Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 01 (13-10-2015)

Introduction and Notions of Quality

Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen

Organisatorisches

Universität Bremen

Generelles

- Einführungsvorlesung zum Masterprofil S & Q
- 6 ETCS-Punkte
- Vorlesung:
  - Montag 12 c.t – 14 Uhr (MZH 1110)
- Übungen:
  - Dienstag 12 c.t. – 14 Uhr (MZH 1470)
- Webseite:
  http://www.informatik.uni-bremen.de/~cxl/lehre/ssq.ws15/

Folien, Übungsblätter, etc.

- Folien
  - ... sind auf Englisch (Notationen!)
  - ... gibt es auf der Homepage
  - ... sind (üblicherweise) nach der Vorlesung verfügbar
- Übungen
  - Übungsblätter gibt es auf dem Web
  - Ausgabe Montag abend/Dienstag morgen
    - Erstes Übungsblatt nächste Woche
  - Abgabe vor der Vorlesung
    - Persönlich hier, oder per Mail bis Montag 12:00

Literatur

- Foliensätze als **Kernmaterial**
- Ausgewählte Fachartikel als **Zusatzmaterial**
  - Auf der Webseite verfügbar.
- Es gibt (noch) keine Bücher, die den Vorlesungsinhalt komplett erfassen.
- Zum weiteren Stöbern:
  - Wird im Verlauf der Vorlesung bekannt gegeben

Prüfungen

- **Fachgespräch** oder **Modulprüfung**
  - Anmeldefristen beachten!
- Individuelle Termine nach Absprache Februar / März
- Notenspiegel Übungsblätter:

Prozent	Note	Prozent	Note	Prozent	Note	Prozent	Note
		89.5-85	1.7	74.5-70	2.7	59.5-55	3.7
100-95	1.0	84.5-80	2.0	69.5-64	3.0	54.5-50	4.0
94.5-90	1.3	79.5-75	2.3	64.5-60	3.3	49.5-0	N/b

- Modulprüfung:
  - Keine Abgabe der Übungsblätter nötig
  - Bearbeitung dringend angeraten

Overview

Universität Bremen

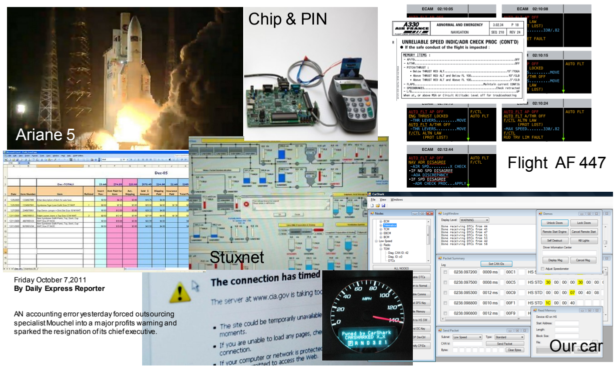
Objectives

- This is an introductory lecture for the topics

  Quality – Safety – Security

- The aim is **not** an introduction into a particular formal method, or even formal methods in general. Rather, we want to give a bird's eye view of everything relevant in connection with developing 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 elaborated jointly by Dieter Hutter, Christoph Lüth, and Jan Peleska.
- The choice of material in each semester reflects personal preferences.

Why bother with Quality and Safety?



Friday October 7,2011
By Daily Express Reporter

AN accounting error yesterday forced outsourcing specialist Mouchel into a major profits warning and sparked the resignation of its chief executive.

Ariane 5

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

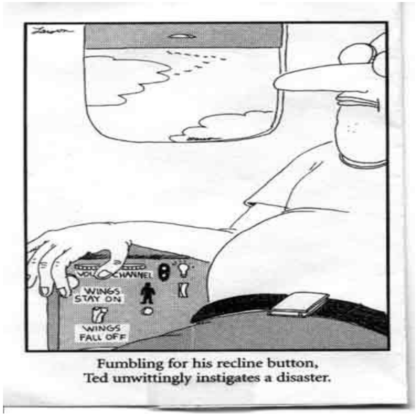
- How could that happen?

What Went Wrong With Ariane Flight 501?

(1) Self-destruction due to instability;
(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;
(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.

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 (eg. planes, cars, railways…)
- Security:
  - Product is protected against potential attacks from people, environment etc.
  - Threats from “outside”
    - Analyze and counteract the abilities of an attacker

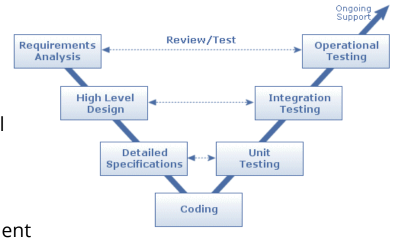


Fumbling for his recline button, Ted unwittingly instigates a disaster.

A safety-critical design flaw – invented by Gary Larson

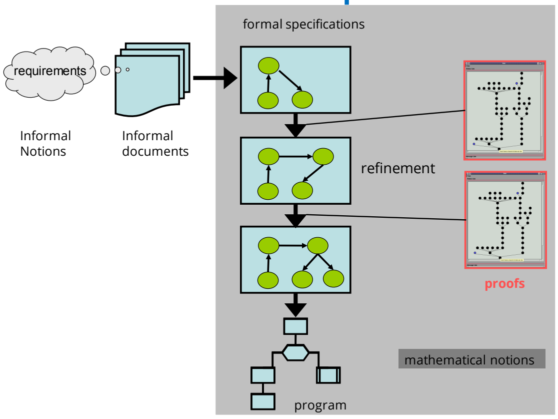
Software Development Models

- Definition of software development process and documents



- Examples:
  - Waterfall Model
  - V-Model
  - Model-Driven Architectures
  - Agile Development

Formal Software Development



Verification and Validation

- **Verification**: have we built the system right?
  - i.e. correct with respect to a reference artefact
    - specification document
    - reference system
    - Model
- **Validation**: have we built the right system
  - i.e. adequate for its intended operation?

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

Overview of Lecture Series

- 01: Concepts of Quality
- 02: Concepts of Safety, Legal Requirements, Certification
- 03: A Safety-critical Software Development Process
- 04: Requirements Analysis
- 05: High-Level Design & Detailed Specification with SysML
- 06: Testing
- 07 and 08: Program Analysis
- 09: Model-Checking
- 10 and 11: Software Verification (Hoare-Calculus)
- 12: Concurrency
- 13: Conclusions

Concepts of Quality

Universität Bremen

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.

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

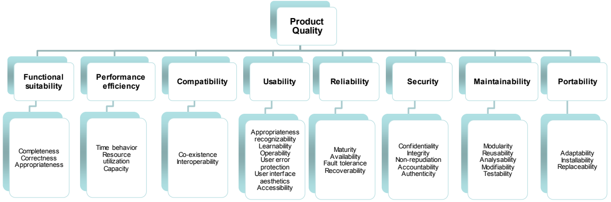
Quality Criteria (cont.)

- 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
- Quality of data model
  - Categorizes data quality attributes

Product Quality Model

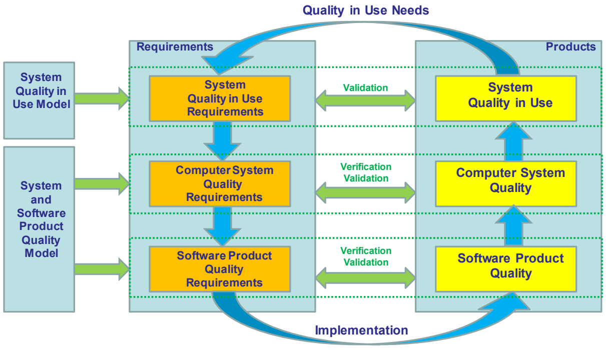


Product Quality

- Functional suitability
  - Completeness
  - Correctness
  - Appropriateness
- Performance efficiency
  - Time behavior
  - Resource utilization
  - Capacity
- Compatibility
  - Co-existence
  - Interoperability
- Usability
  - Appropriateness recognizability
  - Learnability
  - Operability
  - User error protection
  - User interface aesthetics
  - Accessibility
- Reliability
  - Maturity
  - Availability
  - Fault tolerance
  - Recoverability
- Security
  - Confidentiality
  - Integrity
  - Non-repudiation
  - Accountability
  - Authenticity
- Maintainability
  - Modularity
  - Reusability
  - Analysability
  - Modifiability
  - Testability
- Portability
  - Adaptability
  - Installability
  - Replaceability

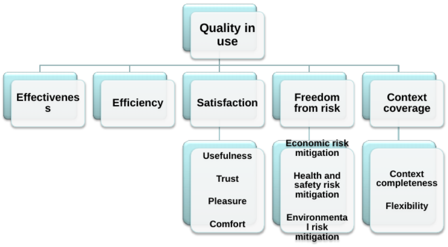
Source: ISO/IEC FDIS 25010

System Quality Life Cycle Model

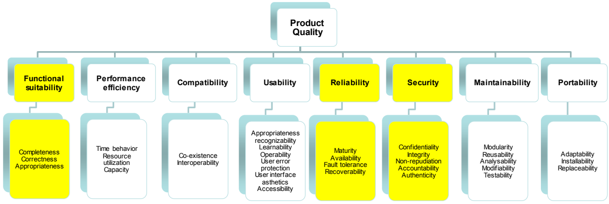


Source: ISO/IEC FDIS 25010

Quality in Use Model



Our Focus of Interest


Product Quality
Functional suitability
Performance efficiency
Compatibility
Usability
Reliability
Security
Maintainability
Portability
Completeness Correctness Appropriateness
Time behavior Resource utilization Capacity
Co-existence Interoperability
Appropriateness recognizability Learnability Operability User error protection User interface asthetics Accessibility
Maturity Availability Fault tolerance Recoverability
Confidentiality Integrity Non-repudiation Accountability Authenticity
Modularity Reusability Analysability Modifiability Testability
Adaptability Installability Replaceability

How can we „guarantee“ safety and security ?

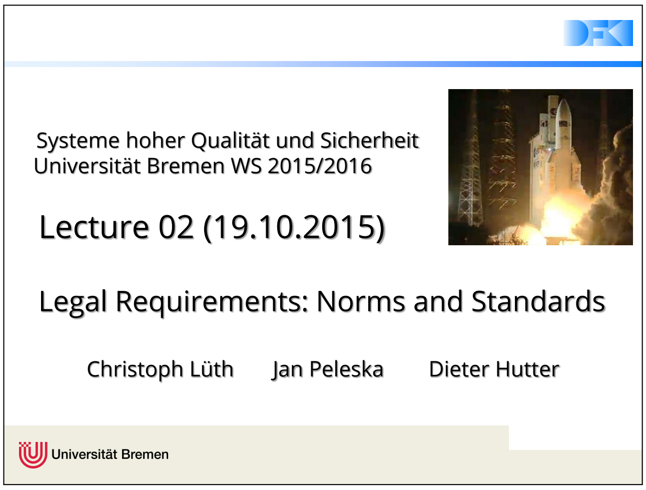
Source: ISO/IEC FDIS 25010

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
  - Standardises maturity of development process
  - Level 1 (initial): Ad-hoc
  - Level 2 (repeatable): process dependent on individuals
  - Level 3 (defined): process defined & institutionalised
  - Level 4 (managed): measured process
  - Level 5 (optimizing): improvement fed back into process

Today‘s 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



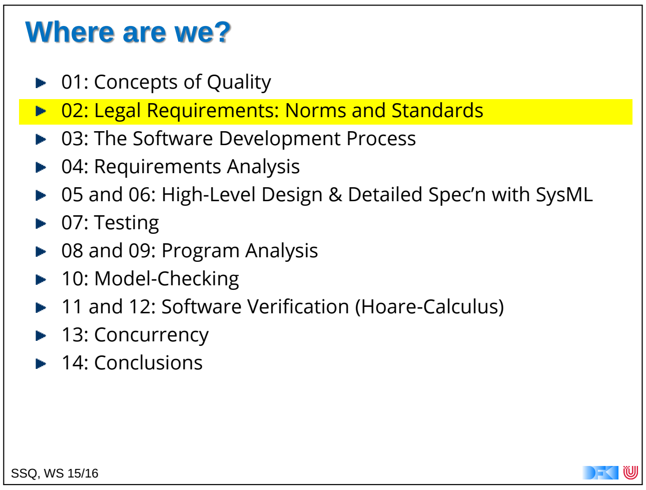
Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 02 (19.10.2015)

Legal Requirements: Norms and Standards

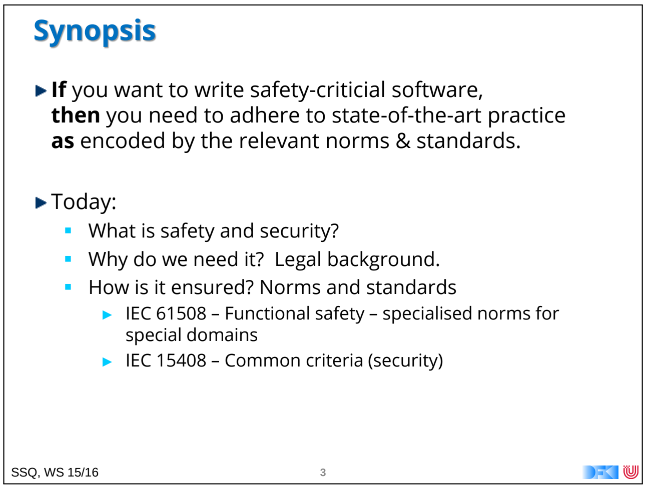
Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen



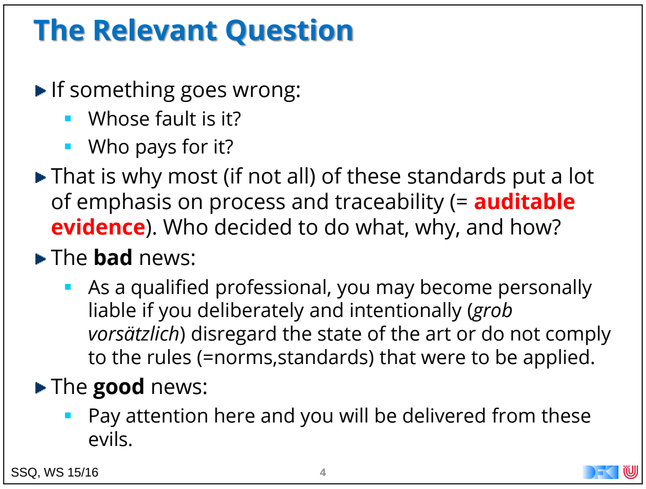
Where are we?

- 01: Concepts of Quality
- 02: Legal Requirements: Norms and Standards
- 03: The Software Development Process
- 04: Requirements Analysis
- 05 and 06: High-Level Design & Detailed Spec'n with SysML
- 07: Testing
- 08 and 09: Program Analysis
- 10: Model-Checking
- 11 and 12: Software Verification (Hoare-Calculus)
- 13: Concurrency
- 14: Conclusions



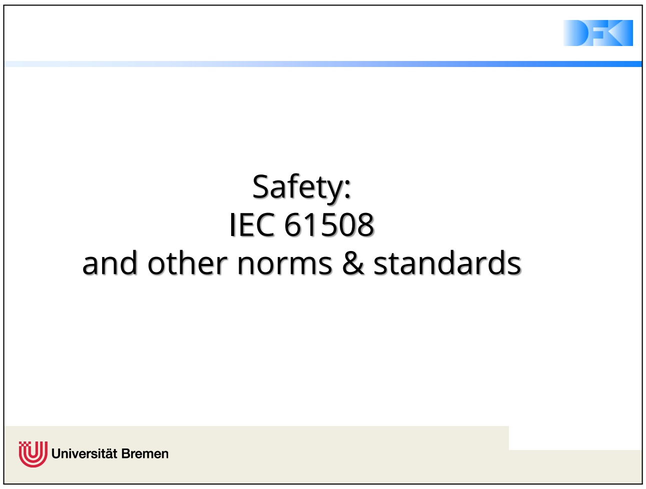
Synopsis

- **If** you want to write safety-criticial software, **then** you need to adhere to state-of-the-art practice **as** encoded by the relevant norms & standards.
- Today:
  - What is safety and security?
  - Why do we need it? Legal background.
  - How is it ensured? Norms and standards
    - IEC 61508 – Functional safety – specialised norms for special domains
    - IEC 15408 – Common criteria (security)



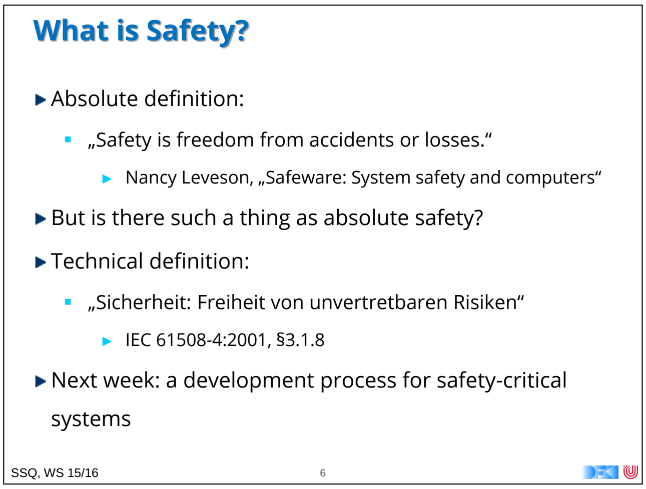
The Relevant Question

- If something goes wrong:
  - Whose fault is it?
  - Who pays for it?
- That is why most (if not all) of these standards put a lot of emphasis on process and traceability (= **auditable evidence**). Who decided to do what, why, and how?
- 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.



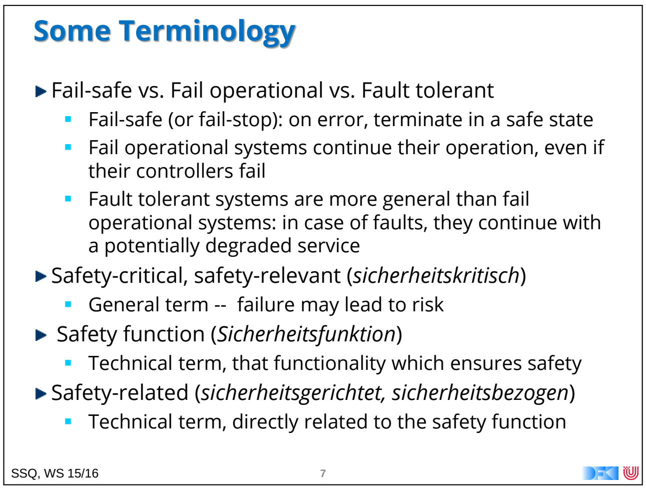
Safety: IEC 61508 and other norms & standards

Universität Bremen



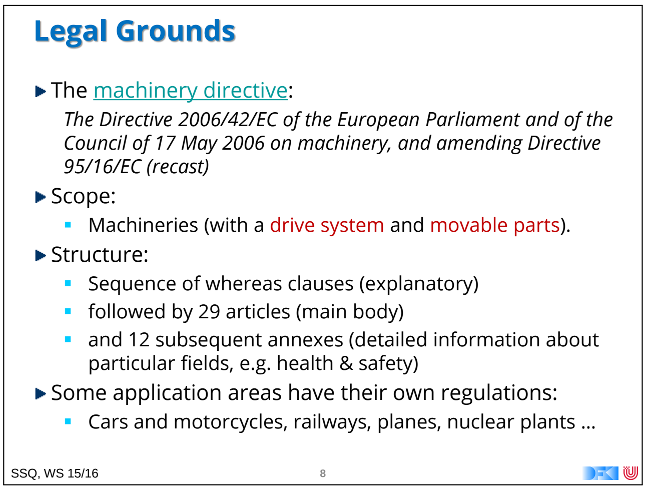
What is Safety?

- 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
- Next week: a development process for safety-critical systems



Some Terminology

- Fail-safe vs. Fail operational vs. Fault tolerant
  - Fail-safe (or fail-stop): on error, terminate in a safe state
  - Fail operational systems continue their operation, even if their controllers fail
  - Fault tolerant systems are more general than fail operational systems: in case of faults, they continue with a potentially degraded service
- 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



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).
- 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 …

What does that mean?

- Relevant for **all** machinery (from tin-opener to AGV [= automated guided vehicle])
- **Annex IV** lists machinery where safety is a concern
- Standards encode current best practice.
  - Harmonised standard available?
- External certification or self-certification
  - Certification ensures and documents conformity to standard.
- **Result:** CE Conformité Européenne
- Sope of the directive is market harmonisation, not safety – that is more or less a byproduct.

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.

The standards quagmire ?

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)

Introducing IEC 61508

- Part 1: Functional safety management, competence, **establishing SIL targets**
- Part 2: Organising and managing the life cycle
- Part 3: **Software requirements**
- Part 4: Definitions and abbreviations
- Part 5: Examples of methods for the determination of safety-integrity levels
- Part 6: Guidelines for the application
- Part 7: Overview of techniques and measures

How does this work?

1. Risk analysis determines the safety integrity level (SIL)
2. A hazard analysis leads to safety requirement specification.
3. Safety requirements must be satisfied
   - Need to verify 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.

Safety Integrity Levels

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

- P: Probabilty of dangerous failure (per hour/year)
- Examples:
  - High demand: car brakes
  - Low demand: airbag control
- Which SIL to choose? → Risk analysis
- Note: SIL only meaningful for specific safety functions.

Establishing target SIL I

- 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 („Neglible")	10^{-6}

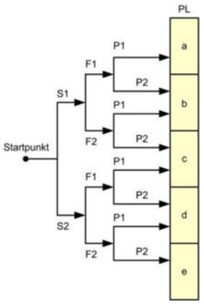
- Example:
  - Safety system for a chemical plant
  - Max. tolerable risk exposure $A=10^{-6}$
  - $B= 10^{-2}$ hazardous events lead to fatality
  - Unprotected process fails $C= 1/5$ years
  - Then Failure on Demand $E = A/(B*C) = 5*10^{-3}$, so SIL 2

Establishing Target SIL II

- 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

Occurence:
F1 – rare occurence
F2 – frequent occurence

Possible avoidance:
P1 – possible
P2 – impossible

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 organisation

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 Software Development Process

- 61508 mandates a V-model software development process
  - More next lecture
- 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 `proven in use´

Proven in Use

- 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).
  - Note that the previous use needs to be to the same specification as intended use (eg. compiler: same target platform).

SIL	Zero Failure		One Failure	
1	12 ops	12 yrs	24 ops	24 yrs
2	120 ops	120 yrs	240 ops	240 yrs
3	1200 ops	1200 yrs	2400 ops	2400 yrs
4	12000 ops	12000 yrs	24000 ops	24000 yrs

Table A.2, Software Architecture

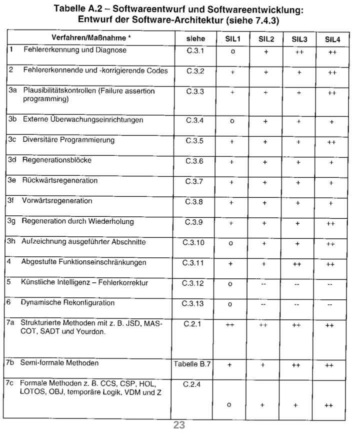


Tabelle A.2 – Softwareentwurf und Softwareentwicklung: Entwurf der Software-Architektur (siehe 7.4.3)

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

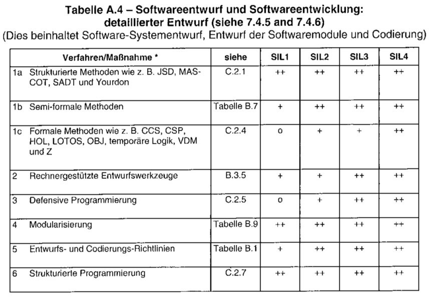


Tabelle A.4 – Softwareentwurf und Softwareentwicklung: detaillierter Entwurf (siehe 7.4.5 and 7.4.6)
(Dies beinhaltet Software-Systementwurf, Entwurf der Softwaremodule und Codierung)

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

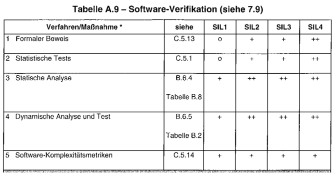


Tabelle A.9 – Software-Verifikation (siehe 7.9)

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.

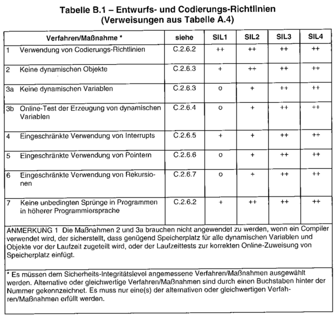


Tabelle B.1 – Entwurfs- und Codierungs-Richtlinien (Verweisungen aus Tabelle A.4)

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

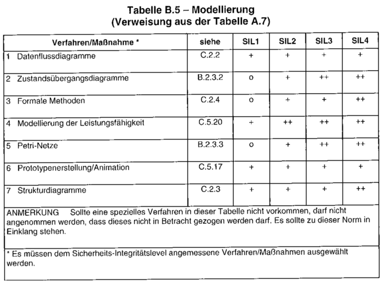


Tabelle B.5 – Modellierung
(Verweisung aus der Tabelle A.7)

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

- Certiciation is the process of showing **conformance** to a **standard**.
- Conformance to IEC 61508 can be shown in two ways:
  - Either that an organisation (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 **accredited** body.
  - In Germany, e.g. the TÜVs or the Berufsgenossenschaften (BGs)
- Also sometimes (eg. DO-178B) called `qualification'.

Security:
The Common Criteria

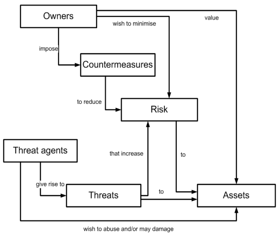
Universität Bremen

Common Criteria (IEC 15408)

- This multipart standard, the Common Criteria (CC), is meant to be used as the basis for evaluation of **security properties** of IT products and systems. By establishing such a common criteria base, the results of an IT security evaluation will be meaningful to a wider audience.
- 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.
- During evaluation, such an IT product or system is known as a **Target of Evaluation** (TOE) .
  - Such TOEs include, for example, operating systems, computer networks, distributed systems, and applications.

General Model

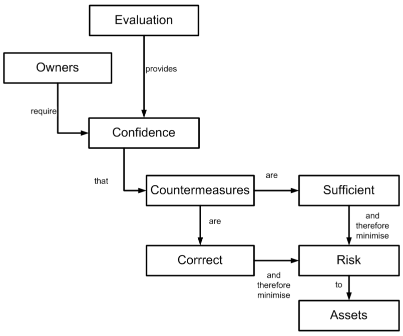
- Security is concerned with the protection of assets. Assets are entities that someone places value upon.
- Threats give rise to risks to the assets, based on the likelihood of a threat being realized and its impact on the assets
- (IT and non-IT) Counter-measures are imposed to reduce the risks to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



Common Criteria (CC)

- The CC addresses protection of information from unauthorized disclosure, modification, or loss of use. The categories of protection relating to these three types of failure of security are commonly called **confidentiality**, **integrity**, and **availability**, respectively.
- The CC may also be applicable to aspects of IT security outside of these three.
- The CC concentrates 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



Requirements Analysis

- The **security environment** includes all the laws, organizational security policies, customs, expertise and knowledge that are determined to be relevant.
  - It thus defines the context in which the TOE is intended to be used.
  - The security environment also includes the 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.
  - For an IT system, such policies may be explicitly referenced, whereas for a general purpose IT product or product class, working assumptions about organizational security policy may need to be made.

Requirements Analysis

- A statement of **assumptions** which are to be met by the environment of the TOE in order for the TOE to be considered secure.
  - This statement can be accepted as axiomatic for the TOE evaluation.
- A statement of **threats** to security of the assets would identify all the threats perceived by the security analysis as relevant to the TOE.
  - The CC characterizes a threat 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.
- An assessment of **risks** to security would qualify each threat with an assessment of the likelihood of such a threat developing into an actual attack, the likelihood of such an attack proving successful, and the consequences of any damage that may result.

Requirements Analysis

- The intent of determining **security objectives** is to address all of the security concerns and to declare which security aspects are either addressed directly by the TOE or by its environment.
  - This categorization is based on a process incorporating engineering judgment, security policy, economic factors and risk acceptance decisions.
  - Corresponds to (part of) requirements definition !
- The results of the analysis of the security environment could then be used to state the security objectives that counter the identified threats and address identified organizational security policies and assumptions.
- The security objectives should be consistent with the stated operational aim or product purpose of the TOE, and any knowledge about its physical environment.

Requirements Analysis

- 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.

Requirements Analysis

- The IT **security requirements** are the refinement of the security objectives into a set of security requirements for the TOE and security requirements for the environment which, if met, will ensure that the TOE can meet its security objectives.
- The CC presents security requirements under the distinct categories of functional requirements and assurance requirements.
- Functional requirements
  - Security behavior 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, ...

Functional Requirement

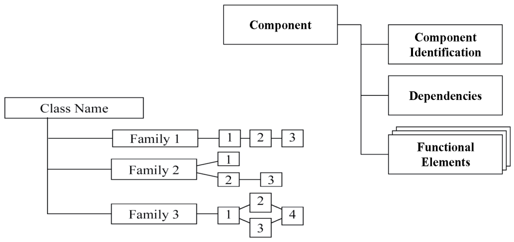
- The **functional requirements** are levied on those functions of the TOE that are specifically in support of IT security, and define the desired security behavior.
- Part 2 defines the CC functional requirements. Examples of functional requirements include requirements for identification, authentication, security audit and non-repudiation of origin.

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

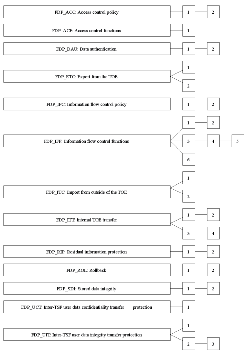
Security Functional Components

- Content and presentation of the functional requirements

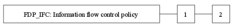


Decomposition of FDP

- FDP : User Data Protection


FDP_ACC: Access control policy
1
2
FDP_ACF: Access control functions
1
FDP_DAU: Data authentication
1
2
FDP_ETC: Export from the TOE
1
2
FDP_IFC: Information flow control policy
1
2
FDP_IFF: Information flow control functions
1
2
3
4
5
6
FDP_ITC: Import from outside of the TOE
1
2
FDP_ITT: Internal TOE transfer
1
2
3
4
FDP_RIP: Residual information protection
1
2
FDP_ROL: Rollback
1
2
FDP_SDI: Stored data integrity
1
2
FDP_UCT: Inter-TSF user data confidentiality transfer protection
1
FDP_UIT: Inter-TSF user data integrity transfer protection
1
2
3

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

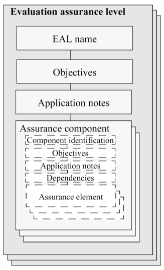
"The CC philosophy is to provide assurance based upon an evaluation (active investigation) of the IT product that is to be trusted. Evaluation has been the traditional means of providing assurance and is the basis for prior evaluation criteria documents. "

CC, Part 3, p.15

Assurance Requirements

- The **assurance requirements** are levied on actions of the developer, on evidence produced and on the actions of the evaluator.
- Examples of assurance requirements include constraints on the rigor of the **development process** and requirements to search for and analyze the impact of potential security vulnerabilities.
- The **degree of assurance** can be varied for a given set of functional requirements; therefore it is typically expressed in terms of increasing levels of rigor built with assurance components.
- Part 3 defines the CC assurance requirements and a scale of **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

Assurance Components: Example

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)

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. semiformally designed and tested
6. semiformally verified design and tested
7. formally verified design and tested

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

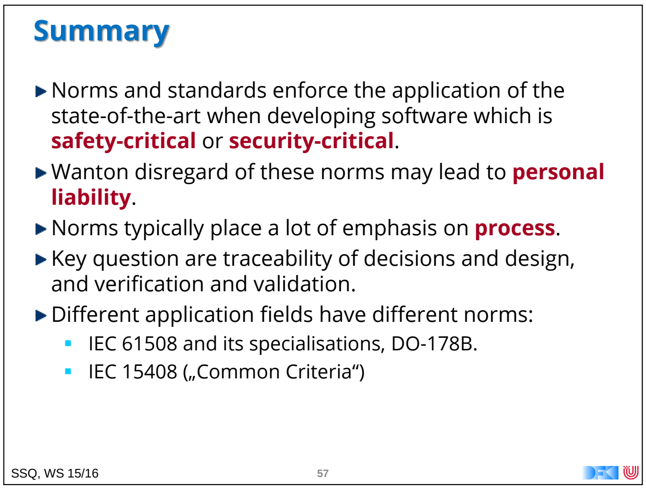
Assurance Requirements

- EAL5 – EAL7 require **formal methods**.
- according to CC Glossary:

  Formal: Expressed in a restricted syntax language with defined semantics based on well-established mathematical concepts.

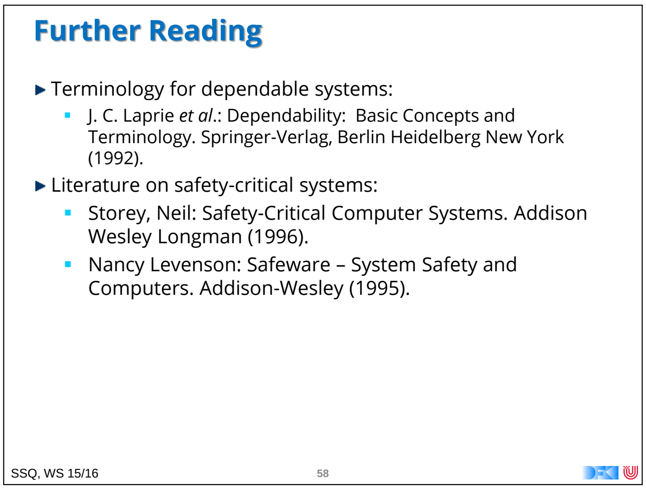
Security Functions

- 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**.



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 specialisations, 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).

Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 03 (26.10.2015)

The Software Development Process

Christoph Lüth Jan Peleska Dieter Hutter

Your Daily Menu

- Models of software development
  - The software development process, and its rôle in safety-critical software development.
  - What kind of development models are there?
  - Which ones are useful for safety-critical software – and why?
  - What do the norms and standards say?
- Basic notions of formal software development
  - What is formal software development?
  - How to specify: properties and hyperproperties
  - Structuring of the development process

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 SysML
- 07: Detailed Specification with SysML
- 08: Testing
- 09 and 10: Program Analysis
- 11: Model-Checking
- 12: Software Verification (Hoare-Calculus)
- 13: Software Verification (VCG)
- 14: Conclusions

Software Development Models

Software Development Process

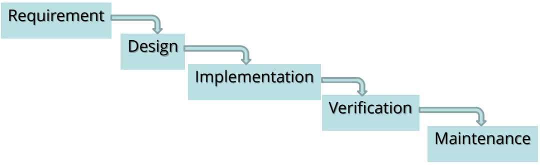
- A software development process is the **structure** imposed on the development of a software product.
- We classify processes according to *models* which specify
  - the artefacts of the development, such as
    - the software product itself, specifications, test documents, reports, reviews, proofs, 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 sofware product, or more? (specifications, test runs, documents, proofs...)

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
  - Less support for non-functional requirements
- Test-driven development
  - Tests as *executable specifications:* write tests first
  - Often used together with the other two

Waterfall Model (Royce 1970)

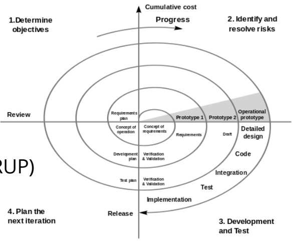
- Classical top-down sequential workflow with strictly separated phases.


Requirement
Design
Implementation
Verification
Maintenance

- Unpractical as actual workflow (no feedback between phases), but even early papers did not *really* suggest this.

Spiral Model (Böhm, 1986)

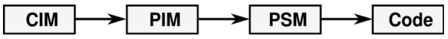
- Incremental development guided by **risk factors**
- Four phases:
  - Determine objectives
  - Analyse risks
  - Development and test
  - Review, plan next iteration
- See e.g.
  - Rational Unified Process (RUP)



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

Model-Driven Development (MDD, MDE)

- Describe problems on abstract level using *a modelling 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)

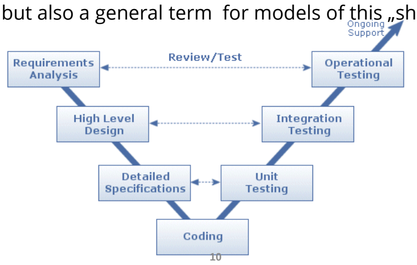


- Variety of tools:
  - Rational tool chain, Enterprise Architect, Rhapsody, Papyrus, Artisan Studio, MetaEdit+, Matlab/Simulink/Stateflow*
  - EMF (Eclipse Modelling Framework)
- Strictly sequential development
- Drawbacks: high initial investment, limited flexibility

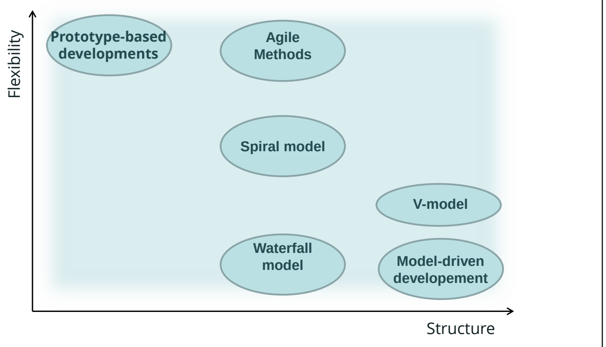
* Proprietary DSL – not related to UML

V-Model

- Evolution of the waterfall model:
  - Each phase is supported by a corresponding testing phase (verification & validation)
  - Feedback between next and previous phase
- Standard model for public projects in Germany
  - … but also a general term for models of this „shape“



Software Development Models



from S. Paulus: Sichere Software

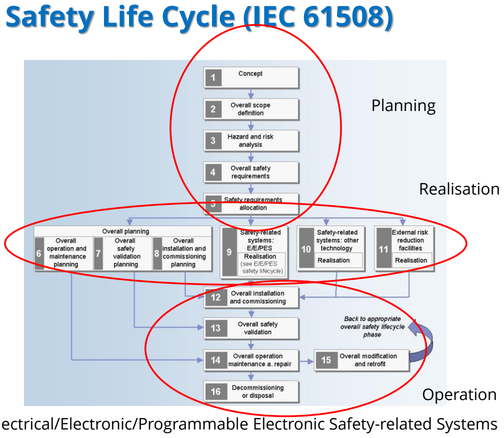
Development Models for Critical Systems

Universität Bremen

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 for initial developments of a new product
  - Agile models (e.g. SCRUM) for maintenance and product extensions

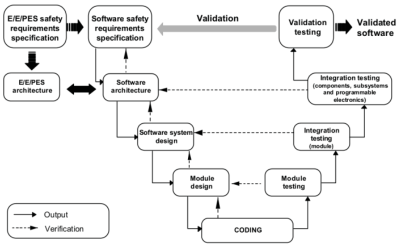
The Safety Life Cycle (IEC 61508)


1 Concept
2 Overall scope definition
3 Hazard and risk analysis
4 Overall safety requirements
5 Safety requirements allocation
Overall planning
6 Overall operation and maintenance planning
7 Overall safety validation planning
8 Overall installation and commissioning planning
9 Safety-related systems: E/E/PES Realisation (see E/E/PES safety lifecycle)
10 Safety-related systems: other technology Realisation
11 External risk reduction facilities Realisation
12 Overall installation and commissioning
13 Overall safety validation
14 Overall operation maintenance a. repair
15 Overall modification and retrofit
16 Decommissioning or disposal
Back to appropriate overall safety lifecycle phase
Planning
Realisation
Operation

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

Development Model in IEC 61508

- IEC 61508 prescribes certain activities for each phase of the life cycle.
- Development is one part of the life cycle.
- IEC 61508 *recommends* V-model.


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING
Output
Verification

Development Model in DO-178B

- DO-178B 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.

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
  - Eg. DO-178B requires all code derives from requirements

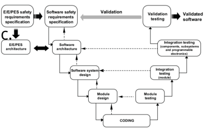
Artefacts in the Development Process

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

Specifications:
- Safety requirement spec.
- System specification
- Detail specification
- User document (safety reference manual)

Implementation:
- Code

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


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

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

Documents must be *identified* and *reconstructable*.
- Revision control and configuration management *mandatory*.

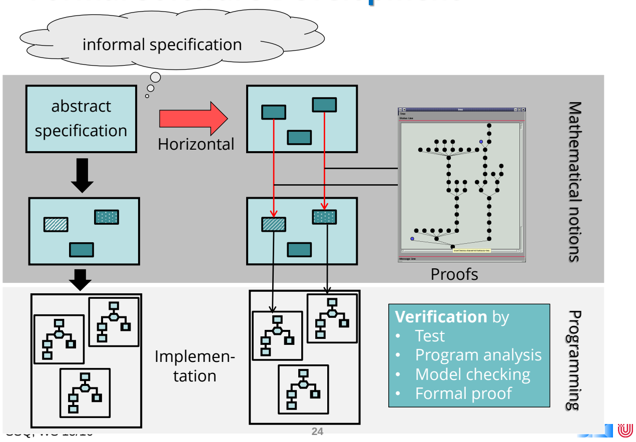
Basic Notions of Formal Software Development

Universität Bremen

Formal Software Development

- In **formal** development, properties are stated in a rigorous way with a precise mathematical semantics.
- These formal specifications can be **proven**.
- Advantages:
  - Errors can be found **early** in the development process, saving time and effort and hence costs.
  - There is a higher degree of trust in the system.
  - Hence, standards recommend use of formal methods for high SILs/EALs.
- Drawback:
  - Higher effort
  - Requires **qualified** personnel (that would be *you*).
- There are tools which can help us by
  - **finding** (simple) proofs for us, or
  - **checking** our (more complicated) proofs.

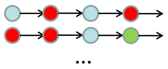
Formal Software Development



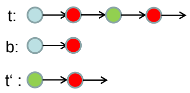
Verification by
- Test
- Program analysis
- Model checking
- Formal proof

A General Notion of Properties

- **Defn:** a **property** is a set of infinite execution traces (i.e. infinite sequences of states)



- Trace t satisfies property P, written $t \vDash P$, iff $t \in P$
- b ≤ t iff $\exists t'. t = b \cdot t'$
  - i.e. b is a *finite* prefix of t



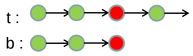
Safety and Liveness Properties

Alpen & Schneider (1985, 1987)

- **Safety** properties
  - *Nothing bad happens*
  - partial correctness, program safety, access control
- **Liveness** properties
  - *Something good happens*
  - Termination, guaranteed service, availability
- **Theorem**: ∀ P . P = $\mathrm{Safe}_P \cap \mathrm{Live}_P$
  - Each property can be represented as a combination of safety and liveness properties.

Safety Properties

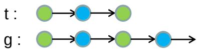
- Safety property S: „Nothing bad happens“
- A bad thing is *finitely* observable and *irremediable*
- S is a safety property iff
  - $\forall t. t \notin S \ \rightarrow (\exists b. \text{finite}\ b \wedge b \leq t \ \rightarrow \forall u. b \leq u \rightarrow u \notin S)$



  - a finite prefix b always causes the bad thing
- **Safety is typically proven by induction.**
  - Safety properties may be enforced by run-time monitors.
  - Safety is testable (i.e. we can test for non-safety)

Liveness Properties

- Liveness property L: „Good things will happen“
- A good thing is always possible and possibly infinite:
- L is a liveness property iff
  - $\forall\, t.\ \text{finite}\ t\ \rightarrow \exists g.\, t \leq g \wedge g \in L$
  - i.e. all finite traces t can be extended to a trace g in L.



- **Liveness is typically proven by well-foundedness.**

Underspecification and Nondeterminism

- A *system* S is characterised by a *set of traces*, $[\![S]\!]$
- A system S *satisfies* a property P, written

  $S \vDash P$ iff $[\![S]\!] \subseteq P$
- Why more than one trace? Difference between:
  - *Underspecification* or *loose specification* – we specify several *possible* implementations, but each implementation should be deterministic.
  - Non-determinism – different program runs might result in different traces.
- Example: a simple can vending machine.
  - Insert coin, chose brand, dispense drink.
  - Non-determinisim due to *internal* or *external* choice.

Security Policies

Many security policies are not properties!

- Examples:
  - Non-Interference (Goguen & Meseguer 1982)
    - Commands of high users have no effect on observations of low users
  - Average response time is lower than k.
- Security policies are examples of hyperproperties.
- A **hyperproperty** H is a set of properties
  - i.e. a set of set of traces.
  - System S satisfies H, $S \vDash H$, iff $[\![S]\!] \in H$.

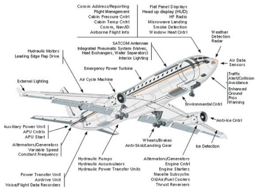
Structuring the Development

Structure in 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
  - Inheritence
- Layers / Views
  - Adresses 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:



Horizontal Structuring: Composition

- Given two systems S_1, S_2, their *sequential composition* is defined as

$$S_1 ; S_2 = \{ s\ \cdot\ t \,|\, s \in\ [\![S_1]\!], t \in\ [\![S_2]\!]\}$$

  - All traces from S_1, followed by all traces from S_2.
- Given two traces s, t, their *interleaving* is defined (recursively) as

  $<> \parallel t = t$

  $s\ \parallel <> = s$

  $a \cdot s \parallel b\ \cdot\ t = \{ a\ \cdot u \,|\, u \in s\ \parallel b\ \cdot t \} \cup \{ b\ \cdot u \,|\ u \in a \cdot s\ \parallel t\}$
- Given two systems S_1, S_2, their *parallel composition* is defined as

$$S_1 \parallel S_2 = \{ s\ \parallel t\ \,| s \in [\![S_1]\!], t \in [\![S_2]\!]\}$$

  - Traces from S_1 interleaved with traces from S_2.

Vertical Structure - Refinement

- 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

Refinement and Properties

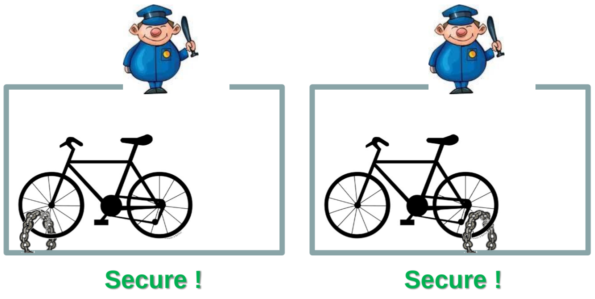
- Refinement typically preserves safety properties.
  - This means if we start with an abstract specification which we can show satisfies the desired properties, and refine it until we arrive at an implementation, we have a system for the properties hold *by construction*:

$$SP \rightsquigarrow SP_1 \rightsquigarrow SP_2 \rightsquigarrow \ldots \rightsquigarrow Imp$$

- However, **security** is typically **not** preserved by refinement nor by composition!

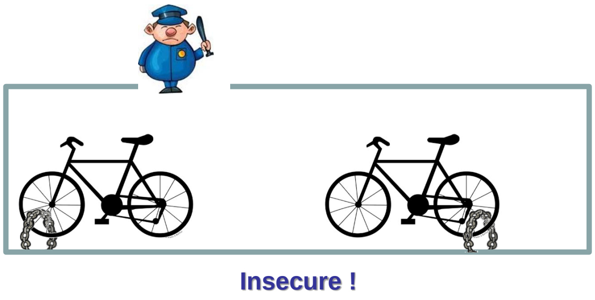
Security and Composition

Only complete bicycles are allowed to pass the gate.



Security and Composition

Only complete bicycles are allowed to pass the gate.



A Formal Treatment of Refinement

- **Def**: T is a refinement of S if $S \sqsubseteq T \Leftrightarrow [\![T]\!] \subseteq [\![S]\!]$
  - Remark: a bit too general, but will do here.
- **Theorem:** Refinement preservers properties:
  If $S \vDash P$ and $S \sqsubseteq T$, then $T \vDash P$.
  - Proof: Recall $S \vDash P \Leftrightarrow [\![S]\!] \subseteq \mathrm{P}$, and $S \sqsubseteq T \Leftrightarrow [\![T]\!] \subseteq [\![S]\!]$, hence $[\![T]\!] \subseteq P \Leftrightarrow T \vDash P$.
- However, refinement does **not** preserve hyperproperties.
  - Why? $S \vDash H \Leftrightarrow [\![S]\!] \in H$, but H **not** closed under subsets.

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.
- Properties: sets of traces
  hyperproperties: sets of properties
- Structuring of the development:
  - Horizontal – e.g. composition
  - Vertical – refinement (data, process and action ref.)
  - Refinement preserves properties (safety), but not hyperproperties (security).

Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 04 (02.11.2015)

Hazard Analysis

Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen

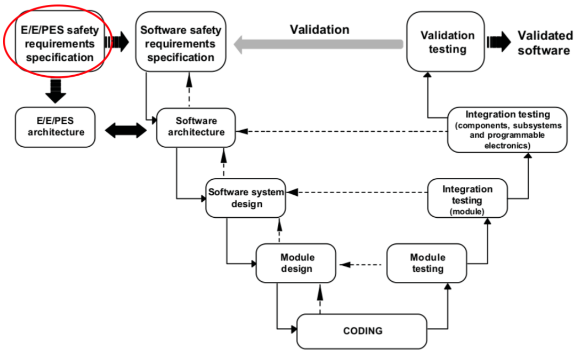
Where are we?

- 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 SysML
- 07: Detailed Specification with SysML
- 08: Testing
- 09 and 10: Program Analysis
- 11: Model-Checking
- 12: Software Verification (Hoare-Calculus)
- 13: Software Verification (VCG)
- 14: Conclusions

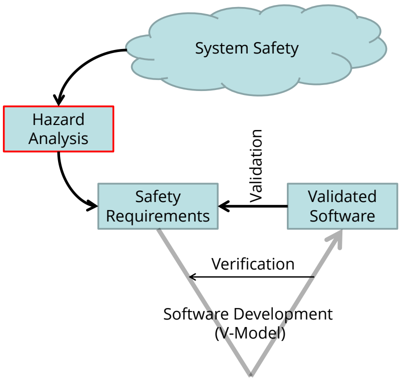
Your Daily Menu

- Hazard Analysis:
  - What's that?
- Different forms of hazard analysis:
  - Failure Mode andEffects Analysis (FMEA)
  - Failure Tree Analysis (FTA)
  - Event Tree Analysis (ETA)

Hazard Analysis in the Development Cycle


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

The Purpose of Hazard Analysis



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

The realisation 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 signifigance of hazards, and to etablish 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 standard 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

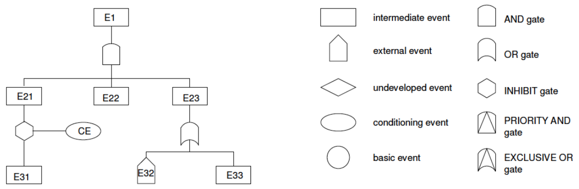
- **Top-down methods** start with an anticipated hazard and work back from the hazard event to potential causes for the hazard
  - Good for finding causes for hazard
  - Good for avoiding the investigation of "non-relevant" errors
  - Bad for detection of missing hazards
- **Bottom-up methods** consider "arbitrary" faults and resulting errors of the system, and investigate whether they may finally cause a hazard
  - Properties are complementary to top-down properties

Hazard Analysis Methods

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

Fault Tree Analysis (FTA)

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



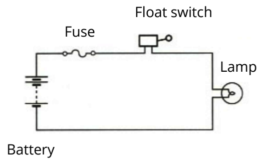
Fault-Tree Analysis: Process Overview

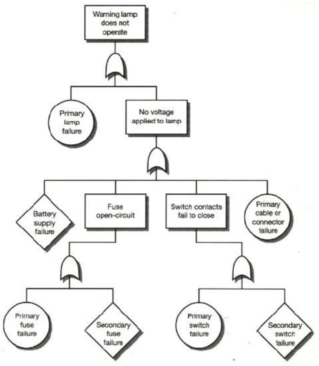
1. Understand system design
2. Define top undesired event
3. Establish boundaries (scope)
4. Construct fault tree
5. Evaluate fault tree (cut sets, probabilities)
6. Validate fault tree (check if correct and complete)
7. Modify fault tree (if required)
8. Document analysis

Fault Tree Analysis: Example 1

Example:

- A lamp warning about low level of brake fluid.
- See circuit diagram.
- Top Undesired Event: Warning lamp off despite low level of fluid.



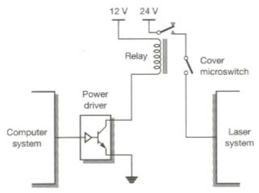


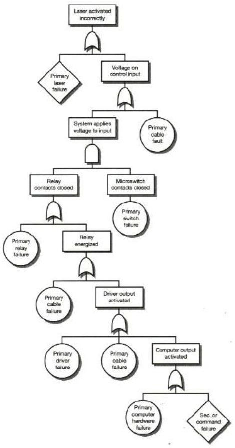
Source: N. Storey, Safety-Critical Computer Systems.

FTA: Example II

Example: A laser 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.



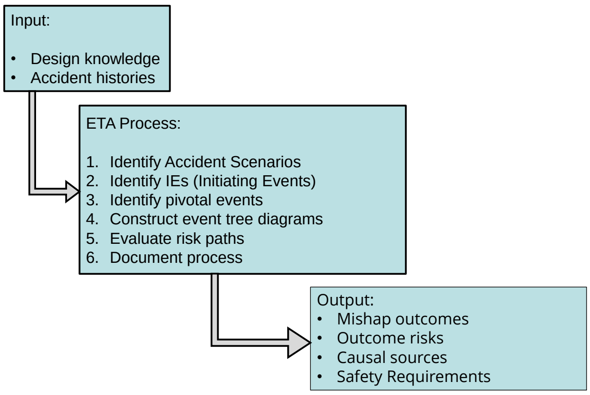


Source: N. Storey, Safety-Critical Computer Systems.

Event Tree Analysis (ETA)

- 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
- O(2^n) complexity

Event Tree Analysis Overview



Input:

- Design knowledge
- Accident histories

ETA Process:

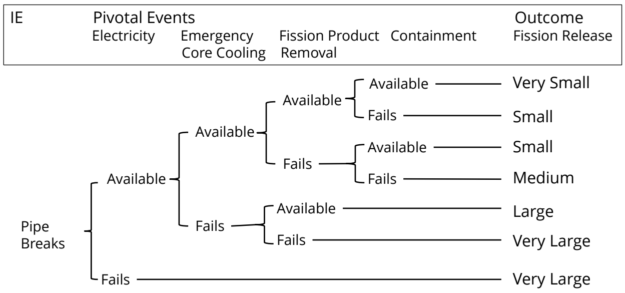
1. Identify Accident Scenarios
2. Identify IEs (Initiating Events)
3. Identify pivotal events
4. Construct event tree diagrams
5. Evaluate risk paths
6. Document process

Output:

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

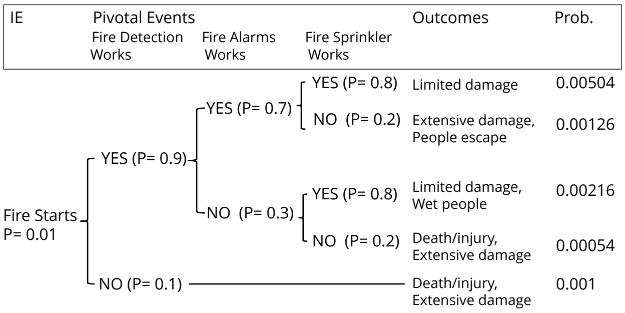
Event Tree Analysis: Example 1

- ▸ Cooling System for a Nuclear Power Plant



Event Tree Analysis: Example 2

- ▸ Fire Detection/Suppression System for Office Building



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

FMEA Example: Airbag Control (Struct.)

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

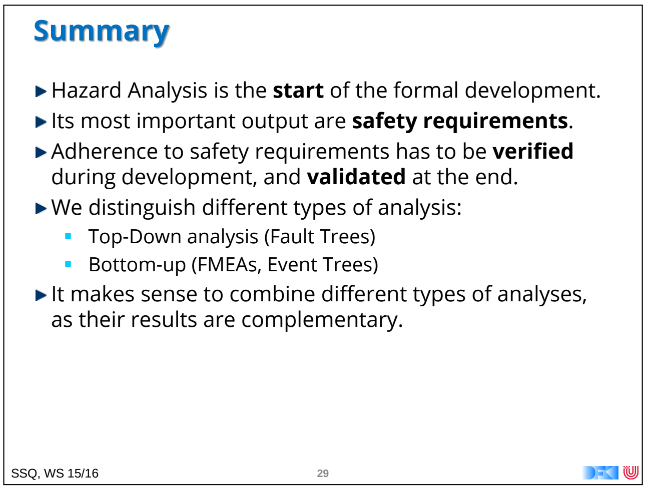
FMEA Example: Airbag Control (Funct.)

ID	Mode	Cause	Effect	Crit.	Appraisal
5-1	Omission	Software terminates abnormally	Airbag not released in emergency.	C1	See 1.1, 1.2.
5-1.1	Omission	- Division by 0	See 1	C1	SR-47.3 Static Analysis
5-1.2	Omission	- Memory fault	See 1	C1	SR-47.4 Static Analysis
5-2	Omision	Software does not terminate	Airbag not released in emergency.	C1	SR-47.5 Static Analysis
5-3	Late	Computation takes too long.	Airbag not released in emergency.	C1	SR-47.6
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

The Seven Principles of Hazard Analysis

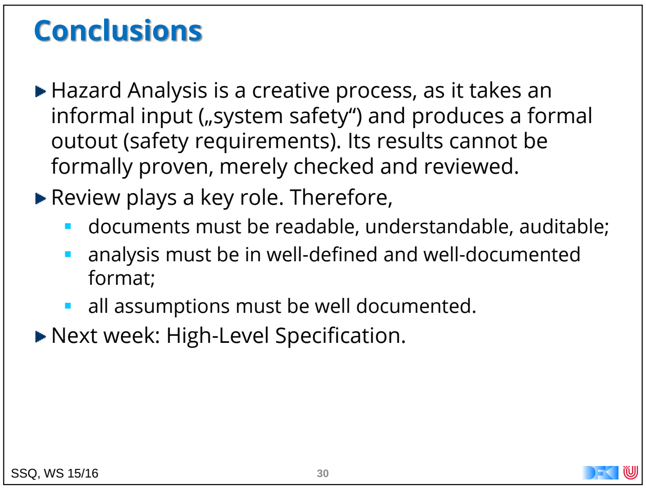
Ericson (2005)

1) Hazards, mishaps and risk are not chance events.
2) Hazards are created during design.
3) Hazards are comprised of three components.
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 outout (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.
- Next week: High-Level Specification.

Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 05 (09-11-2015)

High-Level Design with SysML

Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen

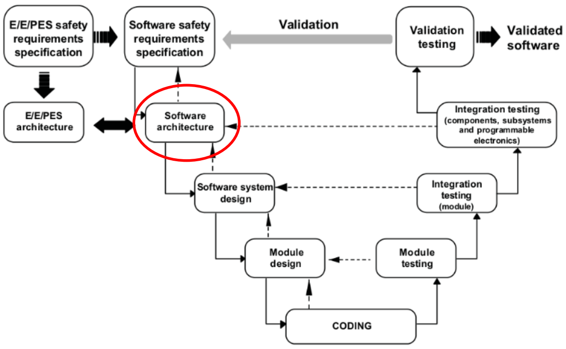
Where are we?

- 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 SysML
- 07: Detailed Specification with SysML
- 08: Testing
- 09 and 10: Program Analysis
- 11: Model-Checking
- 12: Software Verification (Hoare-Calculus)
- 13: Software Verification (VCG)
- 14: Conclusions

Your Daily Menu

- What is high-level design?
  - Describing the **structure** of the system at an abstract level
  - Should fit with **formal model** at lower level
- In which language?
  - Wide-spectrum specification languages such as Z, B, Event-B, CASL, …
  - Architectural languages
  - Modeling languages such as the UML
  - UML is very software-centred, hence SysML
- Today:
  - Introduction to SysML
  - Structural modeling in SysML

High-Level Design in the Development Cycle


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

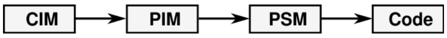
An Introduction to SysML

What is a model?

- „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.
- In other words: an **abstract representation of reality**.
- Purposes of models:
  - Analysing requirements
  - Understanding, communicating and capturing the design
  - Organizing information about a large system
  - Analyse design decisions early in the development process

Model-Driven Development (MDD, MDE)

- Recall the idea of MDD:
  - Describe problems on abstract level using *a modelling 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)



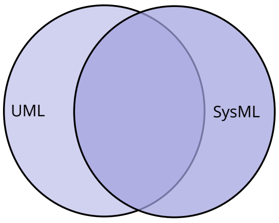
- However, using a modelling language like UML or SysML does not mean one has to employ MDD; in particular, we can still employ V-model-like approaches as required by safety standards.

The Unifed Modeling Language (UML)

- The UML grew out of a wealth of modelling languages in the 1990s, as James Rumbaugh, Grady Booch and Ivar Jacobson all worked at Rational Software.
- It was adopted by the Object Management Group (OMG) in 1997, and approved as ISO standard in 2005.
- UML 2 consists of
  - the superstructure to define diagrams,
  - a core meta-model,
  - the object constraint language (OCL),
  - an interchange format
- UML 2 is not a fixed language, it can be extended and customised using profiles.

The Systems Modeling Language SysML

- SysML is a *modeling language* for **systems engineering**
- Standardised in 2007 by the OMG (Ver. 1.0, now at 1.3)
- SysML Standard available at: http://www.omg.org/spec/SysML/1.3/PDF
- UML vs. SysML:



What for SysML?

- The aim of SysML (much like UML) is to serve as a standardised 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.

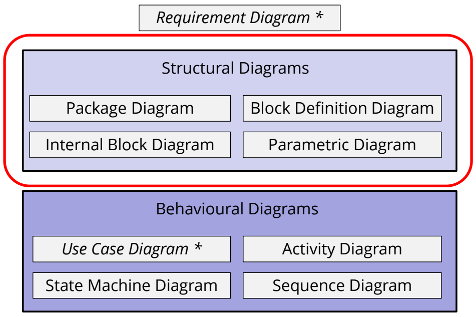
Views in SysML

- 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?
- Parametrisation:
  - 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 obstacles
- 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 5g, can not drive faster than 1m/s, laws concerning movement and trajectory, ...

SysML Diagrams



*Requirement Diagram **

Structural Diagrams

Package Diagram | Block Definition Diagram

Internal Block Diagram | Parametric Diagram

Behavioural Diagrams

*Use Case Diagram ** | Activity Diagram

State Machine Diagram | Sequence Diagram

* Not considered further.

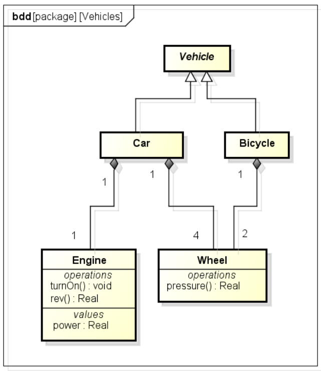
Structural Diagrams in SysML

Block Definition Diagram

- Corresponds to *class diagrams* in the UML
- 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.

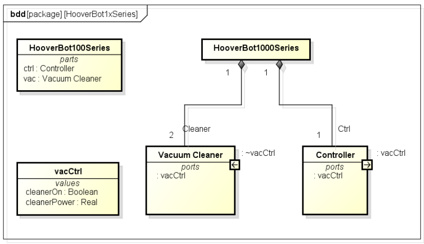
Example 1: Vehicles

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



Example 2: HooverBots

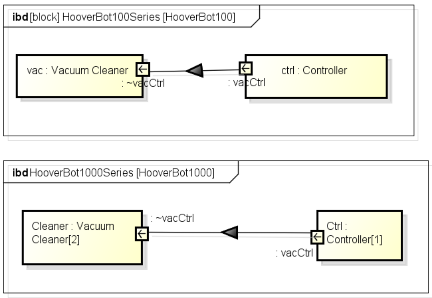
- The hoover bots have a control computer, and a vacuum cleaner.
  - HooverBot 100 has one v/c, Hoover 1000 has two.
  - Two ways to model this (i.e. two views)



Internal Block Diagrams

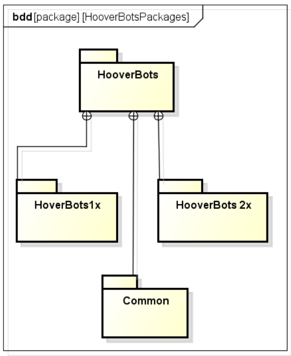
- Internal block diagrams decribe instances of blocks.
- Here, instances for HooverBots
- On this level, we can describe connections between ports (flow specifications)
  - Flow specifications have directions.

HooverBot 100 and 1000


ibd [block] HooverBot100Series [HooverBot100]
vac : Vacuum Cleaner
: ~vacCtrl
: vacCtrl
ctrl : Controller
ibd HooverBot1000Series [HooverBot1000]
Cleaner : Vacuum Cleaner[2]
: ~vacCtrl
: vacCtrl
Ctrl : Controller[1]

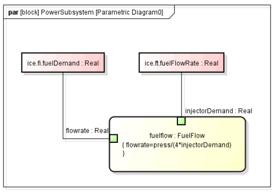
Package Diagrams

- 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.


par [block] PowerSubsystem [Parametric Diagram0]
ice.fi.fuelDemand : Real
ice.ft.fuelFlowRate : Real
injectorDemand : Real
flowrate : Real
fuelflow : FuelFlow
{ flowrate=press/(4*injectorDemand)
}

Source:
http://astah.net/tutorials/sysml/parametric

Modeling Tool: Astah-SysML

- Astah-SysML is available at

  http://astah.net/editions/sysml

- A faculty licence 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.

Summary

- High-level modelling describes the structure of the system at an abstract level.
- SysML is a standardised modelling 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.
- For this: OCL --- next week.

Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 06 (16-11-2015)

Formal Modelling with SysML and OCL

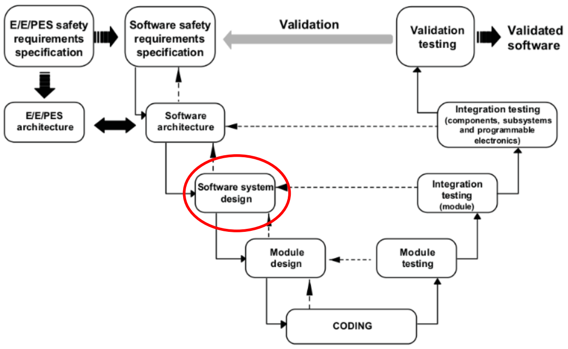
Christoph Lüth   Jan Peleska   Dieter Hutter

Universität Bremen

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 SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09 and 10: Program Analysis
- 11: Model-Checking
- 12: Software Verification (Hoare-Calculus)
- 13: Software Verification (VCG)
- 14: Conclusions

Formal Modelling in the Development Cycle


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

What is OCL?

- OCL is the **Object Constraint Language**.
- 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 **specificication 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 type; all expressions must be well-typed.
  - Types are classes, defined by class diagrams.

OCL can be used for the following:

- 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)

Example: A Flight-Booking System

- Flight destinations are given by
  - an IATA id, and a string
- A flight is given by
  - Source and destination, arrival and departure date, capacity and free seats
- A query asks for
  - a flight from/to at a given time and number of free seats
- Operations:
  - Query
  - Book a flight

Example: A Flight-Booking System

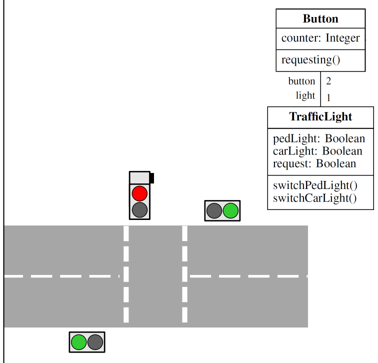
Possible constraints:

- No more free seats than capacity
- Source and destination must be disjoint
- Query must return „correct" flight
- Destination identifiers must be unique
- To book a flight:
  - Possible if enough free seats
  - Afterwards, number of free seats reduced

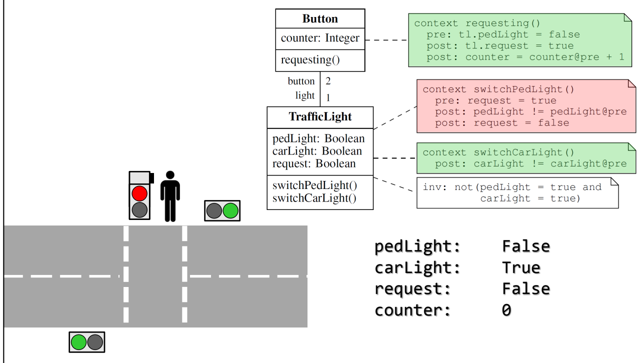
Possible extension:

- Query returns a schedule --- list of connecting flights

Example: The Traffic Light



Example: The Traffic Light



```
context requesting()
  pre: tl.pedLight = false
  post: tl.request = true
  post: counter = counter@pre + 1
```

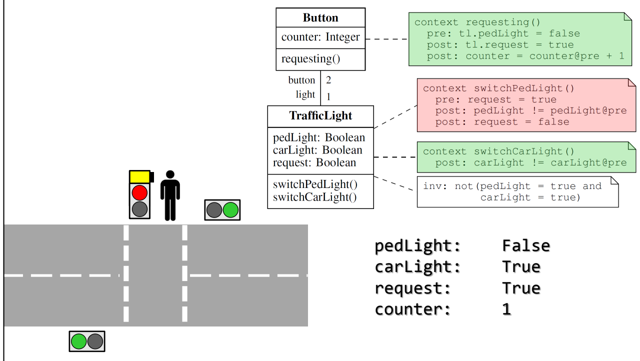
```
context switchPedLight()
  pre: request = true
  post: pedLight != pedLight@pre
  post: request = false
```

```
context switchCarLight()
  post: carLight != carLight@pre
```

```
inv: not(pedLight = true and
         carLight = true)
```

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

Example: The Traffic Light



```
context requesting()
  pre: tl.pedLight = false
  post: tl.request = true
  post: counter = counter@pre + 1
```

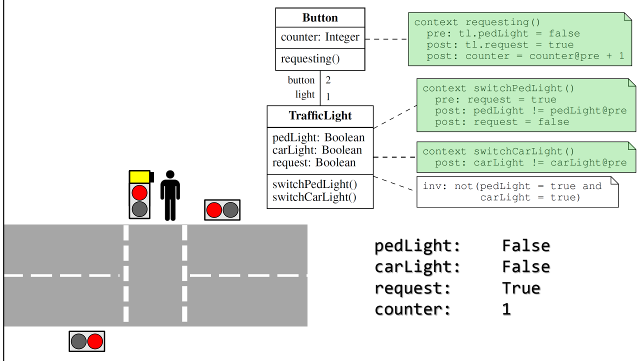
```
context switchPedLight()
  pre: request = true
  post: pedLight != pedLight@pre
  post: request = false
```

```
context switchCarLight()
  post: carLight != carLight@pre
```

```
inv: not(pedLight = true and
         carLight = true)
```

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

Example: The Traffic Light



```
context requesting()
  pre: tl.pedLight = false
  post: tl.request = true
  post: counter = counter@pre + 1
```

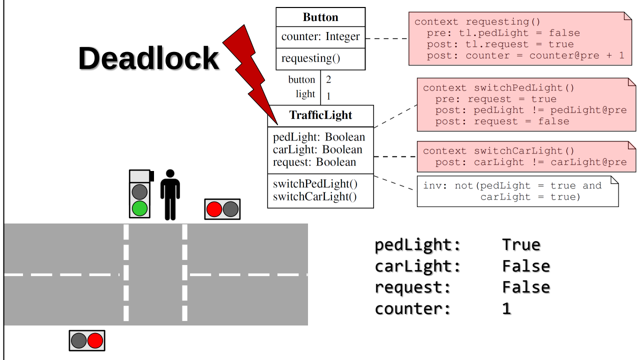
```
context switchPedLight()
  pre: request = true
  post: pedLight != pedLight@pre
  post: request = false
```

```
context switchCarLight()
  post: carLight != carLight@pre
```

```
inv: not(pedLight = true and
         carLight = true)
```

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

Example: The Traffic Light



```
context requesting()
  pre: tl.pedLight = false
  post: tl.request = true
  post: counter = counter@pre + 1
```

```
context switchPedLight()
  pre: request = true
  post: pedLight != pedLight@pre
  post: request = false
```

```
context switchCarLight()
  post: carLight != carLight@pre
```

```
inv: not(pedLight = true and
         carLight = true)
```

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

OCL Basics

- The language is typed: each expression has a type.
- Three-valued logic (**Kleene logic**)
  - Actually, more like four-valued `(null)`
- Expressions always live in a **context:**
  - **Invariants** on classes, interfaces, types.

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

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

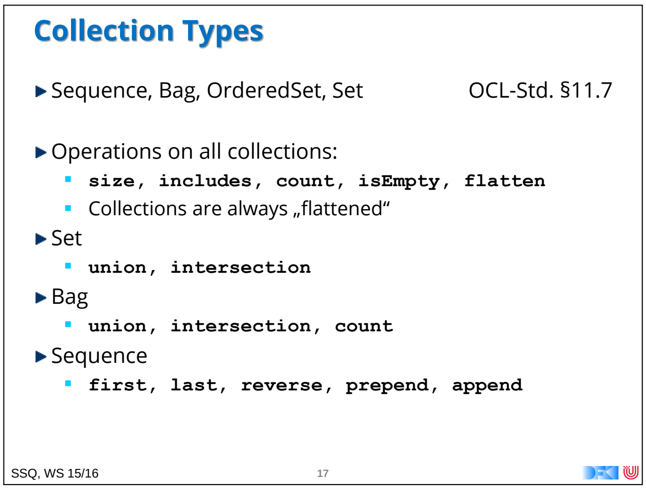
```
context Type :: op(a1: Type) : Type
  pre Name: expr
  post Name: expr
```

OCL Types

- Basic types:
  - **`Boolean, Integer, Real, String`**
  - **`OclAny, OclType, OclVoid`**
- Collection types:
  - **`Sequences, Bag, OrderedSet, Set`**
- Model types

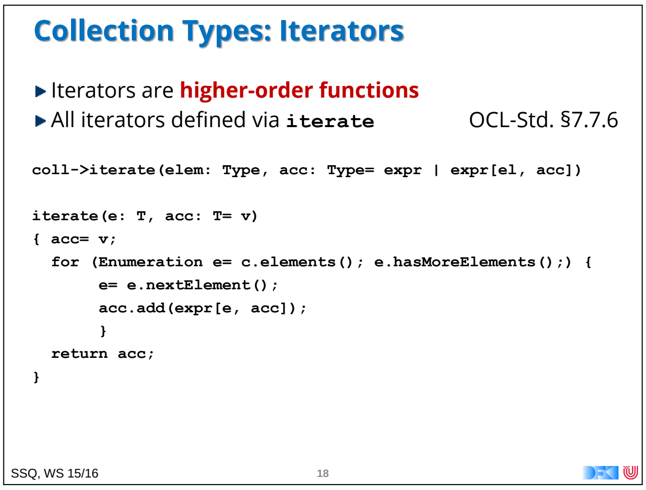
Basic types and operations

- Integer ($\mathbb{Z}$) OCL-Std. §11.5.2
- Real ($\mathbb{R}$) 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. §11.7
- Operations on all collections:
  - **`size, includes, count, isEmpty, flatten`**
  - Collections are always „flattened“
- Set
  - **`union, intersection`**
- Bag
  - **`union, intersection, count`**
- Sequence
  - **`first, last, reverse, prepend, append`**

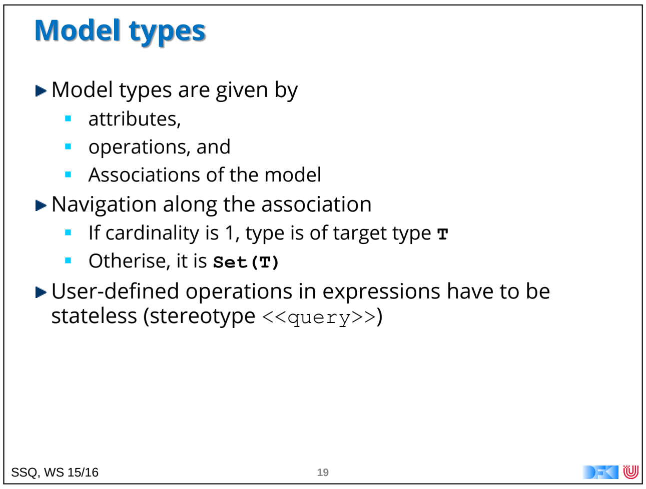


Collection Types: Iterators

- Iterators are **higher-order functions**
- All iterators defined via **`iterate`** OCL-Std. §7.7.6

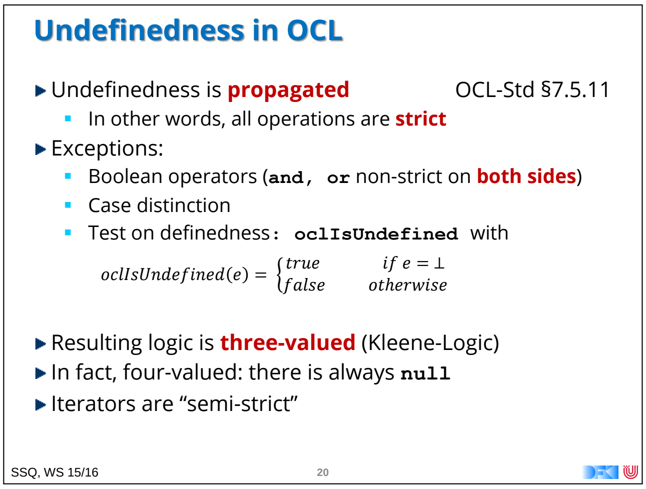
```
coll->iterate(elem: Type, acc: Type= expr | expr[el, acc])

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



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>>`)

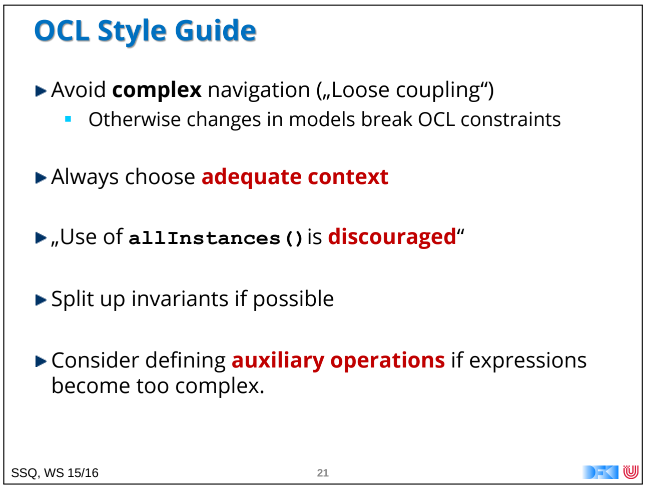


Undefinedness in OCL

- Undefinedness is **propagated** OCL-Std §7.5.11
  - In other words, all operations are **strict**
- Exceptions:
  - Boolean operators (**`and, or`** non-strict on **both sides**)
  - Case distinction
  - Test on definedness**:** **`oclIsUndefined`** with

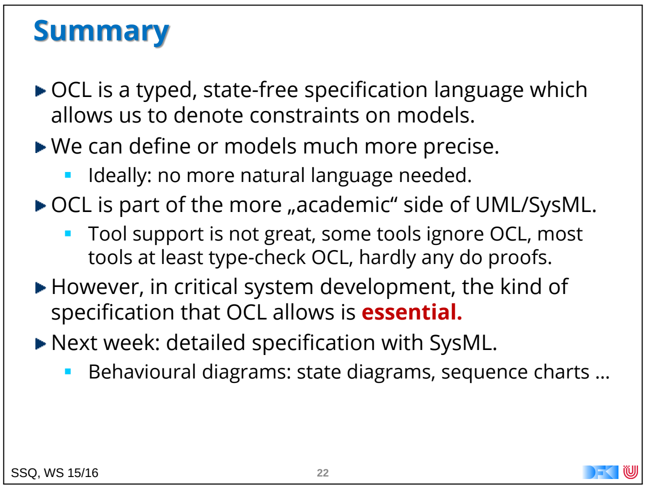
$$oclIsUndefined(e) = \begin{cases} true & if\ e = \bot \\ false & otherwise \end{cases}$$

- Resulting logic is **three-valued** (Kleene-Logic)
- In fact, four-valued: there is always **`null`**
- Iterators are “semi-strict”



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.**
- Next week: detailed specification with SysML.
  - Behavioural diagrams: state diagrams, sequence charts …

Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 07 (23-11-2015)

Detailed Specification with SysML

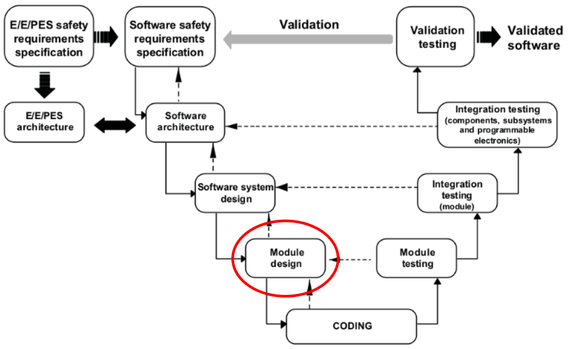
Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen

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 SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09 and 10: Program Analysis
- 11: Model-Checking
- 12: Software Verification (Hoare-Calculus)
- 13: Software Verification (VCG)
- 14: Conclusions

Detailed Specification in the Development Cycle


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

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**:
  - Which operations can be called, what are their pre/post-conditions and effects.
  - This can be modelled using OCL.
  - Alternatively, we can model the system's internal states by a **state machine**, which has states and guarded transitions between them.
- By their (internal) **structure**:
  - Modelling the control flow by flow charts aka. **activity charts.**
  - There are also a variety of **action languages** (platform-independent programming languages) for UML, but these are not standard for SysML.

State Diagrams: Basics

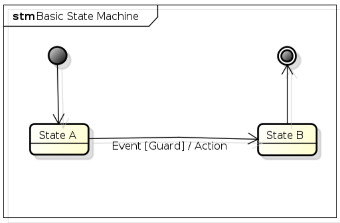
- State diagrams are a particular form of (hierarchical) finite state machines.

A **finite state machine** is given by $M = \langle\Sigma, \rightarrow\rangle$ where
- Σ is a finite set of states, and
- $\rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation which is left-total.

- 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:

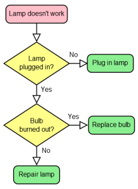
Signal events	***event name/***
Call events	***operation name/***
Time events	**after(t)/**
Change event	**when(e)/**
Entry events	**Entry/**
Exit events	**Exit/**

State Diagram Elements (SysML Ref. §13.2)

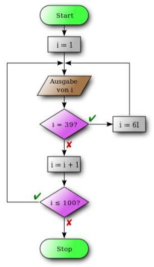
- Choice pseudo state
- Composite state
- Entry point
- Exit point
- Final state
- *History pseudo states*
- Initial pseudo state
- Junction pseudo state
- Receive signal action
- Send signal action
- Action
- Region
- Simple state
- State list
- State machine
- Terminate node
- Submachine state

Activity Charts: Foundations

- The activity charts of SysML (UML) are a variation of old-fashioned **flow charts.**
  - Standardised 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 sequence and conditions for 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)
- This is performed by means of **control flow** and **object flow** models
- Control flow allows to **disable** and **enable** (sub-) activities using these two enumeration values.
- An activity execution results in the execution of a set of actions in some specific order.
- Activity executions may comprise several logical **execution threads.**

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 (SysML Ref. §11.2.1)

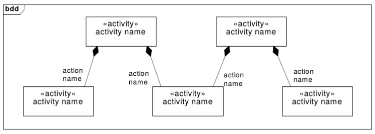
- Nodes:
  - Action nodes
  - Activities
  - Decision nodes
  - Final nodes
  - Fork nodes
  - Initial nodes
  - Local pre/post-conditions
  - Merge nodes
  - Object nodes
  - *Probabilities and rates*
- Paths (arrows):
  - Control flow
  - Object flow
  - Probability and rates
- Activities in BDDs
-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

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.

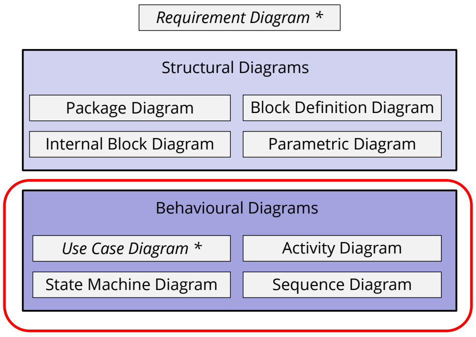
Activity Diagrams – Links With BDDs

- 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).

SysML Diagrams Overview



*Requirement Diagram **

Structural Diagrams

Package Diagram | Block Definition Diagram

Internal Block Diagram | Parametric Diagram

Behavioural Diagrams

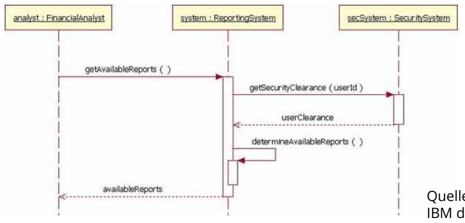
*Use Case Diagram ** | Activity Diagram

State Machine Diagram | Sequence Diagram

* Not considered further.

Sequence Diagrams

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


analyst : FinancialAnalyst
system : ReportingSystem
secSystem : SecuritySystem
getAvailableReports ( )
getSecurityClearance (userId)
userClearance
determineAvailableReports ( )
availableReports

Quelle:
IBM developerWorks

- We may consider concurrency further later on.

Summary

- 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.

Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 08 (30-11-2015)

Testing

Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen

Where are we?

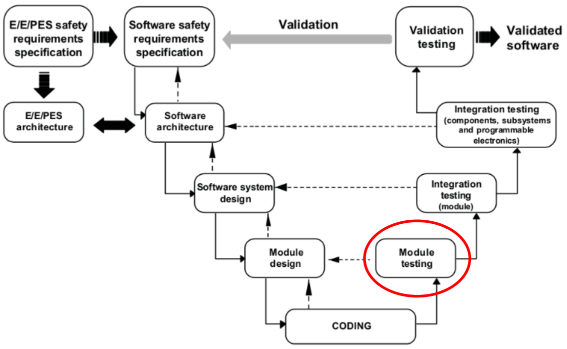
- 01: Concepts of Quality
- 02: Legal Requirements: Norms and Standards
- 03: The Software Development Process
- 04: Hazard Analysis
- 05: High-Level Design with SysML
- 06: Formal Modelling with SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09: Program Analysis
- 10 and 11: Software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

Your Daily Menu

What is testing?

- Different **kinds** of tests.
- Different test methods: **black-box** vs. **white-box**.
- The basic problem: cannot test **all** possible inputs.
- Hence, coverage criteria: how to test **enough**.

Testing in the Development Cycle


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

What is Testing?

> Testing is the process of executing a program or system with the intent of finding errors.
>
> *Myers, 1979*

- In our sense, testing is selected, controlled program execution.
- The **aim** of testing is to detect bugs, such as
  - derivation of occurring characteristics of quality properties compared to the specified ones;
  - inconsistency between specification and implementation;
  - or structural features of a program 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*

The Testing Process

- Test cases, test plan, etc.
- System-under-test (s.u.t.)
- 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** tests 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.
- The 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, eg. Module architecture;
  - **White-box**: the inner structure of the s.u.t. is known, and tests cases are derived from 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

Example: A Company Bonus System
The loyalty bonus shall be computed depending on the time of employment. For employes 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 $40\ {}^{m}/_{s^2}$.

- Equivalence classes or limits?

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 ORM in Python, Java.

Other approaches: Monte-Carlo Testing

- In Monte-Carlo testing (or random testing), we generate **random** input values, and check the results against a given spec.
- This requires **executable** specifications.
- Attention needs to be paid to the **distribution** values.
- Works better with **high-level languages** (Java, Scala, Haskell) where the datatypes represent more information on an abstract level.
  - ScalaCheck, QuickCheck for Haskell
- Example: consider list reversal in C, Java, Haskell
  - Executable spec:
    - Reversal is idempotent.
    - Reversal 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 are elementary statements (e.g. assignments, **return**, **break**, . . .), and control expressions (eg. in conditionals and loops), and
- there is a vertex from *n* to *m* if the control flow can reach node *m* coming from *n*.

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

A Very Simple Programming Language

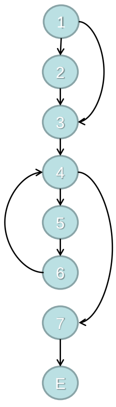
- In the following, we use a very simple language with a C-like syntax.
- **Arithmetic** operators given by
  $$a\ ::= x \mid n \mid a_1\ op_a\ a_2$$
  with x a variable, n a numeral, op_aarith. op. (e.g. +, -, *)
- **Boolean** operators given by
  $$b\ \coloneqq \text{true} \mid \text{false} \mid \text{not}\ b \mid b_1 op_b\ b_2 \mid a_1 op_r\ a_2$$
  with op_b boolean operator (e.g. and, or) and op_r a relational operator (e.g. =, <)
- **Statements** given by
  $S\ ::=$
  $$[x \coloneqq a]^l \mid [skip]^l \mid S_1; S_2 \mid if\ [b]^l\ \{S_1\}\ else\ \{S_2\} \mid while\ [b]^l\ \{S\}$$

We may write the labels als comments

```
x:= a+ 10; /* 1 */ if (y < 3) /* 2 */ { x:= x+1; /* 3 */  } else { y:= y+1; /* 4 */ }
```

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.

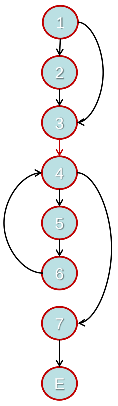
Examples:
- [1,3,4,7, E]
- [1,2,3,4,7, E]
- [1,2,3,4,5,6,4,7, E]
- [1,3,4,5,6,4,5,6,4,7, E]
- …

Coverage

- **Statement coverage**: Each **node** in the cfg is visited at least once.
- **Branch coverage**: Each **vertex** in the cfg is traversed at least once.
- **Decision coverage**: Like branch coverage, but specifies how often **conditions** (branching points) must be evaluated.
- **Path coverage**: Each **path** in the cfg is executed at least once.

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 */
```



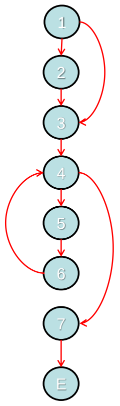
- Which (minimal) path covers all statements?

  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) path covers all vertices?

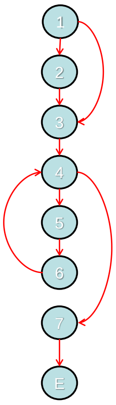
  $p_1 = [1,2,3,4,5,6,4,7,E]$
  $p_2 = [1,3,4,7,E]$
- Which states generate p_1, p_2?

	p_1	p_2
x	−1	0
y	any	any
z	any	any

- Note p_3 (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.
  - 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 (100%);
  - But no **practical** relevance.

Decision Coverage

- Decision coverage is **more** then branch coverage, but less then full **path** coverage.
- Decision coverage requires that for all decisions in the program, each possible outcome is considered once.
- **Problem**: cannot sufficiently distinguish boolean expressions.
  - 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

- **In simple condition coverage**, for each condition in the program, each elementary boolean term evaluates to *True* and *False* at least once.
- Note that this does not say much about the possible value of the condition.
- Examples and possible solutions:

if (temperature > 90 && pressure > 120) {...

C1	*C2*	*Result*
True	True	True
True	False	False
False	True	False
False	False	False

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

False	False	False	← not needed
False	True	False	
True	False	False	
True	True	True	

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

Modified Condition/Decision Coverage

- Modified Condition/Decision Coverage (MC/DC) is required by **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 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

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**.

Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 09 (07-12-2015)

Static Program Analysis

Christoph Lüth   Jan Peleska   Dieter Hutter

Universität Bremen

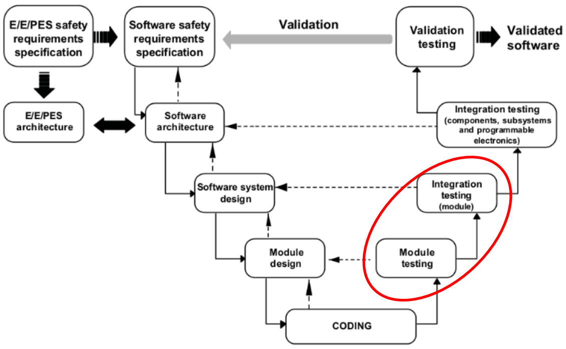
Where are we?

- 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 SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09: Static Program Analysis
- 10 and 11: Software Verification (Hoare-Calculus)
- 12: Model-Checking
- 13: Concurrency
- 14: Conclusions

Today: Static Program Analysis

- Analysis of run-time behavior 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 tasks
  - Does the variable *x* have a constant value ?
  - Is the value of the variable *x* always positive ?
  - Can the pointer *p* be null at a given program point ?
  - What are the possible values of the variable *y* ?
- These tasks can be used for verification (e.g. is there any possible dereferencing of the null pointer), or for optimisation when compiling.

Program Analysis in the Development Cycle


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

Usage of Program Analysis

Optimising compilers

- Detection of sub-expressions that are evaluated multiple times
- Detection of unused local variables
- Pipeline optimisations

Program verification

- Search for runtime errors in programs
- Null pointer dereference
- Exceptions which are thrown and not caught
- Over/underflow of integers, rounding errors with floating point numbers
- Runtime estimation (worst-caste executing time, wcet)
- In other words, **specific** verification **aspects**.

Program Analysis: The Basic Problem

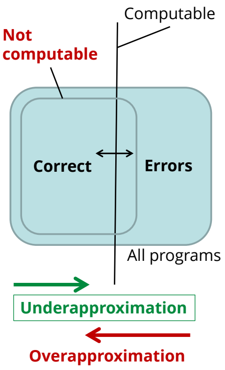
- Basic Problem:

> All interesting program properties are undecidable.

- Given a property P and a program p, we say $p \vDash P$ if a P holds for p. An algorithm (tool) ϕ which decides P is a computable predicate $\phi: p \to Bool$. We say:
  - ϕ is **sound** if   whenever $\phi(p)$ then $p \vDash P$.
  - ϕ is **safe** (or **complete**) if   whenever $p \vDash P$ then $\phi(p)$.
- From the basic problem it follows that there are no sound and safe tools for interesting properties.
  - In other words, all interesting tools must either under- or overapproximate.

Program Analysis: Approximation

- **Underapproximation** only finds correct programs but may miss out some
  - Useful in optimising compilers
  - Optimisation must respect semantics of program, but may optimise.
- **Overapproximation** 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**

Flow Sensitivity

Flow-sensitive analysis

- Considers program's flow of control
- Uses control-flow graph as a representation of the source
- Example: available expressions analysis

Flow-insensitive analysis

- Program is seen as an unordered collection of statements
- Results are valid for any order of statements e.g. *S1* ; *S2* vs. *S2* ; *S1*
- Example: type analysis (inference)

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

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

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) ?

Our Simple Programming Language

- In the last lecture, we introduced a very simple language with a C-like syntax.
- Synposis:

Arithmetic operators given by

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

Boolean operators given by

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

$$op_b \in \{and, or\}, op_r \in \{=, <, \leq, >, \geq, \neq\}$$

Statements given by

$$S \ ::=$$

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

Computing the Control Flow Graph

- To calculate the cfg, we define some functions on the abstract syntax:
  - The initial label (entry point) $\mathrm{init}: S \ \rightarrow Lab$
  - The final labels (exit points) $\mathrm{final}: S \ \rightarrow \ \mathbb{P}(Lab)$
  - The elementary blocks $\mathrm{block}: S \rightarrow \mathbb{P}(Blocks)$ where an elementary block is
    - an assignment [x:= a],
    - or [skip],
    - or a test [b]
  - The control flow $\mathrm{flow}: S \rightarrow \ \mathbb{P}(Lab \times Lab)$ and reverse control $\mathrm{flow}^{\mathrm{R}}: S \rightarrow \ \mathbb{P}(Lab \times Lab)$.
- The **control flow graph** of a program S is given by
  - elementary blocks $\mathrm{block}(S)$ as nodes, and
  - flow(S) as vertices.

Labels, Blocks, Flows: Definitions

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

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

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

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

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

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

$FV(a)$ = 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)}

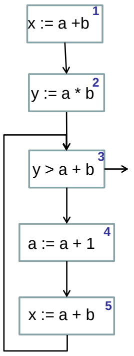
flow^R(P) = {(2, 1), (3, 2), (4, 3), (5, 4), (3, 5)}

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

FV(a + b) = {a, b}

FV(P) = {a, b, x, y}

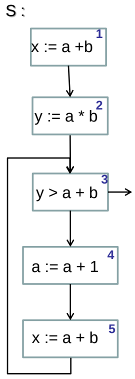
Aexp(P) = {a+b, a*b, a+1}



Available Expression Analysis

- The available expression analysis will determine:

> For each program point, which expressions must have already been computed, and not modified, on all paths to this program point.



Available Expression Analysis

gen([x :=a]^l) = $\{a' \in Aexp(a) | \, x \notin FV('a)\}$
gen([skip]^l) = ∅
gen([b]^l) = $Aexp(b)$

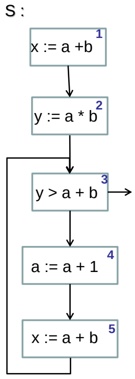
kill([x :=a]^l) = $\{a' \in Aexp(S) | \, x \in FV('a)\}$
kill([skip]^l) = ∅
kill([b]^l) = ∅

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

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

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

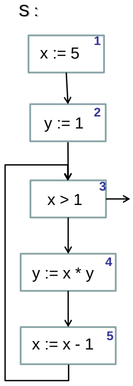
l	AE_{in}	AE_{out}
1	∅	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	∅
5	∅	{a+b}



Reaching Definitions Analysis

- Reaching definitions (assignment) analysis determines if:

> An assignment of the form [x := a]^l may reach a certain program point k if there is an execution of the program where x was last assigned a value at l when the program point k is reached



Reaching Definitions Analysis

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

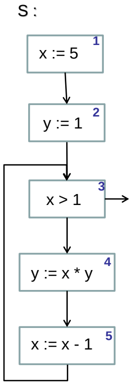
l	kill(B^l)	gen(B^l)
1	{(x,?), (x,1),(x,5)}	{(x, 1)}
2	{(y,?), (y,2),(y,4)}	{(y, 2)}
3	∅	∅
4	{(y,?), (y,2),(y,4)}	{(y, 4)}
5	{(x,?), (x,1),(x,5)}	{(x, 5)}

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

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

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

l	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)}

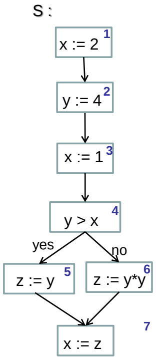


Live Variables Analysis

- 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 live at the exit from that point.

- Application: dead code elemination.



Live Variables Analysis

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

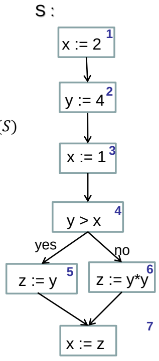
kill([x :=a]^l) = $\{x\}$
kill([skip]^l) = ∅
kill([b]^l) = ∅

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

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

l	kill(*l*)	gen(*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}	∅



First Generalized Schema

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

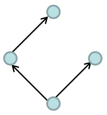
With:

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

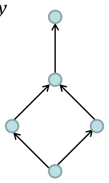
Backward analysis: **Flow** = flow^R, • = IN, ∘ = OUT
Forward analysis: **Flow** = flow, • = OUT, ∘ = IN

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 ("Verbund") is a partial order L = (M, ⊑) such that

- ⊔X and ⊓X exist for all X ⊆ M
- Unique greatest element ⊤ = ⊔M = ⊓∅
- Unique least element ⊥ = ⊓M = ⊔∅

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 : L → L with l being a label
- Knowledge transfer is monotone
  - $\forall\, x, y.\, x \sqsubseteq y \Longrightarrow f_l(x) \sqsubseteq f_l(y)$
- Space F of transfer functions
  - F contains all transfer functions f_l
  - F contains the identity function id: $\forall x \in M.\, id(x) = x$
  - F is closed under composition: $\forall\, f, g \in F.\, (g \circ f) \in F$

The Generalized Analysis

- $\text{Analysis}_\circ(\,l\,) = \sqcup\,\{\text{Analysis}_\bullet(\,l'\,) \mid (l', l) \in Flow(S)\} \sqcup \{\,\iota'_E\,\}$
  with $\iota'_E = \begin{cases} EV & \text{if } l \in E \\ \bot & \text{otherwise} \end{cases}$
- $\text{Analysis}_\bullet(\,l\,) = f_l(\,\text{Analysis}_\circ(\,l\,)\,)$

With:

- L property space representing data flow information with $(L, \sqsubseteq)$ a lattice
- $Flow$ is a finite flow (i.e. $flow$ or $flow^R$)
- EV is an extremal value for the extremal labels E (i.e. $\{init(S)\}$ or $final(S)$
- transfer functions f_l of a space of transfer functions F

Summary

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

Systeme Hoher Sicherheit und Qualität
Universität Bremen WS 2015/2016

Lecture 10 (14.12.2015)

Foundations of Software Verification

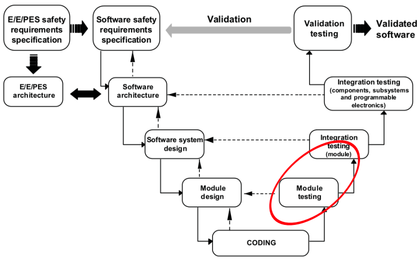
Christoph Lüth   Jan Peleska   Dieter Hutter

Universität Bremen

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 SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09: Program Analysis
- 10: Foundations of Software Verification
- 11: Verification Condition Generation
- 12: Semantics of Programming Languages
- 13: Model-Checking
- 14: Conclusions and Outlook

Today: Software Verification using Floyd-Hoare logic


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

- The Floyd-Hoare calculus proves properties of imperative programs.
- Thus, it is at home in the lower levels of the verification branch, much like the static analysis from last week.
- It is far more powerful than static analysis — and hence, far more complex to use (it requires user interaction, and is not automatic).

Idea

- What does this compute? $P = N!$
- How can we prove this?
- Inuitively, we argue about which value variables have at certain points in the program.
- Thus, to prove properties of imperative programs like this, we need a formalism where we can formalise assertions of the program properties at certain points in the exection, and which tells us how these assertions change with program execution.

```
{1 ≤ N}
P := 1;
C := 1;
while (C ≤ N) {
   P := P ∗ C;
   C := C + 1
};
{P = N!}
```

Floyd-Hoare-Logic

- Floyd-Hoare-Logic consists of a set of rules to derive valid assertions about programs. The assertions are denoted in the form of Floyd-Hoare-Triples $\{P\}\,p\,\{Q\}$, with P the precondition, p a program and Q the postcondition.
- The logical language has both logical variables (which do not change), and program variables (the value of which changes with program execution).
- Floyd-Hoare-Logic has one basic principle and one basic trick.
- The principle is to abstract from the program state into the logical language; in particular, assignment is mapped to substitution.
- The trick is dealing with iteration: iteration corresponds to induction in the logic, and thus is handled with an inductive proof. The trick here is that in most cases we need to strengthen our assertion to obtain an invariant.

Recall Our Small Language

- Arithmetic Expressions (**AExp**)

$$a ::= \mathbf{N} \mid \mathbf{Loc} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$$

  with variables **Loc**, numerals **N**

- Boolean Expressions (**BExp**)

$$b ::= \mathbf{true} \mid \mathbf{false} \mid a_1 = a_2 \mid a_1 < a_2 \mid \neg b \mid b_1 \wedge b2 \mid b_1 \vee b_2$$

- Statements (**Com**)

$$\begin{aligned} c ::= {} & \mathbf{skip} \mid \mathbf{Loc} := \mathbf{AExp} \mid \mathbf{skip} \mid c_1; c_2 \\ & \mid \mathbf{if}\ b\ \{c_1\}\ \mathbf{else}\ \{c_2\} \mid \mathbf{while}\ b\ \{c\} \end{aligned}$$

Semantics of our Small Language

- The semantics of an imperative language is state transition: the program has an ambient state, and changes it by assigning values to certain locations
- Concrete example: execution starting with $N = 3$

$$\begin{array}{|c|c|}\hline P & ? \\ \hline C & ? \\ \hline N & 3 \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|c|}\hline P & 1 \\ \hline C & ? \\ \hline N & 3 \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|c|}\hline P & 1 \\ \hline C & 1 \\ \hline N & 3 \\ \hline\end{array} \rightsquigarrow \begin{array}{|c|c|}\hline P & 1 \\ \hline C & 1 \\ \hline N & 3 \\ \hline\end{array} \rightsquigarrow \ldots \begin{array}{|c|c|}\hline P & 6 \\ \hline C & 4 \\ \hline N & 3 \\ \hline\end{array}$$

Semantics in a nutshell

- Expressions evaluate to values **Val**(in our case, integers)
- A program state maps locations to values: $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{Val}$
- A programs maps an initial state to possibly a final state (if it terminates)
- Assertions are predicates over program states.

Floyd-Hoare-Triples

Partial Correctness ($\models \{P\}\,c\,\{Q\}$)

c is partial correct with precondition P and postcondition Q if:
for all states σ which satisfy P
if the execution of c on σ terminates in σ'
then σ' satisfies Q

Total Correctness ($\models [P]\,c\,[Q]$)

c is total correct with precondition P and postcondition Q if:
for all states σ which satisfy P
the execution of c on σ terminates in σ'
and σ' satisfies Q

- $\models \{\mathbf{true}\}\ \mathbf{while}\ \mathbf{true}\ \{\mathbf{skip}\}\ \{\mathbf{true}\}$ holds
- $\models [\mathbf{true}]\ \mathbf{while}\ \mathbf{true}\ \{\mathbf{skip}\}\ [\mathbf{true}]$ does not hold

Assertion Language

- Extension of **AExp** and **BExp** by
  - logical variables **Var** $v := n, m, p, q, k, l, u, v, x, y, z$
  - defined functions and predicates on **Aexp** $n!, \sum_{i=1}^{n}, \ldots$
  - implication, quantification $b_1 \Rightarrow b_2, \forall v.\, b, \exists v.\, b$
- **Aexpv**

$$a ::= \ \mathbf{N} \mid \mathbf{Loc} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \mid \mathbf{Var} \mid f(e_1, \ldots, e_n)$$

- **Bexpv**

$$\begin{aligned} b ::= \ & \mathbf{true} \mid \mathbf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b2 \mid b_1 \vee b_2 \\ & \mid b_1 \Rightarrow b_2 \mid p(e_1, \ldots, e_n) \mid \forall v.\, b \mid \exists v.\, b \end{aligned}$$

Rules of Floyd-Hoare-Logic

- The Floyd-Hoare logic allows us to derive assertions of the form $\vdash \{P\}\, c\, \{Q\}$
- The calculus of Floyd-Hoare logic consists of six rules of the form

$$\frac{\vdash \{P_1\}\, c_1\, \{Q_1\} \ldots \vdash \{P_n\}\, c_n\, \{Q_n\}}{\vdash \{P\}\, c\, \{Q\}}$$

- This means we can derive $\vdash \{P\}\, c\, \{Q\}$ if we can derive $\vdash \{P_i\}\, c_i\, \{Q_i\}$
- There is one rule for each construction of the language.

Rules of Floyd-Hoare Logic: Assignment

$$\frac{}{\vdash \{B[e/X]\}\, X := e\, \{B\}}$$

- An assigment X:=e changes the state such that at location X we now have the value of expression e. Thus, in the state before the assignment, instead of X we must refer to e.
- It is quite natural to think that this rule should be the other way around.
- Examples:

```
X := 10;
```
$\{0 < 10 \longleftrightarrow (X < 10)[X/0]\}$
```
X := 0
```
$\{X < 10\}$

$\{X < 9 \longleftrightarrow X + 1 < 10\}$
```
X := X+ 1
```
$\{X < 10\}$

Rules of Floyd-Hoare Logic: Conditional and Sequencing

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \mathbf{if}\ \ b\ \{c_0\}\ \ \mathbf{else}\ \{c_1\}\, \{B\}}$$

- In the precondition of the positive branch, the condition b holds, whereas in the negative branch the negation $\neg b$ holds.
- Both branches must end in the same postcondition.

$$\frac{\vdash \{A\}\, c_0\, \{B\} \qquad \vdash \{B\}\, c_1\, \{C\}}{\vdash \{A\}\, c_0; c_1\, \{C\}}$$

- We need an intermediate state predicate B.

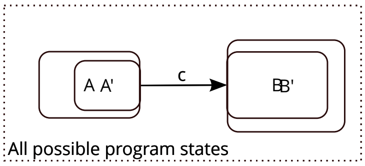
Rules of Floyd-Hoare Logic: Iteration

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \mathbf{while}\ \ b\ \{c\}\, \{A \wedge \neg b\}}$$

- Iteration corresponds to induction. Recall that in (natural) induction we have to show the same property P holds for 0, and continues to hold: if it holds for n, then it also holds for $n + 1$.
- Analogously, here we need an invariant A which has to hold both before and after the body (but not necessarily in between).
- In the precondition of the body, we can assume the loop condition holds.
- The precondition of the iteration is simply the invariant A, and the postcondition of the iteration is A and the negation of the loop condition.

Rules of Floyd-Hoare Logic: Weakening

$$\frac{A' \longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$



- $\models \{A\}\, c\, \{B\}$ means that whenever we start in a state where A holds, c ends (if it does) in state where B holds.
- Further, for two sets of states, $P \subseteq Q$ iff $P \longrightarrow Q$.
- We can restrict the set A to A' ($A' \subseteq A$ or $A' \longrightarrow A$) and we can enlarge the set B to B' ($B \subseteq B'$ or $B \longrightarrow B'$), and obtain $\models \{A'\}\, c\, \{B'\}$.

Overview: Rules of Floyd-Hoare-Logic

$$\frac{}{\vdash \{A\}\ \mathbf{skip}\ \{A\}} \qquad \frac{}{\vdash \{B[e/X]\}\, X := e\, \{B\}}$$

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \mathbf{if}\ \ b\ \{c_0\}\ \ \mathbf{else}\ \{c_1\}\, \{B\}}$$

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \mathbf{while}\ \ b\ \{c\}\, \{A \wedge \neg b\}} \qquad \frac{\vdash \{A\}\, c_0\, \{B\} \qquad \vdash \{B\}\, c_1\, \{C\}}{\vdash \{A\}\, c_0; c_1\, \{C\}}$$

$$\frac{A' \longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

Properties of Hoare-Logic

Soundness

If $\vdash \{P\}\, c\, \{Q\}$, then $\models \{P\}\, c\, \{Q\}$

- If we derive a correctness assertion, it holds.
- This is shown by defining a formal semantics for the programming language, and showing that all rules are correct wrt. to that semantics.

Relative Completeness

If $\models \{P\}\, c\, \{Q\}$, then $\vdash \{P\}\, c\, \{Q\}$ except for the weakening conditions.

- Failure to derive a correctness assertion is always due to a failure to prove some logical statements (in the weakening).
- First-order logic itself is incomplete, so this result is as good as we can get.

The Need for Verification

Consider the following variations of the faculty example.
Which are correct?

```
$\{1 \le N\}$
P := 1;
C := 1;
while (C≤N) {
   C := C+1;
   P := P∗C
}
$\{P = N!\}$
```

```
$\{1 \le N\}$
P := 1;
C := 1;
while (C<N) {
   C := C+1;
   P := P∗C
}
$\{P = N!\}$
```

```
$\{1 \le N \wedge n = N\}$
P := 1;
while (70<N) {
   P := P∗N;
   N := N−1
}
$\{P = n!\}$
```

A Hatful of Examples

```
$\{i = Y \wedge Y \geq 0\}$
X := 1;
while (¬ (Y = 0)) {
 Y := Y−1;
 X := 2∗X
}
$\{X = 2^i\}$
```

```
$\{A \geq 0 \wedge B \geq 0\}$
Q := 0;
R := A−(B∗Q);
while (B ≤ R) {
 Q := Q+1;
 R := A−(B∗Q)
}
$\{A = B * Q + R \wedge R < B\}$
```

```
$\{0 < A\}$
T:= 1;
S:= 1;
I:= 0;
while (S ≤ A) {
   T := T+ 2;
   S := S+ T;
   I := I+ 1
   }
$\{I * I <= A \wedge A < (I + 1) * (I + 1)\}$
```

Summary

- Floyd-Hoare logic in a nutshell:
  - The logic abstracts over the concrete program state by program assertions
  - Program assertions are boolean expressions, enriched by logical variables (and more)
  - We can prove partial correctness assertions of the form $\models \{P\}\, c\, \{Q\}$ (or total $\models [P]\, c\, [Q]$).
- Validity (correctness wrt a real programming language) depends very much on capturing the exact semantics formally.
- Floyd-Hoare logic itself is rarely used directly in practice, verification condition generation is — see next lecture.

Systeme Hoher Sicherheit und Qualität
Universität Bremen WS 2015/2016

Lecture 11 (11.01.2016)

Verification Condition Generation

Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen

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 SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09: Program Analysis
- 10: Foundations of Software Verification
- 11: Verification Condition Generation
- 12: Semantics of Programming Languages
- 13: Model-Checking
- 14: Conclusions and Outlook

Introduction

- In the last lecture, we learned about the Floyd-Hoare calculus.
- It allowed us to state and prove correctness assertions about programs, written as $\{P\}\,c\,\{Q\}$.
- The problem is that proofs of $\vdash \{P\}\,c\,\{Q\}$ are exceedingly tedious, and hence not viable in practice.
- We are looking for a calculus which reduces the size (and tediousness) of Floyd-Hoare proofs.
- The starting point is the relative completeness of the Floyd-Hoare calculus.

Completeness of the Floyd-Hoare Calculus

Relative Completeness

If $\models \{P\}\,c\,\{Q\}$, then $\vdash \{P\}\,c\,\{Q\}$ except for the weakening conditions.

- To show this, one constructs a so-called weakest precondition.

Weakest Precondition

Given a program c and an assertion P, the weakest precondition is an assertion W which

1. is a valid precondition: $\models \{W\}\,c\,\{P\}$
2. and is the weakest such: if $\models \{Q\}\,c\,\{P\}$, then $W \longrightarrow Q$.

- Question: is the weakest precondition unique?
  Only up to logical equivalence: if W_1 and W_2 are weakest preconditions, then $W_1 \longleftrightarrow W_2$.

Constructing the Weakest Precondition

- Consider the following simple program and its verification:

$\{X = x \wedge Y = y\}$
$\longleftrightarrow$
$\{Y = y \wedge X = x\}$
Z:= Y;
$\{Z = y \wedge X = x\}$
Y:= X;
$\{Z = y \wedge Y = x\}$
X:= Z;
$\{X = y \wedge Y = x\}$

- The idea is to construct the weakest precondition inductively.

Constructing the Weakest Precondition

- There are four straightforward cases:

$$\begin{aligned}
\mathrm{wp}(\textbf{skip}, P) &\stackrel{def}{=} P \\
\mathrm{wp}(X := e, P) &\stackrel{def}{=} P[e/X] \\
\mathrm{wp}(c_0; c_1, P) &\stackrel{def}{=} \mathrm{wp}(c_0, \mathrm{wp}(c_1, P)) \\
\mathrm{wp}(\textbf{if}\ b\ \{c_0\}\ \textbf{else}\ \{c_1\}, P) &\stackrel{def}{=} (b \wedge \mathrm{wp}(c_0, P)) \vee (\neg b \wedge \mathrm{wp}(c_1, P))
\end{aligned}$$

- The complicated one is iteration. This is not surprising, because iteration gives computational power (and makes our language Turing-complete). It can be given recursively:

$$\mathrm{wp}(\textbf{while}\ b\ \{c\}, P) \stackrel{def}{=} (\neg b \wedge P) \vee (b \wedge \mathrm{wp}(c, \mathrm{wp}(\textbf{while}\ b\ \{c\}, P)))$$

  A closed formula can be given using Turing's β-predicate, but it is unwieldy to write down.
- Hence, $\mathrm{wp}(c, P)$ is not an effective way to prove correctness.

Verfication Conditions: Annotated Programs

- Idea: invariants specified in the program by annotations.
- Arithmetic and Boolean Expressions (**AExp**, **BExp**) remain as they are.
- Annotated Statements (**ACom**)

$$\begin{aligned}
c ::= \ &\textbf{skip} \mid \textbf{Loc} := \textbf{AExp} \mid \textbf{assert}\ P \mid \textbf{if}\ b\ \{c_1\}\ \textbf{else}\ \{c_2\} \\
&\mid \textbf{while}\ b\ \textbf{inv}\ I\ \{c\} \mid c_1; c_2
\end{aligned}$$

Calculuation Verification Conditions

- For an annotated statement $c \in \mathbf{ACom}$ and an assertion P (the postcondition), we calculuate a set of verification conditions $\mathsf{vc}(c, P)$ and a precondition $\mathsf{pre}(c, P)$.
- The precondition is an auxiliary definition — it is mainly needed to compute the verification conditions.
- If we can prove the verification conditions, then $\mathsf{pre}(c, P)$ is a proper precondition, i.e. $\models \{\mathsf{pre}(c, P)\}\, c\, \{P\}$.

Calculating Verification Conditions

$$
\begin{aligned}
\mathsf{pre}(\mathbf{skip}, P) &\stackrel{def}{=} P\\
\mathsf{pre}(X := e, P) &\stackrel{def}{=} P[e/X]\\
\mathsf{pre}(c_0; c_1, P) &\stackrel{def}{=} \mathsf{pre}(c_0, \mathsf{pre}(c_1, P))\\
\mathsf{pre}(\mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, P) &\stackrel{def}{=} (b \wedge \mathsf{pre}(c_0, P)) \vee (\neg b \wedge \mathsf{pre}(c_1, P))\\
\mathsf{pre}(\mathbf{assert}\ \ Q, P) &\stackrel{def}{=} Q\\
\mathsf{pre}(\mathbf{while}\ \ b\ \ \mathbf{inv}\ \ I\ \{c\}, P) &\stackrel{def}{=} I\\
\\
\mathsf{vc}(\mathbf{skip}, P) &\stackrel{def}{=} \emptyset\\
\mathsf{vc}(X := e, P) &\stackrel{def}{=} \emptyset\\
\mathsf{vc}(c_0; c_1, P) &\stackrel{def}{=} \mathsf{vc}(c_0, \mathsf{pre}(c_1, P)) \cup \mathsf{vc}(c_1, P)\\
\mathsf{vc}(\mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, P) &\stackrel{def}{=} \mathsf{vc}(c_0, P) \cup \mathsf{vc}(c_1, P)\\
\mathsf{vc}(\mathbf{assert}\ \ Q, P) &\stackrel{def}{=} \{Q \longrightarrow P\}\\
\mathsf{vc}(\mathbf{while}\ \ b\ \ \mathbf{inv}\ \ I\ \{c\}, P) &\stackrel{def}{=} \mathsf{vc}(c, I)\ \cup \{I \wedge b \longrightarrow \mathsf{pre}(c, I)\}\\
&\qquad\qquad \cup \{I \wedge \neg b \longrightarrow P\}\\
\mathsf{vc}(\{P\}\, c\, \{Q\}) &\stackrel{def}{=} \{P \longrightarrow \mathsf{pre}(c, Q)\} \cup \mathsf{vc}(c, Q)
\end{aligned}
$$

Correctness of the VC Calculus

Correctness of the VC Calculus

For a annotated program c and an assertion P:

$$\mathsf{vc}(c, P) \implies \{\mathsf{pre}(c, P)\}\, c\, \{P\}$$

- Proof: By induction on c.

Example: Faculty

Let *Fac* be the annotated faculty program:

```
{0 ≤ N}
P := 1;
C := 1;
while C ≤ N inv {P = (C − 1)! ∧ C − 1 ≤ N} {
  P := P * C;
  C := C + 1
}
{P = N!}
```

$$
\begin{aligned}
&\mathsf{vc}(Fac) =\\
&\quad \{\ 0 \leq N \longrightarrow 1 = 0! \wedge 0 \leq N,\\
&\qquad P = (C-1)! \wedge C - 1 \leq N \wedge C \leq N \longrightarrow P \times C = C! \wedge C \leq N,\\
&\qquad P = (C-1)! \wedge C - 1 \leq N \wedge \neg(C \leq N) \longrightarrow P = N!\ \}
\end{aligned}
$$

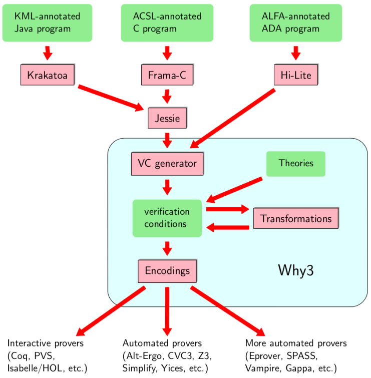
The Framing Problem

- One problem with the simple definition from above is that we need to specify which variables stay the same (framing problem).
  - Essentially, when going into a loop we use lose all information of the current precondition, as it is replaced by the loop invariant.
  - This does not occur in the faculty example, as all program variables are changed.
- Instead of having to write this down every time, it is more useful to modify the logic, such that we specify which variables are modified, and assume the rest stays untouched.
- Sketch of definition: We say $\models \{P, X\}\, c\, \{Q\}$ is a Hoare-Triple with modification set X if for all states σ which satisfy P if c terminates in a state σ', then σ' satisfies Q, and if $\sigma(x) \neq \sigma'(x)$ then $x \in X$.

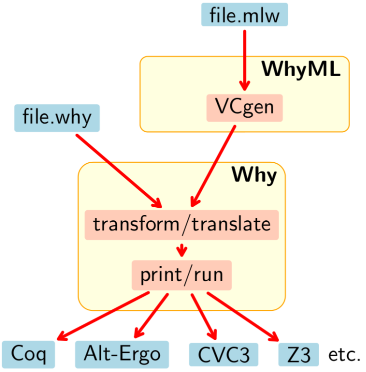
Verification Condition Generation Tools

- The Why3 toolset (`http://why3.lri.fr`)
  - The Why3 verification condition generator
  - Plug-ins for different provers
  - Front-ends for different languages: C (Frama-C), Java (Krakatoa)
- The Boogie VCG (`http://research.microsoft.com/en-us/projects/boogie/`)
- The VCC Tool (built on top of Boogie)
  - Verification of C programs
  - Used in German Verisoft XT project to verify Microsoft Hyper-V hypervisor

Why3 Overview: Toolset


KML-annotated Java program
ACSL-annotated C program
ALFA-annotated ADA program
Krakatoa
Frama-C
Hi-Lite
Jessie
VC generator
Theories
verification conditions
Transformations
Encodings
Why3
Interactive provers (Coq, PVS, Isabelle/HOL, etc.)
Automated provers (Alt-Ergo, CVC3, Z3, Simplify, Yices, etc.)
More automated provers (Eprover, SPASS, Vampire, Gappa, etc.)

Why3 Overview: VCG



Why3 Example: Faculty (in WhyML)

```
let fac(n: int): int
  requires { n >= 0 }
  ensures  { result = fact(n) } =
  let p = ref 0 in
  let c = ref 0 in
  p := 1;
  c := 1;
  while !c <= n do
    invariant { !p= fact(!c-1) /\ !c-1 <= n }
    variant { n- !c }
    p:= !p* !c;
    c:= !c+ 1
    done;
 !p
```

Why3 Example: Generated VC for Faculty

```
goal WP_parameter_fac :
 forall n:int.
  n >= 0 ->
   (forall p:int.
     p = 1 ->
      (forall c:int.
        c = 1 ->
         (p = fact (c - 1) /\ (c - 1) <= n) /\
          (forall c1:int, p1:int.
            p1 = fact (c1 - 1) /\ (c1 - 1) <= n ->
             (if c1 <= n then forall p2:int.
                               p2 = (p1 * c1) ->
                                (forall c2:int.
                                  c2 = (c1 + 1) ->
                                   (p2 = fact (c2 - 1) /\
                                    (c2 - 1) <= n) /\
                                    0 <= (n - c1) /\
                                    (n - c2) < (n - c1))
             else p1 = fact n))))
```

Summary

- Starting from the relative completeness of the Floyd-Hoare calculus, we devised a Verification Condition Generation calculus which makes program verification viable.
- Verification Condition Generation reduces an annotated program to a set of logical properties.
- We need to annotate preconditions, postconditions and invariants.
- Tools which support this sort of reasoning include Why3 and Boogie. They come with front-ends for real programming languages, such as C, Java, C#, and Ada.
- 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).

Systeme Hoher Sicherheit und Qualität
Universität Bremen WS 2015/2016

Lecture 12 (18.01.2016)

Semantics of Programming Languages

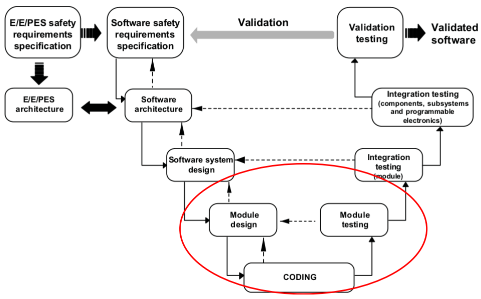
Christoph Lüth   Jan Peleska   Dieter Hutter

Universität Bremen

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 SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09: Program Analysis
- 10: Foundations of Software Verification
- 11: Verification Condition Generation
- 12: Semantics of Programming Languages
- 13: Model-Checking
- 14: Conclusions and Outlook

Semantics in the Development Process


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

Semantics — what does that mean?

> " Semantics: The meaning of words, phrases or systems. "
>
> — *Oxford Learner's Dictionaries*

- In mathematics and computer science, semantics is giving a meaning in mathematical terms. It can be contrasted with syntax, which specifies the notation.
- Here, we will talk about the meaning of programs. Their syntax is described by formal grammars, and their semantics in terms of mathematical structures.
- Why would we want to do that?

Why Semantics?

Semantics describes the meaning of a program (written in a programming language) in mathematical precise and unambiguous way. Here are three reasons why this is a good idea:

- It lets us write better compilers. In particular, it makes the language independent of a particular compiler implementation.
- If we know the precise meaning of a program, we know when it should produce a result and when not. In particular, we know which situations the program should avoid.
- Finally, it lets us reason about program correctness.

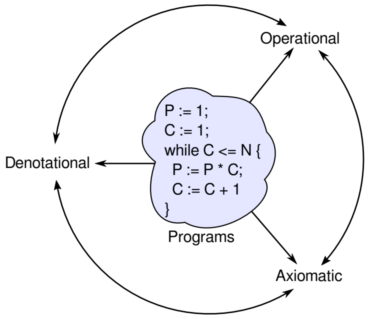
Empfohlene Literatur: Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.

Semantics of Programming Languages

Historically, there are three ways to write down the semantics of a programming language:

- Operational semantics describes the meaning of a program by specifying how it executes on an abstract machine.
- Denotational semantics assigns each program to a partial function on the system state.
- Axiomatic semantics tries to give a meaning of a programming construct by giving proof rules. A prominent example of this is the Floyd-Hoare logic of previous lectures.

A Tale of Three Semantics



```
P := 1;
C := 1;
while C <= N {
  P := P * C;
  C := C + 1
}
```

- Each semantics should be considered a view of the program.
- Importantly, all semantics should be equivalent. This means we have to put them into relation with each other, and show that they agree. Doing so is an important sanity check for the semantics.
- In the particular case of axiomatic semantics (Floyd-Hoare logic), it is the question of correctness of the rules.

Operational Semantics

- Evaluation is directed by the syntax.
- We inductively define relations $\rightarrow$ between configurations (a command or expression together with a state) to an integer, boolean or a state:

$$\begin{aligned}\rightarrow_A &\subseteq (\mathbf{AExp}, \Sigma) \times \mathbb{Z}\\ \rightarrow_B &\subseteq (\mathbf{BExp}, \Sigma) \times \mathit{Bool}\\ \rightarrow_S &\subseteq (\mathbf{Com}, \Sigma) \times \Sigma\end{aligned}$$

where the system state is defined as as

$$\Sigma \overset{def}{=} \mathbf{Loc} \rightharpoonup \mathbb{Z}$$

- $(p, \sigma) \rightarrow_S \sigma'$ means that evaluating the program p in state σ results in state σ', and $(a, \sigma) \rightarrow_A i$ means evaluating expression a in state σ results in integer value i.

Structural Operational Semantics

- The evaluation relation is defined by rules of the form

$$\frac{\langle a, \sigma\rangle \to_A i}{\langle \mathtt{p}\ a_1, \sigma\rangle \to_A f(i)}$$

for each programming language construct p. This means that when the argument a of the construct has been evaluated, we can evaluate the whole expression.

  - This is called structural operational semantics.

- Note that this does not specify an evaluation strategy.
- This evaluation is partial and can be non-deterministic.

IMP: Arithmetic Expressions

Numbers: $\langle n, \sigma\rangle \to_A n$

Variables: $\langle \mathtt{X}, \sigma\rangle \to_A \sigma(\mathtt{X})$

Addition: $$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 + a_1, \sigma\rangle \to_A n + m}$$

Subtraction: $$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 \texttt{ - } a_1, \sigma\rangle \to_A n - m}$$

Multiplication: $$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 \texttt{ * } a_1, \sigma\rangle \to_A n \cdot m}$$

IMP: Boolean Expressions (Constants, Relations)

$$\langle \mathbf{true}, \sigma\rangle \to_B \mathit{True} \qquad \langle \mathbf{false}, \sigma\rangle \to \mathit{False}$$

$$\frac{\langle b, \sigma\rangle \to_B \mathit{False}}{\langle \mathbf{not}\ b, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b, \sigma\rangle \to_B \mathit{True}}{\langle \mathbf{not}\ b, \sigma\rangle \to_B \mathit{False}}$$

$$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 = a_1, \sigma\rangle \to_B \mathit{True}}\ n = m \qquad \frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 = a_1, \sigma\rangle \to_B \mathit{False}}\ n \neq m$$

$$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 < a_1, \sigma\rangle \to_B \mathit{True}}\ n < m \qquad \frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 < a_1, \sigma\rangle \to_B \mathit{False}}\ n \geq m$$

IMP: Boolean Expressions (Operators)

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{True}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{False}}$$

IMP: Boolean Expressions (Operators — Variation)

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}} \qquad \frac{\langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{True}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{False}}$$

What is the difference?

Operational Semantics of IMP: Statements

$$\langle \mathbf{skip}, \sigma\rangle \to_S \sigma$$

$$\frac{\langle a, \sigma\rangle \to_S n}{\langle X := a, \sigma\rangle \to_S \sigma[n/X]} \qquad \frac{\langle c_0, \sigma\rangle \to_S \tau \quad \langle c_1, \tau\rangle \to_S \tau'}{\langle c_0; c_1, \sigma\rangle \to_S \tau'}$$

$$\frac{\langle b, \sigma\rangle \to_B \mathit{True} \quad \langle c_0, \sigma\rangle \to_S \tau}{\langle \mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, \sigma\rangle \to_S \tau} \qquad \frac{\langle b, \sigma\rangle \to \mathit{False} \quad \langle c_1, \sigma\rangle \to_S \tau}{\langle \mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, \sigma\rangle \to_S \tau}$$

$$\frac{\langle b, \sigma\rangle \to_B \mathit{False}}{\langle \mathbf{while}\ b\ \{c\}, \sigma\rangle \to_S \sigma}$$

$$\frac{\langle b, \sigma\rangle \to_B \mathit{True} \quad \langle c, \sigma\rangle \to_S \tau' \quad \langle \mathbf{while}\ b\ \{c\}, \tau'\rangle \to_S \tau}{\langle \mathbf{while}\ b\ \{c\}, \sigma\rangle \to_S \tau}$$

Why Denotational Semantics?

- Denotational semantics takes an abstract view of program: if $c_1 \sim c_2$, they have the "same meaning".
- This allows us, for example, to compare programs in different programming languages.
- It also accommodates reasoning about programs far better than operational semantics. In particular, we can prove the correctness of the Floyd-Hoare rules.
- It gives us compositionality and referential transparency, mapping programming language construct p to denotation ϕ:

$$\mathcal{D}[\![\mathrm{p}(e_1, \ldots, e_n)]\!] = \phi(\mathcal{D}[\![e_1]\!], \ldots, \mathcal{D}[\![e_n]\!])$$

Denotational Semantics

- Programs are denoted by functions on states $\Sigma = \mathbf{Loc} \rightharpoonup \mathbb{Z}$.
- Semantic functions assign a meaning to statements and expressions:

Arithmetic expressions:	$\mathcal{E} : \mathbf{AExp} \to (\Sigma \to \mathbb{Z})$
Boolean expressions:	$\mathcal{B} : \mathbf{BExp} \to (\Sigma \to \mathit{Bool})$
Statements:	$\mathcal{D} : \mathbf{Com} \to (\Sigma \rightharpoonup \Sigma)$

- Note the meaning of a program p is a partial function, reflecting the fact that programs may not terminate.
  - Our expressions always do, but that is because our language is quite simple.

Denotational Semantics of IMP: Arithmetic Expressions

$$\begin{aligned}
\mathcal{E}[\![n]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.n\\
\mathcal{E}[\![X]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\sigma(X)\\
\mathcal{E}[\![a_0 + a_1]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.(\mathcal{E}[\![a_0]\!]\sigma + \mathcal{E}[\![a_1]\!]\sigma)\\
\mathcal{E}[\![a_0 \text{ - } a_1]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.(\mathcal{E}[\![a_0]\!]\sigma - \mathcal{E}[\![a_1]\!]\sigma)\\
\mathcal{E}[\![a_0 \text{ * } a_1]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.(\mathcal{E}[\![a_0]\!]\sigma \cdot \mathcal{E}[\![a_1]\!]\sigma)
\end{aligned}$$

Denotational Semantics of IMP: Boolean Expressions

$$\begin{aligned}
\mathcal{B}[\![\textbf{true}]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\mathit{True}\\
\mathcal{B}[\![\textbf{false}]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\mathit{False}\\
\mathcal{B}[\![\textbf{not}\ b]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\neg\mathcal{B}[\![b]\!]\sigma\\
\mathcal{B}[\![a_0 = a_1]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\begin{cases}\mathit{True} & \mathcal{E}[\![a_0]\!]\sigma = \mathcal{E}[\![a_1]\!]\sigma\\ \mathit{False} & \mathcal{E}[\![a_0]\!]\sigma \neq \mathcal{E}[\![a_1]\!]\sigma\end{cases}\\
\mathcal{B}[\![a_0 < a_1]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\begin{cases}\mathit{True} & \mathcal{E}[\![a_0]\!]\sigma < \mathcal{E}[\![a_1]\!]\sigma\\ \mathit{False} & \mathcal{E}[\![a_0]\!]\sigma \geq \mathcal{E}[\![a_1]\!]\sigma\end{cases}\\
\mathcal{B}[\![b_0 \text{ and } b_1]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\mathcal{B}[\![b_0]\!]\sigma \wedge \mathcal{B}[\![b_1]\!]\sigma\\
\mathcal{B}[\![b_0 \text{ or } b_1]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\mathcal{B}[\![b_0]\!]\sigma \vee \mathcal{B}[\![b_1]\!]\sigma
\end{aligned}$$

Denotational Semantics of IMP: Statements

The simple part:

$$\begin{aligned}
\mathcal{D}[\![\textbf{skip}]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\,\sigma\\
\mathcal{D}[\![X := a]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\,\sigma[\mathcal{E}[\![a]\!]\sigma/X]\\
\mathcal{D}[\![c_0; c_1]\!] &\overset{def}{=} \mathcal{D}[\![c_1]\!] \circ \mathcal{D}[\![c_0]\!]\\
\mathcal{D}[\![\textbf{if}\ \ b\ \{c_0\}\ \ \textbf{else}\ \{c_1\}]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\begin{cases}\mathcal{D}[\![c_0]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{True}\\ \mathcal{D}[\![c_1]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{False}\end{cases}
\end{aligned}$$

The hard part:

$$\mathcal{D}[\![\textbf{while}\ \ b\ \{c\}]\!] = \lambda\sigma \in \Sigma.\begin{cases}\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{False}\\ (\mathcal{D}[\![\textbf{while}\ \ b\ \{c\}]\!] \circ \mathcal{D}[\![c]\!])\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{True}\end{cases}$$

This recursive definition is not constructive — it does not tell us how to construct the function. Worse, it is unclear it even exists in general.

Partial Orders and Least Upper Bounds

To construct fixpoints of the form $x = f(x)$, we need the theory of complete partial orders (cpo's).

Definition (Partial Order)

Given a set X, a partial order $\sqsubseteq\ \subseteq X \times X$ is

(i) transitive: if $x \sqsubseteq y, y \sqsubseteq z$, then $x \sqsubseteq z$

(ii) reflexive: $x \sqsubseteq x$

(iii) anti-symmetric: if $x \sqsubseteq y, y \sqsubseteq x$ then $x = y$

Definition (Least Upper Bound)

For $Y \subseteq X$, the least upper bound $\bigsqcup Y \in X$ is:

(i) $\forall y \in Y.\, y \sqsubseteq \bigsqcup Y$

(ii) for any $z \in X$ such that $\forall y \in Y.\, y \sqsubseteq z$, we have $\bigsqcup Y \sqsubseteq z$

Complete Partial Orders

Definition (Complete Partial Order)

A partial order $\sqsubseteq$ is complete (a cpo) if any ω-chain $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq x_4 \ldots = \{x_i \mid i \in \omega\}$ has a least upper bound $\bigsqcup_{i\in\omega} x_i \in X$.

A cpo is called pointed (pcpo), if there is a smallest element $\perp \in X$. (Note some authors assume all cpos to be pointed.)

Definition (Continuous Function)

Given cpos $(X, \sqsubseteq)$ and $(Y, \leq)$. A function $f : X \to Y$ is

(i) monotone, if $x \sqsubseteq y$ then $f(x) \leq f(y)$

(ii) continuous, if monotone and $f(\bigsqcup_{i\in\omega} x_i) = \bigsqcup_{i\in\omega} f(x_i)$

Fixpoints

Theorem (Each continuous function has a least fixpoint)

Let $(X, \sqsubseteq)$ be a pcpo, and $f : X \to X$ continuous, then f has a least fixpoint $fix(f)$,given as

$$fix(f) = \bigsqcup_{n\in\omega} f^n(\perp)$$

- In our case, the state Σ is made into a pcpo $\Sigma_\perp$ by 'adjoining' a new element $\perp$, ordered as $\perp \sqsubseteq \sigma$.
- This models partial functions: $\Sigma \rightharpoonup \Sigma \cong \Sigma \to \Sigma_\perp$
- $\Sigma \to \Sigma_\perp$ ist a pcpo, ordered as

$$f \sqsubseteq g \longleftrightarrow \forall x. f(x) \sqsubseteq g(x)$$

Concretely, $f \sqsubseteq g$ means that f is defined on fewer states than g.

Denotational Semantics of IMP: Statements

$$\begin{aligned}
\mathcal{D}[\![\textbf{skip}]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\,\sigma\\
\mathcal{D}[\![X := a]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\,\sigma[\mathcal{E}[\![a]\!]\sigma/X]\\
\mathcal{D}[\![c_0; c_1]\!] &\overset{def}{=} \mathcal{D}[\![c_1]\!] \circ \mathcal{D}[\![c_0]\!]\\
\mathcal{D}[\![\textbf{if}\ \ b\ \{c_0\}\ \ \textbf{else}\ \{c_1\}]\!] &\overset{def}{=} \lambda\sigma \in \Sigma.\begin{cases}\mathcal{D}[\![c_0]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{True}\\ \mathcal{D}[\![c_1]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{False}\end{cases}\\
\mathcal{D}[\![\textbf{while}\ \ b\ \{c\}]\!] &\overset{def}{=} fix(\Gamma)\\
\text{where } \Gamma(\phi) &\overset{def}{=} \lambda\sigma \in \Sigma.\begin{cases}\phi \circ \mathcal{D}[\![c]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{True}\\ \sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{False}\end{cases}
\end{aligned}$$

Equivalence of Semantics

Lemma

(i) *For $a \in$ **Aexp**, $n \in \mathbb{N}$, $\mathcal{E}[\![a]\!]\sigma = n$ iff $\langle a, \sigma\rangle \to_A n$*

(ii) *For $b \in$ **BExp**, $t \in Bool$, $\mathcal{B}[\![b]\!]\sigma = t$ iff $\langle b, \sigma\rangle \to_B t$*

Proof: Structural Induction on a and b. □

Lemma

*For $c \in$ **Com**, if $\langle c, \sigma\rangle \to_S \sigma'$ then $\mathcal{D}[\![c]\!]\sigma = \sigma'$*

Proof: Induction over deriviation of $\langle c, \sigma\rangle \to_S \sigma'$. □

Theorem (Equivalence of Semantics)

*For $c \in$ **Com**, and $\sigma, \sigma' \in \Sigma$,*

$$\langle c, \sigma\rangle \to_S \sigma' \textit{ iff } \mathcal{D}[\![c]\!]\sigma = \sigma'$$

The proof of this theorem requires a technique called fixpoint induction which we will not go into detail about here.

Correctness of Floyd-Hoare Rules

Denotational semantics allows us to prove the correctness of the Floyd-Hoare rules.

- We extend the boolean semantic functions $\mathcal{E}$ and $\mathcal{B}$ to **AExpv** and **BExpv**, respectively.
- We can then define the validity of a Hoare triple in terms of denotations:

$$\models \{P\}\, c\, \{Q\} \text{ iff } \forall \sigma.\, \mathcal{B}[\![P]\!]\sigma \wedge \mathcal{D}[\![c]\!]\sigma \neq \bot \longrightarrow \mathcal{B}[\![Q]\!](\mathcal{D}[\![c]\!]\sigma)$$

- We can now show the rules preserve validity, *i.e.* if the preconditions are valid Hoare triples, then so is the conclusion.

Remarks

- Our language and semantics is quite simple-minded. We have not take into account:
  - undefined expressions (such as division by 0 or accessing an undefined variable),
  - side effects in expressions,
  - declaration of variables,
  - pointers, references, pointer arithmetic,
  - input/output (what is the semantic model?), or
  - concurrency.
- However, there are formal semantics for languages such as StandardML, C, or Java, although most of them concentrate on some aspect of the language (*e.g.* Java concurrency is not very well defined in the standard). Only StandardML has a language standard which is written as an operational semantics.

Conclusion

- Programming semantics come in three flavours: operational, denotational, axiomatic.
- Each of these has their own use case:
  - Operational semantics gives details about evaluation of programs, and is good for implementing the programming language.
  - Denotational semantics is abstract and good for high-level reasoning (*e.g.* correctness of program logics or tools).
  - Axiomatic semantics is about program logics, and reasoning about programs.
- Denotational semantics needs the mathematical toolkit of cpos to construct fixpoints.

Systeme Hoher Sicherheit und Qualität
Universität Bremen WS 2015/2016

Lecture 13 (25.01.2016)

Modelchecking with LTL and CTL

Christoph Lüth   Jan Peleska   Dieter Hutter

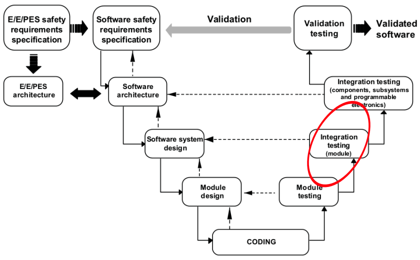
Organisatorisches

- Evaluation: auf der stud.ip-Seite (unter Lehrevaluation)
- Prüfungen & Fachgespräche:
  - KW 7 (15./16. Februar), oder
  - 02. Februar (letzte Semesterwoche, zum Übungstermin).

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 SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09: Program Analysis
- 10: Foundations of Software Verification
- 11: Verification Condition Generation
- 12: Semantics of Programming Languages
- 13: Model-Checking
- 14: Conclusions and Outlook

Modelchecking in the Development Process


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING

- Model-checking proves properties of abstractions of the system.
- Thus, it scales also to higher levels of the development process

Introduction

- In the last lectures, we were verifying program properties with the Floyd-Hoare calculus and related approaches. Program verification was reduced to a deductive problem by translating the program into logic (specifically, state change becomes substitution).
- Model-checking takes a different approach: instead of directly working with the program, we work with an abstraction of the system (a model). Because we build abstractions, this approach is also applicable in the higher verification levels.
- But what are the properties we want to express? How do we express them, and how do we prove them?

The Model-Checking Problem

The Basic Question

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

$$\mathcal{M} \models \phi$$

- What is $\mathcal{M}$? Finite state machines
- What is ϕ? Temporal logic
- How to prove it? Enumerating states — model checking
  - The basic problem: state explosion

Finite State Machines

Finite State Machine (FSM)

A FSM is given by $\mathcal{M} = \langle \Sigma, \rightarrow \rangle$ where

- Σ is a finite set of states, and
- $\rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation, such that $\rightarrow$ is left-total:

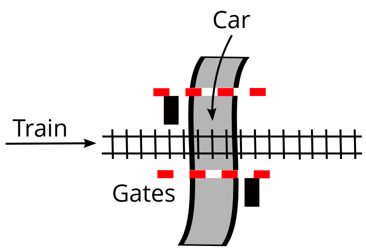
$$\forall s \in \Sigma. \exists s' \in \Sigma. s \rightarrow s'$$

- Many variations of this definition exists, e.g. sometimes we have state variables or labelled transitions.
- 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.

The Railway Crossing

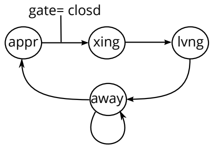
Source: Wikipedia

The Railway Crossing — Abstraction

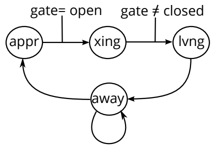


The Railway Crossing — Model

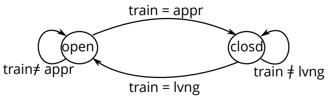
States of the train:



States of the car:



States of the gate:



The FSM

- The states here are a map from variables *Car*, *Train*, *Gate* to the domains

$$\begin{aligned}\Sigma_{Car} &= \{appr, xing, lvng, away\}\\ \Sigma_{Train} &= \{appr, xing, lvng, away\}\\ \Sigma_{Gate} &= \{open, clsd\}\end{aligned}$$

  or alternatively, a three-tuple $S \in \Sigma = \Sigma_{Car} \times \Sigma_{Train} \times \Sigma_{Gate}$.

- The transition relation is given by e.g.

$$\begin{aligned}&\langle away, open, away\rangle \to \langle appr, open, away\rangle\\ &\langle appr, open, away\rangle \to \langle xing, open, away\rangle\\ &\ldots\end{aligned}$$

Railway Crossing — Safety Properties

- Now we want to express safety (or security) properties, such as the following:
  - Cars and trains never cross at the same time.
  - The car can always leave the crossing
  - Approaching trains may eventually cross.
  - There are cars crossing the tracks.
- We distinguish safety properties from liveness properties:
  - Safety: something bad never happens.
  - Liveness: something good will (eventually) happen.
- To express these properties, we need to talk about sequences of states in an FSM.

Linear Temporal Logic (LTL) and Paths

- LTL allows us to talk about paths in a FSM, where a path is a sequence of states connected by the transition relation.
- We first define the syntax of formula,
- then what it means for a path to satisfy the formula, and
- from that we derive the notion of a model for an LTL formula.

Paths

Given a FSM $\mathcal{M} = \langle \Sigma, \to \rangle$, a path in $\mathcal{M}$ is an (infinite) sequence $\langle s_1, s_2, s_3, \ldots \rangle$ such that $s_i \in \Sigma$ and $s_i \to s_{i+1}$ for all i.

- For a path $p = \langle s_1, s_2, s_3, \ldots \rangle$, we write p_i for s_i (selection) and p^i for $\langle s_i, s_{i+1}, \ldots \rangle$ (the suffix starting at i).

Linear Temporal Logic (LTL)

$$\begin{array}{rll}\phi ::= & \top \mid \bot \mid p & \text{— True, false, atomic}\\ \mid & \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \longrightarrow \phi_2 & \text{— Propositional formulae}\\ \mid & \mathsf{X}\,\phi & \text{— Next state}\\ \mid & \Diamond\phi & \text{— Some Future State}\\ \mid & \Box\phi & \text{— All future states (Globally)}\\ \mid & \phi_1\ U\ \phi_2 & \text{— Until}\end{array}$$

- Operator precedence: Unary operators; then U; then $\wedge$, $\vee$; then $\longrightarrow$.
- An atomic formula p above denotes a state predicate. Note that different FSMs have different states, so the notion of whether an atomic formula is satisfied depends on the FSM in question. A different (but equivalent) approach is to label states with atomic propositions.
- From these, we can define other operators, such as $\phi\ R\ \psi$ (release) or $\phi\ W\ \psi$ (weak until).

Satifsaction and Models of LTL

Given a path p and an LTL formula ϕ, the satisfaction relation $p \models \phi$ is defined inductively as follows:

$$\begin{array}{lll}p \models True & & p \models \phi \wedge \psi \text{ iff } p \models \phi \text{ and } p \models \psi\\ p \not\models False & & p \models \phi \vee \psi \text{ iff } p \models \phi \text{ or } p \models \psi\\ p \models p \text{ iff } p(p_1) & & p \models \phi \longrightarrow \psi \text{ iff whenever } p \models \phi \text{ then } p \models \psi\\ p \models \neg\phi \text{ iff } p \not\models \phi & & \end{array}$$

$$\begin{array}{l}p \models \mathsf{X}\,\phi \text{ iff } p^2 \models \phi\\ p \models \Box\phi \text{ iff for all } i\text{, we have } p^i \models \phi\\ p \models \Diamond\phi \text{ iff there is } i \text{ such that } p^i \models \phi\\ p \models \phi\ U\ \psi \text{ iff there is } i\ p^i \models \psi \text{ and for all } j = 1, \ldots, i-1,\ \ p^j \models \phi\end{array}$$

Models of LTL formulae

A FSM $\mathcal{M}$ satisfies an LTL formula ϕ, $\mathcal{M} \models \phi$, iff every path p in $\mathcal{M}$ satisfies ϕ.

The Railway Crossing

- Cars and trains never cross at the same time.

$$\Box\neg(car = xing \wedge train = xing)$$

- A car can always leave the crossing:

$$\Box(car = xing \longrightarrow \Diamond(car = lvng))$$

- Approaching trains may eventually cross:

$$\Box(train = appr \longrightarrow \Diamond(train = xing))$$

- There are cars crossing the tracks:

  $\Diamond(car = xing)$ means something else!

  - Can not express this in LTL!

Computational Tree Logic (CTL)

- LTL does not allow us the quantify over paths, e.g. assert the existance 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 wether $\neg\phi$ 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 an extension of LTL 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 FSM.

CTL Formulae

$$
\begin{array}{rcll}
\phi & ::= & \top \mid \bot \mid p & \text{— True, false, atomic} \\
 & \mid & \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \longrightarrow \phi_2 & \text{— Propositional formulae} \\
 & \mid & \mathrm{AX}\,\phi \mid \mathrm{EX}\,\phi & \text{— All or some next state} \\
 & \mid & \mathrm{AF}\,\phi \mid \mathrm{EF}\,\phi & \text{— All or some future states} \\
 & \mid & \mathrm{AG}\,\phi \mid \mathrm{EG}\,\phi & \text{— All or some global future} \\
 & \mid & \mathrm{A}[\phi_1\ U\ \phi_2] \mid \mathrm{E}[\phi_1\ U\ \phi_2] & \text{— Until all or some}
\end{array}
$$

Satifsfaction

- Note that CTL formulae can be considered to be a LTL formulae with a 'modality' (A or E) added on top of each temporal operator.
- Generally speaking, the A modality says the temporal operator holds for all paths, and the E modality says the temporal operator only holds for all least one path.
  - Of course, that strictly speaking is not true, because the arguments of the temporal operators are in turn CTL forumulae, so we need recursion.
- This all explains why we do not define a satisfaction for a single path p, but satisfaction with respect to a specific state in an FSM.

Satisfaction for CTL

Given an FSM $\mathcal{M} = \langle \Sigma, \rightarrow \rangle$, $s \in \Sigma$ and a CTL formula ϕ, then $\mathcal{M}, s \models \phi$ is defined inductively as follows:

$$
\begin{array}{lcl}
\mathcal{M}, s & \models & \mathit{True} \\
\mathcal{M}, s & \not\models & \mathit{False} \\
\mathcal{M}, s & \models & p \text{ iff } p(s) \\
\mathcal{M}, s & \models & \phi \wedge \psi \text{ iff } \mathcal{M}, s \models \phi \text{ and } \mathcal{M}, s \models \psi \\
\mathcal{M}, s & \models & \phi \vee \psi \text{ iff } \mathcal{M}, s \models \phi \text{ or } \mathcal{M}, s \models \psi \\
\mathcal{M}, s & \models & \phi \longrightarrow \psi \text{ iff whenever } \mathcal{M}, s \models \phi \text{ then } \mathcal{M}, s \models \psi \\
\quad\dots & &
\end{array}
$$

Satisfaction for CTL (c'ed)

Given an FSM $\mathcal{M} = \langle \Sigma, \rightarrow \rangle$, $s \in \Sigma$ and a CTL formula ϕ, then $\mathcal{M}, s \models \phi$ is defined inductively as follows:

$$
\begin{array}{lcl}
\quad\dots & & \\
\mathcal{M}, s & \models & \mathrm{AX}\,\phi \text{ iff for all } s_1 \text{ with } s \rightarrow s_1\text{, we have } \mathcal{M}, s_1 \models \phi \\
\mathcal{M}, s & \models & \mathrm{EX}\,\phi \text{ iff for some } s_1 \text{ with } s \rightarrow s_1\text{, we have } \mathcal{M}, s_1 \models \phi \\
\mathcal{M}, s & \models & \mathrm{AG}\,\phi \text{ iff for all paths } p \text{ with } p_1 = s, \\
 & & \qquad \text{we have } \mathcal{M}, p_i \models \phi \text{ for all } i \geq 2 \\
\mathcal{M}, s & \models & \mathrm{EG}\,\phi \text{ iff there is a path } p \text{ with } p_1 = s \text{ and} \\
 & & \qquad \text{we have } \mathcal{M}, p_i \models \phi \text{ for all } i \geq 2 \\
\mathcal{M}, s & \models & \mathrm{AF}\,\phi \text{ iff for all paths } p \text{ with } p_1 = s \\
 & & \qquad \text{we have } \mathcal{M}, p_i \models \phi \text{ for some } i \\
\mathcal{M}, s & \models & \mathrm{EF}\,\phi \text{ iff there is a path } p \text{ with } p_1 = s \text{ and} \\
 & & \qquad \text{we have; } \mathcal{M}, p_i \models \phi \text{ for some } i \\
\mathcal{M}, s & \models & \mathrm{A}[\phi\ U\ \psi] \text{ iff for all paths } p \text{ with } p_1 = s\text{, there is } i \\
 & & \qquad\quad \text{with } \mathcal{M}, p_i \models \psi \text{ and for all } j < i,\ \mathcal{M}, p_j \models \phi \\
\mathcal{M}, s & \models & \mathrm{E}[\phi\ U\ \psi] \text{ iff there is a path } p \text{ with } p_1 = s \text{ and there is } i \\
 & & \qquad\quad \text{with } \mathcal{M}, p_i \models \psi \text{ and for all } j < i,\ \mathcal{M}, p_j \models \phi
\end{array}
$$

Patterns of Specification

- Something bad (p) cannot happen: AG$\,\neg p$
- p occurs infinitly often: AG(AF p)
- p occurs eventually: AF p
- In the future, p will hold eventually forever: AF AG p
- Whenever p will hold in the future, q will hold eventually: AG($p \longrightarrow$ AF q)
- In all states, p is always possible: AG(EF p)

LTL and CTL

- We have seen that CTL is more expressive than LTL, but (surprisingly), there are properties which we can formalise in LTL but not in CTL!
- Example: all paths which have a p along them also have a q along them.
- LTL: $\Diamond p \longrightarrow \Diamond q$
- CTL: Not AF $p \longrightarrow$ AF q (would mean: if all paths have p, then all paths have q), neither AG($p \longrightarrow$ AF q) (which means: if there is a p, it will be followed by a q).
- The logic CTL^* combines both LTL and CTL (but we will not consider it further here).

State Explosion and Complexity

- The basic problem of model checking is state explosion.
- Even our small railway crossing has $|\Sigma| = |\Sigma_{Car} \times \Sigma_{Train} \times \Sigma_{Gate}| = |\Sigma_{Car}| \cdot |\Sigma_{Train}| \cdot |\Sigma_{Gate}| = 4 \cdot 4 \cdot 2 = 32$ states. Add one integer variable with 2^{32} states, and this gets intractable.
- Theoretically, there is not much hope. The basic problem of deciding wether a particular formula holds is known as the satisfiability problem, and for the temporal logics we have seen, its complexity is as follows:
  - LTL without U is *NP*-complete.
  - LTL is *PSPACE*-complete.
  - CTL is *EXPTIME*-complete.
- The good news is that at least it is decidable. Practically, state abstraction is the key technique. E.g. instead of considering all possible integer values, consider only wether i is zero or larger than zero.

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.
- 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. Use of abstraction and state compression techniques make model-checking bearable.
- Tomorrow: practical experiments with model-checkers (NuSMV and/or Spin)

Systeme Hoher Sicherheit und Qualität
Universität Bremen WS 2015/2016

Lecture 14 (01.02.2016)

Concluding Remarks

Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen

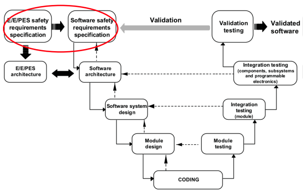
Where are we?

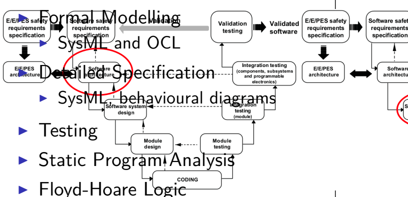
- 01: Concepts of Quality
- 02: Legal Requirements: Norms and Standards
- 03: The Software Development Process
- 04: Hazard Analysis
- 05: High-Level Design with SysML
- 06: Formal Modelling with SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09: Program Analysis
- 10: Foundations of Software Verification
- 11: Verification Condition Generation
- 12: Semantics of Programming Languages
- 13: Model-Checking
- 14: Conclusions and Outlook

Introductive Summary

- This lecture series was about developing systems of high quality and high safety.
- Quality is measured by quality criteria, which guide improvement of the development process. It is basically an economic criterion.
- Safety is “freedom from unacceptable risks”. It is a technical criterion.
- Both high quality and safety can be achieved by the means described in this lecture series.
- Moreover, there is the legal situation: the machinery directive and other laws require (indirectly) you use these techniques where appropriate. This is why these lectures are so important: disregarding this state of the art may make you personally liable.

Quality in the Software Development Process


E/E/PES safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PES architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
CODING



- Hazard analysis
- High-level design
  - SysML, structural diagrams
- Formal Modelling
  - SysML and OCL
- Detailed Specification
  - SysML, behavioural diagrams
- Testing
- Static Program Analysis
- Floyd-Hoare Logic
- Semantics of Programming Languages
- Model-Checking

Examples of Formal Methods in Practice

- Hardware verification:
  - Intel: formal verification of microprocessors
  - Infineon: equivalence checks
- Software verification (research projects):
  - Verisoft — Microsoft Hyper-V (VCC)
  - L4.verified — NICTA, Australia (Isabelle)
- Tools used in industry (excerpt):
  - AbsInt tools: aiT, Astrée, CompCert (C)
  - SPARK tools (Ada)
  - SCADE (MatLab/Simulink)
  - UPAALL, Spin, FDR2, other model checkers

SSQ at University of Bremen

- AG BS (Prof. Jan Peleska): Testing, abstract interpretation.
  - Strong industrial links to aerospace and railway industry, spin-off (Verified Systems)
- DFKI CPS and AG RA (Profs. Rolf Drechsler, Dieter Hutter, Christoph Lüth):
  - Strong industrial links: Infineon, Intel, NXP
  - Hardware and system verification
  - Software verification
  - Security
  - Further application areas: robotics and AAL
- SyDe Graduate College (University of Bremen, DFKI, DLR)
  - Includes more application areas: Space, robotics, real-time image processing

Questions

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 CMM?

Lecture 02: Concepts of Safety and Security

- 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, ISO 15408
- Risk analysis:
  - What is a SIL? Target SIL?
  - How do we obtain a SIL? What does it mean for the development?

Lecture 03: Quality of the Software Development Process

- Which software development models did we encounter?
- Waterfall, spiral, agile, MDD, V-model:
  - How does it work?
  - What are the advantages and disadvantages?
- Which models are appropriate for safety-critical developments?
- What are the typical artefacts (and where do they occur)?
- Formal software development:
  - What is it, and how does it work?
  - How can we define properties, 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?
- Where (in the development process) is it used?
- Basic approaches: bottom-up vs. top-down, and what do they mean?
- Which methods did we encounter?
- FMEA, FTA, Event traces — how do they work, advantages/disadvantages?
- What are the prime verification techniques?

Lecture 05: High-level Design

- High-level specification and modelling:
  - What is it, where in the development process does it take place, what formalisms are useful?
- What is SysML? How does it relate to UML?
- Basic elements of SysML used for high-level design:
  - Structural diagrams:
  - Package diagram
  - Block definition diagram (describes classes, class diagram)
  - Internal block diagrams (describes instances of blocks, flow specifications)
  - Parametric diagram (equational modelling)

Lecture 06: Formal Modelling with SysML and OCL

- What is OCL?
  - A specification language for UML/SysML models
  - Characteristics: pure and typed
- What can we use it for?
  - Invariants on classes and types
  - Pre- and postconditions on operations and methods
- OCL types:
  - Basic types: Boolean, Integer, Real, String; OclAny, OclType, OclVoid
  - Collection types: Sequence, Bag,OrderdedSet, Set
  - Model types
- Logic: three-valued Kleene logic

Lecture 07: Detailed Specification

- What is detailed specification?
  - Specification of single modules — „last" level before code
- What elements are used in specification?
- SysML behavioural diagrams:
  - State diagrams (hierarchical finite state machines)
  - Activity diagrams (flow charts)
  - Sequence diagrams (message sequence charts)
  - Use-case diagrams

Lecture 08: Testing

- What is testing, and what are the aims? What can it achieve, what not?
- What are test elevels?
- What is a black-box test? How are 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 09: Static Program Analysis

- Is what? Where in the development process is it used? What is the difference to testing?
- What is the basic problem, and how is circumvented?
- What does it mean when we say an analysis is sound, or safe?
- What are false positives?
- Did we consider inter- or intraprocedural analysis?
- What examples for forward/backward analysis did we encounter?

Lecture 10: Verification with Floyd-Hoare Logic

- What is Floyd-Hoare logic, what does it do (and what not), and where is used in the development process?
- How does it work?
- What is the difference between $\models \{P\}\, p\, \{q\}$ and $\vdash \{P\}\, p\, \{q\}$?
- What do the notations $\{P\}\, p\, \{Q\}$ and $[P]\, p\, [Q]$ mean?
- What rules does the Floyd-Hoare logic have?
- How are they used?
- Which properties does it have?

Lecture 11: Verification Condition Generation

- What does VCG do?
- How is it related to Floyd-Hoare logic?
- What is a weakest precondition, and how do we calculate it?
- What are program annotations? Why do we need them? How are they used?
- What does $\mathrm{vc}(c, P)$ and $\mathrm{pre}(c, P)$ mean, and how do we calcuate them?
- Which tools do VCG?

Lecture 12: Semantics

- What is semantics? What do we need it for?
- What are the three kinds of semantics, and how to they work?
  - Operational semantics specifies how the program is executed, often as a relation $\langle c, \sigma \rangle \to \sigma$.
  - Denotational semantics models the program as a mathematical entity, often as a partial function $\Sigma \rightharpoonup \Sigma$ using complete partial orders (cpos). Cpos provide mathematical means to handle partiality and fixpoints (iteration).
  - Axiomatic semantics gives proof rules for programs, such as the Floyd-Hoare rules.
  - We can show equivalence of semantics (correctness).
- When do we use which?
  - Operational semantics: implementing the language
  - Denotational semantics: high-level reasoning
  - Axiomatic semantics: reasoning about programs

Lecture 13: Model-Checking with LTL and CTL

- What is model-checking, and how is it used? How does it compare with Floyd-Hoare logic?
- What is the basic question? $\mathcal{M} \models \phi$
  - What do we use for $\mathcal{M}$, ϕ, and do we prove it?
- What is a finite state machine, and what is temporal logic?
- 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?
  - Which one is decidable, and with which complexity?
- What is the basic problem (and limitation) of model-checking?
- Which tools did we see to model-check LTL/CTL?

Module Exams (Modulprüfungen)

- We have the following five areas:
  - Lectures 1 – 4: Quality, Norms and Standards, Development Processes, Requirements Analysis
  - Lecture 5 – 7: SysML
  - Lecture 8 – 9: Testing and Static Program Analysis
  - Lecture 10 – 12: Semantics, Floyd-Hoare Logic and Verification Conditions
  - Lecture 13: Model-Checking with LTL and CTL
- You may choose two areas (except for the first). You need to tell us before the exam starts.
- Questions may come from all lectures, but we will concentrate on the first and your chosen areas.

Final Remark

- Please remember the evaluation (see stud.ip)!

Thank you, and good bye.