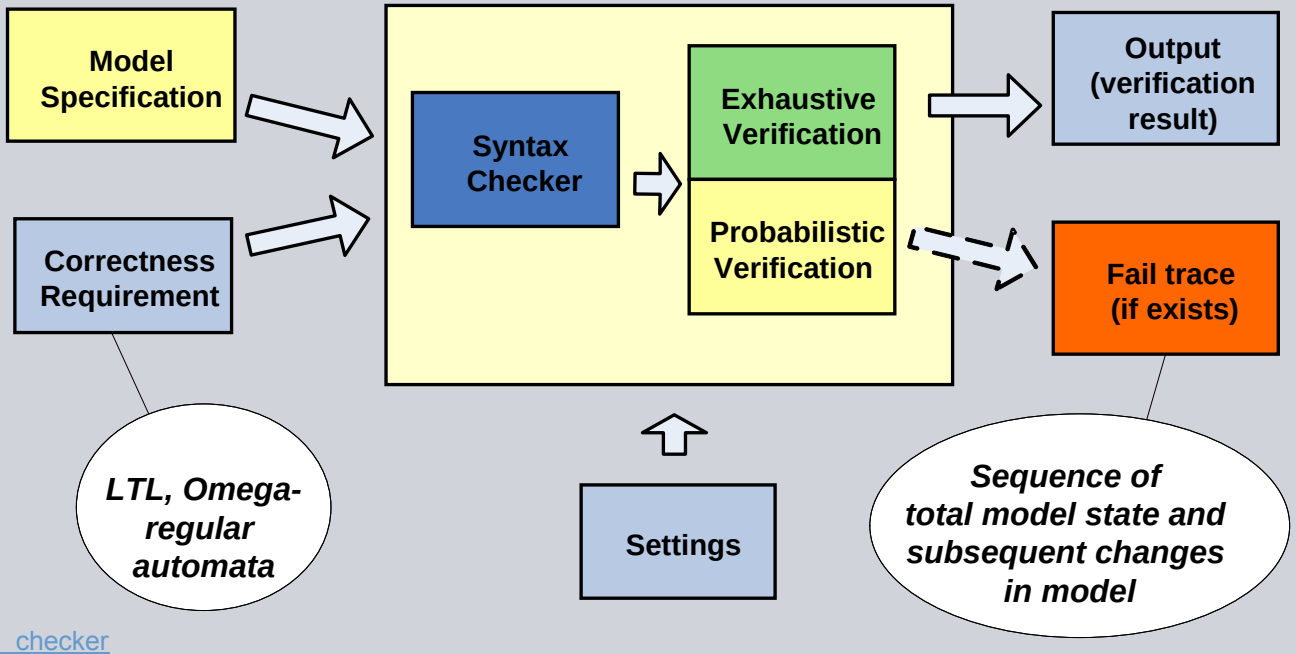


Formal Model Verification – SPIN tool

SPIN (Bell Labs, G. Holzmann):
Characteristics of method:
1) exhaustive verification
2) space compression
3) probabilistic verification (hashing)

Model in PROMELA (C-like language, CSP-based) is automatically translated into the extended FSMs.
These FSMs are verified to be correct according to correctness requirements.
Correctness requirements are presented in Linear Temporal Logic (LTL)

SPIN-based technologies are used:
- **Bell-Labs (network switches, OS Plan 9)**
- **NASA (Cassini mission at Saturn, Deep Space 1)**
- **Siemens CT**

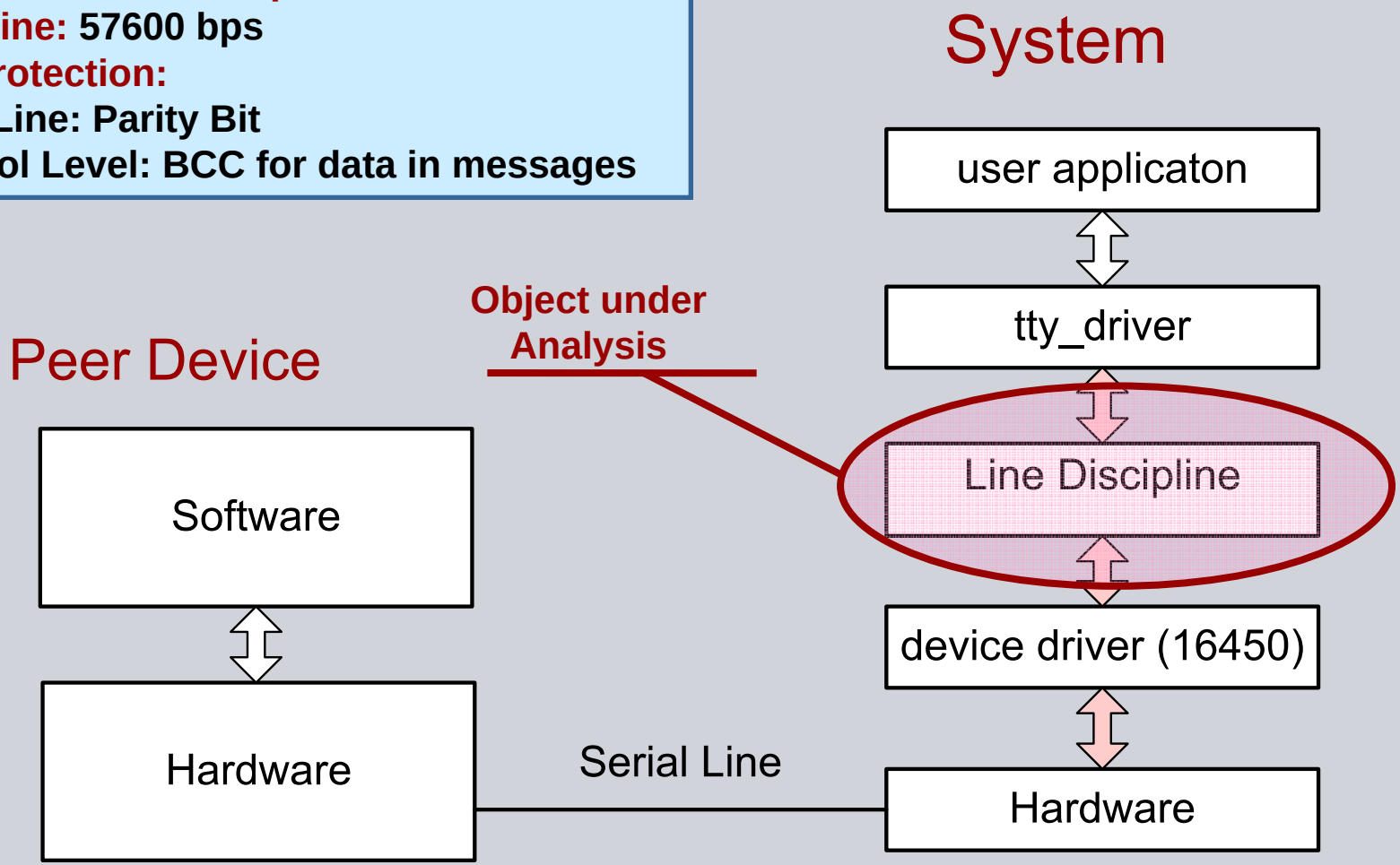


SPIN website
http://spinroot.com/spin/whatispin.html
Wikipedia
http://en.wikipedia.org/wiki/SPIN_model_checker

Example A
System Structure

SIEMENS

Customized Line Discipline
Serial Line: 57600 bps
Error Protection:
- **Serial Line: Parity Bit**
- **Protocol Level: BCC for data in messages**

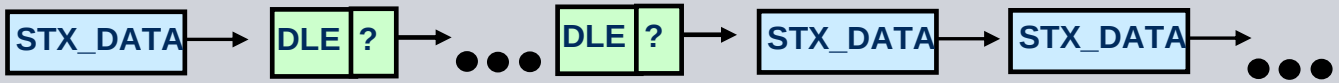


Example A
Safety Property (1)

SIEMENS

Safety Property: Any order of STX_DATA and DLE messages will be correctly received and Receiver achieves TFirstChar state after TDLE2 state.

Modeling Solution: Sender generates valid messages STX_DATA and DLE in all possible sequences.



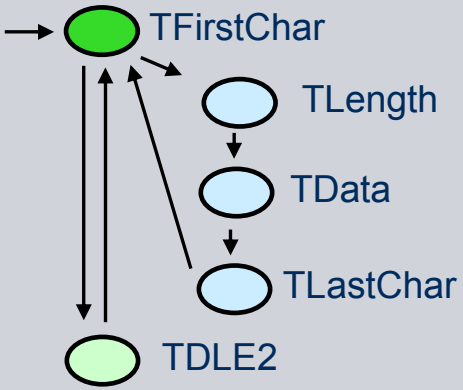
OK: Property Valid

Property Parts:

#define a (ReceiveState==TDLE2)
#define b (ReceiveState==TFirstChar)

Safety Property for Model Verification in Linear Temporal Logic:

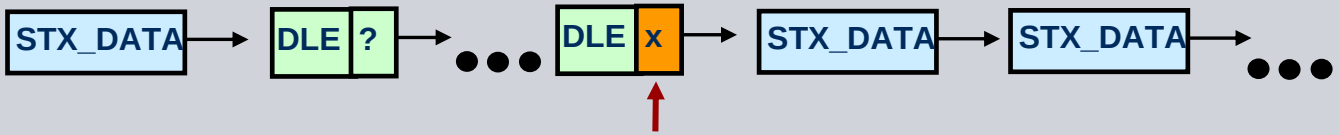
[] (<> a -> <> b)



Example A
Safety Property (2)

SIEMENS

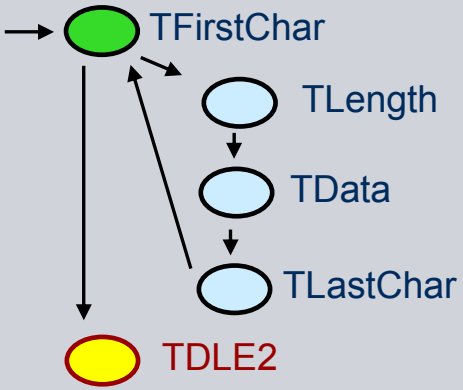
Safety Property: Any order of STX_DATA and DLE messages will be correctly received and Receiver achieves TFirstChar state after TDLE2 state if the noise char comes suddenly.
Modeling Solution: Sender generates valid messages STX_DATA and DLE in all possible sequences. **We allow receiving of noisy char.**



Error: Property Invalid

Property Parts:
#define a (ReceiveState==TDLE2)
#define b (ReceiveState==TFirstChar)

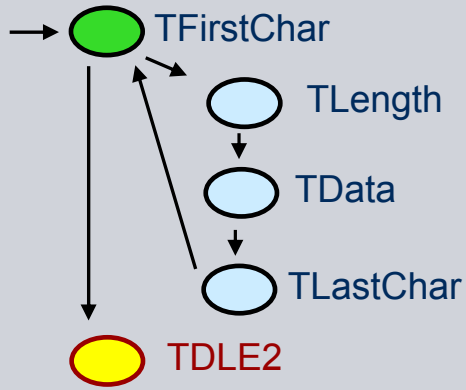
Safety Property for Model Verification in Linear Temporal Logic:
[] (<> a -> <> b)



Example A
Error Analysis for Property (2) : Source Code

SIEMENS

```
switch (theDriverData->ReceiveState) {
  case TFirstChar:
    switch (theChar) {
      (…)
      case TDRVDLE:
        theDriverData->ReceiveState = TDLE2; break;
    }
(…)
  case TDLE2:
    if(theChar == '?' || theChar == TDRVENQ)
      theEvent = TDLEReceived;
    if(atomic_read(&theDriverData->WabtCounter) == 50) {
      atomic_set(&theDriverData->WabtCounter, 0);
      theDriverData->ReceiveState = TFirstChar;
    }
break;
(…)
```



Error:
Property Invalid

Explanation:
SPIN shows that DRV could be blocked if noisy (incorrect) char came after TDRVDLE.

NOTE: the real driver will wait 50 timeouts (~5-20 sec) before it starts receiving of next telegram (e.g. STX_DATA)

Example A
FMV Results for the Driver

SIEMENS

Operational threats identified by SPIN:

- Long delay identified if 1 noise char comes
- Data loss within telegram receiving found

Model-building review:

- Performance bottleneck could create high interrupts latency
- Wrong API-version calls
- “Magic Constants” used in code
- “Dead Code” identified

Summary (1)
Objects and Focus Setting

SIEMENS

Objects under Analysis:
- Protocols and Interfaces (especially under construction)
- Interacting components (e.g. new architecture or critical mechanism)
- Data access and control logic in parallel and distributed system

Fault Types:
- Deadlock/Livelock
- Endangered Safety
- Integrity violation
- Correctness violation
- Non-expected communication order
- Race Conditions
- …

Additional Focus set on:
- System Initialization
- Restart of Components
- Communication Delays and Faults

Robustness Aspects:
- Standard operation
- Unpredictable rare impact
- "Aggressive/Noisy Environment"

SIEMENS

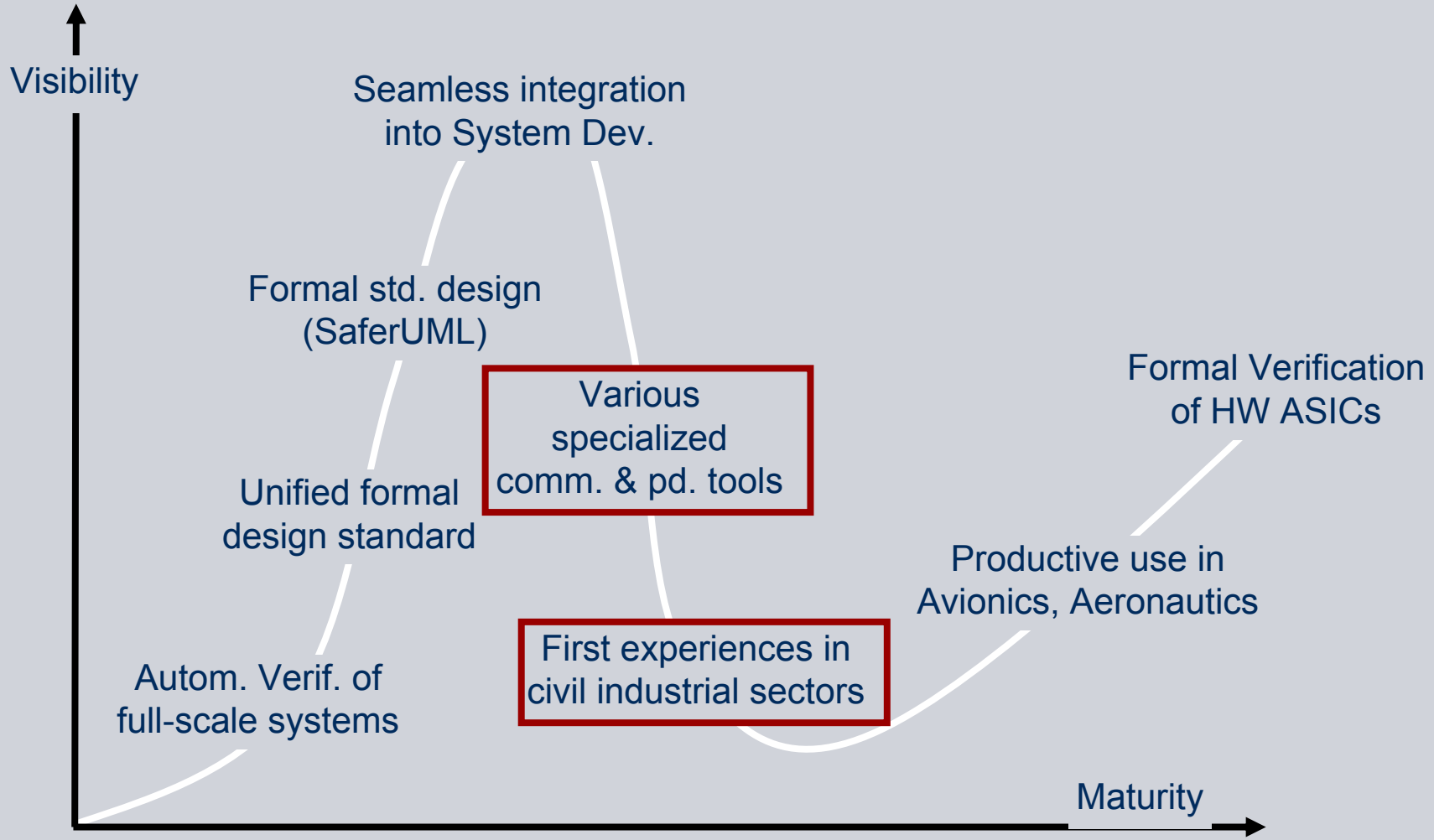
Dos & Don'ts

- Improve verification capabilities in early phases by introducing formal techniques.
- Increase precision in specifications. Apply formal techniques at design and fight ambiguities.
- Promote formally motivated checks into standard peer reviews.
- Focus formal techniques on architectural hotspots: complex, critical, risky, central parts
- Exploit formal results for test case definition: use failure traces to focus tests on design flaws.
- Don't believe in wonders. Formal verification is not cheap and needs invests in early phases.
- Don't act without concept. Tools need evaluation, and competence needs to be built.
- Don't apply formal techniques as rescue belt. It's not to patch ambiguous results from less mature processes.
- Don't believe you will not need to test your software any more. Formal verification does not replace testing phases.
- Don't believe your software is completely verified. You only proved that a model fulfills certain properties. That's it – no more, no less.

Don't split theory and practice.
Closely align work of formal teams and safety/development teams.

SIEMENS

The Formal Hype Cycle



SIEMENS

Challenges for Formal Model Verification

Challenges

- Decrease modeling efforts
- Increase usability, reduce qualification degree needed
- Integration with tools for software development
- Traceability from modeling phase to testing phase
- Automated properties and models extraction from heterogeneous input material
- Again coverage and completeness issues "How much will be sufficient?"
- …

Model Checking gains importance:

"The behavior of even nonbuggy distributed applications can easily defy our human reasoning skills."

Logic Verification of ANSI C code with SPIN
Gerard J. Holzmann

Summary (2)
Reasons for Formal Model Verification

SIEMENS

Standards recommend formal methods and proofs
e.g. IEC61508 : SIL2/3/4 - for design, verification, safety validation.

Formal methods emerge at the industry sector
easier to use tooling (Open Source, Tool Vendors), best practices (space/military, avionics, transportation/automotive, Microsoft).

Formal methods improve precision within development, capturing and ensuring functional and **non-functional properties**.

Early correctness proof of design concepts prevents design faults to propagate into development, test and operation phase.

Byzantine failures with hard to identify root-causes often are the consequence of weakly defined or misunderstood requirements.

Environmental impact and sporadic influences which are hard to test can be incorporated into formal models.

Stronger evidence of safety-related claims; amends test results and **improves acceptance by certification authorities**.

SIEMENS

Conclusions

•Increasing complexity and importance of software

More and more safety-relevant functions, which nowadays might be executed manually by human, will be realized in software and taken over automatically by the technical system.

•Traditionally software plays a subordinate role

In systems engineering and also in relevant standards the current perspective on software is that of an subordinate element. This is expected to change with the growing pervasiveness of software especially in safety-relevant development.

•Formal verification in practice applied to selected software parts

In the current practice formal verification is applied to verify selected system aspects. It already proved usefulness and applicability**.**

•Cost and complexity of formal techniques are further high

Up to now formal verification is not an easy-to-use technique. At this time it is not seen to enable a complete software/system verification.

•Formal Verification does not/will never replace systematic testing

Formal verification adds precision to the traditional verification process. It extends, but does not replace rigorous testing. Size limitations and abstractions of models through formal verification are to be carefully verified in reality.

Thank you for your attention.

Do you have some questions ?

Backup

Fundamental Concepts of Dependability (A. Avizieniz, J.-C. Laprie, B. Randell)

SIEMENS

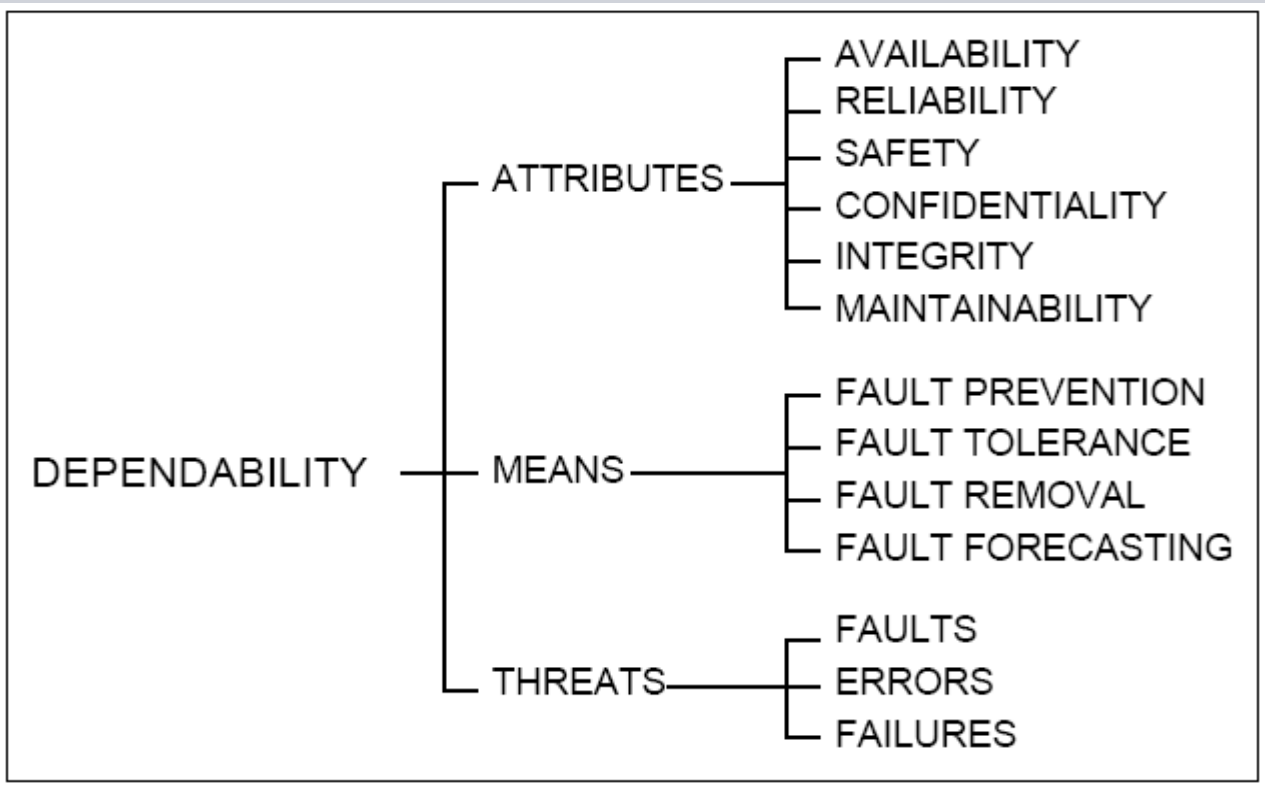


Figure 1 - The dependability tree

Computing systems are characterized by four fundamental properties: functionality, performance, cost, and dependability.

Concepts of Dependability developed by A. Avizieniz, J.-C. Laprie, B. Randell

SIEMENS

Definitions: Dependability Attributes

Dependability is an integrative concept that encompasses the following system attributes:

- **Availability:** readiness for correct service
- **Reliability:** continuity of correct service
- **Safety:** absence of catastrophic consequences on the user(s) and the environment
- **Confidentiality:** absence of unauthorized disclosure of information
- **Integrity:** absence of improper system state alterations
- **Maintainability:** ability to undergo repairs and modifications

Compound attributes:

- **Survivability:** system capability to resist a hostile environment so that it can fulfill its mission (MIL-STD-721, DOD-D-5000.3)
- **Security:** Dependability with respect to the prevention of unauthorized access and/or handling of information (Laprie, 1992)

*** RAM / RAMS:** acronyms for reliability, availability, maintainability, and safety

SIEMENS

IEC61508-3: Formal Methods

Formal methods are a specification technique.

Formal methods (see IEC61508-7, C2.4) are for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z.

Formal methods are recommended (R SIL2/3, HR SIL4) for

- 7.2/Table A.1:
  Software safety requirements specification
- 7.4.3/Table A.2:
  Software design and development: software architecture design
- 7.4.5/Table A.4:
  Software design and development: detailed design
- 7.7/Table A.7/Table B5
  Modeling in the context of software safety validation

Sometimes mixed up with semi-formal methods e.g. finite state machines (FSM)

- semi-formal methods (table B.7):
  Logic/function block diagrams, sequence diagrams, data flow diagrams, finite state machines/state transition diagrams, e.a.

SIEMENS

IEC61508-7, C2.4: Formal Methods (ii)

•Focus: Logic/HW

- **HOL – Higher Order Logic for HW verification**
- **Temporal logic – Formal demonstration of safety and operational requirements**

•Focus: Sequential processes

- **OBJ – Algebraic specification of operations on abstract data types (ADT, similar to ADA packages).**
- **Z – Specification language notation for sequential systems**
- **VDM – Vienna Development Method (VDM++ concur. extension)**

•Focus: Communicating concurrent processes

- **LOTOS, extends CCS – Calculus of Communicating Systems**
- **CSP – Communicating Sequential Processes**

•Other semi-formal techniques (see B.2.3.2)

- **Finite state machines/state transition diagrams for control structures**
- **Petri nets (graph theory) for concurrent, asynchronous control flow; extension: time concept, data/information flow**

SIEMENS

IEC61508-3: Formal Proofs

Formal proofs are recommended (R SIL2/3, HR SIL4) for

- IEC61508-3/7.9/Table A.9: Software verification

Formal proofs are a static means for software verification

NOTE 3 – In the early phases of the software safety lifecycle verification is static, for example inspection, review, formal proof. When code is produced dynamic testing becomes possible. It is the combination of both types of information that is required for verification.

For example code verification of a software module by static means includes such techniques as software inspections, walk-throughs, static analysis, formal proof. Code verification by dynamic means includes functional testing, white-box testing, statistical testing.

It is the combination of both types of evidence that provides assurance that each software module satisfies its associated specification.

Sometimes mixed up with static analysis e.g. symbolic execution:

- Static analysis (table A.9, table B.8): e.g. Walk-through/design reviews, control flow / data flow analysis, or symbolic execution, e.a.

Outline of Formal Model Verification

Main Steps:

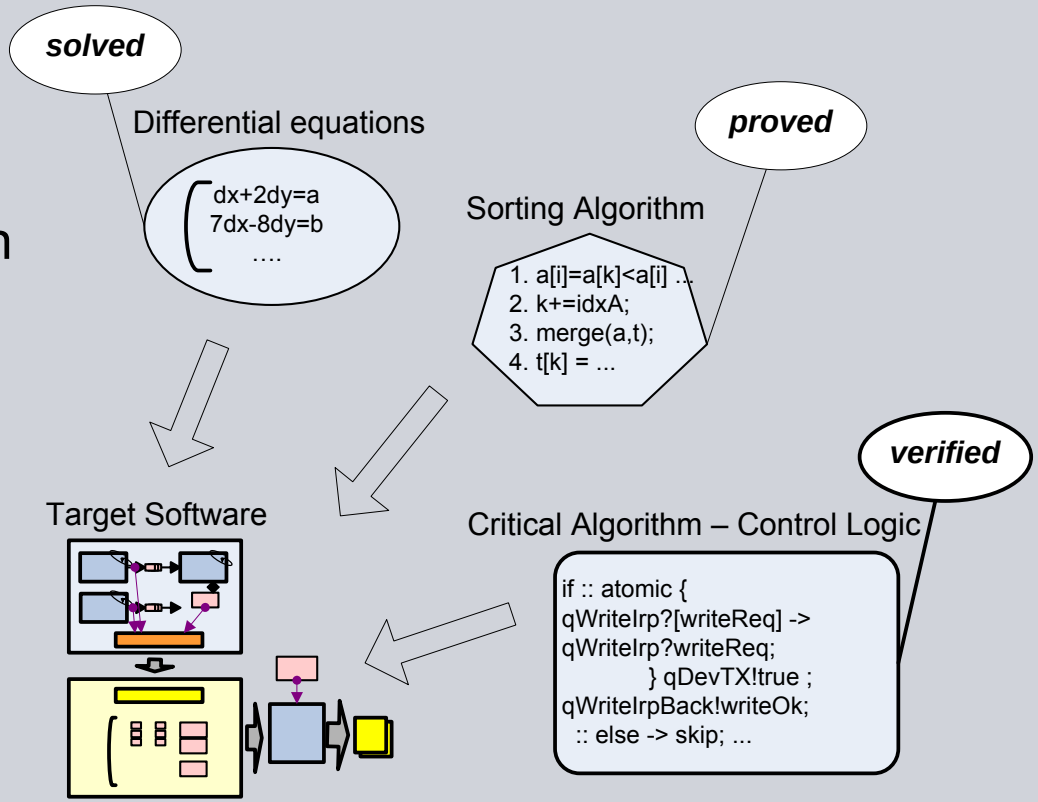
Objects under Analysis
identification in software project

Correctness properties definition
for Objects under Analysis

Creation of model for
Objects in PROMELA

Model verification by SPIN and
findings analysis

Report preparation on
verification results



Further promising actions:

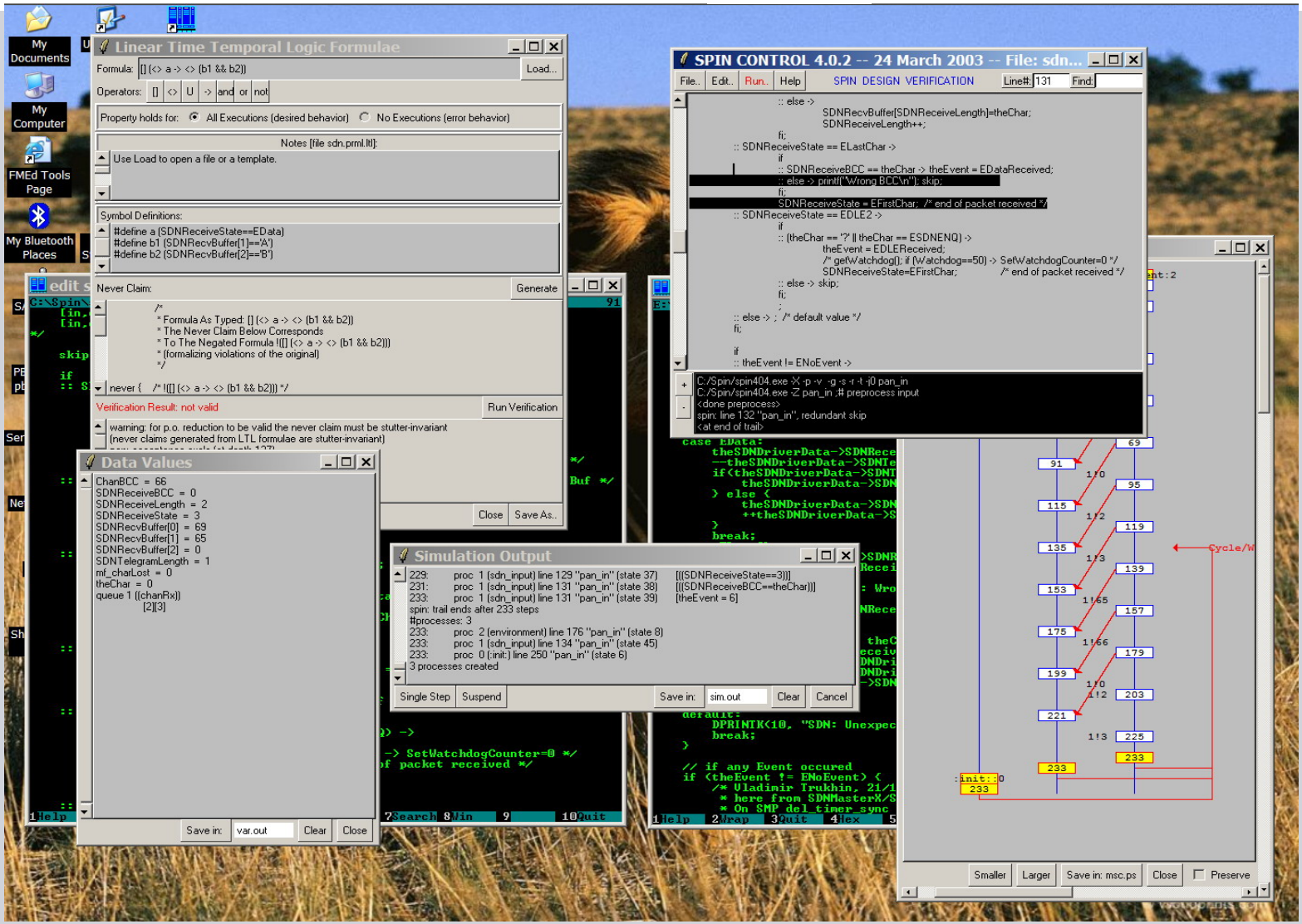
- Automate procedure of model creation from C/C++ sources

SPIN model checker (ii)

SIEMENS

Wikipedia http://en.wikipedia.org/wiki/SPIN_model_checker
SPIN is a tool for software model checking. It was written by Gerard J. Holzmann and others, and has evolved for more than 15 years. SPIN is an automata-based model checker. Systems to be verified are described in Promela (*Pro*cess *Me*ta *La*nguage), which supports modeling of asynchronous distributed algorithms as non-deterministic automata. Properties to be verified are expressed as Linear Temporal Logic (LTL) formulas, which are negated and then converted into Büchi automata as part of the model-checking algorithm. In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user. Since 1995, (approximately) annual SPIN workshops have been held for SPIN users, researchers, and those generally interested in model checking. In 2001, the Association for Computing Machinery awarded SPIN its System Software Award.
Holzmann, G. J., *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004. ISBN 0-321-22862-6.
SPIN website http://spinroot.com/spin/whatispin.html

Snapshot of SPIN Verification Screen



SPIN model checker (iii) – References

SIEMENS

Wikipedia http://en.wikipedia.org/wiki/SPIN_model_checker
SPIN Website http://spinroot.com/spin/whatispin.html
An overview paper of Spin, with verification examples, is:
The Model Checker Spin,
IEEE Trans. on Software Engineering,
Vol. 23, No. 5, May 1997, pp. 279-295.
(PDF)
The automata-theoretic foundation for Spin:
An automata-theoretic approach to automatic program verification,
by Moshe Y. Vardi, and Pierre Wolper,
Proc. First IEEE Symp. on Logic in Computer Science,
1986, pp. 322-331.
(PDF)

SIEMENS

Linear Temporal Logic : SYNTAX

LTL - Linear Temporal Logic
Best to specify safety and correctness properties
See also http://en.wikipedia.org/wiki/Linear_Temporal_Logic

SYNTAX
Grammar: ltl ::= opd | (ltl) | ltl binop ltl | unop ltl

Unary Operators (unop):
- [] (the temporal operator *always*),
- <> (the temporal operator *eventually*),
- ! (the boolean operator for *negation*)

Binary Operators (binop):
- U (the temporal operator *strong until*)
- V (the dual of U): (p V q) == !(!p U !q)
- && (the boolean operator for *logical and*)
- || (the boolean operator for *logical or*)
- -> (the boolean operator for *logical implication*)
- <-> (the boolean operator for *logical equivalence*)

Operands (opd): Predefined: true, false

SIEMENS

Linear Temporal Logic (ii)

Extension of classical logic ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\forall$, $\exists$)

- works over an (infinite) sequence of states

New operators:

$\circ$ *next time*

$\circ\ \varphi$ φ holds at time t + 1

$\Diamond$ *eventually*

$\Diamond\ \varphi$ φ holds at some time t + n

$\Box$ *always*

$\Box\ \varphi$ φ holds for all future times t + n

U *until*

$\varphi\ U\ \psi$ φ holds for all future times until the time where ψ holds

$\cap$ *release*

$\varphi \cap \psi$ either φ holds forever, or until φ and ψ holds at the same time

Modell-basierte Sicherheitsanalyse

Dr. Frank Ortmeier
Lehrstuhl Softwaretechnik und Programmiersprachen
Universität Augsburg

Warum?

- Trends bei sicherheitskritischen Systemen
  - Steigende Komplexität
  - Zunehmende Kritikalität der Konsequenzen
  - Immer mehr Funktionalitäten werden von Software übernommen
- Konsequenzen
  - Sicherheitsanalyse wird zunehmend schwieriger
  - Anforderungen an die Qualität der Resultate steigt


Ariane 5


Tschernobyl

Boeing 737 - Chicago

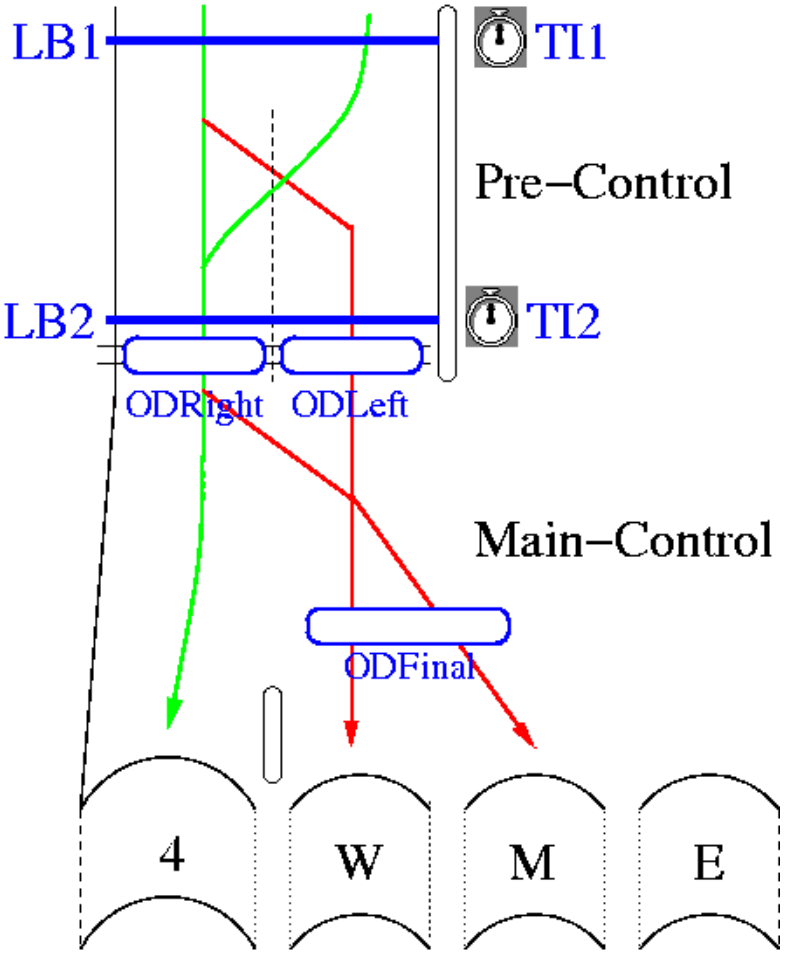

Airbus – Puerto Plata

Ein Beispielsystem

- Beitrag zu einer Industriekooperation mit (Siemens)
- Erweiterung des neuen Hamburger Elbtunnels um eine 4. Röhre
- Inbetriebnahme 2004

Die Höhenkontrolle

- Komponenten:
  - Lichtschranken (LB)
  - Überkopfdetektoren (OD)
  - Timers (TI)
  - Steuersoftware
- Umgebung:
  - Normale Autos
  - Hohe Fahrzeuge (HV)
  - Überhohe Fahrzeuge (OHV)



Sicherheitsziele

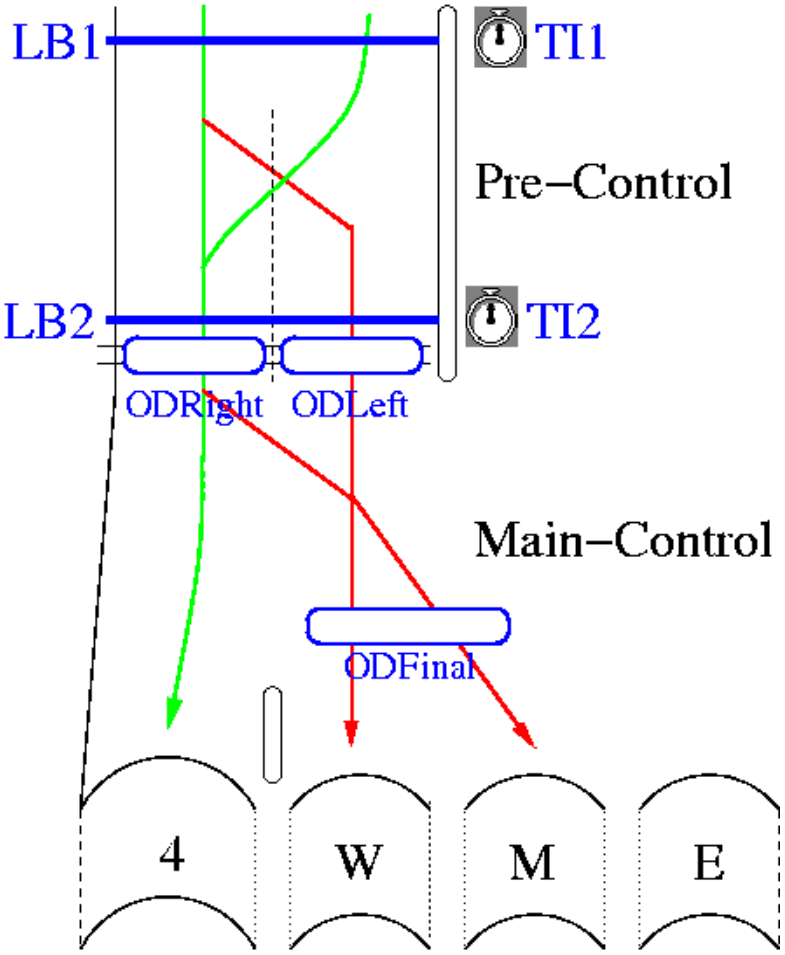
- Antagonistische Sicherheitsziele:
  - „Keine Kollisionen“

    falls ein überhohes Fahrzeug auf eine falsche Röhre zu steuert, muss ein Nothalt signalisiert werden.

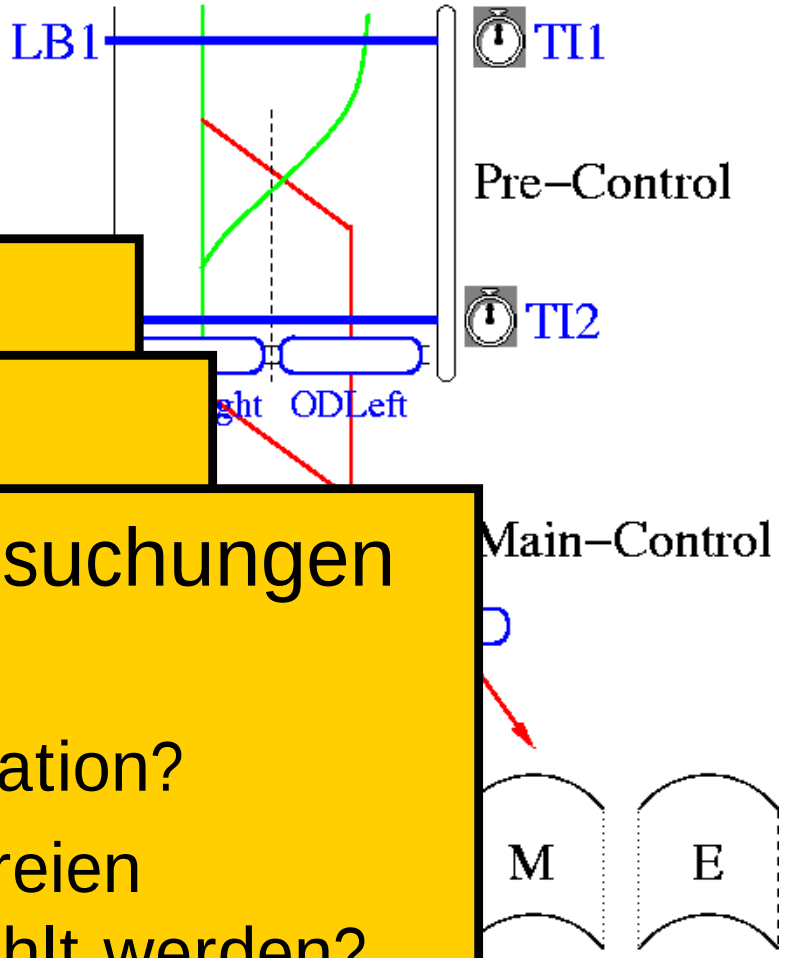
    (hazard Kollision)
  - „Keine Fehlalarme“

    ein Alarm darf nicht ausgelöst werden, falls kein OHV auf einer falschen Route fährt.

    (hazard Fehlalarm)



Sicherheitsrelevante Fragen



Frage 1: Funktionale Korrektheit

-
-
-
-

Frage 2: Fehlertoleranz

-
-

Frage 3: Quantitative Untersuchungen

- Wie sicher ist das System?
- Was ist die optimale Konfiguration?
- Welche Werte sollen für die freien Parameter des Systems gewählt werden?

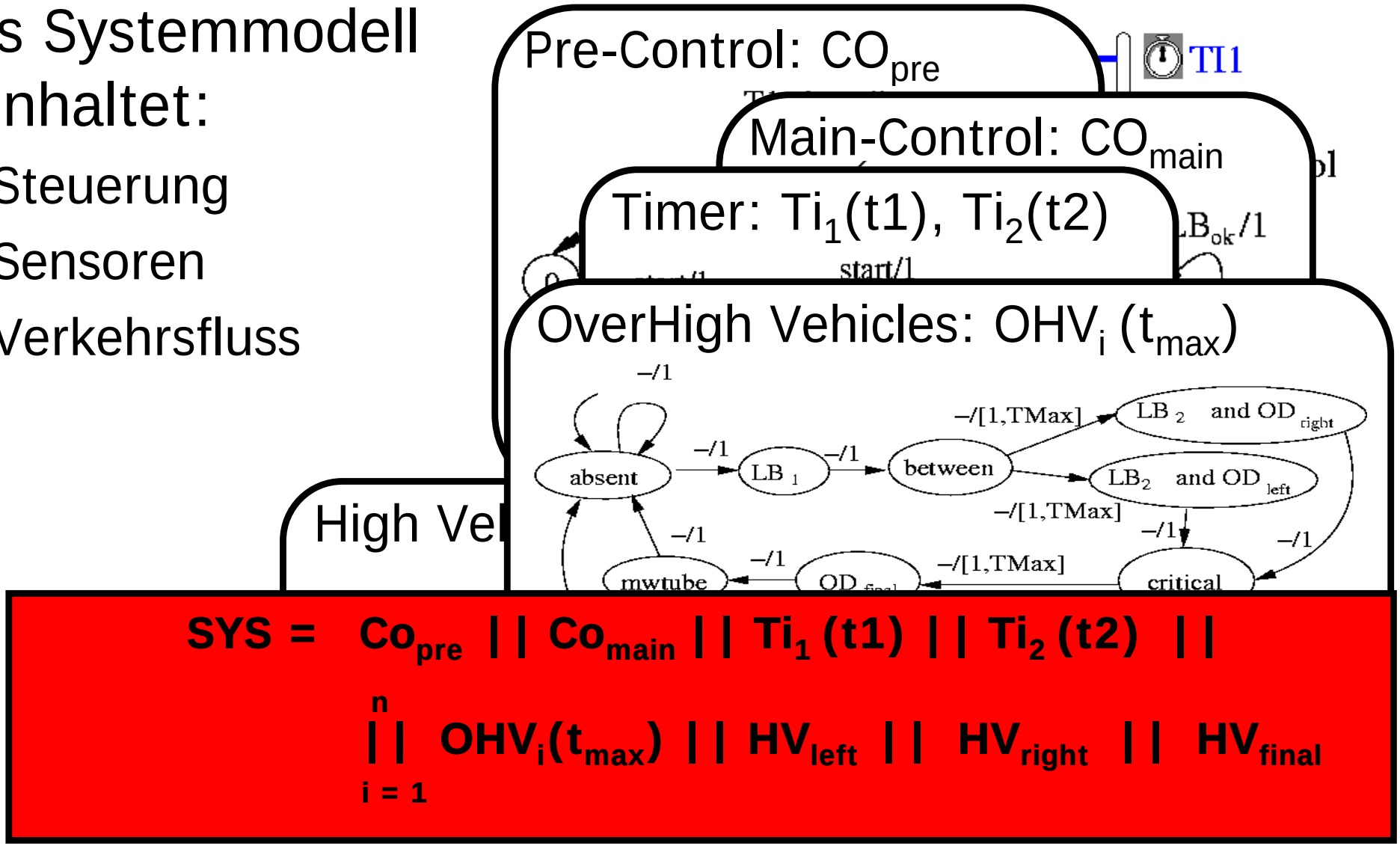
Modell-basierte Sicherheitsanalyse

- Modell-basierte Sicherheitsanalyse beruht auf der Untersuchung eines Modells des Gesamtsystems

- Dazu gehören Modelle der
  - Steuersoftware
  - Hardwarekomponenten
  - Umgebung
  - Fehler-/Ausfallmodi

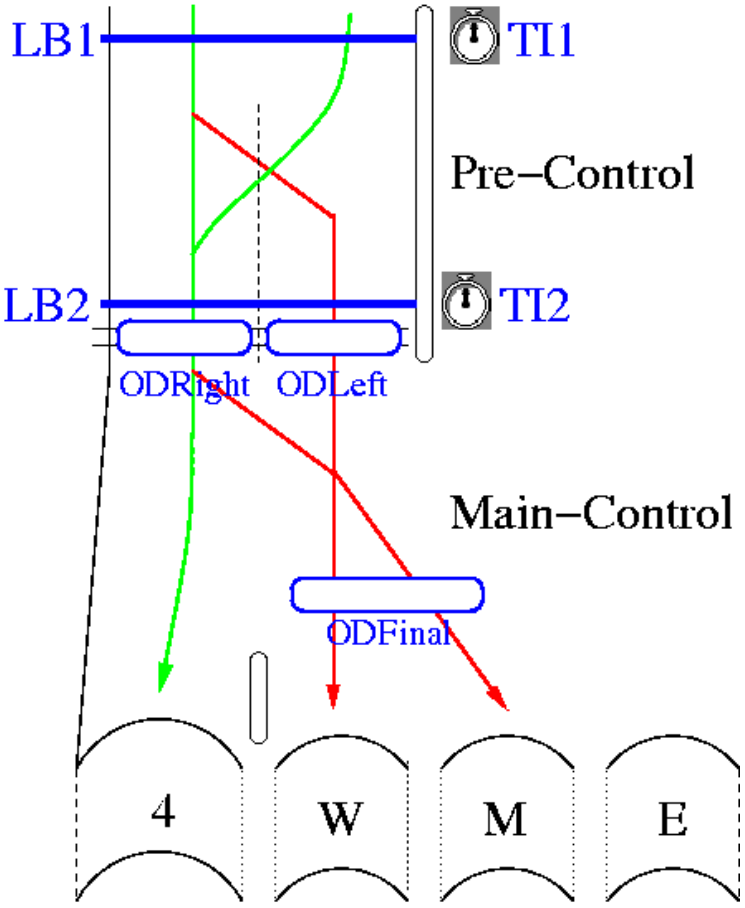
Modellbildung am Beispiel Elbtunnel

- Das Systemmodell beinhaltet:
  - Steuerung
  - Sensoren
  - Verkehrsfluss



Beispiel: Elbtunnel

- Formales Modell beinhaltet außerdem **Fehler- und Asufallmodi**:
  - Ausfälle der Überkopfdetektoren
    - Misdetektion (MD_{Left}, MD_{Right}, MD_{Final})
    - Fehldetektion (FD_{Left}, FD_{Right}, FD_{Final})
    - -> insgesamt: 6 Fehlermodi
  - Ausfälle der Überkopfdetektoren
    - Fehldetektion (FD_{LB1}, FD_{LB2})
    - -> insgesamt: 2 Fehlermodi
  - „Fehlermodi“ der Fahrzeugführer
    - LKW-Fahrer ignoriert StVO und fährt auf der linken Spur (HV_{Left}, HV_{Final})
    - -> insgesamt: 2 Fehlermodi
  - Überhohe Fahrzeuge geraten in einen Stau (OT_1, OT_2)
    - Insgesamt -> 2 Fehlermodi



➡ Formale Fehlermodellierung (nicht in diesem Vortrag)

[Ortmeier, Schellhorn 2007]

Frage 1: Funktionale Korrektheit

- Zentrale Frage: „Erfüllt das System seine Sicherheitseigenschaft?“

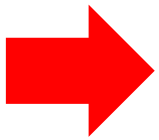
- Traditionelle Antworten:
  - Beachten von Design- und Konstruktionsrichtlinien
  - Orientierung an Erfahrungswerten
  - Expertenbefragungen/-reviews
  - Tests

Beobachtung

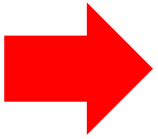
- Funktionale Korrektheit (Frage 1) ist ein Spezialfall von Fehlertoleranz (Frage 2)

- Funktionale Korrektheit: „Erfüllt das System seine Sicherheitseigenschaft (im ungestörten Betrieb)?“

- Fehlertoleranz: „Erfüllt das System seine Sicherheitseigenschaft, obwohl n-Ausfälle auftreten?“



Funktionale Korrektheit und Fehlertoleranz können gemeinsam betrachtet werden.



Deduktive-Ursache-Wirkungsanalyse (DCCA)

DCCA - Definitionen

Informell:

Sei Δ die Menge aller betrachteten Fehlermodi, eine Teilmenge Γ μ Δ heißt kritisch für einen hazard H, gdw. es einen Ablauf gibt

a) auf dem der hazard H auftritt und

b) auf dem keine Ausfälle aus Δ n Γ zuvor aufgetreten sind.

Formal:

$$\text{Critical}(\Gamma) \Leftrightarrow \text{SYS} \models E\,(\text{only}_{\Delta}(\Gamma)\ \text{until H})$$

$$\text{wobei only}_{\Delta}(\Gamma) \Leftrightarrow \bigwedge_{F \in \Delta \setminus \Gamma} \neg F$$

A Menge heißt minimal kritisch, gdw. keine echte Teilmenge von ihr kritisch ist.

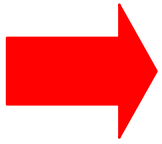
DCCA – Theorem

Definition:

DCCA ist die Bestimmung minimal kritischer Mengen. Eine vollständige DCCA ist die Bestimmung aller minimal kritischen Mengen.

Theorem:

Wenn ein Element jeder minimal kritischen Menge verhindert wird, so wird auch der hazard sicher verhindert.



- Keine Ursachen wurden vergessen
- Minimal kritische Menge beschreiben das intuitive Verständnis einer Ursache-Wirkungsrelation
  - Ausfälle führen – in zumindest einem denkbaren Szenario – zum Hazard
  - Hazard kann nicht auftreten ohne, dass zuvor die Ausfälle aufgetreten sind

Frage 1: Funktionale Korrektheit

- Ist die leere Menge von Fehlermodi kritisch?
- Beweisverpflichtung:

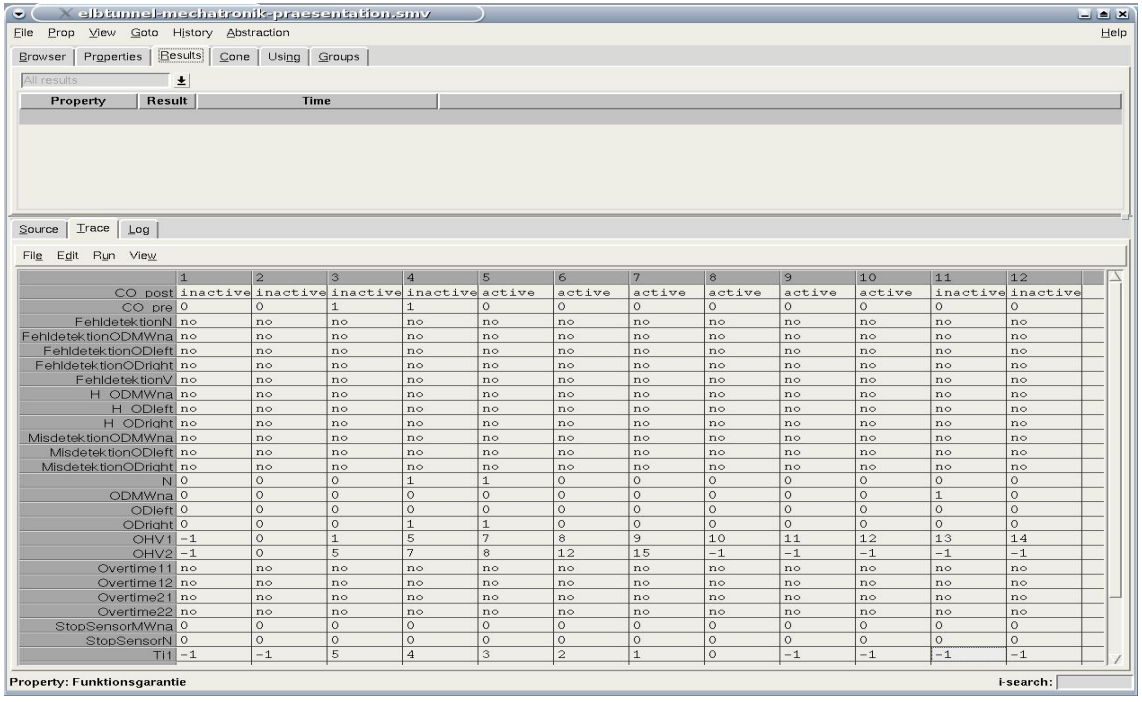
$$SYS \vdash E\ (only_{\Delta}(;)\ until\ H)$$

- In Umgangssprache:

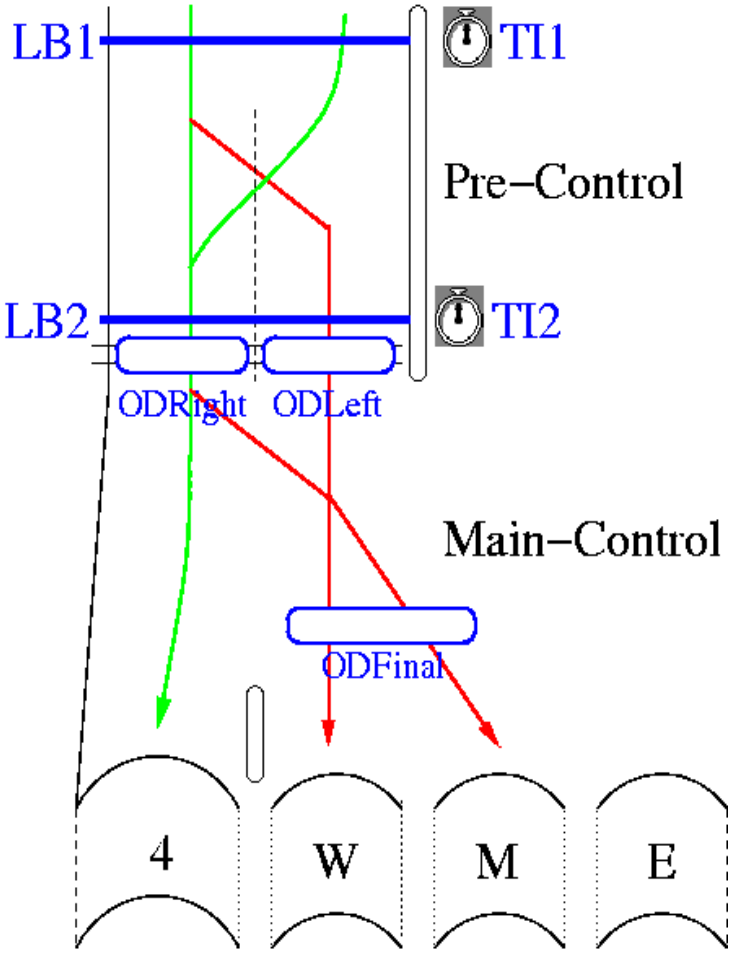
  „Gibt es einen Ablauf, auf dem der hazard H auftritt und kein Komponentenausfall zuvor aufgetreten ist?“ (= Funktionale Inkorrektheit)

Antwort auf Frage 1 am Beispiel

- Leider nein!
- Gegenbeispiel wird generiert

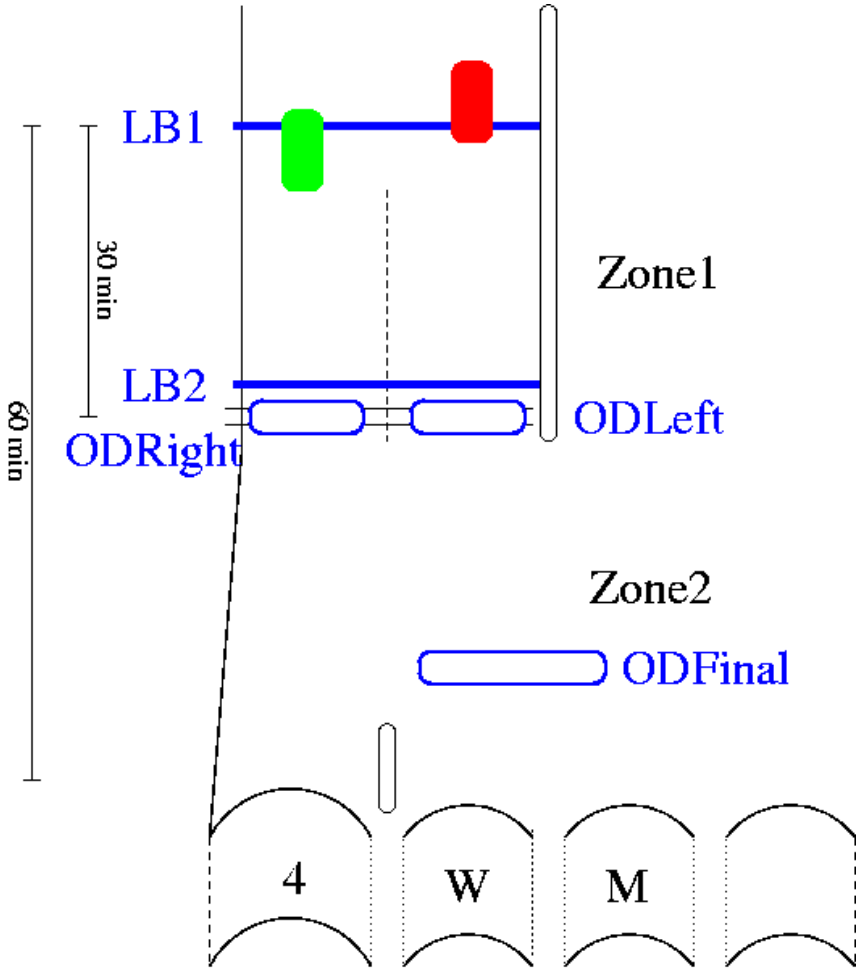


	1	2	3	4	5	6	7	8	9	10	11	12
CO_post	inactive	inactive	inactive	inactive	active	active	active	active	active	active	inactive	inactive
CO_pre	0	0	1	1	0	0	0	0	0	0	0	0
FehldetektionN	no	no	no	no	no	no	no	no	no	no	no	no
FehldetektionODMWna	no	no	no	no	no	no	no	no	no	no	no	no
FehldetektionODleft	no	no	no	no	no	no	no	no	no	no	no	no
FehldetektionODright	no	no	no	no	no	no	no	no	no	no	no	no
FehldetektionV	no	no	no	no	no	no	no	no	no	no	no	no
H_ODMWna	no	no	no	no	no	no	no	no	no	no	no	no
H_ODleft	no	no	no	no	no	no	no	no	no	no	no	no
H_ODright	no	no	no	no	no	no	no	no	no	no	no	no
MisdetektionODMWna	no	no	no	no	no	no	no	no	no	no	no	no
MisdetektionODleft	no	no	no	no	no	no	no	no	no	no	no	no
MisdetektionODright	no	no	no	no	no	no	no	no	no	no	no	no
N	0	0	0	1	1	0	0	0	0	0	0	0
ODMWna	0	0	0	0	0	0	0	0	0	0	1	0
ODleft	0	0	0	0	0	0	0	0	0	0	0	0
ODright	0	0	0	1	1	0	0	0	0	0	0	0
OHV1	-1	0	1	5	7	8	9	10	11	12	13	14
OHV2	-1	0	5	7	8	12	15	-1	-1	-1	-1	-1
Overtime11	no	no	no	no	no	no	no	no	no	no	no	no
Overtime12	no	no	no	no	no	no	no	no	no	no	no	no
Overtime21	no	no	no	no	no	no	no	no	no	no	no	no
Overtime22	no	no	no	no	no	no	no	no	no	no	no	no
StopSensorMWna	0	0	0	0	0	0	0	0	0	0	0	0
StopSensorN	0	0	0	0	0	0	0	0	0	0	0	0
TI1	-1	-1	5	4	3	2	1	0	-1	-1	-1	-1



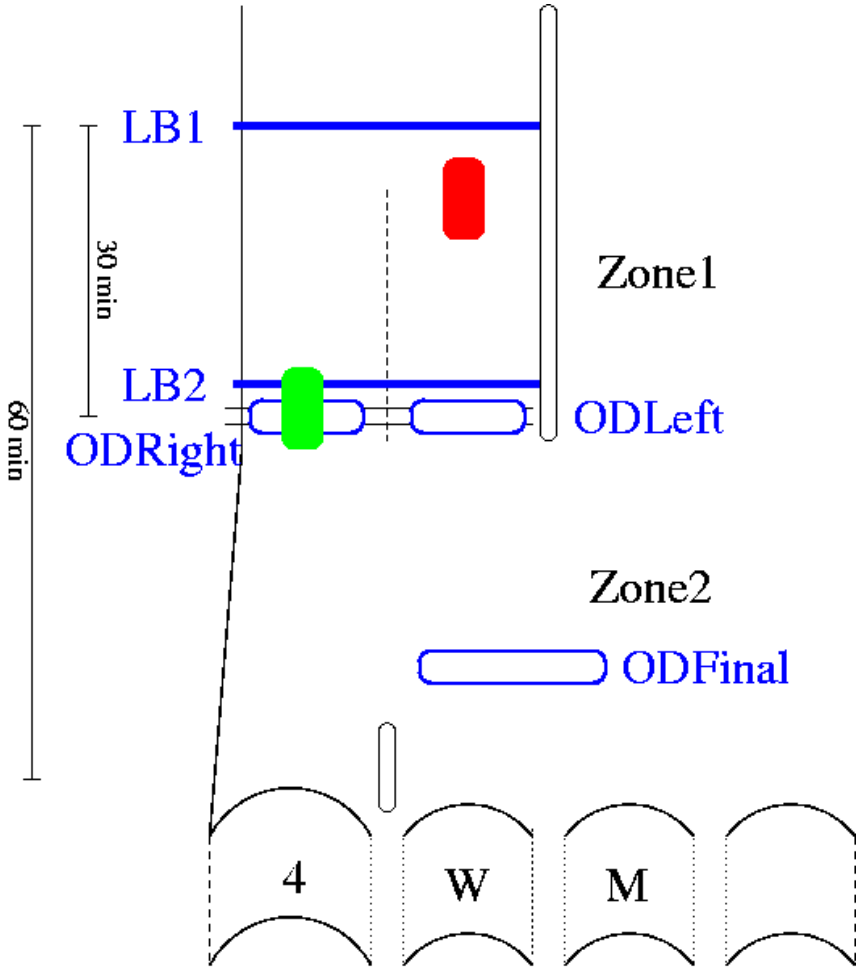
Wo liegt das Problem?

- T=0min
  - zwei OHVs fahren gleichzeitig durch LB1
  - OHV-Zähler wird um 1 erhöht (ZV=1)
  - ABER: das System hat ein OHV „vergessen“



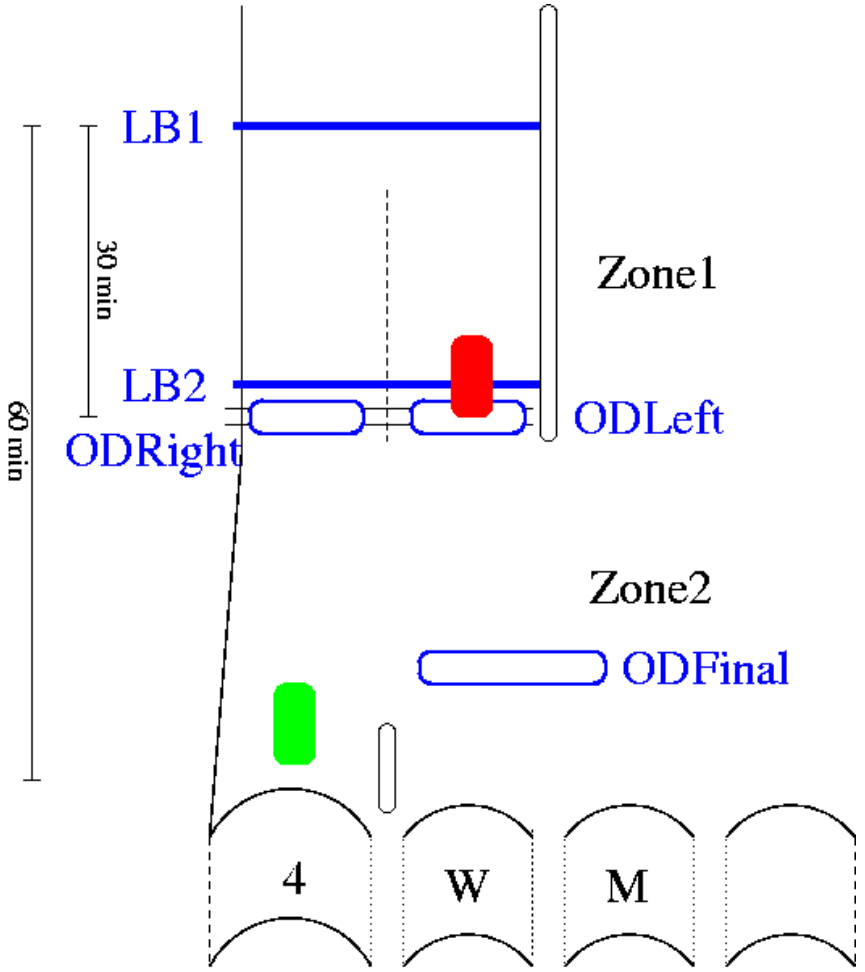
Wo liegt das Problem?

- T=5min
  - grünes OHV passiert LB2
  - LB2 wird deaktiviert (OHV-Zähler ZV=0)
  - beide OHVs bewegen sich mit stark unterschiedlicher Geschwindigkeit
  - ODFinal wird aktiviert



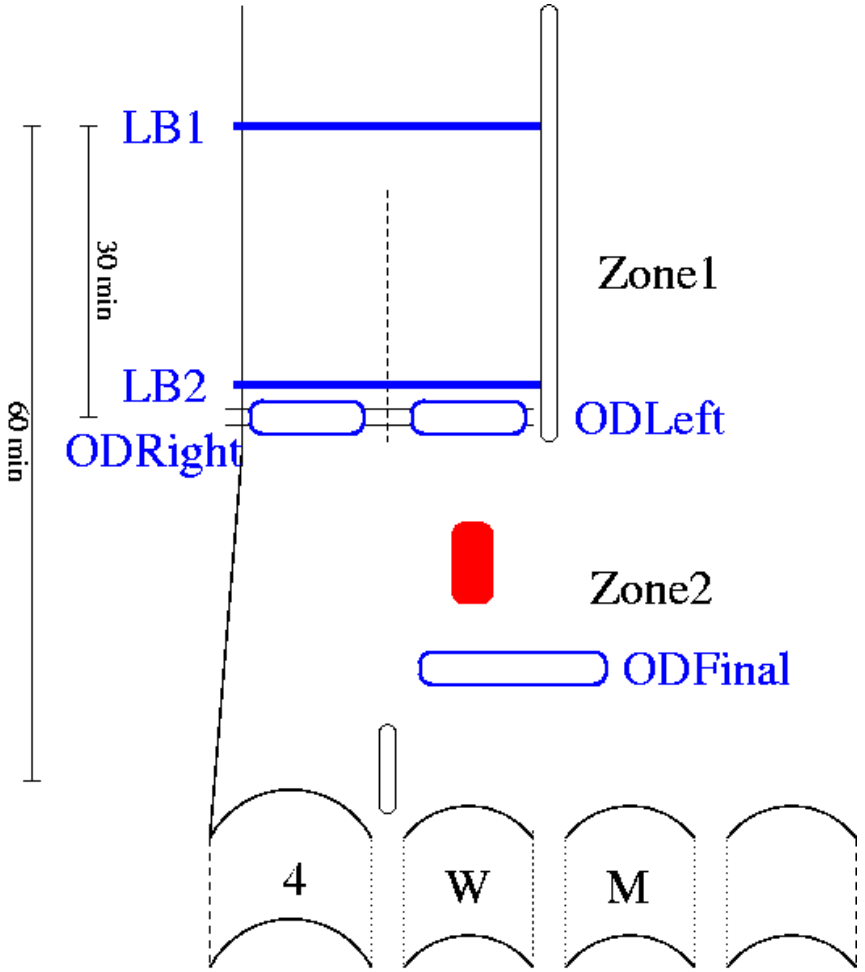
Wo liegt das Problem?

- T=20min
  - rotes OHV passiert LB2
  - aber LB2 ist deaktiviert
  - grünes OHV passiert den Tunnel korrekt



Wo liegt das Problem?

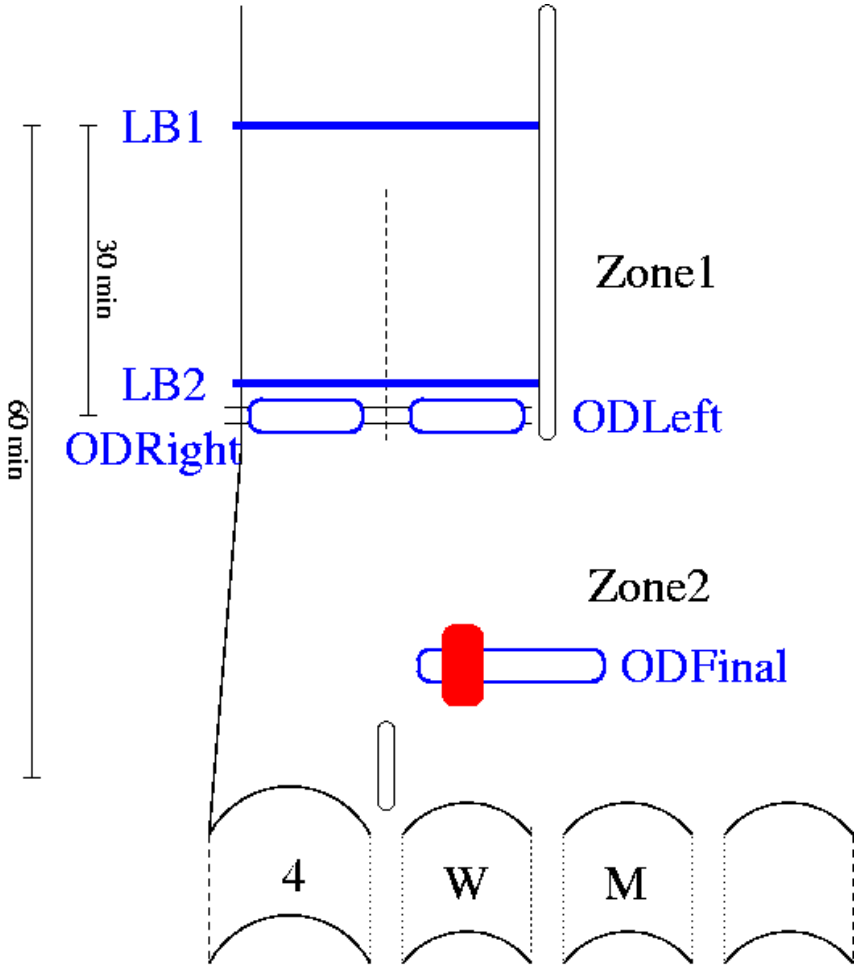
- T=35min
  - ODFinal wird deaktiviert, wegen Timeout von Timer2 (da LB2 von grünem OHV bei T=5min passiert wurde)



Wo liegt das Problem?

- T=40min
  - rotes OHV passiert ODFinal,
  - aber der Sensor ist deaktiviert

KOLLISION



Elbtunnel - Sicherheitslücke

- Sicherheitslücke war bislang unentdeckt!
- Problem wurde den Entscheidungsträgern vorgelegt

- Lösung (durch die Träger vorgegeben):
  - Durchfahren von LB1 auf der linken Spur wird per StVO verboten

- Konsequenz (für die Analyse):
  - Füge diese Situation zur Menge der Fehlermodi hinzu:

$$\Delta' = \Delta \cup \{OHV_{Sim}\}$$

Antwort auf Frage 1 am Beispiel

- Ist die leere Menge von Fehlermodi kritisch?
- Beweisverpflichtung:

$$SYS \vdash E\ (\ only_{\Delta'}(;)\ until\ H)$$

- In Umgangssprache:

  „Gibt es einen Ablauf, auf dem der hazard H auftritt und kein Komponentenausfall zuvor aufgetreten ist?“ (= Funktional Inkorrektheit)

- Ergebnis: NEIN!
- Folgerung:
  - Funktionsgarantie (=Antwort auf Frage 1)

Frage 2: Fehlertoleranz

- Welche n-elementigen Menge von Fehlermodi sind kritisch?
- Beweisverpflichtung:

$$SYS \vdash E\ (\ only_{\Delta'}(\{F_i\})\ until\ H)$$

- In Umgangssprache:

  „Welche einzelne Ausfälle können zum Systemversagen führen?“ (vgl. FMEA)

  „Welche Kombinationen von Ausfällen können zum Systemversagen führen?“ (vgl. FTA, ETA, ...)

- Ergebnis: critical sets

Vollständige DCCA

- Kombinationen von 2,3, ... Fehlermodi werden untersucht
- Exponentielle Zahl an Beweisverpflichtungen

- Im Beispiel:
  - 13 Fehlermodi werden untersucht
  - Worst case 2^{13}= 8192 Beweise notwendig!!
- Aber: ...

Aufwand DCCA

- Kritikalität ist monoton
  - Falls Γ kritisch ist, dann ist jede Obermenge davon kritisch.

$$\Gamma_1 \mu \Gamma_2) (\text{critical}(\Gamma_1)) \text{critical}(\Gamma_2))$$

- Im Beispiel:
  - Für den Elbtunnel sind nur 18 von 8192 Beweisen notwendig um alle minimal kritischen Mengen zu finden
  - Zusammen: weniger als 1 Minute Rechenzeit mit SMV

Antwort auf Frage 2 am Beispiel

- Minimal kritische Mengen für <u>Fehlalarme</u>
  - $\{MD_{Right}\}$
  - $\{FD_{Left}\}$
  - $\{HV_{Left}\}$
  - $\{HV_{Final}\}$
  - $\{FD_{Final}\}$
  - $\{FD_{LB2}\}$

- Minimal kritische Mengen für <u>Kollision</u>
  - $\{OHV_{Sim}\}$
  - $\{MD_{Final}\}$
  - $\{OT_1\}$
  - $\{OT_2\}$
  - $\{FD_{LB2}\}$

Frage 3: Ausfallwahrscheinlichkeiten

- Typisches Vorgehen:
  - Wahrscheinlichkeiten für Komponentenausfälle werden vorgegeben
  - Systemausfallwahrscheinlichkeit wird aus diesen Werten und den cut sets approximiert

$$P(hazard) = \sum_{all\ min.CS} P(CS)$$

$$P(CS) = \prod_{FM \in CS} P(FM)$$

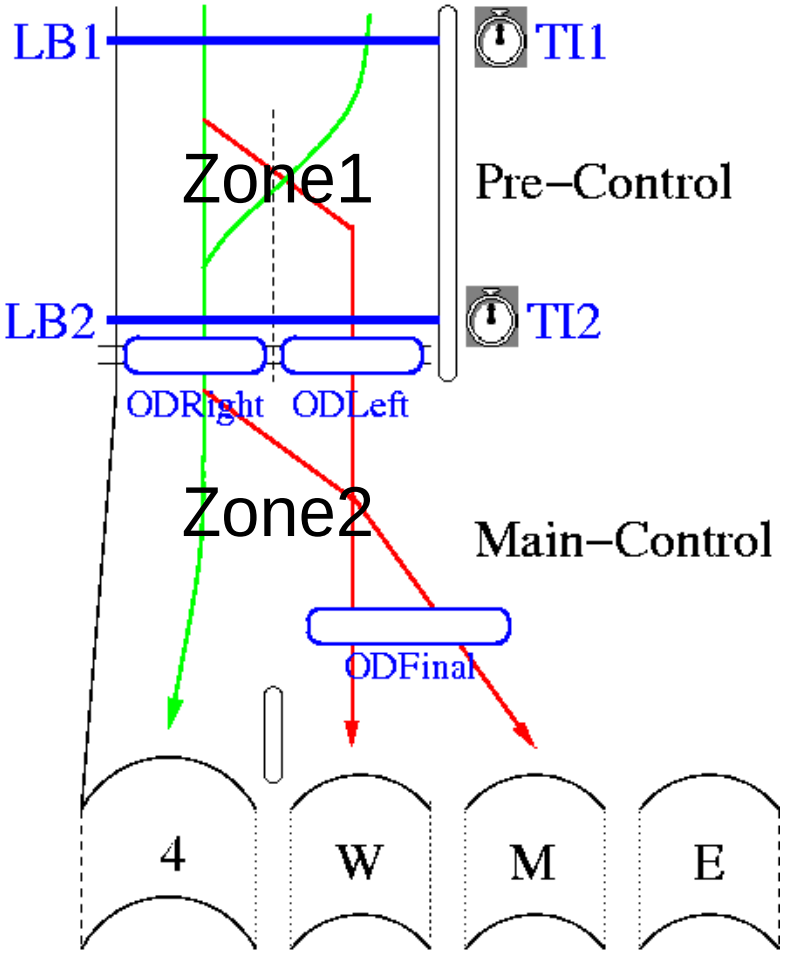
[Vesely, Leveson, ...]

  - Ergebnisse für das Beispiel:
    - P(Fehlalarm) ~ $3*10^{-4}$ /min
    - P(Kollision) ~ $3*10^{-8}$ /min

ABER: Wieso sind das feste Werte????

Beispiel Elbtunnel

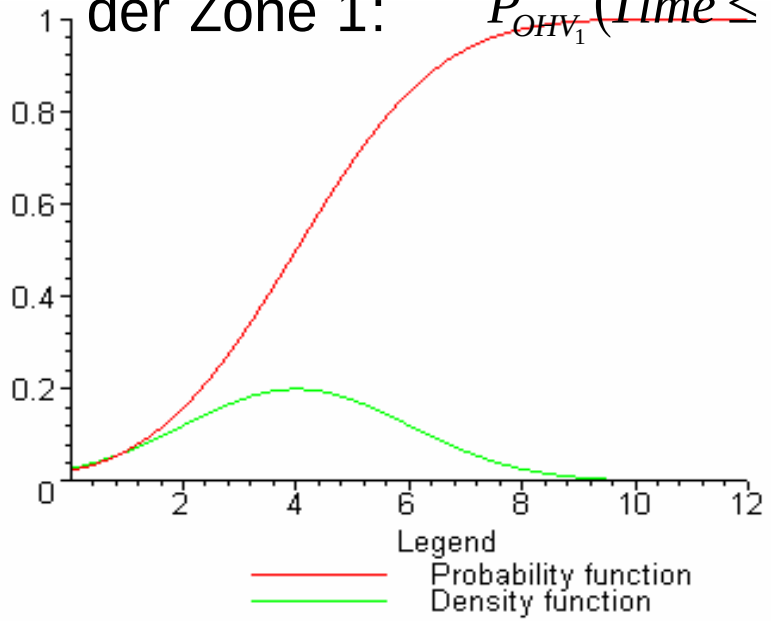
- Ein Single-point-of-failure:
  - „OHV bleibt in einem Stau in Zone 1 länger als die Laufzeit von T1“ ($\{OT_1\}$)
  - $P(\{OT_1\}) = ??$



Alternative

- Stat. Modellierung der Fahrzeiten der OHVs
  - Z.B. Normalverteilung (Erwartungswert 4 min., Standard-abweichung 2 min.)

Fahrzeit zum Durchqueren der Zone 1: $P_{OHV_1}(Time \leq t)$



- Parameterisierte Wahrscheinlichkeit

$$P(OT_1)(runtime1) = 1 - P_{OHV}(Time \leq runtime1)$$

Ergebnis

- Setze diese parametrisierten Wahrscheinlichkeiten zusammen

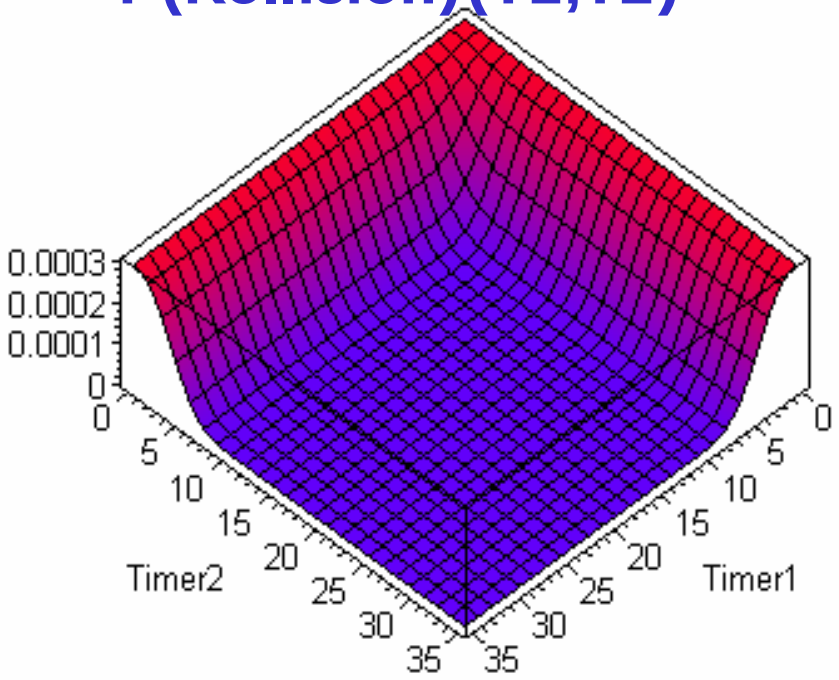
$$P(hazard)(x_1,...,x_n) = \sum_{all\ min.CS} P(CS)(x_1,...,x_n)$$

-> Parametrisierte Wahrscheinlichkeit für den Ausfall des Gesamtsystems

Beispiel: Elbtunnel

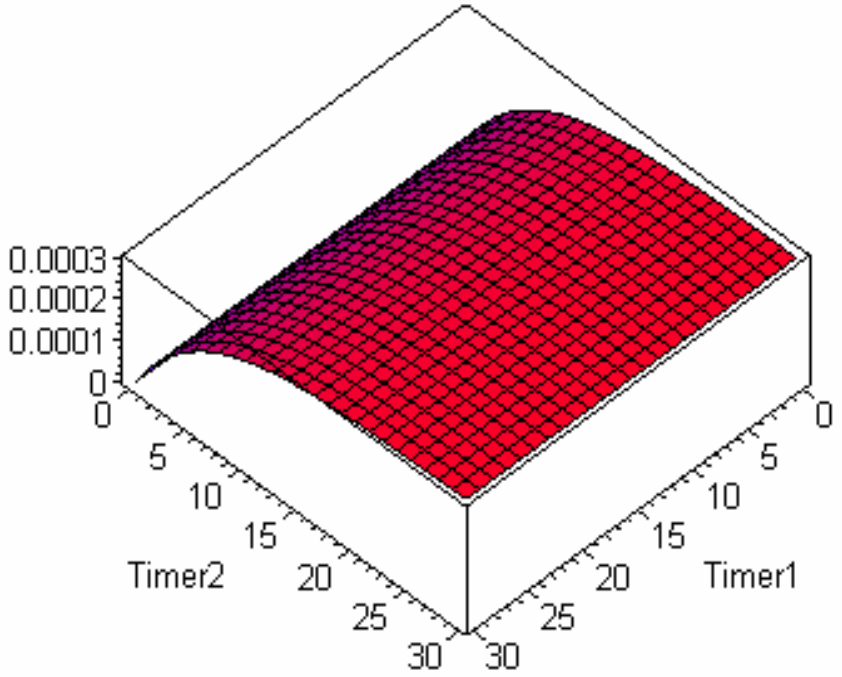
Kollision

P(Kollision)(T1,T2)



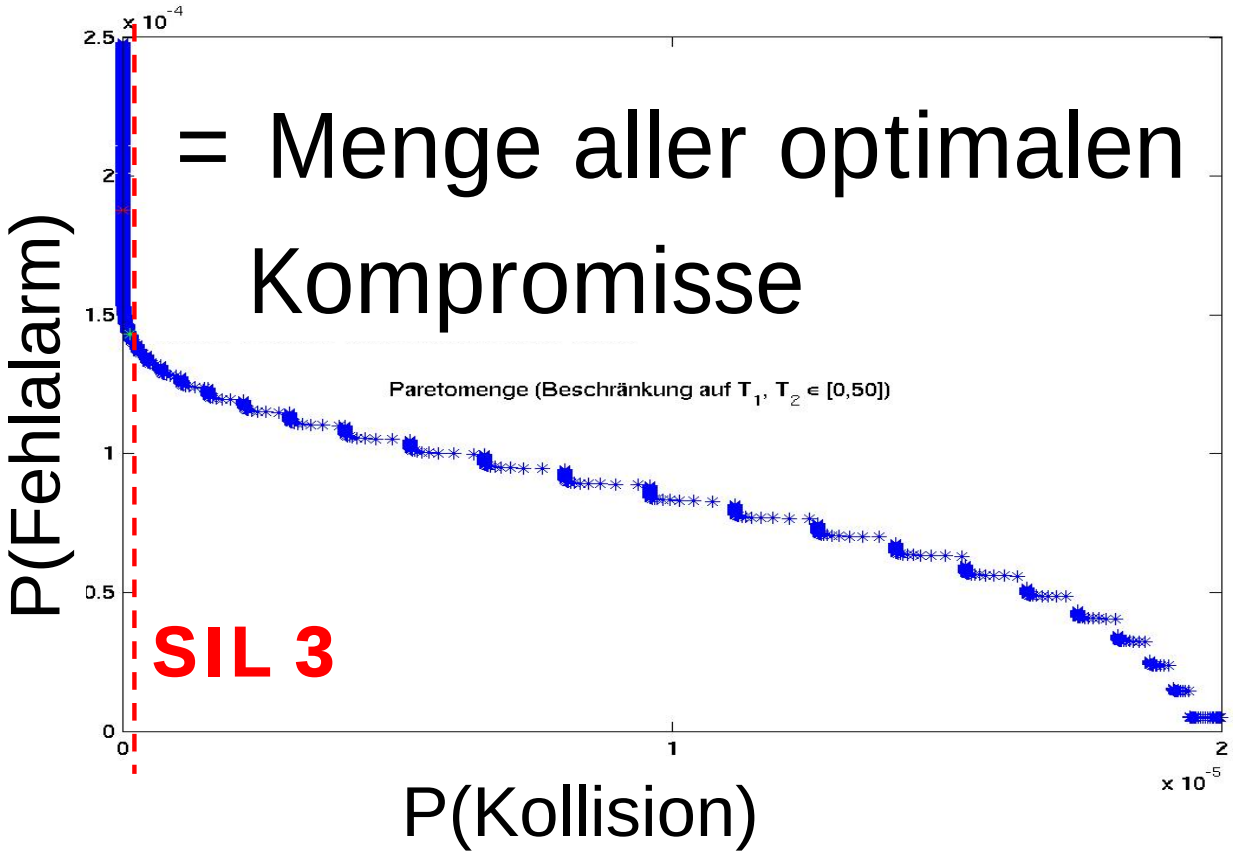
Fehlalarm

P(Fehlalarm)(T1,T2)



Antagonistische Ziele
-> optimaler Kompromiss??

Pareto-Optimierung



= Menge aller optimalen Kompromisse

Fehlalarme sinken um 10%, bei Einhaltung von SIL 3

Optimale Timerlaufzeiten:

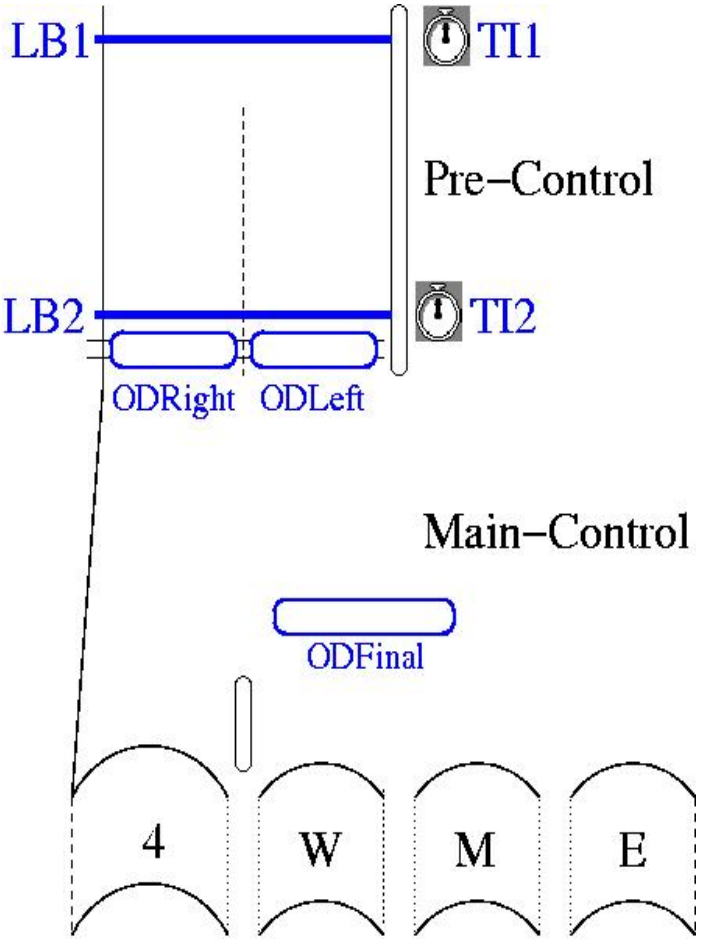
Timer1 ca. 19 Minuten
Timer2 ca. 15.6 Minuten
(statt jeweils 30 Min)

Wieso trotzdem nur 10% Reduktion bei Fehlalarmen?

Wieso nur 10 % Verbesserung?

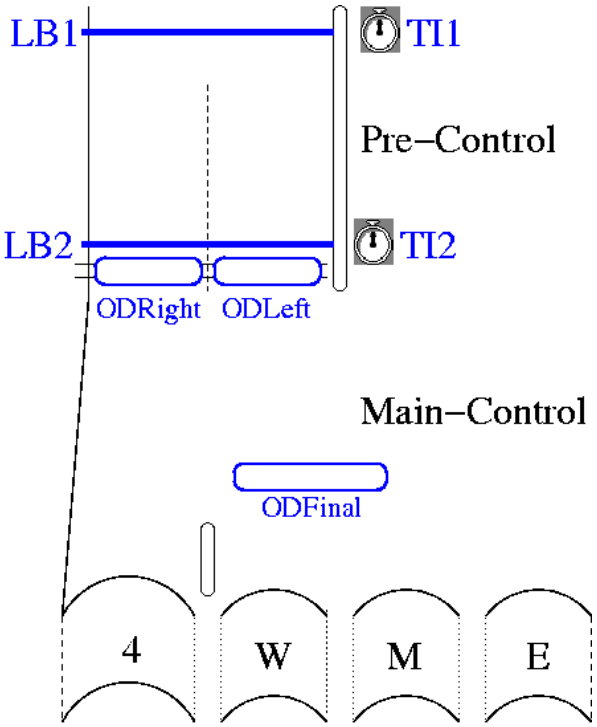
- Über 80% der Fehlalarme werden durch LKWs bei OD_{final} ausgelöst, wenn ein OHV den Tunnel korrekt passiert.
- Wenn ein OHV den Tunnel korrekt durchfährt, wird fast sicher ein Alarm ausgelöst.
- Die Zahl solcher Fehlalarme wächst linear mit Anzahl der OHVs.

Nicht akzeptabler Designfehler!!

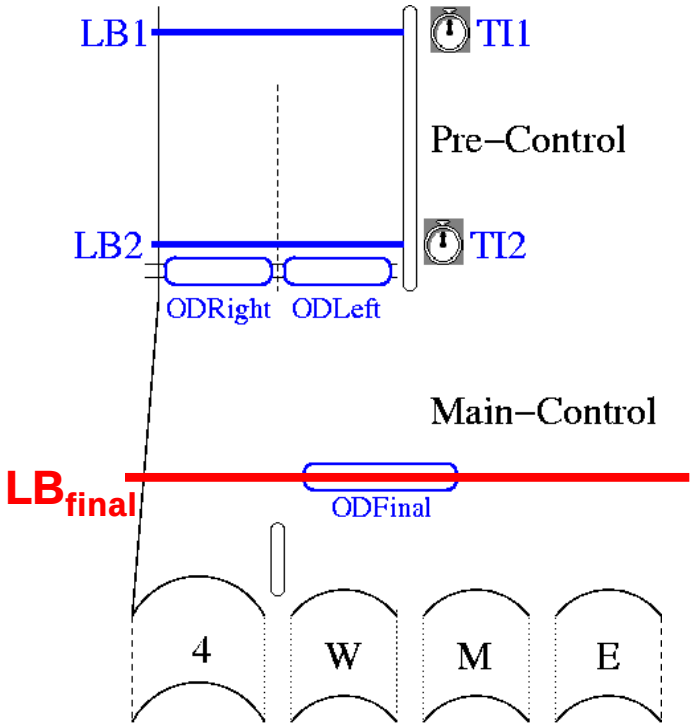


Variantenexploration

Geplantes System



Designvariante



Vorteile wachsen mit der Zahl der OHVs

Zusammenfassung

- Komplexität der sicherheitskritischen Systeme wächst zunehmend
- Steigende Kritikalität schlägt sich in steigenden Sicherheitsanforderungen nieder
- Modell-basierte Ansätze unterstützen den Ingenieur und können in verschiedenen Entwurfsstadien eingesetzt werden

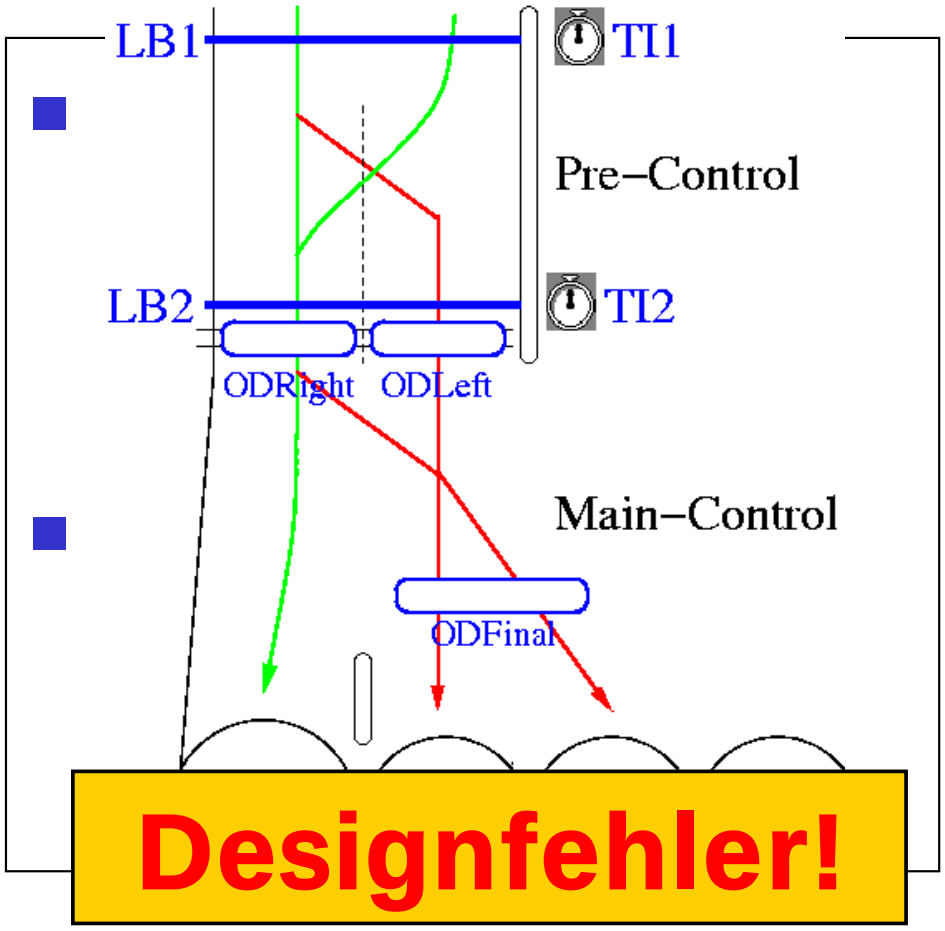
- Toolintegration möglich

Vielen Dank...

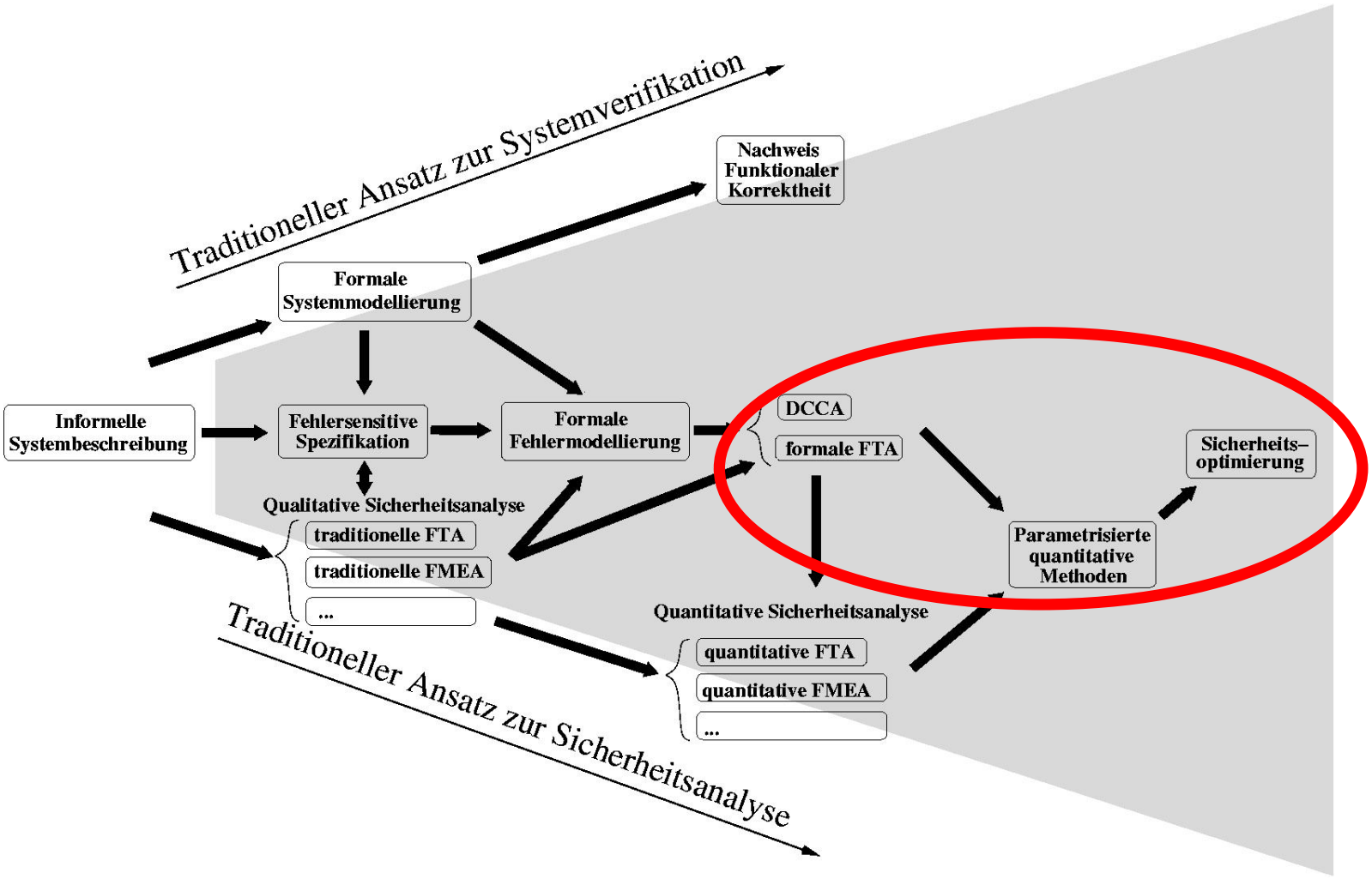
Dr. Frank Ortmeier
Lehrstuhl Softwaretechnik und Programmiersprachen
Universität Augsburg

Ergebnisse für das Beispiel

- Minimal kritische Mengen für Fehlalarme
  - $\{MD_{Right}\}$
  - $\{FD_{Left}\}$
  - $\{HV_{Left}\}$
  - $\{HV_{Final}\}$
  - $\{FD_{Final}\}$
  - $\{FD_{LB2}\}$



Der ForMoSA Ansatz



Formal Models Safety Analysis (**www.informatik.uni-augsburg.de/swt**)

[Ortmeier et al 04]

ISIS

Das Navigationssystem
für angemessene Qualität und hohe Effizienz

Inhalt

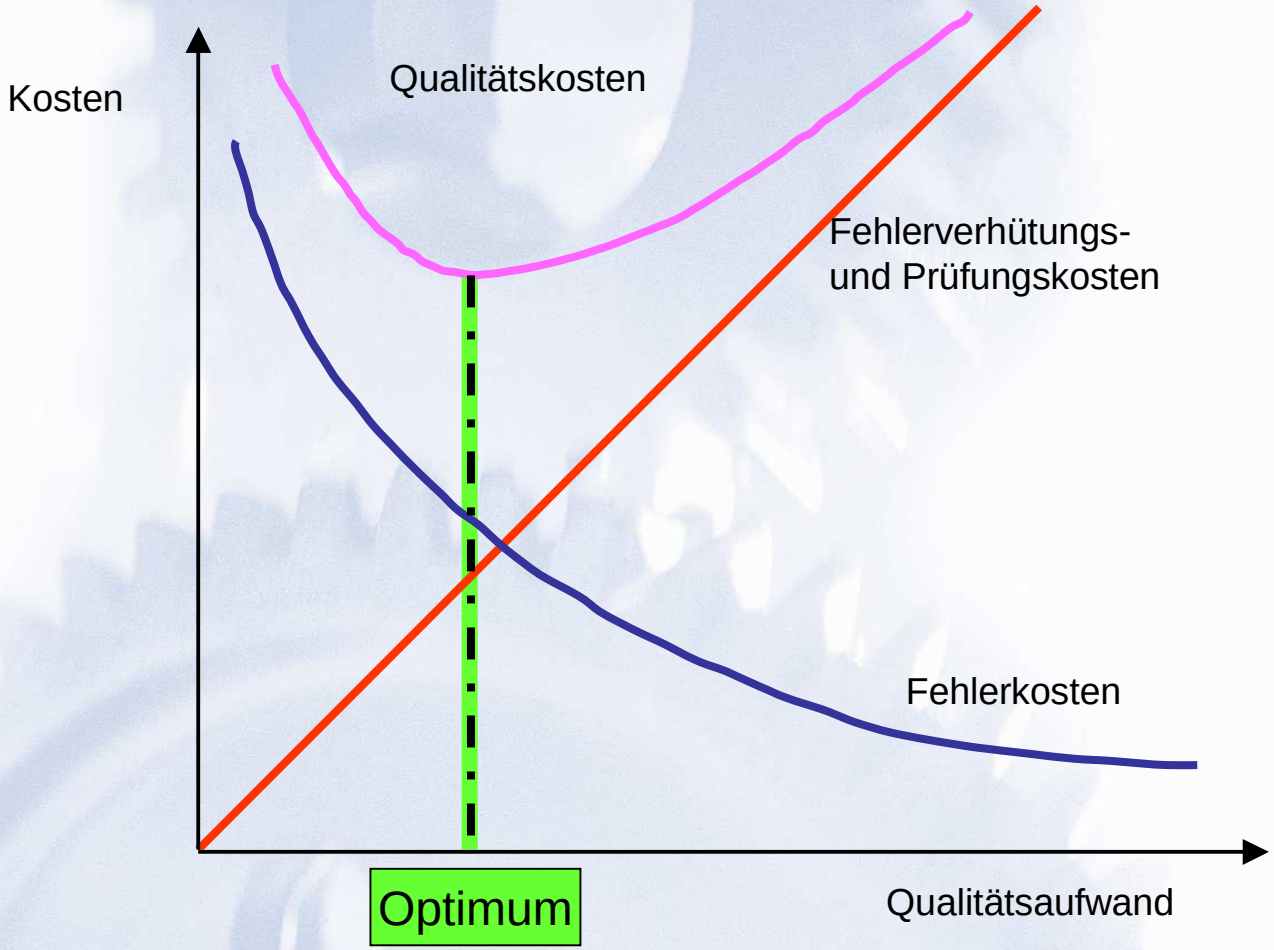
- Softwarequalität und Prozessqualität
- ISIS: das Ziel
- Messen der Prozessqualität
- Der Werkzeugzoo
- Die Wirkung
- Maßnahmen zur Prozessoptimierung
- Voraussetzungen für den sinnvollen Einsatz
- Spin-off: Messen der Softwarequalität
- Schlussbetrachtung

Softwarequalität: zwei Sichten

- Die Qualität eines Softwaresystems besteht aus der
  - → *Inneren Qualität:* Verständlichkeit, Wartungs-, Erweiterungs- und Änderungsfreundlichkeit (Sicht des Entwicklers)
  - → *Äußeren Qualität*: der Funktionalität, der Korrektheit, der Bedienbarkeit, der Performanz (Sicht des Benutzers)
- Üblicherweise liegt der Fokus auf der äußeren Qualität, die innere Qualität wird sträflich vernachlässigt
  - → Wesentliche Ursache für die Softwarekrise
  - → Hohe Kosten bei Änderungen/Erweiterungen

Die Qualität eines Softwaresystems resultiert aus der Qualität des Entwicklungsprozesses

Softwarequalität: Aufwandsoptimum



andrena OBJECTS

ISIS: Das Ziel

Kosten der Softwareproduktion minimieren durch Steuerung der Prozessqualität

Messung der Prozessqualität

- Mehrere Methoden auf dem Markt
  - → CMMI
  - → SPICE
  - → usw.
- Alle zu schwergewichtig, nicht praktikabel

- EN ISO 9001
  - → Leichtgewichtig
  - → Prozessorientiert (Messung der Kundenzufriedenheit als Steuergröße)
  - → Für alle möglichen Produktionsbereiche geeignet
  - → Zu dünn für Softwareentwicklung

ISIS: Messung der Prozessqualität

- Definition weniger Indikatormetriken, die
  - → in Summe das Ganze abdecken
  - → möglichst aussagekräftig und
  - → leicht zu erheben sind
- Bewertung der Messdaten durch Projektion auf eine Skala von 0 bis 100 (Qualitätsniveaus)
- Verdichtung der gewichteten Qualitätsniveaus zu einem Leitindikator (Prozessqualitätsindex)

Die Auswahl der Indikatoren, die Bewertungsalgorithmen und die Gewichtung der Indikatoren basieren auf 200 Jahren Entwicklungserfahrung

ISIS: Indikatormetriken

Kundenzufriedenheit	17 %
Anzahl der Programmierfehler	15 %
Schätzabweichung	11 %

Testabdeckung	13 %
Packages in Zyklen	11 %
Cyclomatic Complexity (Anzahl Methoden >5)	10 %
Average Component Dependency	9 %
Klassen größer 20 Methoden	6 %
Methoden größer 15 LOC	6 %
Compiler Warnungen	2 %

ISIS: wo liegt das Optimum?

- PQI = 65 bis 85 und
- etwa 1,5 Bugs pro Doppelzentner Entwickler und Monat

ISIS: Der Werkzeugzoo

1. ProjektLogbuch (*andrena Eigenentwicklung)*
2. Automatisierte statische Analyse
   - → *STAN oder*
   - → *Sotograph (Lizenz)*
3. BugCollector (*andrena Eigenentwicklung*)
4. Testabdeckung (*EclEmma, Freeware)*

ProjektLogbuch: zentrales ISIS-Werkzeug

- Integriert die Basiswerkzeuge und Metriken
- Verdichtet die Indikatoren zu Indizes
- Historisiert die Qualitätsdaten
- Visualisiert

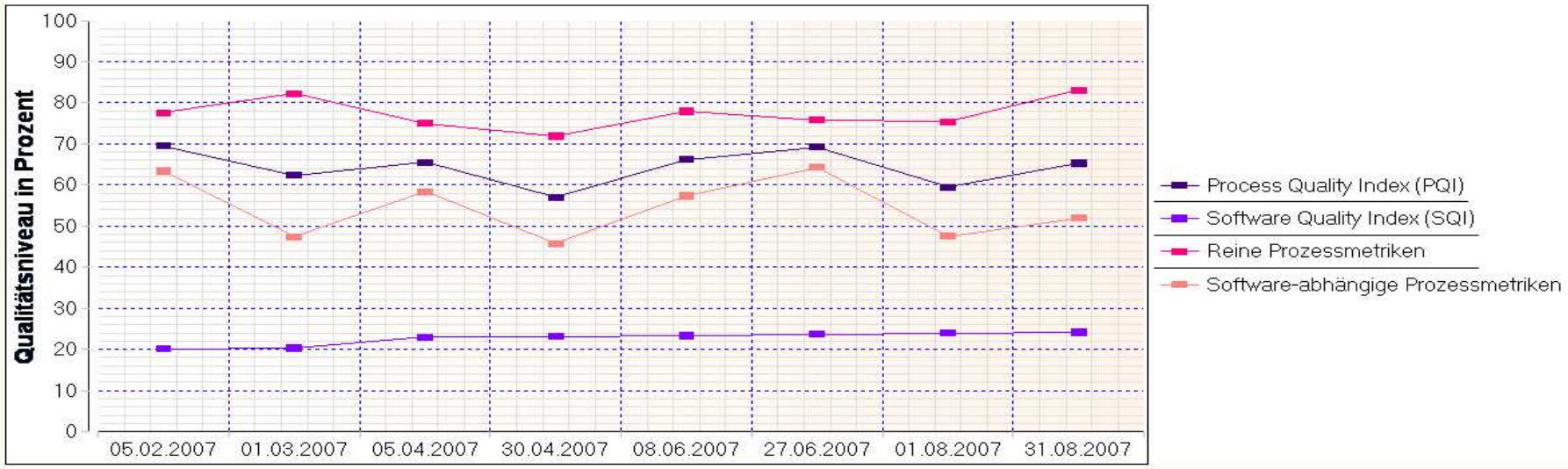
Das ProjektLogbuch ermöglicht transparente Produktion

Projektlogbuch: Beispiel aus der Produktion

Projekt Logbuch

Datei Messpunkt Ansicht Tools Hilfe

Datum	PQI	SQI	KuZ	%	Bugs	%	TA	%	PC	%	SAW	%	CC	%	ACD	%	KG	%	ML	%	CW	%	Beme
03.01.07	0	19	0	0	0	0	15.56	0	195	0	0	0	1105	0	35.13	0	401	0	5618	0	2733	0	neu
05.02.07	70	20	8	80	5	76	16.15	68	204	0	34	76	1103	100	26.58	100	406	100	5744	39	2608	0	neu
01.03.07	62	20	8	80	1	92	16.95	100	210	0	41	72	1109	51	26.42	23	408	78	5804	19	1338	47	neu
05.04.07	66	23	8	80	4	69	18.35	100	213	42	35	75	1114	47	26.31	24	409	72	5825	41	116	100	neu
30.04.07	57	23	7	70	5	60	18.59	58	213	18	12	91	1113	70	26.12	38	413	36	5845	34	111	100	neu
08.06.07	66	23	8	80	3	75	18.96	77	213	11	29	79	1110	100	26.46	0	417	82	5851	75	110	100	neu
27.06.07	69	24	7	70	2	78	19.73	100	212	45	24	82	1106	97	26.97	0	418	69	5859	49	93	100	neu
01.08.07	59	24	8	80	4	81	20.6	98	215	0	61	61	1109	59	27.54	0	421	54	5863	54	84	100	
31.08.07	65	24	8	80	1	91	20.89	67	209	83	32	77	1108	55	27.48	22	423	22	5863	39	85	31	3.3 ec



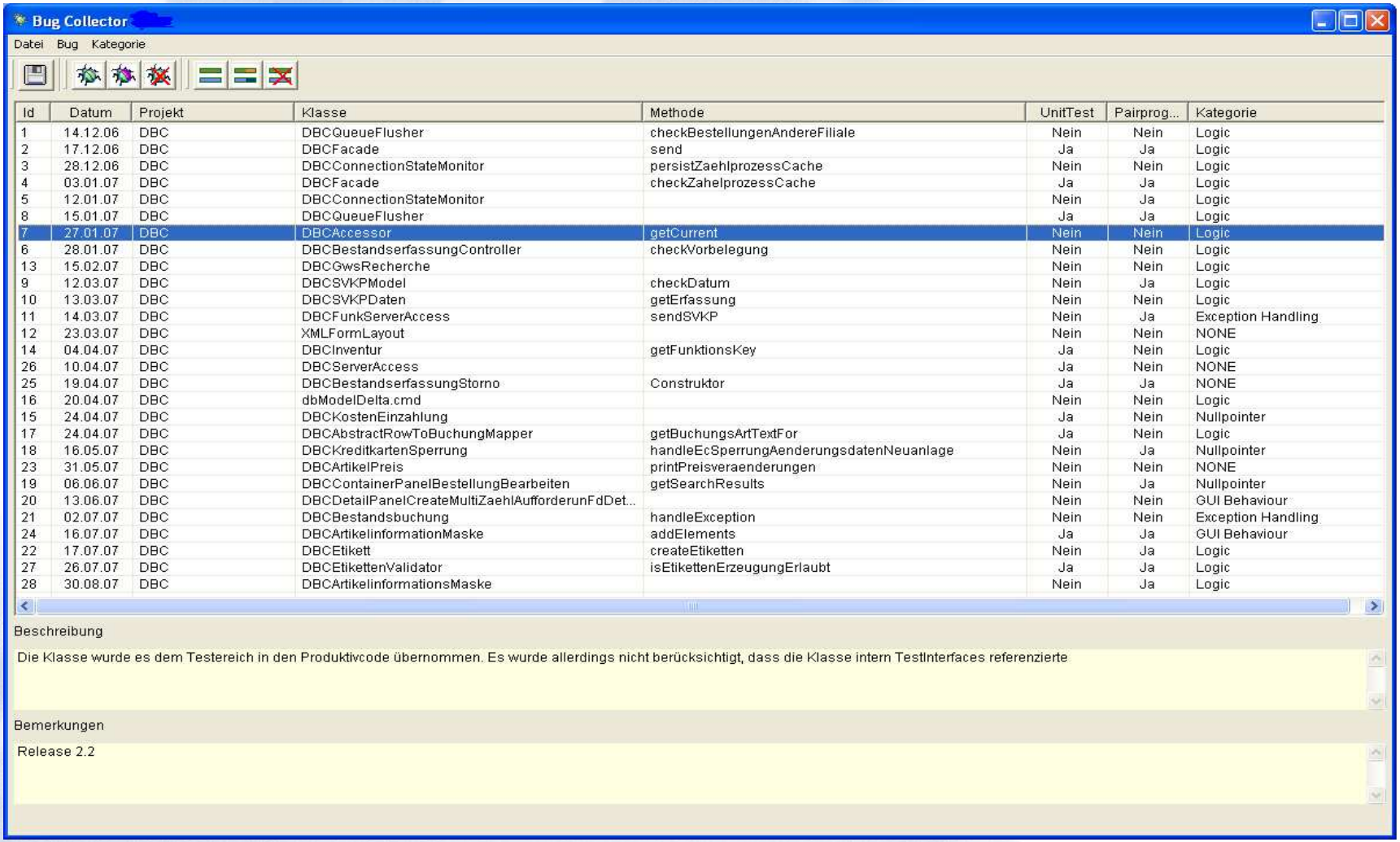
Statische Code-Analyse: Alternativen

- STAN
  - → Leichtgewichtig
  - → Vollständig in das ProjektLogbuch integriert (automatisierte Analyse)
  - → Auch als Standalone zu erwerben
- Sotograph
  - → Das mächtigste und stabilste Werkzeug auf dem Markt
  - → Teilweise in das ProjektLogbuch integriert (automatisierte Analyse)
  - → Für die manuelle Sichtung des Codes und zur Kontrolle von Änderungen über Delta-Analyse

BugCollector

- Zweck: Sammeln und Analysieren von Programmierfehlern (Bugs), um aus ihnen zu lernen. Vermeidung von Verdrängung
- Definition Bug: von der Spezifikation abweichendes Verhalten in Code, welcher vom Entwicklungsteam freigegeben wurde
- Wichtig: Keine individuelle Zuordnung von Fehlern
- Erkenntnisse aus unserem Projekt: häufig Fehler bei alleiniger Programmierung und/oder fehlenden automatisierten Tests

BugCollector: Beispiel aus der Produktion



Id	Datum	Projekt	Klasse	Methode	UnitTest	Pairprog...	Kategorie
1	14.12.06	DBC	DBCQueueFlusher	checkBestellungenAndereFiliale	Nein	Nein	Logic
2	17.12.06	DBC	DBCFacade	send	Ja	Ja	Logic
3	28.12.06	DBC	DBCConnectionStateMonitor	persistZaehlprozessCache	Nein	Nein	Logic
4	03.01.07	DBC	DBCFacade	checkZahelprozessCache	Ja	Ja	Logic
5	12.01.07	DBC	DBCConnectionStateMonitor		Nein	Ja	Logic
8	15.01.07	DBC	DBCQueueFlusher		Ja	Ja	Logic
7	27.01.07	DBC	DBCAccessor	getCurrent	Nein	Nein	Logic
6	28.01.07	DBC	DBCBestandserfassungController	checkVorbelegung	Nein	Nein	Logic
13	15.02.07	DBC	DBCGwsRecherche		Nein	Nein	Logic
9	12.03.07	DBC	DBCSVKPModel	checkDatum	Nein	Ja	Logic
10	13.03.07	DBC	DBCSVKPDaten	getErfassung	Nein	Nein	Logic
11	14.03.07	DBC	DBCFunkServerAccess	sendSVKP	Nein	Ja	Exception Handling
12	23.03.07	DBC	XMLFormLayout		Nein	Nein	NONE
14	04.04.07	DBC	DBCInventur	getFunktionsKey	Ja	Nein	Logic
26	10.04.07	DBC	DBCServerAccess		Ja	Nein	NONE
25	19.04.07	DBC	DBCBestandserfassungStorno	Construktor	Ja	Ja	NONE
16	20.04.07	DBC	dbModelDelta.cmd		Nein	Nein	Logic
15	24.04.07	DBC	DBCKostenEinzahlung		Ja	Nein	Nullpointer
17	24.04.07	DBC	DBCAbstractRowToBuchungMapper	getBuchungsArtTextFor	Ja	Nein	Logic
18	16.05.07	DBC	DBCKreditkartenSperrung	handleEcSperrungAenderungsdatenNeuanlage	Nein	Ja	Nullpointer
23	31.05.07	DBC	DBCArtikelPreis	printPreisveraenderungen	Nein	Nein	NONE
19	06.06.07	DBC	DBCContainerPanelBestellungBearbeiten	getSearchResults	Nein	Ja	Nullpointer
20	13.06.07	DBC	DBCDetailPanelCreateMultiZaehlAufforderunFdDet...		Nein	Nein	GUI Behaviour
21	02.07.07	DBC	DBCBestandsbuchung	handleException	Nein	Nein	Exception Handling
24	16.07.07	DBC	DBCArtikelinformationMaske	addElements	Ja	Ja	GUI Behaviour
22	17.07.07	DBC	DBCEtikett	createEtiketten	Nein	Ja	Logic
27	26.07.07	DBC	DBCEtikettenValidator	isEtikettenErzeugungErlaubt	Ja	Ja	Logic
28	30.08.07	DBC	DBCArtikelinformationsMaske		Nein	Ja	Logic

Beschreibung

Die Klasse wurde es dem Testereich in den Produktivcode übernommen. Es wurde allerdings nicht berücksichtigt, dass die Klasse intern TestInterfaces referenzierte

Bemerkungen

Release 2.2

Testabdeckung: Alternativen

EclEmma

- Freeware in Eclipse Umgebung
- misst die mittlere Testabdeckung und
- identifiziert die lokalen Defizite auf Klassen- und Methodenebene

ISIS: die Wirkung

- Schaffung von Qualitätsbewusstsein durch intensive Diskussion über Softwarequalität, Metriken, Prozesse
- Zeitnahe Reaktion auf Qualitätsprobleme
- Transparente Produktion
  - → Schafft Vertrauen bei Auftraggeber und Management
  - → Führt zu einer Verstetigung der Produktion
- In allen Projekten hat sich nach Einführung von ISIS die Codequalität deutlich verbessert
- ISIS macht Spaß, erzeugt hohes Selbstvertrauen, Motivation, Kreativität

ISIS: Maßnahmen zur Prozessoptimierung

- Strukturierung der Produktionskette
- Verbesserung der Anforderungsanalyse
- Verbesserung der Planung
- Verbesserung der Schätzmethode
- Konsequente pragmatische Anwendung von XP-Techniken
- Manuelle Entwicklertests als Ergänzung zu automatisierten Tests
- Konsequente Automatisierung

ISIS: Voraussetzungen

- Erfahrungen in Teilen des Teams Erfahrungen zu
  - → sauberer objektorientierter Programmierung
  - → XP-Techniken
  - → Automatisierung
- Mechanismus, der die Zirkulation und Verbreitung von Wissen im Team gewährleistet (z. B. Pairprogramming)
- Inkrementelle Entwicklung, Iterative Planung und Steuerung, Retrospektiven (bei uns mit Scrum)
- Spezialwissen für die Beherrschung der Werkzeuge
- Bereitschaft zu einer lernorientierten Fehlerkultur

ISIS: Messen der Softwarequalität

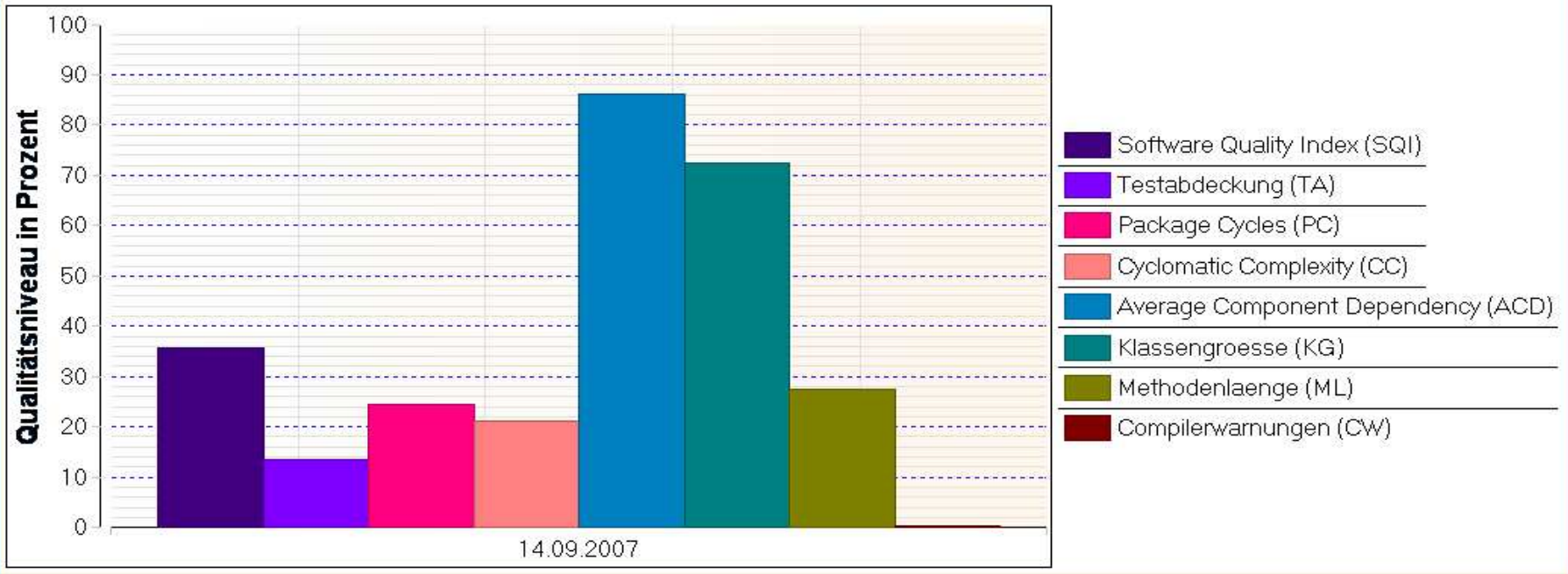
- Softwarequalität kann sehr effizient gemessen werden
- Aufwand liegt bei 1 Personentag
- Einsatzmöglichkeiten:
  - → Software Reviews
  - → Monatliches Qualitätsmonitoring

ISIS: Beispiel Softwarequalität

Softwareanalyse

Datei Messpunkt Ansicht Hilfe

Datum	SQI	TA	%	PC	%	CC	%	ACD	%	KG	%	ML	%	CW	%
14.09.07	36	13.6	14	49	24	241	21	3.32	86	36	72	797	27	340	0



Das andrena Qualitätsmanagementsystem

EN ISO 9001

+

ISIS

=

Agiles Qualitätsmanagementsystem

AdaCore
The GNAT Pro Company

Ada 2005 for Real-Time, Embedded and High-Integrity Systems

Jose F. Ruiz <ruiz@adacore.com>
Senior Software Engineer

Ada Deutschland Software Workshop 2008
Karlsruhe, 24th January, 2008

Outline of the presentation

- **Ada**
  - For embedded high-integrity real-time systems
- **Ada 2005**
  - The Ravenscar tasking profile
  - Flexible real-time scheduling algorithms
  - CPU clocks and timers
  - Timing events
  - Flexible object-oriented features

Ada for High-Integrity Applications

- **Ada promotes safety / reliability**
  - Readability
  - Compile-time checking (strong typing)
  - Encapsulation and data abstraction
  - Deterministic language semantics (ISO Standard)
    - Implementation must document effect where language semantics offers flexibility

- **Support modern software engineering techniques**
  - High abstraction level constructions integrated within the language
    - tasking, OOP, templates, modularity, data abstraction and encapsulation, …
  - General design philosophy promotes sound software engineering

- **Specific features**
  - Real-Time and High-Integrity Annexes
  - Language subsets

- **Guidelines documents**
  - *Guide for Ada in High-Integrity Systems* (an ISO Technical Report)
  - *Guide for Ada Ravenscar Profile in High-Integrity Systems*

Ada for Real-Time Systems

- **Concurrency**
  - Within the language
    - Avoid error-prone low-level constructions
  - Well-defined semantics for scheduling
  - Safe / efficient mutual exclusion
    - Avoidance of unbounded priority inversion
  - Ravenscar profile
    - Restricted set of tasking features amenable to schedulability analysis and certification
- **Asynchrony**
  - Asynchronous events / event handlers
    - Connection with interrupts
  - Asynchronous Transfer of Control
    - Timeout
    - Task termination
  - Preemptive task abortion
  - Asynchronous task control
- **Time**
  - Support for high-resolution monotonic clock and absolute and relative delays

Ada for Embedded Systems

- **Specific Annex for low-level support**
  - Access to hardware-specific features
- **Access to machine operations**
  - Assembly and intrinsic subprograms
- **Representation support**
  - Address, alignment, size, layout
- **Shared variable control**
  - Atomic, volatile,...
- **Storage management**
  - Specific storage pools
    - User-defined managers that may be placed in specific memory regions, and that may be suitable for real-time systems because they can be made predictable

What is N
ew in Ada 2005?

- **The Ravenscar profile**

- **Task elaboration and finalization**
  - Partition elaboration policy for high-integrity systems (atomic elaboration)
  - Task termination procedures

- **Restriction pragmas**
  - *No_Relative_Delay*, *Max_Entry_Queue_Length*,…

- **Time and clocks**
  - Timing events
  - Execution time clocks
  - Execution time budgets
    - for task groups also

- **Scheduling**
  - New dispatching policies
    - Non-preemptive, round robin, Earliest Deadline First (EDF)
  - Dynamic ceiling priorities
  - Priority Specific dispatching

- **Object-Oriented Programming**
  - Interfaces
  - Object notation

The Ravenscar Profile

- **A subset of the Ada tasking model**
- **Defined to meet safety-critical real-time requirements**
  - Determinism
  - Schedulability analysis
  - Memory-boundedness
  - Execution efficiency and small footprint
  - Suitability for certification
- **State-of-the-art concurrency constructs**
  - Adequate for most types of real-time software

The Ravenscar Profile (II)

- **Set of tasks / interrupts to be analyzed is fixed and has static properties**
  - Tasks, protected objects only at library level
    - No dynamic allocation of tasks or protected objects
  - Each task is infinite loop
    - single “triggering” action (delay or event)
- **Memory usage is deterministic**
  - Tasks descriptors and stacks are statically created at compile time
    - No implicit heap usage
- **Program execution is deterministic**
  - Simple protected objects
    - at most one entry, at most one caller queued
  - Task creation and activation is very simple and deterministic
    - Tasks created at initialization, then activated and executed according to their priority

The Ravenscar Tasking Model

- **A single processor**
- **A fixed number of tasks**
- **Single invocation event per task**
  - Time-triggered or event-triggered
- **Task interaction using shared data**
  - Mutual exclusive access
- **Remove constructions difficult to analyse**
  - No asynchronous control, no abort, ...

Most violations detected at compile time

The Ravenscar Tasking Model (II)

- **Scheduling policy**
  - Preemptive fixed-priorities
- **Locking policy**
  - Ceiling priority for bounding priority inversion
- **Remove non-deterministic constructions**
  - No relative delays, no task termination, no abort, ...

Supports sound real-time development techniques, such as Rate Monotonic Analysis and Response Time Analysis

Example: Cyclic tasks

```
task body Cyclic is
   Period : constant Time_Span := Seconds (1);
   Next_Activation : Time := Clock;
begin
   loop
      delay until Next_Activation;
      --  Do something
      Next_Activation := Next_Activation + Period;
   end loop;
end Cyclic;
```

```
task body Cyclic_With_Deadline is
   Period : constant Time_Span := Seconds (1);
   Next_Activation : Time := Clock;
begin
   loop
      delay until Next_Activation;
      Next_Activation := Next_Activation + Period;
      select
         delay until Next_Activation;
         --  Notify missed deadline
      then abort
         --  Do something
      end select;
   end loop;
end Cyclic_With_Deadline;
```

Real-Time Scheduling

- **Ada 95 provides**
  - Complete and well defined set of language primitives for Fixed Priority Scheduling
- **Ada 2005 allows new schemes**
  - Non-preemptive
  - Round Robin
  - Earliest Deadline First (EDF)
  - Mixed policies within a partition

Timing Events

- **A means of defining code that is executed at a future point in time**
  - Efficient stand-alone timer
- **Does not need a task**
  - Executed directly in the context of the interrupt handler
  - Reduce the number of
    - tasks in a program
    - Context switches
- **Similar in notion to interrupt handling**
  - Time itself generates the interrupt
- **Useful for**
  - Short time-triggered procedures
  - Imprecise computation

Example: Task deadlines with timing events

```
protected Watchdog is
   pragma Interrupt_Priority (Interrupt_Priority'Last);
   procedure Timeout (Event : in out Timing_Event);
   entry Is_OK;
private
   Panic : Boolean := False;
end Watchdog;

protected body Watchdog is
   procedure Timeout (Event : in out Timing_Event) is
   begin
      -- Alarm !!!
      Panic := True;
   end Timeout;
   entry Is_OK when Panic is
   begin
      -- Panic mode activated
      Panic := False;
   end Is_OK;
end Watchdog;
```

```
task body Cyclic_With_Deadline is
   Period : constant Time_Span := Seconds (1);
   Next_Activation : Time := Clock;
   Deadline_Event : Timing_Event;
   Alarm_Cancelled : Boolean;
begin
   loop
      delay until Next_Activation;
      Next_Activation := Next_Activation + Period;
      Set_Handler
        (Event    => Deadline_Event,
         At_Time => Next_Activation,
         Handler  => Watchdog.Timeout'Access);
      select
        Watchdog.Is_OK;
        -- Notify missed deadline
      then abort
        -- Do something
        -- Notify end of computation
        Cancel_Handler
          (Deadline_Event, Alarm_Cancelled);
      end select;
   end loop;
end Cyclic_With_Deadline;
```

Execution Time Support

- **Monitor and control task execution time**
  - Every task has an execution time clock
  - Fire an event when a task execution time reaches a specified value
  - Useful in high-integrity (fault tolerant) applications for detecting
    - Wrong WCET estimations
    - Software errors
- **Allocate and support bugdets for groups of tasks**
  - Useful for some scheduling policies, such as those for aperiodic servers

Example: Iterative computation

```
task body Iterative_Task is
   Stop_Time : CPU_Time :=
      Ada.Execution_Time.Clock + Milliseconds (10);
begin
   while
      Ada.Execution_Time.Clock < Stop_Time
   loop
      --  Do something
   end loop;
end Iterative_Task;
```

```
task body Iterative_Task_2 is
   ID : aliased Task_ID := Current_Task;
   Budget_Manager : Timer (ID'Access);
begin
   Set_Handler
      (Budget_Manager, Milliseconds (10),
       Overrun.Timeout'Access);
   select
      Overrun.Stop_Task;
   then abort
      loop
         --  Do something
      end loop;
   end select;
end Iterative_Task_2;
```

```
protected Overrun is
   entry Stop_Task;
   procedure Timeout (TM : in out Timer);
private
   Budget_Overrun : Boolean := False;
end Overrun;

protected body Overrun is
   entry Stop_Task when Budget_Overrun is
   begin
      --  Budget overrun
      Budget_Overrun := False;
   end Stop_Task;
   procedure Timeout (TM : in out Timer) is
   begin
      --  Stop computation
      Budget_Overrun := True;
   end Timeout;
end Overrun;
```

Example: Budgets

```
protected Overrun is
   entry Stop_Task;
   procedure Handler (TM : in out Timer);
private
   Budget_Overrun : Boolean := False;
end Overrun;

protected body Overrun is
   entry Stop_Task when Budget_Overrun is
   begin
      --  Budget overrun
      Budget_Overrun := False;
   end Stop_Task;
   procedure Handler (TM : in out Timer) is
   begin
      --  We have a problem
      Budget_Overrun := True;
   end Handler;
end Overrun;
```

```
task body Cyclic_With_Budget is
   Period : constant Time_Span := Seconds (1);
   Next_Activation : Time := Clock;
   ID : aliased Task_ID := Current_Task;
   Budget_Manager : Timer (ID'Access);
   Alarm_Cancelled : Boolean;
begin
   loop
      delay until Next_Activation;
      Next_Activation := Next_Activation + Period;
      Set_Handler
        (TM       => Budget_Manager,
         At_Time => Next_Activation,
         Handler  => Overrun.Handler'Access);
      select
        Overrun.Stop_Task;
        --  Notify missed deadline
      then abort
        --  Do something
        --  Notify end of computation
        Cancel_Handler
          (Budget_Manager, Alarm_Cancelled);
      end select;
   end loop;
end Cyclic_With_Budget;
```

Safe Object Oriented Programming

- **Type extension and inheritance**
  - Powerful
    - Cover most object-oriented design methods
      - Code reuse, programming by extension, etc.
  - Fine for safety-critical systems
- **Dynamic dispatching**
  - Actual flow of control not known statically
  - Worrisome for safety-critical system
- **Controlling dynamic dispatching**
  - Avoid class-wide types
    - In Ada, methods are statically bound by default
  - Enforced by a language-defined restriction (*No_Dispatch*)
  - Each operation declare explicitly whether it is intended to inherit

Abstract interfaces

- **Limited form of multiple inheritance**
  - Java-like

    Multiple inheritance of specifications, and single inheritance of implementation

- **Extends the Java model**
  - Protected, task, and synchronized interfaces
    - abstraction that can be implemented either with an active task or with a passive monitor
    - Seamless integration between OO and multi-tasking features
- **Much of the power of multiple inheritance**
  - Without most of the implementation and semantic difficulties

Example: Interface

```
type Person is interface;
function Name (This : Person) return Name_Type is abstract;
function Gender (This : Person) return Gender_Type is abstract;

type Worker is interface;
function Name (This : Worker) return Name_Type is abstract;
function Salary (This : Worker) return Natural is abstract;

type Employee is new Person and Worker with
  record
    Name : Name_Type;
    Sex  : Gender_Type;
    Wage : Natural;
  end record;

function Name (This : Employee) return Name_Type;
function Gender (This : Employee) return Gender_Type;
function Salary (This : Employee) return Natural;
```

Example: Synchronized interface

```
type Processing_Entity is task interface;
procedure Replicate (This : Processing_Entity) is abstract;

type Buffer is synchronized interface;
procedure Put (This : in out Buffer; Item : Element) is abstract;
procedure Get (This : in out Buffer; Item : out Element) is abstract;

task type Server_Buffer is new Processing_Entity and Buffer with
   entry Replicate;
   entry Put (Item : Element);
   entry Get (Item : out Element);
end Server_Buffer;
```

Conclusions

- **Increasing need for safe programming**
  - Ada has an impressive track record in avionics, train control, other safety-critical domains
  - Ada is being considered in new domains

- **Ada 2005 addresses the needs of the real-time and high-integrity communities**
  - Expressive, even in safety-critical subsets
    - Safe tasking
    - Safe OOP
  - Flexible
    - New scheduling policies, new capabilities and features
  - High-level abstractions, but ...
    - Deterministic
    - Time analyzable

Verification of Safety Critical Systems

Software-Workshop Technologiepark Karlsruhe 24.01.2008

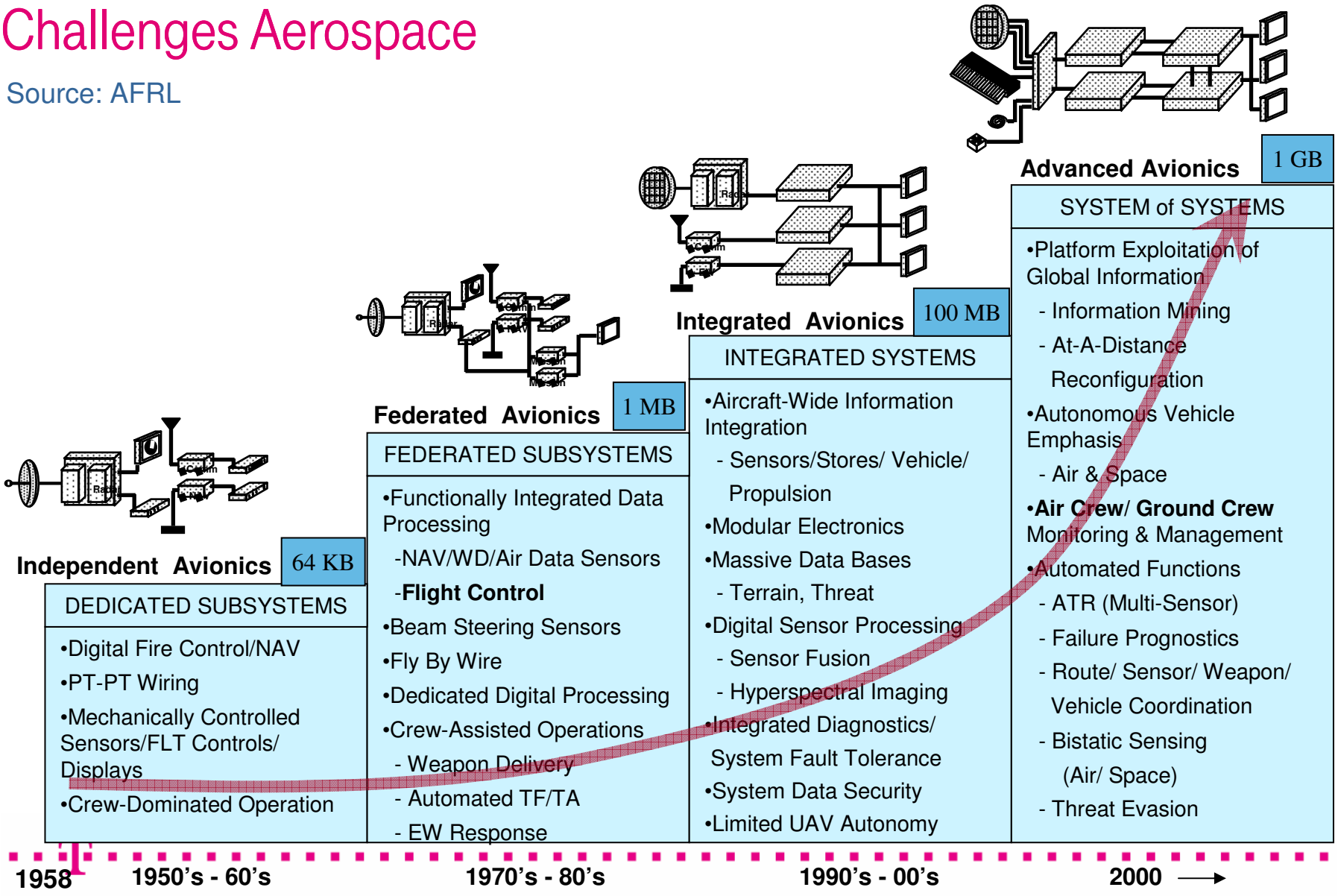
Dr. Christoph Diesch

Verification of Safety Critical Systems
Structure

- Challenges in Aerospace and Automotive
- Fields of Activities
- An Aerospace Example
- V&V Strategy – Theory
  - Requirements
  - Elements of the Strategy
  - Optimization
- V&V Strategy – Experience
  - Effort – Bad Case – Good Case
- Example „Early Verification“
- Example „End-to-End Test“
- 2 Automation Concepts

Verification of Safety Critical Systems
Challenges Aerospace

Source: AFRL



Verification of Safety Critical Systems
Challenges Automotive

Elektronische Einspritzung
Check Control
Geschwindigkeitsregler
Zentralverriegelung
...

1970

Elektronische Getriebesteuerung
Elektronische Klimaregelung
ASC Anti Slip Control
ABS Anti Blocking System
Telefon
Sitzheizungssteuerung
Autom. Spiegelabblendung

1980

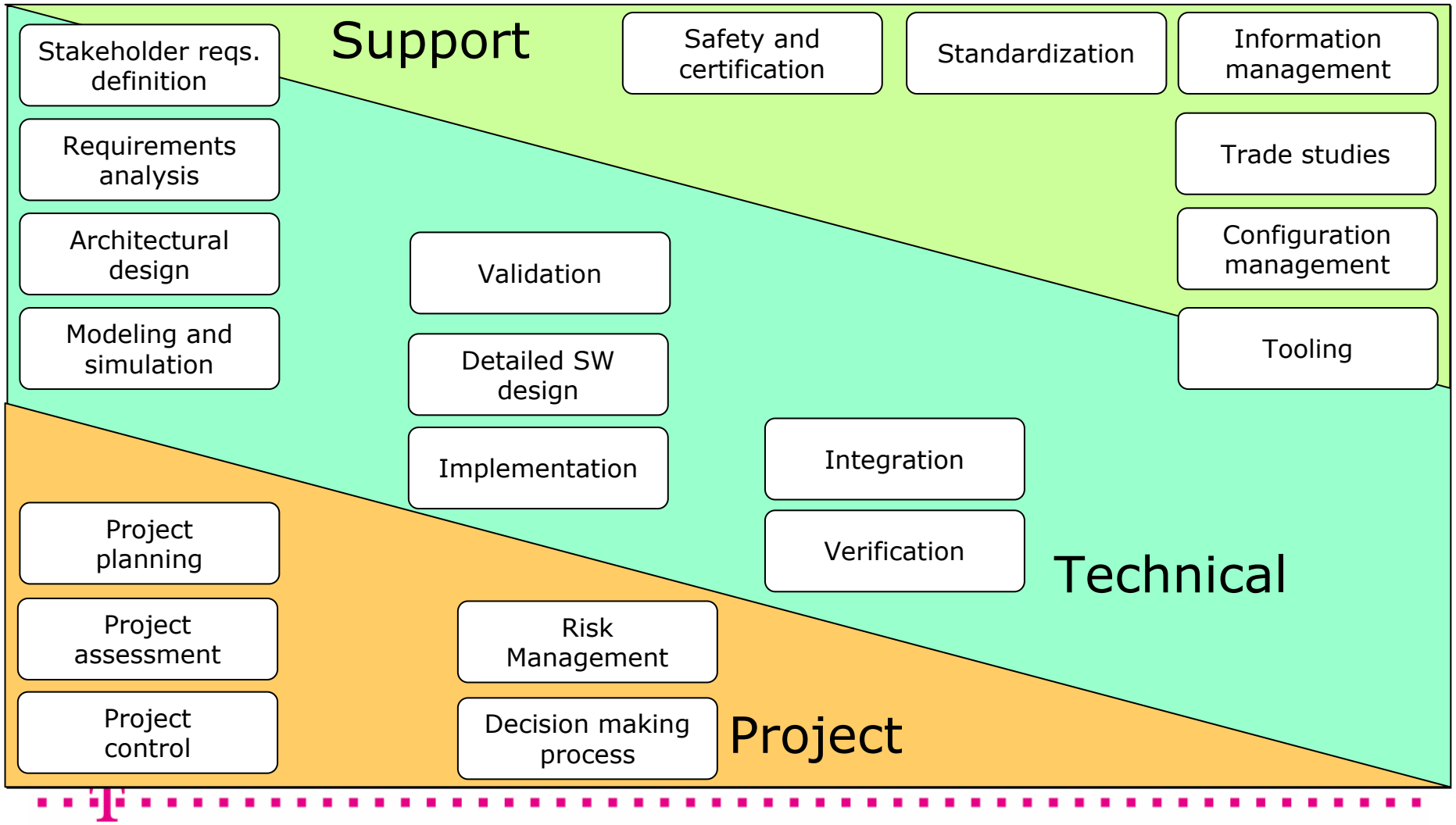
Navigationssystem
CD-Wechsler
ACC Active Cruise Control
Airbags
DSC Dynamic Stability Control
Adaptive Getriebesteuerung
Rollstabilisierung
Xenon Licht
BMW Assist
RDS/TMC
Spracheingabe
Notruf

1990

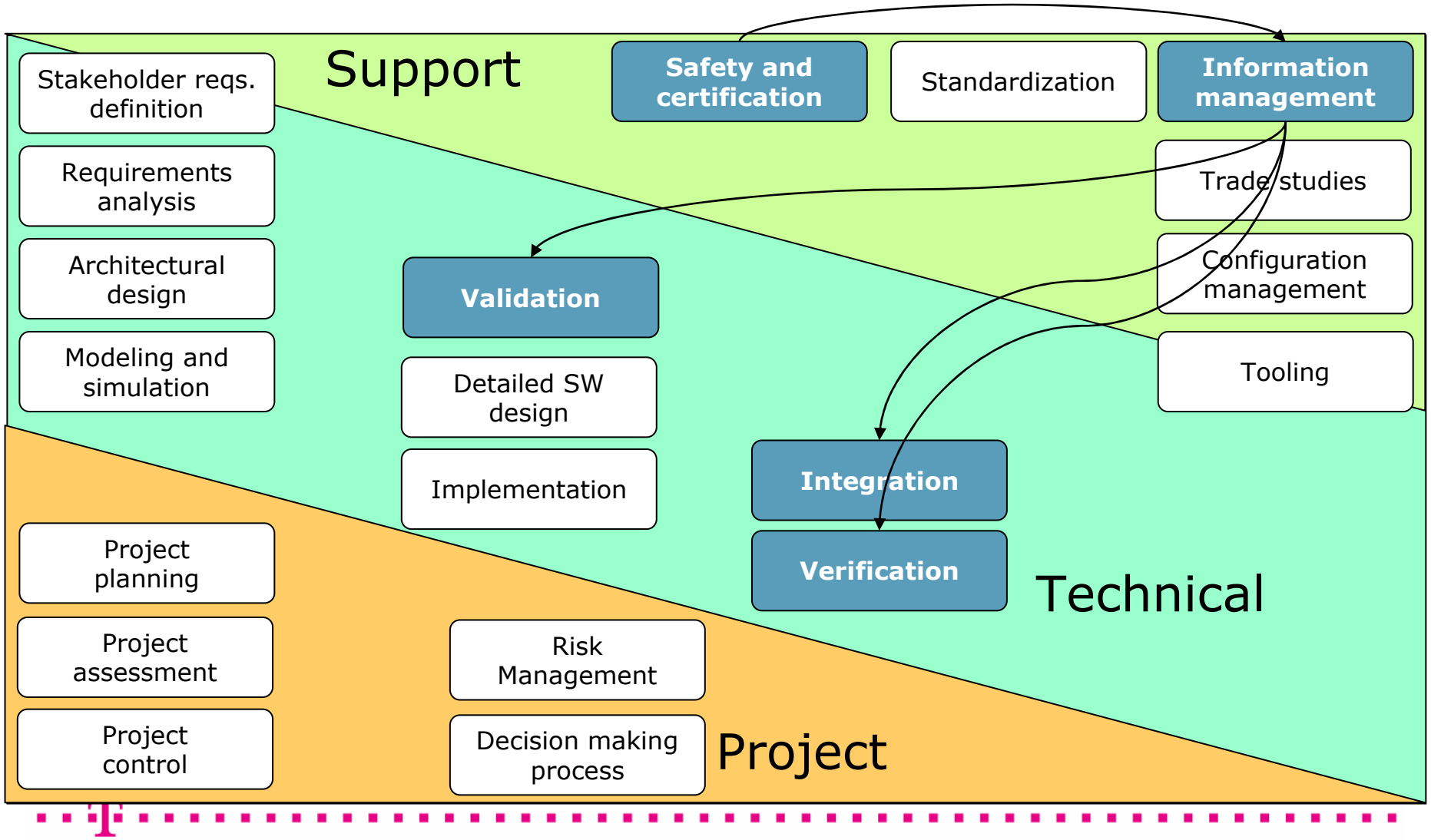
ACC Stop&Go
BFD
ALC
KSG
Internet Portal
GPRS, UMTS
Telematics
Online Services
Blue-Tooth
Car Office
Local Hazard Warning
Integrated Safety System
Steer/Brake-By-Wire
I-Drive
Spurhalteunterstützung
Personalisierung
Force Feedback Pedal

2000

Verification of Safety Critical Systems
Fields of Activities

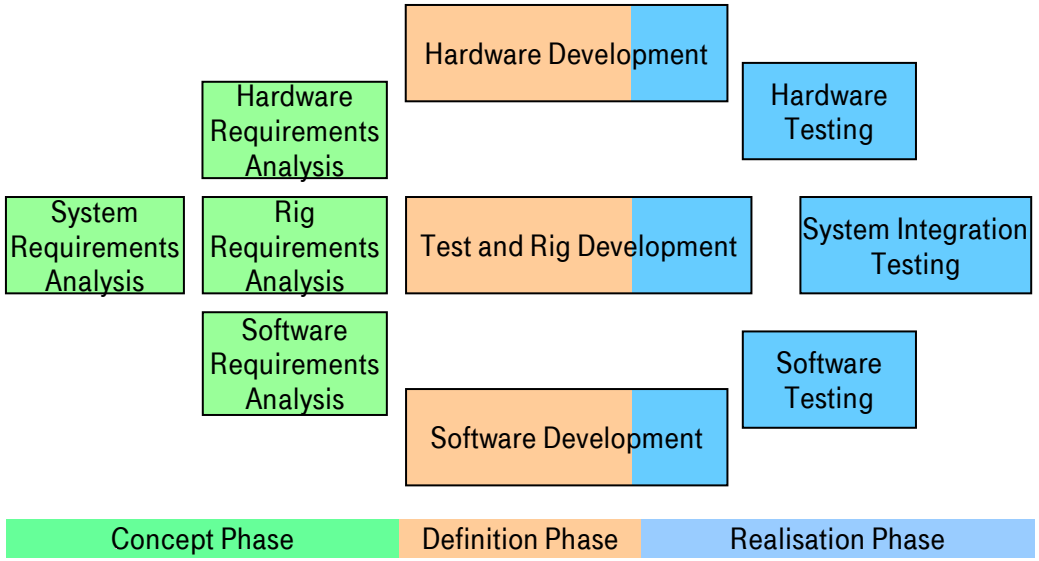


Verification of Safety Critical Systems
Fields of Activities



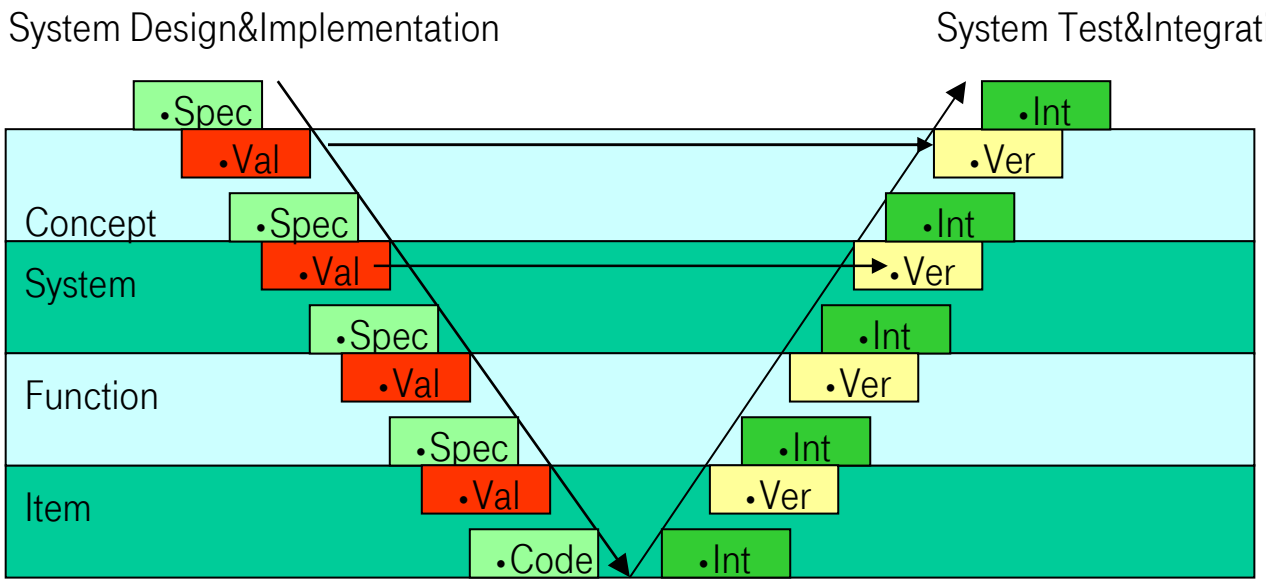
Verification of Safety Critical Systems

Development Process („Classic Approach“ ca. 1985)



- Individual System Development for each Program
- Hierachical System Breakdown (V-Modell)
- Standardized SW-Entwicklungs-modell (e.g. DoD-Std 2167A)
- Strikt tracing Requirements -> Implementation -> Verification
- Documentation of all (Intermediate) Results
- One time execution of the Development Process (Waterfall Model)
- Long Developtimes (10 and more years)

Verification of Safety Critical Systems
Development Process



Verification of Safety Critical Systems

Development Process („New Challanges“ since 1995)

- Significant Extension of functionality and thus complexity
- Development cost reduction by
  - Reduction of development duration (typical 5 years)
  - Utilisation of readily available products (Commercial Off The Shelf)
  - Modifications of readily available products (Modified Off the Shelf)
  - Industrialization of SW-Development (Executable specifications, CASE-Tools)
- Broadening of application base
  - Covering of multiple application scenarios (> 20 variants for military avionic systeme) with configurable Basis-SW
  - Concurrent support for different development / configuration stages in operative use
  - Modularized SW-Design
- Support for SW-Maintenance by
  - Integration of additional functionality
  - Extraction of obsoleten SW-Componentes

Verification of Safety Critical Systems

Concequences for requirements to verification.

Classic requirements	New requirements
Standardized development and verification process	Adaptation of verification process e.g. for COTS/MOTS und reused components
Long lasting verification run (months)	Significant reduced verification run duration (days)
Strict Traceability Requirements -> Implementation -> Verification	
Complete Documentation of all (intermediate) results	
Few (ideal1!) test run for SW-Verification	Multiple (1 per configuration / variant) test runs

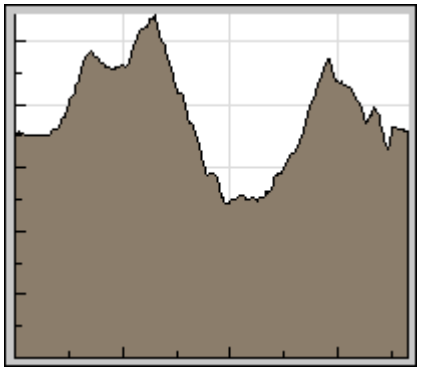
- ARP4754 / DO-178B / DO-254 conformal verification process
- Modular verification concept, close coupling with configuration management
- Reduction of test run duration
- Reduction of test error rate (wrong good, wrong failed)

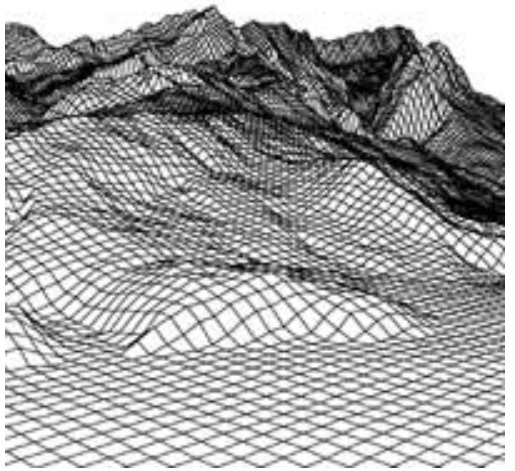
⇒ Utiliztion of Test-Tools (Cantata, VectorCast, TestMATE, ...)

⇒ Automatic test run execution and document generation

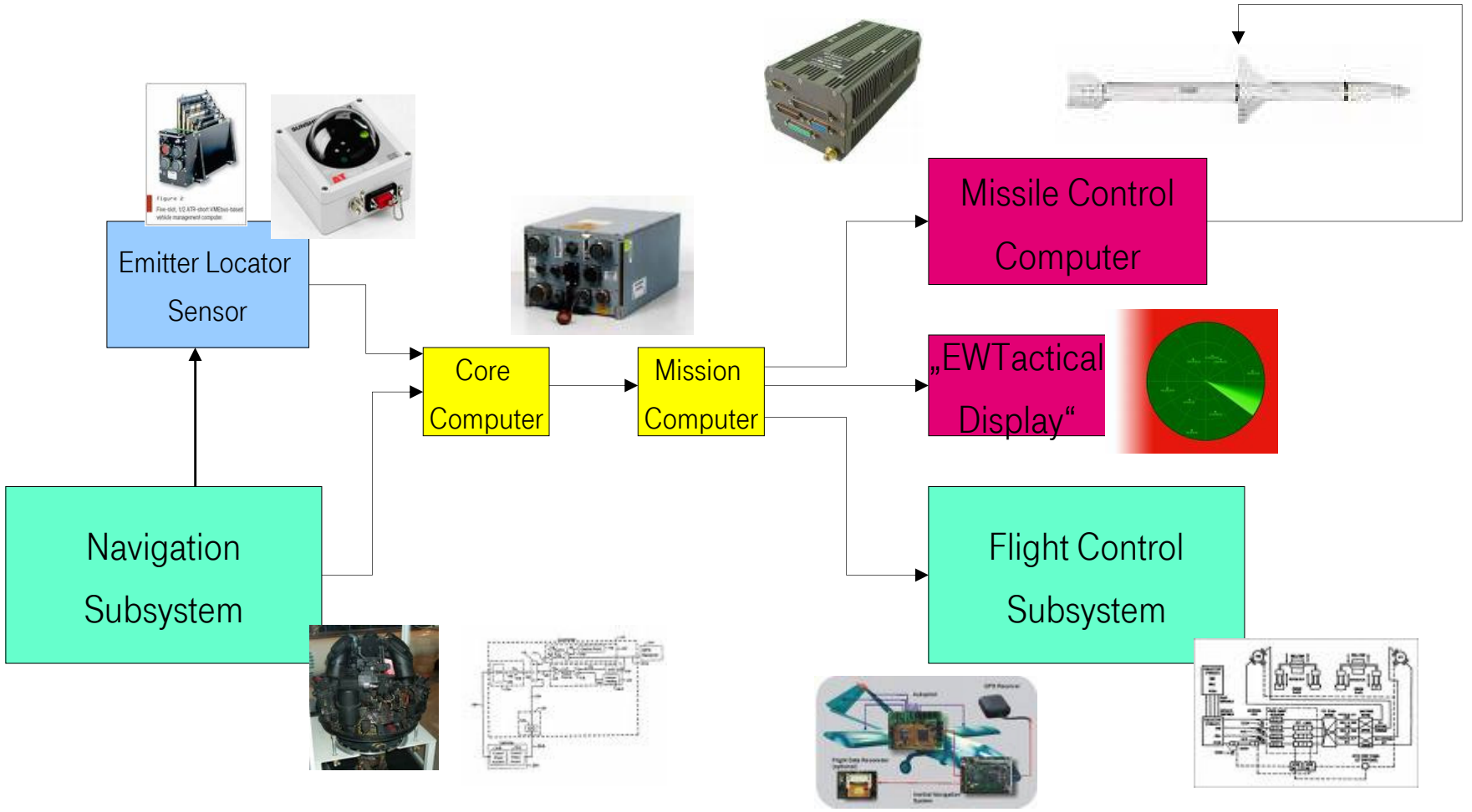
⇒ **Transition from manufactur to industrial testing**

Verification of Safety Critical Systems
An Aerospace Example





Verification of Safety Critical Systems
System Breakdown



Verification of Safety Critical Systems
What to Deal With - Methods of Verification

- Simulation
- Analysis, Engineering Judgement
- Similarity of requirements or design
- Demonstration, Prototyping or Mock-up
- Reviews or Audits
- Inspection
- Test
- Operational Trials

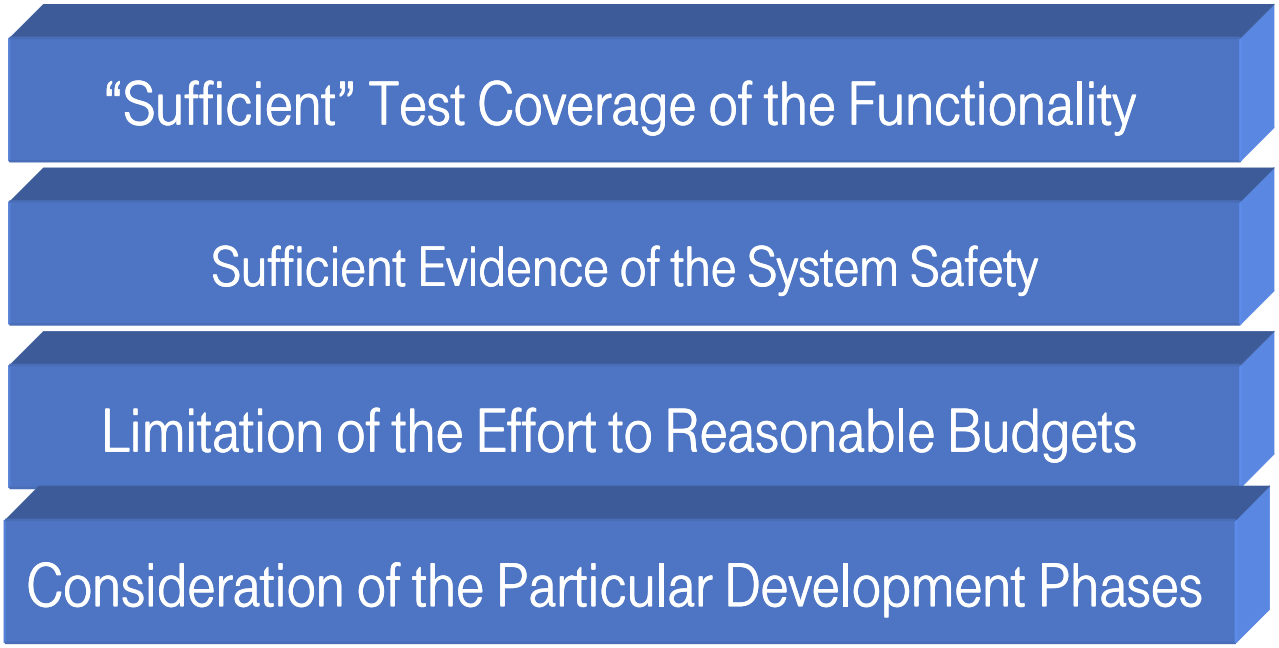
- Flight Test
- Aircraft Ground Test
- System Integration Test
- HW-SW Integration Test (Bench)
- SW-SW Integration Test
- Coding Unit Test

Verification of Safety Critical Systems
Let‘s Find an „Optimized Verification Strategy“

From Theory.....

..... To Experience

Verification of Safety Critical Systems
Requirements on an Optimized Verification Concept



Verification of Safety Critical Systems

Elements of a Good Verification Strategy

- Establish Automated Tests Early
- Realize the Coherence of Functions and Test
- Use Data Bases and Generators
- Coordinate all Test Stages
- Realize End to End Tests
- Verify Requirements &Functions Early

Verification of Safety Critical Systems
Essential Columns of the Verification Strategy - Focusing

Use the specific advantages of each test stage

Automate Tests

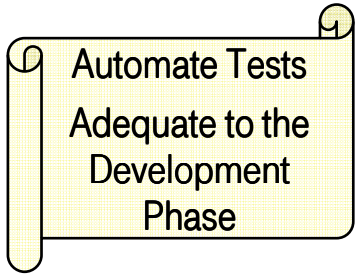
Adequate to the Development Phase

Verification of Safety Critical Systems
Optimized Strategy (1)

Use the specific advantages of each test stage

▪ Verify requirements&functions early	→	Early requirements&design verification
▪ Realize end to end tests	→	User's needs
▪ Coordinate all test stages	→	Integrated test concept
▪ Realize the coherence of functions and test	→	Coverage and traceability

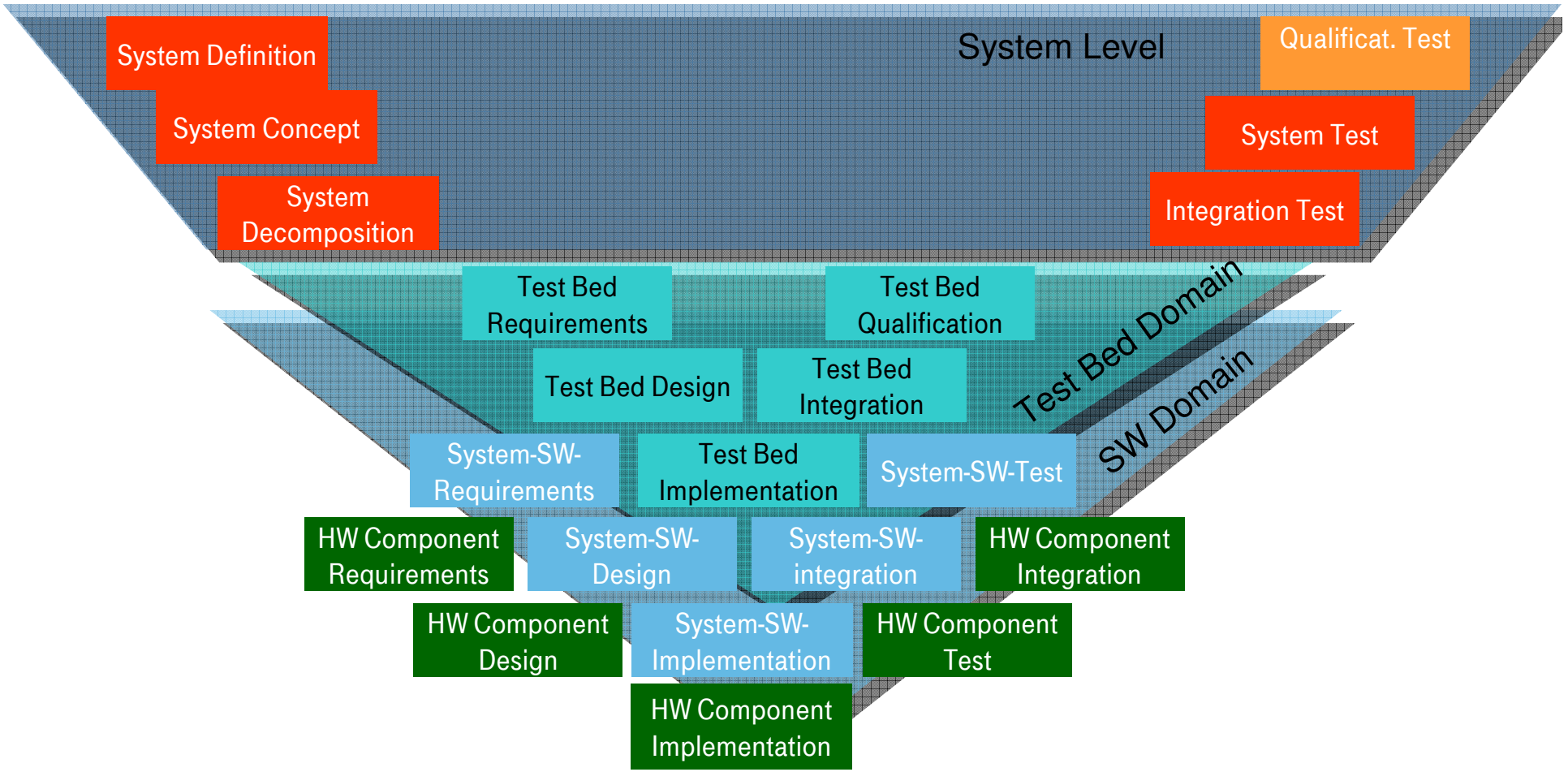
Verification of Safety Critical Systems
Optimized Strategy (2)



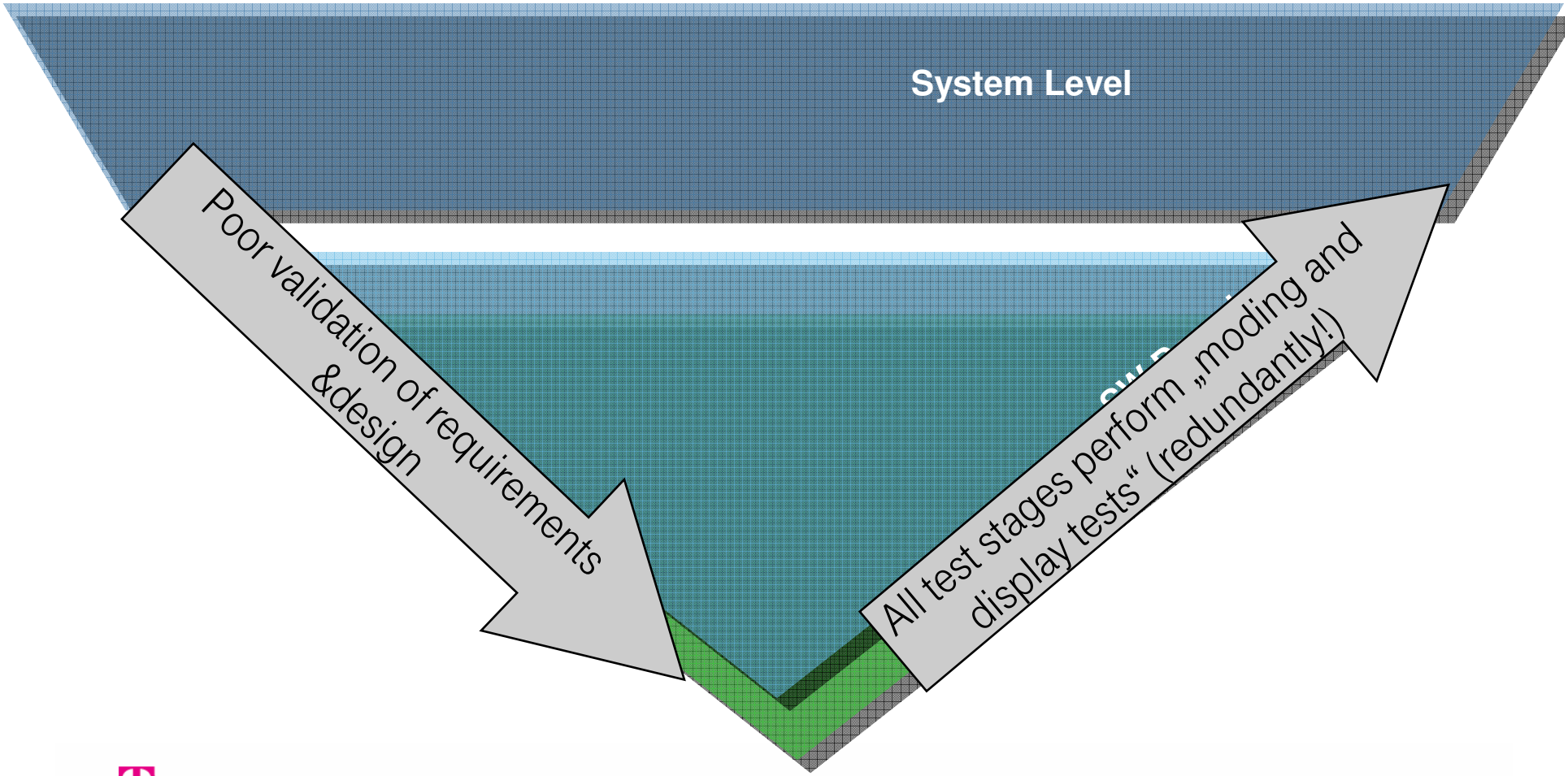
- Establish automated tests early → Reduction of repetitive effort
- Use data bases and document generators → Reduction of document effort

Verification of Safety Critical Systems
Experience

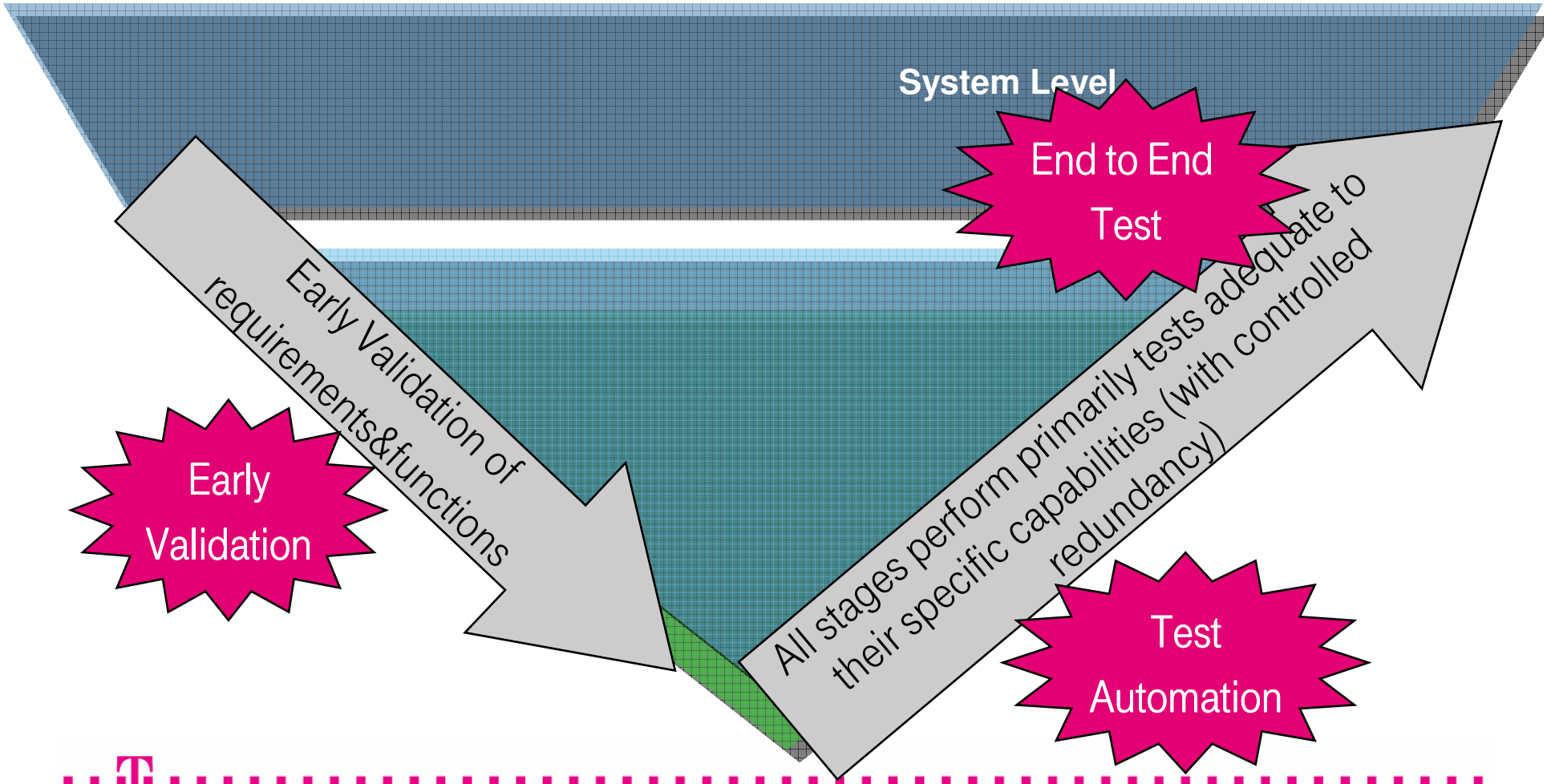
Verification of Safety Critical Systems
Effort



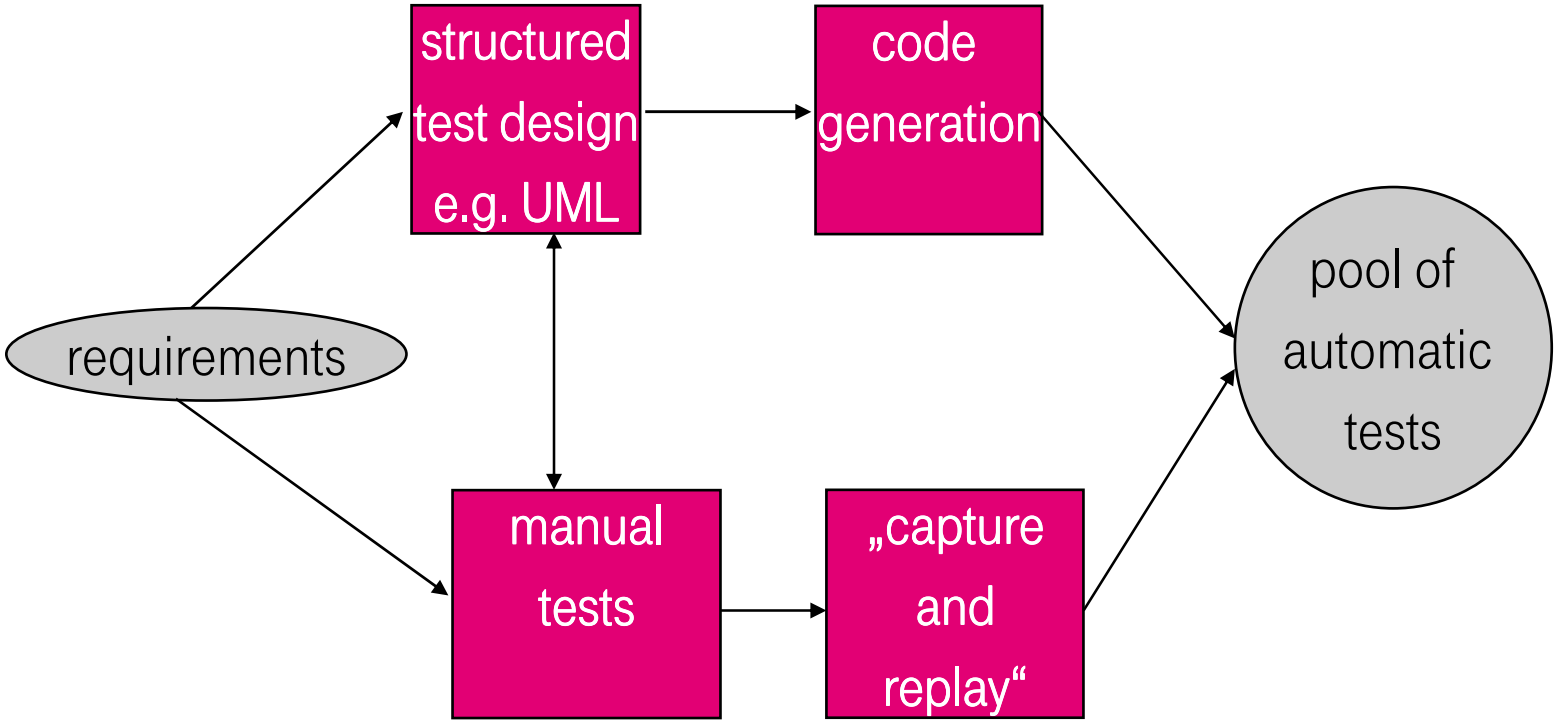
Verification of Safety Critical Systems
„Bad Case“



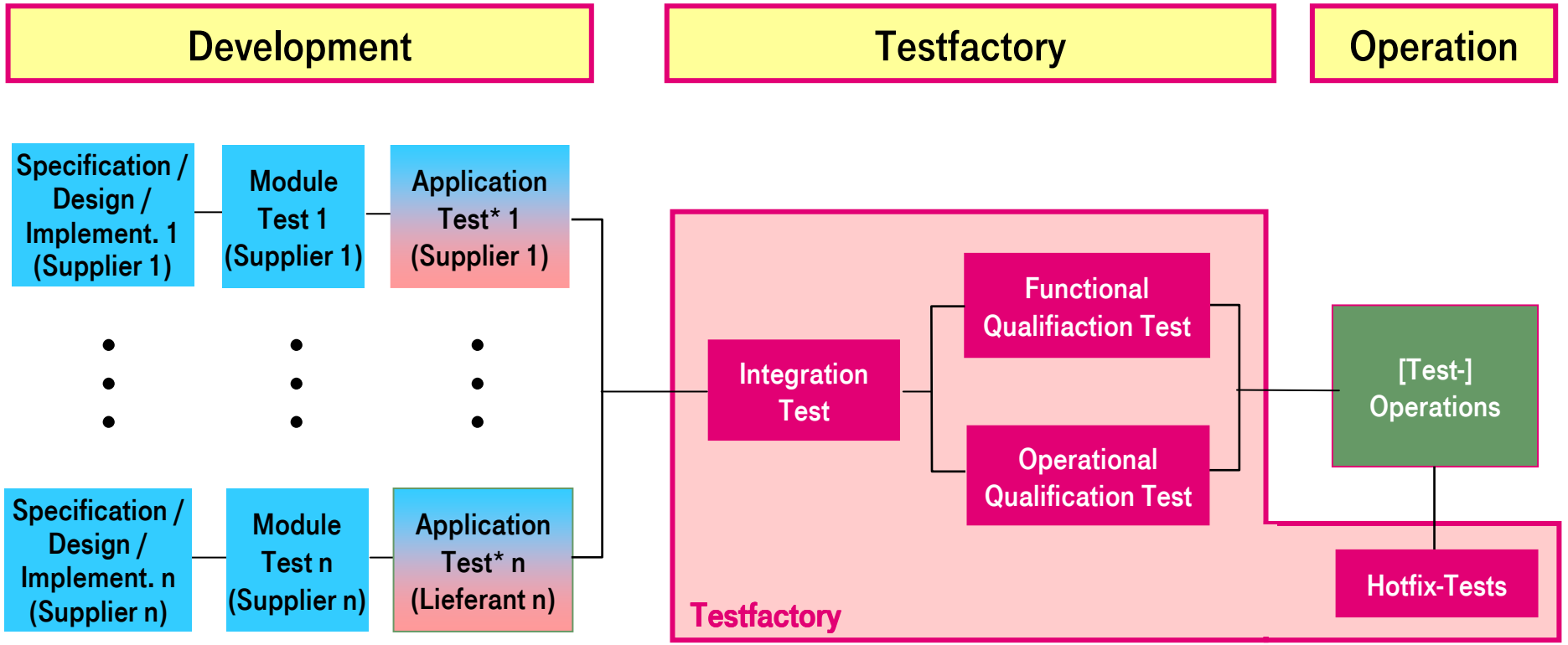
Verification of Safety Critical Systems
„Good Case“



Verification of Safety Critical Systems
Two Automation Concepts



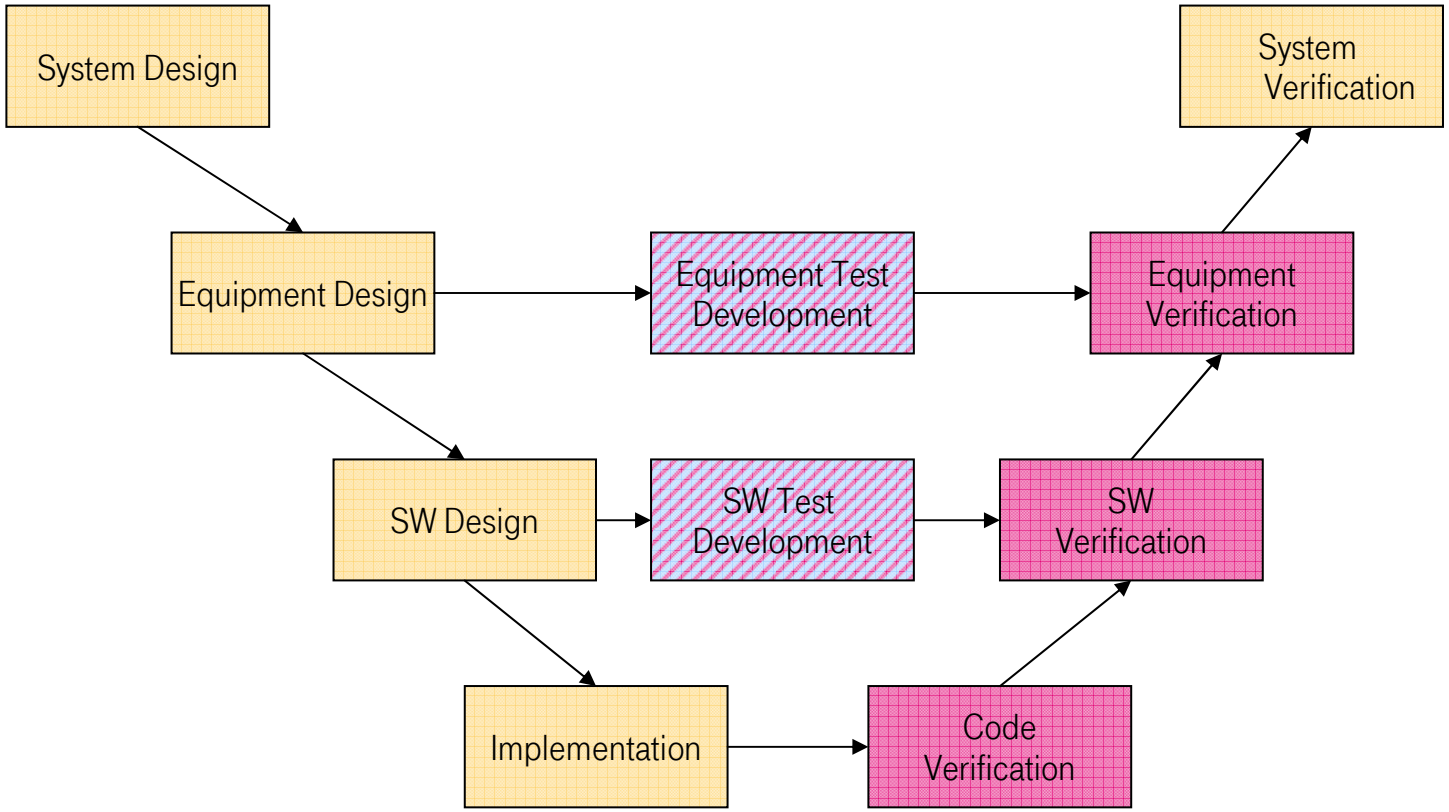
Verification of Safety Critical Systems
Testfactory Concept



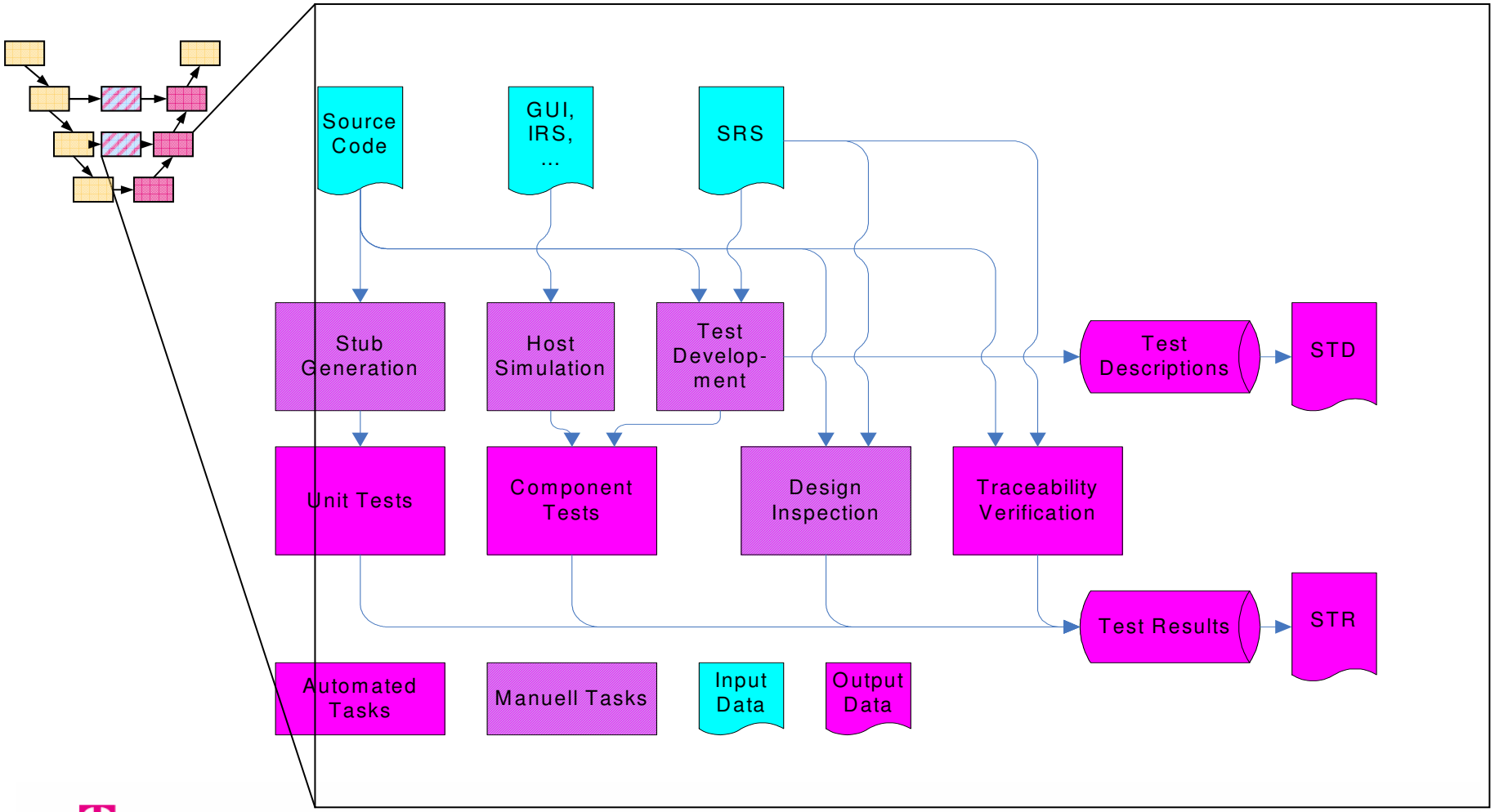
*incl. bilateral interfaces

Operational roll out

Verification of Safety Critical Systems
Development Process



Verification of Safety Critical Systems
SW Verification



Einführung modellbasierter Technologien im industriellen SW-Engineering Umfeld

Hubert B. Keller

Software-Workshop "Effiziente Entwicklung zuverlässiger Software und methodisches Instrumentarium"
Technologiepark Karlsruhe, 24.01.2008

Übersicht

- Situation und Ziel
- Modellbasierte Technologien
- Technologieeinführung
- Prozessaspekte
- Projekt konkret
- Bewertung und Ausblick

Situation und Ziel

Existierendes System

- Altsystem 300 PA (eng gekoppeltes und verteiltes Gerätesystem)
- Alter > 10 Jahre
- Innovation < 10%, Wartung > 90%
- Monolithische Struktur, prozedural, HW-orientiert

Ziel

- Flexibel auf Requirements reagieren (Skalierung, …
- Hohe Qualität (Architektur, Weiterentwicklung, Doku, …)
- Effizienz (Entwicklung, Portabilität, time-to-market, …)
- Zertifizierbarer Kernel mit in-house und externen Add-Ons
- Security Aspekte berücksichtigen

Situation und Ziel - Fehler und Phasen

- HSE Out of Control – Analyse (2003)
  - 44% had inadequate specification as their primary cause
  - 15% design and implementation
  - 6% installation and commissioning
  - 15% operation and maintenance
  - 20% changes after commissioning

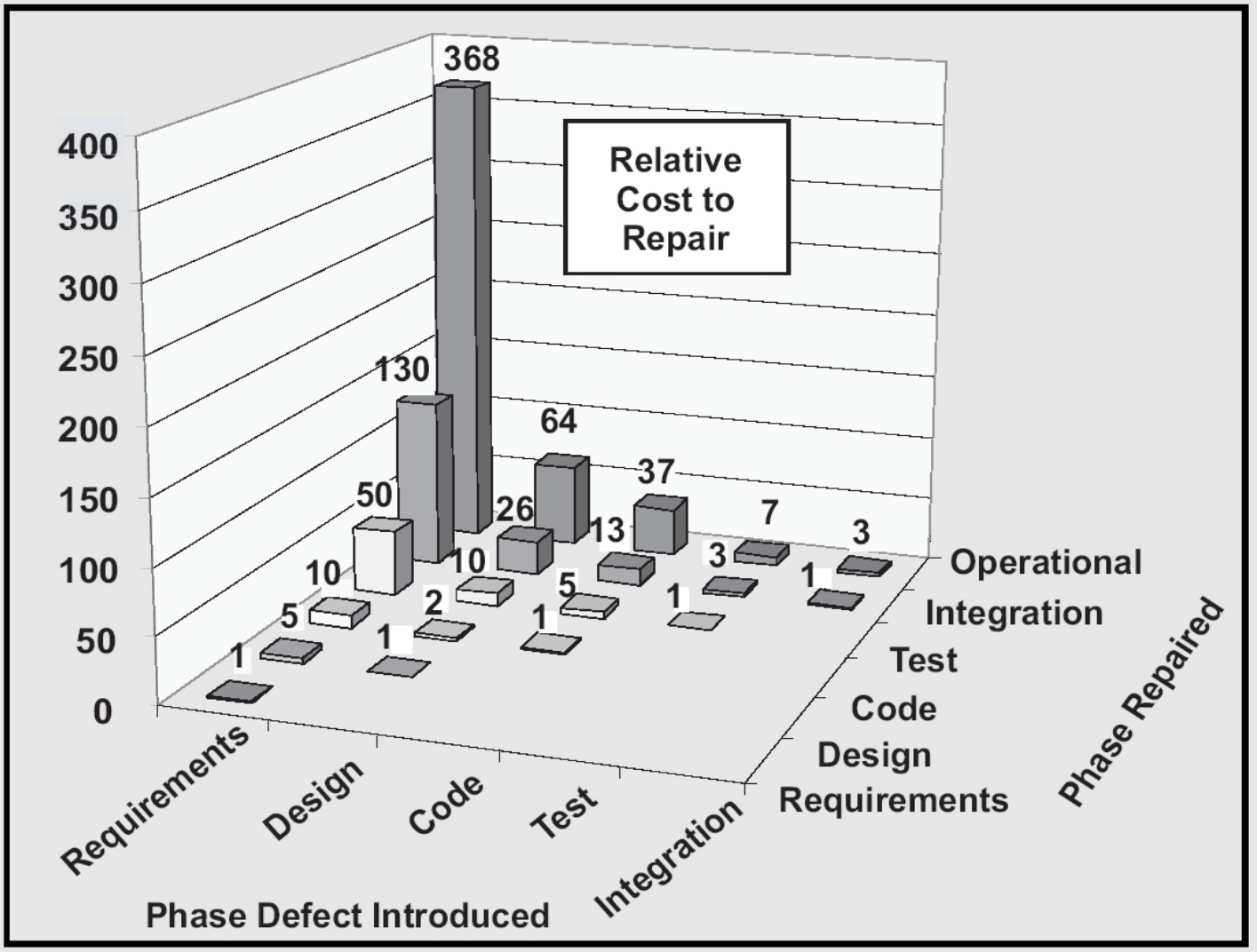
- 3/5 of control system failures are built-in before operation commences

Analysephase wird deutlich unterschätzt
→ Mehr Fokus auf Analysemodell!

(Out of control - Why control systems go wrong and how to prevent failure. Health and Safety Executive (2003), No 238)

Situation und Ziel - Kostenpropagierung

Figure 4: *Relative Cost of Software Fault Propagation*



Situation und Ziel - Fehler und Kosten

- Tom DeMarco / Nosek und Palvia:
  - 75% der Gesamtkosten müssen z. Zt. für die Wartungsphase eingesetzt werden (Fehlerbehebung statt Neuentwicklung)
  - Bis zu 50% dieser Kosten resultieren aus Verständnisproblemen

- Ramberger (Distribution of Error Types):
  - 49% sind Dokumentationsfehler
  - 54% „value never returned“ und
  - 31% „value not documented“

Modellbasierte Technologien - Warum Modelle?

Mensch als Entwickler

- Software Engineering erfordert
  - Kognitionsfähigkeiten [Denkfehler]
  - Explizierungsfähigkeit [Beschreibungsfehler]
  - Kommunikationsfähigkeit [Verständnisfehler]
  - Teamfähigkeit (Fachkompetenz ++)

- Modelle als Grundlage zur Entwicklung
  - Terminologie und Nomenklatur als Grundlage
  - Syntax, Semantik und Pragmatik von Modellbeschreibungen
  - Know How / Wissensmanagement

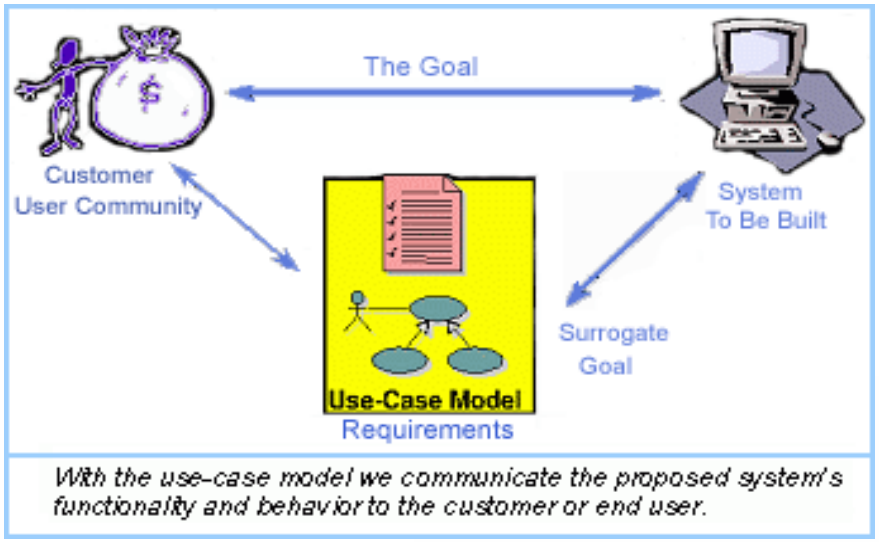
Modellbasierte Technologien - Engineering

Engineering von SW Systemen

- SW Engineering auf Modellbasis
  - UML als Modellierungssprache (Struktur und Verhalten)
  - Konzentration auf logische Struktur
  - Verständlichkeit
  - Transformationsgedanke

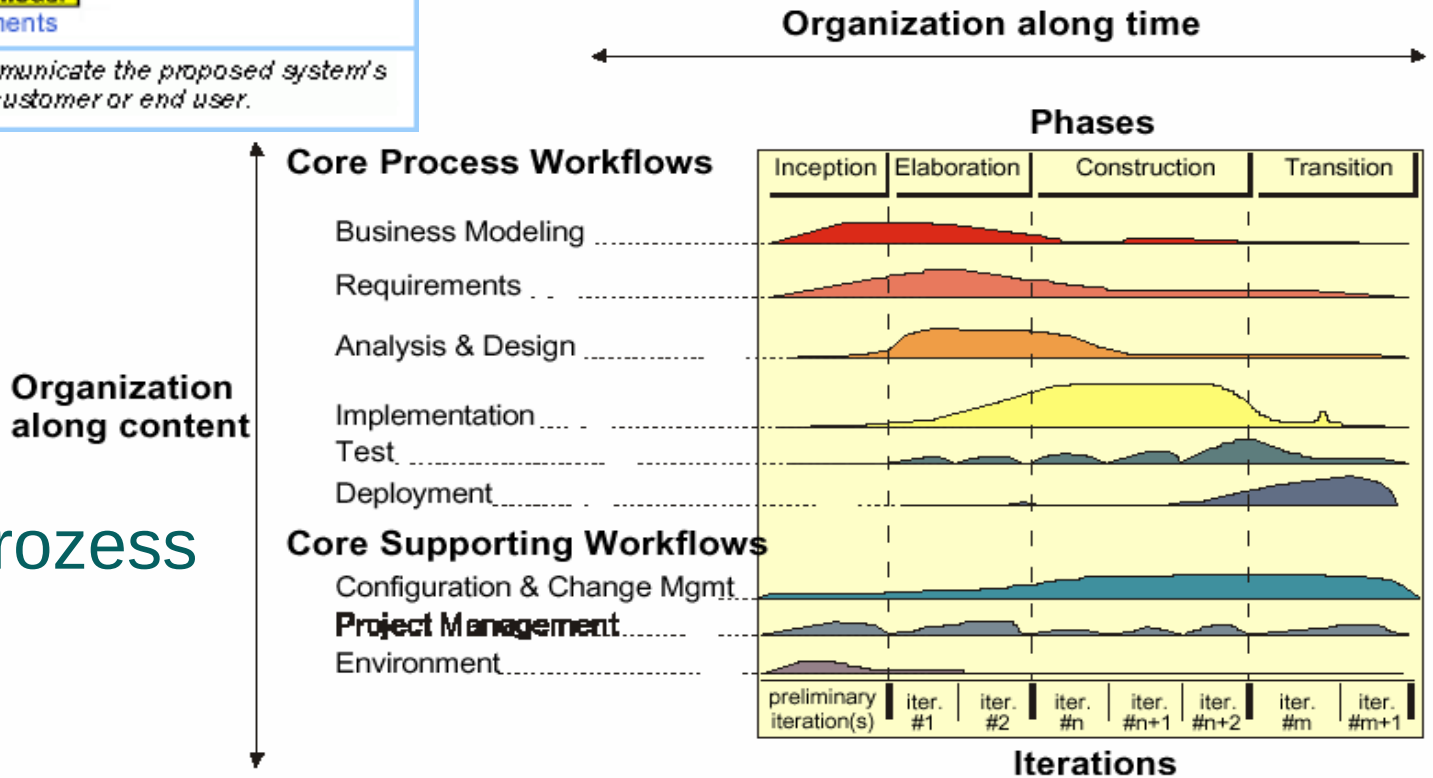
- SW ReEngineering auf Modellbasis (Migration)
  - Modell als Anforderungsdefinition
  - Modell als Maß für Wiederverwendbarkeit Altsystem
  - Modell als Zieldefinition (Machbarkeit)

Modellbasierte Technologien - MDA und RUP

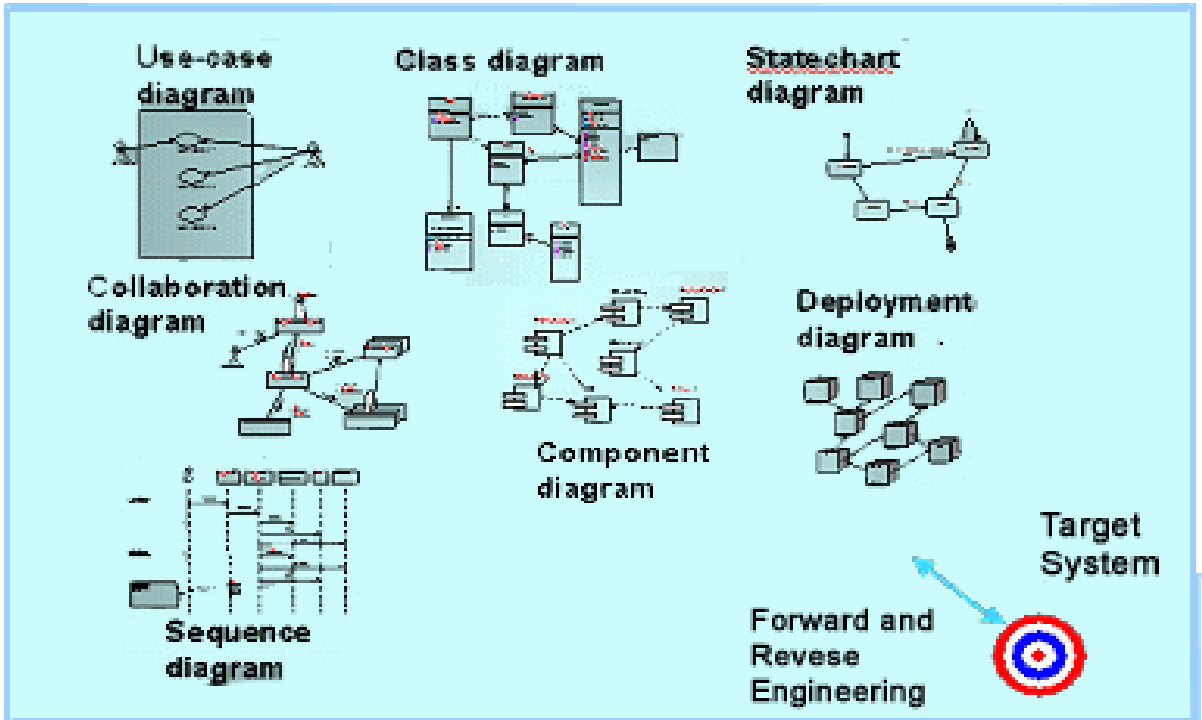


Projekt und Ziel

Weg und Prozess



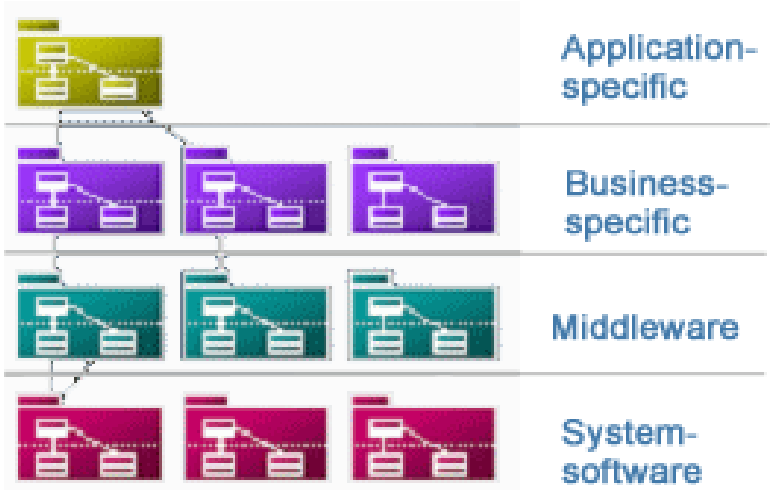
Modellbasierte Technologien - I*terativ*, I*nkrementell*, OO



Visual modeling tools enable managers and developers to clarify important design decisions and to manage system complexity.

Konzepte und UML Modellelemente

Architekturziel

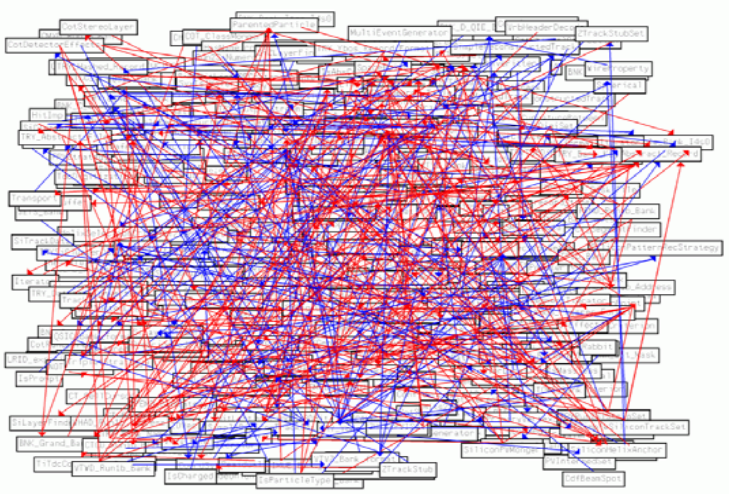


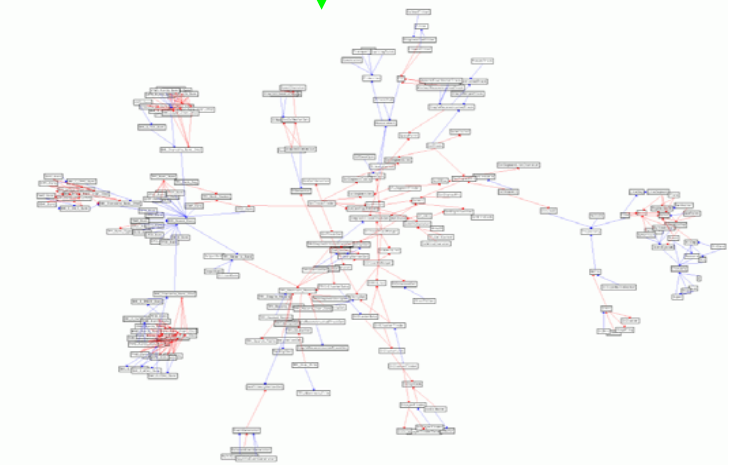
Component-based architecture with layers

Modellbasierte Technologien - ReEngineering

ReEngineering – Strategische Bedeutung

- Investition in ein Software-System, mit den Zielen:
  - flexiblere Architekturen
  - bessere Strukturierung
  - kostengünstige Funktionserweiterungen, Fehlerbehebungen und Technologiewechsel



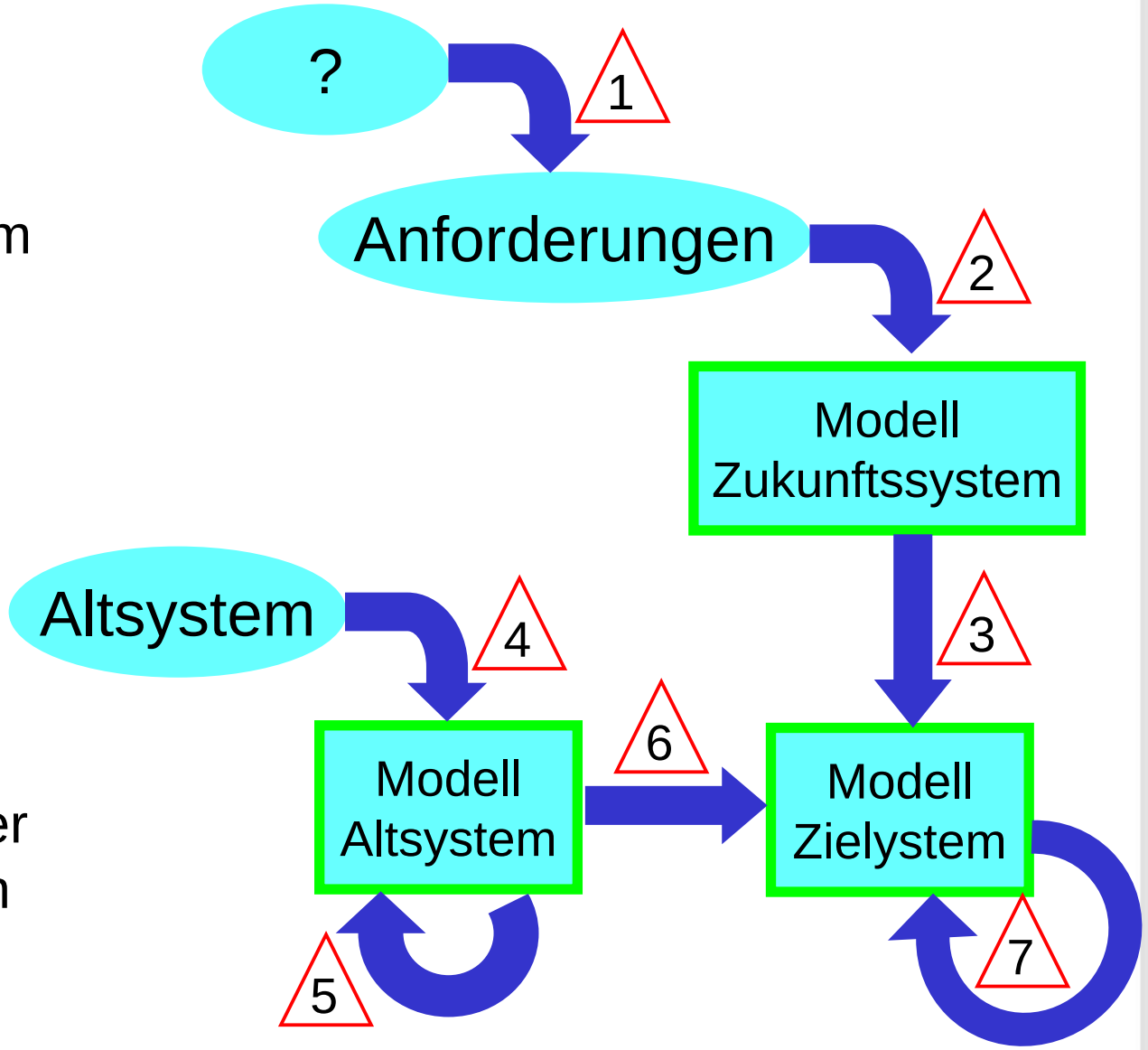


- Investitionsschutz durch Wiederverwendung
  - Extraktion von Know-How und Code in Form von Komponenten aus Altsystemen

Modellbasierte Technologien – ReUse Altsystem

Aktivitäten

1. Anforderungsanalyse
2. Modellierung Zukunftssystem
3. Entwicklung Kernsystem
4. Modellbildung, Reverse Engineering
5. Modellanalyse/ReUseability
6. Modell Migration
7. Iterative, inkrementelle Erweiterung des Zielsystemkerns mit iterativer und inkrementeller Migration des Altsystems



Technologieeinführung - Vorgehen

- Vermittlung der Technologien vor Projektbeginn (Schulung)
- Anwendung und Vertiefung projektbegleitend (Grundverständnis + learning bei doing)
- Vermittlung des Prozessgedankens und der Vorgehensweise (RE → Kern-Requirements → Architektur → Vorgehen inkrementell, iterativ, OO)
- Vermittlung neuer / bewährter Methoden
- Konzept zur Integration von Altsystemen (vorhandene Werte) bzw. Restrukturierung

Technologieeinführung - … Vorgehen

- Freistellen von Personal und Motivation für Ziel (Zeitdruck, Lernbereitschaft, ...)
- Definition von Leitprojekten (Kernsysteme, Leitlinien, gesteuerte Evolution)
- Zusammenführung von
  - Geschäftsprozess-gesteuerte Requirements-Analyse
  - Architekturierung und Verhaltensmodellierung
  - mit funktionalen Komponenten des Altsystems
- Freischaufeln von Wartung – hin zu modellbasierter Entwicklung und automatischer Codegenerierung

Technologieeinführung - weitere Aspekte

- Requirements Engineering für Kernanforderungen
  → Zielsystemarchitektur (Kern-Anforderungen, Zukunftsträger)

- Risikoanalyse und –bewertung
  → frühes Abklären, Prototyp, Zeit, Aufwand

- Funktionale und nichtfunktionale Anforderungen
  → Zielsystemeinbettung / Umfeld / Zukunftsfähigkeit / Priorität HW

- Projektplanung / zeitliche Randbedingungen
  →Egoismus (StoryCard), echte Personalverfügbarkeit, Reviewprozesse, …

Prozessaspekte – SW als Produkt

- Aktivitäten (Schritte) im Entwicklungsprozess und zugehörige Methoden
- Produkte als Ergebnis von Aktivitäten
- Verwaltung der Produkte
- Personen und Rollen
- SWE + KM + QS + PM
- Definition von Rückkopplungen im Prozess (einschließlich Kundeneinsatz) – Ziel CMMI

Prozessaspekte - Phasen

- Analysephase – Requirements Engineering / Use Case Modellierung / Nebenläufigkeit und Verteilung
- Designphase – Komponenten, Interaktion, Verhalten, Architekturen, Client-Server, Kommunikation
- Automatische Codegenerierung – Transformation von Modellen mit Anreicherung und Prüfung / Templates / UML Profil
- Test- und Integrationsphase – Modellgetriebenes Testen in Planung
- Versions- und Variantenverwaltung - Standardtool

Prozessaspekte - Rollenverständnis

- Rollen, z.B.
  - Architekt des Systems
  - Business Process Analyse/Requirements Management
  - Begriffsweltkonsistenz
  - MDA und UML Profilierung (Modell und Bedeutung)

Prozessaspekte - Review

- Kernanforderungen darstellen, durchgehen und verifizieren
- Semantik Modellkomponenten und Konsistenz Modell
- Architekturierung-Ableitung
  - Architektur und Grundprinzipien des Systemkonzepts (Client-Server, ...)

- Systemkomponenten (wesentliche) darstellen
  - Teilsystemstrukturierung und Interaktion
  - Anbindung an Altsystem

- Prinzip: Anforderung, Konzeption, Modell, Alternativentscheidung

Projekt konkret – Use Cases

Aus Vertraulichkeitsgründen sind die Folien aus dem konkreten Projekt im Netz nicht verfügbar.
Für Rückfragen zu Details stehe ich gerne zur Verfügung.

Ich bitte um Verständnis.
Danke.

Bewertung und Ausblick

- Machbarkeitsstudie
  - Technologievermittlung
  - Projektbegleitende Technologievertiefung
  - Projektbegleitende Beratung
  - Projektbegleitende Analysen (Risiken, Schwachstellen)
- Definition von Leitprojekten
- Schaffung einer kritischen Masse (Projekte/Menschen) zur gesteuerten Evolution der Technologie bei Indsutriepartner
- Modelle geeignet für
  - Transparenz von Know How, In-Team Kommunikation
  - Kommunikation zu Vertrieb, Produktmanagement, GF, Kunden

… Bewertung

- Use Case für Requirements Engineering
- Erarbeitung einer konsistenten Begriffswelt
- Modelle für Architektur und funktionalem Verhalten

- Werkzeugeinsatz **OpenAmeos** (MDA basierte Codegenerierung, GUI Layout, Versionsverwaltung im Sinne Continous Integration,…)
- MDA für Effizienz- und Qualitätssteigerung (Ausschluss zufälliger Fehler, Systemgenerierung über UML Profilierung und Templates)

- Prozessdefinition als Rückgrat der Software Entwicklung mit klarer Rollenfestlegung
  - Architekturverantwortlicher
  - MDA Verantwortlicher

Ausblick

- Fokuswechsel von HW-orientierter zu SW-orientierter Strategie
- Mehrwert transparent und erkannt
- Leitprojekt als Keimzelle für umfassende Technologieeinführung

- Gruppe Leitprojekt zur Abteilung aufgewertet
- Produkt aus Leitprojekt zukünftig Kernprodukt

- Umfeldentwicklungen müssen sich an Leitprojektvorgehen orientieren
- CMMI Zertifizierung angestrebt / SW Prozessdefinition in Vorbereitung

- Modellbasierte Technologien bringen definitiv Mehrwert
  (30-35 PA Umfang mit ca. 20 PA trotz MDE + RE \ Vorwissen)

Dr. Peter DENCKER
ETAS GmbH

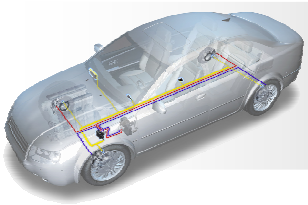
Modellbasierte Entwicklung effizienter SW

in der Automobilindustrie

Software-Workshop

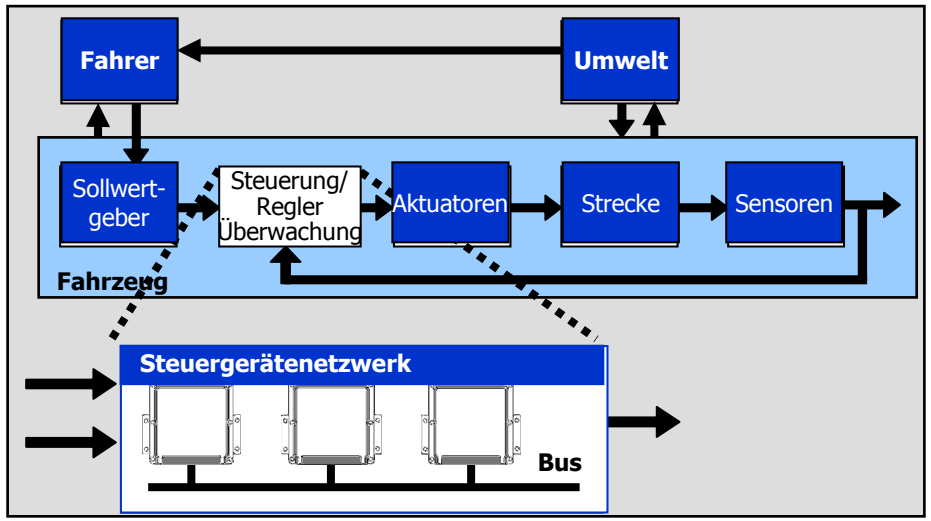
24.01.2008, Technologiepark Karlsruhe

Agenda



- **Ist Automotive Software speziell?**
- **Konsequenzen aus den speziellen Anforderungen**
- **Effizienz im Entwicklungsprozess**
- **Automotive Software Modellierungswerkzeuge**
- **Warum automatische Codeerzeugung?**
- **ASCET Code Generation**
- **Code Effizienz**
- **AUTOSAR**
- **Zusammenfassung**

Ist Automotive Software speziell?
Beispiel: Fahrwerksregelungssystem

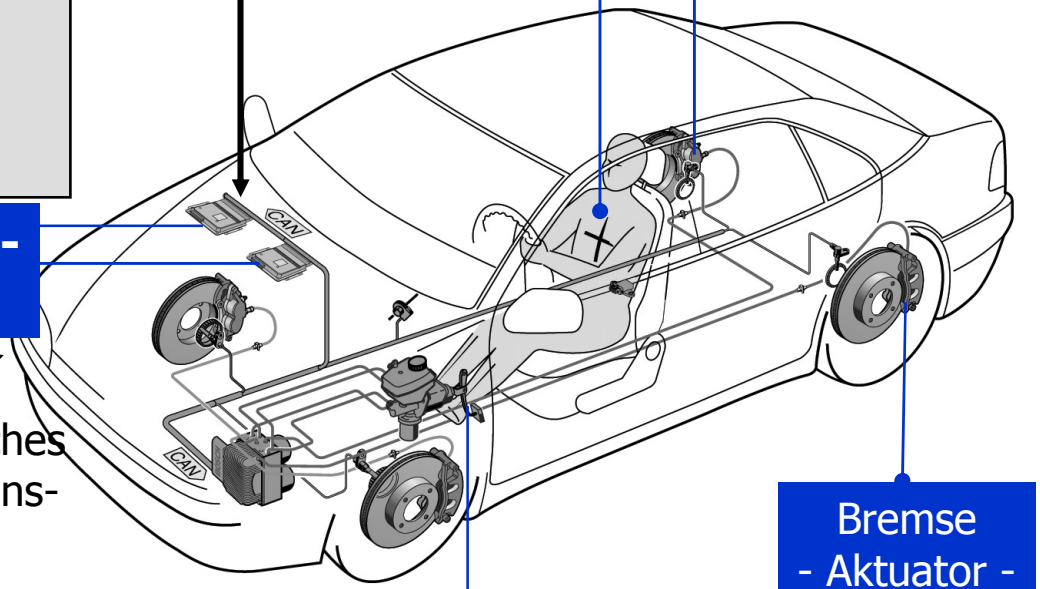


Werkzeug (Service Tester)

Raddrehzahl-sensor

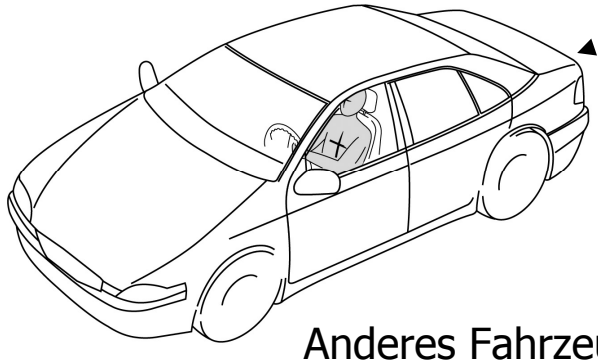
Fahrer

Fahrzeug



Steuergeräte-netzwerk

Umwelt



Dynamisches Navigations-system

Anderes Fahrzeug

Bremse - Aktuator -

Bremspedal - Sollwertgeber -

Automotive Software ist speziell, weil

- Der Preis von Steuergeräten in Verbindung mit **großen Stückzahlen** ergibt eine **extreme Kostensensibilität**
  - Beispiel VW Golf: sinkt der Preis des Motorsteuergeräts um 20€, so **spart** VW **400 Mio €**!

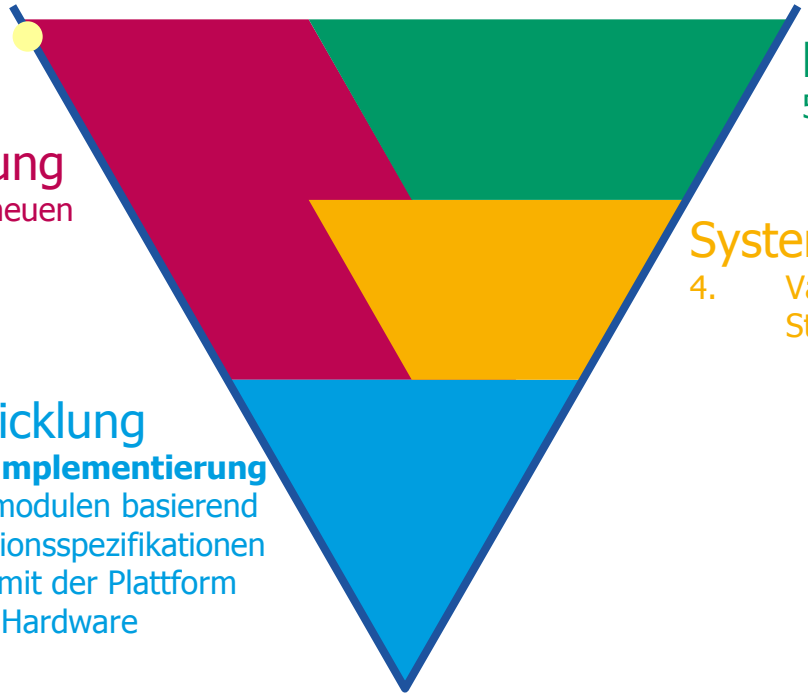
Modell	Golf 1	Golf 2	Golf 3	Golf 4
Stückzahl	6.780.050	6.301.000	4.805.900	4.300.000

- Mangelnde **Zuverlässigkeit** der Software ist ein **Kostenrisiko**
  - 9. Dezember, 2007:
    Ford ruft in dem USA 1,17 Millionen Fahrzeuge wegen einer Fehlfunktion des Motorsensors zurück.
  - Geht man von 100$ aus pro Fahrzeug, wird das sehr teuer!

Automotive Software ist speziell, weil

- Die **Variantenvielfalt** fordert **spezielle Software Architekturen** und eigenen Entwicklungsmethoden
  - Beispiel Daimler: im Jahr 2006 wurde nur eine „Dublette" der S-Klasse in Sindelfingen gebaut, d.h. es wurden 10.000 **verschiedene** S-Klasse PKW in 2006 gebaut.
  - Gleichzeitig **große Zahl** verschiedener PKW **Modelle** von vielen Herstellern
- Die **hohe Modellfrequenz** erfordert **schnelle Entwicklungszyklen**
  - Beispiel VW: 5 neue Golf Modellreihen in 30 Jahren, d.h. alle 6 Jahre ein kompletter Redesign, alle 2-3 Jahre eine Modellpflege.
- **90%** aller **neuer Fahrzeugfunktionalität** kommt aus der **Elektronik**
- **60%** der **Elektronikentwicklung** sind **SW Entwicklungskosten**

Automotive Software ist speziell
Konsequenzen für den Entwicklungsprozess



Funktionsentwicklung
1. **Spezifikation** von neuen Steuerfunktionen

Softwareentwicklung
2. **Design** und **Implementierung** von Softwaremodulen basierend auf den Funktionsspezifikationen
3. **Integration** mit der Plattform Software und Hardware

Systemtest
4. Validierung und Verifikation des Steuergerätes

Messen / Kalibrieren
5. Finale Abstimmung (Kalibrierung) der Steuerfunktionen

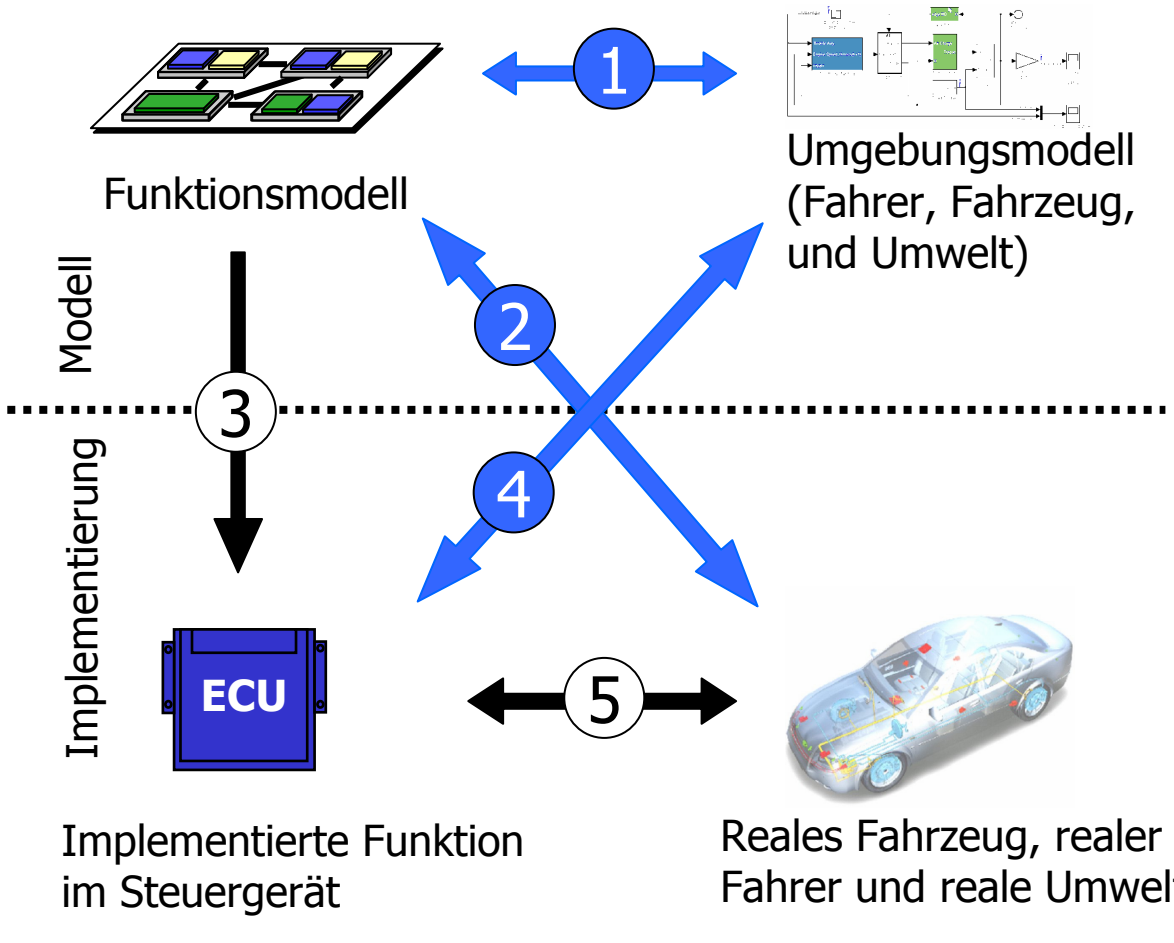
Durchgängige Toolunterstützung des obigen V-Modells
von der Spezifikation bis zum Kalibrieren

Automotive Software ist speziell
Konsequenzen für die Software/Hardware Architektur

- Software Architektur
  - Aufteilung in Plattformsoftware und Applikationssoftware (Varianten)
  - Parameter, Kennlinien und -felder statt Funktionen (Varianten)
  - Integer-Arithmetik statt Floatingpoint-Arithmetik (Stückkosten)
- Standardisierte Steuergeräte mit Plattform Software (Zuversässigkeit, Stückkosten)
- Prototyping so früh wie möglich (Zuverlässigkeit, Zykluszeit)
  - Bypass Technologie ist die Folge
- Meistens wird eine neue Funktion aus bestehenden Funktionen abgeleitet und ist nicht komplett neu (Modellfrequenz).
- Sehr hohe Codeeffizienz (Stückkosten, Speicher+Geschwindigkeit)

Effizienz im Entwicklungsprozess

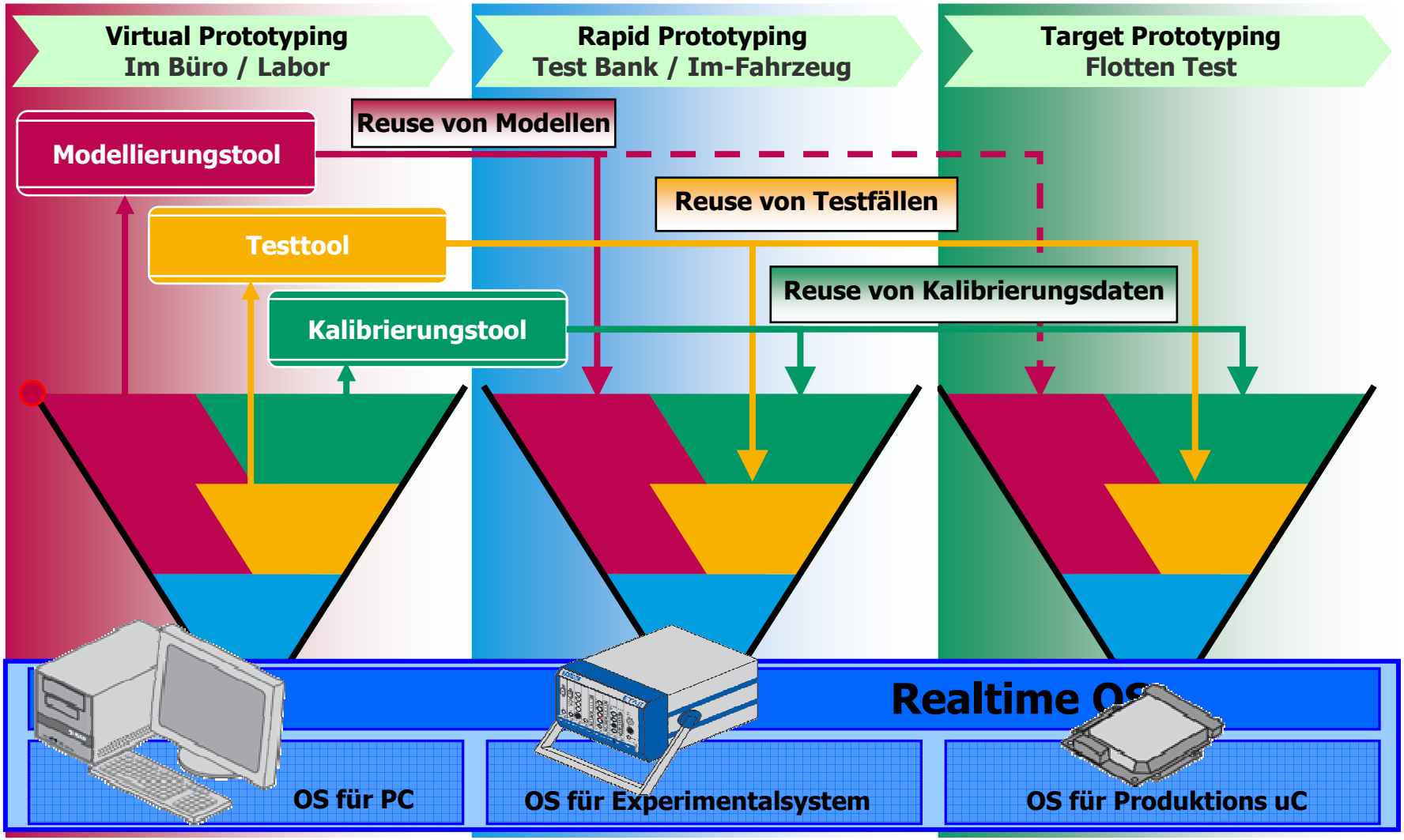
Integrierte und virtuelle Entwicklung sichert Qualität



1. Modellierung und Simulation
   Virtual Prototyping auf PC
   Virtuelle Kalibrierung auf PC
2. Rapid Prototyping und Vor-kalibrierung im realen Fahrzeug
3. Implementierung der Funktion im Steuergerät
4. Steuergeräte-Funktionstest mit Laborfahrzeug
   Vorkalibrierung in einer virtuellen Umgebung
5. Funktionsvalidierung und Parameterabstimmung im realen Fahrzeug

Effizienz im Entwicklungsprozess

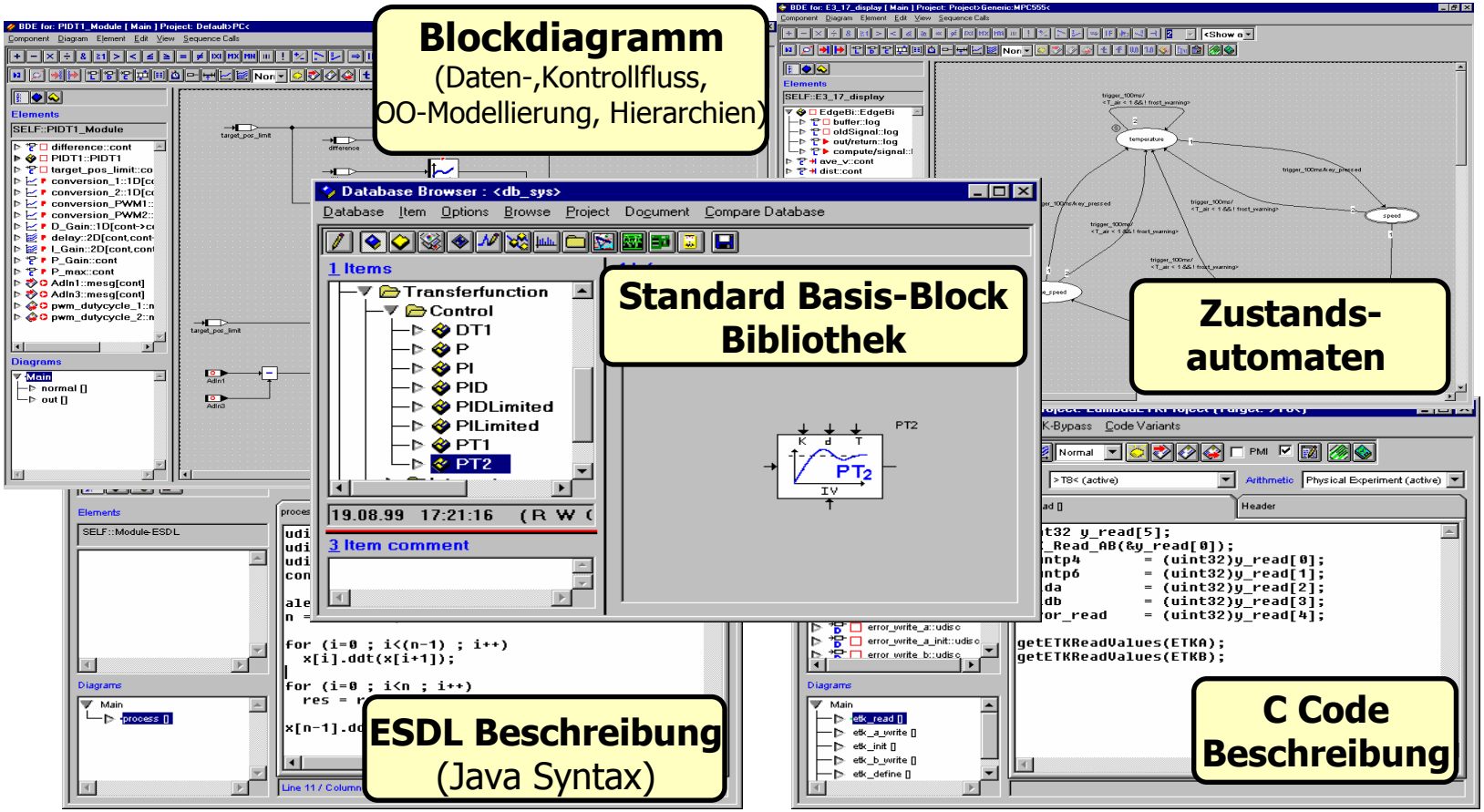
Wiederverwendung vom virtuellen bis zum realen Prototyp



Automotive Software Modellierungswerkzeuge

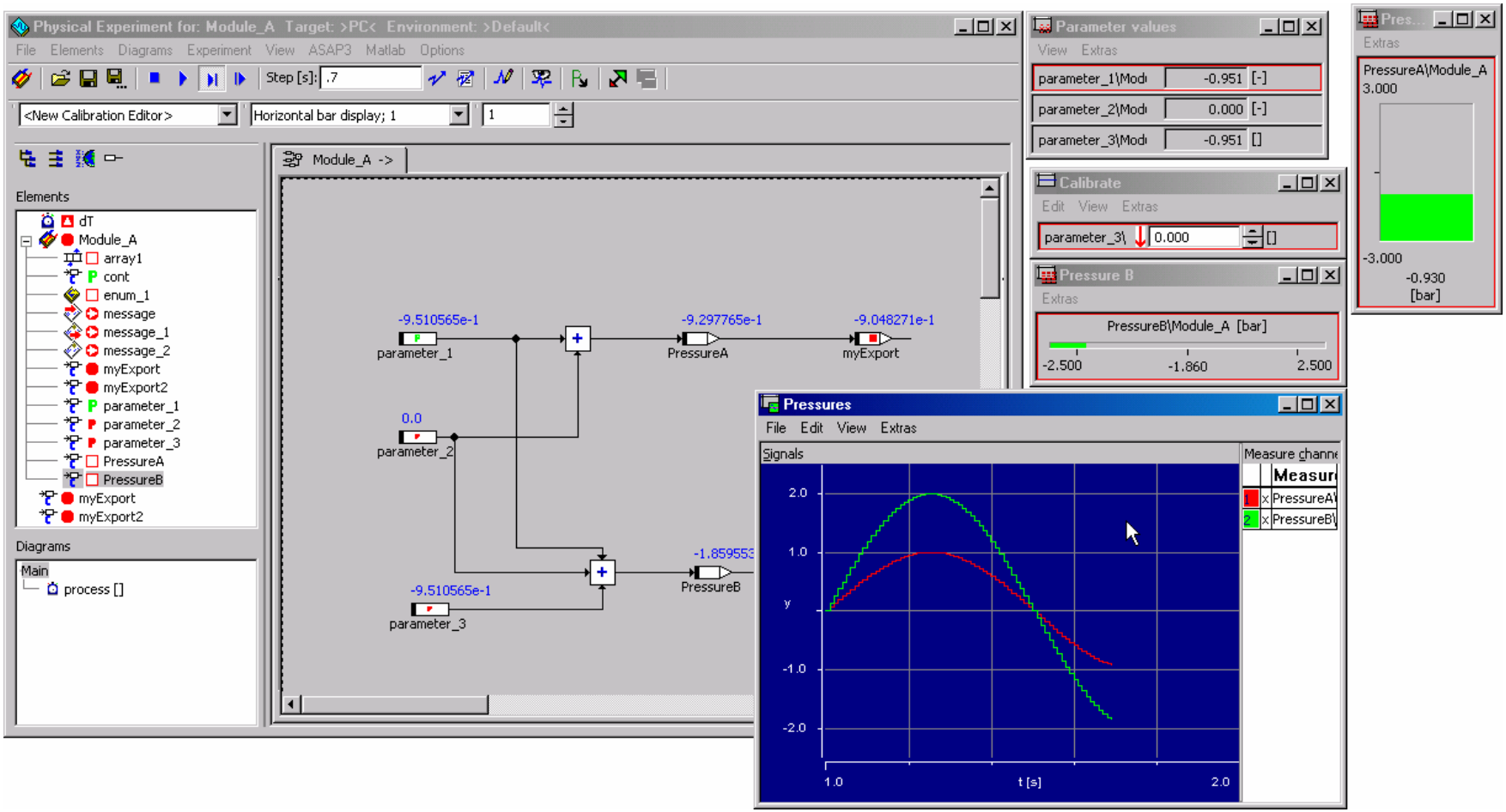
Überblick – Verfügbare Modellierungselemente

Steuergeräte Softwareentwicklungsumgebung



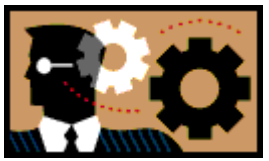
Automotive Software Modellierungswerkzeuge

Experimentalumgebung - PC und Experimentaltarget



Wozu Automatische Codeerzeugung?

Programmier Fehler im Hand Code



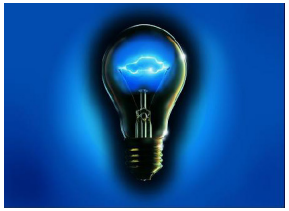
Programmierfehler wie z.B.

```
if (v_wheel=v_max) {/* should be (v_wheel== v_max)*/
        do_something(v_wheel);
   }


y = a/(x-z);   /* (x-z) could turn 0 unexpectedly */


int32 dosomething(a uint32,b uint16) {
        uint16 locVar;
        …              /*locVar should be initialised */
        return locVar;
        }
…
```

Wozu Automatische Codeerzeugung?
Modelltransformation zur Integer Arithmetik



- Ein einfaches Beispiel: $p_E = \frac{(v_A + v_B) * k_C}{k_D}$
  - v_A, v_B sind Geschwindigkeiten [**m/s**]
  - p_E ist der Bremsdruck [**bar**]
  - k_C Formfaktor [**Ns/m**]
  - k_D Konversionsparameter von Kraft zu Druck [**N/bar**]

- Zusätzliche physikalische Information
  - v_A, v_B mit Wertebereich [0..100] **m/s** und Quantisierung q = 1/100 **m/s**
  - p_E → [0..300] **bar**, q = ¼ **bar**
  - k_C → [-20..20] **Ns/m**, q = 0,3 **Ns/m**
  - k_D → [10..20] **N/bar**, q = 0,1 **N/bar**

ASCET Codeerzeugung

Limitierter quantisierter Integer Code

BDE for: Module_BDE_Example [Main] Project: Project_Example [ANSI-C/Object]

Component Diagram Element Edit View Sequence Calls

```
/* public process []   */

void MODULE_BDE_EXAMPLE_IMPL_process (void)
{
    /* temp. variables */
    sint32 _t1sint32;

    {
        _t1sint32
            = (sint32)(A + B)   *   C   *   (sint32)3   /   (sint32)25;
        _t1sint32 = ((D == (sint16)0) ? _t1sint32 : _t1sint32   /   D);
        /* process: sequence call #1   */
        /* assignment to E: min=0, max=1200, hex=4phys+0, limit=(maxBitLength: true, assign: true), zero incl.=true   */
        E = ((_t1sint32 >= (sint32)0) ? ((_t1sint32 <= (sint32)1200) ? (uint16)_t1sint32 : (uint16)1200) : (uint16)0);

    }
}
```

Diagrams

- Main
  - process []

Elements | Data | Implementation | CaliberRM

Name	Type	Impl. Type	Impl. Min	Impl. Max	Q	Formula	Limit to ma bit length	Limit Assignmer	Zero not incl.	Min
D	cont	uint8	10	20	0	F_01	Auto		No	1.0
E	mesg[cont]	uint16	0	1200	0	F_025	Auto	Yes	No	0.0

ASCET Codeerzeugung
Optimierung

- **Benutze 2er Potenz Näherungswerte für Literale**
  - schnellere Berechnung, auf Kosten der Genauigkeit -

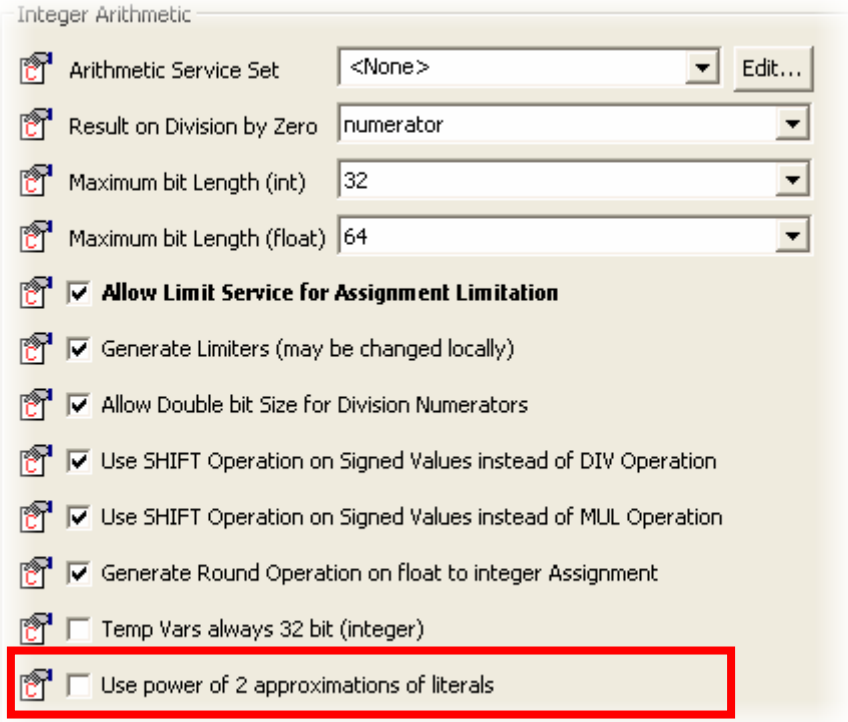
Z.B.:

Näherungswert für Literal **0.866** ist

28377/217 ~ 0,86599731**

anstelle von

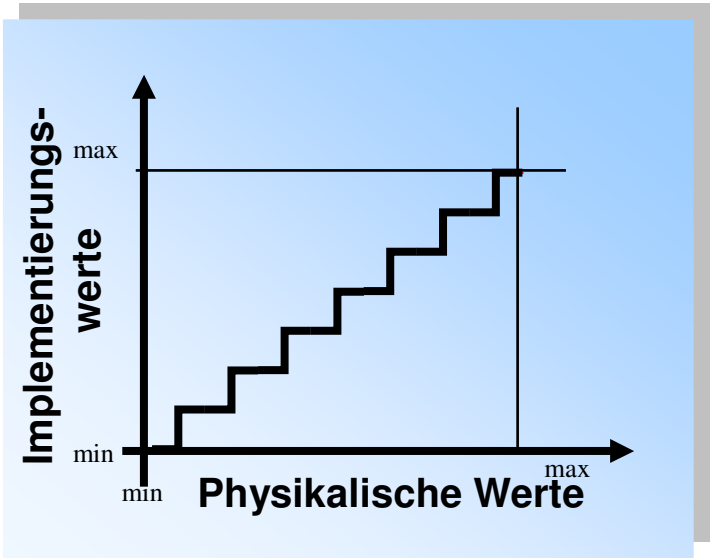
433/500 = 0.866



ASCET Codeerzeugung
Codegenerator Eigenschaften

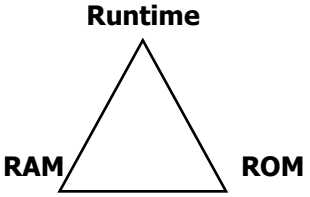
- Automatische Transformation geschieht unter folgenden Randbedingungen
  - max. verfügbare Bitlänge des Ziel-prozessors
  - Implementierungstypen
  - Wertebereiche
  - Konversionsformeln

  mit **automatischer**
  - Behandlung von Überläufen
  - Erzeugung von Quantisierungs- und Korrekturfaktoren (Konversionsfaktoren werden kombiniert)
  - Erzeugung von limitiertem Code
  - Vermeidung von Division durch "0" / "geschützte" Division durch "0"



Codeeffizienz

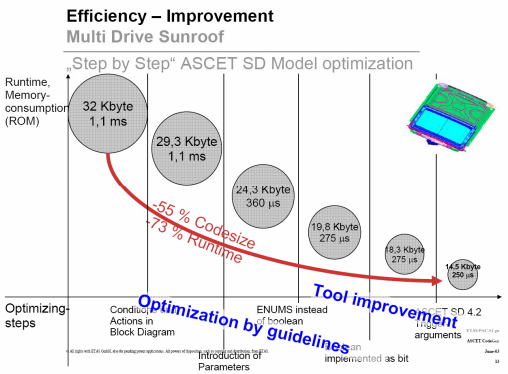
Kundenaussagen



- BMW AG

  "Because of modelling guidelines the demand for runtime, ROM and flash is almost identical for automatically generated C-Code and C-Code which is coded by hand."

  Ref: BMW/Department Head Electronic Development, Mr. J. Hauser – ETAS Competence Exchange Symposium 2004



- Robert Bosch GmbH – GS (Gasoline System)

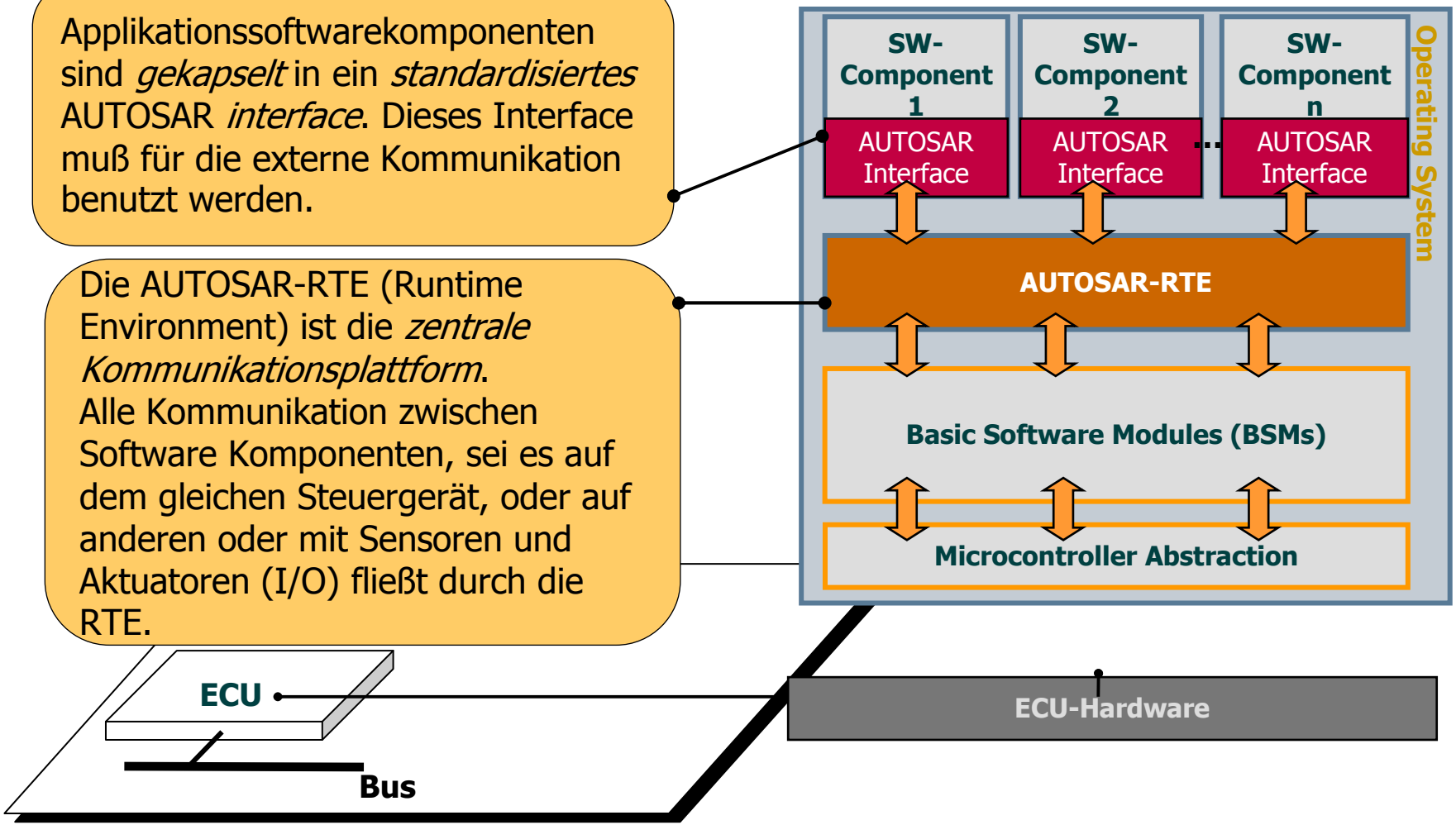
  „For complex models, the deployment of ASCET V5.1 may yield time savings in the area of up to 30 percent.„

  Ref: BOSCH-GS/, Mr. Nicolaou – ETAS RealTimes

AUTOSAR : Ein neuer Standard
Steuergerätesoftwarearchitektur, in Zukunft standardisiert

Applikationssoftwarekomponenten sind *gekapselt* in ein *standardisiertes* AUTOSAR *interface*. Dieses Interface muß für die externe Kommunikation benutzt werden.

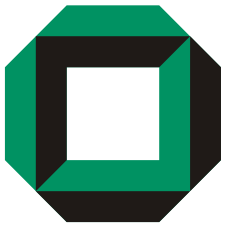
Die AUTOSAR-RTE (Runtime Environment) ist die *zentrale Kommunikationsplattform*.
Alle Kommunikation zwischen Software Komponenten, sei es auf dem gleichen Steuergerät, oder auf anderen oder mit Sensoren und Aktuatoren (I/O) fließt durch die RTE.



Zusammenfassung – Modellbasierte Entwicklung ist effizient und liefert effiziente Software

Mehr als **60+ Millionen** Fahrzeuge, deren Steuergerätesoftware modellbasiert mit ASCET entwickelt wurde, sprechen für:

- Code **Qualität** die **höher** ist als Handcodierung
- Eine **sehr effiziente** Codeerzeugung durch ASCET
- Leichte Konfigurierbarkeit der Eigenschaften des automatisch erzeugten Codes



Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

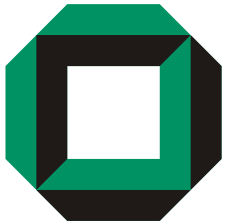
Software Components and Software Architecture

Software Design on its Road to an Engineering Discipline

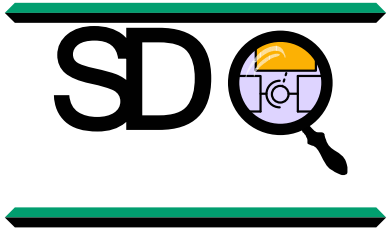
Prof. Dr. Ralf Reussner

Vortragender: Steffen Becker

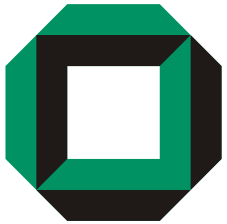
Abteilungsleiter Software Engineering, FZI



Overview

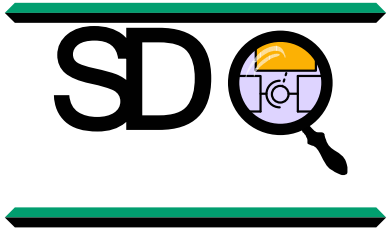


- Is Software Engineering an Engineering Discipline?
- Role of Software Components
- Palladio Component Model
  - Parametric Contracts
  - Prediction of Quality Properties
- Example
- Conclusions



Elements of an Engineering Discipline

[Shaw&Garlan95]



Craft

- Customer and Developer often the same person
- Talent ad Experience instead of Understanding

Industrial Manu-facturing

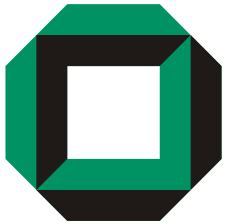
- Division of Labor
- Education of Specialists
- Use of third party tools

Engineering

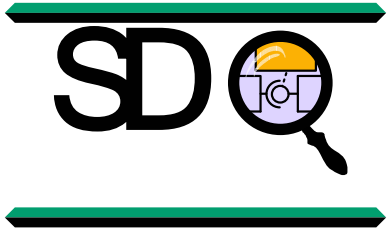
- Goal-driven optimisation of
  - Products
  - Processes

  requires
- Understanding of the effects of design decisions and changes

→ Theories on products and processes



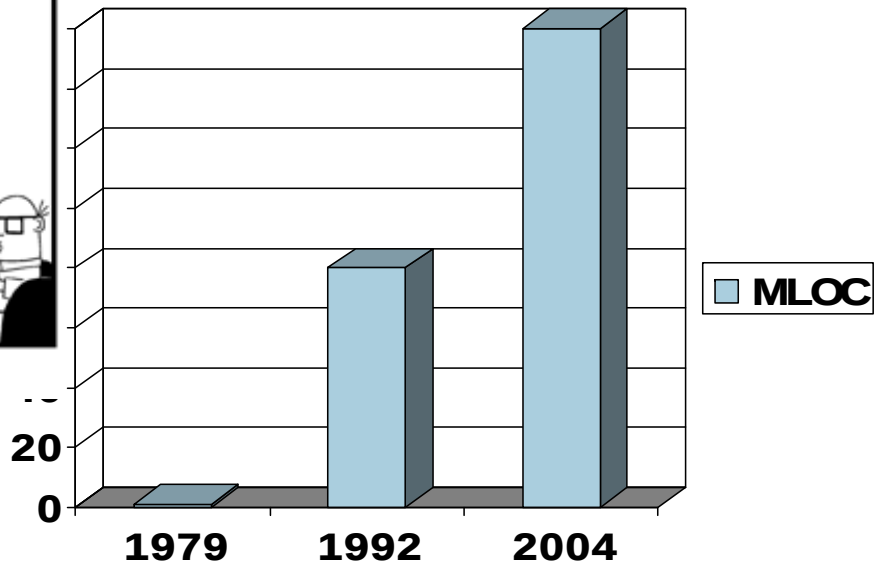
No Progress in SE?



- The same problems since 1968 (first Software Engineering Conference)
- „the problem of achieving sufficient reliability in the data systems...“
- „the difficulties of meeting schedules and specifications on large software projects“
- „the highly controversial question of whether software should be priced separately from hardware“

Where stands „Software Engineering“ as an Engineering Discipline?

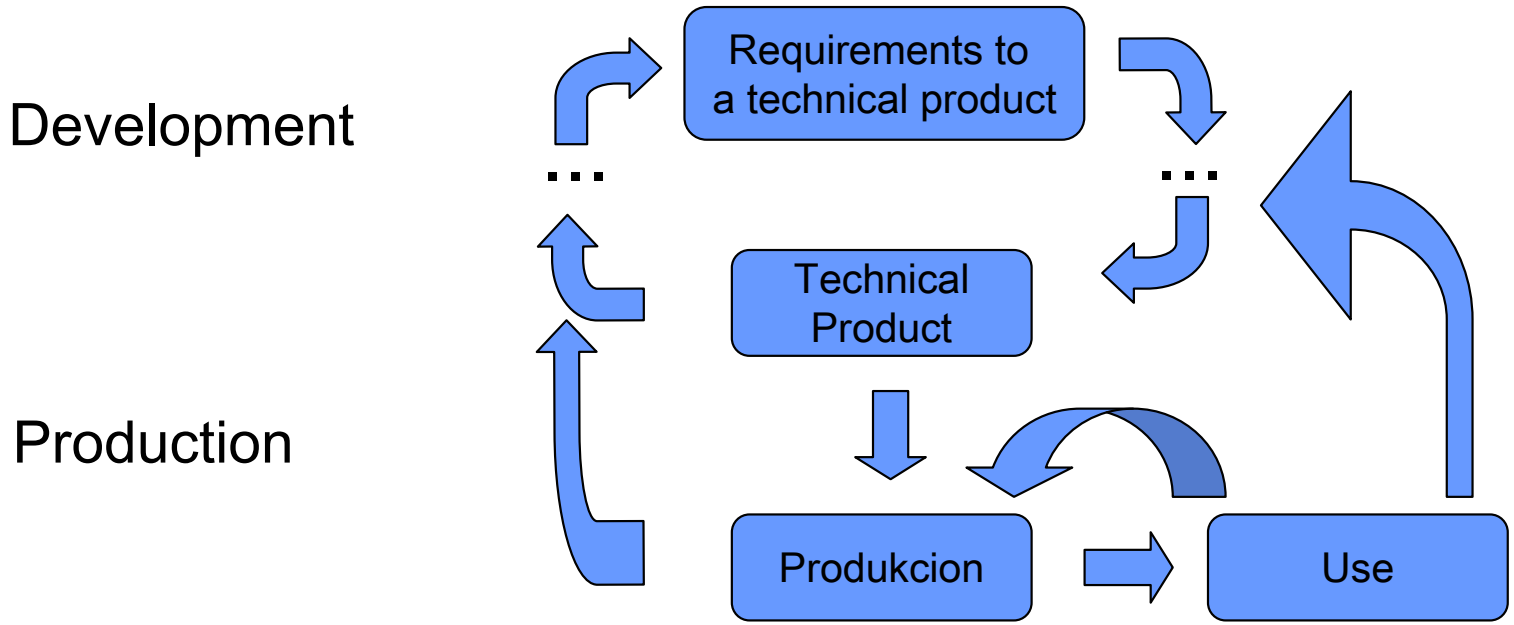
© Scott Adams, Inc./Dist. by UFS, Inc.



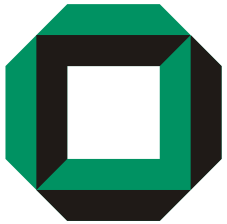
Size of what is considered as “large” software systems

- Progress: the same problems since decades, but for considerably larger and complex systems
- „Planing crisis“ instead of a „Software crisis“ [Glass00]:
  - Budgets and schedules are rarely done by the developer, much more by managers, sales persons and customers

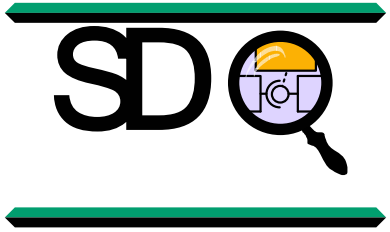
Development and Production



- Technical Production: well understood, planable, repeatable
- Problems of Software Engineering are problems in development, not production



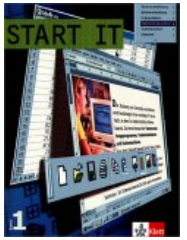
Software Engineering: industrial manufacturing

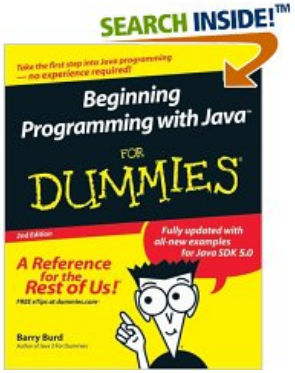


- division of labor
  - Roles
  - Tools (e.g., Versioning)
- Use of specialised tools
- (Spezialised Education)
- Design patterns as a vocabular on proven solutions to recurring problems

Problems

- Lack of Understanding and Professionalism
  - „New Motors in three month.“
  - „Sky scrapers in 5 days.“
  - Why do not we find books like:
    - „Heart Transplantations for Dummies“
    - „Nuclear Weapons in 21 days“
    - „Flying the Airbus: Easy Access!“
  - Sky scrapers as large garden houses
- Counter productive avoidance of up front costs
- Real problem of integrating and using legacy systems





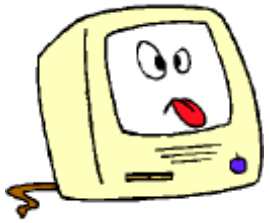
Treatment of Quality Properties today

1. Specification

2. Ignoring

3. Testing

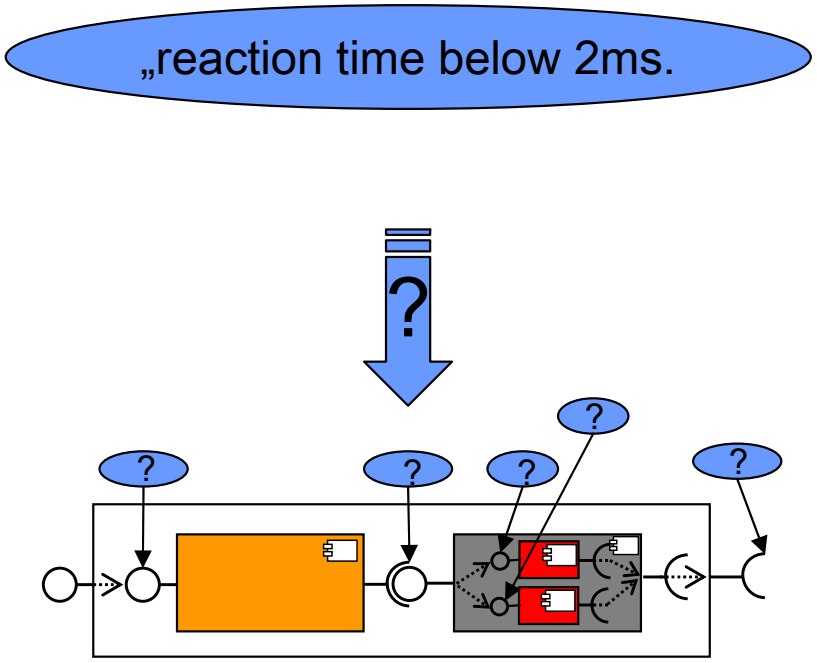
4. Re-Implementing / Re-Designing / Re-Negotiating



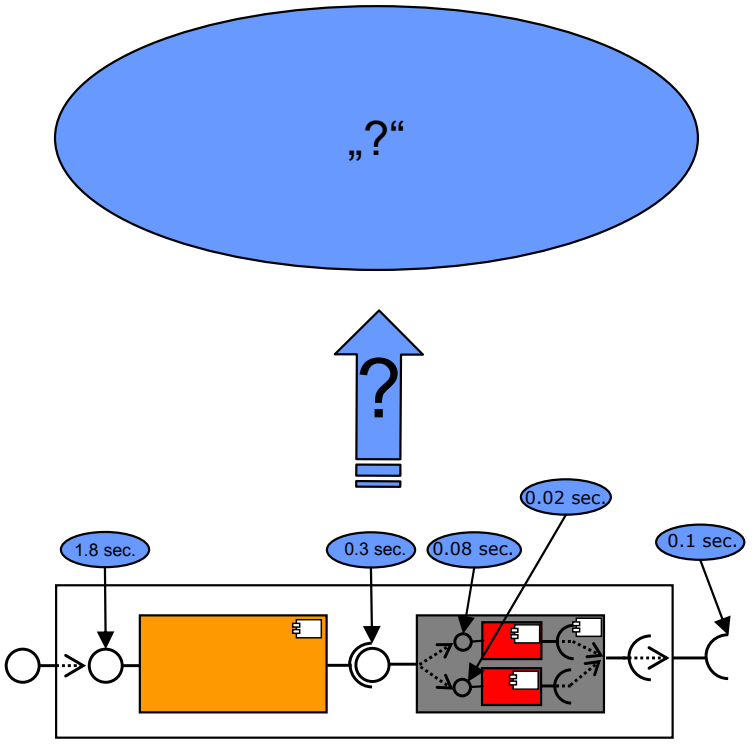
Missing Properties of an Engineering Discipline

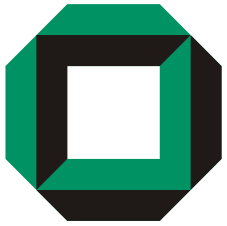
- Systematic Treatment of Quality Attributes

Decomposition of global System-Requirements

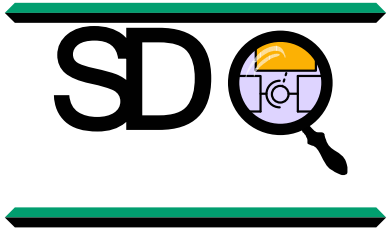


Prediction of global System-Properties



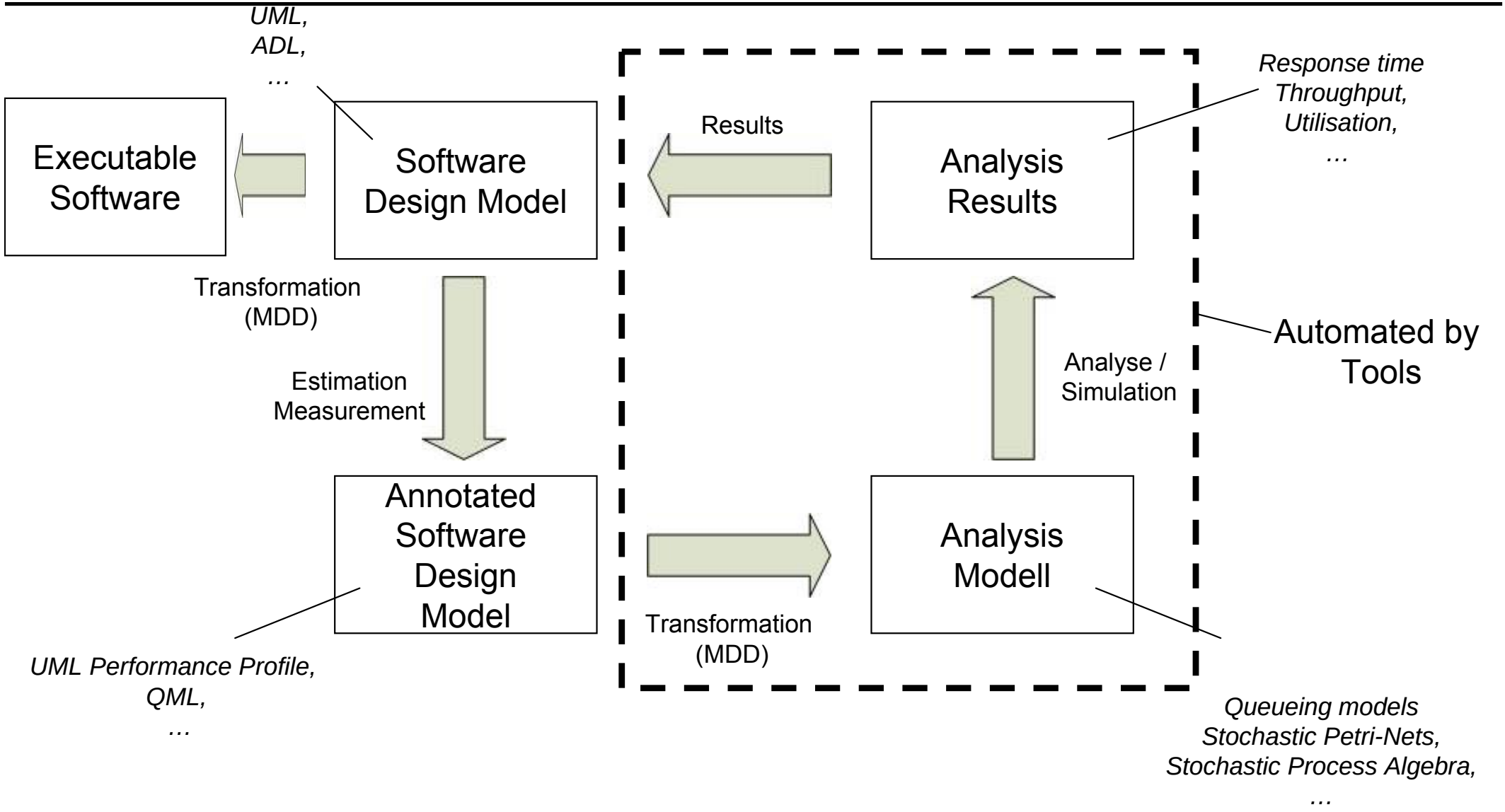


Role of Components in an Engineering Discipline



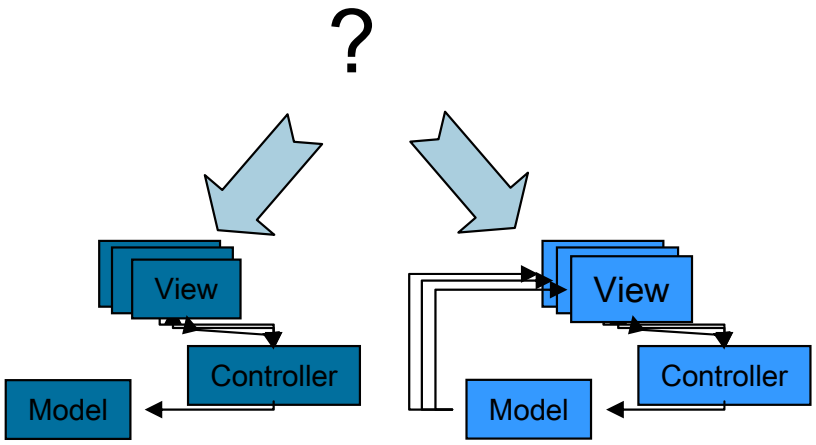
- All engineering disciplines have components.
- Components lower the degrees of freedom during development and, hence, increase the predictability of quality attributes.
- The re-use of components blurrs the boundaries between development of new software, evolution of software and integration of software (which reflects just the reality).
- Re-use of components / composition of systems is isomorphic to re-use / composition of prediction models

Model-based Prediction of Quality



Scenarios for Model-based Quality Prediction

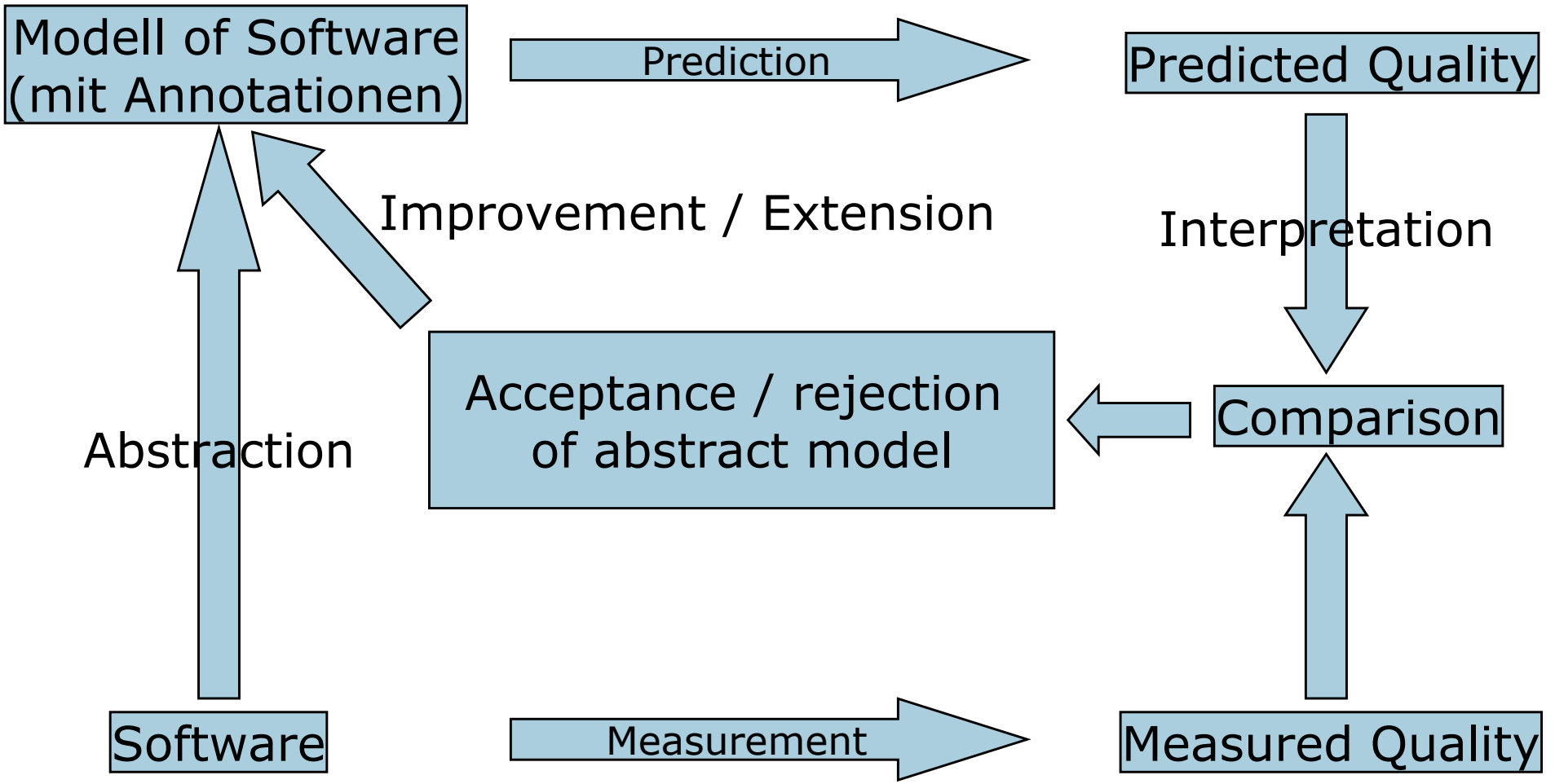
- Evaluation of Design Alternatives



- Dimeniosing of Ressources (“sizing”)

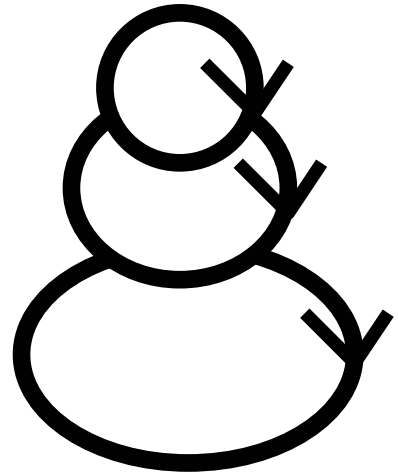
- Change of Usage Profile

Scientific Approach

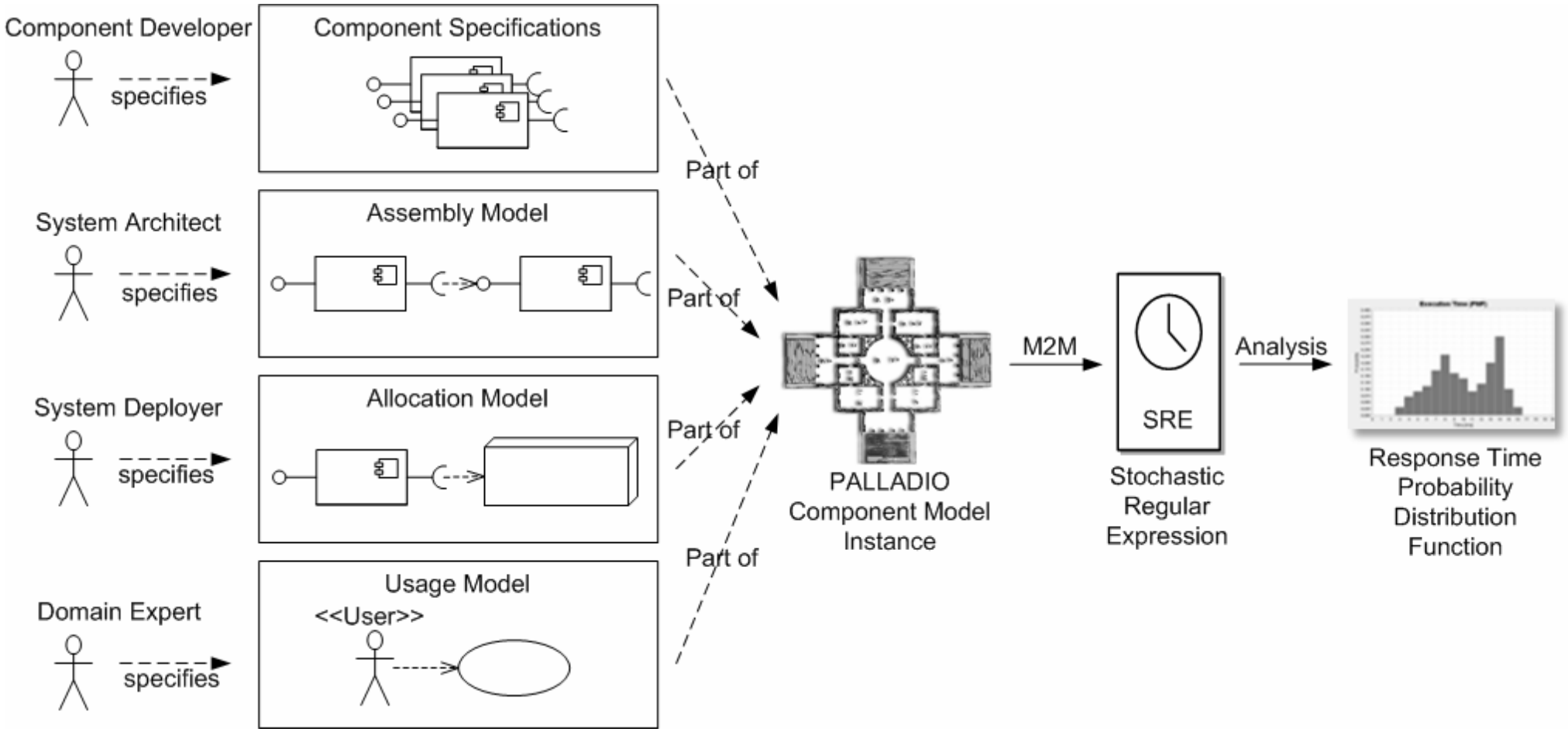


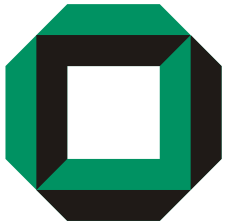
Further Validation and Deployment

- Type 1: Validation of Prediction Model
- Type 2: Validation of Applicability
  - Case Studies and Controlled Experiemts with Students
- Typ 3: Validation of Benefits
  - in comparison to different methods
  - Limitations of the Approach
  - Required prerequisites
  - FZI
  - Industrial Partners

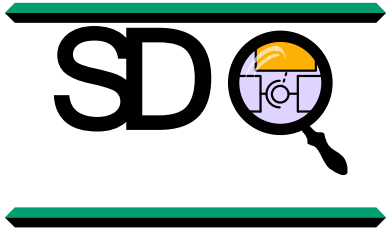


Palladio Component Model





What is a component?



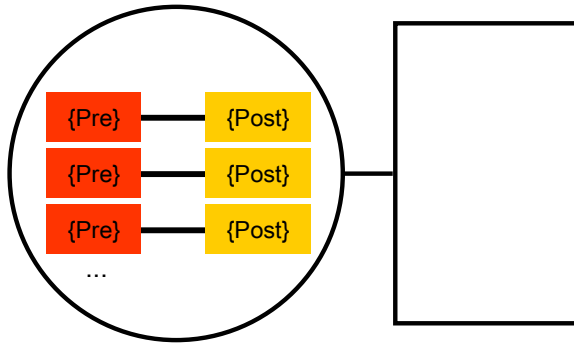
- "A component is a contractually specified building unit of software which can be readily composed or deployed."
  - "readily composed or deployed":
    - without having to understand the interna as a human
    - these are the two main things to be done with components
  - not necessarily "black-box": Information on interna can be available to tools.
- "Components are for composition, much beyond is unclear…" (Clemens Szyperski)

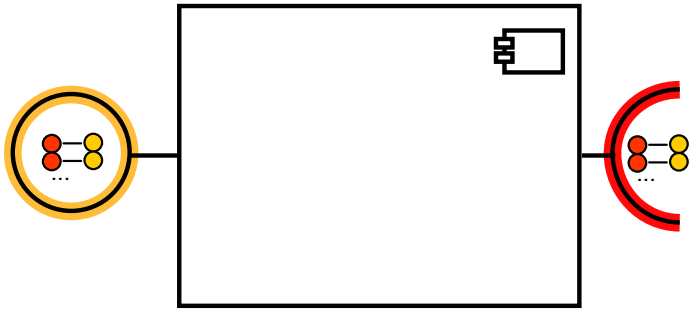
Contractual use of Components

- Design-by-contract (B. Meyer, 1992)
  - " The service supplier guarantees the post-condition, if the client guarantees the precondition of the service."
- Contractual Use of Components

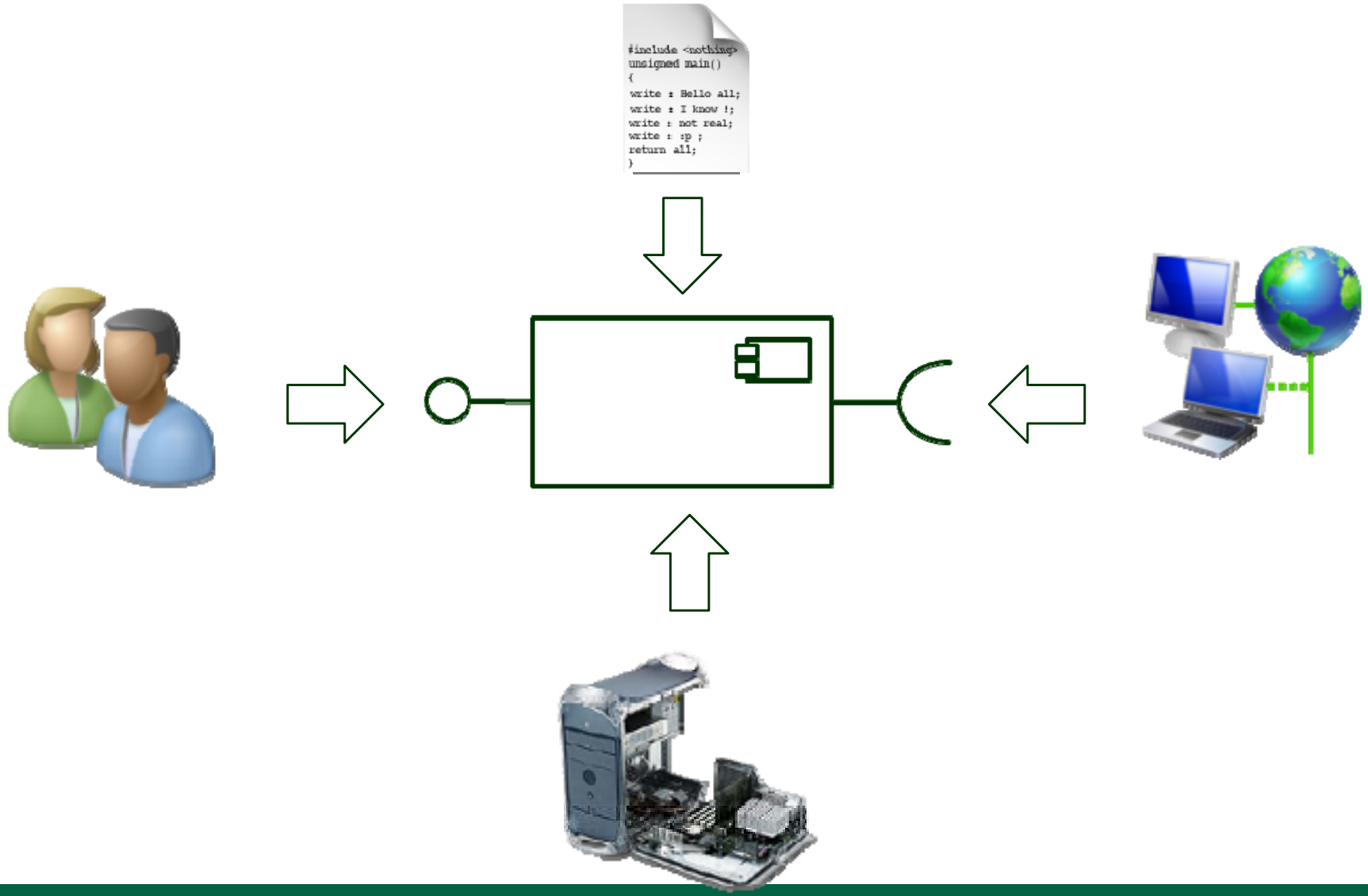
  (at system-(re-) configuration time)

  "The component offers the provided services (as specified in the provides interfaces), if the environment guarantees the required services (as specified in the requires interfaces)."



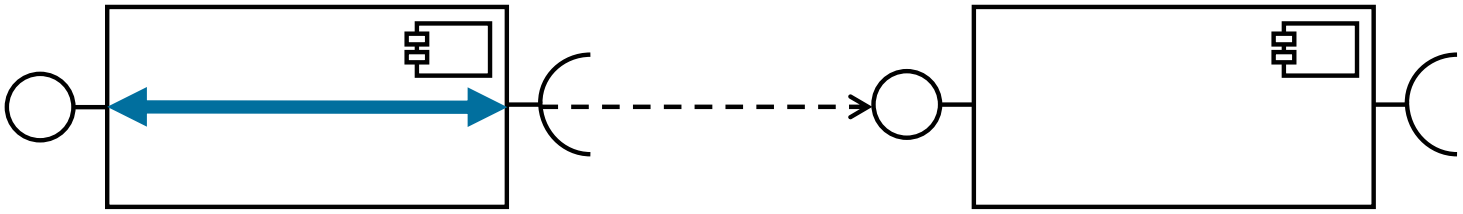


Factors on Component Quality



Parametric Contracts for Components

- Computation of environment-dependent provides-interface
- Computation of use-dependent requires-interface (‘wp-calculus’)
- Fine-grain quality description / adaptation of large-grain components



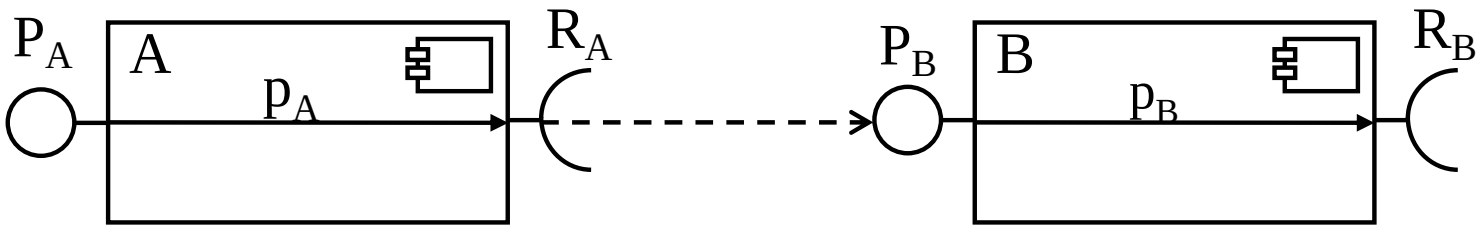
Computation of Parametric Contracts

- Computation of context-dependent provides-interface:

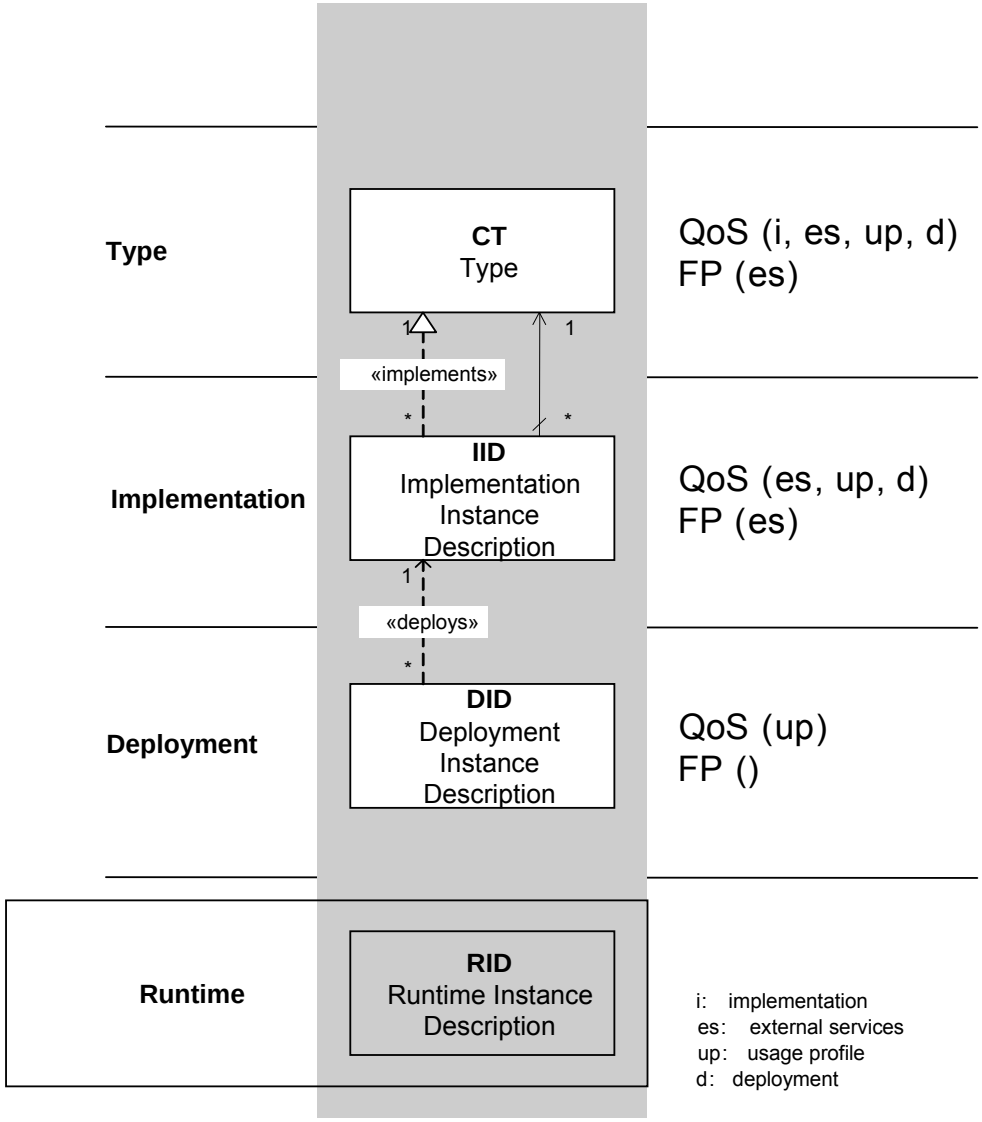
1. $(R_A = p_A(P_A))$
2. $R_{A'} := R_A \cap P_B$
3. $P_{A'} = p_A^{-1}(R_{A'})$

- Computation of context-dependent requires-interface:

1. $P_{B'} := R_A \cap P_B$
2. $R_{B'} = p_B(P_{B'})$

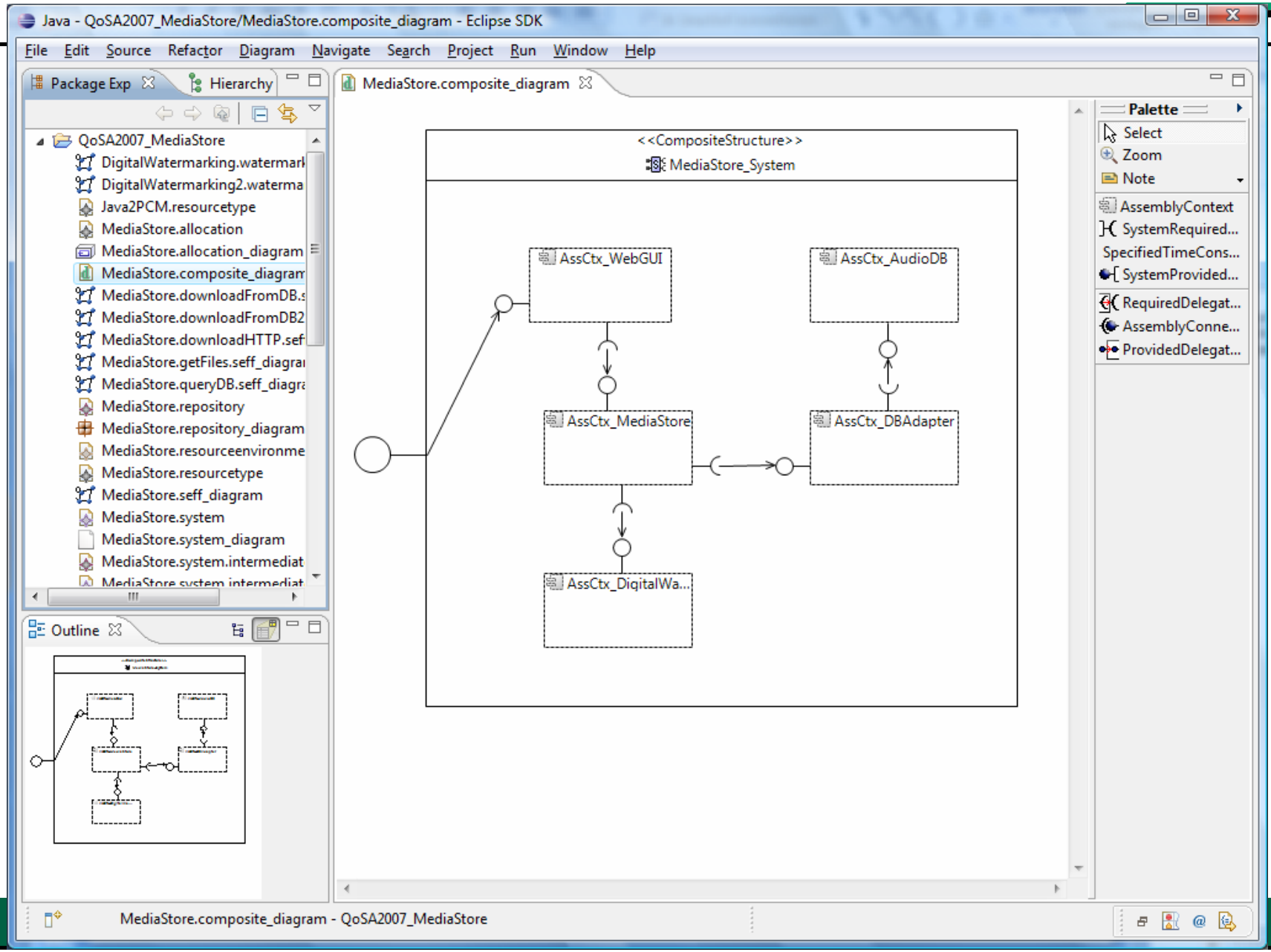


Different Abstraction of Components

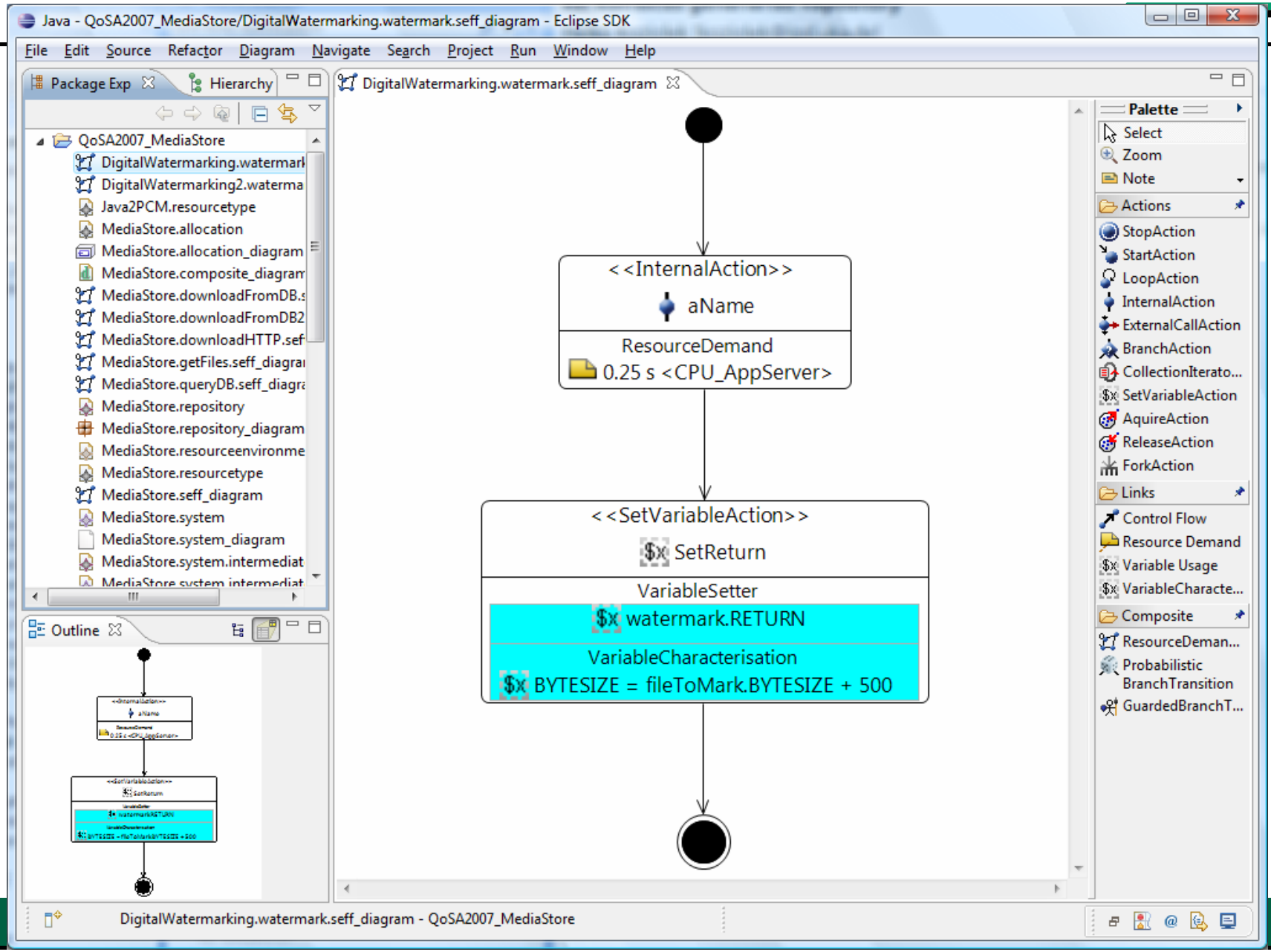


Not considered within the Palladio ComponentModel

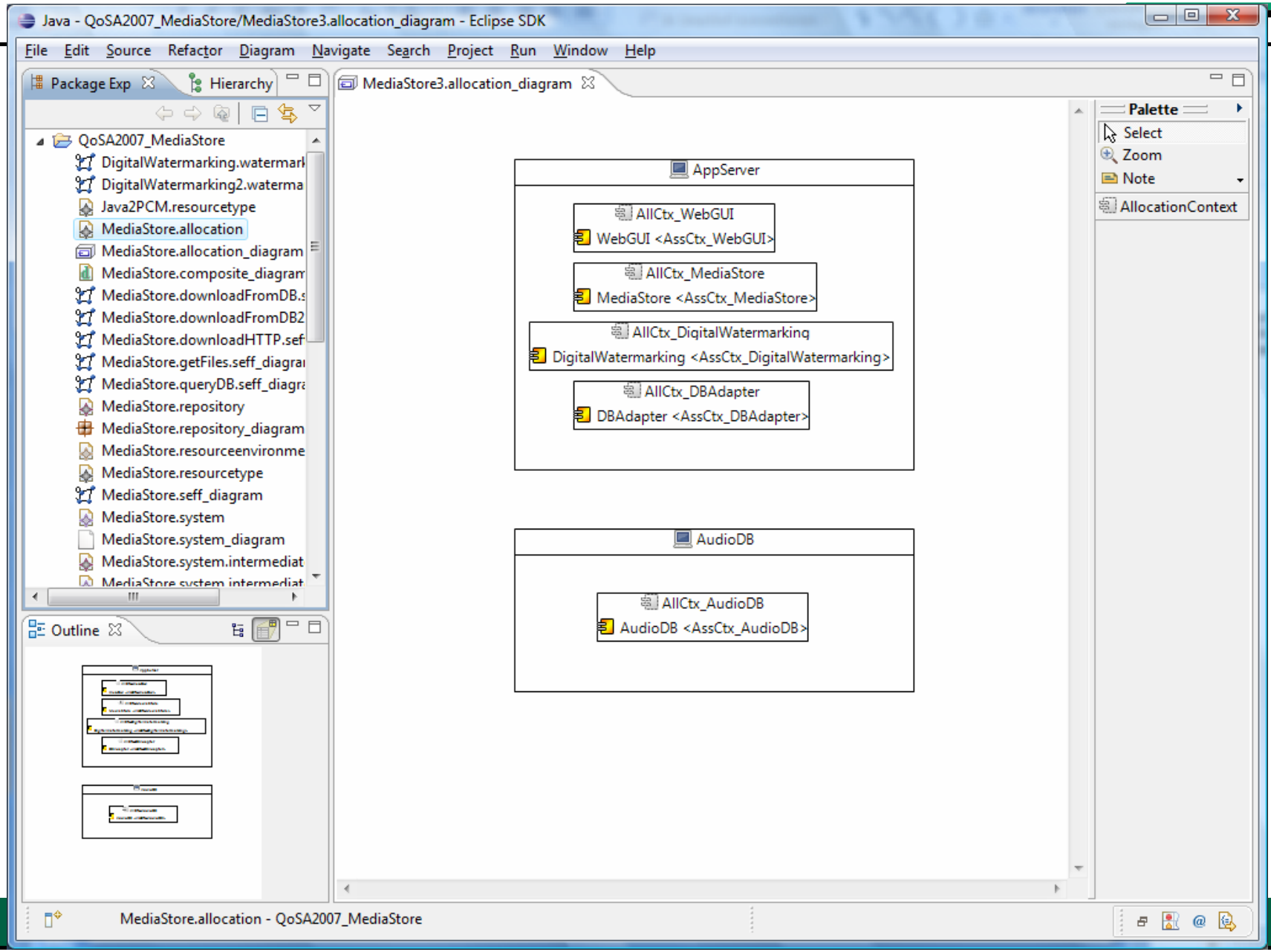
Assembly Model



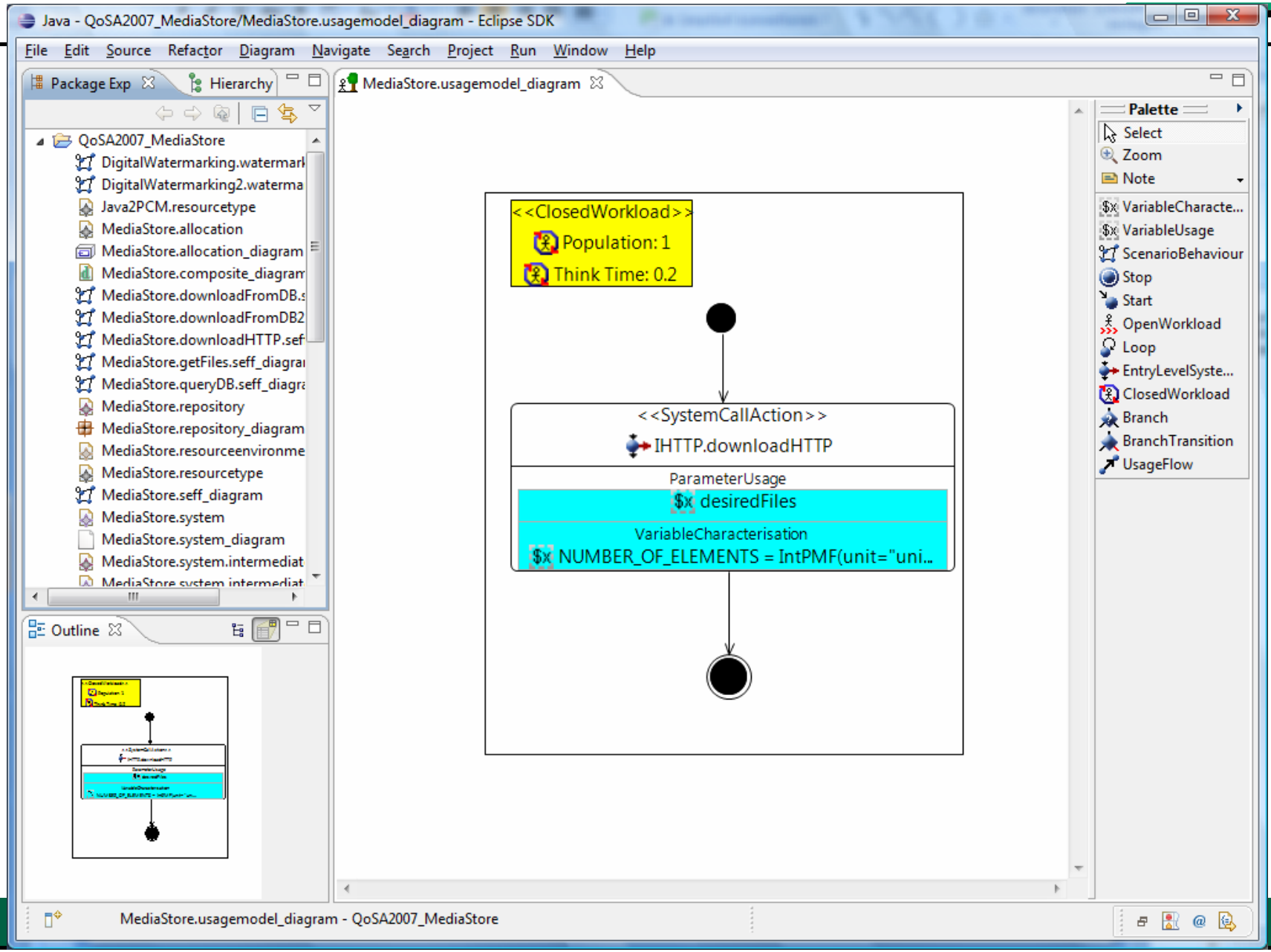
Component Behaviour Specification



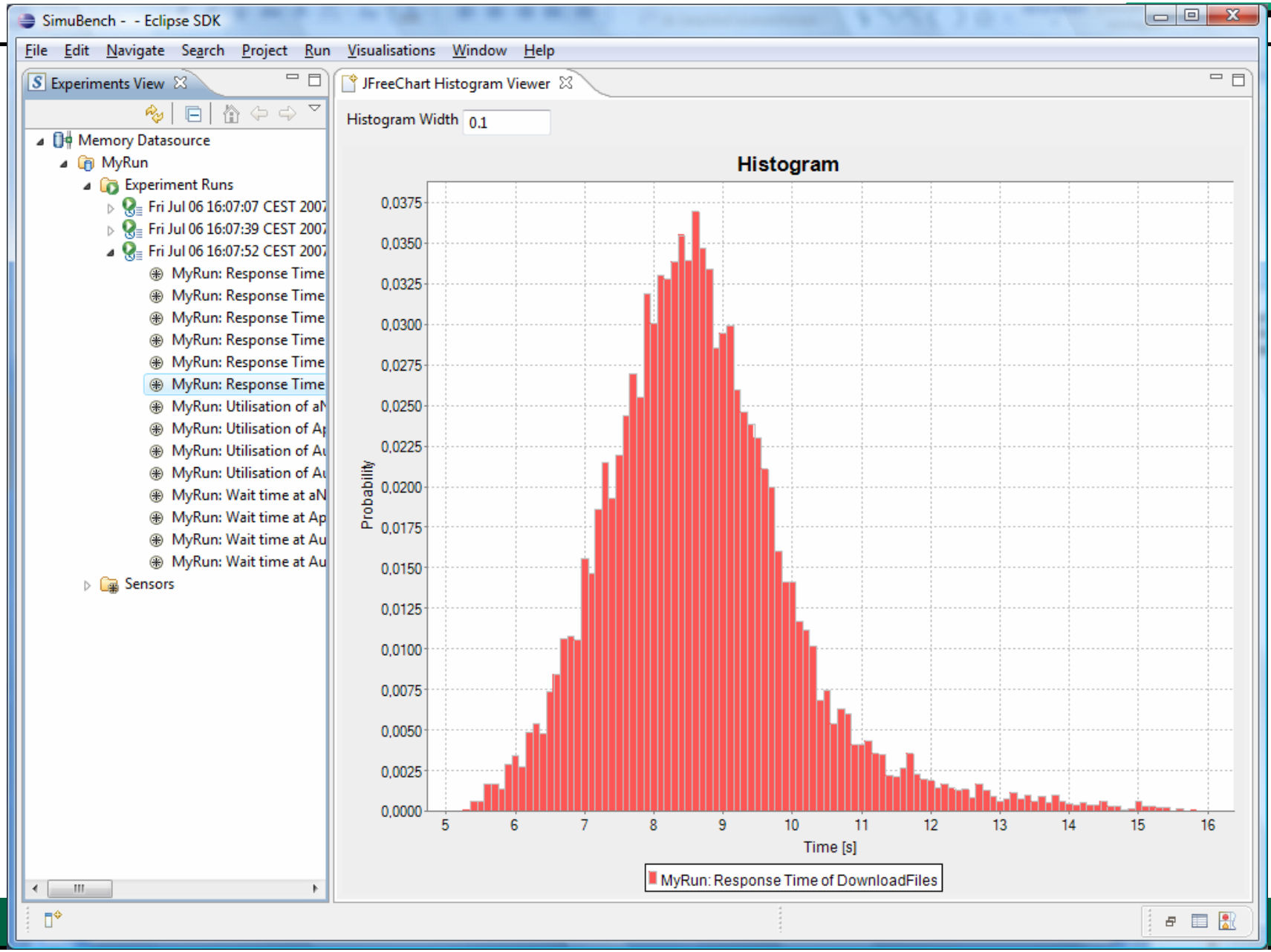
Allocation Model



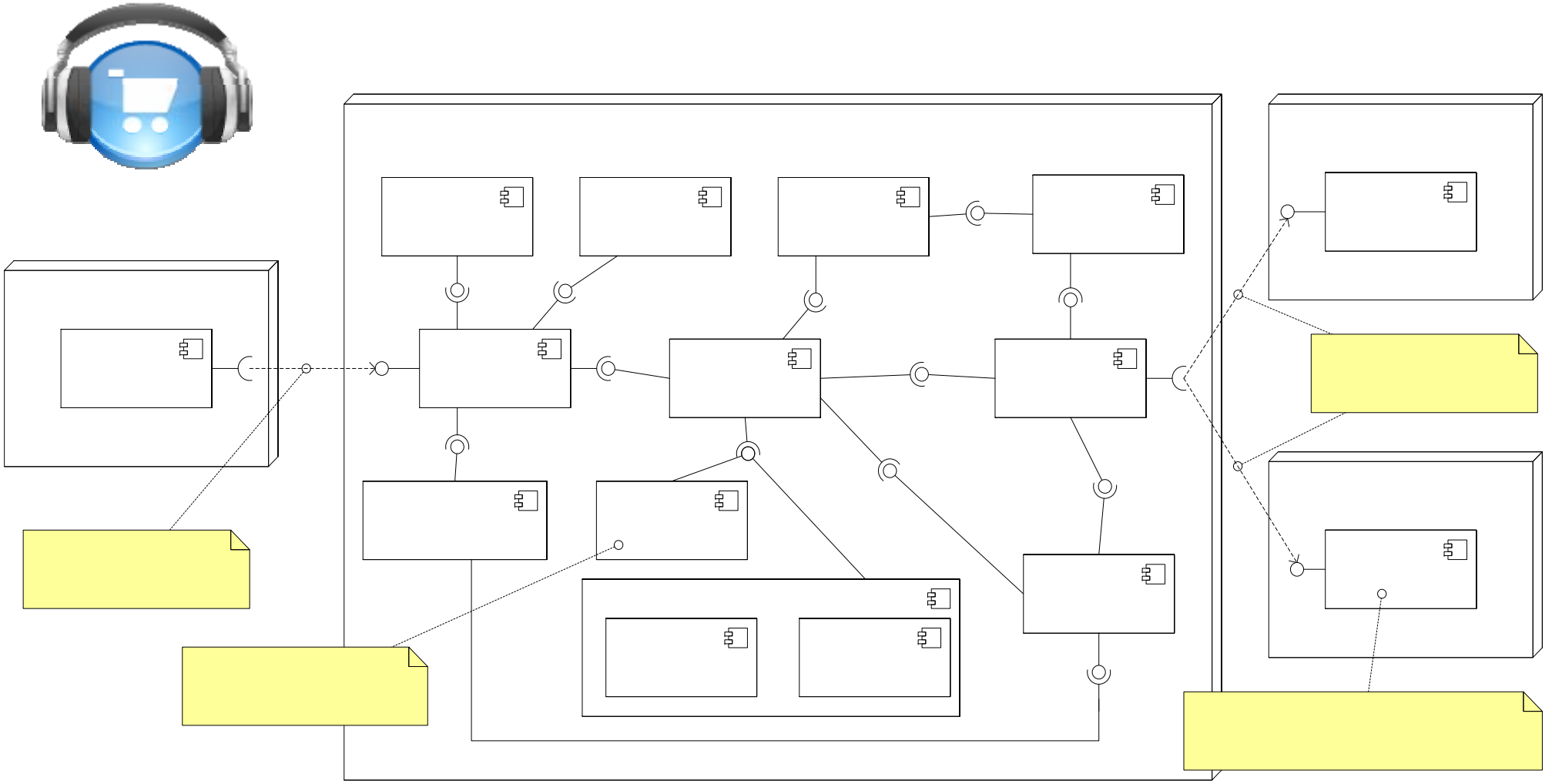
Usage Model


Java - QoSA2007_MediaStore/MediaStore.usagemodel_diagram - Eclipse SDK
File Edit Source Refactor Diagram Navigate Search Project Run Window Help
Package Exp
Hierarchy
QoSA2007_MediaStore
DigitalWatermarking.watermark
DigitalWatermarking2.waterma
Java2PCM.resourcetype
MediaStore.allocation
MediaStore.allocation_diagram
MediaStore.composite_diagram
MediaStore.downloadFromDB.s
MediaStore.downloadFromDB2
MediaStore.downloadHTTP.sef
MediaStore.getFiles.seff_diagra
MediaStore.queryDB.seff_diagra
MediaStore.repository
MediaStore.repository_diagram
MediaStore.resourceenvironme
MediaStore.resourcetype
MediaStore.seff_diagram
MediaStore.system
MediaStore.system_diagram
MediaStore.system.intermediat
Outline
MediaStore.usagemodel_diagram
<<ClosedWorkload>>
Population: 1
Think Time: 0.2
<<SystemCallAction>>
IHTTP.downloadHTTP
ParameterUsage
$x desiredFiles
VariableCharacterisation
$x NUMBER_OF_ELEMENTS = IntPMF(unit="uni...
Palette
Select
Zoom
Note
VariableCharacte...
VariableUsage
ScenarioBehaviour
Stop
Start
OpenWorkload
Loop
EntryLevelSyste...
ClosedWorkload
Branch
BranchTransition
UsageFlow
MediaStore.usagemodel_diagram - QoSA2007_MediaStore

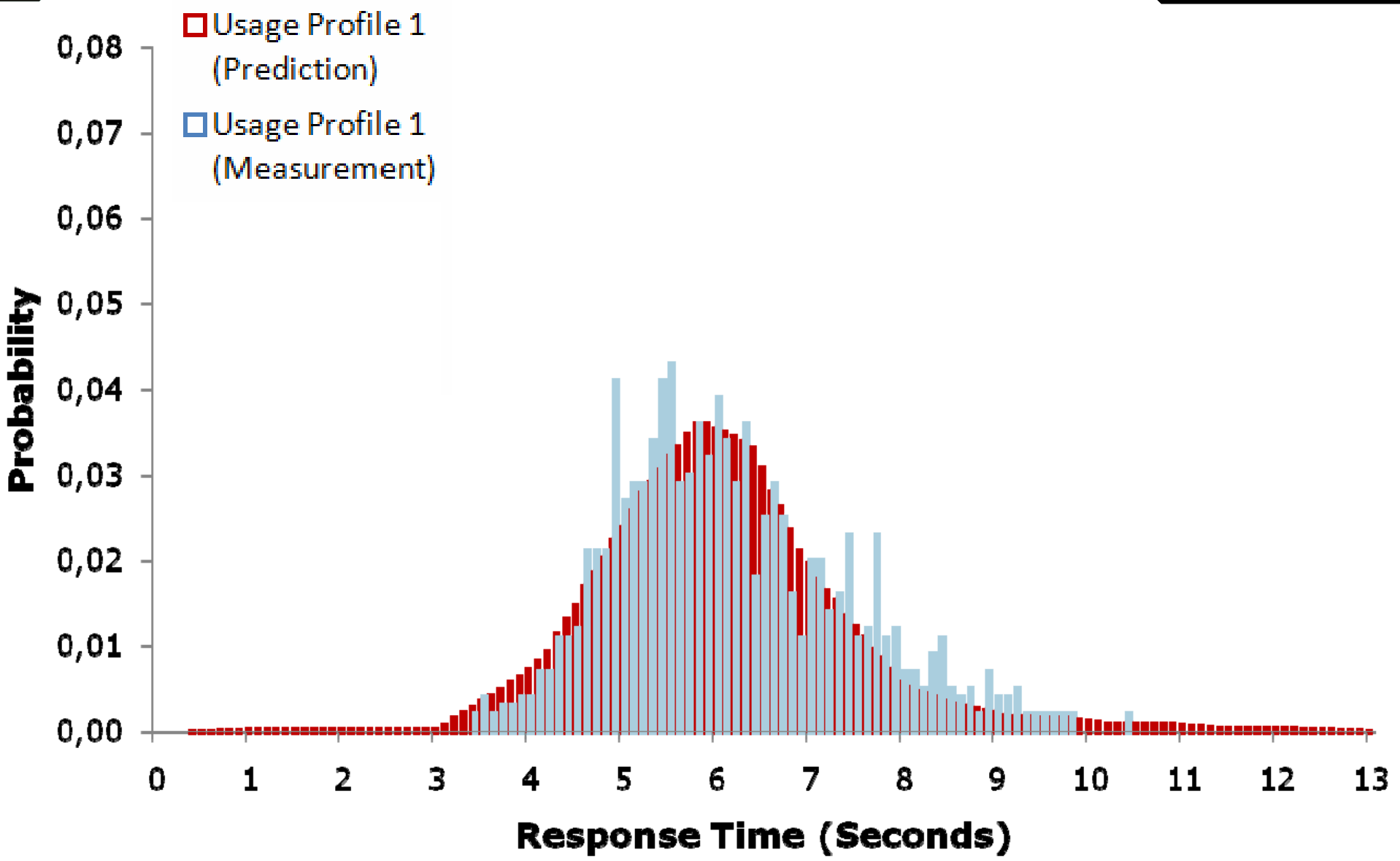
Result: Distribution of Response Time

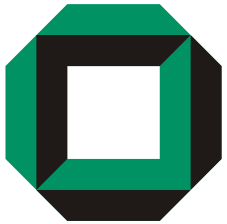


MediaStore - Architecture

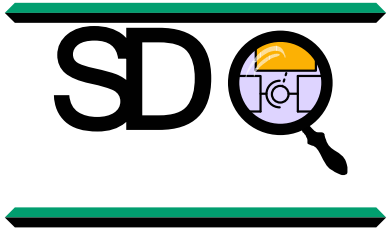


Results


Usage Profile 1 (Prediction)
Usage Profile 1 (Measurement)
Probability
0,08
0,07
0,06
0,05
0,04
0,03
0,02
0,01
0,00
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Response Time (Seconds)



Conclusions



- Prediction and Understanding of the Consequences of Design Decsions is THE central characteristic of an engineering discipline.
- Components and MDD lower the degrees of freedom in implementation
- Creativity is on design-model level
- Quality-driven Design requires prediction models
  - automatically generated from design models
- Definition of design and prediction models follows the scientific process of the natural sciences.
  - No proofs possible, but empirical validations necessary

Software Engineering becomes "architecture-centric".
Code-centration is as meaningful as
"**melting-tin-centration**" of an Electrical Engineer

Klaus Krogmann:
Modell-Reconstruction

Johannes Stammel:
Performanz-Maintain-
ability-Trade-off

Steffen Becker:
Model transformations
and meta modelling

Jens Happe:
Parallelism and analysis model

Michael Kuperberg:
Influence of deployment

Henning Groenda:
Architecture evaluation

Chris Rathfelder:
Architecture evaluation

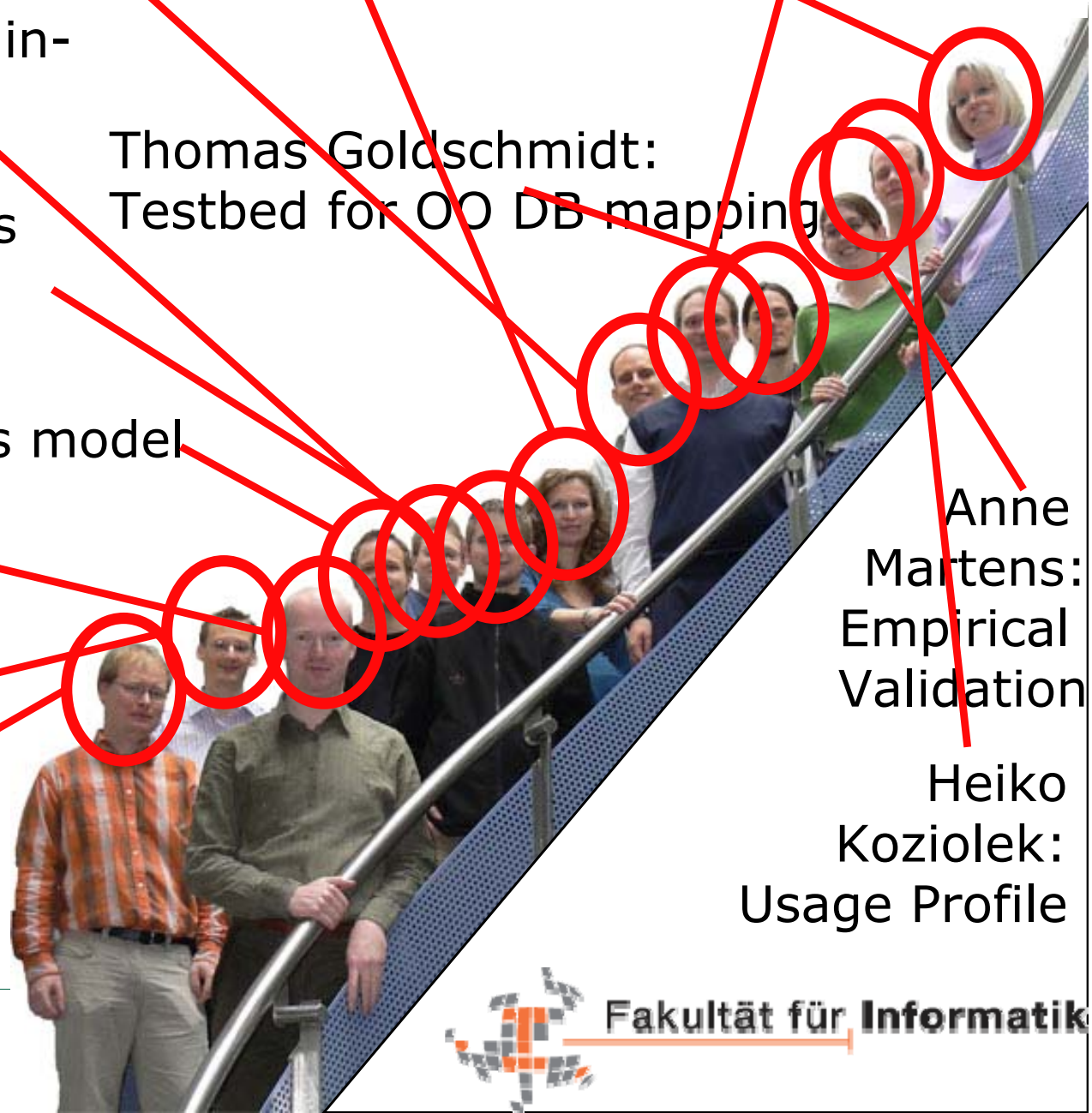
Elena Kienhöfer /
Elke Sauer:
Sekretaries

A Visitor

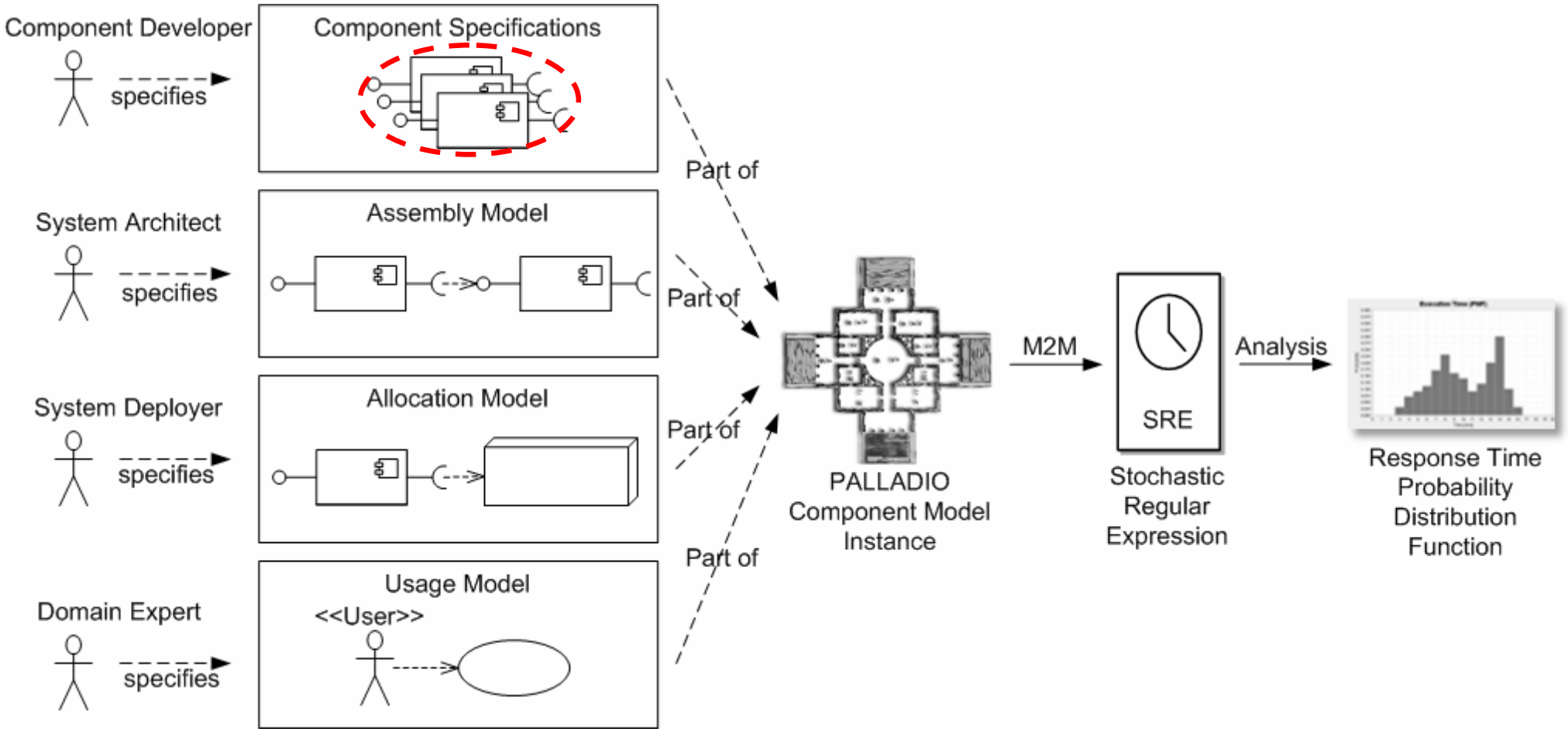
Thomas Goldschmidt:
Testbed for OO DB mapping

Anne
Martens:
Empirical
Validation

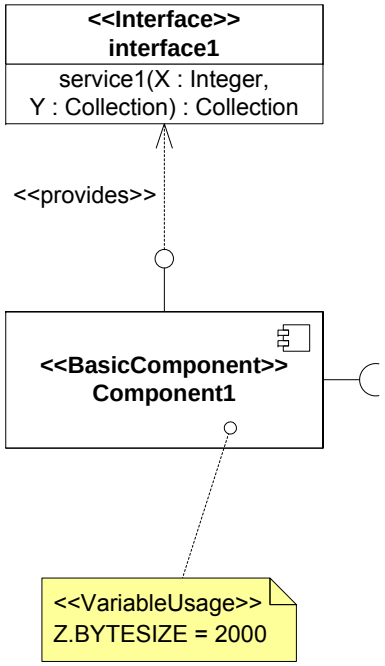
Heiko
Koziolek:
Usage Profile



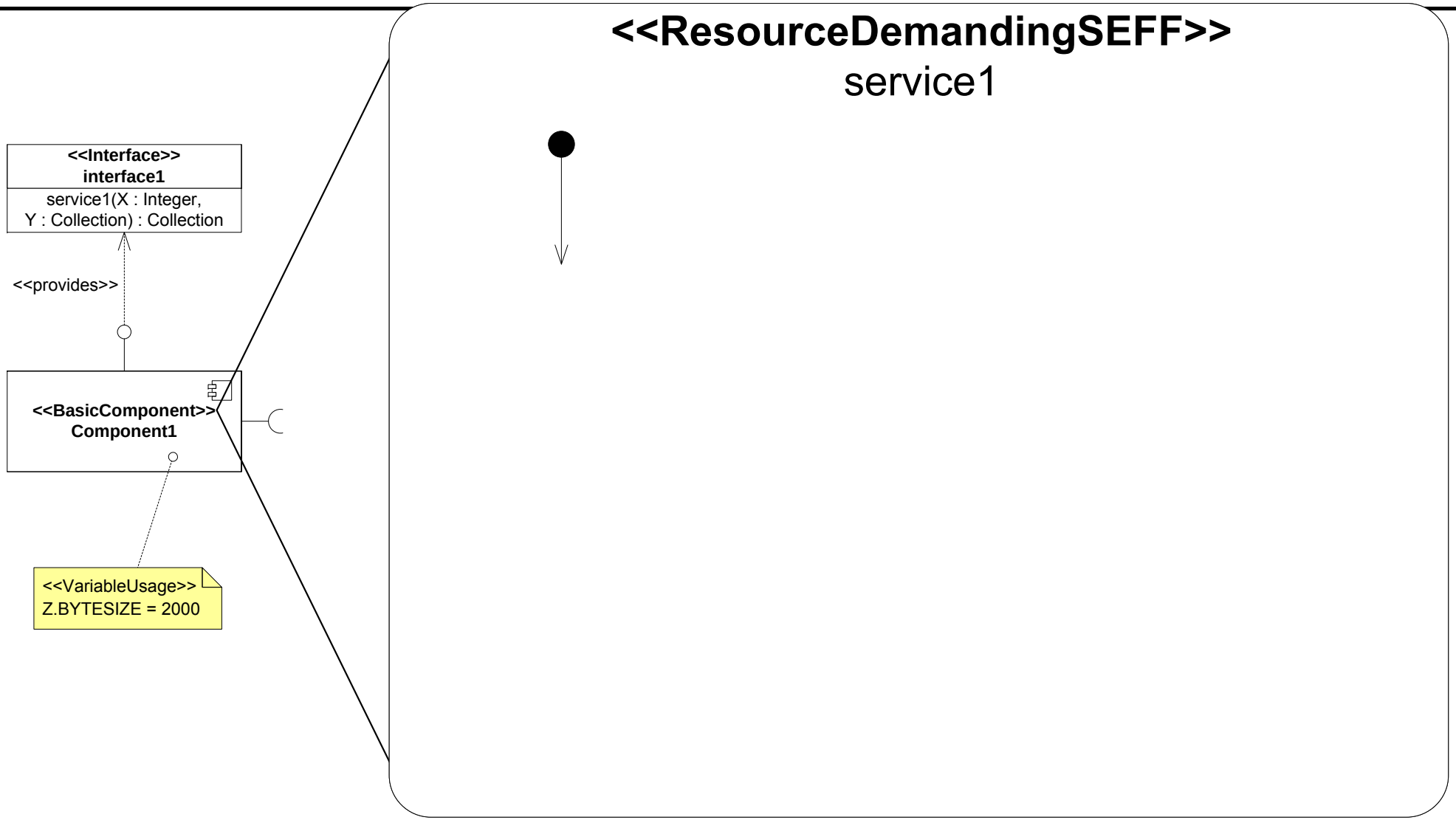
Palladio Component Model



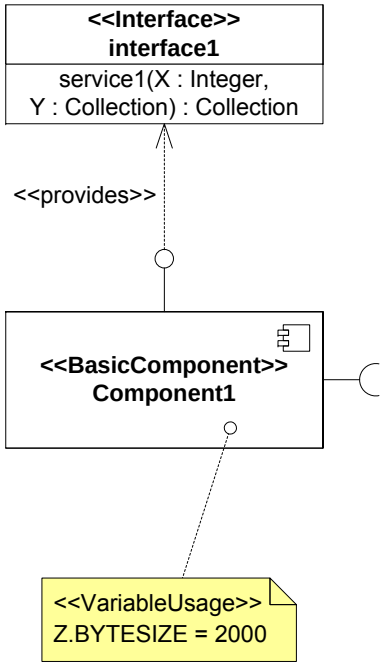
Service Effect Specification

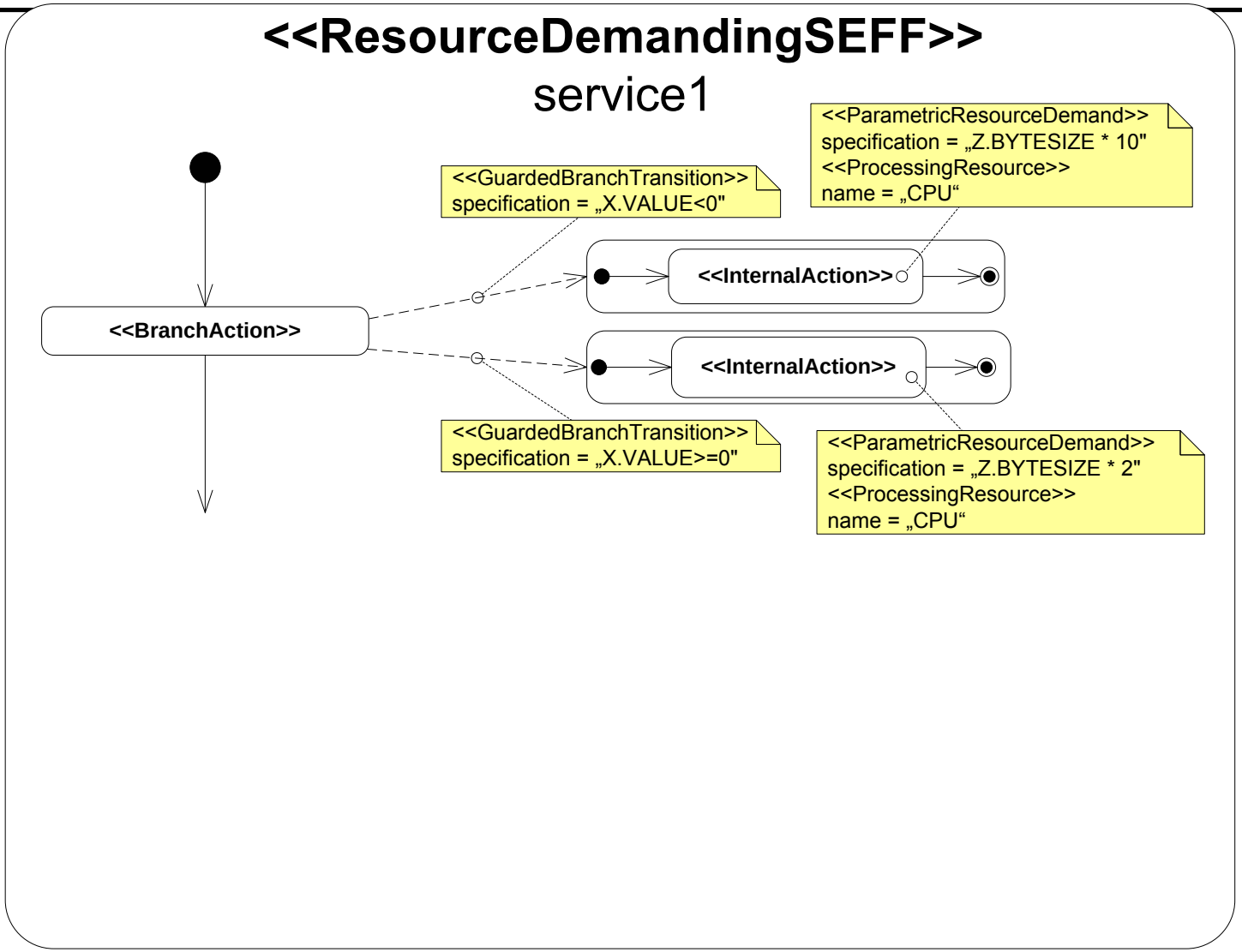


Service Effect Specification

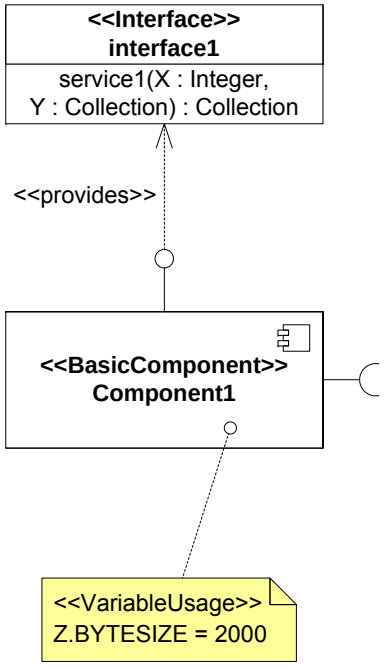


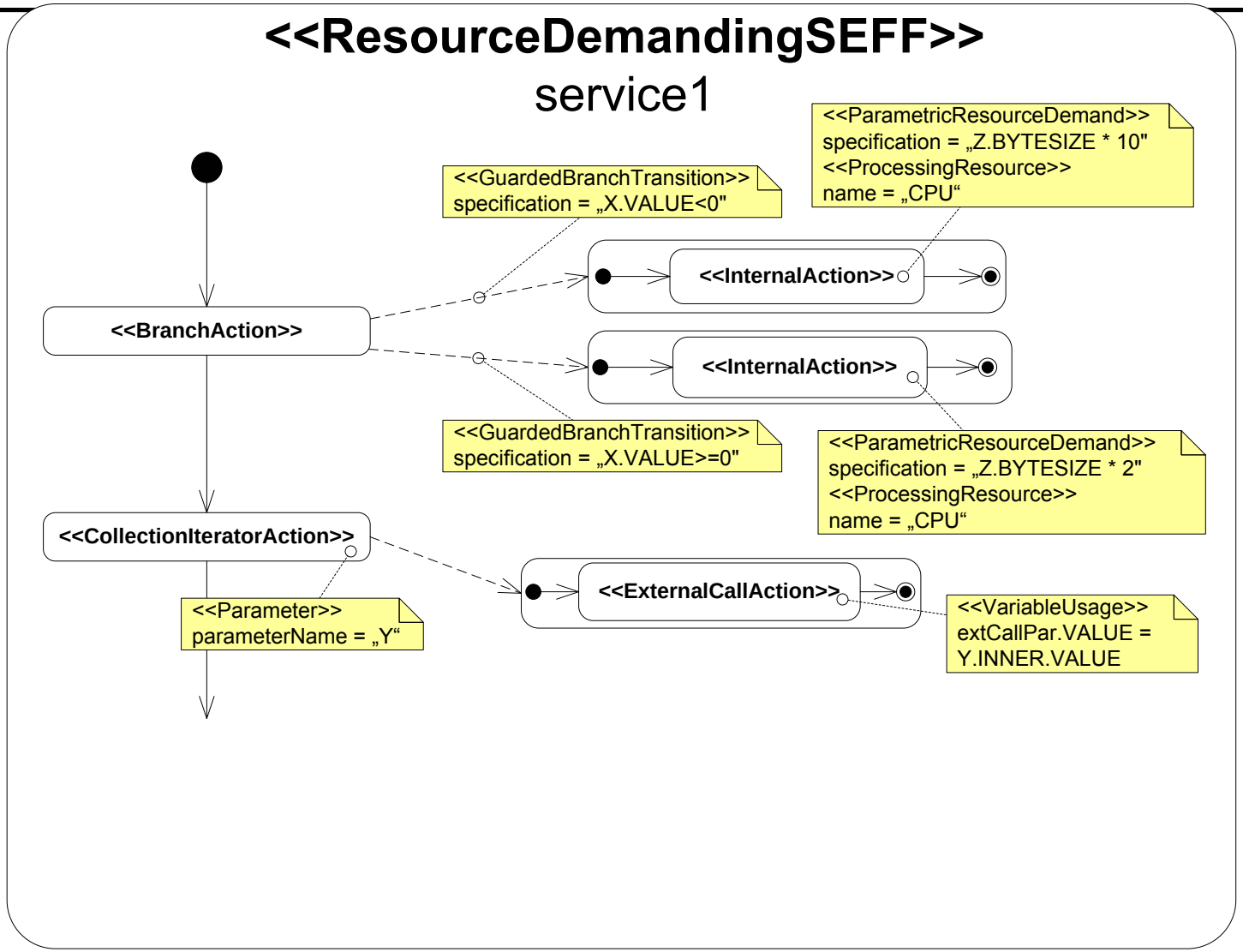
Service Effect Specification



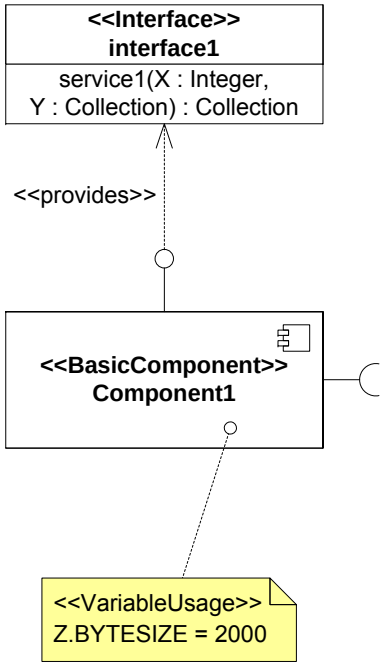


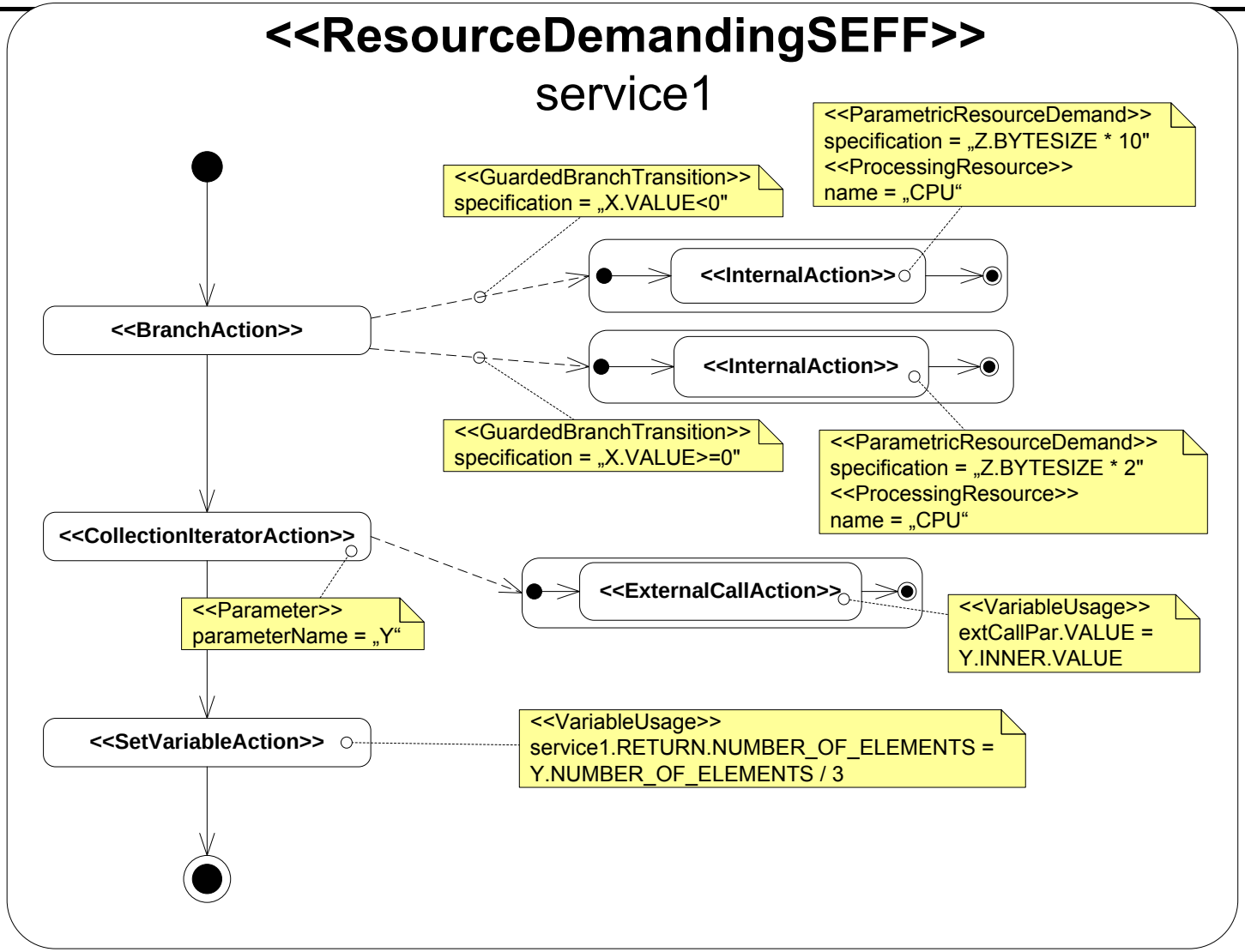
Service Effect Specification



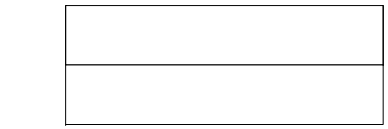


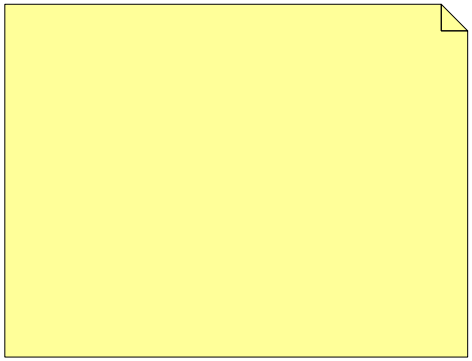
Service Effect Specification

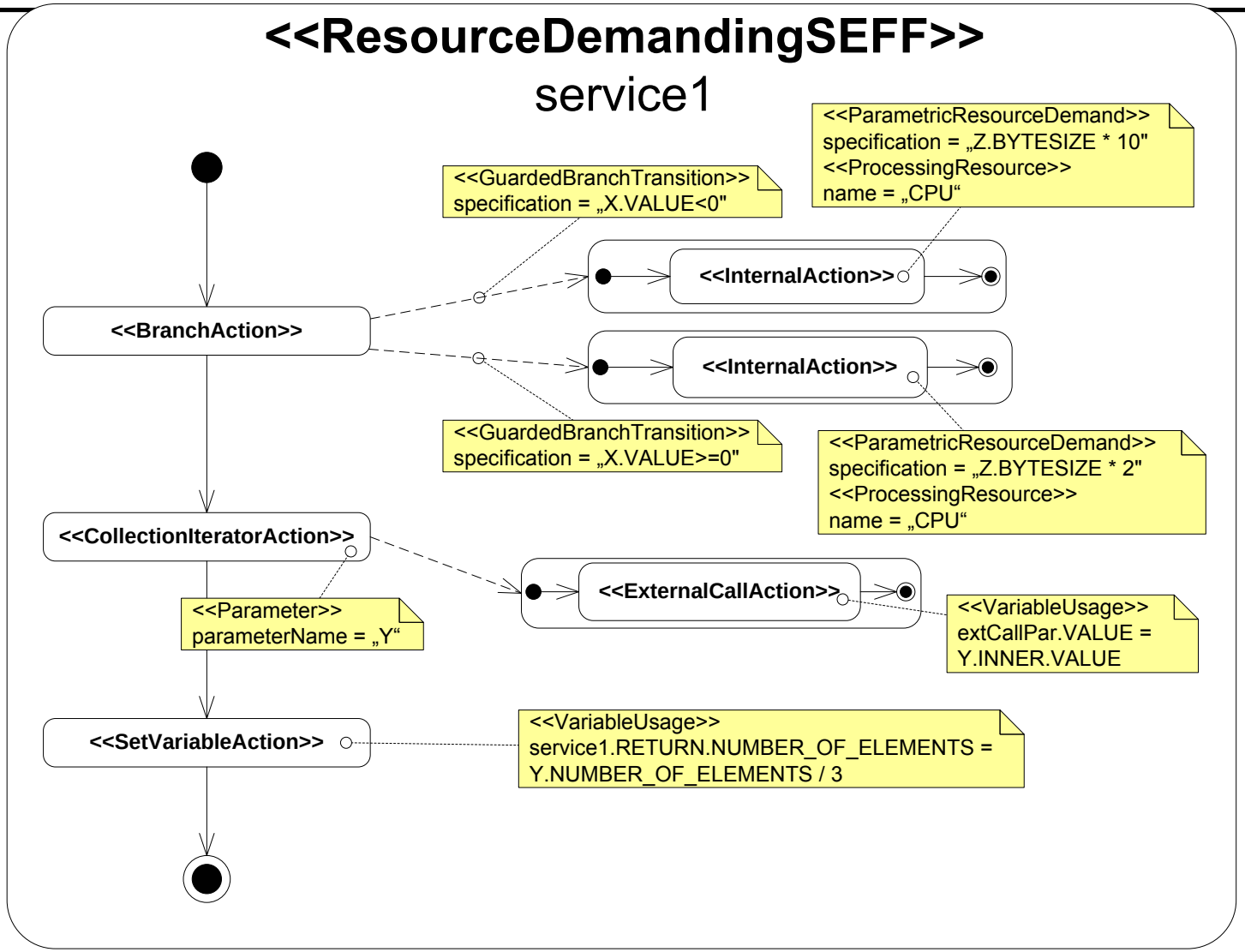




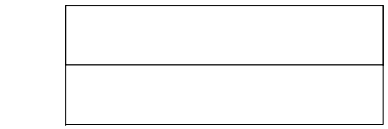
Service Effect Specification

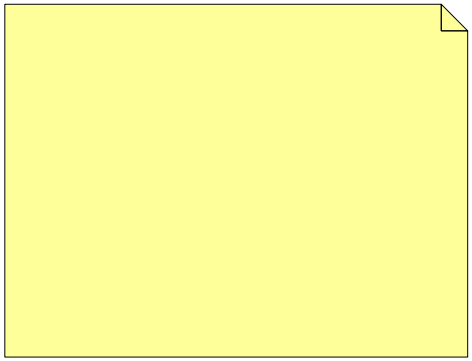


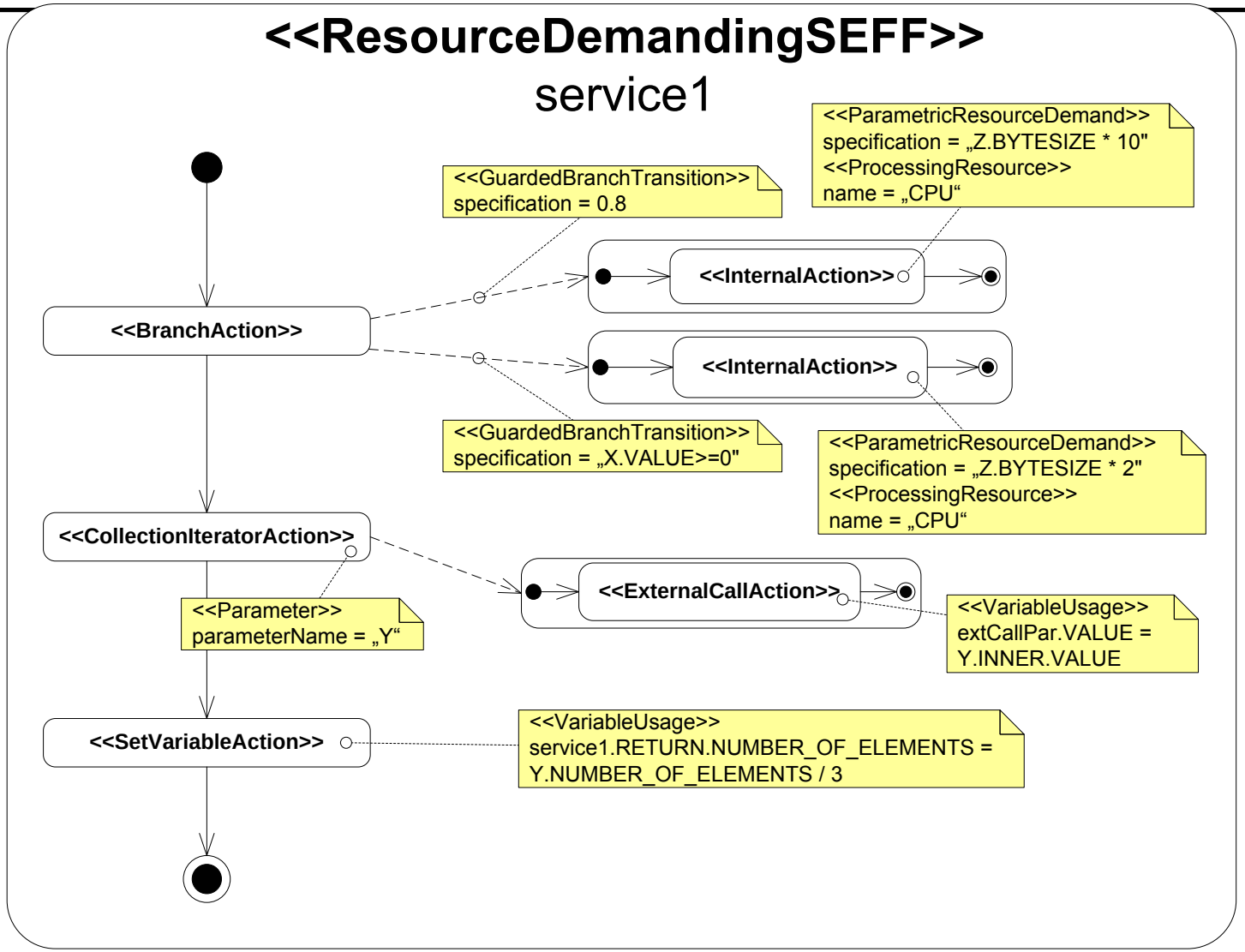




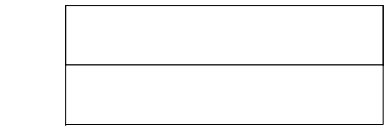
Service Effect Specification

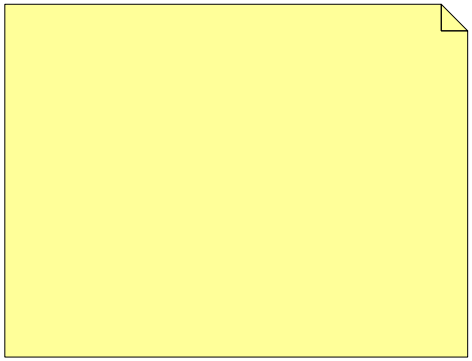


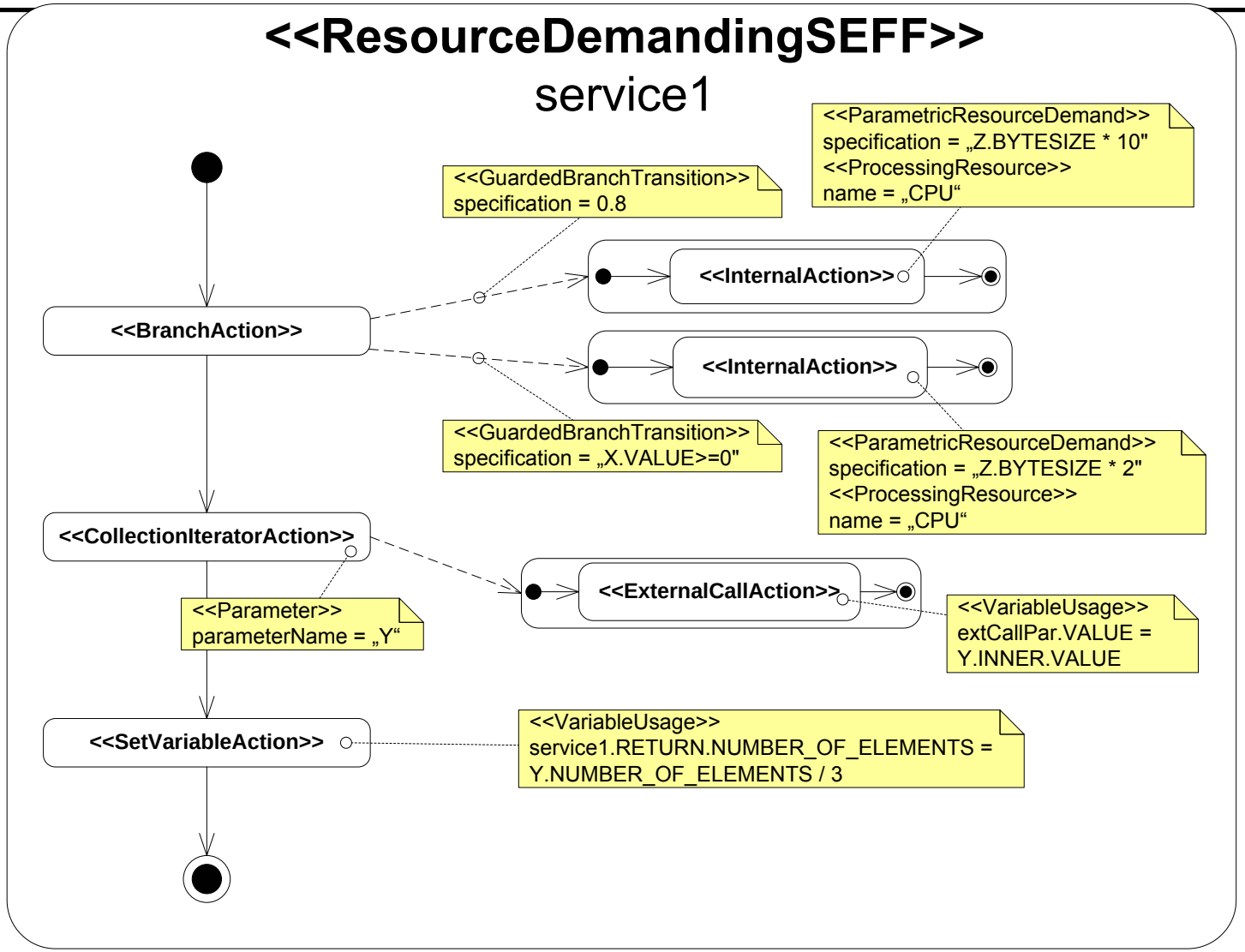




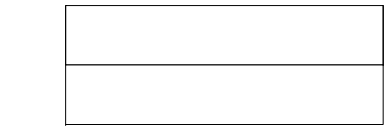
Service Effect Specification

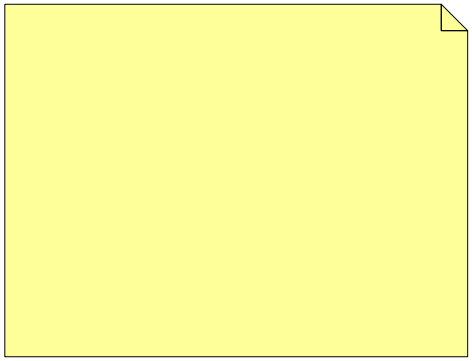


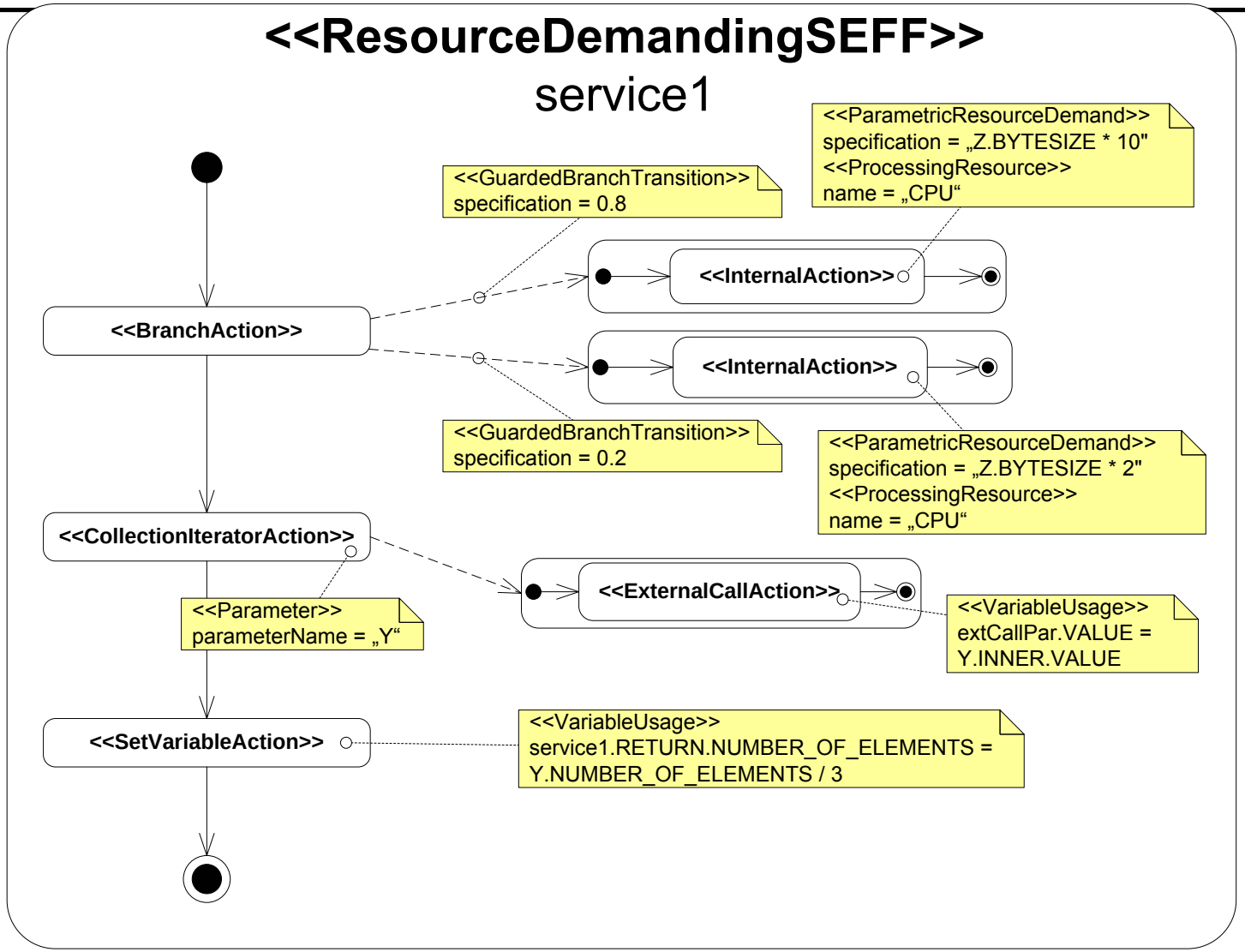




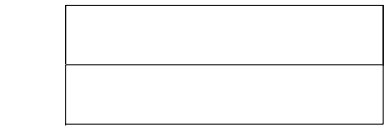
Service Effect Specification

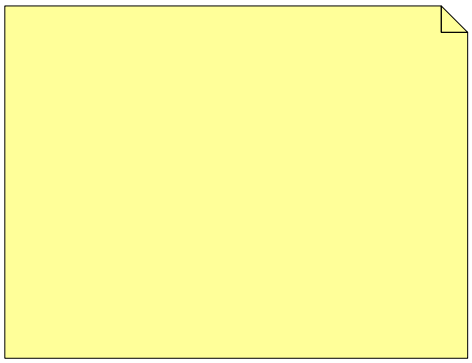


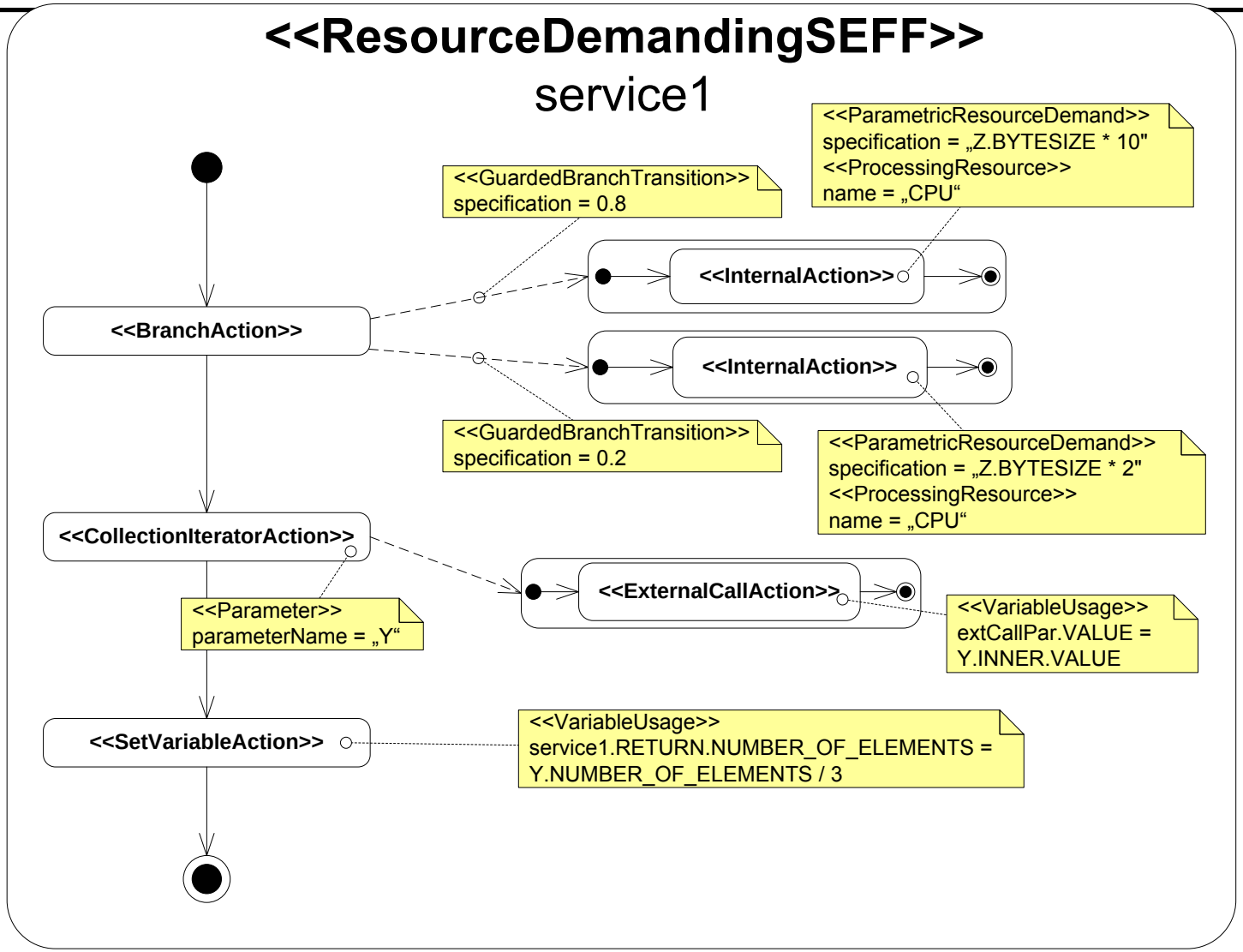




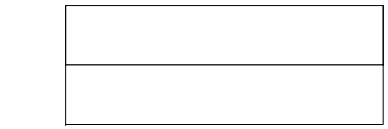
Service Effect Specification

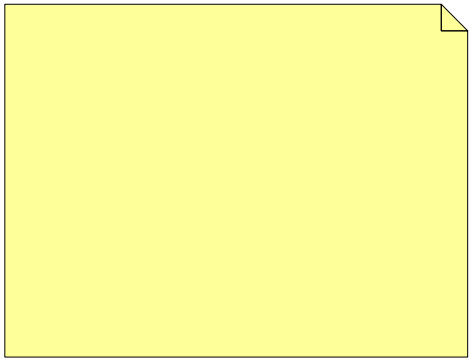


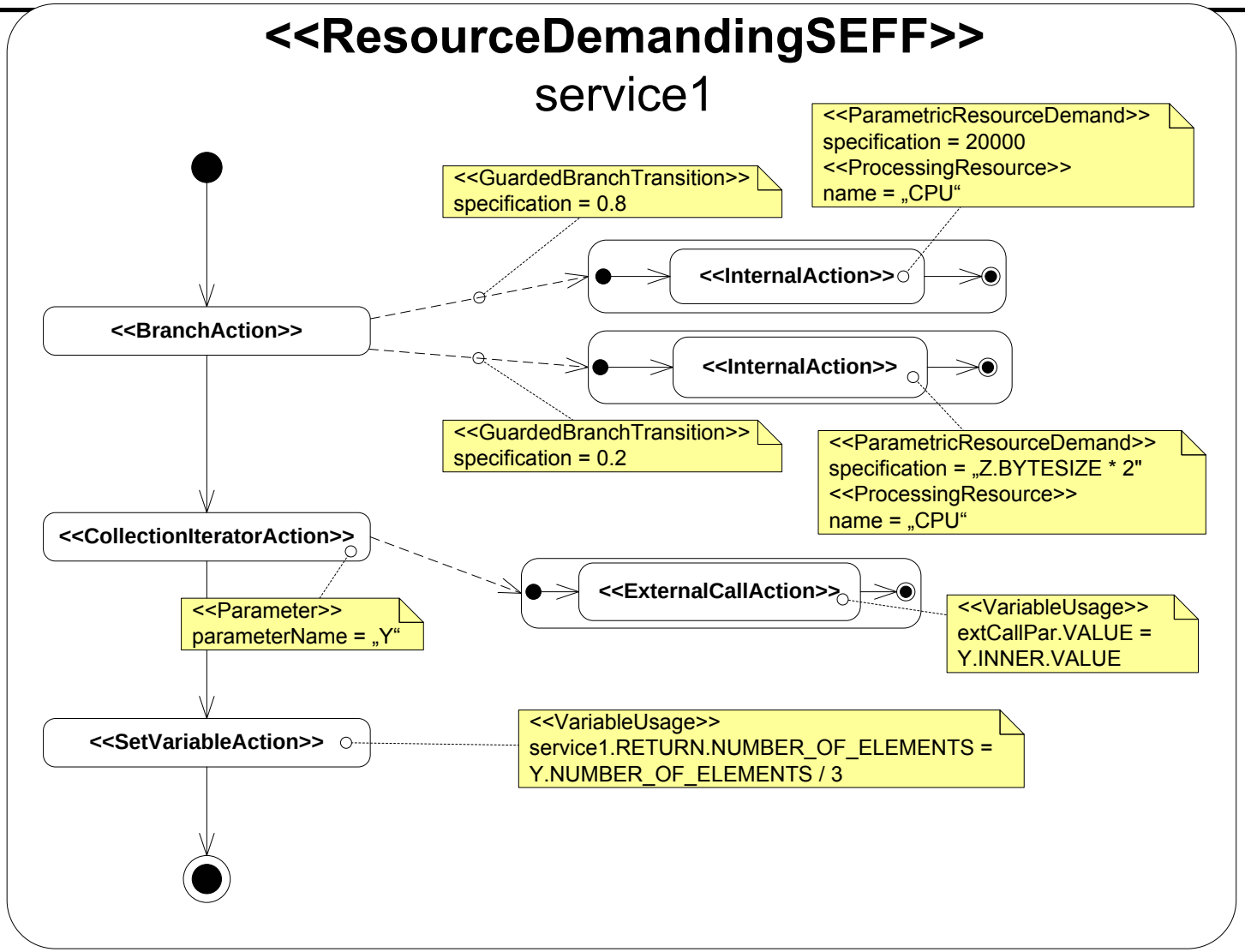




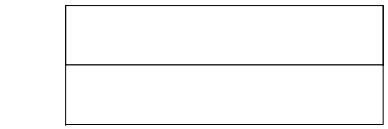
Service Effect Specification

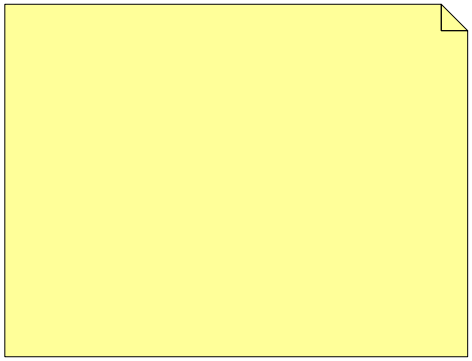


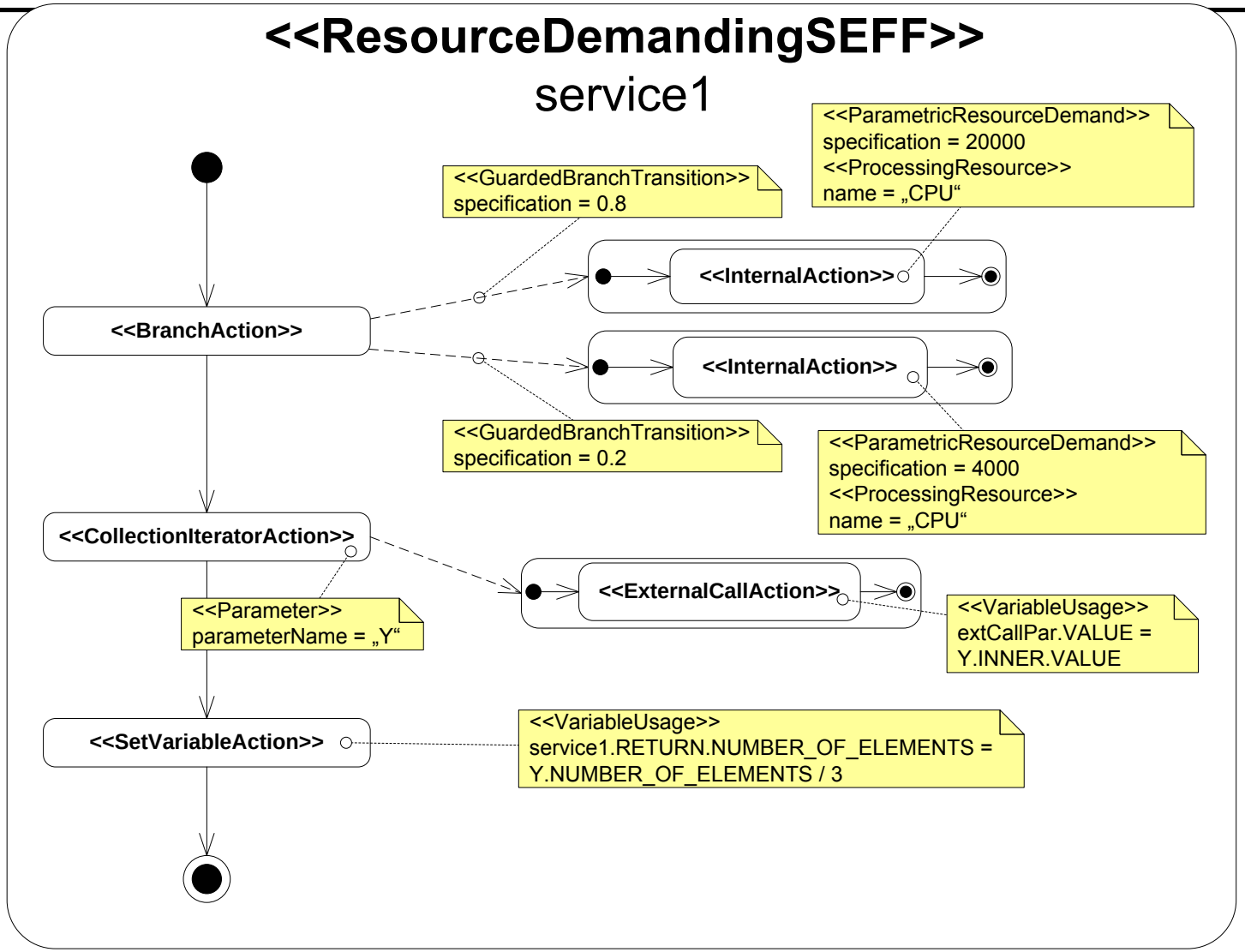




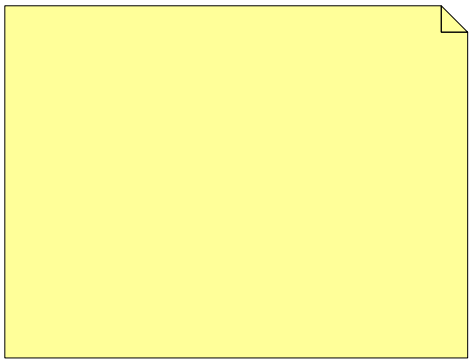
Service Effect Specification

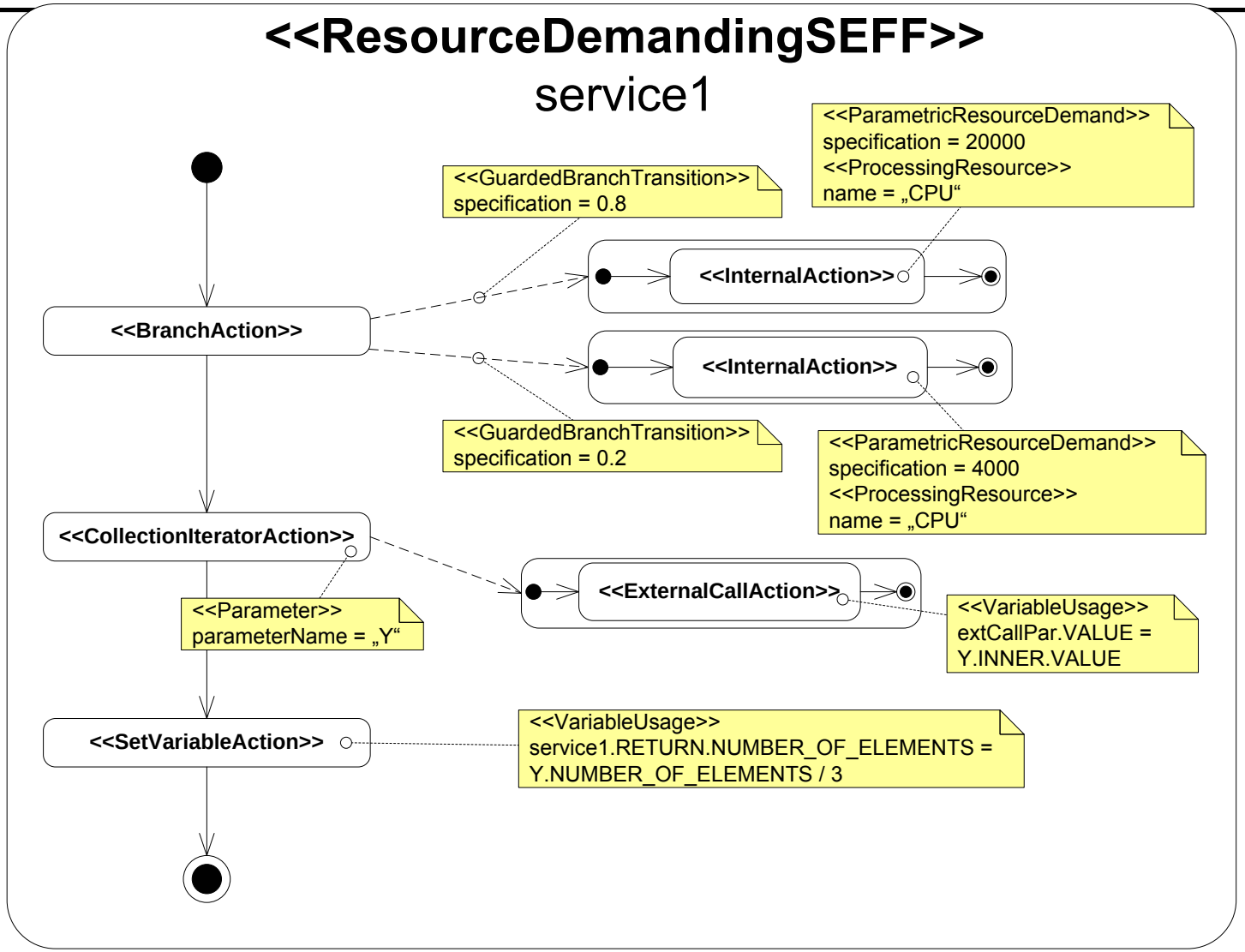




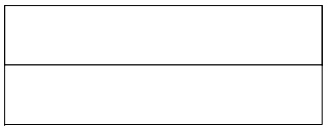


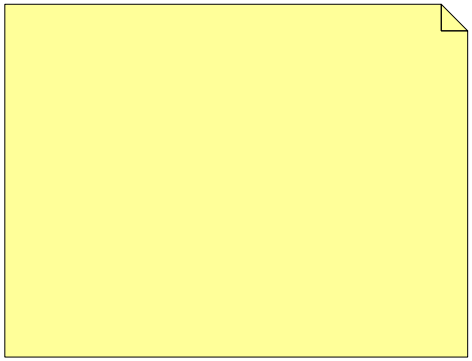
Service Effect Specification

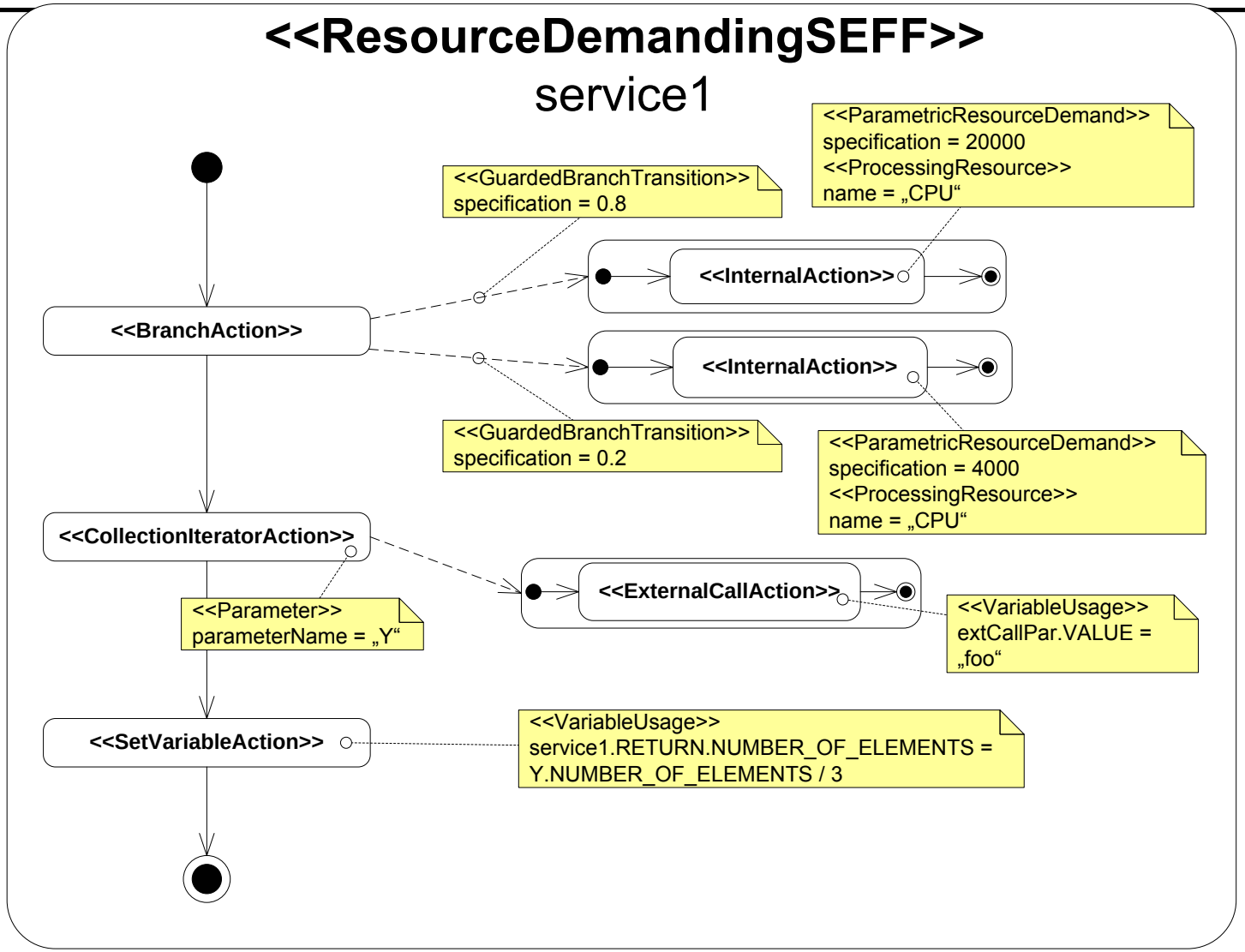




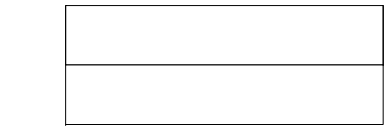
Service Effect Specification

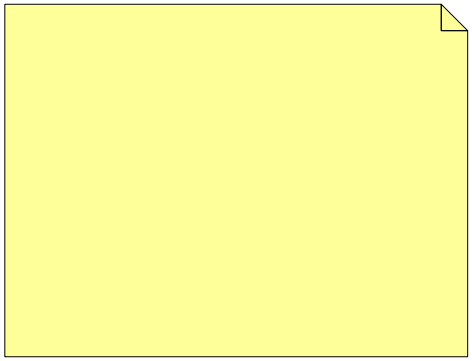


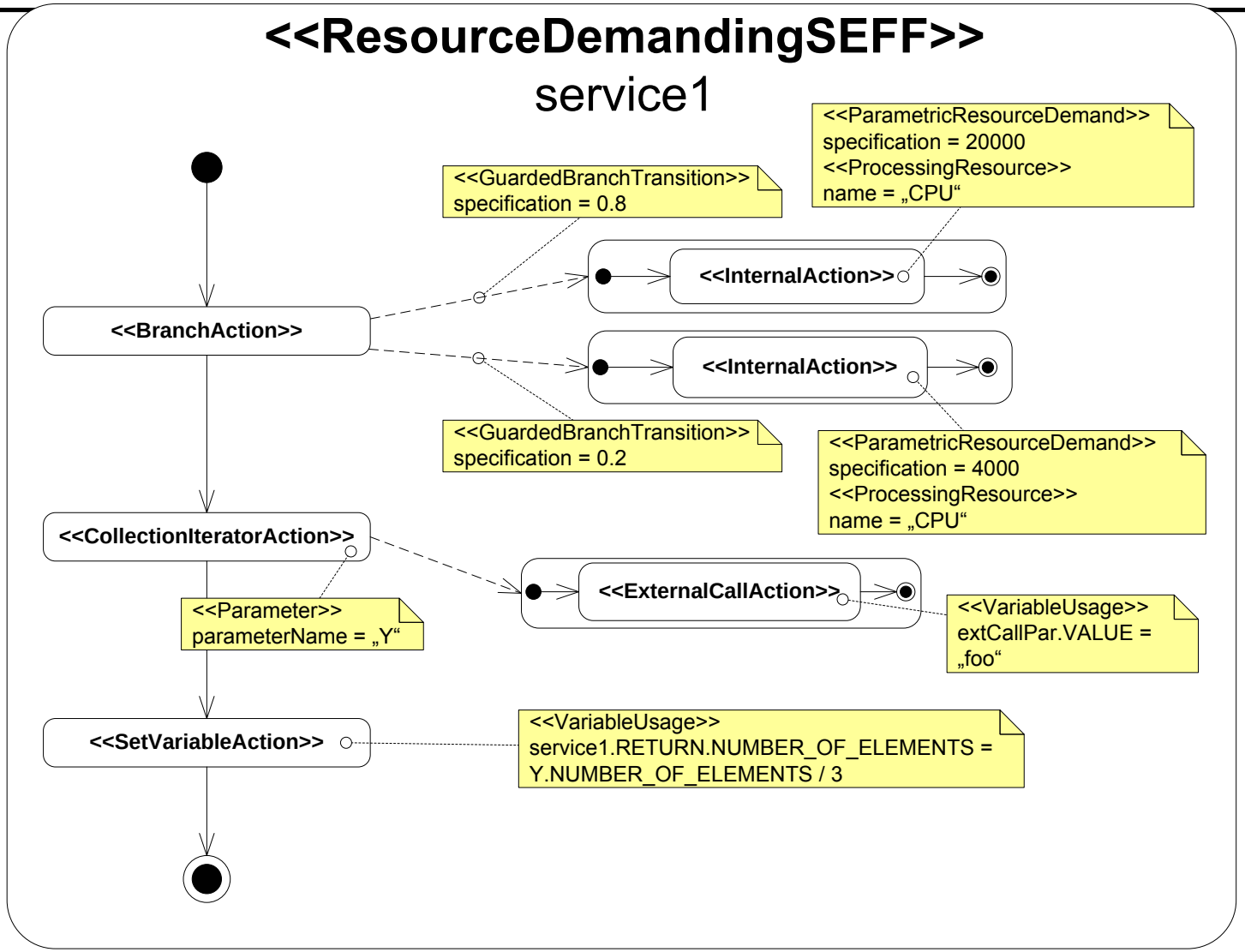




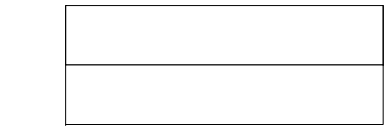
Service Effect Specification

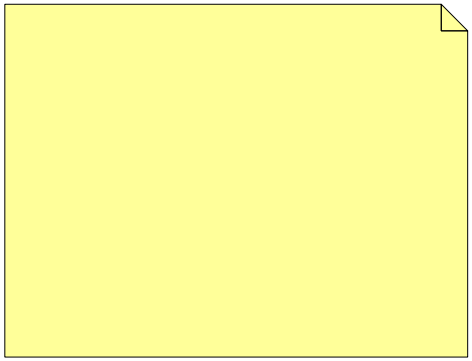


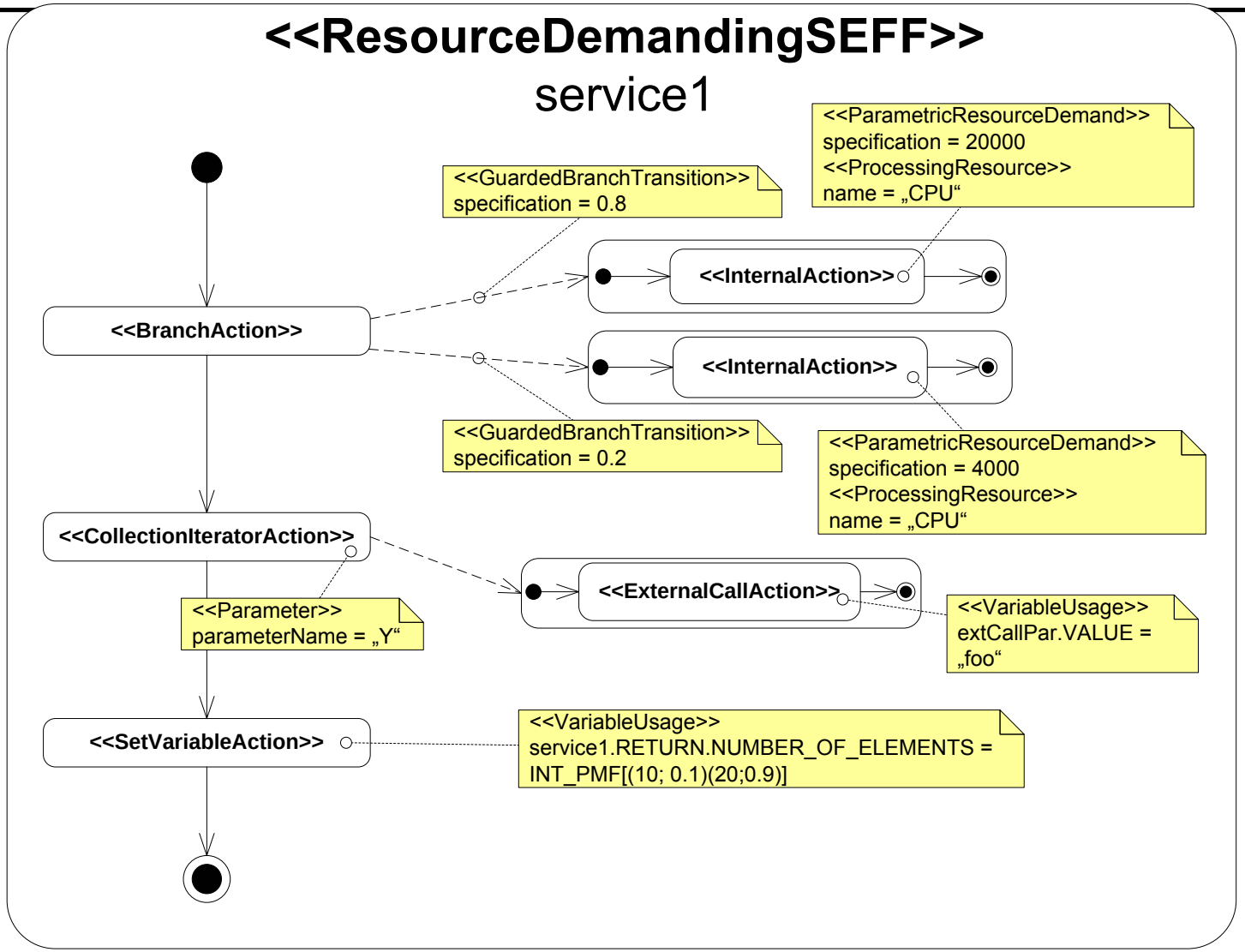




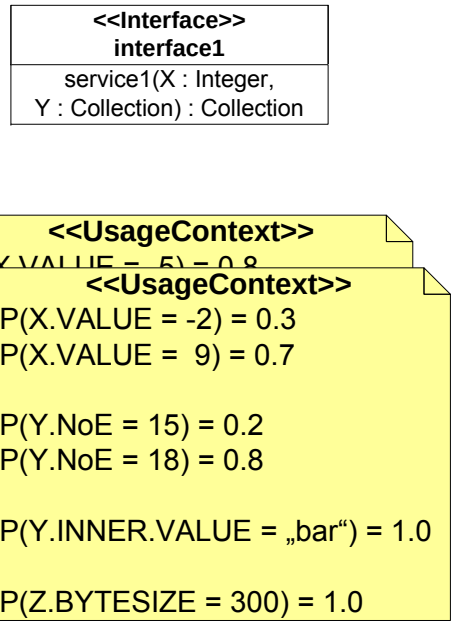
Service Effect Specification

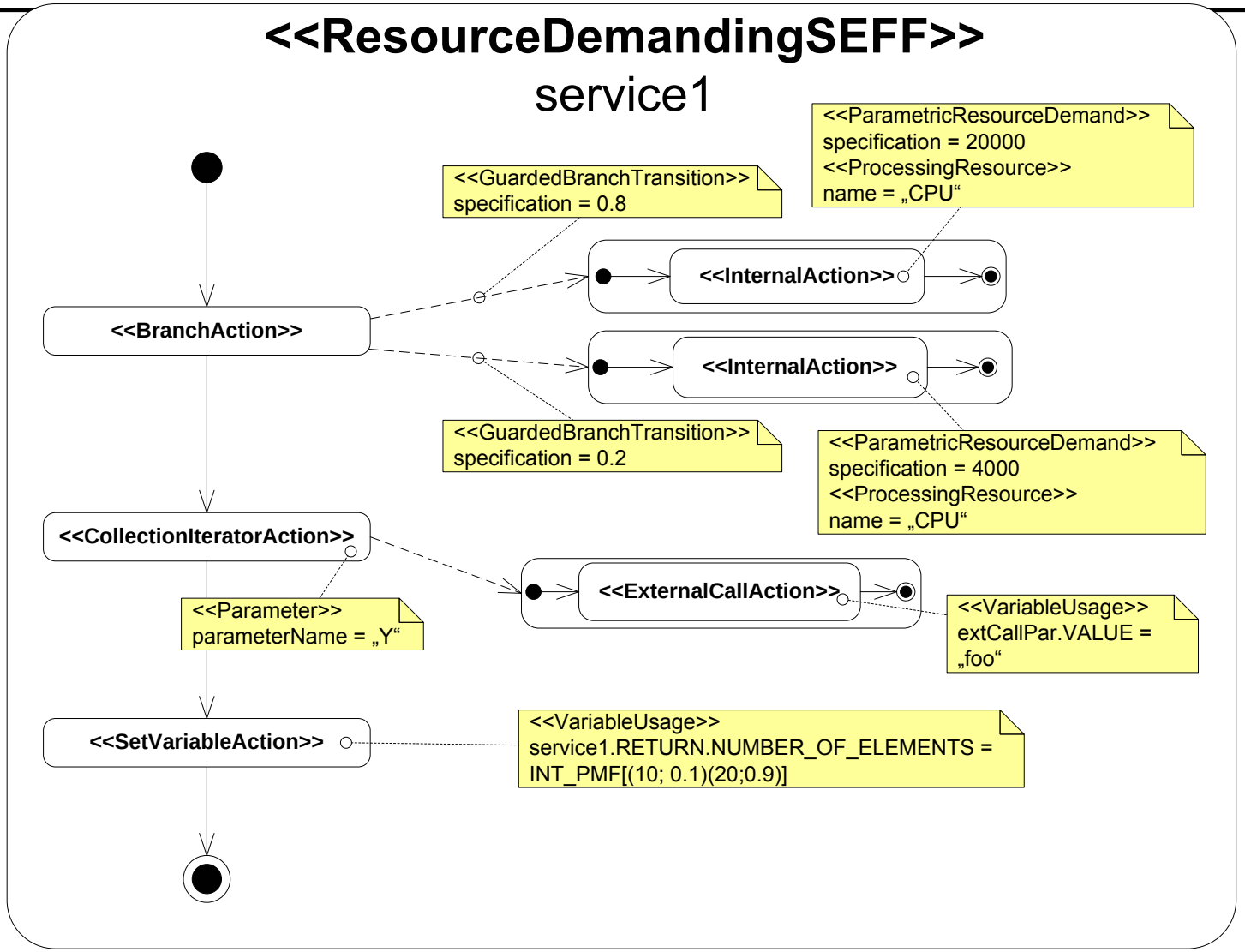




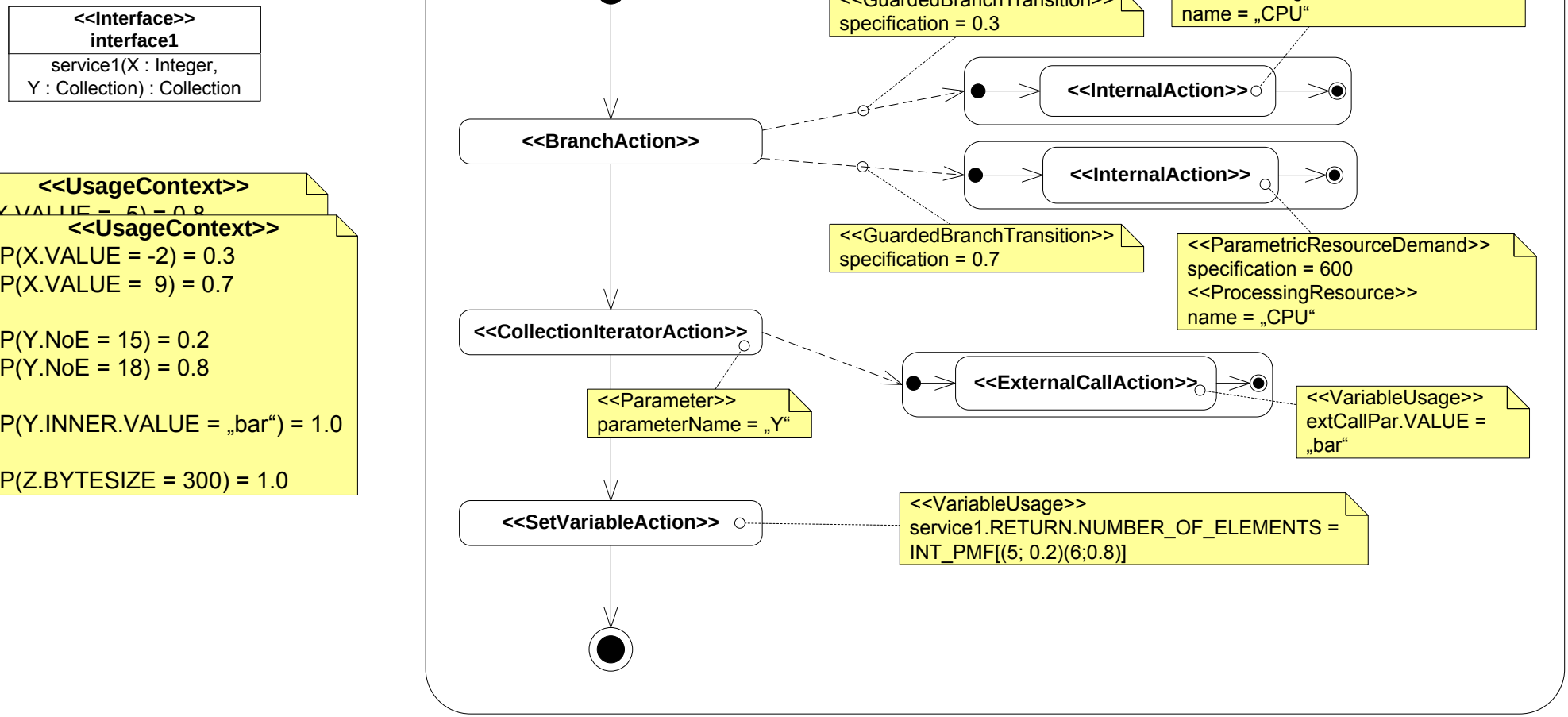


Service Effect Specification

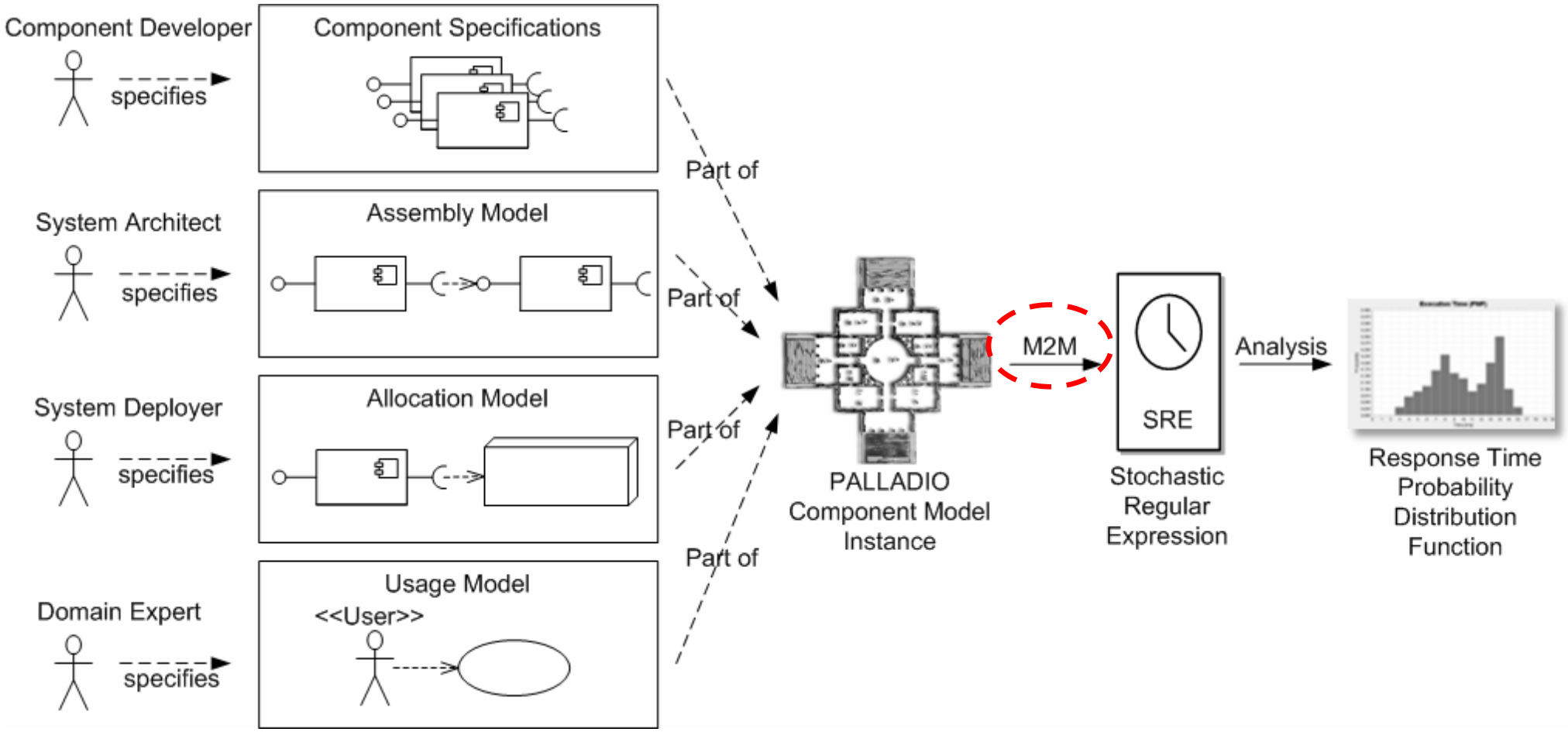


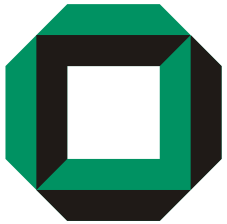


Service Effect Specification

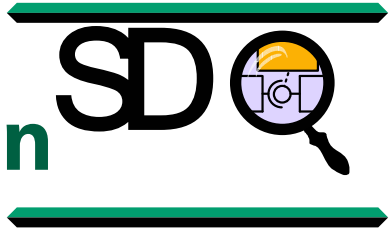


Palladio Component Model





Stochastic Regular Expression



Sequence

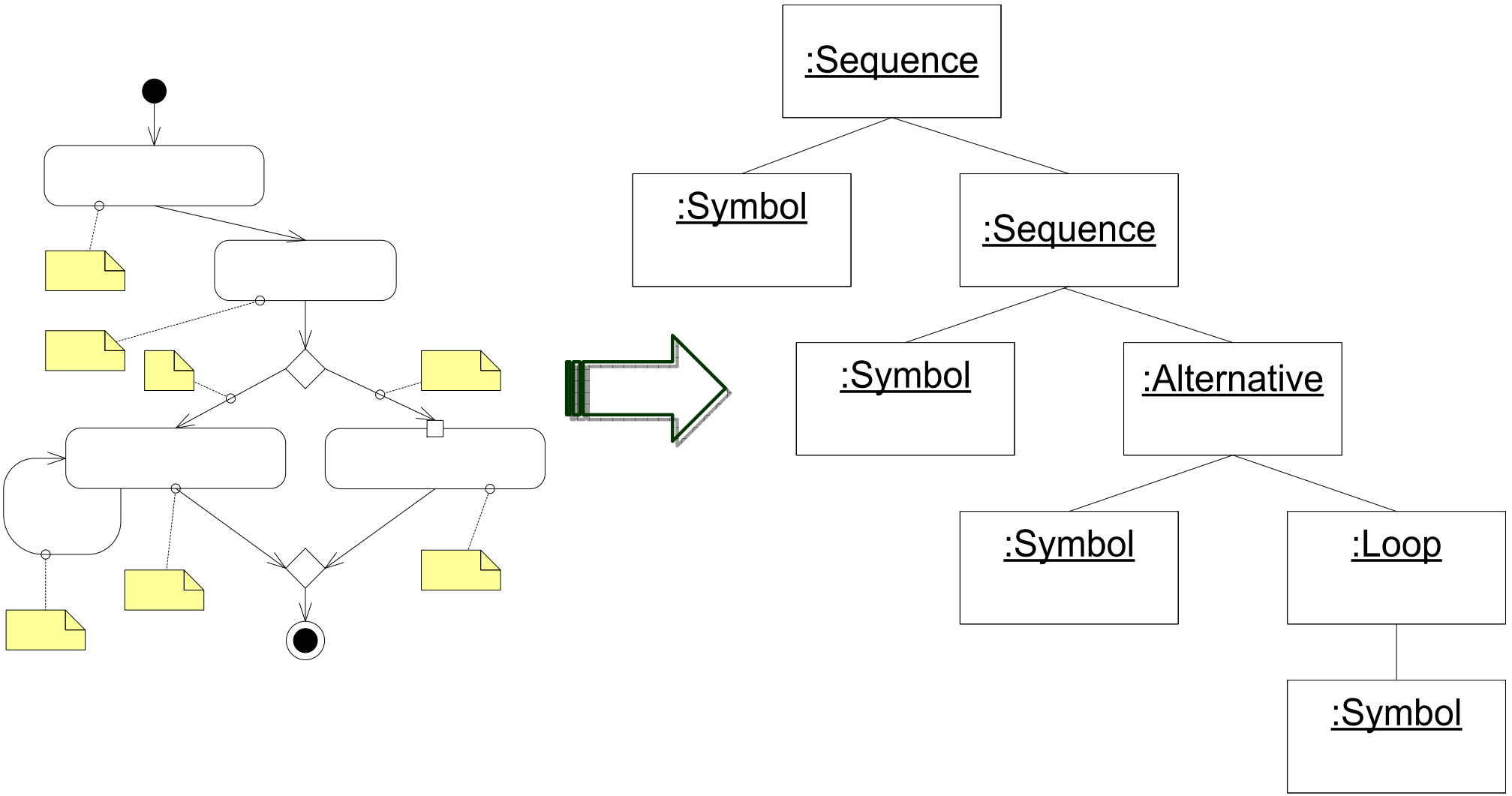
Loop Iterations (l)

$$P := a \mid P \cdot Q \mid P +_{\pi} Q \mid P^{*(l)}$$

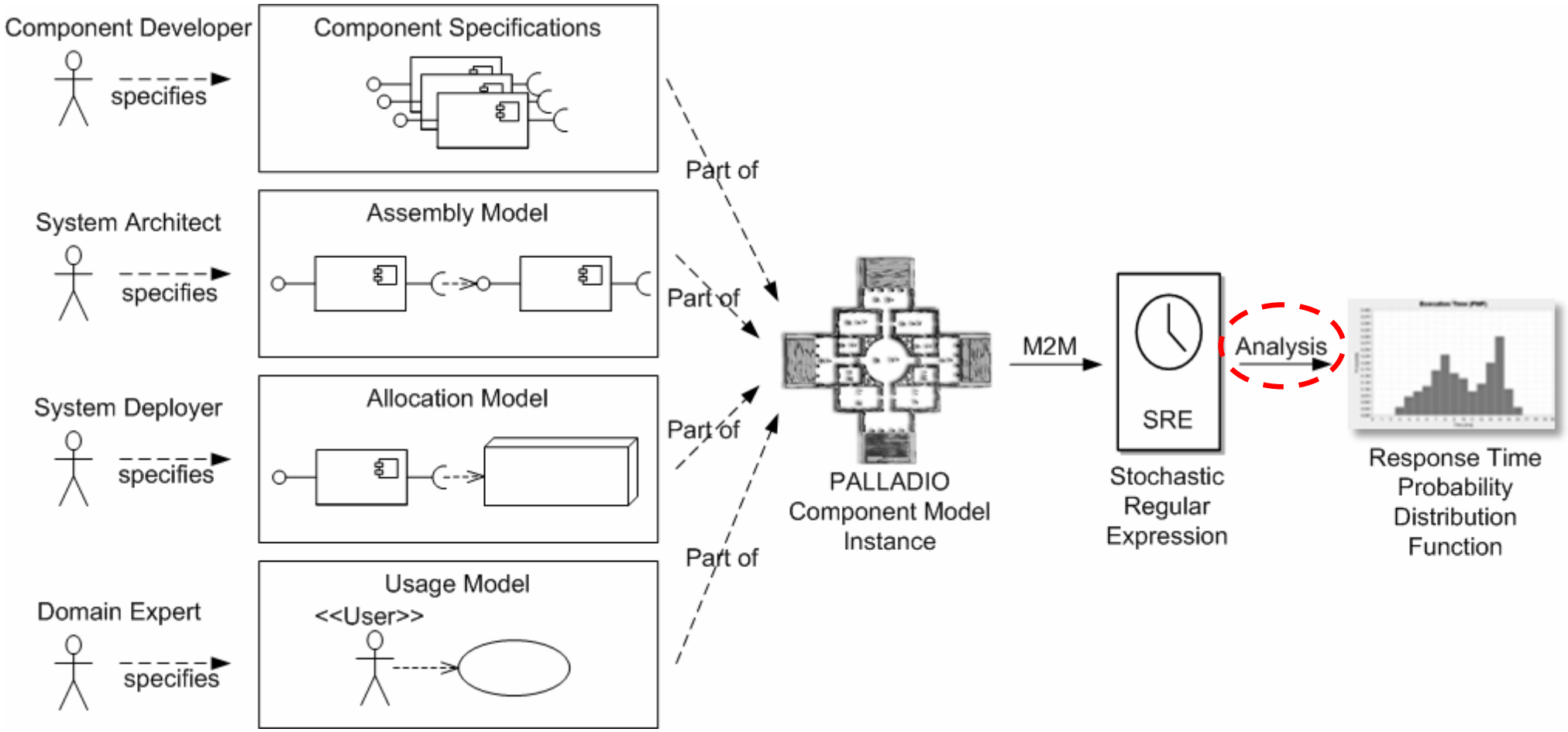
Symbol

Alternative Probability π, 1-π

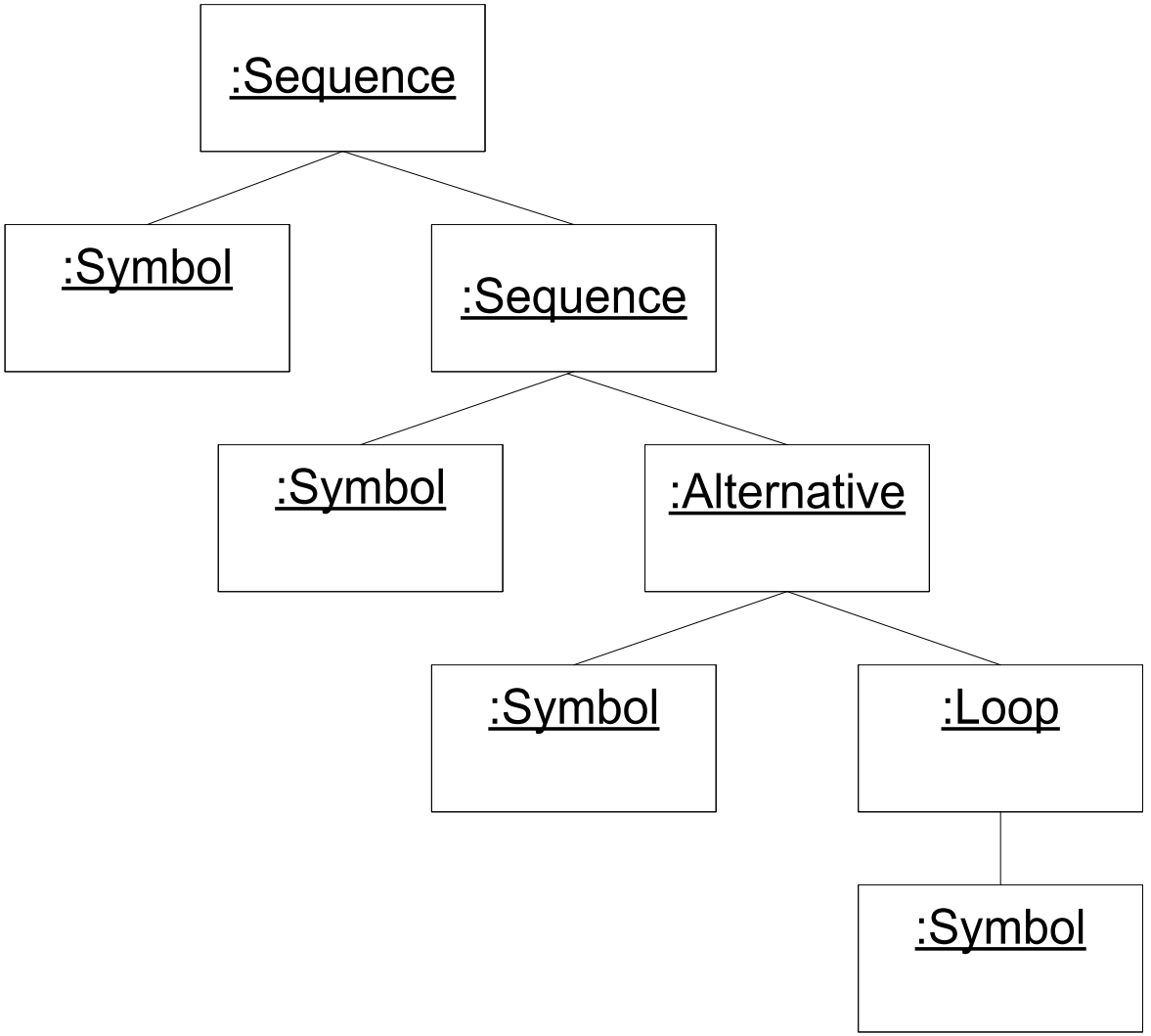
Transformation



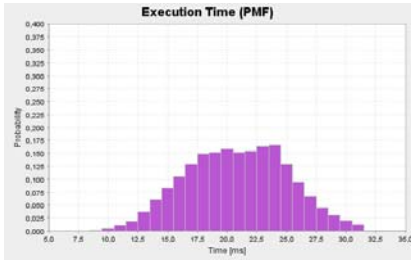
Palladio Component Model

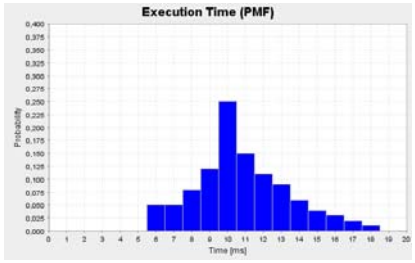


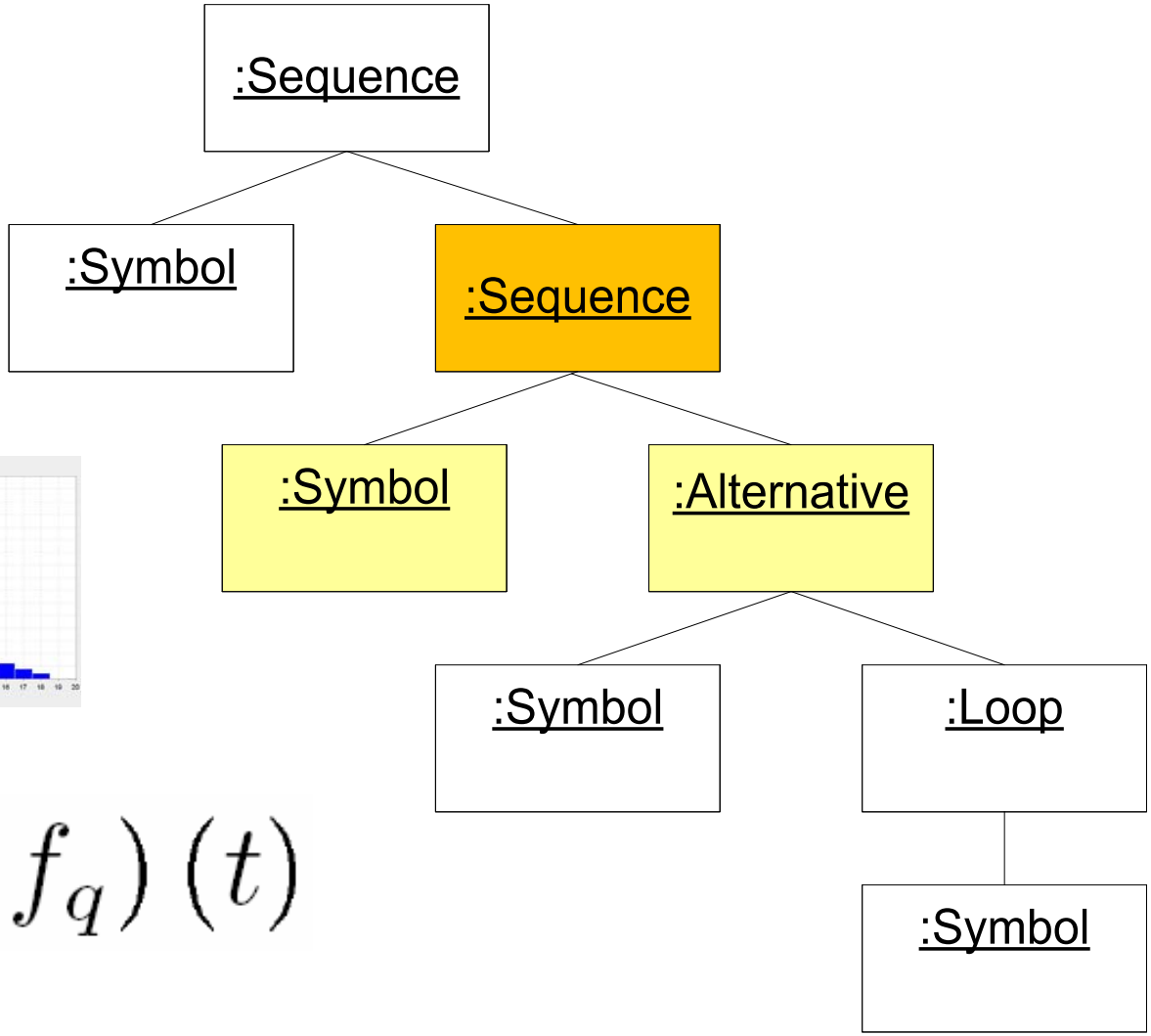
Model Solution



Model Solution

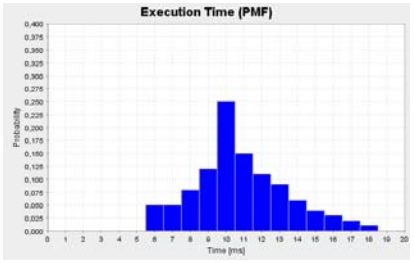


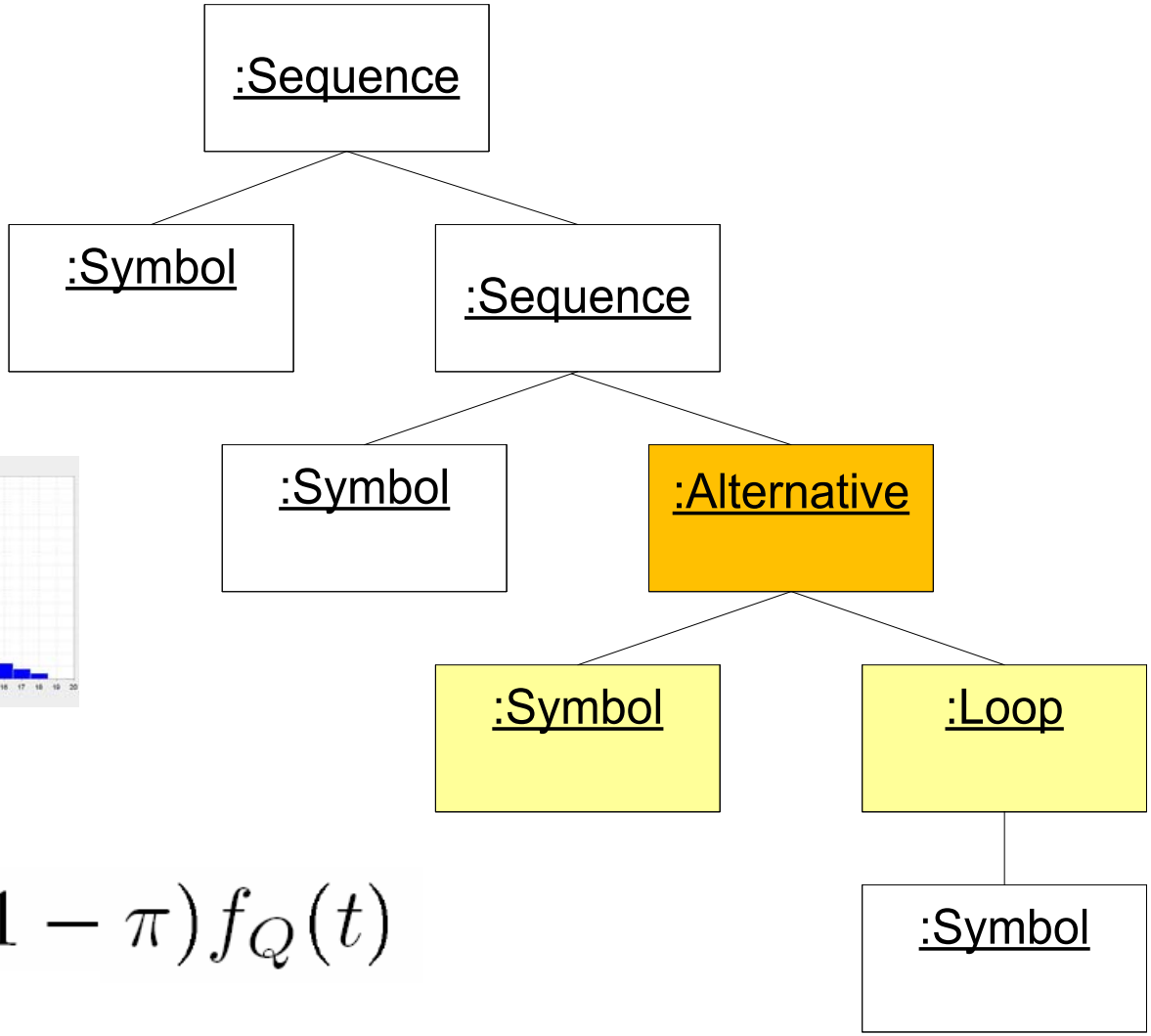




$$f_{P \cdot Q}(t) = (f_p \circledast f_q)(t)$$

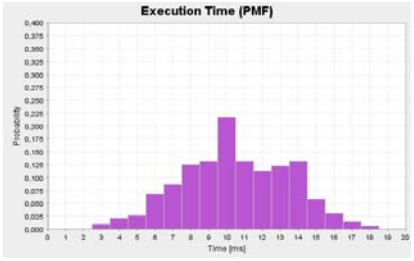
Model Solution

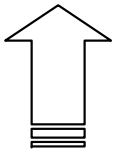


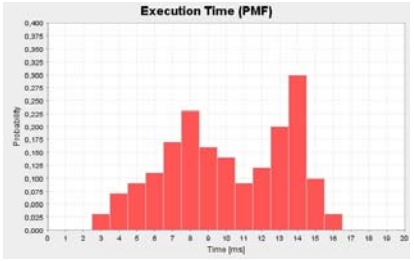


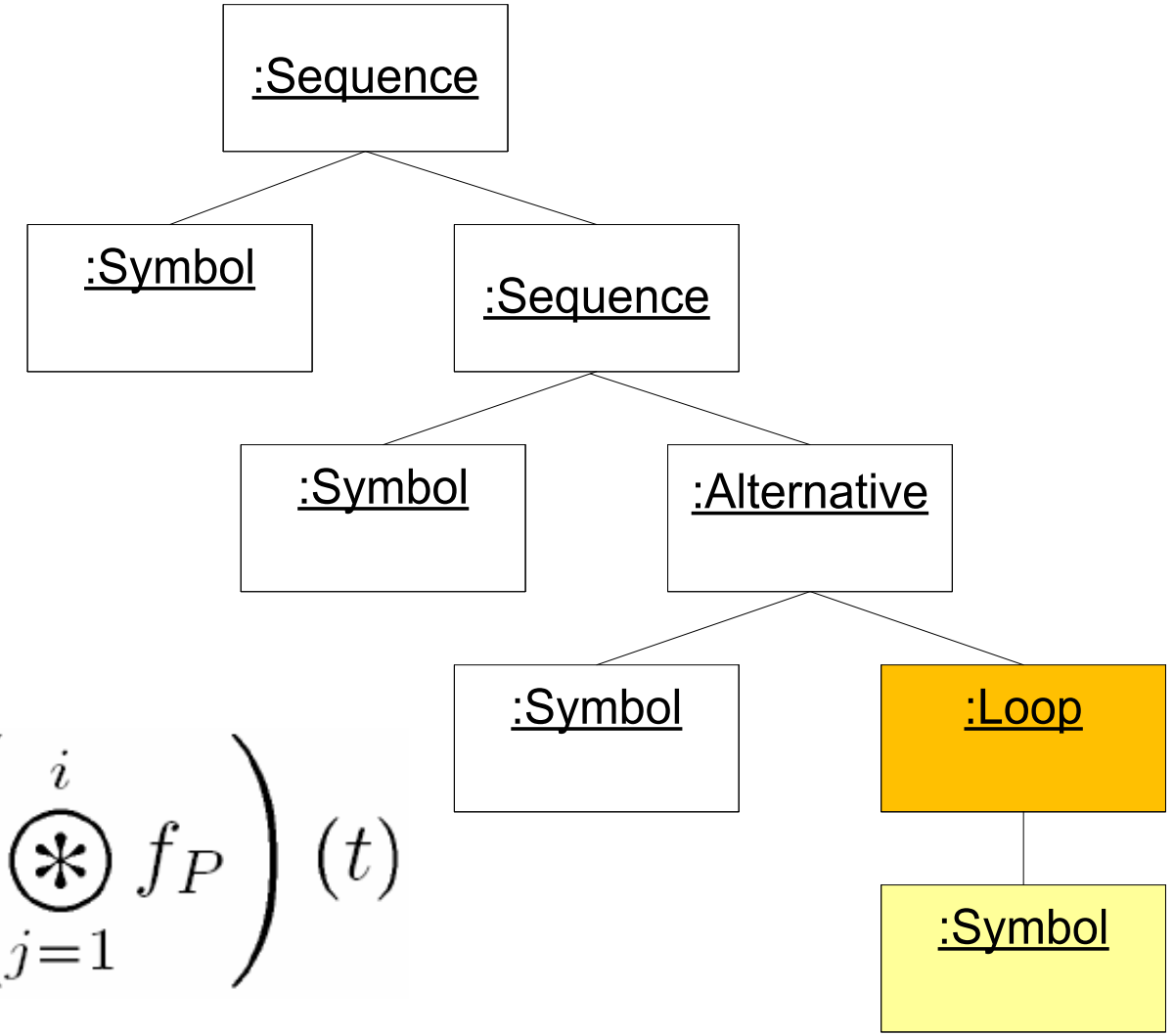
$$f_{P+_\pi Q}(t) = \pi f_P(t) + (1-\pi) f_Q(t)$$

Model Solution



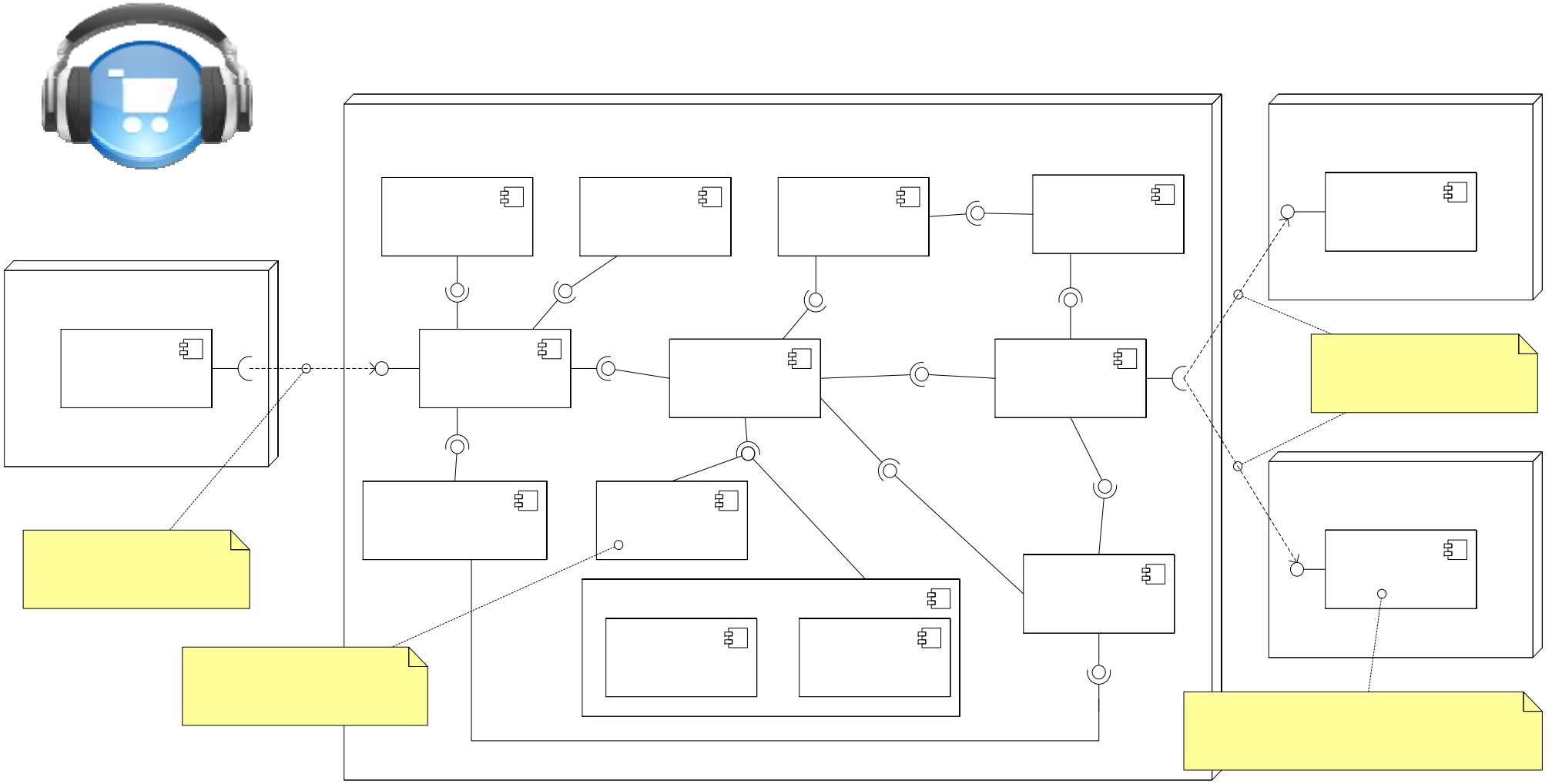




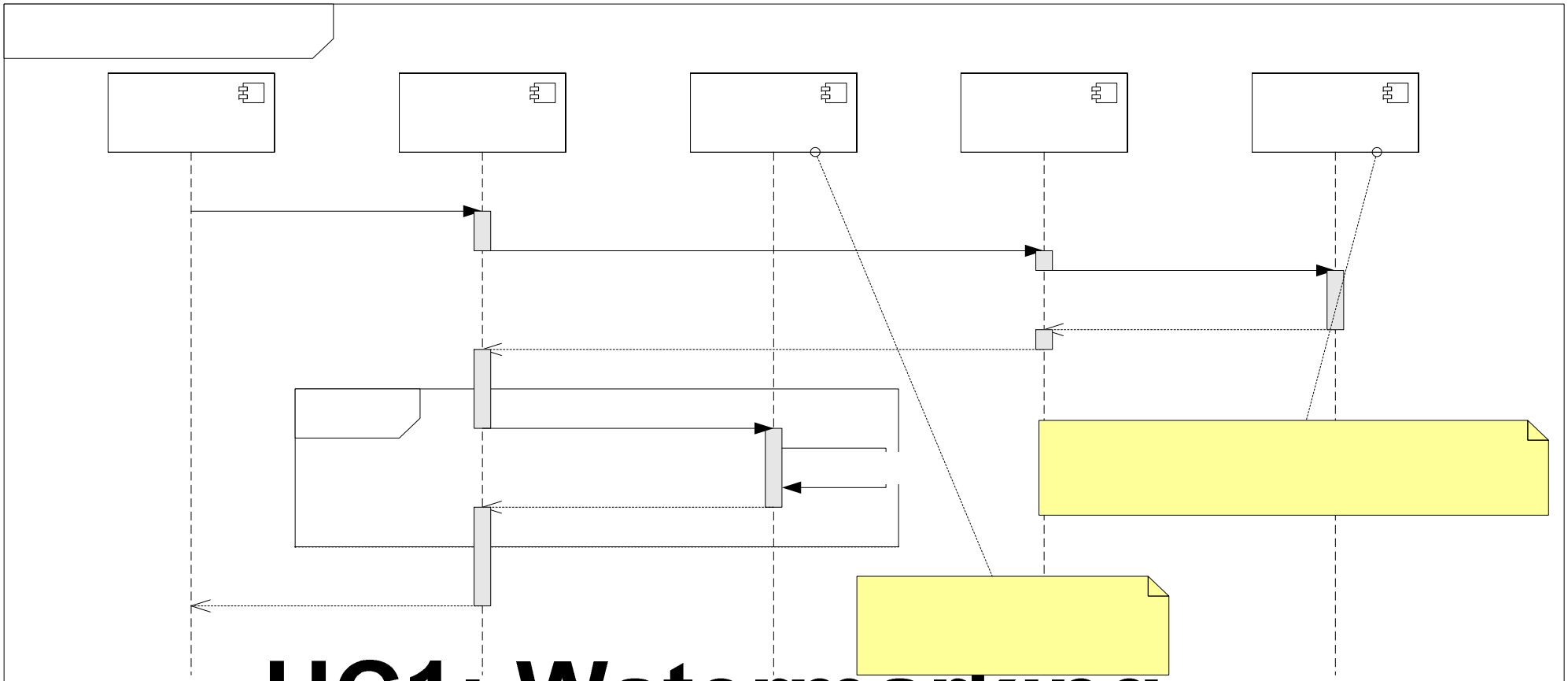


$$f_{P*(l)}(t) = \sum_{i=0}^{N} p_l(i) \left(\circledast_{j=1}^{i} f_P \right)(t)$$

MediaStore - Architecture



Download - Use Case



UC1: Watermarking

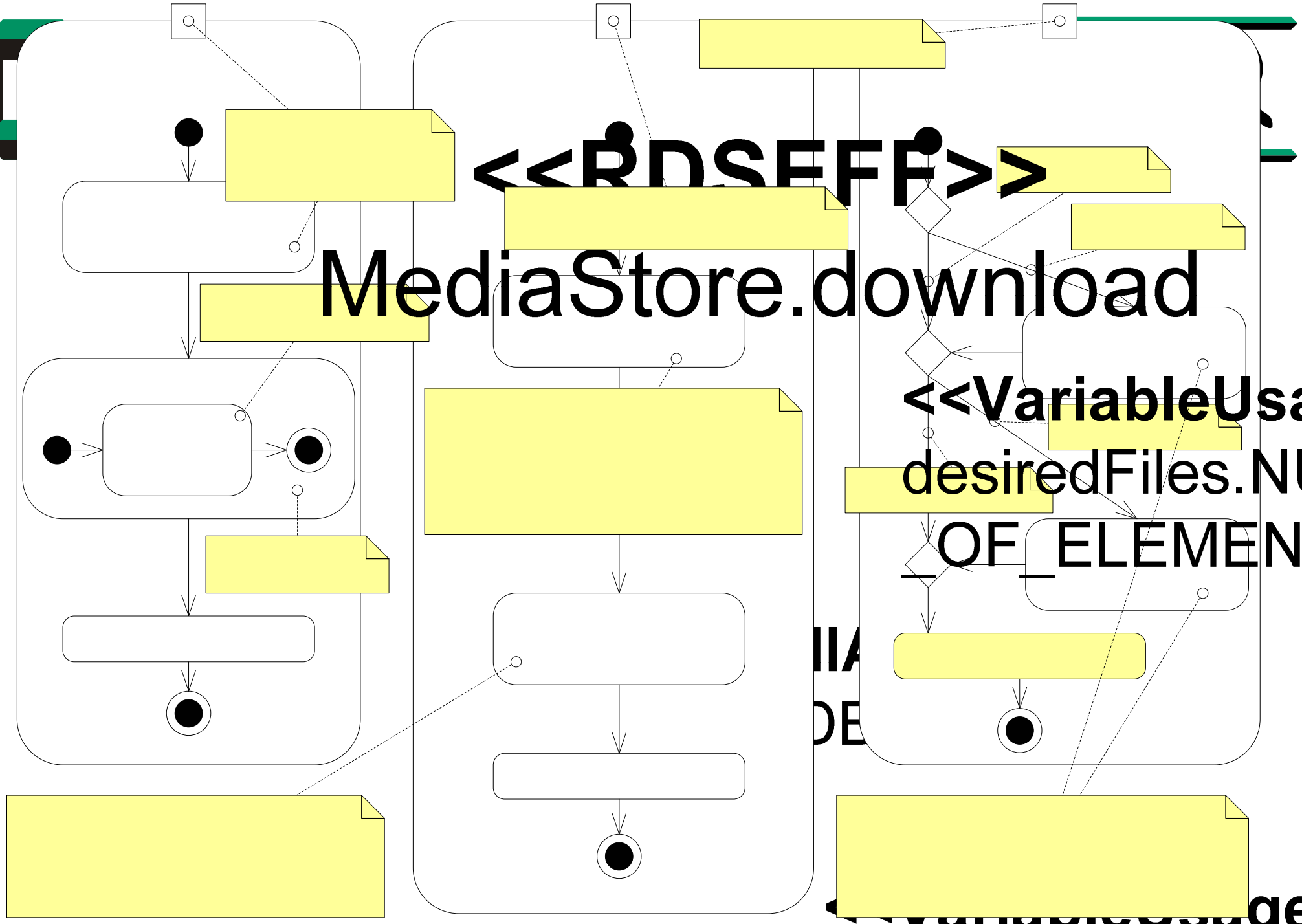
WebGUI

Mee

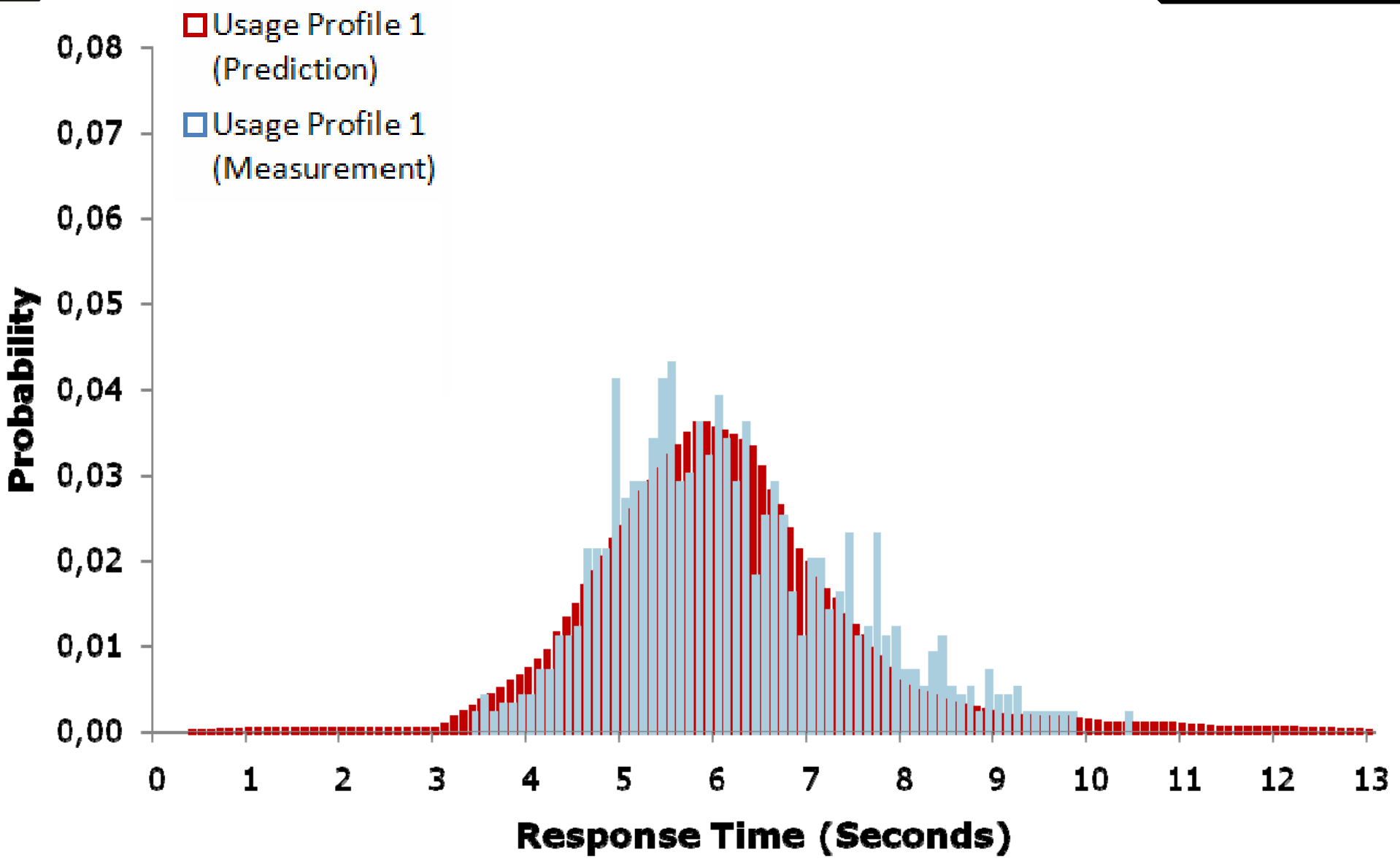
Case Study: Usage Profiles

Parameter Characterisation	Usage Profile 1 (AudioFiles)	Usage Profile 2 (VideoFiles)
DesiredFiles.NUMBER_OF_ELEMENTS	10-14	1
StoredFiles.NUMBER_OF_ELEMENTS	250000	10000
StoredFiles.INNER.BYTESIZE	1-15 MB (audio)	95-105MB (video)
ProbIncludeID.VALUE	1.0	1.0
ProbIncludeLyrics.VALUE	0.0	1.0

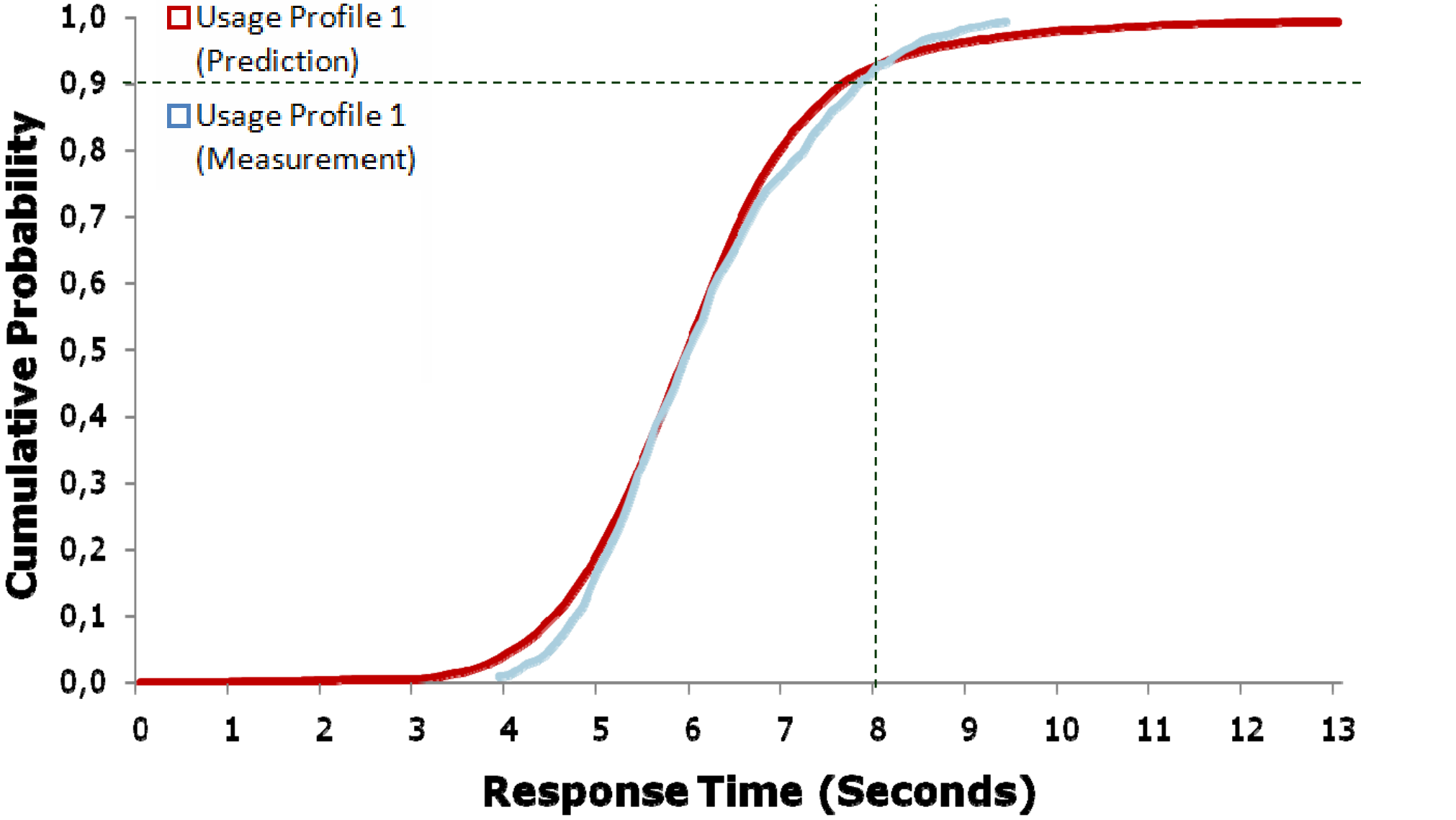
<<UsageModel>>
MediaStoreUsage


<<RDSEFF>>
MediaStore.download
<<VariableUsag
desiredFiles.NUM
_OF_ELEMENTS
IIA
DE
<<VariableUsage>
fileToMark.BYTESI

Results


Usage Profile 1 (Prediction)
Usage Profile 1 (Measurement)
Probability
0,08
0,07
0,06
0,05
0,04
0,03
0,02
0,01
0,00
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Response Time (Seconds)

Results


Usage Profile 1
(Prediction)
Usage Profile 1
(Measurement)
Cumulative Probability
1,0
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Response Time (Seconds)