Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 1:
Introduction and Notions of Quality

Christoph Lüth, Dieter Hutter, Jan Peleska

Organisatorisches

Generelles

- Einführungsvorlesung zum Masterprofil S & Q
- 6 ETCS-Punkte
- Vorlesung:
  - Dienstag 12 – 14 Uhr (MZH 1110)
- Übung:
  - Donnerstag 16 – 18 Uhr (MZH 4140)
- Veranstalter:
  - Christoph Lüth `<clueth@uni-bremen.de>`, MZH 4186, Tel. 59830
  - Helmar Hutschenreuter `<hutschen@uni-bremen.de>`
- Material (Folien, Artikel, Übungsblätter) auf der Homepage:
  http://www.informatik.uni-bremen.de/~clueth/lehre/ssq.ws19

Vorlesung

- Foliensätze als **Kernmaterial**
  - Sind auf Englisch (Notationen!)
  - Nach der Vorlesung auf der Homepage verfügbar
- Ausgewählte Fachartikel als **Zusatzmaterial**
  - Auf der Homepage verlinkt (ggf. in StudIP)
- Bücher nur für einzelne Teile der Vorlesung verfügbar:
  - Nancy Leveson: Engineering a Safer World
  - Ericson: Hazard Analysis Techniques for System Safety
  - Nilson, Nilson: Principles of Program Analysis
  - Winskel: The Formal Semantics of Programming Languages
- Zum weiteren Stöbern:
  - Wird im Verlauf der Vorlesung bekannt gegeben

Übungen

- Übungsblätter:
  - „Leichtgewichte" Übungsblätter, die in der Übung bearbeitet und schnell korrigiert werden können.
  - Übungsblätter vertiefen Vorlesungsstoff.
  - Bewertung gibt schnell Feedback.
- Übungsbetrieb:
  - Gruppen bis zu 3 StudentInnen
  - Ausgabe der Übungsblätter Dienstag in der Übung
    - Zeitgleich auf der Homepage
    - Erstes Übungsblatt: diese Woche (17.10.2019)
  - Bearbeitung: während der Übung
  - Abgabe: bis Donnerstag abend

Prüfungsform

- Bewertung der Übungen:
  - A (sehr gut (1.0) – nichts zu meckern, nur wenige Fehler)
  - B (gut (2.0) – kleine Fehler, im großen und ganzen gut)
  - C (befriedigend (3.0) – größere Fehler oder Mängel)
  - Nicht bearbeitet (oder zu viele Fehler)
- Prüfungsleistung:
  - Teilnahme am Übungsbetrieb (20%)
    - Übungen keine Voraussetzung
  - Mündliche Prüfung am Ende des Semesters (80%)
    - Einzelprüfung, ca. 20- 30 Minuten

Ziel der Vorlesung

- Methoden und Techniken zur Entwicklung sicherheitskritischer Systeme
- Überblick über verschiedene Mechanismen
  d.h. auch Überblick über vertiefende Veranstaltungen
  - Theorie reaktiver Systeme
  - Grundlagen der Sicherheitsanalyse und des Designs
  - Formale Methoden der Softwaretechnik
  - Einführung in die Kryptographie
  - Qualitätsorientierter Systementwurf
  - Test von Schaltungen und Systemen
  - Informationssicherheit -- Prozesse und Systeme
- Verschiedene Dimensionen
  - Hardware vs. Software
  - Security vs. Safety
  - Qualität der Garantien

Overview

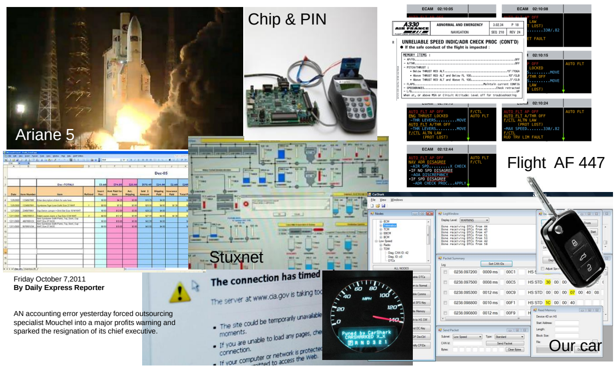
Objectives

- This is an introductory lecture for the topics

  Quality – Safety – Security

- Bird's eye view of everything relevant related to the development of systems of high quality, high safety or high security.
- The lecture reflects the fundamentals of the research focus quality, safety & security at the department of Mathematics and Computer Science (FB3) at the University of Bremen. This is one of the three focal points of computer science at FB3, the other two being Digital Media and Artificial Intelligence, Robotics & Cognition.
- This lecture is read jointly (and in turns) by Dieter Hutter, Christoph Lüth, and Jan Peleska.
- The choice of material in each semester reflects personal preferences.

Why bother with Quality, Safety, and Security ?



Ariane 5

- Ariane 5 exploded on its virgin flight (Ariane Flight 501) on 4.6.1996.

- How could that happen?

What Went Wrong With Ariane Flight 501?

(1) Self-destruction due to instability; ← The Accident
(2) Instability due to wrong steering movements (rudder);
(3) On-board computer tried to compensate for (assumed) wrong trajectory;
(4) Trajectory was calculated wrongly because own position was wrong;
(5) Own position was wrong because positioning system had crashed;
(6) Positioning system had crashed because transmission of sensor data to ground control failed with integer overflow; ← The root cause
(7) Integer overflow occurred because values were too high;
(8) Values were too high because positioning system was integrated unchanged from predecessor model, Ariane-4;
(9) This assumption was not documented because it was satisfied tacitly with Ariane-4.
(10) Positioning system was redundant, but both systems failed (systematic error).
(11) Transmission of data to ground control also not necessary.

Railway Accident in Bad Aibling 2016

- Two trains collided on a single-track line close to Bad Aibling

- Human error ?
  - cf. Nancy Leveson: Engineering a Safer World

Recent Crashes of Boeing 737 MAX

- Lion Air flight JT 610 29.10.2018 06:33 near Jakarta
- Ethopian Airlines flight ET 302 10.03.2019 08:44 near Addis Ababa
- Accidents:
  - New planes in perfect weather fly into the ground.
- Causes:
  - Manoeuvring Characteristics Augmentation System (MCAS) automatically pushes down nose of aircrafts in risk of stall.
  - What happens when sensor readings are faulty?
  - MCAS can be switched off, but not permanently – warning lights and permanent switch off are premium features.
  - Pilots not trained with MCAS.
  - See here: https://www.bbc.com/news/world-africa-47553174
  - MCAS introduced for cost reasons.
- Accidents caused by push for low costs, poor user interface and sloppy certification process.
- See also: Air France flight AF 447

What is Safety and Security?

Safety:

- product achieves acceptable levels of risk or harm to people, business, software, property or the environment in a specified context of use
- Threats from "inside"
  - Avoid malfunction of a system
  - E.g. planes, cars, railways
- Threats from "outside"
  - Protect product against force majeure ("acts of god")
  - E.g. Lightening, storm, floods, earthquake, fatigue of material, loss of power

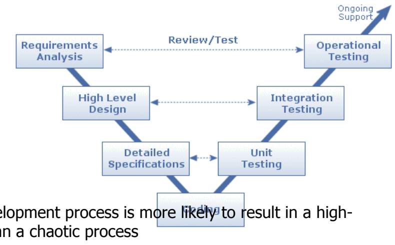
What is Safety and Security?

Security:

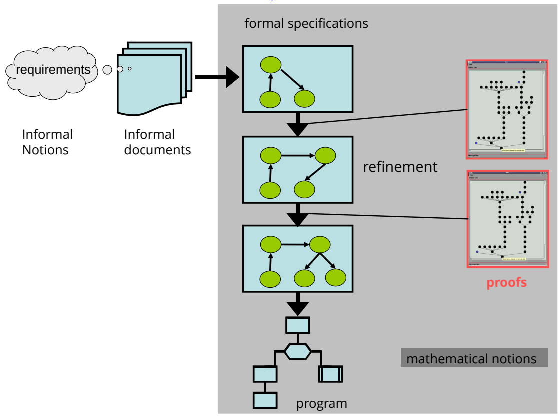
- Product is protected against potential attacks from people, environment etc.
- Threats from "outside"
  - Analyze and counteract the abilities of an attacker
- Threats from "inside"
  - Monitor activities of own personnel:
  - Selling of sensitive company data
  - Insertion of Trojans during HW/SW design
- In this context: "cybersecurity" (not physical security)

Software Development Models

- Definition of software development process and documents
- Examples:
  - Waterfall Model
  - V-Model
  - Model-Driven Architectures
  - Agile Development
- Motivation:
  - A well-defined development process is more likely to result in a high-quality product than a chaotic process
  - "Process quality ensures product quality"



Formal Software Development



Verification and Validation (V&V)

- **Verification**: have we built the system right?
  - i.e. correct with respect to a reference artefact
    - specification document
    - reference system
    - model

Korrektheit

- **Validation**: have we built the right system?
  - i.e. effective (or adequate) for its intended operation?

Wirksamkeit

V&V Methods

- **Testing**
  - Test case generation, black- vs. white box
  - Hardware-in-the-loop testing: integrated HW/SW system is tested
  - Software-in-the-loop testing: only software is tested
  - Program runs using symbolic values
- **Simulation**
  - An executable model is tested with respect to specific properties
  - This is also called Model-in-the-Loop Test
- Static/dynamic **program analysis**
  - Dependency graphs, flow analysis
  - Symbolic evaluation
- **Model checking**
  - Automatic proof by reduction to finite state problem
- **Formal Verification**
  - Symbolic proof of program properties

Where are we?

- 01: Concepts of Quality
- 02: Legal Requirements: Norms and Standards
- 03: The Software Development Process
- 04: Hazard Analysis
- 05: High-Level Design with SysML
- 06: Formal Modelling with OCL
- 07: Testing
- 08: Static Program Analysis
- 09-10: Software Verification
- 11-12: Model Checking
- 13: Conclusions

Concepts of Quality

What is Quality?

- Quality is the collection of its characteristic properties
- Quality model: decomposes the high-level definition by associating attributes (also called characteristics, factors, or **criteria**) to the quality conception
- Quality **indicators** associate **metric values** with **quality criteria**, expressing "how well" the criteria have been fulfilled by the process or product.
  - The idea is that to **measure** quality, with the aim of continuously **improving** it.
  - Leads to **quality management** (TQM, Kaizen)

Quality Criteria: Different „Dimensions" of Quality

- For the development of artifacts quality criteria can be measured with respect to the
  - development process (**process quality**)
  - final product (**product quality**)
- Another dimension for structuring quality conceptions is
  - **Correctness**: the consistency with the product and its associated requirements specifications
  - **Effectiveness**: the suitability of the product for its intended purpose

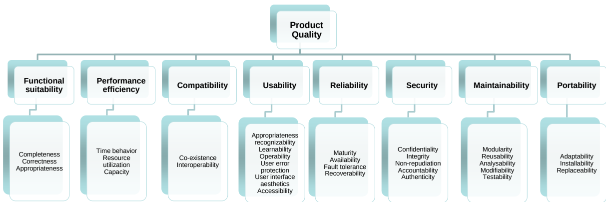
Quality Criteria (cont.)

- A third dimension structures quality according to product properties:
  - **Functional properties**: the specified services to be delivered to the users
  - **Structural properties**: architecture, interfaces, deployment, control structures
  - **Non-functional properties**: usability, safety, reliability, availability, security, maintainability, guaranteed worst-case execution time (WCET), costs, absence of run-time errors, …

Quality (ISO/IEC 25010/12)

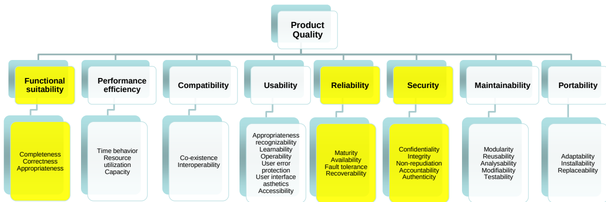
- "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models"
  - Quality model framework (replaces the older ISO/IEC 9126)
- Product quality model
  - Categorizes system/software product quality properties
- Quality in use model
  - Defines characteristics related to outcomes of interaction with a system
  - Also known as „end user experience" („UX")
- Quality of data model
  - Categorizes data quality attributes

Product Quality Model



Source: ISO/IEC FDIS 25010

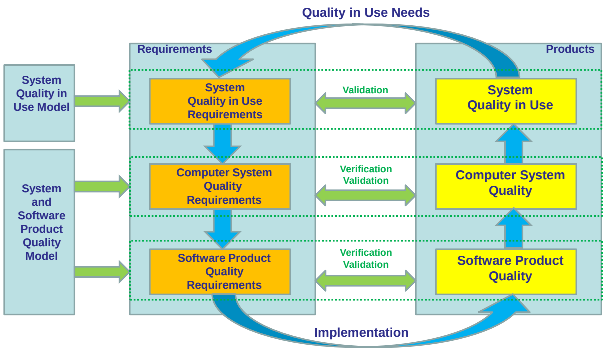
Our Focus of Interest



How can we „guarantee" safety and security ?

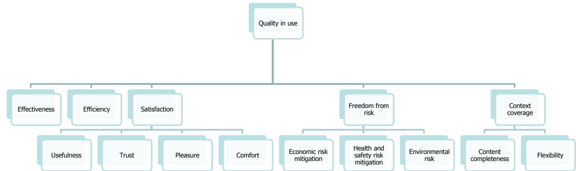
Source: ISO/IEC FDIS 25010

System Quality Life Cycle Model



Source: ISO/IEC FDIS 25010

Quality in Use Model



Other Norms and Standards

- ISO 9001 (DIN ISO 9000-4):
  - Standardizes definition and supporting principles necessary for a quality system to ensure products meet requirements
  - "Meta-Standard"
- CMM (Capability Maturity Model), Spice (ISO 15504)
  - Standardizes maturity of development process
  - Level 1 (initial): Ad-hoc
  - Level 2 (repeatable): process dependent on individuals
  - Level 3 (defined): process defined & institutionalized
  - Level 4 (managed): measured process
  - Level 5 (optimizing): improvement feed back into process

Summary

- Quality
  - collection of characteristic properties
  - quality indicators measuring quality criteria
- Relevant aspects of quality here
  - Functional suitability
  - Reliability
  - Security
- Next week
  - Concepts of Safety, Legal Requirements, Certification

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 02:
Legal Requirements - Norms and Standards

Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

Why Bother with Norms?

Why bother with norms?

If you want (or need to) to write safety-criticial software
then you need to adhere to state-of-the-art practice
as encoded by the relevant norms & standards.

- The **bad** news:
  - As a qualified professional, you may become **personally liable** if you deliberately and intentionally (*grob vorsätzlich*) disregard the state of the art or do not comply to the rules (= norms, standards) that were to be applied.
- The **good** news:
  - Pay attention here and you will be delivered from these evils.
- Caution: applies to all kinds of software.

Because in case of failure…

- Whose fault is it? Who pays for it? ("Produkthaftung")
  - European practice: extensive regulation
  - American practice: judicial mitigation (lawsuits)
- Standards often put a lot of emphasis on process and traceability (**auditable evidence**). Who decided to do what, why, and how?
- What are norms relevant to safety and security?
  Examples:
    - Safety: IEC 61508 – Functional safety
      - specialised norms for special domains
    - Security: IEC 15408 – Common criteria
      - In this context: "cybersecurity", not "guns and gates"
- What is regulated by such norms?

Emergent Properties

- An **emergent property** of a system is one that cannot be attributed to a single system component, but results from the overall effect of system components inter-operating with each other and the environment
- **Safety and Security are emergent properties.**
  - They can only be analyzed in the context of the complete system and its environment
  - Safety and security can never be derived from the properties of a single component, in particular, never from that of a software component alone

What is Safety?

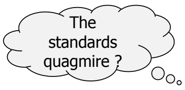
- Absolute definition:
  - „Safety is freedom from accidents or losses."
    Nancy Leveson, „Safeware: System safety and computers"
- But is there such a thing as absolute safety?
- Technical definition:
  - „Sicherheit: Freiheit von *unvertretbaren* Risiken"
    - IEC 61508-4:2001, §3.1.8

Legal Grounds

- The **machinery directive**: *The Directive 2006/42/EC of the European Parliament and of the Council of 17 May 2006 on machinery, and amending Directive 95/16/EC (recast)*
- Scope:
  - Machineries (with a **drive system** and **movable parts**)
- Objective:
  - **Market harmonization** (not safety)
- Structure:
  - Sequence of whereas clauses (explanatory)
  - followed by 29 articles (main body)
  - and 12 subsequent annexes (detailed information about particular fields, e.g. health & safety)
- Some application areas have their **own regulations**:
  - Cars and motorcycles, railways, planes, nuclear plants …

The Norms and Standards Landscape

- First-tier standards (A-Normen)
  - General, widely applicable, no specific area of application
  - Example: IEC 61508
- Second-tier standards (B-Normen)
  - Restriction to a particular area of application
  - Example: ISO 26262 (IEC 61508 for automotive)
- Third-tier standards (C-Normen)
  - Specific pieces of equipment
  - Example: IEC 61496-3 ("Berührungslos wirkende Schutzeinrichtungen")
- Always use most specific norm.



Norms for the Working Programmer

- IEC 61508:
  - *"Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES)"*
  - Widely applicable, general, considered hard to understand
- ISO 26262
  - Specialisation of 61508 to cars (automotive industry)
- DIN EN 50128:2011
  - Specialisation of 61508 to software for railway industry
- RTCA DO 178-B and C (new developments require C):
  - "*Software Considerations in Airborne Systems and Equipment Certification*"
  - Airplanes, NASA/ESA
- ISO 15408:
  - "*Common Criteria for Information Technology Security Evaluation"*
  - Security, evolved from TCSEC (US), ITSEC (EU), CTCPEC (Canada)

Functional Safety: IEC 61508 and friends

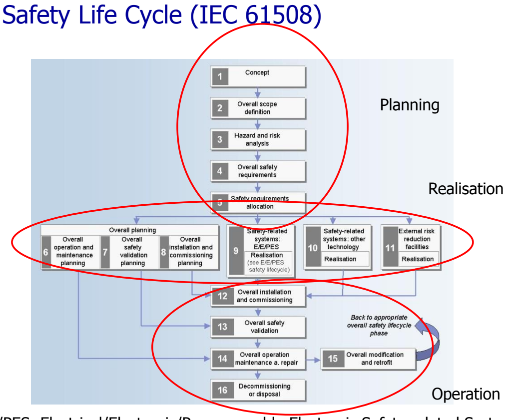
What is regulated by IEC 61508?

1. Risk analysis determines the safety integrity level (SIL).
2. Hazard analysis leads to safety requirement specification.
3. Safety requirements must be satisfied by product:
   - Need to verify that this is achieved.
   - SIL determines amount of testing/proving etc.
4. Life-cycle needs to be managed and organised:
   - Planning: verification & validation plan.
   - Note: personnel needs to be qualified.
5. All of this needs to be independently assessed.
   - SIL determines independence of assessment body.

The Seven Parts of IEC 61508

1. General requirements
2. Requirements for E/E/PES safety-related systems
   - Hardware rather than software
3. **Software requirements**
4. Definitions and abbreviations
5. Examples of methods for the determination of safety-integrity levels
   - *Mostly informative*
6. Guidelines on the application of Part 2 and 3
   - *Mostly informative*
7. Overview of techniques and measures

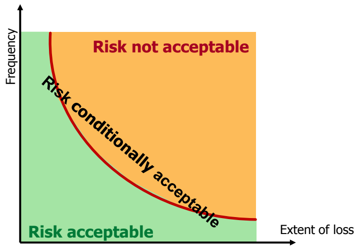
The Safety Life Cycle (IEC 61508)



E/E/PES: Electrical/Electronic/Programmable Electronic Safety-related Systems

Safety Integrity Levels

- What is the risk by operating a system?
- Two factors:
  - How likely is a failure ?
  - What is the damage caused by a failure?



Safety Integrity Levels

- Maximum average probabilty of a **dangerous failure** (per hour/per demand) depending on how often it is used:

SIL	High Demand (more than once a year)	Low Demand (once a year or less)
4	$10^{-9} < P/hr < 10^{-8}$	$10^{-5} < P < 10^{-4}$
3	$10^{-8} < P/hr < 10^{-7}$	$10^{-4} < P < 10^{-3}$
2	$10^{-7} < P/hr < 10^{-6}$	$10^{-3} < P < 10^{-2}$
1	$10^{-6} < P/hr < 10^{-5}$	$10^{-2} < P < 10^{-1}$

- Examples:
  - High demand: car brakes
  - Low demand: airbag control
- Note: SIL only meaningful for **specific safety functions**.

Establishing target SIL (Quantitative)

- IEC 61508 does not describe standard procedure to establish a SIL target, it allows for alternatives.
- Quantitative approach
  - Start with target risk level
  - Factor in fatality and frequency

Maximum tolerable risk of fatality	Individual risk (per annum)
Employee	10^{-4}
Public	10^{-5}
Broadly acceptable („Negligible“)	10^{-6}

- Example: Safety system for a chemical plant
  - Max. tolerable risk exposure: $A=10^{-6}$ (per annum)
  - Ratio of hazardous events leading to fatality: $B= 10^{-2}$
  - Risk of failure of unprotected process: C= 1/5 per annum (ie. 1 in 5 years)
  - Risk of hazardous event, unprotected: $B*C= 2*10^{-3}$ (ie. 1 in 5000 years)
  - Risk of hazardous event, protected A = E*B*C (with E *failure on demand*)
  - Calculate E as $E = A/(B*C) = 5*10^{-4}$, so SIL 3
- More examples: airbag, safety system for a hydraulic press

Establishing target SIL (Quantitative)

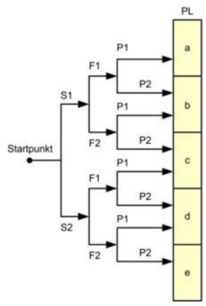
- Example: Safety system for a hydraulic press
  - Max. tolerable risk exposure: $A=10^{-4}$ per annum, i.e. $A'= 10^{-8}$ per hour
  - Ratio of hazardous events leading to serious injury: B= 1/100
    - Worker will not willfully put his hands into the press
  - Risk of failure of unprotected process: C= 50 per hour
    - Press operates
  - Risk of hazardous event, unprotected: B*C= 1/2 per hour
  - $E = A'/(B*C) = 2*10^{-8}$, so SIL 3
- Example: Domestic appliance, e.g. heating iron
  - Overheating may cause fire
  - Max. tolerable risk exposure: $A=10^{-5}$ per annum, i.e. $A'= 10^{-9}$ per hour
  - Study suggests 1 in 400 incidents leads to fatality, i.e. B*C= 1/400
  - Then $E= A'/B*C = 10^{-9}*400 = 4*10^{-7}$, so SIL 3

Establishing Target SIL (Qualitative)

- Qualitative method: risk graph analysis (e.g. DIN 13849)
- DIN EN ISO 13849:1 determines the **performance level**

PL	SIL
a	-
b	1
c	2
d	3
e	4

Relation PL to SIL



Source: Peter Wratil (Wikipedia)

Severity of injury:
S1 - slight (reversible) injury
S2 – severe (irreversible) injury

Occurrence:
F1 – rare occurrence
F2 – frequent occurrence

Possible avoidance:
P1 – possible
P2 – impossible

Numerical Characteristics

- The standard IEC 61508 defines the following numerical characteristics per safety integrity level:
  - PFD, average probability of failure to perform its design function on demand (average probability of dangerous failure on demand of the safety function), i.e. the probability of unavailability of the safety function leading to dangerous consequences
  - PFH, the probability of a dangerous failure per hour (average frequency of dangerous failure of the safety function)
- Failure on demand = "function fails when it is needed"

What does the SIL mean for the development process?

- In general:
  - „Competent“ personnel
  - Independent assessment („four eyes“)
- SIL 1:
  - Basic quality assurance (e.g. ISO 9001)
- SIL 2:
  - Safety-directed quality assurance, more tests
- SIL 3:
  - Exhaustive testing, possibly formal methods
  - Assessment by separate department
- SIL 4:
  - State-of-the-art practices, formal methods
  - Assessment by separate organization

Some Terminology

- Error handling:
  - Fail-safe (or fail-stop): terminate in a safe state
  - Fail-operational systems: continue operation, even if controllers fail
  - Fault-tolerant systems: continue with a potentially degraded service (more general than fail operational systems)
- Safety-critical, safety-relevant (*sicherheitskritisch*)
  - General term -- failure may lead to risk
- Safety function (*Sicherheitsfunktion*)
  - Technical term, that functionality which ensures safety
- Safety-related (*sicherheitsgerichtet, sicherheitsbezogen*)
  - Technical term, directly related to the safety function

Increasing SIL by redudancy

- One can achieve a higher SIL by combining **independent** systems with lower SIL („*Mehrkanalsysteme*“).
- Given two systems A, B with failure probabilities P_A, P_B, the chance for failure of both is (with P_{CC} probablity of common-cause failures):

$$P_{AB} = P_{CC} + P_A P_B$$

- Hence, combining two SIL 3 systems **may** give you a SIL 4 system.
- However, be aware of **systematic** errors (and note that IEC 61508 considers all software errors to be systematic).
- Note also that for fail-operational systems you need three (not two) systems.
- The degree of independence can be increased by software diversity: channels are equipped with software following the same specification but developed by independent teams

The Software Development Process

- 61508 in principle allows any software lifecycle model, but:
  - No specific process model is given, illustrations use a V-model, and no other process model is mentioned.
- Appx A, B give normative guidance on measures to apply:
  - Error detection needs to be taken into account (e.g. runtime assertions, error detection codes, dynamic supervision of data/control flow)
  - Use of strongly typed programming languages (see table)
  - Discouraged use of certain features:
    - recursion(!), dynamic memory, unrestricted pointers, unconditional jumps
  - Certified tools and compilers must be used or tools "proven in use".

Proven in Use: Statistical Evaluation

- As an alternative to systematic development, statistics about usage may be employed. This is particularly relevant:
  - for development tools (compilers, verification tools etc),
  - and for re-used software (modules, libraries).
- The norm (61508-7 Appx. D) is quite brief about this subject. It states these methods should only be applied by those "competent in statistical analysis".
- The problem: proper statistical analysis is more than just "plugging in numbers".
  - Previous use needs to be to the same specification as intended use (eg. compiler: same target platform).
  - Uniform distribution of test data, indendent tests.
  - Perfect detection of failure.

Proven in Use: Statistical Evaluation

- Statistical statements can only be given with respect to a confidence level ($\lambda = 1 - p$), usually $\lambda = 0.99$ or $\lambda = 0.9$.
- With this and all other assumptions satisfied, we get the following numbers from the norm:
  - For on-demand: observed demands without failure (P_1: accepted probability of failure to perform per demand)
  - For continuously-operated: observed hours w/o failure (P_2: accepted probability of failure to perform per hour of operation)

SIL	On-Demand			Continuously Operated		
	P_1	$\lambda = 99\%$	$\lambda = 90\%$	P_2	$\lambda = 99\%$	$\lambda = 90\%$
1	$< 10^{-1}$	46	3	$< 10^{-5}$	$4.6 \cdot 10^5$	$3 \cdot 10^5$
2	$< 10^{-2}$	460	30	$< 10^{-6}$	$4.6 \cdot 10^6$	$3 \cdot 10^6$
3	$< 10^{-3}$	4600	3000	$< 10^{-7}$	$4.6 \cdot 10^7$	$3 \cdot 10^7$
4	$< 10^{-4}$	46000	30000	$< 10^{-8}$	$4.6 \cdot 10^8$	$3 \cdot 10^8$

Source: Ladkin, Littlewood: Practical Statistical Evaluation of Critical Software.

Table A.2 - Software Architecture

Tabelle A.2 – Softwareentwurf und Softwareentwicklung: Entwurf der Software-Architektur (siehe 7.4.3)

Verfahren/Maßnahme *	siehe	SIL1	SIL2	SIL3	SIL4
1 Fehlererkennung und Diagnose	C.3.1	o	+	++	++
2 Fehlererkennende und -korrigierende Codes	C.3.2	+	+	+	++
3a Plausibilitätskontrollen (Failure assertion programming)	C.3.3	+	+	+	++
3b Externe Überwachungseinrichtungen	C.3.4	o	+	+	+
3c Diversitäre Programmierung	C.3.5	+	+	+	++
3d Regenerationsblöcke	C.3.6	+	+	+	+
3e Rückwärtsregeneration	C.3.7	+	+	+	+
3f Vorwärtsregeneration	C.3.8	+	+	+	+
3g Regeneration durch Wiederholung	C.3.9	+	+	+	++
3h Aufzeichnung ausgeführter Abschnitte	C.3.10	o	+	+	++
4 Abgestufte Funktionseinschränkungen	C.3.11	+	+	++	++
5 Künstliche Intelligenz – Fehlerkorrektur	C.3.12	o	--	--	--
6 Dynamische Rekonfiguration	C.3.13	o	--	--	--
7a Strukturierte Methoden mit z. B. JSD, MASCOT, SADT und Yourdon.	C.2.1	++	++	++	++
7b Semi-formale Methoden	Tabelle B.7	+	+	++	++
7c Formale Methoden z. B. CCS, CSP, HOL, LOTOS, OBJ, temporäre Logik, VDM und Z	C.2.4	o	+	+	++

Table A.4 - Software Design & Development

Tabelle A.4 – Softwareentwurf und Softwareentwicklung: detaillierter Entwurf (siehe 7.4.5 and 7.4.6)
(Dies beinhaltet Software-Systementwurf, Entwurf der Softwaremodule und Codierung)

Verfahren/Maßnahme *	siehe	SIL1	SIL2	SIL3	SIL4
1a Strukturierte Methoden wie z. B. JSD, MASCOT, SADT und Yourdon	C.2.1	++	++	++	++
1b Semi-formale Methoden	Tabelle B.7	+	++	++	++
1c Formale Methoden wie z. B. CCS, CSP, HOL, LOTOS, OBJ, temporäre Logik, VDM und Z	C.2.4	o	+	+	++
2 Rechnergestützte Entwurfswerkzeuge	B.3.5	+	+	++	++
3 Defensive Programmierung	C.2.5	o	+	++	++
4 Modularisierung	Tabelle B.9	++	++	++	++
5 Entwurfs- und Codierungs-Richtlinien	Tabelle B.1	+	++	++	++
6 Strukturierte Programmierung	C.2.7	++	++	++	++

Table A.9 – Software Verification

Tabelle A.9 – Software-Verifikation (siehe 7.9)

Verfahren/Maßnahme *	siehe	SIL1	SIL2	SIL3	SIL4
1 Formaler Beweis	C.5.13	o	+	+	++
2 Statistische Tests	C.5.1	o	+	+	++
3 Statische Analyse	B.6.4 Tabelle B.8	+	++	++	++
4 Dynamische Analyse und Test	B.6.5 Tabelle B.2	+	++	++	++
5 Software-Komplexitätsmetriken	C.5.14	+	+	+	+

Table B.1 – Coding Guidelines

- Table C.1, programming languages, mentions:
  - ADA, Modula-2, Pascal, FORTRAN 77, C, PL/M, Assembler, …
- Example for a guideline:
  - MISRA-C: 2004, Guidelines for the use of the C language in critical systems.

Tabelle B.1 – Entwurfs- und Codierungs-Richtlinien (Verweisungen aus Tabelle A.4)

Verfahren/Maßnahme *	siehe	SIL1	SIL2	SIL3	SIL4
1 Verwendung von Codierungs-Richtlinien	C.2.6.2	++	++	++	++
2 Keine dynamischen Objekte	C.2.6.3	+	++	++	++
3a Keine dynamischen Variablen	C.2.6.3	o	+	++	++
3b Online-Test der Erzeugung von dynamischen Variablen	C.2.6.4	o	+	++	++
4 Eingeschränkte Verwendung von Interrupts	C.2.6.5	+	+	++	++
5 Eingeschränkte Verwendung von Pointern	C.2.6.6	o	+	++	++
6 Eingeschränkte Verwendung von Rekursionen	C.2.6.7	o	+	++	++
7 Keine unbedingten Sprünge in Programmen in höherer Programmiersprache	C.2.6.2	+	++	++	++

ANMERKUNG 1 Die Maßnahmen 2 und 3a brauchen nicht angewendet zu werden, wenn ein Compiler verwendet wird, der sicherstellt, dass genügend Speicherplatz für alle dynamischen Variablen und Objekte vor der Laufzeit zugeteilt wird, oder der Laufzeittests zur korrekten Online-Zuweisung von Speicherplatz einfügt.

* Es müssen dem Sicherheits-Integritätslevel angemessene Verfahren/Maßnahmen ausgewählt werden. Alternative oder gleichwertige Verfahren/Maßnahmen sind durch einen Buchstaben hinter der Nummer gekennzeichnet. Es muss nur eine(s) der alternativen oder gleichwertigen Verfahren/Maßnahmen erfüllt werden.

Table B.5 - Modelling

Tabelle B.5 – Modellierung (Verweisung aus der Tabelle A.7)

Verfahren/Maßnahme *	siehe	SIL1	SIL2	SIL3	SIL4
1 Datenflussdiagramme	C.2.2	+	+	+	+
2 Zustandsübergangsdiagramme	B.2.3.2	o	+	++	++
3 Formale Methoden	C.2.4	o	+	+	++
4 Modellierung der Leistungsfähigkeit	C.5.20	+	++	++	++
5 Petri-Netze	B.2.3.3	o	+	++	++
6 Prototypenerstellung/Animation	C.5.17	+	+	+	+
7 Strukturdiagramme	C.2.3	+	+	+	++

ANMERKUNG Sollte eine spezielles Verfahren in dieser Tabelle nicht vorkommen, darf nicht angenommen werden, dass dieses nicht in Betracht gezogen werden darf. Es sollte zu dieser Norm in Einklang stehen.

* Es müssen dem Sicherheits-Integritätslevel angemessene Verfahren/Maßnahmen ausgewählt werden.

Certification

- Certification is the process of showing **conformance** to a **standard**.
  - Also sometimes (e.g. DO-178B) called `qualification‘.
- Conformance to IEC 61508 can be shown in two ways:
  - either that an organization (company) has in principle the ability to produce a product conforming to the standard,
  - or that a specific product (or system design) conforms to the standard.
- Certification can be done by the developing company (self-certification), but is typically done by an **notified body** ("benannte Stellen").
  - In Germany, e.g. the TÜVs or *Berufsgenossenschaften*;
  - In Britain, professional role (ISA) supported by IET/BCS;
  - Aircraft certification in Europe: EASA (European Aviation Safety Agency)
  - Aircraft certification in US: FAA (Federal Aviation Administration)

Security:
IEC 15408 - The Common Criteria

Recall: Security Criteria

- Confidentiality
- Integrity
- Availability
- Authenticity
- Accountability
- Non-repudiation

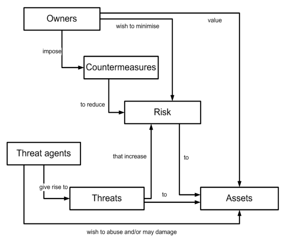
Common Criteria (IEC 15408)

- Established in 1996 as a harmonization of various norms to evaluate security properties of IT products and systems (e.g. ITSEC (Europe), TCSEC (US, "orange book"), CTCPEC (Canada))
- Basis for evaluation of **security properties** of IT products (or parts of) and systems (the **Target of Evaluation** TOE).
- The CC is useful as a guide for the development of products or systems with IT security functions and for the procurement of commercial products and systems with such functions.

Common Criteria

General Model

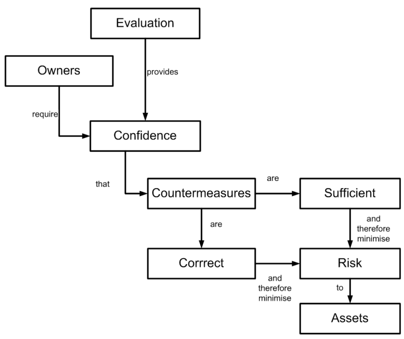
- Security is concerned with the protection of assets. Assets are entities that someone places value upon.
- Threats give rise to risks to the assets, based on the likelihood of a threat being realized and its impact on the assets
- (IT and non-IT) Counter-measures are imposed to reduce the risks to assets.



Security Goals

- Protection of information from unauthorized disclosure, modification, or loss of use:
  - **confidentiality**, **integrity**, and **availability**
  - may also be applicable to aspects
- Focus on **threats** to that information arising from human activities, whether malicious or otherwise, but may be applicable to some non-human threats as well.
- In addition, the CC may be applied in other areas of IT, but makes no claim of competence outside the strict domain of IT security.

Concept of Evaluation



Security Environment

- Laws, organizational security policies, customs, expertise and knowledge relevant for TOE
  - Context in which the TOE is intended to be used.
  - Threats to security that are, or are held to be, present in the environment.
- A statement of applicable organizational security policies would identify relevant policies and rules.
- Assumptions about the environment of the TOE are considered as axiomatic for the TOE evaluation.

Security Objectives

- Identification of all of the security concerns
  - Aspects addressed directly by the TOE or by its environment.
  - Incorporating engineering judgment, security policy, economic factors and risk acceptance decisions.
- Analysis of the security environment results in security objectives that counter the identified threats and address identified organizational security policies and assumptions.
- The security objectives for the environment would be implemented within the IT domain, and by non-technical or procedural means.
- Only the security objectives for the TOE and its IT environment are addressed by IT security requirements

Threats and Their Risks

- **Threats** to security of the assets relevant to the TOE.
  - in terms of a threat agent,
  - a presumed attack method,
  - any vulnerabilities that are the foundation for the attack, and
  - identification of the asset under attack.
- **Risks** to security. Assess each threat
  - by its likelihood developing into an actual attack,
  - its likelihood proving successful, and
  - the consequences of any damage that may result.

Security Requirements

- Refinement of security objectives into
  - Requirements for TOE and
  - Requirements for the environment
- **Functional requirements**
  - Functions in support for security of IT-system
  - E.g. identification & authentication, cryptography,…
- **Assurance Requirements**
  - Establishing confidence in security functions
  - Correctness of implementation
  - E.g. development, life cycle support, testing, …

Security Functions

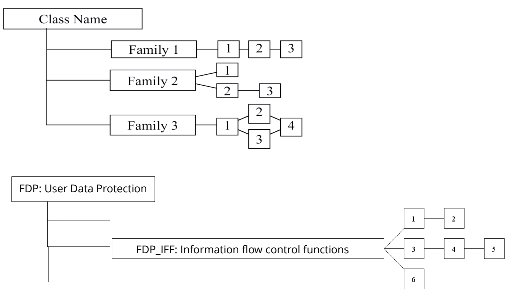
- The **statement of TOE security functions** shall cover the IT security functions and shall specify how these functions satisfy the TOE security functional requirements. This statement shall include a bi-directional mapping between functions and requirements that clearly shows which functions satisfy which requirements and that all requirements are met.
- Starting point for **design process**.

Security Functional Components

- Class FAU: Security audit
- Class FCO: Communication
- Class FCS: Cryptographic support
- **Class FDP: User data protection**
- Class FIA: Identification and authentication
- Class FMT: Security management
- Class FPR: Privacy
- Class FPT: Protection of the TSF
- Class FRU: Resource utilisation
- Class FTA: TOE access
- Class FTP: Trusted path/channels

Security Functional Components

- Content and presentation of the functional requirements



FDP – Information Flow Control

FDP_IFC.1 Subset information flow control

Hierarchical to: No other components.

Dependencies: FDP_IFF.1 Simple security attributes

FDP_IFC.1.1 The TSF shall enforce the [assignment: *information flow control SFP*] on [assignment: *list of subjects, information, and operations that cause controlled information to flow to and from controlled subjects covered by the SFP*].

FDP_IFC.2 Complete information flow control

Hierarchical to: FDP_IFC.1 Subset information flow control

Dependencies: FDP_IFF.1 Simple security attributes

FDP_IFC.2.1 The TSF shall enforce the [assignment: *information flow control SFP*] on [assignment: *list of subjects and information*] and all operations that cause that information to flow to and from subjects covered by the SFP**.**

FDP_IFC.2.2 The TSF shall ensure that all operations that cause any information in the TOE to flow to and from any subject in the TOE are covered by an information flow control SFP.

Assurance Requirements

Assurance Approach

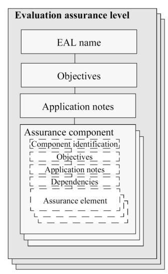
"The CC philosophy is to provide assurance based upon an evaluation (active investigation) of the IT product that is to be trusted. Evaluation has been the traditional means of providing assurance and is the basis for prior evaluation criteria documents. "

CC, Part 3, p.15

Assurance Requirements

- Concerning actions of the developer, evidence produced and actions of the evaluator.
- Examples:
  - Rigor of the **development process**
  - Search for and analysis of the impact of potential security vulnerabilities.
- **Degree of assurance**
  - varies for a given set of functional requirements
  - typically expressed in terms of increasing levels of rigor built with assurance components.
- **Evaluation assurance levels** (EALs) constructed using these components.

Part 3 Assurance levels



Assurance Components

- Class APE: Protection Profile evaluation
- Class ASE: Security Target evaluation
- Class ADV: Development
- Class AGD: Guidance documents
- Class ALC: Life-cycle support
- Class ATE: Tests
- Class AVA: Vulnerability assessment
- Class ACO: Composition

Evaluation Assurance Level

- EALs define levels of assurance (no guarantees)

1. Functionally tested
2. Structurally tested
3. Methodically tested and checked
4. Methodically designed, tested, and reviewed
5. Semi-formally designed and tested
6. Semi-formally verified design and tested
7. Formally verified design and tested

EAL5 – EAL7 require formal methods

Assurance class	Assurance Family	Assurance Components by Evaluation Assurance Level						
		EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
Development	ADV_ARC		**1**	1	1	1	1	1
	ADV_FSP	**1**	**2**	**3**	**4**	**5**	5	**6**
	ADV_IMP				**1**	1	**2**	2
	ADV_INT					**2**	**3**	3
	ADV_SPM						**1**	1
	ADV_TDS		**1**	**2**	**3**	**4**	**5**	**6**
Guidance documents	AGD_OPE	**1**	1	1	1	1	1	1
	AGD_PRE	**1**	1	1	1	1	1	1
Life-cycle support	ALC_CMC	**1**	**2**	**3**	**4**	4	**5**	5
	ALC_CMS	**1**	**2**	**3**	**4**	**5**	5	5
	ALC_DEL		**1**	1	1	1	1	1
	ALC_DVS			**1**	1	1	**2**	2
	ALC_FLR							
	ALC_LCD			**1**	1	1	1	**2**
	ALC_TAT				**1**	**2**	**3**	3
Security Target evaluation	ASE_CCL	**1**	1	1	1	1	1	1
	ASE_ECD	**1**	1	1	1	1	1	1
	ASE_INT	**1**	1	1	1	1	1	1
	ASE_OBJ	**1**	**2**	2	2	2	2	2
	ASE_REQ	**1**	**2**	2	2	2	2	2
	ASE_SPD		**1**	1	1	1	1	1
	ASE_TSS	**1**	1	1	1	1	1	1
Tests	ATE_COV		**1**	**2**	2	2	**3**	3
	ATE_DPT			**1**	1	**3**	3	**4**
	ATE_FUN		**1**	1	1	1	**2**	2
	ATE_IND	**1**	**2**	2	2	2	2	**3**
Vulnerability assessment	AVA_VAN	**1**	**2**	2	**3**	**4**	**5**	5

Assurance Components
Example: Development

ADV_FSP.1 Basic functional specification

EAL-1: … The functional specification shall describe the purpose and method of use for each SFR-enforcing and SFR-supporting TSFI.

EAL-2: … The functional specification shall completely represent the TSF.

EAL-3: + … The functional specification shall summarize the SFR-supporting and SFR-non-interfering actions associated with each TSFI.

EAL-4: + … The functional specification shall describe all direct error messages that may result from an invocation of each TSFI.

EAL-5: … The functional specification shall describe the TSFI using a semi-formal style.

EAL-6: … The developer shall provide a formal presentation of the functional specification of the TSF. The formal presentation of the functional specification of the TSF shall describe the TSFI using a formal style, supported by informal, explanatory text where appropriate.

Degree of Assurance

(TSFI : Interface of the TOE Security Functionality (TSF), SFR : Security Functional Requirement)

Conclusion

Summary

- Norms and standards enforce the application of the state-of-the-art when developing software which is **safety-critical** or **security-critical.**
- Wanton disregard of these norms may lead to **personal liability.**
- Norms typically place a lot of emphasis on **process.**
- Key question are traceability of decisions and design, and verification and validation.
- Different application fields have different norms:
  - IEC 61508 and its specializations, e.g. DO-178B.
  - IEC 15408 („Common Criteria")

Further Reading

- Terminology for dependable systems:
  - J. C. Laprie *et al.*: Dependability: Basic Concepts and Terminology. Springer-Verlag, Berlin Heidelberg New York (1992).
- Literature on safety-critical systems:
  - Storey, Neil: Safety-Critical Computer Systems. Addison Wesley Longman (1996).
  - Nancy Levenson: Safeware – System Safety and Computers. Addison-Wesley (1995).
- A readable introduction to IEC 61508:
  - David Smith and Kenneth Simpson: Functional Safety. 2nd Edition, Elsevier (2004).

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 3:
The Software Development Process

Christoph Lüth, Dieter Hutter, Jan Peleska

Organisatorisches

- Die Übung am Donnerstag, 31.10.2019, fällt aus (Reformationstag).
- Nächste Übung am Dienstag, 05.11.2019.

Where are we?

Software Development Models

Software Development Process

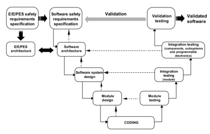
- A software development process is the **structure** imposed on the development of a software product.
- We classify processes according to **models** which specify
  - the artefacts of the development, such as
    - the software product itself, specifications, test documents, reports, reviews, proofs, plans etc;
  - the different stages of the development;
  - and the artefacts associated to each stage.
- Different models have a different focus:
  - Correctness, development time, flexibility.
- What does quality mean in this context?
  - What is the **output**? Just the software product, or more? (specifications, test runs, documents, proofs…)

Artefacts in the Development Process

Planning:
- Document plan
- V&V plan
- QM plan
- Test plan
- Project manual

Specifications:
- Requirements
- System specification
- Module specification
- User documents

Implementation:
- Source code
- Models
- Documentation

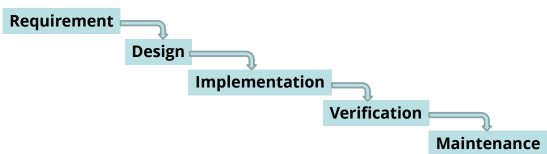


Verification & validation:
- Code review protocols
- Test cases, procedures, and test results
- Proofs

Possible formats:
- Documents:
  - Word documents
  - Excel sheets
  - Wiki text
  - Database (Doors)
- Models:
  - UML/SysML diagrams
  - Formal languages: Z, HOL, etc.
  - Matlab/Simulink or similar diagrams
- Source code

Waterfall Model (Royce 1970)

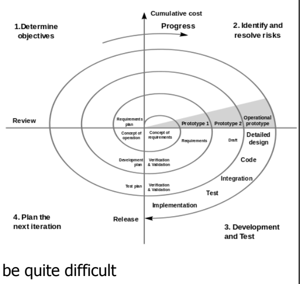
- Classical top-down sequential workflow with strictly separated phases.



- Unpractical as an actual workflow (no feedback between phases), but even the original paper did **not** really suggest this.

Spiral Model (Böhm 1986)

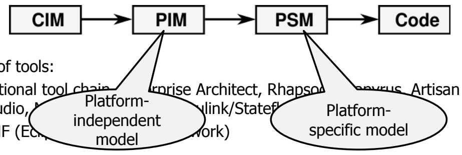
- Incremental development guided by **risk factors**
- Four phases:
  - Determine objectives
  - Analyse risks
  - Development and test
  - Review, plan next iteration
- See e.g.
  - Rational Unified Process (RUP)



- Drawbacks:
  - Risk identification is the key, and can be quite difficult

Model-Driven Development (MDD, MDE)

- Describe problems on abstract level using *a modeling language* (often a *domain-specific language*), and derive implementation by model transformation or run-time interpretation.
- Often used with UML (or its DSLs, eg. SysML)



- Variety of tools:
  - Rational tool chain, ...rprise Architect, Rhapso... ...Artisan Studio, ...link/Statef...
  - EMF (Ec... ...work)
- Strictly sequential development
- Drawbacks: high initial investment, limited, reverse engineering and change management (code changes to model changes) is complex

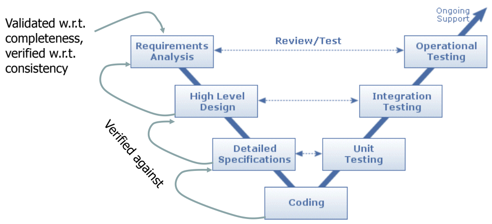
* Proprietary DSL – not related to UML

Agile Methods

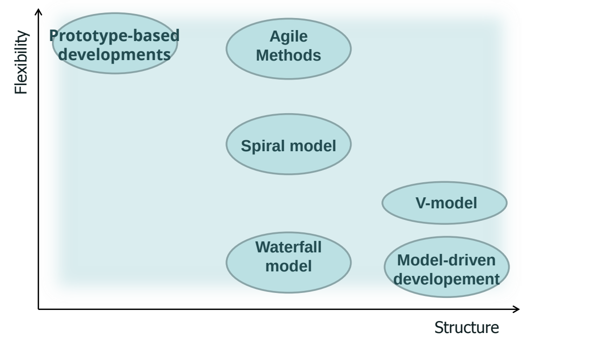
- Prototype-driven development
  - E.g. Rapid Application Development
  - Development as a sequence of prototypes
  - Ever-changing safety and security requirements
- Agile programming
  - E.g. Scrum, extreme programming
  - Development guided by functional requirements
  - Process structured by rules of conduct for developers
  - Rules capture best practice
  - Less support for non-functional requirements
- Test-driven development
  - Tests as *executable specifications:* write tests first
  - Often used together with the other two

V-Model

- Evolution of the waterfall model:
  - Each phase supported by corresponding verification & validation phase
  - Feedback between next and previous phase
- Standard model for public projects in Germany
  - … but also a general term for models of this „shape"
- Current: V-Modell XT („extreme tailoring")
  - Shape gives depencies, not development sequence



Software Development Models



from S. Paulus: Sichere Software

Development Models for Safety-Critical Systems

Development Models for Critical Systems

- Ensuring safety/security needs structure.
  - …but *too much* structure makes developments bureaucratic, which is *in itself* a safety risk.
  - Cautionary tale: Ariane-5
- Standards put emphasis on **process.**
  - Everything needs to be planned and documented.
  - Key issues: **auditability**, **accountability**, **traceability**.
- Best suited development models are variations of the V-model or spiral model.
- A new trend? V-Model XT allows variations of original V-model, e.g.:
  - V-Model for initial developments of a new product
  - Agile models (e.g. Scrum) for maintenance and product extensions

Auditability and Accountability

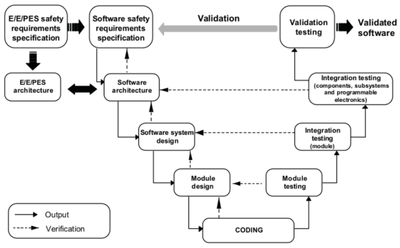
- Version control and configuration management is **mandatory** in safety-critical development (auditability).
- Keeping track of all artifacts contributing to a particular instance (**build**) of the system (**configuration**), and their **versions**.
- **Repository** keeps all artifacts in all versions.
  - Centralised: one repository vs. distributed (every developer keeps own repository)
  - General model: check out – modify – commit
  - Concurrency: enforced **lock**, or **merge** after commit.
- Well-known systems:
  - Commercial: ClearCase, Perforce, Bitkeeper…
  - Open Source: Subversion (centralised); Git, Mercurial (distributed)

Traceability

- The idea of being able to follow requirements (in particular, safety requirements) from requirement spec to the code (and possibly back).
- On the simplest level, an Excel sheet with (manual) links to the program.
- More sophisticated tools include DOORS:
  - Decompose requirements, hierarchical requirements
  - Two-way traceability: from code, test cases, test procedures, and test results back to requirements
  - E.g. DO-178B requires all code derives from requirements
- The SysML modelling language has traceability support:
  - Each model element can be traced to a requirement.
  - Special associations to express traceability relations.

Development Model in IEC 61508

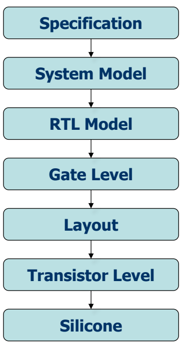
- IEC 61508 in principle allows any development model, but:
  - It requires safety-directed activities in each phase of the life cycle (safety life cycle, cf. last lecture).
  - Development is one part of the life cycle.
- The only development model mentioned is a V-model:



Development Model in DO-178B/C

- DO-178B/C defines different *processes* in the SW life cycle:
  - Planning process
  - Development process, structured in turn into
    - Requirements process
    - Design process
    - Coding process
    - Integration process
  - Verification process
  - Quality assurance process
  - Configuration management process
  - Certification liaison process
- There is no conspicuous diagram, but the Development Process has sub-processes suggesting the phases found in the V-model as well.
  - Implicit recommendation of the V-model.

Development Model for Hardware



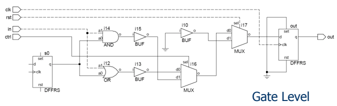

```
SC_MODULE(example) {
sc_in_clk clk;
sc_in<bool> rst, in, ctrl; sc_out<bool> out;
int o, s0;

void tick() {
  if (rst.read) o= 0;
  else if (!ctrl.read) o= s0 | in.read;
       else o= s0 & in.read;
  out.write(o); s0= o;
}
...
}
```

System-Model: SystemC

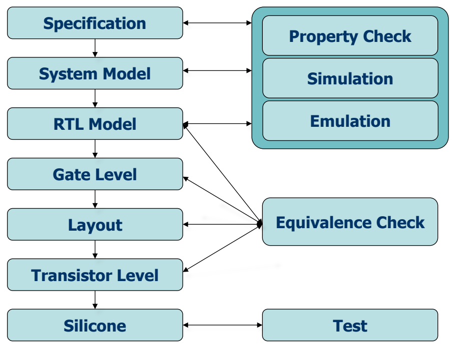
```
always @(posedge clk)
  if (rst) out <= 0;
  else
   if (! ctrl)  out <= s0 | in;
   else      out <= s0 & in;
```

Register-Transfer-Ebene: Verilog



Gate Level

Development Model for Hardware



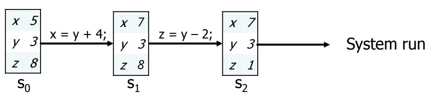
Basic Notions of Formal Software Development

Formal Software Development

- In a formal development, properties are stated in a rigorous way with a **precise mathematical semantics**.
- Formal specification requirements can be **proven**.
- **Advantages**:
  - Errors can be found early in the development process.
  - High degree of confidence into the system.
  - Recommend use of formal methods for high SILs/EALs.
- **Drawbacks**:
  - Requires a lot of effort and is thus expensive.
  - Requires qualified personnel (that would be *you*).
- There are tools which can help us by
  - finding (simple) proofs for us (model checkers), or
  - checking our (more complicated) proofs (theorem provers).

Formal Semantics

- **States** and transitions between them:



- **Operational semantics** describes relation between states and transitions:

$$\frac{s \vdash e \to n}{s \vdash x = e \;\to\; s[x/n]} \qquad \text{hence:} \qquad \frac{s_0 \vdash y+4 \to 7}{s_0 \vdash x = y+4 \;\to\; s_1}$$

- **Formal proofs**; e.g. proving

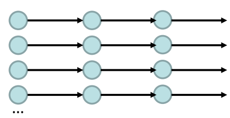
  x = y + 4; z = y – 2;

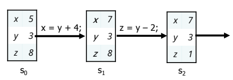
  yields the same final state as

  z = y - 2; x = y + 4;

Semantics of Programs and Requirements

- Set of all possible system runs





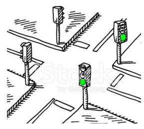
- **Requirements** related to safety and security:
  - Requirements on single states ?
  - Requirements on system runs ?
  - Requirements on sets of system runs ?

Alpern & Schneider
Clarkson & Schnei

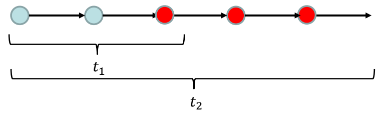
Some Notions

- Let b, t be two traces then
  $b \le t$ iff. $\exists t'. t = b \cdot t'$ i.e. b is a *finite* prefix of t
- A **property** is a set of infinite execution traces (like a program)
  - Trace t satisfies property P, written $t \vDash P$, iff $t \in P$
- A **hyperproperty** is a set of sets of infinite execution traces (like a set of programs)
  - A system (set of traces) S satisfies H iff S ∊ H
  - An observation Obs is a finite set of finite traces
  - $Obs \le S$ (Obs is a prefix of S) iff
    Obs is an observation and $\forall m \in Obs. \exists t \in S. m \le t$

Requirements on States: Safety Properties



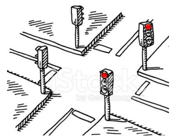
- Safety property *S*: „Nothing bad happens"
  - i.e. the system will never enter a *bad* state
  - E.g. "Lights of crossing streets do not go green at the same time"
- A bad state:
  - can be **immediately** recognized;
  - **cannot be sanitized** by following states.
- *S* is a safety property iff
  $\forall t.\ t \notin S \Longrightarrow (\exists t_1.\ t_1 \le t \Longrightarrow \forall t_2.\ t_1 \le t_2 \Longrightarrow t_2 \notin S)$, t_1 finite



Proving Safety Properties

- In the previous specification, t_1 is **finite**. As a consequence,
  - a property is a safety property if and only if its violation can be detected on a finite trace.
- Safety properties are typically proven by induction
  - Base case: initial states are good (= not bad)
  - Step case: each transition transforms a good state again in a good state
- Safety properties can be enforced by run-time monitors
  - Monitor checks following state in advance and allows execution only if it is a good state

Requirements on Runs: Liveness Properties



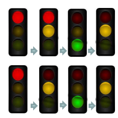
- Liveness property L:
  - „Good things will happen eventually"
  - E.g. "my traffic light will go green eventually * "
- A good thing is always possible and possibly infinite.
- L is a liveness property iff
  - $\forall t.\ \mathrm{finite}(t) \rightarrow \exists t_1.\ t \cdot t_1 \in L$
  - i.e. all finite traces t can be extended to a trace in L.

* Achtung: "eventually" bedeutet "irgendwann" oder "schlussendlich" aber *nicht* "eventuell" !

Satisfying Liveness Properties

- Liveness properties cannot (!) be enforced by run-time monitors.
- Liveness properties are typically proven by the help of well-founded orderings
  - Measure function *m* on states *s*
  - Each transition decreases *m*
  - $t \in L$ if we reach a state with minimal *m*
- E.g. measure denotes the number of transitions for the light to go green

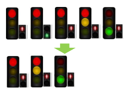
Requirements on Sets of Runs: Safety Hyperproperties



- Safety hyperproperty: „System never behaves bad"
  - No bad thing happens in a finite set of finite traces
  - (the prefixes of) different system runs do not exclude each other
  - E.g. "the traffic light cycle is always the same"
- A bad system can be recognized by a bad observation (set of finite runs)
  - A bad observation cannot be sanitized regards less how we continue it or add additional system runs
  - E.g. two system runs having different traffic light cycles
- S is a safety hyperproperty iff (see safety property):

$$\forall T. T \notin S \Longrightarrow (\exists Obs.\ Obs \le T \Longrightarrow \forall T'. Obs \le T' \Longrightarrow T' \notin S)$$

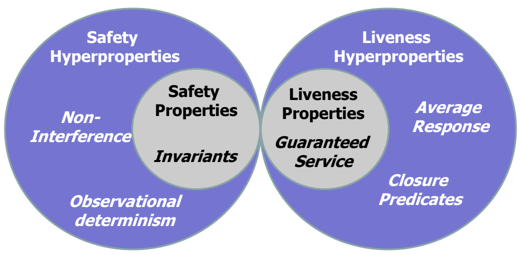
Requirements on Sets of Runs: Liveness Hyperproperties



- Liveness hyperproperty S:
  „The system will eventually develop to a good system"
  - Considering any finite part of a system behavior, the system eventually develops into a "good" system (by continuing appropriately the system runs or adding new system runs)
  - E.g. "Green light for pedestrians can always be omitted"
- L is liveness hyperproperty iff
  $\forall T.\ \exists G. T \le G \land G \in L$
  - T is a finite set of finite traces (observation)
  - Each observation can be explained by a system G satisfying L
- Examples:
  - Average response time
  - Closure operations in information flow control
  - Fair scheduling

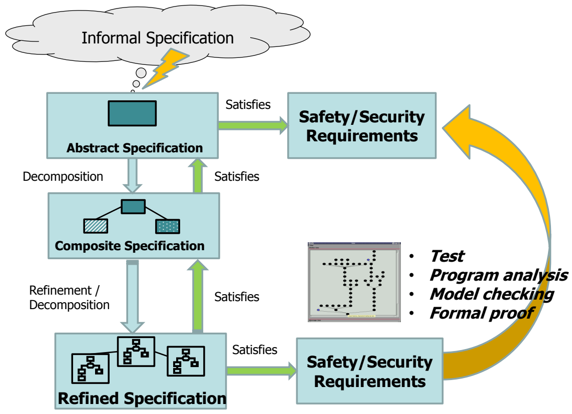
Landscape of (Hyper)Properties

- Each (hyper-) property can be represented as a combination of safety and liveness (hyper-) properties.



Structuring the Formal Development

The Global Picture

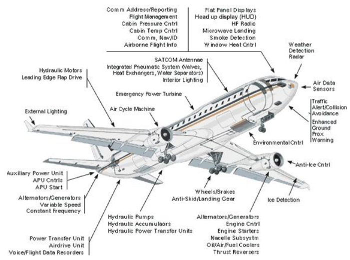


Structuring the Development

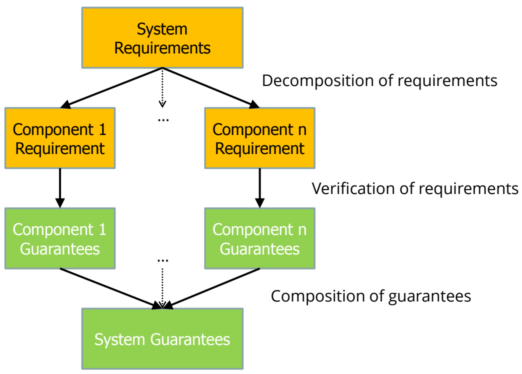
- Horizontal structuring:
  - Modularization into components
  - Composition and Decomposition
  - Aggregation
- Vertical structuring:
  - Abstraction and refinement from design specification to implementation
  - Declarative vs. imparative specification
  - Inheritance of properties
- Views:
  - Addresses multiple aspects of a system
  - Behavioral model, performance model, structural model, analysis model(e.g. UML, SysML)

Horizontal Structuring (informal)

- Composition of components
  - Dependent on the individual layer of abstraction
  - E.g. modules, procedures, functions,…
- Example:

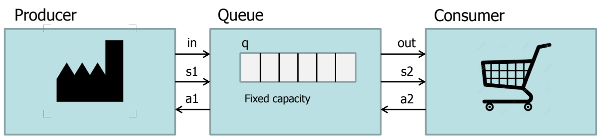


Modular Structuring of Requirements



Mutual Dependencies: Assume/Guarantee

- Safety requirement: Queue does not loose any items.




```
Loop:
  if (s1 == a1) {
   send(x, in);  s1 = not s1 }
```

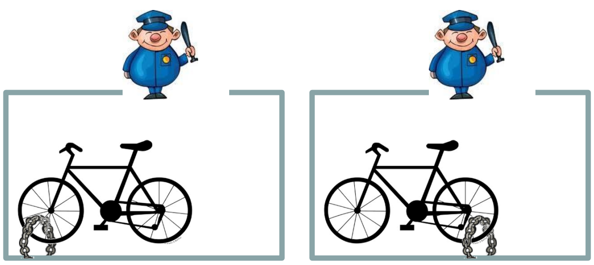
```
Loop:
  if (s1 != a1 && |q| < max)  {
     enq(q, in);
     a1 = not a1;
     }
  if (s2 == a2 && |q| > 0)  {
     deq(q, out);
     s2 != not s2
     }
```

```
Loop:
  if (s2 != a2) then {
     read(y, out);
     a2 = not a2;
     consume(y)
     }
```

- Components depend on each other!
- Initialization ?

Composition of Security Guarantees

Only complete bicycles are allowed to pass the gate.

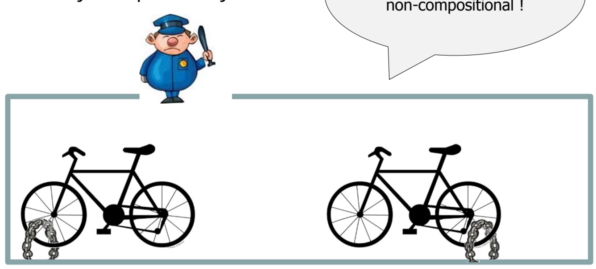


Secure ! **Secure !**

Composition of Security Guarantees

Only complete bicycles are a

Security properties are non-compositional !



Insecure !

Concurrent shared variable programs are non-compositional

```
long long x;

Thread1() {
        x = 1;
}
// @post: x == 1

Thread2() {
        x = (1 << 64);
}
// @post: x == (1 << 64)


(Thread1() || Thread2());
// @post: x == 1 or x == (1<<64)
```

Global variable

Post conditions hold in absence of concurrent threads

Does composition hold?

Concurrent shared variable programs are non-compositional

```
long long x;

(Thread1() || Thread2());

// @post: x == 1 or x == (1<<64) or x == (1<<64) + 1
```

- This post-condition cannot be derived from any logical composition of the original post-conditions of `Thread1()` and `Thread2()`
- For writing a 128bit integer to memory, two writes on the memory bus are required. As a consequence, the final value of `x` may also be `(1<<64) + 1`

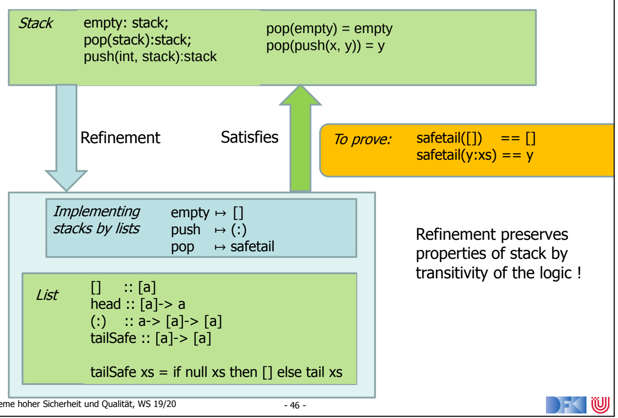
Vertical Structuring - Refinement

- **Idea**: start at an abstract description and add details step by step

  From abstract specification to an implementation

- What do we want to refine?
  - Algorithm: algebraic refinement
  - Data: data refinement
  - Process: process refinement
  - Events: action refinement

Algebraic Refinement



Even More Refinements

- Data refinement
  - Abstract datatype is „implemented“ in terms of the more concrete datatype
  - Simple example: define stack with lists
- Process refinement
  - Process is refined by excluding certain runs
  - Refinement as a reduction of underspecification by eliminating possible behaviours
- Action refinement
  - Action is refined by a sequence of actions
  - E.g. a stub for a procedure is refined to an executable procedure

Conclusion & Summary

- Software development models: structure vs. flexibility
- Safety standards such as IEC 61508, DO-178B suggest development according to V-model.
  - Specification and implementation linked by verification and validation.
  - Variety of artefacts produced at each stage, which have to be subjected to external review.
- Safety / Security Requirements
  - Properties: sets of traces
  - Hyperproperties: sets of properties
- Structuring of the development:
  - Horizontal – e.g. composition
  - Vertical – refinement (e.g. algebraic, data, process…)

Universität Bremen

Systeme hoher Sicherheit und Qualität

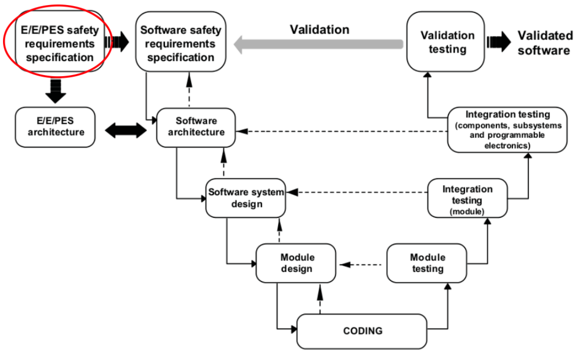
WS 2019/2020

Lecture 4: Hazard Analysis

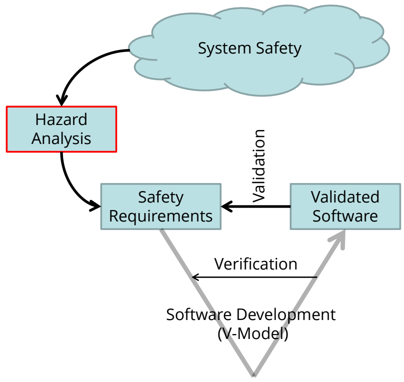
Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

Hazard Analysis in the Development Cycle



The Purpose of Hazard Analysis



Hazard Analysis systematically determines a list of **safety requirements**.

The realization of the safety requirements by the software product must be **verified**.

The product must be **validated** wrt. the safety requirements.

Hazard Analysis ...

- provides the basic **foundations** for **system safety**.
- is performed to **identify** hazards, hazard **effects**, and hazard **causal** factors.
- is used to determine **system risk**, to determine the significance of hazards, and to establish **design measures** that will eliminate or mitigate the identified hazards.
- is used to **systematically** examine systems, subsystems, facilities, components, software, personnel, and their interrelationships.

Clifton Ericson: *Hazard Analysis Techniques for System Safety.* Wiley-Interscience, 2005.

Form and Output of Hazard Analysis

The **output** of hazard analysis is a list of safety requirements and **documents** detailing how these were derived.

- Because the process is informal, it can only be **checked** by **reviewing**.
- It is therefore **critical** that
  - standard forms of analysis are used,
  - documents have a standardized form, and
  - all assumptions are documented.

Classification of Requirements

- Requirements to ensure:
  - safety
  - security
- Requirements for:
  - hardware
  - software
- Characteristics / classification of requirements:
  - according to the type of a property

Classification of Hazard Analysis

- **Top-down methods** start with an anticipated hazard and work backwards from the hazard event to potential causes for the hazard.
  - Good for finding causes for hazard;
  - good for avoiding the investigation of "non-relevant" errors;
  - bad for detection of missing hazards.
- **Bottom-up methods** consider "arbitrary" faults and resulting errors of the system, and investigate whether they may finally cause a hazard.
  - Properties are complementary to top-down properties;
  - Not easy with software where the structure emerges during development.

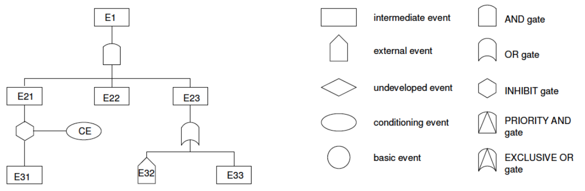
Hazard Analysis Methods

- **Fault Tree Analysis (FTA)** – top-down
- **Event Tree Analysis (ETA) –** bottom-up
- **Failure Modes and Effects Analysis (FMEA)** – bottom up
- Cause Consequence Analysis – bottom up
- HAZOP Analysis – bottom up

Fault Tree Analysis

Fault Tree Analysis (FTA)

- Top-down deductive failure analysis (of undesired states)
  - Define undesired top-level event (UE);
  - Analyze all causes affecting an event to construct fault (sub)tree;
  - Evaluate fault tree.



FTA: Cut Sets

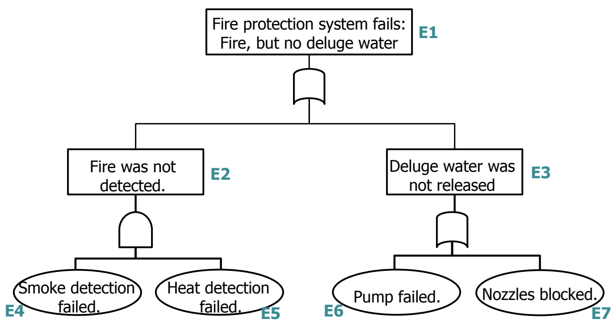
- A **cut set** is a set of events that cause the top UE to occur (also called a fault path).
- Cut sets reveal critical and weak links in a system.
- Extension- **probabilistic** fault trees:
  - Annotate events with probabilities;
  - Calculate probabilities for cut sets.
  - We do not pursue this further here, as it is mainly useful for hardware faults.
- Cut sets can be calculated top down or bottom up.
  - MOCUS algorithm (Ericson, 2005)
  - Corresponds to the DNF of underlying formula.
  - What happens to priority AND, conditioning and inhibiting events (modelled as implication?).

Fault-Tree Analysis: Process Overview

1. Understand system design
2. Define top undesired event
3. Establish boundaries (scope)
4. Construct fault tree
5. Evaluate fault tree (cut sets, probabilities)
6. Validate fault tree (check if correct and complete)
7. Modify fault tree (if required)
8. Document analysis

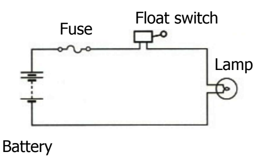
Fault Tree Analysis: First Simple Example

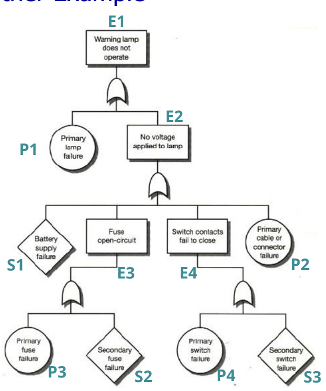
- Consider a simple **fire protection system** connected to smoke/heat detectors.



Fault Tree Analysis: Another Example

- A lamp warning about low level of brake fluid.
- Top undesired event: warning lamp off despite low level of fluid.



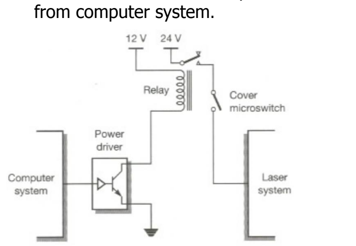


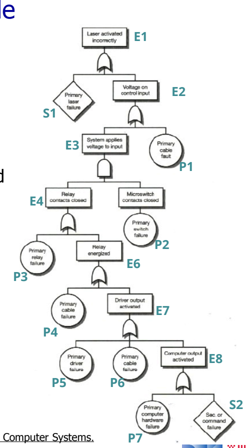
Source: N. Storey, Safety-Critical Computer Systems.

Fault Tree Analysis: Final Example

A laser is operated from a control computer system.

- The laser is connected via a relay and a power driver, and protected by a cover switch.
- Top Undesired Event: Laser activated without explicit command from computer system.





Source: N. Storey, Safety-Critical Computer Systems.

FTA - Conclusions

- Advantages:
  - Structured, rigorous, methodical approach;
  - Can be effectively performed and computerized, commercial tool support;
  - Easy to learn, do, and follow;
  - Combines hardware, software, environment, human interaction.
- Disadvantages:
  - Can easily become time-consuming and a goal in itself rather than a tool if not careful;
  - Modelling sequential timing and multiple phases is difficult.

Event Tree Analysis

Event Tree Analysis (ETA)

- Bottom-up method
- Applies to a chain of cooperating activities
- Investigates the effect of activities failing while the chain is processed
- Depicted as binary tree; each node has two leaving edges:
  - Activity operates correctly
  - Activity fails
- Useful for calculating risks by assigning probabilities to edges
- Complexity: $\mathcal{O}(2^n)$

Event Tree Analysis - Overview

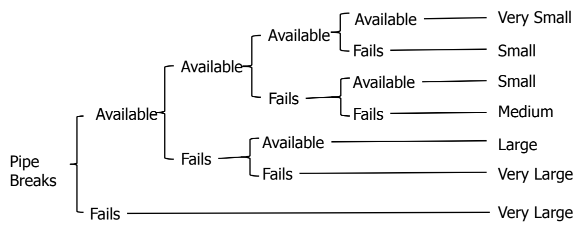
Input:
- Design knowledge
- Accident histories

ETA Process:
1. Identify Accident Scenarios
2. Identify IEs (Initiating Events)
3. Identify pivotal events
4. Construct event tree diagrams
5. Evaluate risk paths
6. Document process

Output:
- Mishap outcomes
- Outcome risks
- Causal sources
- Safety Requirements

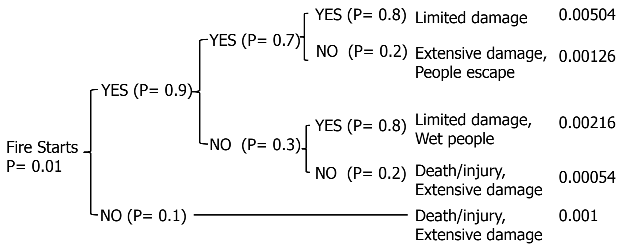
Example: Cooling System for a Nuclear Power Plant

Initating Event	*Pivotal Events*				*Outcome*
	Electricity	Emergency Core Cooling	Fission Product Removal	Containment	Fission Release



Probabilistic ETA: Fire Detection/Suppression System for Office Building

Initating Event Prob.	*Pivotal Events*			*Outcome*	*Prob.*
	Fire Detection Working	Fire Alarms Working	Fire Sprinkler Working		



ETA - Conclusions

- Advantages:
  - Structured, rigorous and metodical;
  - Can be effectively computerized, tool support is available;
  - Easy to learn, do, and follow;
  - Combines hardware, software, environment and human interaction;
  - Can be effectively performed on varying levels of system detail.
- Disadvantages:
  - An ETA can only have one IE;
  - Can overlook subtle system dependencies;
  - Partial success/failure not distinguishable.

Failure Mode and Effects Analysis

Failure Modes and Effects Analysis (FMEA)

- Analytic approach to review potential failure modes and their causes.
- Three approaches: *functional*, *structural* or *hybrid*.
- Typically performed on hardware, but useful for software as well.
- It analyzes
  - the failure mode,
  - the failure cause,
  - the failure effect,
  - its criticality,
  - and the recommended action,

and presents them in a **standardized table**.

Software Failure Modes

Guide word	Deviation	Example Interpretation
omission	The system produces no output when it should. Applies to a single instance of a service, but may be repeated.	No output in response to change in input; periodic output missing.
commission	The system produces an output, when a perfect system would have produced none. One must consider cases with both, correct and incorrect data.	Same value sent twice in series; spurious output, when inputs have not changed.
early	Output produced before it should be.	Really only applies to periodic events; Output before input is meaningless in most systems.
late	Output produced after it should be.	Excessive latency (end-to-end delay) through the system; late periodic events.
value (detectable)	Value output is incorrect, but in a way, which can be detected by the recipient.	Out of range.
value (undetectable)	Value output is incorrect, but in a way, which cannot be detected.	Correct in range; but wrong value

Criticality Classes

- Risk as given by the *risk mishap index* (MIL-STD-882):

Severity	**Probability**
1. Catastrophic	A. Frequent
2. Critical	B. Probable
3. Marginal	C. Occasional
4. Negligible	D. Remote
	E. Improbable

- Names vary, principle remains:
  - Catastrophic – single failure
  - Critical – two failures
  - Marginal – multiple failures/may contribute

PROBABILITY LEVELS

Description	Level	Specific Individual Item	Fleet or Inventory
Frequent	A	Likely to occur often in the life of an item.	Continuously experienced.
Probable	B	Will occur several times in the life of an item.	Will occur frequently.
Occasional	C	Likely to occur sometime in the life of an item.	Will occur several times.
Remote	D	Unlikely, but possible to occur in the life of an item.	Unlikely, but can reasonably be expected to occur.
Improbable	E	So unlikely, it can be assumed occurrence may not be experienced in the life of an item.	Unlikely to occur, but possible.
Eliminated	F	Incapable of occurence. This level is used when potential hazards are identified and later eliminated.	Incapable of occurence. This level is used when potential hazards are identified and later eliminated.

SEVERITY CATEGORIES

Description	Severity Category	Mishap Result Criteria
Catastrophic	1	Could result in one or more of the following: death, permanent total disability, irreversible significant environmental impact, or monetary loss equal to or exceeding $10M.
Critical	2	Could result in one or more of the following: permanent partial disability,injuries or occupational illness that may result in hospitalization of at least three personnel, reversible significant environmental impact, or monetary loss equal to or exceeding $1M but less than $10M.
Marginal	3	Could result in one or more of the following: injury or occupational illness resulting in one or more lost work day(s), reversible moderate environmental impact, or monetary loss equal to or exceeding $100K but less than $1M.
Negligible	4	Could result in one or more of the following: injury or occupational illness not resulting in a lost work day, minimal environmental impact, or monetary loss less than $100K.

Source:MIL-STD-822E, www.system-safety.org/Documents/MIL-STD-882E.pdf

FMEA Example: Airbag Control

- Consider an **airbag control system**, consisting of
  - the airbag with gas cartridge;
  - a control unit with
    - Output: Release airbag
    - Input: Accelerometer, impact sensors, seat sensors, …
- FMEA:
  - Structural: what can be broken?
    - Mostly hardware faults.
  - Functional: how can it fail to perform its intended function?
    - Also applicable for software.

Airbag Control (Structural FMEA)

ID	**Mode**	**Cause**	**Effect**	**Crit.**	**Appraisal**
1	Omission	Gas cartridge empty	Airbag not released in emergency situation	C1	SR-56.3
2	Omission	Cover does not detach	Airbag not released fully in emergency situation	C1	SR-57.9
3	Omission	Trigger signal not present in emergency.	Airbag not released in emergency situation	C1	Ref. To SW-FMEA
4	Comm.	Trigger signal present in non-emergency	Airbag released during normal vehicle operation	C2	Ref. To SW-FMEA

Airbag Control (Functional FMEA)

ID	**Mode**	**Cause**	**Effect**	**Crit.**	**Appraisal**
5-1	Omission	Software terminates abnormally	Airbag not released in emergency.	C1	See 5-1.1, 5-1.2.
5-1.1	Omission	- Division by 0	See 5-1	C1	SR-47.3 Static Analysis
5-1.2	Omission	- Memory fault	See 5-1	C1	SR-47.4 Static Analysis
5-2	Omission	Software does not terminate	Airbag not released in emergency.	C1	SR-47.5 Termination Proof
5-3	Late	Computation takes too long.	Airbag not released in emergency.	C1	SR-47.6 WCET Analysis
5-4	Comm.	Spurious signal generated	Airbag released in non-emergency	C2	SR-49.3
5-5	Value (u)	Software computes wrong result	Either of 5-1 or 5-4.	C1	SR-12.1 Formal Verification

FMEA - Conclusions

- Advantages:
  - Easily understood and performed;
  - Inexpensive to perform, yet meaningful results;
  - Provides rigour to focus analysis;
  - Tool support available.
- Disadvantages:
  - Focuses on single failure modes rather than combination;
  - Not designed to identify hazard outside of failure modes;
  - Limited examination of human error, external influences or interfaces.

Conclusions

The Seven Principles of Hazard Analysis

Source: Ericson (2005)

1) Hazards, mishaps and risk are not chance events.
2) Hazards are created during design.
3) Hazards are comprised of three components (HE, IM, T/T).
4) Hazards and mishap risk is the core safety process.
5) Hazard analysis is the key element of hazard and mishap risk management.
6) Hazard management involves seven key hazard analysis types.
7) Hazard analysis primarily encompasses seven hazard analysis techniques.

Summary

- Hazard Analysis is the **start** of the formal development.
- Its most important output are **safety requirements**.
- Adherence to safety requirements has to be **verified** during development, and **validated** at the end.
- We distinguish different types of analysis:
  - Top-Down analysis (Fault Trees)
  - Bottom-up (FMEAs, Event Trees)
- It makes sense to combine different types of analyses, as their results are complementary.

Conclusions

- Hazard Analysis is a creative process, as it takes an informal input („system safety") and produces a formal output (safety requirements). Its results cannot be formally proven, merely checked and reviewed.
- Review plays a key role. Therefore,
  - documents must be readable, understandable, auditable;
  - analysis must be in well-defined and well-documented format;
  - all assumptions must be well documented.

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 05:

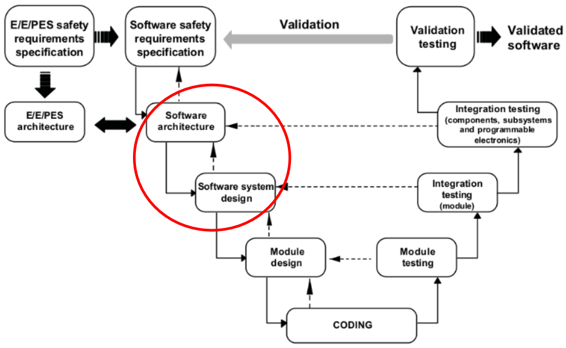
High-Level Design with SysML

Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

- 01: Concepts of Quality
- 02: Legal Requirements: Norms and Standards
- 03: The Software Development Process
- 04: Hazard Analysis
- 05: High-Level Design with SysML
- 06: Formal Modelling with OCL
- 07: Testing
- 08: Static Program Analysis
- 09-10: Software Verification
- 11-12: Model Checking
- 13: Conclusions

High-Level Design in the Development Cycle



What is a model?

- Different notions of models in physics, philosophy or computer science

A model is a representation in a certain medium of something in the same or another medium. The model captures the important aspects of the thing being modelled from a certain point of view and simplifies or omits the rest.

Rumbaugh, Jacobson, Booch: UML Reference Manual.

- Here: an abstraction of a system / a software / a development
- Purposes of models:
  - Understanding, communicating and capturing the design
  - Organizing decisions / information about a system
  - Analyzing design decisions early in the development process
  - Analyzing requirements

Different notions of models

- In **physics:** Models give mathematical representations of some part of reality
  - **Example**. Space-time models for understanding our universe.
- In **philosophy:** Models attach meaning to symbols and syntax
  - **Example**. Ontologies are used to a specify set of concepts and categories in a subject area or domain that shows their properties and the relations between them.
- In **computer science:** Models are used to specify systems to be built
  - **Example**. Class diagrams model the collection of classes to be programmed or used in a library, and the relations between these classes.
- In **organizational theory:** Models are used to specify organizations, companies, projects
  - **Example**. Organization charts

An Introduction to SysML

The Unified Modeling Language (UML)

- Grew out of a wealth of modelling languages in the 1990s (James Rumbaugh, Grady Booch and Ivar Jacobson at Rational)
- Adopted by the Object Management Group (OMG) in 1997, and approved as ISO standard in 2005.
- UML 2.5 consists of
  - a core meta-model,
  - a concrete modeling syntax,
  - the object constraint language (OCL),
  - an interchange format
- UML 2 is not a fixed language, it can be extended and customized using **profiles**.

- SysML is a ***modeling language*** for systems engineering
- Standardized in 2007 by the OMG (May 2017 at Ver 1.5)
- Latest SysML standard at https://www.omg.org/spec/SysML/About-SysML/

What for SysML?

- Serving as a standardized notation allowing all stakeholders to understand and communicate the salient aspects of the system under development
  - the requirements,
  - the structure (static aspects), and
  - the behaviour (dynamic aspects)
- Certain aspects (diagrams) of the SysML are **formal**, others are **informal**
  - Important distinction when developing critical systems
- All diagrams are **views** of one underlying model

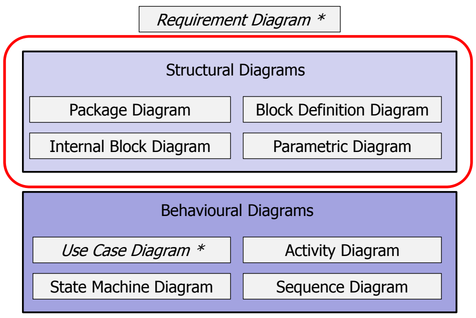
Different Views in SysML

- Structure:
  - How is the system constructed?
    How does it decompose?
- Behaviour:
  - What can we observe? Does it have a state?
- Requirements:
  - What are the requirements? Are they met?
- Parametrization:
  - What are the constraints (physical/design)?
- … and possibly more.

Example: A Cleaning Robot (HooverBot)

- Structure:
  - Has an engine, wheels (or tracks?), a vacuum cleaner, a control computer, a battery…
- Behaviour:
  - General: starts, then cleans until battery runs out, returns to charging station
  - Cleaning: moves in irregular pattern, avoids obstacle
- Requirements:
  - Must cover floor when possible, battery must last at least six hours, should never run out of battery, …
- Constraints:
  - Can only clean up to 5 g, can not drive faster than 1m/s, laws concerning movement and trajectory, …

SysML Diagrams



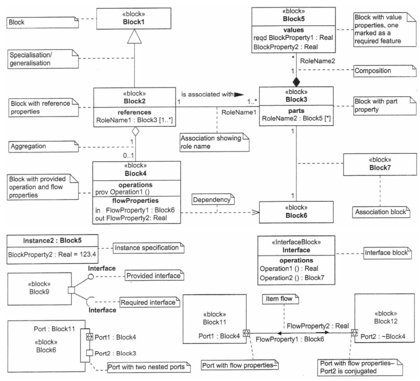
* Not considered further.

Structural Diagrams in SysML

Block Definition Diagram

- Blocks are the **basic building elements** of a model
  - Models are *instances* of blocks
- Block definition diagrams model **blocks** and their **relations**:
  - Inheritance
  - Association
- Blocks can also model interface definitions.
- Corresponds to **class diagrams** in the UML.
- Blocks modelling concurrent processes or HW units with a specific behaviour can be associated with state machines or activity charts (see below) specifying the behavior of the block. This behaviour is called the **classifier behaviour**. The block is marked with stereotype <<activity>> or <<stateMachine>>

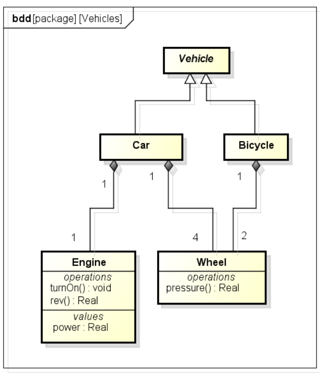
BDD – Summary of Notation



Quelle: Holt, Perry. SysML for Systems Engineering.

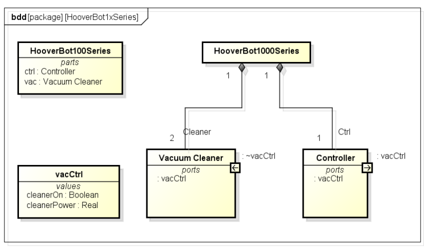
Example 1: Vehicles

- A vehicle can be a car, or a bicycle.
- A car has an engine
- A car has 4 wheels,
  a bicycle has 2 wheels
- Engines and wheels have operations and values
- In SysML, engine and wheel are **parts** of car and bicycle.



Example 2: HooverBots

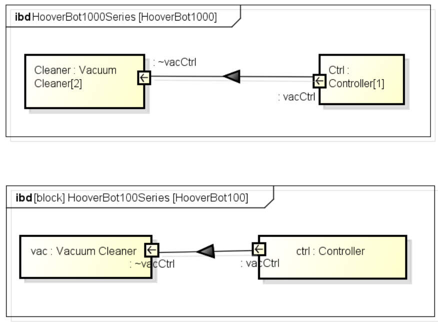
- The hoover bots have a control computer, and a vacuum cleaner (v/c).
  - HooverBot 100 has one v/c, Hoover 1000 has two.
  - Two ways to model this (i.e. two views):



Internal Block Diagrams

- Internal block diagrams decribe instances of blocks
- Here, instances for HooverBots
- On this level, we can describe connections between ports (flow specifications)
  - Flow specifications have directions.
  - Item flow specifications have directions.
  - Variants of ports
    - Proxy ports – typed by **interface blocks**
    - Full ports ("real physical interface") – typed by normal blocks
    - "normal, unspecified" ports " – typed by normal blocks

Example: HooverBot 100 and 1000

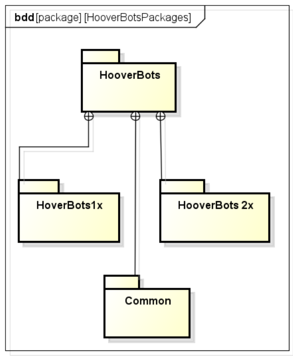


Modelling the system context

- SysML provides a special diagram type "Context Diagram" for modeling the target system as a black box, together with its interfaces to the operational environment.
- Alternatively, the context can be modeled by
  - a bdd showing the target system and the blocks of the operational environment, and
  - an ibd showing the target system block, the blocks of the operation environment, and the ports and item flows representing the interfaces between target system and environment.

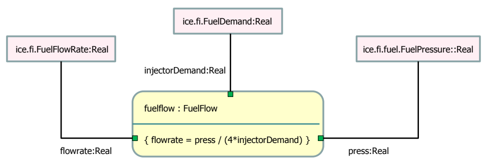
Package Diagrams

- Packages are used to group diagrams, much like directories in the file system.
- Not considered much in the following.



Parametric Diagrams

- Parametric diagrams describe constraints between properties and their parameters.
- It can be seen as a restricted form of an internal block diagram, or as equational modeling as in Simulink.



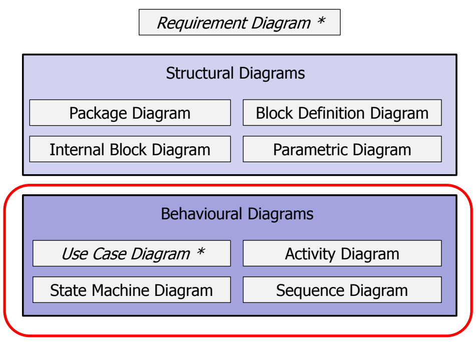
Relation of fuel flowrate to FuelDemand and FuelPressure value properties (Source: OMG SysML v1.2)

Modeling Tool: Astah-SysML

- Astah-SysML is available at

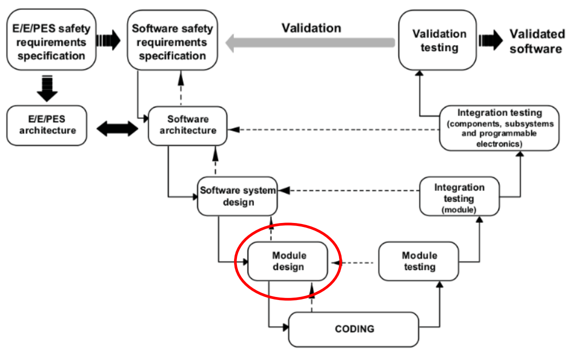
  http://astah.net/editions/sysml
- A faculty license is available for FB3 Uni Bremen
  - Non-commercial use only, do not distribute!
- The tool not only helps with the drawing, it also keeps track of the relationship between the diagrams: you edit the model rather than the diagrams.

SysML Diagrams Overview



* Not considered further.

Detailed Specification in the Development Cycle



Why detailed Specification?

- **Detailed specification** is the specification of single modules making up our system.
- This is the „last" level both in abstraction and detail before we get down to the code – in fact, some specifications at this level can be automatically translated into code.
- Why **not** write code straight away?
  - We want to stay platform-independent.
  - We may not want to get distracted by details of our target platform.
  - At this level, we have a better chance of finding errors or proving safety properties.

Levels of Detailed Specification

We can specify the basic modules:

- By their (external) **behaviour**
  - Operations defined by their pre/post-conditions and effects (e.g. in OCL)
  - Modeling the system's internal states by a state machine (i.e. states and guarded transitions)
- By their (internal) **structure**
  - Modeling the control flow by flow charts (aka. activity charts)
  - By action languages (platform-independent programming languages for UML, but these are not standard for SysML)

State Diagrams: Basics

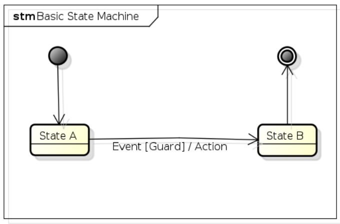
- State diagrams are a particular form of (hierarchical) **finite state machines**:

Definition: Finite State Machine (FSM)
A FSM is given by $\mathcal{M} = \langle \Sigma, I, \rightarrow \rangle$ where
- Σ is a finite set of **states**,
- $I \subseteq \Sigma$ is a set of **initial** states, and
- $\rightarrow \subseteq \Sigma \times \Sigma$ is a **transition relation**, s.t. $\rightarrow$ is left-total:
$$\forall s \in \Sigma . \exists s' \in \Sigma . s \rightarrow s'$$

- Example: a simple coffee machine
- We will explore FSMs in detail later.
- In hierarchical state machines, a state may contain another FSM (with initial/final states).
- State Diagrams in SysML are taken unchanged from UML.

Basic Elements of State Diagrams

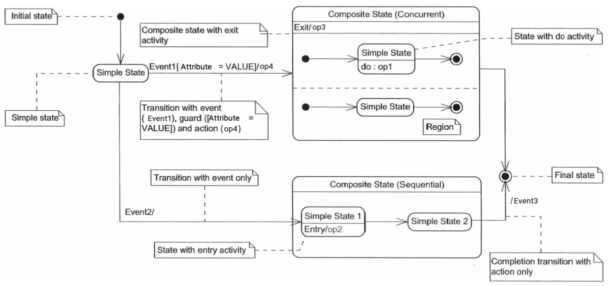
- States
  - Initial/Final
- Transitions
- Events (Triggers)
- Guards
- Actions (Effects)



What is an Event?

- *„The specification of a noteworthy occurence which has a location in time and space."* (UML Reference Manual)
- SysML knows:
  - Signal events **`event name/`**
  - Call events **`operation name/`**
  - Time events **`after(t)/`**
  - Change event **`when(e)/`**
  - Entry events **`Entry/`**
  - Exit events **`Exit/`**

SMDs – Summary of Notation



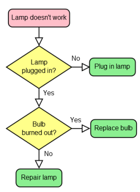
Quelle: Holt, Perry. SysML for Systems Engineering.

State Diagram Elements (SysML Ref. §13.2)

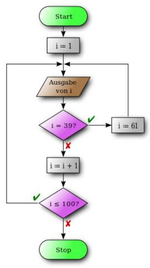
- Choice pseudo state
- Composite state
- Entry point
- Exit point
- Final state
- *History pseudo states*
- Initial pseudo state
- Junction pseudo state
- Receive signal action
- Send signal action
- Action
- Region
- Simple state
- State list
- State machine
- Terminate node
- Submachine state

Activity Charts: Foundations

- The activity charts of SysML (UML) are a variation of good old-fashioned **flow charts.**
  - Those were standardized as DIN 66001 (ISO 5807).
- Flow charts can describe programs (right example) or non-computational activities (left example)
- SysML activity charts are extensions of UML activity charts.



Quelle: Wikipedia



Quelle: Erik Streb, via Wikipedia

Basics of Activity Diagrams

- Activities model the work flow of low-level behaviours:
  "*An activity is the specification of parameterized behaviour as the coordinated sequencing of subordinate unites whose individual elements are actions.*"
  (UML Ref. §12.3.4)
- Diagram comprises of actions, decisions, joining and forking activities, start/end of work flow.
- Control flow allows to disable and enable (sub-) activities.
- An activity execution results in the execution of a set of actions in some specific order.

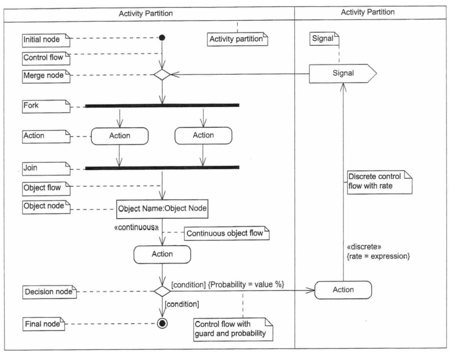
What is an Action?

- A terminating basic behaviour, such as
  - Changing variable values [UML Ref. §11.3.6]
  - Calling operations [UML Ref. §11.3.10]
  - Calling activities [UML Ref. §12.3.4]
  - Creating and destroying objects, links, associations
  - Sending or receiving signals
  - Raising exceptions .
- Actions are part of a (potentially larger, more complex) behaviour.
- Inputs to actions are provided by ordered sets of pins:
  - A pin is a typed element, associated with a multiplicity
  - Input pins transport typed elements to an action
  - Actions deliver outputs consisting of typed elements on output pins

Elements of Activity Diagrams

- Nodes:
  - Action nodes
  - Activities
  - Decision nodes
  - Final nodes
  - Fork nodes
  - Initial nodes
  - Local pre/post-conditions
  - Merge nodes
  - Object nodes
  - *Probabilities and rates*
- Paths (arrows):
  - Control flow
  - Object flow
  - Probability and rates
- Activities in BDDs
- Partitions
- Interruptible Regions
- Structured activities

Activity Diagrams – Summary of Notation



Quelle: Holt, Perry. SysML for Systems Engineering.

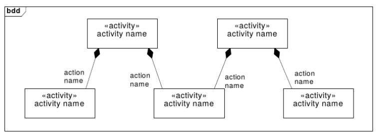
Behavioural Semantics

- Semantics is based on **token flow** – similar to Petri Nets, see [UML Ref. pp. 326]
  - A token can be an input signal, timing condition, interrupt, object node (representing data), control command (call, enable) communicated via input pin, …
  - An executable node (action or sub-activity) in the activity diagram begins its execution, when the required tokens are available on their input edges.
  - On termination, each executable node places tokens on certain output edges, and this may activate the next executable nodes linked to these edges.

Activity Diagrams – Links With BDDs

Block definition diagrams may show:

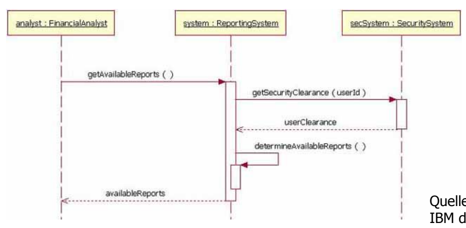
- Blocks representing activities



- One activity may be composed of other activities – composition indicates parallel execution threads of the activities at the "part end".
- One activity may contain several blocks representing **object nodes** (which represent data flowing through the activity diagram).

Sequence Diagrams

- Sequence Diagrams describe the flow of messages between actors.
- Extremely useful, but also extremely limited.



Quelle:
IBM developerWorks

- We consider concurrency in more depth later on.

Summary

- High-level modeling describes the structure of the system at an abstract level.
- SysML is a standardized modeling language for systems engineering, based on the UML.
  - We disregard certain aspects of SysML in this lecture.
- SysML structural diagrams describe this structure:
  - block definition diagrams,
  - internal block definition diagrams,
  - package diagrams.
- We may also need to describe formal constraints, or invariants.

Summary (cont.)

- Detailed specification means we specify the internal structure of the modules in our systems.
- Detailed specification in SysML:
  - State diagrams are hierarchical finite state machines which specify states and transitions.
  - Activity charts model the control flow of the program.
- More behavioural diagrams in SysML:
  - Sequence charts model the exchange of messages between actors.
  - Use case diagrams describe particular uses of the system.

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 06:

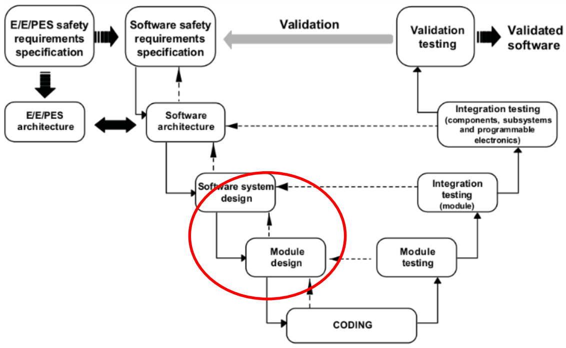
Formal Modeling with OCL

Christoph Lüth, Dieter Hutter, Jan Peleska

mit Folien v. Bernhard Beckert (KIT)

Where are we?

Formal Modeling in the Development Cycle



What is OCL?

- OCL is the **Object Constraint Language**.
  - Standardized by OMG actual version is OCL 2.4
  - Available at https://www.omg.org/spec/OCL/
- What is OCL?
  - *„A formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model."* (OCL standard, §7)
- Why OCL?
  - *„A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. "* (OCL standard, §7.1)

Characteristics of the OCL

- OCL is a pure **specification language**
  - OCL expressions do not have side effects
- OCL is **not** a programming language.
  - Expressions are not executable (though some may be)
- OCL is **typed** language
  - Each expression has a type; all expressions must be well-typed
  - Types are classes, defined by class diagrams

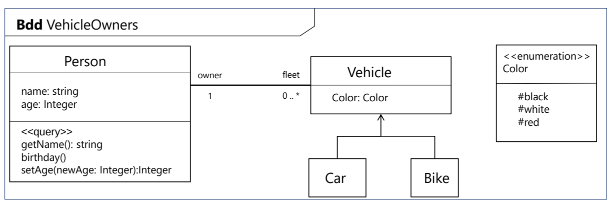
Usage of the OCL

- as a query language
- to specify invariants on classes and types in the class
- to specify type invariant for Stereotypes
- to describe pre- and post conditions on Operations and Methods
- to describe guards
- to specify target (sets) for messages and actions
- to specify constraints on operations
- to specify derivation rules for attributes for any expression over a UML model.

(OCL standard, §7.1.1)

OCL by Example

Why is SysML not enough?



What about requirements like:
- The minimal age of car owners
- The maximal number of cars (of a specific color) owned
- The maximal number of owners of a car

OCL Basics

- The language is **typed**: each expression has a type.
- Multiple-valued logic (true, false, undefined).

- Expressions always live in a **context**:
  - **Invariants** on classes, interfaces, types.

```
context Class
  inv Name: expr
```

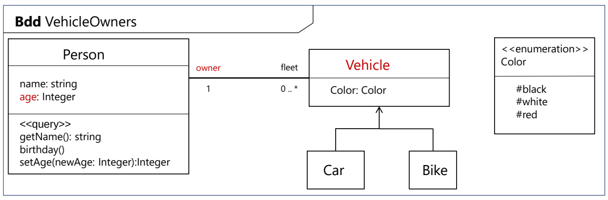
  - **Pre/postconditions** on operations or methods

```
context Class :: op(a1: Type, …, an: Type) : Type
  pre Name: expr
  post Name: expr
```


OCL Types

- Basic types:
  - `Boolean, Integer, Real, String`
  - `OclAny` – Enthält alle Typen
  - `OclVoid` – In allen Typen enthalten, nur eine Instanz `null`
  - `OclInvalid` – Fehlerwert (nur eine Instanz `invalid`)
- Collection types:
  - `Sequences, Bag, OrderedSet, Set`
- Model types

Invariants of Classes



"A vehicle owner must be at least 18 years old"

```
context   Vehicle
inv:      self.owner.age >= 18
```


Basic types and operations

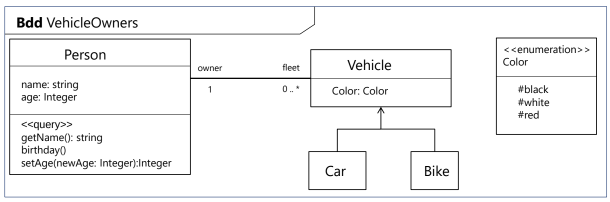
- Integer (ℤ) OCL-Std. §11.5.2
- Real (ℝ) OCL-Std. §11.5.1
  - `Integer` is a subclass of `Real`
  - `round, floor` from `Real` to `Integer`
- String (Zeichenketten) OCL-Std. §11.5.3
  - `substring, toReal, toInteger, characters`, etc.
- Boolean (Wahrheitswerte) OCL-Std. §11.5.4
  - `or, xor, and, implies`
  - Relationen auf `Real, Integer, String`

Collection Types

Sequence, Bag, OrderedSet, Set OCL-Std. §11.6, §11.7

- Operations on all collections:
  - `size, includes, count, isEmpty, flatten`
  - Collections are always „flattened"
  - Syntax: `collection->operation(…)`
- Set, OrderedSet
  - `union, intersection`
- Bag
  - `union, intersection, count`
- Sequence (lists)
  - `first, last, reverse, prepend, append`

Collections



"Nobody has more than 3 vehicles"

```
context   Person
Inv:      self.fleet->size <= 3
```


Collection Types: Quantification

We can quantify over collections: OCL-Std. §11.9.1

- Universal quantification :

```
coll->forAll(elem: Type| expr[elem]) : Boolean
```

- Existential quantification:

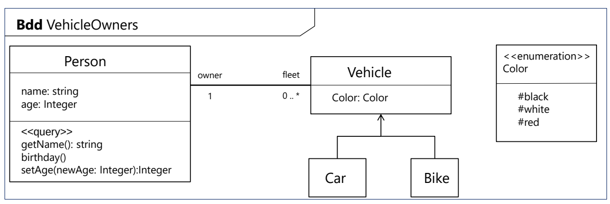
```
coll->exists(elem: Type| expr[elem]) : Boolean
```

- Comprehension operator:

```
coll->select(elem: Type| expr[elem]) : Coll[Type]
```

where `expr` is an expression of type `Boolean`.

Universal Quantification



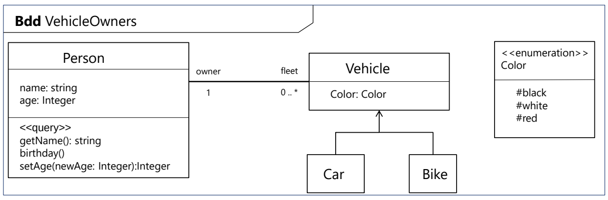
"All vehicles of a person are black"

```
context     Person
inv:        self.fleet->forAll(v | v.color = #black)
```

"No person has more than three black vehicles"

```
context     Person
inv:        self.fleet->select(v | v.color = #black)->size <= 3
```

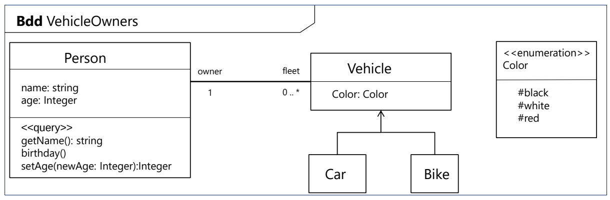
Universal Quantification



"A person younger than 18 owns no cars"

```
context  Person
inv:      self.age < 18 implies
                self.fleet -> forAll(v | not v.oclIsKindOf(Car))
```

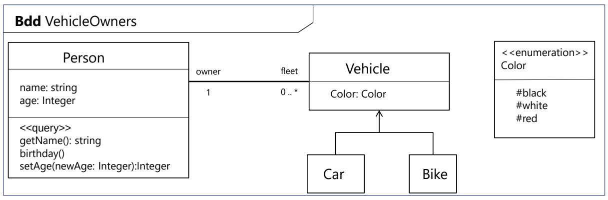

Existential Quantification



"There is a red car"

```
context   Car
inv:      Car.allInstances()->exists(c | c.color=#red)
```

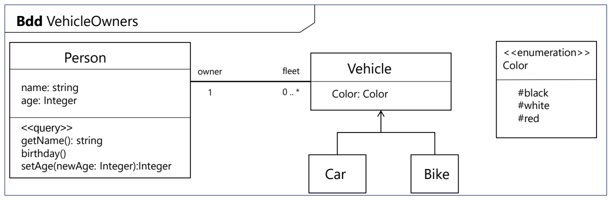

Pre/Post Conditions



"If **`setAge(a)`** is called with a non-negative argument **`a`**, then **`a`** becomes the new value of the attribute age."

```
context   Person::setAge(a:int)
pre:      a >= 0
post:     self.age = a
```


Pre/Post Conditions



"Calling birthday() increments the age of a person by 1."

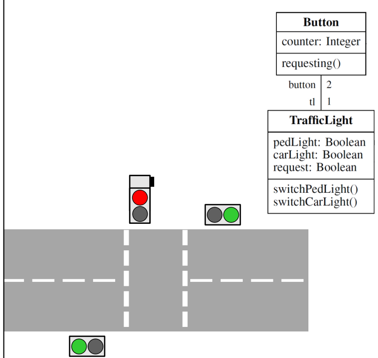
```
context   Person::birthday()
post:     self.age = self.age@pre + 1
```


Dynamic Aspects

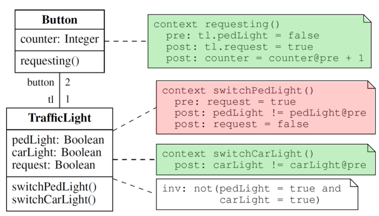
Modelling Dynamic Aspects

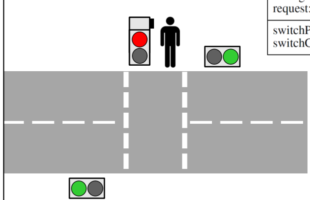
- Block diagrams model the **static structure** of the system: classes, attributes and the type of the operations. The possible **system states** are all instances of these model types.
- Invariants and pre/post conditions can be used to model the **dynamic aspects** of the system. In particular, they model all possible **state transitions** between the system states.
- An operation can become **active** (there is a state transition emanating from it) if the invariant holds, and the precondition holds. If there are no active state transitions, the system is **deadlocked**.
  - *Deadlocks should be avoided.*

Example: The Traffic Light



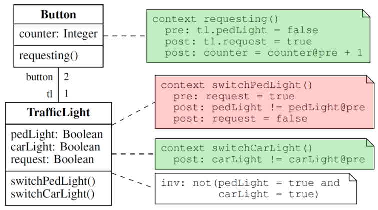
Example: The Traffic Light

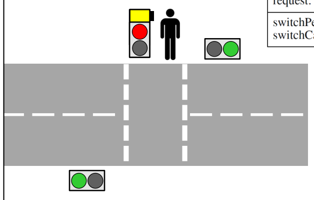




```
pedLight:    False
carLight:    True
request:     False
counter:     0
```

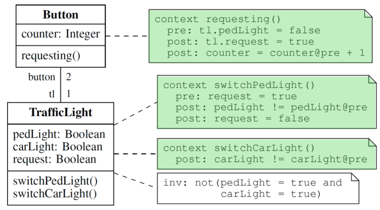
Example: The Traffic Light

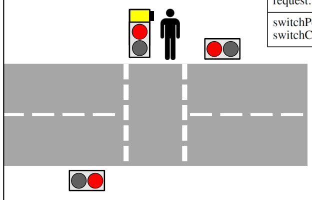




```
pedLight:    False
carLight:    True
request:     True
counter:     1
```

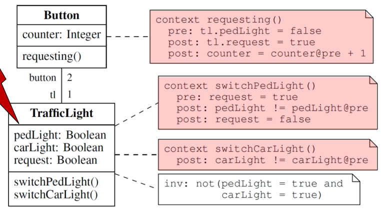

Example: The Traffic Light

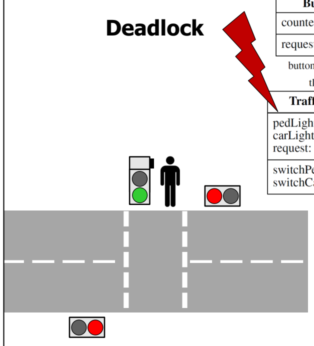




```
pedLight:    False
carLight:    False
request:     True
counter:     1
```


Example: The Traffic Light






```
pedLight:    True
carLight:    False
request:     False
counter:     1
```


OCL Details

Model types

- Model types are given by
  - Attributes,
  - Operations, and
  - Associations of the model
- Navigation along the association
  - If cardinality is 1, type is of target type `T`
  - Otherwise, it is `Set(T)`
- User-defined operations in expressions have to be stateless (stereotype `<<query>>`)

Collection Types: Iterators

acc stands for "accumulator"

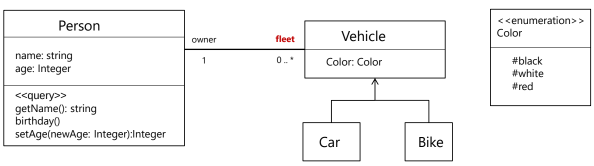
- Quantifiers are a special case of iterators.
  - Think of `all/any` in Haskell defined via `fold`
- All iterators defined via `iterate` OCL-Std. §7.6.6

```
coll->iterate(elem: T; acc: T2 = initial_expr
              | expr[elem, acc]) : T2
```

where `expr` of type `T` denotes a function on `elem` and `acc`

```
c.iterate(e: T, acc: T2 = v) = {
  acc= v;
  for (Enumeration e= c.elements(); e.hasMoreElements();) {
       acc= expr[e, acc];
       e= e.nextElement();
       }
  return acc;
}
```


Collection Types: Iterators



"A person owns at most 3 black vehicles"

```
context   Person
inv:      self.fleet->iterate(v; acc:Integer = 0
                             | if (v.color = #black)
                               then acc + 1 else acc
                            endif ) <= 3
```


Undefinedness in OCL

- Each domain of a basic type has two values denoting "**undefinedness**": OCL-Std §A.2.1.1
  - *null* or ε stands for "undefined", e.g. if an attribute value has not been set or is not defined (Type `OclVoid`)
  - *invalid* or $\bot$ stands for "invalid" and signals an error in the evaluation of an expression (e.g. division by 0, or application of a partial function) (Type `OclInvalid`)
  - As subtypes: `OclInvalid` $\subseteq$ `OclVoid` $\subseteq$ all other types
- Undefinedness is **propagated.**
  - In other words, all operations are **strict**: „an *invalid* or *null* operand causes an *invalid* result".

The OCL Logic

- Exceptions to strictness:
  - Boolean operators (see below)
  - Case distinction
  - Test on definedness: **`oclIsUndefined`** with

$$oclIsUndefined(e) = \begin{cases} true & if\ e = \bot \lor e = null \\ false & otherwise \end{cases}$$

- The domain type for **`Boolean`** also contains null and invalid.
  - The resulting logic is **four-valued.**
  - It is a **Kleene-Logic**: $A \rightarrow B \equiv \neg A \lor B$
  - Boolean operators (**`and, or, implies, xor)`** are **non-strict on both sides**.
  - But equality (like all other relations) is strict: $\bot = \bot$ is $\bot$

OCL Boolean Operators: Truth Table

b_1	b_2	b_1 and b_2	b_1 or b_2	b_1 xor b_2	b_1 implies b_2	not b_1
false	false	false	false	false	true	true
false	true	false	true	true	true	true
true	false	false	true	true	false	false
true	true	true	true	false	true	false
false	ε	false	ε	ε	true	true
true	ε	ε	true	ε	ε	false
false	⊥	false	⊥	⊥	true	true
true	⊥	⊥	true	⊥	⊥	false
ε	false	false	ε	ε	ε	ε
ε	true	ε	true	ε	true	ε
ε	ε	ε	ε	ε	ε	ε
ε	⊥	⊥	⊥	⊥	⊥	ε
⊥	false	false	⊥	⊥	⊥	⊥
⊥	true	⊥	true	⊥	true	⊥
⊥	⊥ or ε	⊥	⊥	⊥	⊥	⊥

- Legend: ⊥ is *invalid*, ε is *null*. OCL-Std §A .2.1.3, Table A.2

OCL Style Guide

- Avoid **complex** navigation („Loose coupling").
  - Otherwise changes in models break OCL constraints.
- Always choose **adequate context.**
- „Use of **`allInstances()`** is **discouraged**"
- Split up invariants if possible.
- Consider defining **auxiliary operations** if expressions become too complex.

Summary

- OCL is a typed, state-free specification language which allows us to denote constraints on models.
- We can define or models much more precise.
  - Ideally: no more natural language needed.
- OCL is part of the more „academic" side of UML/SysML.
  - Tool support is not great, some tools ignore OCL, most tools at least type-check OCL, hardly any do proofs.
- However, in critical system development, the kind of specification that OCL allows is **essential.**
- Try it yourself: USE – Tool http://useocl.sourceforge.net
  Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming, 69:27-34, 2007.

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

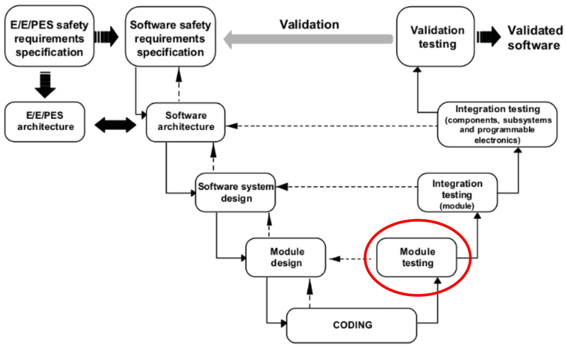
Lecture 07:

Testing

Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

Testing in the Development Cycle



What is Testing?

Testing is the proce
intent of finding err

BUT: testing can prove the **absence** of errors under certain hypotheses – so-called **complete** test methods

see http://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf

This concept is closely related to model checking

It is well known that Dijkstra hated model checking and frowned upon testing …

- In our sense, testi
- The **aim** of testing
  - derivation of
    compared to
  - inconsistency b
  - structural features o… that cause a faulty behavior of a program

Program testing can be used to show the presence of bugs, but never to show their absence.
E.W. Dijkstra, 1972

Why is testing so important?

- Even if one day code can be completely verified using formal methods, tests will still be required because--
- for embedded systems, the correctness of the HW/SW integration must be verified by testing, because--
- as of today, it is infeasible to provide a correct and complete formal model for complete HW/SW systems, comprising:
  - source code,
  - machine code,
  - CPU micro code,
  - firmware on interface hardware,
  - CPUs, busses, caches, memory, and interface boards.
- This will stay infeasible in the foreseeable future

The Testing Process

- Test cases, test plan, etc.
- System-under-test (s.u.t.)
  - Aka. TOE (target-of-evaluation) in CC
  - Aka. Implementation-under-test
- Warning -- test literature is quite expansive:

Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.
Hetzel, 1983

Test Levels

- **Component** and **unit tests**
  - test at the interface level of single components (modules, classes)
- **Integration test**
  - testing interfaces of components fit together
- **System test**
  - functional and non-functional test of the complete system from the user's perspective
- **Acceptance test**
  - testing if system implements contract details

Test Methods

- Static vs. dynamic
  - With **static** tests, the code is **analyzed** without being run. We cover these methods as static program analysis later
  - With **dynamic** tests, we **run** the code under controlled conditions, and check the results against a given specification
- Central question: where do the **test cases** come from?
  - **Black-box**: the inner structure of the s.u.t. is opaque, test cases are derived from specification **only.**
  - **Grey-box:** some inner structure of the s.u.t. is known, e.g. module architecture.
  - **White-box**: the inner structure of the s.u.t. is known, and tests cases are derived from the source code **and** coverage objectives for the source code

Black-Box Tests

- Limit analysis:
  - If the specification limits input parameters, then values **close** to these limits should be chosen
  - Idea is that programs behave **continuously**, and errors occur at these limits
- Equivalence classes:
  - If the input parameter values can be decomposed into **classes** which are treated equivalently, test cases have to cover all classes
- Smoke test:
  - "Run it, and check it does not go up in smoke."

Example: Black-Box Testing

- Equivalence classes or limits?

> **Example: A Company Bonus System**
> The loyalty bonus shall be computed depending on the time of employment. For employees of more than three years, it shall be 50% of the monthly salary, for employees of more than five years, 75%, and for employees of more than eight years, it shall be 100%.

- Equivalence classes or limits?

> **Example: Air Bag**
> The air bag shall be released if the vertical acceleration a_v equals or exceeds $15\ {}^{m}/_{s^2}$. The vertical acceleration will never be less than zero, or more than $40\ {}^{m}/_{s^2}$.

Black-Box Tests

- Quite typical for **GUI tests**, or **functional testing**
- Testing **invalid input**: depends on programming language – the stronger the typing, the less testing for invalid input is required
  - Example: consider lists in C, Java, Haskell
  - Example: consider object-relational mappings[1] (ORM) in Python, Java

1) Translating e.g. SQL-entries to objects

Complete Model-based Black-box Testing

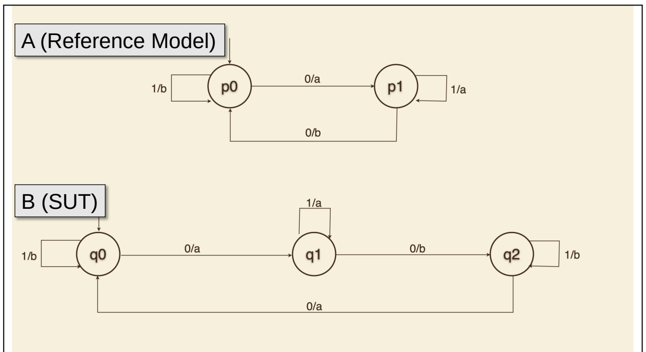
- Create a model *M* of the expected system behaviour
- Specify a **fault model** *(M, ≤, Dom)* with reference model *M*, **conformance relation** ≤ and **fault domain** *Dom* (a collection of models that may or may not conform to M)
- Derive test cases from fault model
- The resulting test suite is **complete** if
  - Every conforming SUT will pass all tests (**soundness**)
  - Every non-conforming SUT whose true behavior is reflected by a member of the fault domain fails at least on test case (**exhaustiveness**)
  - (nothing is guaranteed for SUT behaviors outside the fault domain)

Example: the W-Method

- The W-Method specifies a recipe for constructing complete test suites for finite state machines (FSMs) with conformance relation "~" **language equivalence (I/O-equivalence):**
  - Create a state cover *V*
  - Create a characterization set *W*
  - Assume that implementation has at most m ≥ n states (n is the number of states in the observable, minimized reference model)
  - Create test suite according to formula

$$\mathcal{W} = V.(\bigcup_{i=0}^{m-n+1} I^i).W$$

I : input alphabet
I^i : input traces of length i
$A.B$: all traces of A concatenated with all traces from B



$$n = 2 \qquad m = 3 \qquad V = \{\varepsilon, 0\} \qquad W = \{1\}$$

$$\bigcup_{i=0}^{m-n+1} I^i = \{\varepsilon, 0, 1, 0.0, 0.1, 1.0, 1.1\}$$

$$\mathcal{W}(A) = V.(\bigcup_{i=0}^{m-n+1} I^i).W = \{1, 0.1, 1.1, 0.0.1, 0.1.1, 1.0.1, 1.1.1, 0.0.0.1, 0.0.1.1, 0.1.0.1, 0.1.1.1\}$$

SUT error is uncovered by 0.0.0.1

Property-based Testing

- In property-based testing (or random testing), we generate **random** input values, and check the results against a given **executable** specification.
- Attention needs to be paid to the **distribution** values.
- Works better with **high-level languages,** where the datatypes represent more information on an abstract level and where the language is powerful enough to write comprehensive executable specifications (i.e. Boolean expressions).
  - Implementations for e.g. Haskell (QuickCheck), Scala (ScalaCheck), Java
- Example: consider list reversal in C, Java, Haskell
  - Executable spec: reversal is idempotent and distributes over concatenation.
  - Question: how to generate random lists?

White-Box Tests

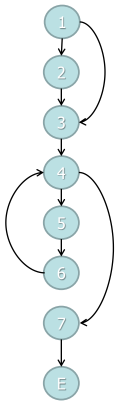
- In white-box tests, we derive test cases based on the structure of the program (**structural testing**)
  - To abstract from the source code (which is a purely syntactic artefact), we consider the **control flow graph** of the program.

> **Def: Control Flow Graph (CFG)**
> - nodes as elementary statements (e.g. assignments, **return**, **break**, . . .), as well as control expressions (e.g. in conditionals and loops), and
> - vertices from *n* to *m* if the control flow can reach a node *m* coming from a node *n*.

- Hence, **paths** in the CFG correspond to **runs** of the program.

Example: Control-Flow Graph

```
if (x < 0)  /*1*/  {
   x= – x; /*2*/
}
z = 1;  /*3*/
while (x > 0)  /*4*/ {
  z = z * y;  /*5*/
  x = x – 1;   /*6*/
}
return z;  /*7*/
```



An execution path is a path though the cfg ending with an exit node.

Examples:

- [1,3,4,7, E]
- [1,2,3,4,7, E]
- [1,2,3,4,5,6,4,7, E]
- [1,3,4,5,6,4,5,6,4,7, E]
- …

Coverage

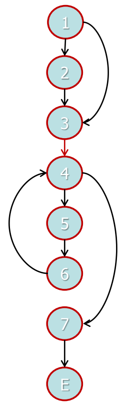
- **Statement coverage**:
  Measures the percentage of statements that were covered by the tests.
  100% statement coverage is reached if each **node** in the CFG has been visited at least once.
- **Branch coverage**:
  Measures the percentage of **edges** (emanating from branching or non-branching nodes) covered by the tests.
  100% branch coverage is reached if every edge of the CFG has been traversed at least once.
- **Path coverage**:
  Measures the percentage of CFG **paths** that have been covered by the tests.
  100% path coverage is achieved if every path of the CFG has been covered at least once.

Decision Coverage

- **Decision coverage**:
  Measures the coverage of conditional branches (i.e., edges emanating from conditional nodes). 100% decision coverage is reaches if the tests cover all conditional branches.
- **Decision coverage vs. branch coverage:**
  - If branch coverage is 100%, then decision coverage is 100% and vice versa.
  - A lower percentage p < *100%* of branch coverage, however, has a different meaning than a decision coverage of p, because
  - branch coverage considers all edges, whereas
  - decision coverage considers edges emanating from decision nodes only

Example: Statement Coverage

```
if (x < 0)    /*1*/  {
   x= – x;   /*2*/
 }
z = 1;  /*3*/
while (x > 0)  /*4*/ {
  z = z * y;   /*5*/
  x = x – 1;   /*6*/
 }
return z;   /*7*/
```



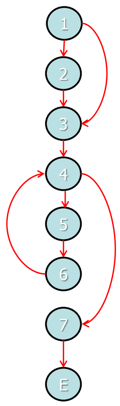
- Which (minimal) path covers all statements?

  p = [1,2,3,4,5,6,4,7,E]
- Which state generates p?

  x = -1
  y any
  z any

Example: Branch Coverage

```
if (x < 0)    /*1*/  {
   x= – x;   /*2*/
 }
z = 1;  /*3*/
while (x > 0)  /*4*/ {
  z = z * y;   /*5*/
  x = x – 1;   /*6*/
 }
return z;   /*7*/
```



- Which (minimal) paths cover all vertices?

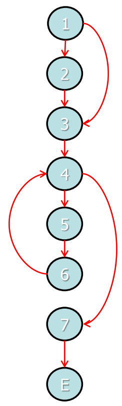
  $p_1 = [1,2,3,4,5,6,4,7,E]$
  $p_2 = [1,3,4,7,E]$
- Which states generate p_1, p_2?

	p_1	p_2
x	-1	0
y	any	any
z	any	Any

- Note p_3 with x= 1 does not add coverage.

Example: Path Coverage

```
if (x < 0)    /*1*/  {
   x= – x;   /*2*/
 }
z = 1;  /*3*/
while (x > 0)  /*4*/ {
  z = z * y;   /*5*/
  x = x – 1;   /*6*/
 }
return z;   /*7*/
```



- How many paths are there?
- Let $q_1 = [1,2,3]$
  $q_2 = [1,3]$
  $p = [4,5,6]$
  $r = [4,7,E]$

  then all paths are

  $$P = (q_1|q_2)\ p^*\ r$$
- Number of possible paths:

  $$|P| = 2 \cdot MaxInt - 1$$

Statement, Branch and Path Coverage

- **Statement Coverage**:
  - Necessary but not sufficient, not suitable as only test approach.
  - Detects dead code (code which is never executed).
  - About 18% of all defects are identified.
- **Branch coverage**:
  - Least possible single approach.
  - Needs to be achieved by (specification-based) tests for avionic software of DAL-C – does not suffice for DAL-B or DAL-A.
  - Detects dead code, but also frequently executed program parts.
  - About 34% of all defects are identified.
- **Path Coverage:**
  - Most powerful structural approach;
  - Highest defect identification rate (close to 100%);
  - But no **practical** relevance.

Decision Coverage Revisited

- Decision coverage requires that for all decisions in the program, each possible outcome is considered once.
- **Problem**: cannot sufficiently distinguish Boolean expressions.
  - Example: for A || B, the following are sufficient:

A	B	Result
False	False	False
True	False	True

  - But this does not distinguish A || B from A; B is effectively not tested.

Decomposing Boolean Expressions

- The binary Boolean operators include conjunction $x \wedge y$, disjunction $x \vee y$, or anything expressible by these (e.g. exclusive disjunction, implication)

> **Elementary Boolean Terms**
> An elementary Boolean term does not contain binary Boolean operators, and cannot be further decomposed.

- An elementary term is a variable, a Boolean-valued function, a relation (equality $=$, orders $<, \leq, >, \geq$, etc.), or a negation of these.
- This is a fairly syntactic view, e.g. $x \leq y$ is elementary, but $x < y \vee x = y$ is not, even though they are equivalent.
- In formal logic, these are called **literals.**

Simple Condition Coverage

- For each decision in the program, each elementary Boolean term (condition) evaluates to *True* and *False* at least once
- Note that this does not say much about the possible value of the condition
- Example:

```
if (temperature > 90 && pressure > 120) {… }
```

C1	*C2*	*Result*	
False	False	False	
False	True	False	-- These two would be enough
True	False	False	-- for condition coverage
True	True	True	

Modified Condition Coverage

- It is not always possible to generate all possible combinations of elementary terms, e.g. 3 <= x && x < 5.
- In modified (or minimal) condition coverage, all possible combinations of those elementary terms the value of which determines the value of the whole condition need to be considered.
- Example: 3 <= x && x < 5

3 <= x	x < 5	Result
False	False	False
False	True	False
True	False	True
True	True	True

- Another example: (x > 1 && ! p) || p

Modified Condition/Decision Coverage

- Modified Condition/Decision Coverage (MC/DC) is required by the "aerospace norm" **DO-178B** for Level A software.
- It is a **combination** of the previous coverage criteria defined as follows:
  - Every point of entry and exit in the program has been invoked at least once;
  - Every decision in the program has taken all possible outcomes at least once;
  - Every condition (i.e. elementary Boolean terms earlier) in a decision in the program has taken all possible outcomes at least once;
  - Every condition in a decision has been shown to independently affect that decision's outcome.

How to achieve MC/DC

- **Not**: Here is the source code, what is the minimal set of test cases?
- **Rather**: From requirements we get test cases, do they achieve MC/DC?
- Example:
  - Test cases:

Test case	**1**	**2**	**3**	**4**	**5**
Input A	F	F	T	F	T
Input B	F	T	F	T	F
Input C	T	F	F	T	T
Input D	F	T	F	F	F
Result Z	*F*	*T*	*F*	*T*	*T*

Source Code:
Z = (A || B) && (C || D)

Question: do test cases achieve MC/DC?

Source: Hayhurst *et al*, A Practical Tutorial on MC/DC. NASA/TM2001-210876

Example: MC/DC

Determining MC/DC:

1. Are all decisions covered?
2. Eliminate masked inputs (recursively)
   - False for && masks other input
   - True for || masks other input
3. Remaining unmasked test cases must cover all conditions.

Source Code
Z = (A || B) && (C || D)

Test case	**1**	**2**	**3**	**4**	**5**
Input A	F	F	T	F	T
Input B	F	T	F	T	F
Input C	T	F	F	T	T
Input D	F	T	F	F	F
Result Z	*F*	*T*	*F*	*T*	*T*

Here:

- Result is both F and T, so decisions covered.
- Masking:
  - In test case 1, C and D are masked
  - In test case 3, A and B are masked
  - Recursive masking as shown
- Remaining cases cover T, F for A, B, C, D
  - MC/DC achieved
  - In fact, test case 4 not even needed (?)

Summary

- (Dynamic) Testing is the controlled execution of code, and comparing the result against an expected outcome.
- Testing is (traditionally) the main way for **verification.**
- Depending on how the test cases are derived, we distinguish **white-box** and **black-box** tests.
- In black-box tests, we can consider **limits** and **equivalence classes** for input values to obtain test cases.
- In white-box tests, we have different notions of **coverage**: statement coverage, path coverage, condition coverage, etc.
- Next week: **Static testing** aka. static **program analysis**

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

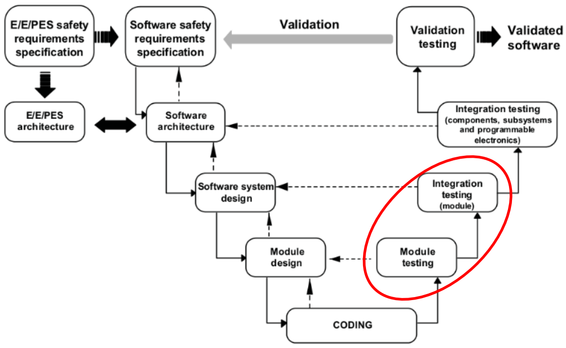
Lecture 08:

Static Program Analysis

Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

Program Analysis in the Development Cycle



Static Program Analysis

- Analysis of run-time behaviour of programs **without executing them** (sometimes called static testing).
- Analysis is done for **all** possible runs of a program (i.e. considering all possible inputs).
- Typical questions answered:
  - Does the variable *x* have a constant value ?
  - Is the value of the variable *x* always positive ?
  - Are all pointer dereferences valid (or NULL)?
  - Are all arithmetic operations well-defined (no over-/underflow)?
  - Do any unhandled exceptions occur?
- These tasks can be used for **verification** or for **optimization** when compiling.

Usage of Program Analysis

Optimizing compilers
- Detection of sub-expressions that are evaluated multiple times
- Detection of unused local variables
- Pipeline optimizations

Program verification
- Search for runtime errors in programs (program safety):
  - Null pointer or other illegal pointer dereferences
  - Array access out of bounds
  - Division by zero
- Runtime estimation (worst-caste executing time, wcet)

In other words, **specific** verification **aspects.**

Runtime Errors

- Program analysis often aims at finding errors that are independent of the specific functional specification, but violate the semantic rules of the programming language.
- These errors are called **runtime errors**, such as:
  - Division by zero, or violation of other preconditions
  - Exceptions which are thrown and not caught
  - Dereferencing NULL pointers, reading or writing to illegal addresses
  - Violation of array boundaries or heap memory boundaries
  - Use of uninitialized heap or stack data
  - Unintended non-terminating loops or recursion, stack overflow
  - Illegal type cast or class cast
  - Overflows (integer or real number cannot be represented in the available registers) or underflows (generation of a floating point number that is to small to be represented)
  - Memory leaks

Program Analysis: The Basic Problem

Given a property P and a program p: $p \vDash P$ iff P holds for p

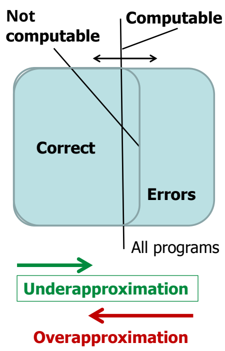
- Wanted: a terminating algorithm $\phi(p, P)$ which computes $p \vDash P$
  - ϕ is sound if $\phi(p, P)$ implies $p \vDash P$
  - ϕ is complete if $\neg\phi(p, P)$ implies $\neg\, p \vDash P$
  - If ϕ is sound and complete then ϕ is a decision procedure

The **basic problem** of static program analysis: virtually all interesting program properties are **undecidable**! (cf. Gödel, Turing)

- From the basic problem it follows that there are no sound and complete tools for interesting properties.
- Tools for interesting properties are either
  - sound (under-approximating) or
  - complete (over-approximating).

Program Analysis: Approximation

- **Under-approximation** is sound but not complete. It only finds correct programs but may miss out some.
  - Useful in **optimizing compilers;**
  - Optimization must preserve semantics of program, but is optional.
- **Over-approximation** is complete but not sound. It finds all errors but may find non-errors (**false positives**).
  - Useful in verification;
  - Safety analysis must find all errors, but may report some more.
  - Too high rate of false positives may hinder acceptance of tool.



Program Analysis Approach

- Provides **approximate** answers
  - yes / no / don't know or
  - superset or subset of values
- Uses an **abstraction** of program's behavior
  - Abstract data values (e.g. sign abstraction)
  - Summarization of information from execution paths e.g. branches of the if-else statement
- **Worst-case** assumptions about environment's behavior
  - e.g. any value of a method parameter is possible.
- Sufficient **precision** with good **performance.**

Analysis Properties: Flow Sensitivity

Flow-insensitive analysis

- Program is seen as an unordered collection of statements
- Results are valid for any order of statements e.g. S_1 ; S_2 vs. S_2 ; S_1
- Example: type analysis (inference)

Flow-sensitive analysis

- Considers program's flow of control
- Uses control-flow graph as a representation of the source
- Example: available expressions analysis (expressions that need not be re-computed at a certain point during compilation)

Analysis Properties: Context **Sensitivity**

Context-sensitive analysis

- Stack of procedure invocations and return values of method parameters
- Results of analysis of the method *M* depend on the caller of *M*

Context-insensitive analysis

- Produces the same results for all possible invocations of *M* independent of possible callers and parameter values.

Intra- vs. Inter-procedural Analysis

Intra-procedural analysis

- Single function is analyzed in isolation.
- Maximally pessimistic assumptions about parameter values and results of procedure calls.

Inter-procedural analysis

- Procedure calls are considered.
- Whole program is analyzed at once.

Data-Flow Analysis

Focus on questions related to values of variables and their lifetime

Selected analyses:

- **Available expressions (forward analysis)**
  - Which expressions have been computed already without change of the occurring variables (optimization) ?
- **Reaching definitions (forward analysis)**
  - Which assignments contribute to a state in a program point? (verification)
- **Very busy expressions (backward analysis)**
  - Which expressions are executed in a block regardless which path the program takes (verification) ?
- **Live variables (backward analysis)**
  - Is the value of a variable in a program point used in a later part of the program (optimization) ?

A Simple Programming Language

- **Arithmetic** expressions:

$$a \ ::= x \mid n \mid a_1 \, op_a \, a_2$$

  - Arithmetic operators: $op_a \in \{+,-,*,/\}$
- **Boolean** expressions:

$$b \ \coloneqq \text{true} \mid \text{false} \mid \text{not } b \mid b_1 op_b \, b_2 \mid a_1 op_r \, a_2$$

  - Boolean operators: $op_b \in \{and, or\}$
  - Relational operators: $op_r \in \{=,<,\leq,>,\geq,\neq\}$
- **Statements**:

$$S ::= [x := a]^l \mid [\textbf{skip}]^l \mid S1; S2 \mid \textbf{if}\, [b]^l \ S1 \ \textbf{else}\ S2 \mid \textbf{while}\ [b]^l\ S$$

- Note this abstract syntax, operator precedence and grouping statements is not covered. We can use { and } to group statements, and (and) to group expressions.

Computing the Control Flow Graph

- To calculate the CFG, we define some functions on the abstract syntax S:
  - The initial label (entry point) init: $S \rightarrow Lab$

$$init([x \coloneqq a]^l) = l$$
$$init([skip]^l) = l$$
$$init\ (S_1; S_2) = init\ (S_1)$$
$$init\ (if\ [b]^l\ \{S_1\}\ else\ \{S_2\} = l$$
$$init\ (while\ [b]^l\ \{\ S\ \} = l$$

  - The final labels (exit points) final: $S \rightarrow \mathbb{P}(Lab)$

$$final([x \coloneqq a]^l) = \{l\}$$
$$final\ ([skip]^l) = \{l\}$$
$$final\ (S_1; S_2) = final\ (S_2)$$
$$final(if\ [b]^l\ \{\ S_1\} else\ \{S_2\})$$
$$= final\ (S_1) \cup final\ (S_2)$$
$$final(while\ [b]^l\ \{S\}\ ) = \{l\}$$

  - The elementary blocks $blocks: S \rightarrow \mathbb{P}(Blocks)$ where an elementary block is an assignment [x:= a], or [skip], or a test [b]

$$blocks([x \coloneqq a]^l) = \{[x \coloneqq a]^l\}$$
$$blocks([skip]^l) = \{\ [skip]^l\}$$
$$blocks(S_1; S_2) = blocks(S_1) \cup blocks(S_2)$$
$$blocks(if\ [b]^l\ \{S_1\}\ else\ \{S_2\})$$
$$= \{[b]^l\ \} \cup blocks(S_1) \cup blocks(S_2)$$
$$blocks(while\ [b]^l\ \{S\}) = \{[b]^l\} \cup blocks(S)$$

Computing the Control Flow Graph

- The control flow $\text{flow}: S \rightarrow \mathbb{P}(Lab \times Lab)$ and reverse control $\text{flow}^{\text{R}}: S \rightarrow \mathbb{P}(Lab \times Lab)$

$$flow\ ([x \coloneqq a]^l) = \emptyset$$
$$flow\ ([skip]^l) = \emptyset$$
$$flow\ (S_1; S_2) = flow\ (S_1) \cup flow\ (S_2) \cup \{(l, init(S_2)) \mid l \in final(S_1)\}$$
$$flow\ (if\ [b]^l\ \{S_1\}\ else\ \{S_2\ \}\ ) = flow\ (S_1) \cup flow(S_2) \cup \{(l, init(S_1)), (l, init(S_2))\}$$
$$flow\ (while\ ([b]^l\ \{S\}\ ) = flow(S) \cup \{(l, init\ (S))\} \cup \{(l', l) | l' \in final(S)\}$$

$$flow^R(S) = \{(l', l) |\ (l, l') \in flow(S)\}$$

- The **control flow graph** of a program S is given by
  - elementary blocks $block(S)$ as nodes, and
  - *flow(S)* as vertices.
- Additional useful definitions

$labels(S) = \{l \mid [B]^l \in blocks(S)\}$

$FV(a)$ = free variables in a

$Aexp(S)$ = non-trival subexpressions in S (variables and constants are trivial)

An Example Program

P = [x := a+b][1]; [y := a*b][2]; while [y > a+b][3] { [a:=a+1][4]; [x:= a+b][5] }

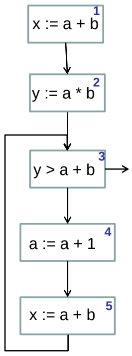
init(P) = 1
final(P) = {3}

blocks(P) =
{ [x := a+b][1], [y := a*b][2], [y > a+b][3], [a:=a+1][4], [x:= a+b][5]}

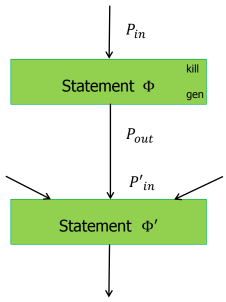
flow(P) = {(1, 2), (2, 3), (3, 4), (4, 5), (5, 3)}
flow^R(P) = {(2, 1), (3, 2), (4, 3), (5, 4), (3, 5)}

labels(P) = {1, 2, 3, 4, 5)

FV(a+b) = {a, b} -- Free variables
FV(P) = {a, b, x, y}
Aexp(P) = {a+b, a*b, a+1} -- Available expressions



Program Analysis CFG : General Idea



Locally for each statement:

Relationship between P_{in} and P_{out}:

- kill : part of P_{in} that is invalidated by Φ
- gen : additional part that is generated by Φ

$$P_{out} = (P_{in} \setminus kill) \cup gen$$

Globally for each link:

$$P'_{in} = \bigcup P_{out} \text{ or } P'_{in} = \bigcap P_{out}$$

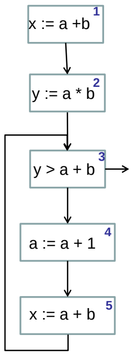
We obtain constraints for P_{in} and P_{out} for all statements and links.

Solve CSP by a constraint solver.

Available Expression Analysis

- The available expression analysis will determine for each program point:
  - which non-trivial expressions have been already computed in prior statements (and are still valid)
  - „Caching of expressions"
- *Forwards* analysis

S :



Available Expression Analysis

gen([x :=a]^l) = *{ exp ∈ Aexp(a) | x ∉ FV(exp) }*
gen([skip]^l) = ∅
gen([b]^l) = *Aexp(b)*
kill([x :=a]^l) = *{ exp ∈ Aexp(S) | x ∈ FV(exp) }*
kill([skip]^l) = ∅
kill([b]^l) = ∅

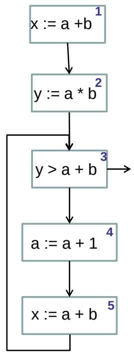
$$AE_{in}(l) = \begin{cases} \emptyset, \text{ if } l \in init(S) \\ \bigcap \{ AE_{out}(l') \mid (l', l) \in flow(S)\}, \text{ otherwise} \end{cases}$$

$$AE_{out}(l) = \left(AE_{in}(l) \setminus kill(B^l)\right) \cup gen(B^l), \text{ where } B^l \in blocks(S)$$

l	kill(B^l)	gen(B^l)
1	∅	{a+b}
2	∅	{a*b}
3	∅	{a+b}
4	{a+b, a*b, a+1}	∅
5	∅	{a+b}

l	AE_{in}	AE_{out}
1	∅	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	∅
5	∅	{a+b}

S :

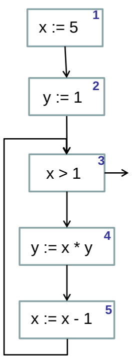


Reaching Definitions Analysis

- Reaching definitions (assignment) analysis determines if:
  - An assignment of the form *[x := a]^l* reaches a program point *k*

    if **there is** an execution path where *x* was last assigned at *l* when the program reaches *k*
- *Forwards* analysis

S :



Reaching Definitions Analysis

gen([x :=a]^l) = $\{ (x, l)\}$
gen([skip]^l) = ∅
gen([b]^l) = ∅
kill([skip]^l) = ∅
kill([b]^l) = ∅
kill([x :=a]^l) =
$\{(x, ?)\} \cup \{ (x, k) \mid B^k \text{ is an assigment in } S\}$

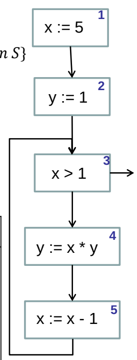
$$RD_{in}(l) = \begin{cases} \{ (x, ?) \mid x \in FV(S)\} \text{ if } l \in init(S) \\ \bigcup \{ RD_{out}(l') \mid (l', l)\} \in flow(S) \text{ otherwise} \end{cases}$$

$$RD_{out}(l) = \left(RD_{in}(l) \setminus kill(B^l)\right) \cup gen(B^l) \text{ where } B^l \in blocks(S)$$

l	kill(B^l)	gen(B^l)
1	{(x,?), (x,1),(x,5)}	{(x, 1)}
2	{(y,?), (y,2),(y,4)}	{(y, 2)}
3	∅	∅
4	{(y,?), (y,2),(y,4)}	{(y, 4)}
5	{(x,?), (x,1),(x,5)}	{(x, 5)}

l	RD_{in}	RD_{out}
1	{(x,?), (y,?)}	{(x,1), (y,?)}
2	{(x,1), (y,?)}	{(x,1), (y,2)}
3	{(x,1), (x,5), (y,2), (y,4)}	{(x,1), (x,5), (y,2), (y,4)}
4	{(x,1), (x,5), (y,2), (y,4)}	{(x,1), (x,5),(y,4)}
5	{(x,1), (x,5),(y,4)}	{(x,5),(y,4)}

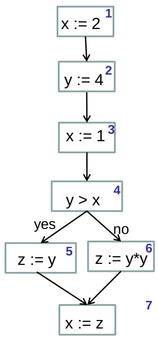
S :



Live Variables Analysis

- A variable x is **live** at some program point (label *l*) if there exists if there exists a path from *l* to an exit point that does not change the variable
- Live Variables Analysis determines:
  - for each program point, which variables *may* be still live at the exit from that point.
- Application: dead code elemination.
- *Backwards* analysis

S :



Live Variables Analysis

gen([x :=a]^l) = $FV(a)$
gen([skip]^l) = ∅
gen([b]^l) = $FV(b)$
kill([x :=a]^l) = $\{x\}$
kill([skip]^l) = ∅
kill([b]^l) = ∅

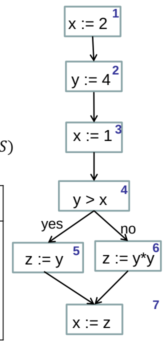
$$LV_{out}(l) = \begin{cases} \emptyset \quad \text{if } l \in final(S) \\ \bigcup \ \{LV_{in}(l') \mid (l', l) \in flow^R(S)\} \text{ otherwise} \end{cases}$$

$$LV_{in}(l) = \left(LV_{out}(l) \setminus kill(B^l)\right) \cup gen(B^l) \quad \text{where } B^l \in blocks(S)$$

l	kill(B^l)	gen(B^l)
1	{x}	∅
2	{y}	∅
3	{x}	∅
4	∅	{x, y}
5	{z}	{y}
6	{z}	{y}
7	{x}	{z}

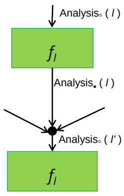
l	LV_{in}	LV_{out}
1	∅	∅
2	∅	{y}
3	{y}	{x, y}
4	{x, y}	{y}
5	{y}	{z}
6	{y}	{z}
7	{z}	∅

S :



First Generalized Schema

- $Analysis_\circ(l) = \begin{cases} \mathbf{EV} & \text{if } l \in \mathbf{E} \\ \square\{Analysis_\bullet(l') \mid (l', l) \in \mathbf{Flow}(S)\} & \text{otherwise} \end{cases}$
- $Analysis_\bullet(l) = f_l(Analysis_\circ(l))$



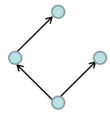
With:

- **EV** is the initial / final analysis information
- **E** is either {init(S)} or final(S)
- □ is either ⋃ or ⋂
- **Flow** is either flow or $flow^R$
- f_l is the transfer function associated with $B^l \in blocks(S)$

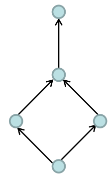
Forward analysis: **Flow** = flow, • = OUT, ∘ = IN
Backward analysis: **Flow** = $flow^R$, • = IN, ∘ = OUT

Partial Order

- $L = (M, \sqsubseteq)$ is a partial order iff
  - Reflexivity: $\forall x \in M. x \sqsubseteq x$
  - Transitivity: $\forall x, y, z \in M. x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  - Anti-symmetry: $\forall x, y \in M. x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$



- Let $L = (M, \sqsubseteq)$ be a partial order, $S \subseteq M$
  - $y \in M$ is upper bound for S ($S \sqsubseteq y$) iff $\forall x \in S. x \sqsubseteq y$
  - $y \in M$ is lower bound for S ($y \sqsubseteq S$) iff $\forall x \in S. y \sqsubseteq x$
  - Least upper bound $\bigsqcup X \in M$ of $X \subseteq M$:
    - $X \sqsubseteq \bigsqcup X \wedge \forall y \in M. X \sqsubseteq y \Rightarrow \bigsqcup X \sqsubseteq y$
  - Greatest lower bound $\sqcap X$ of $X \subseteq M$:
    - $\sqcap X \sqsubseteq X \wedge \forall y \in M. y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$



Lattice

A **lattice** ("Verband") is a partial order L = (M, ⊑) such that

(1) ⊔X and ⊓X exist for all $X \subseteq L$

(2) Unique greatest element $\top = \bigsqcup L$
(3) Unique least element $\bot = \sqcap L$

(1) Alternatively (for finite M), binary operators ⊔ and ⊓ ("meet" and "join") such that
$x, y \sqsubseteq x \sqcup y$ and $x \sqcap y \sqsubseteq x, y$

Transfer Functions

- Transfer functions to propagate information along the execution path (i.e. from input to output, or vice versa)
- Let $L = (M, \sqsubseteq)$ be a lattice. Let F be the set of transfer functions of the form
  $f_l : M \to M$ with l being a label
- Knowledge transfer is monotone
  - $\forall x, y.\ x \sqsubseteq y \Longrightarrow f_l(x) \sqsubseteq f_l(y)$
- Space $\boldsymbol{F}$ of transfer functions
  - $\boldsymbol{F}$ contains all transfer functions f_l
  - $\boldsymbol{F}$ contains the identity function id $\forall x \in M.\ id(x) = x$
  - $\boldsymbol{F}$ is closed under composition $\forall f, g \in \boldsymbol{F}.\ (g \circ f) \in \boldsymbol{F}$

The Generalized Analysis

- $Analysis_\circ(l) = \bigsqcup \{Analysis_\bullet(l') \mid (l', l) \in F\} \sqcup \{\iota_E^l\}$

  with $\iota_E^l = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{otherwise} \end{cases}$

- $Analysis_\bullet(l) = f_l(Analysis_\circ(l))$

With:

- M property space representing data flow information with $(M, \sqsubseteq)$ being a lattice
- A space F of transfer functions f_l and a mapping f from labels to transfer functions in F
- F is a finite flow (i.e. $flow$ or $flow^R$)
- ι is an extremal value for the extremal labels E (i.e. $\{init(S)\}$ or $final(S)$)

Instances of Framework

	Available Expr.	Reaching Def.	Live Vars.
M	$\mathscr{P}$(AExpr)	$\mathscr{P}$(Var x L)	$\mathscr{P}$(Var)
⊑	⊇	⊆	⊆
⊔	∩	∪	∪
⊥	AExpr	∅	∅
ι	∅	{(x, ?) | x ∈ FV(S)}	∅
E	{ init(S) }	{ init(S) }	final(S)
F	flow(S)	flow(S)	$flow^R$(S)
$\boldsymbol{F}$	{ f : $M \to M$ | $\exists\ m_k, m_g$. f(m) = ($m \setminus m_k$) ∪ m_g }		
f_l	$f_l(m)$ = ($m \setminus$ kill(B^l)) ∪ gen(B^l) where $B^l \in$ blocks(S)		

Limitations of Data Flow Analysis

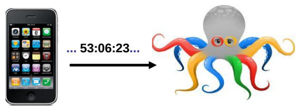
- The general framework of data flow analysis treats all outgoing edges **uniformly**. This can be a problem if conditions influence the property we want to analyse.
- Example: show no division by 0 can occur.
- Property space:
  - $M_0 = \{\bot, \{0\}, \{1\}, \{0,1\}\}$ (ordered by inclusion)
  - $M = Loc \to M_0$ (ordered pointwise)
  - $app_\sigma(t) \in M_0$ „approximate evaluation" of t under $\sigma \in M$
  - $cond_\sigma(b) \in M$ strengthening of $\sigma \in M$ under condition b
  - $gen\,[x = a] = \sigma[x \mapsto app_\sigma(a)]$
  - Kill needs to distinguish wether cond'n holds:
    $kill[b]_\sigma^{if} = cond_\sigma(b)$ $kill[b]_\sigma^{then} = cond_\sigma(!\ b)$
- This leads us to **abstract interpretation.**

Summary

- Static Program Analysis is the analysis of run-time behavior of programs without executing them (sometimes called static testing)
- Approximations of program behaviors by analyzing the program's CFG
- Analysis include
  - available expressions analysis
  - reaching definitions
  - live variables analysis
  - program slicing
- These are instances of a more general framework
- These techniques are used commercially, e.g.
  - AbsInt aiT (WCET)
  - Astrée Static Analyzer (C program safety)

Program Analysis for Information Flow Control

Confidentiality as a property of dependencies:



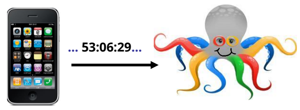
- The GPS data 53:06:23 N 8:51:08 O is confidential.
- The information on the GPS data must not leave Bob's mobile phone
- First idea: 53:06:23 N 8:51:08 O does not appear (explicitly) on the output line.
  - too strong, too weak
- Instead: The output of Bob's smart phone does not **depend** on the GPS setting
  - Changing the location (e.g. to 53:06:29 N 8:51:04 O) will not change the observed output of Bob's smart phone

Note: Confidentiality is formalized as a notion of dependability.

Confidentiality as Dependability

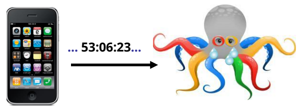
Confidential action:

change location (from 53:06:23 N 8:51:08 O) to 53:06:29 N 8:51:04 O



Insecure system:
output 53:06:29 depends on GPS data

Secure System:
output 53:06:23 does not depend on GPS data



Program Slicing

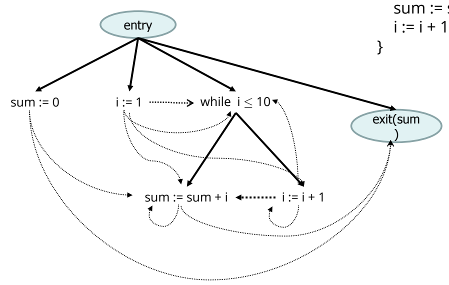
- Which parts of the program compute the message ?
- Do these parts contain GPS data ?
  - If yes: GPS data influence message (data leak)
  - If no: message is independent of GPS data

- Program Dependence Graph
  - Nodes are statements and conditions of a program
  - Links are either
    - Control dependences (similar to CFG)
    - Data flow dependences (connecting assignment with usage of variables)

Example

⟶ Control dependences
⇢ Data flow dependences

```
sum := 0;
i := 1;
while  i ≤ 10 {
   sum := sum + i;
   i := i + 1
}
```



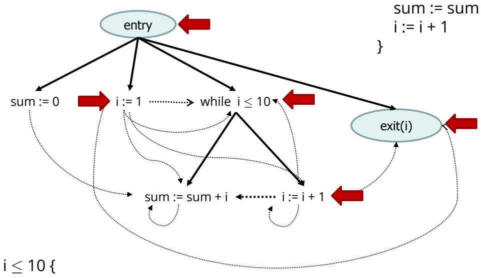
Backward Slice

- Let G be a program dependency graph and
- S be subset of nodes in G
- Let n ⇒ m := n ⟶ m ∨ n ⇢ m
- Then, the backward slice BS(G, S) is a graph G′ with
  - N(G′) = { n | n ∈ N(G) ∧ ∃ m ∈ S. n ⇒* m }
  - E(G′) = {n ⟶ m | n ⟶ m ∈ E(G) ∧ n, m ∈ N(G′) } ∪
    {n ⇢ m | n ⇢ m ∈ E(G) ∧ n, m ∈ N(G′) }
- Backward slice BS(G, S) computes same values for variables occurring in S as G itself

Example

⟶ Control dependences
⇢ Data flow dependences

```
sum := 0;
i := 1;
while  i ≤ 10 {
   sum := sum + i;
   i := i + 1
}
```



BS:

```
i := 1;
while  i ≤ 10 {
   i := i + 1
}
```

Universität Bremen

Systeme hoher Sicherheit und Qualität

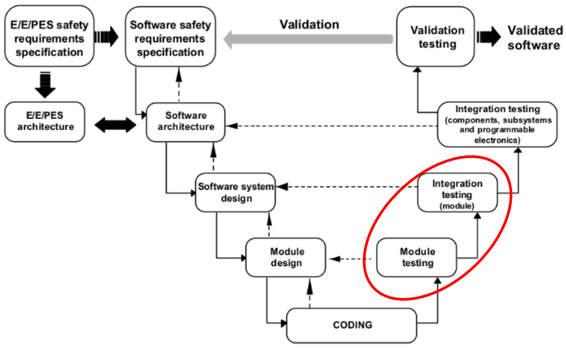
WS 2019/2020

Lecture 09:
Software Verification
with Floyd-Hoare Logic

Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

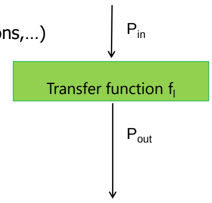
Software Verification in the Development Cycle



Static Program Analysis

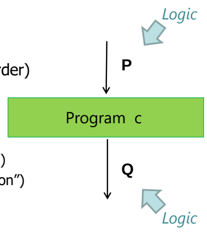
Transfer functions to propagate information along the execution path (i.e. from input to output, or vice versa)

- Information is encoded as a lattice $L = (M, \sqsubseteq)$.
- Transfer functions mapping information
  - $f_l: M \rightarrow M$ with l being a label
  - Knowledge transfer is monotone $\forall x, y.\ x \sqsubseteq y \Longrightarrow f_l(x) \sqsubseteq f_l(y)$
  - Restricted to a specific type of knowledge (Reachable Definitions, Available Expressions,…)
- What about a more general approach
  - Maintaining arbitrary knowledge ?
  - Knowledge representation ?



General Transfer Relations

- Transfer relations:
  - Knowledge P, Q is represented in logic (first-order)
  - {P} c {Q} denotes
    If P is known before executing c (and c terminates)
    then Q is known (P "precondition", Q "postcondition")
  - {P} c {Q} are called Floyd-Hoare triples



Charles Antony Richard Hoare: An axiomatic basis for computer programming (1969)
Robert W Floyd: Assigning Meanings to Programs (1967)

Software Verification

- Software Verification **proves** properties of programs. That is, given the basic problem of program P satisfying a property p we want to show that for **all possible inputs and runs** of P , the property p holds.
- Software verification is far **more powerful** than static analysis. For the same reasons, it cannot be fully automatic and thus requires user interaction. Hence, it is **complex to use**.
- Software verification does not have false negatives, only failed proof attempts. If we can prove a property, it holds.
- Software verification is used in **highly critical systems**.

The Basic Idea

- What does this program compute?
  - The index of the maximal element of the array a if it is non-empty.
- How to prove it?
  - (1) We need a language in which to **formalise** such **assertions**.
  - (2) We need a notion of meaning (**semantics**) for the program.
  - (3) We need to way to **deduce valid assertions**.
- Floyd-Hoare logic provides us with (1) and (3).

```
i := 0;
x := 0;
while (i < n) {
  if (a[i] ≥ a[x]) {
    x := i;
  }
  i := i + 1;
}
```

Formalizing correctness:

$\mathrm{array}(a, n) \wedge n > 0 \Longrightarrow$
$a[x] = \max(a, n)$
$\forall i.\ 0 \le i < n \Longrightarrow$
$a[i] \le \max(a, n)$
$\exists j.\ 0 \le j < n \Longrightarrow$
$a[j] = \max(a, n)$

Recall our simple programming language

- **Arithmetic** expressions:

$$a \ ::= x \mid n \mid a_1[a_2] \mid a_1\ op_a\ a_2$$

  - Arithmetic operators: $op_a \in \{+, -, *, /\}$
- **Boolean** expressions:

$$b \ ≔ \text{true} \mid \text{false} \mid \text{not}\ b \mid b_1 op_b\ b_2 \mid a_1 op_r\ a_2$$

  - Boolean operators: $op_b \in \{and, or\}$
  - Relational operators: $op_r \in \{=, <, \le, >, \ge, \neq\}$
- **Statements**:

$$S ::= x := a \mid \textbf{skip} \mid S1; S2 \mid \textbf{if}\ (b)\ S1\ \textbf{else}\ S2 \mid \textbf{while}\ (b)\ S$$

  - Labels from basic blocks omitted, only used in static analysis to derive cfg.
  - Note this abstract syntax, operator precedence and grouping statements is not covered.

Semantics of our simple language

- The semantics of an **imperative** language is state transition: the program has an ambient state, which is changed by assigning values to certain locations.
- Example:

x	?
y	12
z	?

σ —— x := 5 ——→

x	5
y	12
z	?

$\sigma_1 = \sigma[x/5]$ —— z := x + y ——→

x	5
y	12
z	17

$\sigma_2 = \sigma_1[z/17] = \sigma[x/5, z/17]$ —— x := x + 1 ——→

x	6
y	12
z	17

$\sigma_3 = \sigma_2[x/6] = \sigma[x/6, z/17]$

- Semantics in a nutshell:

> **Expressions** evaluate to values Val (for our language integers).
> **Locations** Loc are variable names.
> A **program state** maps locations to values: $\Sigma = Loc \rightharpoonup Val$
> A program maps an initial state to a final state, **if it terminates**.
> **Assertions** are predicates over program states.

Semantics in a nutshell

- There are three major ways to denote semantics.

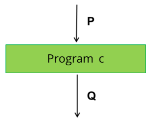
(1) As a relation between program states, described by an abstract machine (**operational semantics**).
(2) As a function between program states, defined for each statement of the programming langauge (**denotational semantics**).
(3) As the set of all assertions which hold for a program (**axiomatic semantics**).

- Floyd-Hoare logic covers the third aspect, but it is important that all three semantics agree.
  - We will not cover semantics in detail here, but will concentrate on how to **use** Floyd-Hoare logic to prove correctness.

Extending our simple language

- We introduce a set Var of **logical variables.**
- **Assertions** are boolean expressions, which may not be executable, and arithmetic expressions containing logical variables.
- Arithmetic assertions
  $$ae ::= x \mid X \mid n \mid ae_1[ae_2] \mid ae_1\ op_a\ ae_2 \mid f(ae_1, \ldots, ae_n)$$
  - where $x \in Loc, X \in Var, op_a \in \{+,-,*,/\}$
- Boolean assertions:
  $$be := \text{true} \mid \text{false} \mid \text{not}\ be \mid be_1 op_b\ be_2 \mid ae_1 op_r\ ae_2 \mid p(ae_1, \ldots, ae_n) \mid \forall X.be \mid \exists X.be$$
  - Boolean operators: $op_b \in \{\wedge, \vee, \Longrightarrow\}$
  - Relational operators: $op_r \in \{=, <, \leq, >, \geq, \neq\}$

Floyd-Hoare Triples



The basic build blocks of Floyd-Hoare logic are Hoare triples of the form $\{P\}c\,\{Q\}$.

- P, Q are assertions using variables in Loc and Var
  - e.g. x < 5 + y, Odd(x), …
- A state σ satisfies P (written $\sigma \vDash P$) iff $P[^{\sigma(x)}/_x]$ is true for all $x \in Loc$ and all possible values for $X \in Var$:
  - e.g. let $\sigma =$

x	5
y	12
z	17

then σ satisfies x < 5 + y, Odd(x)

- A formula P describes a set of states, i.e. all states that satisfy the formula P.

Partial and Total Correctness

- **Partial correctness**: $\vDash \{P\}c\{Q\}$
  - c is partial correct with precondition P and postcondition Q iff, for all states σ which satisfy P and for which the execution of c terminates in some state σ' then it holds that σ' satisfies Q:
    $$\forall \sigma. \sigma \vDash P \wedge \exists \sigma'. \langle \sigma, c \rangle \to \sigma' \Longrightarrow \sigma' \vDash Q$$
- **Total correctness**: $\vDash [P]c[Q]$
  - c is total correct with precondition P and postcondition Q iff, for all states σ which satisfy P the execution of c terminates in some state σ' which satisfies Q:
    $$\forall \sigma. \sigma \vDash P \Longrightarrow \exists \sigma'. \langle \sigma, \quad c \rangle \to \sigma' \wedge \sigma' \vDash Q$$
- Examples: $\vDash \{true\} while(true)\ skip\ \{true\}$,
  $\nvDash [true]\ while(true) skip\ [true]$

Reasoning with Floyd-Hoare Triples

- How do we know that $\vDash \{P\}c\{Q\}$ in practice ?
- Calculus to derive triples, written as $\vdash \{P\}c\{Q\}$
  - Rules operate along the constructs of the programming language (cf. operational semantics)
  - Only one rule is applicable for each construct (!)
  - Rules are of the form
    $$\frac{\vdash \{P_1\}c_1\{Q_1\}, \ldots, \vdash \{P_n\}c_n\{Q_n\}}{\vdash \{P\}c\,\{Q\}}$$
    meaning we can derive $\vdash \{P\}c\{Q\}$ if all $\vdash \{P_i\}c_i\{Q_i\}$ are derivable.

Floyd-Hoare Rules: Assignment

- Assignment rule:
  $$\frac{}{\vdash \{P[^e/_x]\}\ x := e\ \{P\}}$$
- $P[^e/_x]$ replaces all occurrences of the program variable x by the arithmetic expression e.
- Examples:
  - $\vdash \{0 < 10\}\, x := 0\, \{x < 10\}$
  - $\vdash \{\underbrace{x - 1 < 10}_{x < 11}\}\, x := x - 1\, \{x < 10\}$
  - $\vdash \{\underbrace{x + 1 + x + 1 < 10}_{x + x < 8}\}\ x := x + 1\, \{x + x < 10\}$

Rules: Sequencing and Conditional

- Sequence:
  $$\frac{\vdash \{P\}\ c_1\ \{Q\} \quad \vdash \{Q\}\ c_2\ \{R\}}{\vdash \{P\}\ c_1; c_2\ \{R\}}$$
  - Needs an intermediate state predicate Q.
- Conditional:
  $$\frac{\vdash \{P \wedge b\}\ c_1\ \{Q\} \quad \vdash \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}(b)\ c_1 \textbf{else}\ c_2\ \{Q\}}$$
  - Two preconditions capture both cases of b and $\neg\, b$.
  - Both branches end in the same postcondition Q.

Rules: Iteration and Skip

$$\frac{\vdash \{P \wedge b\}\ c\ \{P\}}{\vdash \{P\}\ \textbf{while}\ (b)\ c\ \{P \wedge \neg\, b\}}$$

- P is called the **loop invariant**. It has to hold both before and after the loop (but not necessarily in the whole body).
- Before the loop, we can assume the loop condition b holds.
- After the loop, we know the loop condition b does not hold.
- In practice, the loop invariant has to be **given**– this is the creative and difficult part of working with the Floyd-Hoare calculus.

$$\frac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}}$$

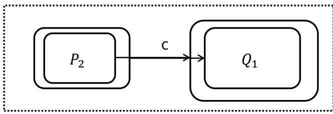
- **skip** has no effect: pre- and postcondition are the same.

Final Rule: Weakening

- Weakening is crucial, because it allows us to change pre- or postconditions by applying rules of logic.

$$\frac{P_2 \Longrightarrow P_1 \quad \vdash \{P_1\}\ c\ \{Q_1\} \quad Q_1 \Longrightarrow Q_2}{\vdash \{P_2\}\ c\ \{Q_2\}}$$

- We can **weaken** the precondition and **strengthen** the postcondition:
  - $P \Longrightarrow Q$ means that all states in which P holds, Q also holds.
  - $\models \{P\}c\{Q\}$ means whenever c starts in a state in which P holds, it ends in a state in which Q holds.
  - So, we can reduce the starting set, and enlarge the target set.



How to derive and denote proofs

// $\{P\}$
// $\{P_1\}$
x:= e;
// $\{P_2\}$
// $\{P_3\}$
while (x< n) {
  // $\{P_3 \wedge x < n\}$
  // $\{P_4\}$
  z := a
  // $\{P_3\}$
}
// $\{P_3 \wedge \neg(x < n)\}$
// $\{Q\}$

- The example shows $\vdash \{P\}c\{Q\}$
- We annotate the program with valid assertions: the precondition in the preceding line, the postcondition in the following line.
- The sequencing rule is applied implicitly.
- Consecutive assertions imply weaking, which has to be proven separately.
  - In the example:
    $P \Longrightarrow P_1$,
    $P_2 \Longrightarrow P_3$,
    $P_3 \wedge x < n \Longrightarrow P_4$,
    $P_3 \wedge \neg(x < n) \Longrightarrow Q$

More Examples

P ==
 $p \coloneqq 1;$
 $c \coloneqq 1;$
 while $(c \leq n)$ {
  $p \coloneqq p * c;$
  $c \coloneqq c + 1$
 }

Specification:
 $\vdash \{1 \leq n\}$
 P
 $\{p = n!\}$

Invariant:
 $p = (c-1)!$

Q ==
 $p \coloneqq 1;$
 while $(0 < n)$ {
  $p \coloneqq p * n;$
  $n \coloneqq n - 1$
 }

Specification:
 $\vdash \{1 \leq n \wedge n = N\}$
 Q
 $\{p = N!\}$

Invariant:
 $p = \prod_{i=n+1}^{N} i$

R ==
 $r \coloneqq a;$
 $q \coloneqq 0;$
 while $(b \leq r)$ {
  $r \coloneqq r - b;$
  $q \coloneqq q + 1$
 }

Specification:
 $\vdash \{a \geq 0 \wedge b \geq 0\}$
 R
 $\{a = b * q + r \wedge 0 \leq r \wedge r < b\}$

Invariant:
 $a = b * q + r \wedge 0 \leq r$

How to find an Invariant

- Going backwards: try to split/weaken postcondition Q into negated loop-condition and „something else" which becomes the invariant.
- Many while-loops are in fact for-loops, i.e. they count uniformly:

$i \coloneqq 0;$
while $(i < n)$ {
 …;
 $i \coloneqq i + 1$
}

- In this case:
  - If post-condition is $P(n)$, invariant is $P(i) \wedge i \leq n$.
  - If post-condition is $\forall j. 0 \leq j < n. P(j)$ (uses indexing, typically with arrays), invariant is $\forall j. j \leq 0 < i. i \leq n \wedge P(j)$.

Summary

- Floyd-Hoare-Logic allows us to **prove** properties of programs.
- The proofs cover all possible inputs, all possible runs.
- There is **partial** and **total correctness**:
  - Total correctness = partial correctness + termination.
- There is one rule for each construct of the programming language.
- Proofs can in part be constructed automatically, but iteration needs an **invariant** (which cannot be derived mechanically).
- Next lecture: correctness and completeness of the rules.

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 10:

Verification Condition Generation

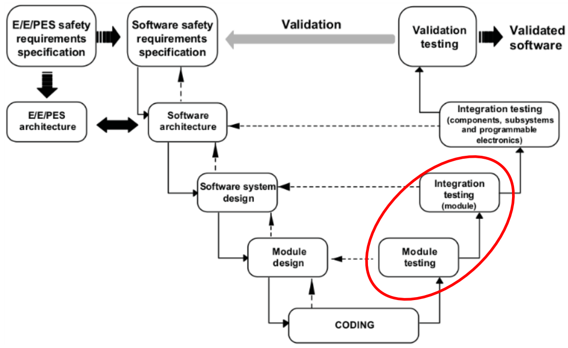
Christoph Lüth, Dieter Hutter, Jan Peleska

Frohes Neues Jahr!

Where are we?

- 01: Concepts of Quality
- 02: Legal Requirements: Norms and Standards
- 03: The Software Development Process
- 04: Hazard Analysis
- 05: High-Level Design with SysML
- 06: Formal Modelling with OCL
- 07: Testing
- 08: Static Program Analysis
- 09: Software Verification with Floyd-Hoare Logic
- 10: Verification Condition Generation
- 11-12: Model Checking
- 13: Conclusions

VCG in the Development Cycle



Introduction

- In the last lecture, we introduced Hoare triples. They allow us to state and prove correctness assertions about programs, written as $\{P\}\,p\,\{Q\}$
- We introduced two notions, namely:
  - Syntactic derivability, $\vdash \{P\}\,p\,\{Q\}$ (the actual Floyd-Hoare calculus)
  - Semantic satisfaction, $\vDash \{P\}\,p\,\{Q\}$
- Question: how are the two related?
- The answer to that question also offers help with a practical problem: proofs with the Floyd-Hoare calculus are exceedingly long and tedious. Can we automate them, and how?

Correctness and Completeness

- In general, given a syntactic calculus with a semantic meaning, **correctness** means the syntactic calculus implies the semantic meaning, and **completeness** means all semantic statements can be derived syntactically.
  - Cf. also Static Program Analysis
- **Correctness** should be a basic property of verification calculi.
- **Completeness** is elusive due to Gödel's first incompleteness theorem:
  - Any logics which is strong enough to encode the natural numbers and primitive recursion* is incomplete.**

* Or any other notion of computation.
** Or inconsistent, which is even worse.

Correctness of the Floyd-Hoare calculus

Theorem (Correctness of the Floyd-Hoare calculus)

If $\vdash \{P\}\,p\,\{Q\}$, then $\vDash \{P\}\,p\,\{Q\}$.

- Proof: by induction on the derivation of $\vdash \{P\}\,p\,\{Q\}$.
- More precisely, for each rule we show that:
  - If the conclusion is $\vdash \{P\}\,p\,\{Q\}$, we can show $\vDash \{P\}\,p\,\{Q\}$
  - For the premisses, this can be assumed.
- Example: for the assignment rule, we show that

Completeness of the Floyd-Hoare calculus

- Predicate calculus is incomplete, so we cannot hope F/H is complete. But we get the following:

Theorem (Relative completeness)
If $\vDash \{P\}\,p\,\{Q\}$, then $\vdash \{P\}\,p\,\{Q\}$ *except* for the proofs occuring in the weakenings.

- To show this, we construct the **weakest precondition**.

Weakest precondition
Given a program c and an assertion P, the weakest precondition $wp(c, P)$ is an assertion W such that
1. W is a valid precondition $\vDash \{W\}\,c\,\{P\}$
2. And it is the weakest such: for any other Q such that $\vDash \{Q\}\,c\,\{P\}$, we have $W \rightarrow Q$.

Constructing the weakest precondition

- Consider a simple program and its verification:

$\{\, x = X \wedge y = Y \}$
$\leftrightarrow$
$\{\, y = Y \wedge x = X \}$
```
z := y;
```
$\{\, z = Y \wedge x = X \}$
```
y := x;
```
$\{\, z = Y \wedge y = X \}$
```
x := z;
```
$\{\, x = Y \ \wedge y = X \}$

- Note how proof is **constructed backwards systematically**.
- The idea is to construct the weakest precondition inductively.
- This also gives us a methodology to automate proofs in the calculus.

Constructing the weakest precondition

- There are four straightforward cases:
  - (1) $wp(\mathbf{skip}, P) = P$
  - (2) $wp(X \coloneqq e, P) = P\,[e\,/\,X]$
  - (3) $wp(c_0; c_1, P) = wp(c_0, wp(c_1, P))$
  - (4) $wp(\mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, P) = (b \wedge wp(c_0, P)) \vee (\neg\, b \wedge wp(c_1, P))$
- The complicated one is iteration (unsurprisingly, since it is the source of the computational power and Turing-completeness of the language). It can be given recursively:
  - (5) $wp(\mathbf{while}\ b\ \{c\}, P) = (\neg\, b \wedge P) \vee wp\big(c, wp\ (\mathbf{while}\ b\ \{c\}, P)\big)$
- A closed formula can be given, but it can be infinite and is not practical. It shows the relative completeness, but does not give us an effective way to automate proofs.
- Hence, $wp(c, P)$ is not effective for proof automation, but it shows the right way: we just need something for iterations.

Verification Conditions: Annotations

- The idea is that we have to give the invariants manually by annotating them.
- We need a language for this:
  - Arithmetic expressions and boolean expressions stays as they are.
  - Statements are augmented to **annotated statements**:

    S ::= x := a | **skip** | S1; S2 | **if** (b) S1 **else** S2
    | **assert** P | **while** (b) **inv** P S
  - Each while loop needs to its invariant annotated.
    - This is for partial correctness, total correctness also needs a **variant**: an expression which is strictly decreasing in a well-founded order such as $(<, \mathbb{N})$ after the loop body.
  - The assert statement allows us to force a weakening.

Preconditions and Verification Conditions

- We are given an annotated statement c, a precondition P and a postcondition Q.
  - We want to know: when does $\vDash \{P\}\ c\ \{Q\}$ hold?
- For this, we calculate a **precondition** $pre(c, Q)$ and a **set** of **verification conditions** $vc(c, Q)$.
  - The idea is that if all the verification conditions hold, then the precondition holds:

$$\bigwedge_{R \in vc(c,Q)} R \Rightarrow \vDash \{pre(c, Q)\} c\ \{Q\}$$
  - For the precondition P, we get the additional weaking $P \Rightarrow pre(c, Q)$.

Calculation Verification Conditions

- Intuitively, we calculate the verification conditions by stepping through the program backwards, starting with the postcondition Q.
- For each of the four simple cases (assignment, sequencing, case distinction and ***skip***), we calculate new current postcondition Q
- At each iteration, we calculate the precondition R of the loop body working backwards from the invariant I, and get two verification conditions:
  - The invariant I and negated loop condition implies Q.
  - The invariant I and loop condition implies R.
- Asserting R generates the verification condition $R \Rightarrow Q$.
- Let's try this.

Example: deriving VCs for the factorial.

{ 0 <= n }
{ 1 == (1-1)! && (1- 1) <= n }
p := 1;
{ p == (1-1)! && (1- 1) <= n }
c := 1;
{ p == (c-1)! && (c- 1) <= n }
while (c <= n)
 inv (p == (c-1)! && c-1 <= n) {
 { p*c == ((c+1)-1)! &&
 ((c+1)- 1) <= n }
 p := p* c;
 { p == ((c+1)-1)! && ((c+1)- 1) <= n }
 c := c+1;
 { p == (c-1)! && (c- 1) <= n }
 }
{ p = n! }

VCs (unedited):
1. p == (c-1)! && (c- 1) <= n && ! (c <= n) ==> p= n!
2. p == (c-1)! && c-1 <= n && c<= n ==> p* c= ((c+1)-1)! && ((c+1)-1) <= n
3. 0 <= n ==> 1= (1-1)! && 1-1 <= n

VCs (simplified):
1. p == (c-1)! && c- 1 == n ==> p= n!
2. p == (c-1)! && c-1 <= n && c<= n ==> p* c= c!
3. p == (c-1)! && c-1 <= n && c<= n ==> c <= n
4. 0 <= n ==> 1= 0!
5. 0 <= n ==> 0 <= n

Formal Definition

- Calculating the precondition:

$pre(\mathbf{skip}, Q) = Q$
$pre(X \coloneqq e, Q) = Q\ [e\ /\ X]$
$pre(c_0; c_1, Q = pre(c_0, pre(c_1, Q))$
$pre(\mathbf{if}\ (b)\ c_0\ \mathbf{else}\ c_1, Q) = \big(b \wedge pre(c_0, Q)\big) \vee \big(\neg\, b \wedge pre(c_1, Q)\big)$
$pre(\mathbf{assert}\ R, Q) = R$
$pre\ (\mathbf{while}\ (b) \mathbf{inv}\ I\ c, Q) = I$

- Calculating the verification conditions:

$vc(skip, Q) = \emptyset$
$vc(X \coloneqq e, Q) = \emptyset$
$vc(c_0; c_1, Q) = vc\big(c_0, pre(c_1, Q)\big) \cup vc(c_1, Q)$
$vc(\mathbf{if}\ (b)\ c_0\ \mathbf{else}\ c_1, Q) = vc(c_0, Q) \cup vc(c_1, Q)$
$vc(\mathbf{while}\ (b)\ \mathbf{inv}\ I\ c, Q) = vc(c, I) \cup \{I \wedge b \Rightarrow pre(c, I), I \wedge \neg b \Rightarrow Q\}$
$vc(\mathbf{assert}\ R, Q) = \{R \Rightarrow Q\}$

- The main definition:

$$vcg(\{P\}\ c\ \{Q\}) = \{P \Rightarrow pre(c, Q)\} \cup vc(c, Q)$$

Another example: integer division

{ 0 <= a && 0 <= b }
{ 1 }
r := a;
{ 2 }
q := 0;
{ 3 }
while (b <= r)
 inv (a == b* q + r && 0 <= r) {
 { 4 }
 r := r- b;
 { 5 }
 q := q+1;
 { 6 }
 }
{ a == b* q + r && 0 <= r && r < b }

Correctness of VC

- The correctness calculus is correct: if we can prove all the verifcation conditons, the program is correct w.r.t to given pre- and postconditions.
- Formally:

> **Theorem** (Correctness of the VCG calculus)
> Given assertions P and Q (with P the precondition and Q the postcondition), and an annotated program, then
> $$\bigwedge_{R \in vcg(c,Q)} R \Rightarrow \models \{P\}\, c\, \{Q\}$$

- Proof: by induction on c.

Using VCG in Real Life

We have just a toy language, but VCG can be used in real life. What features are missing?

- **Modularity**: the language must have modularity concepts, e.g. functions (as in C), or classes (as in Java), and we must be able to verify them separately.
- **Framing**: in our simple calculus, we need to specify which variables stay the same (e.g. when entering a loop). This becomes tedious when there are a lot of variables involved; it is more practical to specify which variables may change.
- **References**: languages such as C and Java use references, which allow aliasing. This has to be modelled semantically; specifically, the assignment rule has to be adapted.
- **Machine arithmetic**: programs work with machine words and floating point representations, not integers and real numbers. This can be the cause of insidious errors.

VCG Tools

- Often use an intermediate language for VCG and front-ends for concrete programming languages.
- The Why3 toolset (`http://why3.lri.fr`)
  - A verification condition generator
  - Front-ends for different languages:
    C (Frama-C), Java (defunct?)
- Boogie (Microsoft Research)
  - Frontends for programming languages such C, C#, Java.
- VCC – a verifying C compiler built on top of Boogie
  - Interactive demo:
    `https://www.rise4fun.com/Vcc/`

VCC Example: Binary Search

- A correct (?) binary search implementation:

```
#include <limits.h>

unsigned int bin_search(unsigned int a [], unsigned int a_len, unsigned int key)
{
  unsigned int lo= 0;
  unsigned int hi= a_len;
  unsigned int mid;

  while (lo <= hi)
     {
      mid= (lo+ hi)/2;
      if (a[mid] < key) lo= mid+1;
      else hi= mid;
    }

  if (!(lo < a_len && a[lo] == key)) lo= UINT_MAX;

  return lo;
}
```


VCC: Correctness Conditions?

- We need to annotate the program.
- Precondition:
  - `a` is an array of length `a_len`;
  - The array `a` is sorted.
- Postcondition:
  - Let `r` be the result, then:
  - if `r` is `UINT_MAX`, all elements of `a` are unequal to `key`;
  - if `r` is not `UINT_MAX`, then `a[r] == key`.
- Loop invariants:
  - `hi` is less-equal to `a_len`;
  - everything „left" of `lo` is less then `key`;
  - everything „right" of `hi` is larger-equal to `key`.

VCC Example: Binary Search

- Source code as annotated for VCC:

```
#include <limits.h>
#include <vcc.h>
unsigned int bin_search(unsigned int a [], unsigned int a_len, unsigned int key)
  _(requires \thread_local_array(a, a_len))
  _(requires \forall unsigned int i, j; i < j && j < a_len ==> a[i] <= a[j])
  _(ensures \result != UINT_MAX ==> a[\result] == key)
  _(ensures \result == UINT_MAX ==> \forall unsigned int i; i < a_len ==> a[i] !=  key)
{
  unsigned int lo= 0;
  unsigned int hi= a_len;
  unsigned int mid;

  while (lo <= hi)
    _(invariant hi <= a_len)
    _(invariant \forall unsigned int i; i < lo ==> a[i] <  key)
    _(invariant \forall unsigned int i; hi <= i && i < a_len ==>a[i] >= key)
    {
      mid= (lo+ hi)/2;
      if (a[mid] < key) lo= mid+1;
      else hi= mid;
    }
  if (!(lo < a_len && a[lo] == key)) lo= UINT_MAX;
  return lo;
}
```


Binary Search: the Corrected Program

- Corrected source code:

```
#include <limits.h>
#include <vcc.h>
unsigned int bin_search(unsigned int a [], unsigned int a_len, unsigned int key)
  _(requires \thread_local_array(a, a_len))
  _(requires \forall unsigned int i, j; i < j && j < a_len ==> a[i] <= a[j])
  _(ensures \result != UINT_MAX ==> a[\result] == key)
  _(ensures \result == UINT_MAX ==> \forall unsigned int i; i < a_len ==> a[i] !=  key)
{
  unsigned int lo= 0;
  unsigned int hi= a_len;
  unsigned int mid;

  while (lo < hi)
    _(invariant hi <= a_len)
    _(invariant \forall unsigned int i; i < lo ==> a[i] <  key)
    _(invariant \forall unsigned int i; hi <= i && i < a_len ==>a[i] >= key)
    {
      mid= (hi-lo)/2+ lo;
      if (a[mid] < key) lo= mid+1;
      else hi= mid;
    }
  if (!(lo < a_len && a[lo] == key)) lo= UINT_MAX;
  return lo;
}
```


Summary

- Starting from the relative completeness of the Floyd-Hoare calculus, we devised a verification condition generation (vcg) calculus which makes program verification viable.
- Verification condition generation reduces the question whether the given pre/postconditions hold for a program to the validity of a set of logical properties.
  - We do need to annotate the while loops with invariants.
  - Most of these logical properties can be discharged with automated theorem provers.
- To scale to real-world programs, we need to deal with framing, modularity (each function/method needs to be verified independently), and machine arithmetic (integer word arithmetic and floating-points).

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 11:

Foundations of Model Checking

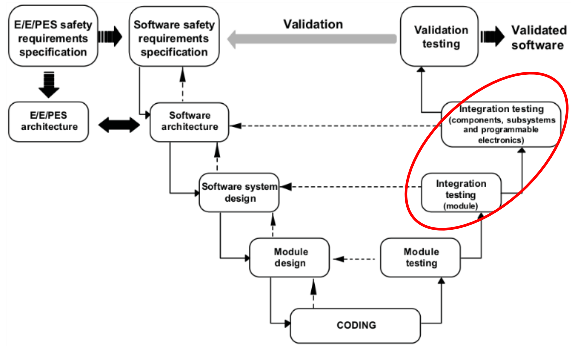
Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

Introduction

- In the last lectures, we were verifying program properties with the Floyd-Hoare calculus (or verification condition generation). Program verification translates the question of program correctness into a **proof** in program logic (the Floyd-Hoare logic), turning it into a **deductive problem**.
- Model-checking takes a different approach: instead of directly working with the (source code) of the program, we work with an **abstraction** of the system (the system **model**). Because we build an abstraction, this approach is also applicable at higher verification levels. (It is also complimentary to deductive verification.)
- The **key questions** are: how do these models look like? What properties do we want to express, and how do we express and prove them?

Model Checking in the Development Cycle



Introduction

- Model checking operates on (abstract) state machines
  - Does an abstract system satisfy some behavioral property e.g. liveness (deadlock) or safety properties
    - consider traffic lights in Requirement Engineering
    - Example: "green must always follow red"
- Automatic analysis if state machine is finite
  - Push-button technology
  - User does not need to know logic (at least not for the proof)
- Basis is satisfiability of boolean formula in a finite domain (SAT). However, finiteness does not imply efficiency – all interesting problems are at least NP-complete, and SAT is no exception (Cook's theorem).

The Model-Checking Problem

The **Basic Question**:

Given a model $\mathcal{M}$ and property ϕ, we want to know if

$$\mathcal{M} \vDash \phi$$

- What is $\mathcal{M}$?
  - A finite-state machine or Kripke structure.
- What is ϕ?
  - Temporal logic
- How to prove it?
  - By enumerating the states and thus construct a model (hence the term model checking)
  - The basic problem: state explosion

Finite State Machine (FSM)

Definition: Finite State Machine (FSM)

A FSM is given by $\mathcal{M} = \langle \Sigma, I, \rightarrow \rangle$ where
- Σ is a finite set of **states**,
- $I \subseteq \Sigma$ is a set of **initial** states, and
- $\rightarrow \subseteq \Sigma \times \Sigma$ is a **transition relation**, s.t. $\rightarrow$ is left-total:
  $$\forall s \in \Sigma. \exists s' \in \Sigma. s \rightarrow s'$$

- Variations of this definition exists, e.g. no initial states.
- Note there is no final state, and no input or output (this is the key difference to **automata**).
- If $\rightarrow$ is a function, the FSM is deterministic, otherwise it is non-deterministic.

First Example: A Simple Drink Dispenser

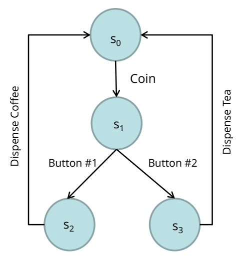
1) Insert a coin.
2) Press button: tea or coffee
3) Tea or coffee dispensed
4) Back to 1)

FSM:

$$\Sigma = \{ s_0, s_1, s_2, s_3 \}$$

$$I = \{ s_0 \}$$

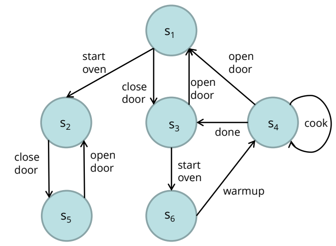
$$\rightarrow = \{ (s_0, s_1), (s_1, s_2), (s_2, s_3), (s_1, s_3), (s_2, s_0), (s_3, s_0) \}$$



Note operation names are for decoration purposes only.

Example: A Simple Oven

- The oven has more states and operations:
  open and close door,
  - turn oven on and off,
  - warm up and cook.
- How do they interact?
- FSM:



Questions to ask

We want to answer **questions** about the system **behaviour** like

- Can the cooker heat with the door open?
- When the start button is pushed, will the cooker eventually heat up?
- When the cooker is correctly started, will the cooker eventually heat up?
- When an error occurs, will it be still possible to cook?

We are interested in questions on the development of the system over time, i.e. possible **traces** of the system given by a succession of states.

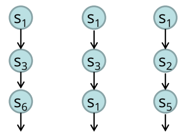
The tool to formalize and answer these questions is **temporal logic.**

Temporal Logic

Expresses properties of possible succession of states

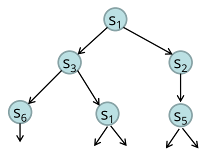
Linear Time

- Every moment in time has a unique successor
- Infinite sequences of moments
- Linear Temporal Logic LTL



Branching Time

- Every moment in time has several successors
- Infinite tree
- Computational Tree Logic CTL



Kripke Structures

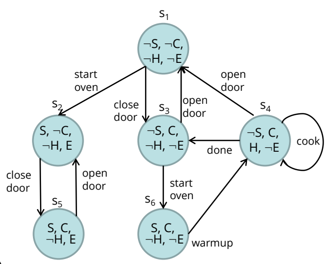
- In order to talk about propositions, we label the states of a FSM with propositions which hold there. This is called a **Kripke structure**.

Definition: Kripke structure
Given a set $Prop$ of **propositions**, then a Kripke structure is given by $K = \langle \Sigma, I, \rightarrow, V \rangle$ where
- Σ is a finite set of states,
- $I \subseteq \Sigma$ is a set of initial states,
- $\rightarrow \subseteq \Sigma \times \Sigma$ is a left-total transition relation, and
- $V: Prop \rightarrow 2^{\Sigma}$ is a valuation function mapping propositions to the set of states in which they hold

- Equivalent formulation: for each state, set of propositions which hold in this state, i.e. $V': \Sigma \rightarrow 2^{Prop}$

Kripke Structure: Example

- Example: Cooker
- Propositions:
  - Cooker is starting: S
  - Door is closed: C
  - Cooker is hot: H
  - Error occurred: E



- Kripke structure:
  - $\Sigma = \{s_1, \ldots, s_6\}$
  - $I = \{s_1\}$
  - $\rightarrow = \{(s_1, s_2), (s_2, s_5), (s_5, s_2), (s_1, s_3)$ $(s_3, s_1), (s_3, s_6), (s_6, s_4), (s_4, s_4),$ $(s_4, s_3), (s_4, s_1)\}$
  - $V(S) = \{s_2, s_5, s_6\}$,
    $V(C) = \{s_3, s_4, s_5, s_6\}$,
    $V(H) = \{s_4\}$, $V(E) = \{s_2, s_5\}$

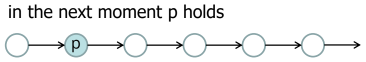
Semantics of Kripke Structures (Prop)

- We now want to define a logic in which we can formalize temporal statements, i.e. statements about the behaviour of the system and its changes over time.
- The basis is **open propositional logic** (PL): negation, conjunction, disjunction, implication*.
- With that, we define how a PL-formula ϕ holds in a Kripke structure K at state s, written as $K, s \models \phi$.
- Let $K = \langle \Sigma, I, \rightarrow, V \rangle$ be a Kripke structure, $s \in \Sigma$, and ϕ a formula of propositional logic, then
  - $K, s \models p$ — if $p \in Prop$ and $s \in V(p)$
  - $K, s \models \neg\phi$ — if not $K, s \models \phi$
  - $K, s \models \phi_1 \wedge \phi_2$ — if $K, s \models \phi_1$ and $K, s \models \phi_2$
  - $K, s \models \phi_1 \vee \phi_2$ — if $K, s \models \phi_1$ or $K, s \models \phi_2$

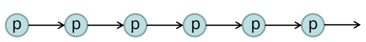
* Note implication is derived: $\phi_1 \rightarrow \phi_2 = \neg\phi_1 \vee \phi_2$

Linear Temporal Logic

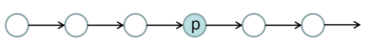
- The formulae of LTL are given as
  $\phi \ ::= \ p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$ — Propositional formulae
  $\mid X\,\phi \mid G\,\phi \mid F\,\phi \mid \phi_1\, U\, \phi_2$ — Temporal operators
- X p: in the next moment p holds



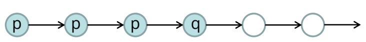
- G p: p holds in all moments



- F p: there is a moment in the future when p will hold

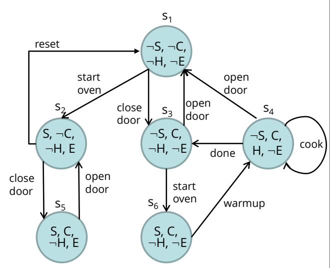


- p U q: p holds in all moments until q holds



Examples of LTL formulae

- If the cooker heats, then is the door closed?
  $G(H \rightarrow C)$ ✓
- Is it always possible to recover from an error?
  $G\,(E \rightarrow F\,\neg E)$ ✗
  - Need to add a transition.
- Is it always possible to cook (heat up, then cook)?
  $F\,(S \rightarrow X\,C)$ ✗
  - Always possible to "avoid" cooking.
  - Cannot express "there are paths in which we can always cook".



Paths in an FSM/Kripke Structure

- A **path** in an FSM (or Kripke structure) is a sequence of states starting in one of the initial states and connected by the transition relation (essentially, a **run** of the system).
- Formally: for an FSM $M = \langle \Sigma, I, \rightarrow \rangle$ or a Kripke structure $K = \langle \Sigma, I, \rightarrow, V \rangle$, a **path** is given by a sequence $s_1 s_2 s_3 \ldots \in \Sigma^*$ such that $s_1 \in I$ and $s_i \rightarrow s_{i+1}$.
- For a path $p = s_1 s_2 s_3 \ldots$, we write
  - p_i for **selecting** the i-th element s_i and
  - p^i for the **suffix** starting at position i, $s_i s_{i+1} s_{i+2} \ldots$

Semantics of LTL in Kripke Structures

Let $K = \langle \Sigma, I, \rightarrow, V \rangle$ be a Kripke Structure and ϕ an LTL formula, then we say $K \vDash \phi$ (**ϕ holds in K**), if $K, s \vDash \phi$ for all paths $s = s_1 s_2 s_3 \ldots$ in K, where:

- $K, s \vDash p$ if $p \in Prop$, $s_1 \in V(p)$
- $K, s \vDash \neg\phi$ if not $K, s \vDash \phi$
- $K, s \vDash \phi_1 \wedge \phi_2$ if $K, s \vDash \phi_1$ and $K, s \vDash \phi_2$
- $K, s \vDash \phi_1 \vee \phi_2$ if $K, s \vDash \phi_1$ or $K, s \vDash \phi_2$

- $K, s \vDash X\,\phi$ if $K, s^2 \vDash \phi$
- $K, s \vDash G\,\phi$ if $K, s^n \vDash \phi$ for all $n > 0$
- $K, s \vDash F\,\phi$ if $K, s^n \vDash \phi$ for some $n > 0$
- $K, s \vDash \phi\,U\,\psi$ if $K, s^n \vDash \psi$ for some $n > 0$, and for all $i, 0 < i < n$, we have $K, s^i \vDash \phi$

More examples for the cooker

- Question: does the cooker work?
- Specifically, cooking means that first the door is open, then the oven heats up, cooks, then the door is open again, and all without an error.
  - $c = \neg C \wedge X(S \wedge X(H \wedge F \neg C)) \wedge G\,\neg E$ – not quite.
  - $c = (\neg C \wedge \neg E) \wedge X(S \wedge \neg E \wedge X(H \wedge \neg E \wedge F(\neg C \wedge \neg E)))$ – better
- So, does the cooker work?
  - There is at least one path s.t. c holds eventually.
  - This is **not** $G\,F\,c$, which says that all paths must eventually cook (which might be too strong).
  - We cannot express this in LTL; this is a principal limitation.

Computational Tree Logic (CTL)

- LTL does not allow us the quantify over paths, e.g. assert the existence of a path satisfying a particular property.
- To a limited degree, we can solve this problem by negation: instead of asserting a property ϕ, we check whether $\neg\phi$ is satisfied; if that is not the case, ϕ holds. But this does not work for mixtures of universal and existential quantifiers.
- **Computational Tree Logic (CTL)** is another temporal logic which allows this by adding universal and existential quantifiers to the modal operators.
- The name comes from considering paths in **the computational tree** obtained by unwinding the transition relation of the Kripke structure.

Computational Tree Logic (CTL)

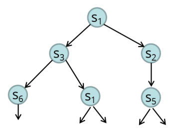
- The formulae of **CTL** are given as

$$\phi \ ::= p \mid \neg\,\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \qquad \text{Propositional formulae}$$
$$\mid AX\,\phi \mid EX\,\phi \mid AG\,\phi \mid EG\,\phi \qquad \text{Temporal operators}$$
$$\mid AF\,\phi \mid EF\,\phi \mid \phi_1\,AU\,\phi_2 \mid \phi_1 EU\,\phi_2$$

- Note that CTL formulae can be considered to be a LTL formulae with a **modality** (*A* or *E*) added to each temporal operator.
  - Generally speaking, the *A* modality says the temporal operator holds for all paths, and the *E* modality says it only holds for all least one path.
- Hence, we do not define a **satisfaction** for a single path *p*, but with respect to a specific state in an FSM.

Computational Tree Logic (CTL)

- Specifying possible paths by combination
  - Branching behavior
    All paths: A, exists path: E
  - Succession of states in a path
    Temporal operators X, G, F, U



- For example:
  - AX p : in all paths the next state satisfies p
  - EX p : there is an path in which the next state satisfies p
  - p AU q : in all paths p holds as long as q does not hold
  - EF p : there is an path in which eventually p holds

Semantics of CTL in Kripke Structures

For a Kripke structure $K = \langle \Sigma, I, \rightarrow, V \rangle$ and a CTL-formula ϕ, we say $K \vDash \phi$ (**ϕ holds in K**) if $K, s \vDash \phi$ for all $s \in I$, where $K, s \vDash \phi$ is defined inductively as follows (omitting the clauses for propositional operators $p, \neg, \wedge, \vee$):

- $K, s \vDash AX\,\phi$ iff for all s' with $s \rightarrow s'$, we have $K, s' \vDash \phi$
- $K, s \vDash EX\,\phi$ iff for some s' with $s \rightarrow s'$, we have $K, s' \vDash \phi$
- $K, s \vDash AG\,\phi$ iff for all paths p with $p_1 = s$, we have $K, p_i \vDash \phi$ for all $i \geq 2$.
- $K, s \vDash EG\,\phi$ iff for some path p with $p_1 = s$, we have $K, p_i \vDash \phi$ for all $i \geq 2$.
- $K, s \vDash AF\,\phi$ iff for all paths p with $p_1 = s$, we have $K, p_i \vDash \phi$ for some i
- $K, s \vDash EF\,\phi$ iff for some path p with $p_1 = s$, we have $K, p_i \vDash \phi$ for some i
- $K, s \vDash \phi\,AU\,\psi$ iff for all paths p with $p_1 = s$, there is i with $K, p_i \vDash \psi$ and for all $j < i, K, p_j \vDash \phi$
- $K, s \vDash \phi\,EU\,\psi$ iff for some path p with $p_1 = s$, there is i with $K, p_i \vDash \psi$ and for all $j < i, K, p_j \vDash \phi$

Examples of CTL propositions

- If the cooker is hot, then is the door closed

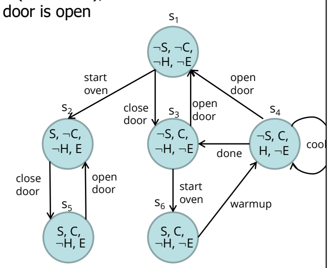
$$AG\,(H \rightarrow C)$$

- It is always possible to eventually cook (heat is on), and then eventually get the food (i.e. the door is open afterwards):

$$AF\,(H \rightarrow AF\,\neg\,C)$$

- It is always possible that the cooker will eventually warmup.

$$AG\,(EF\,(\neg H \wedge EX\,H))$$



LTL, CTL and CTL*

- CTL is more expressive than LTL, but (surprisingly) there are also properties we can express in LTL but not in CTL:
  - The formula $(F\phi) \to F\psi$ cannot be expressed in CTL
    - "When ϕ occurs somewhere, then ψ also occurs somewhere."
    - Not: $(AF\phi) \to AF\psi$, nor $AG(\phi \to AF\,\psi)$
  - The formula $AG\,(EF\phi)$ cannot be expressed in LTL
    - "For all paths, it is always the case that there is some path on which ϕ is eventually true."
- CTL* - Allow for the use of temporal operators (X, G, F, U) without a directly preceding path quantifier (A, E)
  - e.g. AGF φ is allowed
- CTL* subsumes both LTL and CTL.

Complexity and State Explosion

- Even our small oven example has 6 states with 4 labels each. If we add one integer variable with 32 bits (e.g. for the heat), we get 2^{32} additional states.
- Theoretically, there is not much hope. The basic problem of deciding whether a formula holds (**satisfiability problem**) for the temporal logics we have seen has the following complexity:
  - LTL without U is NP-complete;
  - LTL is PSPACE-complete;
  - CTL (and CTL*) are EXPTIME-complete.
- This is known as **state explosion**.
- But at least it is **decidable**. Practically, state abstraction is the key technique, so e.g. for an integer variable i we identify all states with $i \leq 0$, and those with $0 < i$.

Safety and Liveness Properties

- Safety: nothing bad ever happens
  - E.g. "x is always not equal 0"
  - Safety properties are falsified by a bad (reachable) state
  - Safety properties can falsified by a finite prefix of an execution trace
- Liveness: something good will eventually happen
  - E.g. "system is always terminating"
  - Need to keep looking for the good thing forever
  - Liveness properties can be falsified by an infinite-suffix of an execution trace: e.g. finite list of states beginning with the initial state followed by a *cycle* showing you a loop that can cause you to get stuck and never reach the "good thing"

Summary

- Model-checking allows us to show to show properties of systems by enumerating the system's states, by **modelling systems** as **finite state machines**, and expressing **properties** in **temporal logic**.
- Note difference to deductive verification (Floyd-Hoare logic): that uses the **source code** as the basis, here we need to construct a **model** of the system.
  - The model can be wrong – on the other hand we can construct the model and check properties before even building the system.
  - Model checking is **complementary** to deductive verification.
- We considered Linear Temporal Logic (**LTL**) and Computational Tree Logic (**CTL**). LTL allows us to express properties of single paths, CTL allows quantifications over all possible paths of an FSM.
- The basic problem: the system state can quickly get huge, and the basic **complexity** of the problem is horrendous, leading to so-called **state explosion.** But the use of abstraction and state compression techniques make model-checking bearable.
- Next week: tools for model checking.

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 12:

Tools for Model Checking

Christoph Lüth, Dieter Hutter, Jan Peleska

Organisatorisches

- Prüfungstermine
  - 06.03.2020, 12- 18 Uhr
  - 02.04.2020, ganztägig
- Scheinbedingungen:
  - Note aus der mündlichen Prüfung
  - Benotung der Übungsblätter: A = 1.3, B = 2.3, C = 3.3
  - Kann als Bonus (nicht Malus) mit 20% hinzugerechnet werden.

Where are we?

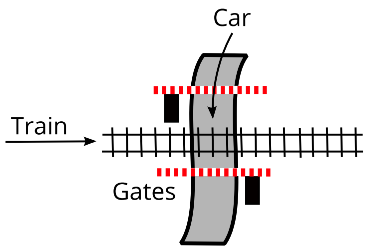
Introduction

- In the last lecture, we saw the **basics of model-checking**: how to model systems on an abstract level with **FSM** or **Kripke structures**, and how to specify their properties with **temporal logic** (LTL and CTL).
- This was motivated by the promise of "efficient tool support".
- So how does this tool support look like, and how does it work? We will hopefully answer these two questions in the following…
- Brief overview:
  - An **Example**: The Railway Crossing.
  - Modelchecking with **NuSMV** and **Spin**.
  - Algorithms for Model Checking.

The Railway Crossing

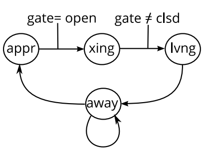
Quelle: Wikipedia

First Abstraction

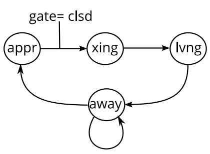


The Model

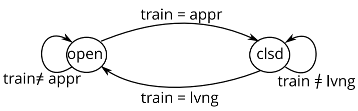
States of the car:



States of the train:



States of the gate:



The Finite State Machine

- The states of the FSM is given by mapping variables $car, train, gate$ to the domains

$$\Sigma_{car} = \{appr, xing, lvng, away\}$$
$$\Sigma_{train} = \{appr, xing, lvng, away\}$$
$$\Sigma_{gate} = \{open, clsd\}$$

- Or alternatively, states are a 3-tuples

$$s \in \Sigma = \Sigma_{car} \times \Sigma_{train} \times \Sigma_{gate}$$

- The transition relation is given by

$$\langle away, away, open\rangle \rightarrow \langle appr, away, open\rangle$$
$$\langle appr, away, open\rangle \rightarrow \langle xing, away, open\rangle$$
$$\langle appr, appr, clsd\rangle \rightarrow \langle appr, xing, clsd\rangle$$
$$\langle appr, xing, clsd\rangle \rightarrow \langle appr, lvng, clsd\rangle$$
$$\langle appr, lvng, clsd\rangle \rightarrow \langle appr, away, open\rangle$$
$$\ldots$$

Properties of the Railway Crossing

- We want to express properties such as
  - Cars and trains may never cross at the same time.
  - The car can always leave the crossing.
  - Approaching trains may eventually cross.
  - It is possible for cars to cross the tracks.
- The first two are **safety properties**, the last two are **liveness properties**.
- To formulate these in temporal logic, we first need the **basic propositions** which talk about the variables of the state.

Basic Propositions

- The basic propositions $Prop$ are given as equalities over the state variables:
  $(car = v) \in Prop$ mit $v \in \Sigma_{car}$,
  $(train = v) \in Prop$ mit $v \in \Sigma_{train}$,
  $(gate = v) \in Prop$ mit $v \in \Sigma_{gate}$
- The Kripke structure valuation V maps each basic proposition to all states where this equality holds.

The Properties

- Cars and trains never cross at the same time:
  $$G\neg(\, car = xing \wedge train = xing)$$
- A car can always leave the crossing:
  $$G\,(car = xing \rightarrow F\,(car = lvng))$$
- Approaching trains may eventually cross:
  $$G\,(train = appr \rightarrow F\,(train = xing))$$
- There are cars which are crossing the tracks:
  $$EF\,(car = xing)$$
  - Not expressible in LTL, $F\,(car = xing)$ means something stronger („there is always a car which eventually crosses")

Model-Checking Tools: NuSMV2

- NuSMV is a reimplementation of SMV, the first model-checker to use BDDs. NuSMV2 also adds SAT-based model checking.
- Systems are modelled as synchronous FSMs (Mealy automata) *or asynchronous processes**.
- Properties can be formulated in LTL and CTL.
- Written in C, open source. Latest version 2.6.0 from Oct. 2015.
- Developed by Fondazione Bruno Kessler, Carnegie Mellon University, the University of Genoa and the University of Trento.

* This is apparently depreciated now.

Model-Checking Tools: Spin

- Spin was originally developed by Gerard Holzmann at Bell Labs in the 80s.
- Systems modelled in Promela (Process Meta Language): asynchronous communication, non-deterministic automata.
- Spin translates the automata into a C program, which performs the actual model-checking.
- Supports LTL and CTL.
- Latest version 6.4.7 from August 2017.
- Spin won the ACM System Software Award in 2001.

Conclusions

- Tools such as **NuSMV2** and **Spin** make model-checking feasible for moderately sized systems.
- This allows us to find errors in systems which are hard to find by testing alone.
- The key ingredient is **efficient state abstraction**.
  - But careful: **abstraction** must **preserve properties**.

Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

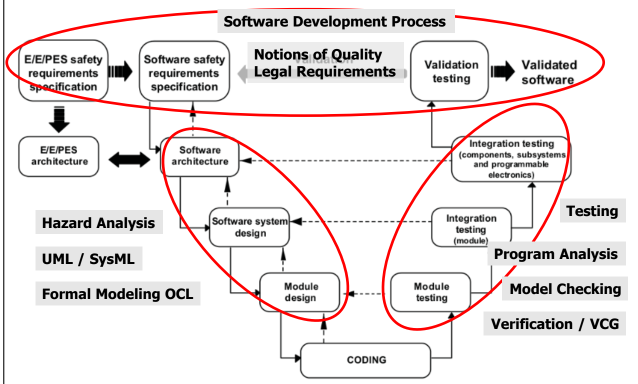
Lecture 13:

Concluding Remarks

Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

The Global Picture



Examples of Formal Methods in Practice

- Hardware verification:
  - Intel: formal verification of microprocessors (Pentium/i-Core)
  - Infineon: equivalence checks (Aurix Tricore)
- Software verification:
  - Microsoft: Windows device drivers
  - Microsoft: Hyper-V hypervisor (VCC, VeriSoft project)
  - NICTA (Aus): L4.verified (Isabelle)
- Tools used in Industry (excerpt):
  - AbsInt tools: aiT, Astree, CompCert (C)
  - SPARK tools (ADA)
  - SCADE (MatLab/Simulink)
  - UPAAL, Spin, FDR2, other model checkers

Safe and Secure Systems – Uni Bremen

- AG Betriebssysteme - Verteilte Systeme / Verified Systems (Peleska)
  - Testing, abstract interpretation
- AG Rechnerarchitektur / DFKI (Drechsler, Hutter, Lüth)
  - System verification, model checking, security
- AG Datenbanksysteme (Gogolla)
  - UML, OCL
- AG Softwaretechnik (Koschke)
  - Software engineering, reuse

Organisatorisches

- Bitte nehmt an der **Evaluation auf stud.ip** teil!
- Was war euer Eindruck vom Übungsbetrieb im Vergleich zum herkömmlichen Übungsbetrieb?
  - Man lernt mehr – weniger?
  - Es ist mehr – weniger Arbeit?
  - Kommentare in Freitextfeldern bei der stud.ip Evaluation.
- Wir bieten an folgenden Terminen mündliche Prüfungen an:
  - 05.03.2020 und 06.03.2020
  - 02.04.2020
  - Anmeldung per Mail (es liegen

Questions*

* Which might be asked in an exam, hypothetically speaking.

General Remarks

- The exam lasts 20-30 minutes, and is taken solitary.
- We are not so much interested in well-rehearsed details, but rather in principles.
- We have covered a lot of material – an exam may well not cover all of it.
  - We will rather go into detail on some lectures than spend the exam with a couple of well-rehearsed phrases from each slide.
  - Emphasis will be on the later parts of the course (SysML/OCL, testing, static analysis, Floyd-Hoare logic, model-checking) rather than the first.
  - If you do not know an answer, just say so – we can move on to a different question.

Lecture 01: Concepts of Quality

- What is quality? What are quality criteria?
- What could be useful quality criteria?
- What is the conceptual difference between ISO 9001 and the CMM (or Spice)?

Lecture 02: Legal Requirements

- What is safety?
- Norms and Standards:
  - Legal situation
  - What is the machinery directive?
  - Norm landscape: first, second, third-tier norms
  - Important norms: IEC 61508, ISO 26262, DIN EN 50128, Do-178B/C, ISO 15408,...
- Risk Analysis:
  - What is SIL, and what is for? What is a target SIL?
  - How do we obtain a SIL?
  - What does it mean for the development?

Lecture 03: SW Development Process

- Which software development models did we encounter?
- How do the following work, and what are their respective advantages/disadvantages:
  - Waterfall model, spiral model, agile development, MDD, V-model
- Which models are appropriate for safety-critical systems?
- Formal software development:
  - What is it, and how does it work?
  - What kind of properties are there, how are they defined?
  - Development structure: horizontal vs. vertical, layers and views

Lecture 04: Hazard Analysis

- What is hazard analysis for, and what are its main results?
- Where in development process is it used?
- Basic approaches:
  - bottom-up vs. top-down (what does that mean?)
- Which methods did we encounter?
  - How do they work, advantages/disadvantages?

Lecture 05: High-level design with SysML

- What is a model (in general, in UML/SysML)?
- What is UML, what is SysML, what are the differences?
- Basic elements of SysML for high-level design:
  - Structural diagrams
    - Package diagram, block definition diagram, internal block diagram
  - Behavioural Diagrams:
    - Activity diagram, state machine diagram, sequence diagram
  - How do we use this diagrams to model a particular system, e.g. a coffee machine?

Lecture 06: Formal Modeling with OCL

- What is OCL? What is used for, and why?
- Characteristics of OCL (pure, not executable, typed)
- What can it be used for?
- OCL types:
  - Basic types
  - Collection types
  - Model types
- OCL logic: four-valued Kleene logic

Lecture 07: Testing

- What is testing, what are the aims? What can testing achieve, what not?
- What are test levels (and which do we know)?
- What are test methods?
- What is a black-box test? How are the test cases chosen?
- What is a white-box test?
- What is the control-flow graph of a program?
- What kind of coverages are there, and how are they defined?

Lecture 08: Static Program Analysis

- What is that? What is the difference to testing?
- What is the basic problem, and how is it handled?
- What does we mean when an analysis is sound/complete? What is over/under approximation?
- What analysis did we consider? How did they work?
  - What are the gen/kill sets?
  - What is forward/backward analysis?

Lecture 09: Floyd-Hoare-Logic

- What is the basic idea, and what are the basic ingredients?
- Why do we need assertions, and logical variables?
- What do the following notations mean:
  - $\models \{P\}\ c\ \{Q\}$
  - $\models [P] c\ [Q]$
  - $\vdash \{P\}\ c\ \{Q\}$
- How does Floyd-Hoare logic work?
- What rules does it have?
- How is Tony Hoare's last name pronounced?

Lecture 10: Verification Condition Generation

- What do completeness and soundness of the Floyd-Hoare logic mean?
- Which of these properties does it have?
- What is the weakest precondition, and how do we calculate it?
- What are program annotations, why do we need them, and how are they used?
- What are verification conditions, and how are they calculated?

Lecture 11/12: Model Checking

- What is model-checking, and how is it used?
- What is the difference to Floyd-Hoare logic?
- What is a FSM/Kripke structure (and what is the difference)?
- Which models of time did we consider?
- For LTL, CTL:
  - What are the basic operators, when does a formula hold, and what kind of properties can we formulate?
  - Which one is more powerful?
  - Are they decidable (with which complexity)?
- Which tools did we see? What are their differences/communalities?

Thank you, and good bye.