Systeme hoher Sicherheit und Qualität

Wintersemester 2013-14

Christoph Lüth
MZH 3100, christoph.lueth@dfki.de, cxl@informatik.uni-bremen.de
Christian Liguda
MZH 3180, christian.liguda@dfki.de

Deutsches Forschungszentrum für Künstliche Intelligenz

Universität Bremen

Inhalt der Vorlesung

- Organisatorisches
- Überblick über die Veranstaltung
- Was ist Qualität?

ORGANISATORISCHES

Generelles

- Einführungsvorlesung zum Masterprofil **Sicherheit und Qualität**
- 6 ETCS-Punkte
- Vorlesung
  - Montag 12 c.t – 14 Uhr (MZH 1110)
- Übungen:
  - Dienstag 12 c.t. – 14 Uhr (MZH 1450)
- Webseite:
  http://www.informatik.uni-bremen.de/~cxl/lehre/sqs.ws13/

Folien, Übungsblätter, etc.

Folien

- Folien sind auf Englisch (Notationen!)
- Folien der Vorlesung gibt es auf der Homepage
- Folien sind (üblicherweise) nach der Vorlesung verfügbar

Übungen

- Übungsblätter gibt es auf dem Web
- Ausgabe Montag abend/Dienstag morgen
  - Erstes Übungsblatt **heute**
- Abgabe **vor** der Vorlesung
  - Persönlich hier, oder per Mail bis **Montag 12:00**

Literatur

- Foliensätze als Kernmaterial
- Ausgewählte Fachartikel als Zusatzmaterial
- Es gibt (noch) keine Bücher, die den Vorlesungsinhalt komplett erfassen
  (Wer hat Lust, bei einem Skript mitzuhelfen?)
- Zum weiteren Stöbern
  - Wird im Verlauf der Vorlesung bekannt gegeben

Prüfungen

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

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

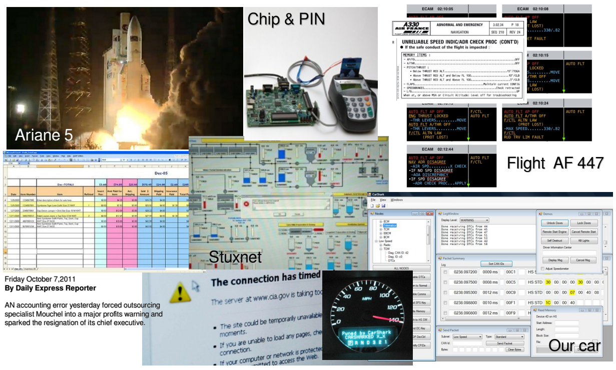
- Modulprüfung
  - Keine Abgabe der Übungsblätter nötig
    (aber Bearbeitung dringend angeraten !!!)

OVERVIEW

Objectives

- This is an introductory lecture for the topics
  Quality – Safety – 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
- Recall: the three focal points of computer science research at the FB3 are
  - Digital Media
  - Artificial Intelligence and Cognition
  - Quality, Safety & Security
- Disclaimer
  - “Lecture Eintopf”
  - Choice of material reflects personal preferences

Why Bother with S & Q?

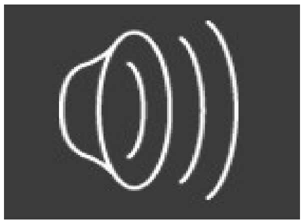


Why did Ariane-5 crash?

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

Engineering Sciences

- Mathematical theories
  - Statics
  - Computational models



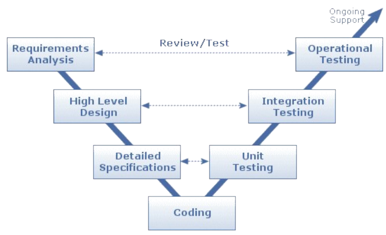
What is Safety and Security

- Safety
  - product achieves acceptable levels of risk or harm to people, business, software, property or the environment in a specified context of use
  - Threats from “inside”
    - Avoid malfunction of a system (e.g. planes, cars, railways…)
- Security
  - Product is protected against potential attacks from people, environment etc.
  - Threats from “outside”
    - Analyze and counteract the abilities of an attacker

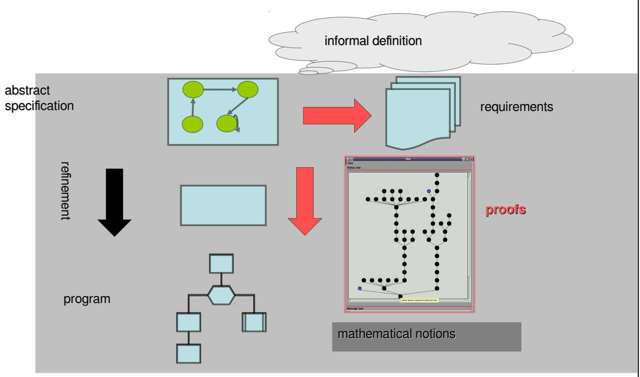
Software Development

Definition of software engineering processes and documents

- V-model
- Model Driven Architectures
- Agile Development



Formal Software Development



Verification & Validation

- Verification: have we built the system right (i.e. correct)?
- Validation: have we built the right system (i.e. adequate)?
- Testing
  - Test case generation, black- vs. white box
- Symbolic evaluation
  - Program runs using symbolic values
- Static/dynamic program analysis
  - Dependency graphs, flow analysis
- Model checking
  - Formal verification of finite state problem
- Formal Verification
  - Formal verification of requirements, program properties…

Overview of Lecture Series

- Lecture 01: Concepts of Quality
- Lecture 02: Concepts of Safety, Legal Requirements, Certification
- Lecture 03: A Safety-critical Software Development Process
- Lecture 04: Requirements Analysis
- Lecture 05: High-Level Design & Detailed Specification

- Lecture 06: Testing
- Lecture 07 and 08: Program Analysis
- Lecture 09: Model-Checking
- Lecture 10 and 11: Software Verification (Hoare-Calculus)

- Lecture 12: Concurrency
- Lecture 13: Conclusions

Concepts of Quality

Universität Bremen

What is Quality

- The 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

- For the development of artifacts quality criteria can be measured with respect to the
  - development process (process quality) *(later in this lecture)*
  - 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.)

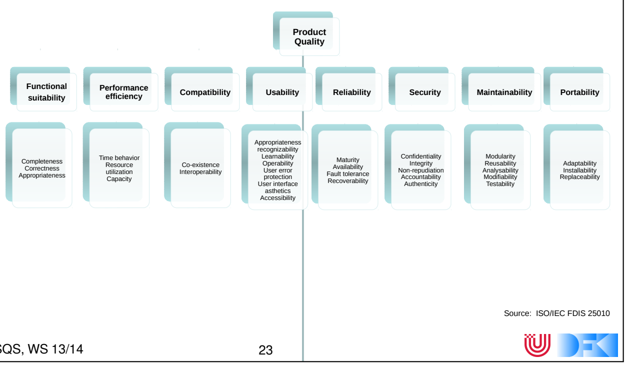
- A third dimension structures quality according to product properties:
  - Functional properties: the specified services to be delivered to the users
  - Structural properties: architecture, interfaces, deployment, control structures
  - Non-functional properties: usability, safety, reliability, availability,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

Quality (ISO/IEC 25010/12)

Quality model framework

- 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



Source: ISO/IEC FDIS 25010

Functional Suitability

- The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions
- Characteristics
  - **Completeness**: degree to which the set of functions cover the specified tasks and objectives
  - **Correctness**: degree to which a system / product provides the correct results within the needed degree of precision
  - **Appropriateness:** degree to which the functions facilitate the accomplishment of specified tasks and objectives

Performance Efficiency

- The capability of the software product to provide appropriate performance, relative to the amount of resources used, when used under specified conditions
- Characteristics
  - **Time behavior:** degree to which the response and processing times and throughput rates of a product meet requirement, when performing its functions
  - **Resource utilization:** degree to which the amounts and types of resources used by a product meet requirements when performing its functions
  - **Capacity:** degree to which the maximum limits of a product parameter meet requirements

Compatibility

- The capability of the software product to exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment
- Characteristics
  - **Co-Existence:** degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product
  - **Interoperability**: degree to which two or more systems, products or components can exchange information and use the information that has been exchanged

Usability

- The capability of the software product to be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use
- Characteristics
  - **Appropriateness Recognizability**: degree to which users can recognize whether a product is appropriate for their needs
  - **Learnability**: degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use
  - **Operability:** degree to which a product or system has attributes that make it easy to operate and control
  - **User Error Protection:** degree to which a system protects users against making errors
  - **User Interface Aesthetics:** degree to which a user interface enables pleasing and satisfying interaction for the user
  - **Accessibility:** degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use

Reliability

- The capability of the software product to perform specified functions under specified conditions for a specified period of times
- Characteristics
  - **Maturity**: degree to which a system meets needs for reliability under normal operation
  - **Availability**: degree to which a system, product or component is operational and accessible when required for use
  - **Fault tolerance**: degree to which a system, product or component operates as intended despite the presence of hardware or software faults
  - **Recoverability:** degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system

Security

- The capability of the software product to protect information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization
- Characteristics
  - **Confidentiality:** degree to which a product or system ensures that data are accessible only to those authorized to have access
  - **Integrity:** degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data
  - **Non-Repudiation:** degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later
  - **Accountability:** degree to which the actions of an entity can be traced uniquely to the entity
  - **Authenticity:** degree to which the identity of a subject or resource can be proved to be the one claimed

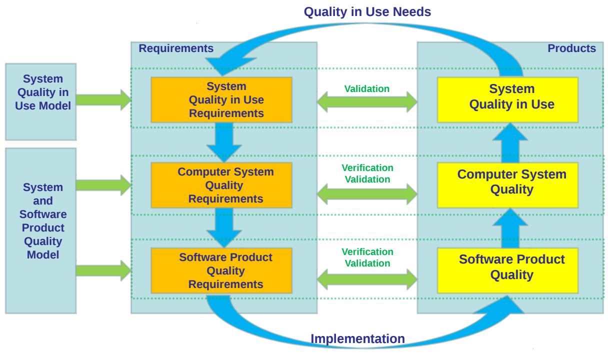
Maintainability

- The degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers
- Characteristics
  - **Modularity:** degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components
  - **Reusability:** degree to which an asset can be used in more than one system, or in building other assets
  - **Analysability:** degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified
  - **Modifiability:** degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality
  - **Testability:** degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met

Portability

- The capability of the software product to be from one hardware, software or other operational or usage environment to another
- Characteristics
  - **Adaptability:** degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments
  - **Installability:** degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment
  - **Replaceability:** degree to which a product can be replaced by another specified software product for the same purpose in the same environment

System Quality Life Cycle Model



Source: ISO/IEC FDIS 25010

Quality in Use Model

Effectiveness

- The accuracy and completeness with which users achieve specified goals
- No further characteristics

Efficiency

- The resources expended in relation to the accuracy and completeness with which users achieve goals
- No further characteristics

Satisfaction

- The degree to which user needs are satisfied when a product or system is used in a specified context of use
- Characteristics
  - **Usefulness:** degree to which a user is satisfied with their perceived achievement of pragmatic goals, including the results of use and the consequences of use
  - **Trust:** degree to which a user or other stakeholder has confidence that a product or system will behave as intended
  - **Pleasure:** degree to which a user obtains pleasure from fulfilling their personal needs
  - **Comfort:** degree to which the user is satisfied with physical comfort

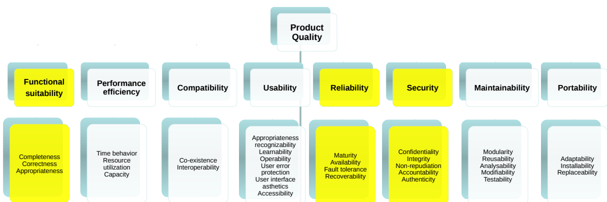
Freedom From Risk (Safety)

- The capability of the software product to mitigate the potential risk to economic status, human life, health, or the environment
- Characteristics
  - **Economic risk mitigation:** degree to which a product or system mitigates the potential risk to financial status, efficient operation, commercial property, reputation or other resources in the intended contexts of use
  - **Health and safety risk mitigation:** degree to which a product or system mitigates the potential risk to people in the intended contexts of use
  - **Environmental risk mitigation:** degree to which a product or system mitigates the potential risk to property or the environment in the intended contexts of use

Context Coverage

- The capability of the software product to be used with effectiveness, efficiency, freedom from risk and satisfaction in both specified contexts of use and in contexts beyond those initially explicitly identified
- Characteristics
  - **Context completeness:** degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in all the specified contexts of use
  - **Flexibility:** degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in contexts beyond those initially specified in the requirements

Focus of Interest



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 & institionalized
    - Level 4 (Managed): measured process
    - Level 5 (optimizing): improvement fed back into process

Summary

- Quality:
  - collection of characteristic properties
  - quality **indicators** measuring quality **criteria**

- Relevant **aspects** of quality here:
  - Functional suitability
  - Reliability
  - Security

Systeme hoher Qualität und Sicherheit
Universität Bremen, WS 2013/14

Lecture 02 (28.10.2013)

Concepts of Safety and Security

Christoph Lüth
Christian Liguda

Where are we?

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

Synopsis

- **If** you want to write safety-criticial software,
  then you need to adhere to state-of-the-art practise
  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
    - 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. 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.
- The **good** news:
  - Pay attention here and you will be sorted.

Safety:
IEC 61508
and other norms & standards

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 safety-critical development process

Some Terminology

- Fail-safe vs. Fail operational
- Safety-critical, safety-relevant (*sicherheitskritisch*)
  - General term -- failure may lead to risk
- Safety function (*Sicherheitsfunktion*)
  - Techncal 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)
- **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
- Note that the scope 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.

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
  - Specialisation of 61508 to software for railway industry
- RTCA DO 178-B:
  - *"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

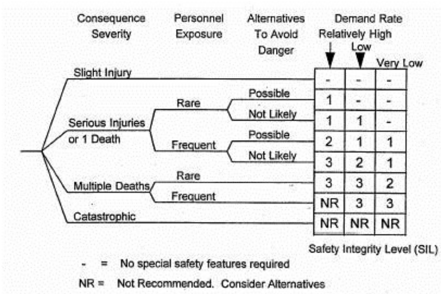
- IEC 61508 does not describe standard procedure to establish a SIL target, it allows for alternatives:
- Quantitative approach
  - Start with target risk level
  - Factor in fatality and frequency

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

- Risk graph approach



Example: safety braking system for an AGV

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

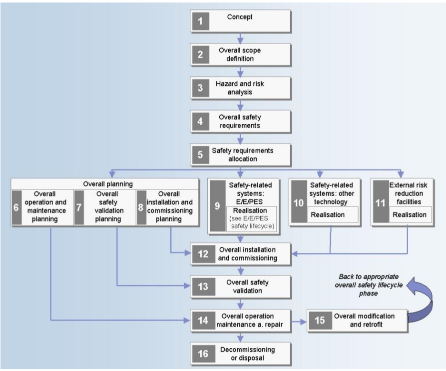
Increasing SIL by redudancy

- 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 Safety Life Cycle



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 (in particular, modules).
  - 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

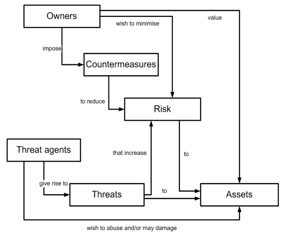
Common Criteria (IEC 15408)

- 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.

Common Criteria

General Model

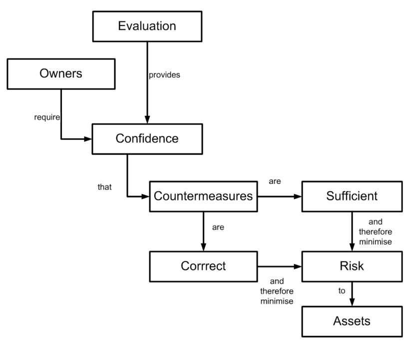
- Security is concerned with the protection of assets. Assets are entities that someone places value upon.
- Threats give rise to risks to the assets, based on the likelihood of a threat being realized and its impact on the assets
- (IT and non-IT) Counter-measures are imposed to reduce the risks to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



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. Developement, life cycle support, testing, ...

Functional Requirement

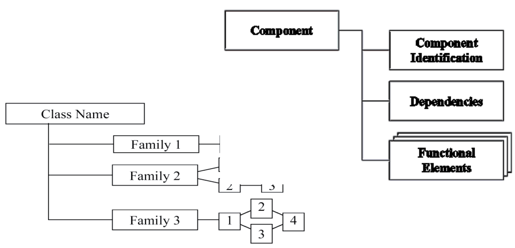
- The **functional requirements** are levied on those functions of the TOE that are specifically in support of IT security, and define the desired security behavior.
- Part 2 defines the CC functional requirements. Examples of functional requirements include requirements for identification, authentication,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

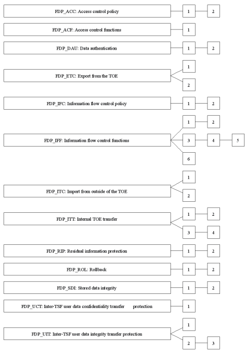
Security Functional Components

- Content and presentation of the functional requirements



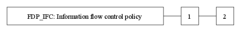
Decomposition of FDP

- FDP : User Data Protection



FDP – 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

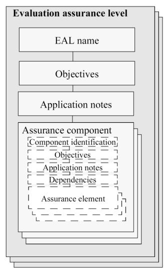
"The CC philosophy is to provide assurance based upon an evaluation (active investigation) of the IT product that is to be trusted. Evaluation has been the traditional means of providing assurance and is the basis for prior evaluation criteria documents. "

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.

Systeme hoher Qualität und Sicherheit
Universität Bremen, WS 2013/14

Lecture 03 (04.11.2013)
Quality of the Software Development Process

Christoph Lüth
Christian Liguda

Universität Bremen

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:
  - How to specifiy: properties
  - Structuring of the development process

Where are we?

- Lecture 01: Concepts of Quality
- Lecture 02: Concepts of Safety and Security, Norms and Standards
- Lecture 03: Quality of the Software Development Process
- Lecture 04: Requirements Analysis
- Lecture 05: High-Level Design & Detailed Specification
- Lecture 06: Testing
- Lecture 07 and 08: Program Analysis
- Lecture 09: Model-Checking
- Lecture 10 and 11: Software Verification (Hoare-Calculus)
- Lecture 12: Concurrency
- Lecture 13: Conclusions

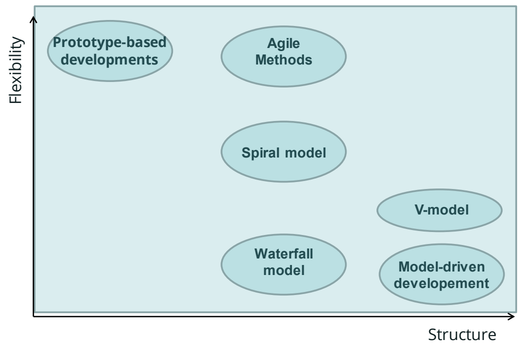
Software Development Models

Universität Bremen

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...)

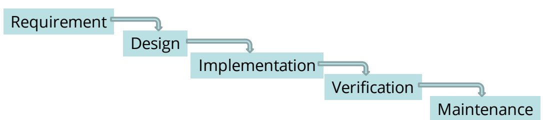
Software Development Models



from S. Paulus: Sichere Software

Waterfall Model (Royce 1970)

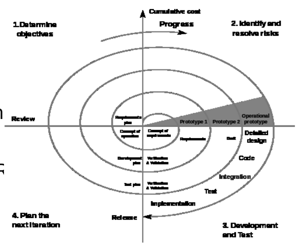
- Classical top-down sequential workflow with strictly separated phases.



- 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 (F



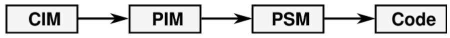
- Drawbacks:
  - Risk identification is the key, and can be quite difficult

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

Model-Driven Development (MDD, MDE)

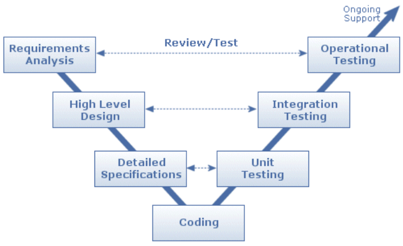
- Describe problems on abstract level using *a modelling language* (often a *domain-specific language*), and derive implementation by model transformation or run-time interpretation.
- Often used with UML (or its DSLs, eg. SysML)



- Variety of tools:
  - Rational tool chain, Enterprise Architect
  - EMF (Eclipse Modelling Framework)
- Strictly sequential development
- Drawbacks: high initial investment, limited flexibility

V-Model

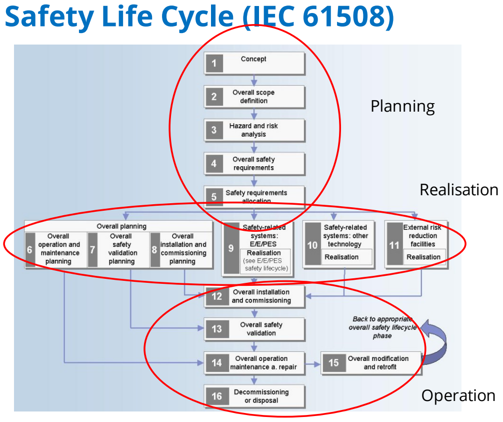
- Evolution of the waterfall model:
  - Each phase is supported by a corresponding testing phase (verification & validation)
  - Feedback between next and previous phase
- Standard model for public projects in Germany
  - … but also a general term for models of this „shape“



Development Models for Critical Systems

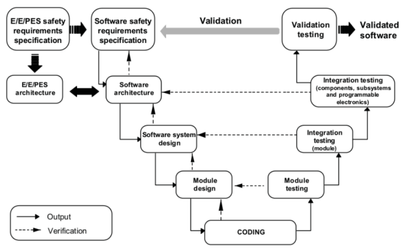
- Ensuring safety/security needs structure.
  - …but *too much* structure makes developments bureaucratic, which is *in itself* a safety risk.
  - Cautionary tale: Ariane-5
- Standards put emphasis on *process*.
  - Everything needs to be planned and documented.
- Best suited development models are variations of the V-model or spiral model.

The Safety Life Cycle (IEC 61508)



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 *recommends* V-model.



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
  - Integral process
- There is no conspicuous diagram, but these are the phases found in the V-model as well.
  - Implicit recommendation.

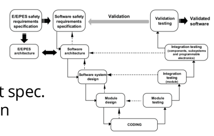
Artefacts in the Development Process

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
- Tests and test scripts
- Proofs



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

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

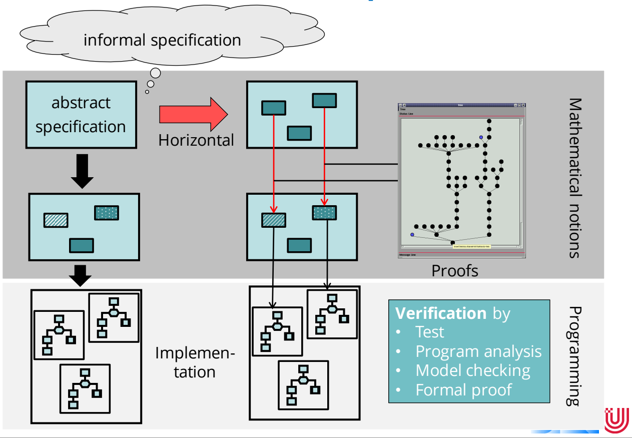
Basic Notions of Formal Software Development

Universität Bremen

Formal Software Development

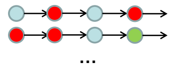
- In **formal** development, properties are stated in a rigorous way with a precise mathematical semantics.
- These formal specifications can be **proven**.
- Advantages:
  - Errors can be found **early** in the development process, saving time and effort and hence costs.
  - There is a higher degree of trust in the system.
  - Hence, standards recommend use of formal methods for high SILs/EALs.
- Drawback:
  - 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

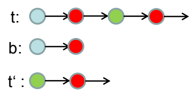


Properties

- A general notion of **properties**.
- Properties as set of infinite execution traces (i.e. infinite sequences of states)



- Trace t satisfies property P, written $P \vDash t$, iff $t \in P$
- b ≤ t iff ∃ t′ . t = b • t′
  - i.e. b is a *finite* prefix of t



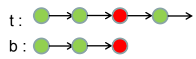
Safety and Liveness Properties

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 . $P = Safe_P \cap Live_P$
  - Each property can be represented as a combination of safety and liveness properties.

Safety Properties

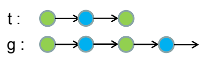
- Safety property S: „Nothing bad happens"
- A bad thing is *finitely* observable and *irremediable*
- S is a safety property iff
  - $\forall t. t \notin S \rightarrow (\exists b. \text{finite } b \wedge b \leq t \rightarrow \forall u. b \leq u \rightarrow u \notin S)$



  - a finite prefix b always causes the bad thing
- **Safety is typically proven by induction**
  - Safety properties may be enforced by run-time monitors.

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*.
- A system S *satisfies* a property P, written

  $S \vDash P$ iff $S \subseteq P$

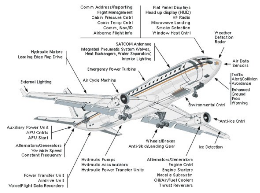
  (i.e. $\forall t \in S. t \in P$, all traces satisfy the property P).
- Why more than one trace? Difference between:
  - *Underspecification* or *loose specification* – we specify several *possible* implementations.
  - Non-determinism – different program runs might result in different traces.
- Example: a simple can vending machine.
  - Insert coin, chose brand, dispense drink.
  - Non-determinisim due to *internal* or *external* choice.

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

$$S_1; S_2 = \{ s \cdot t \mid s \in S_1, t \in S_2 \}$$

  - All traces from S_1, followed by all traces from S_2.

- Given two traces s, t, their *interleaving* is defined (recursively) as
  $<> \parallel t = t$
  $s \parallel <> = s$
  $a \cdot s \parallel b \cdot t = \{ a \cdot u \mid u \in s \parallel b \cdot t \} \cup \{ b \cdot u \mid u \in a \cdot s \parallel t \}$

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

$$S_1 \parallel S_2 = \{ s \parallel t \mid s \in S_1, t \in S_2 \}$$

  - 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

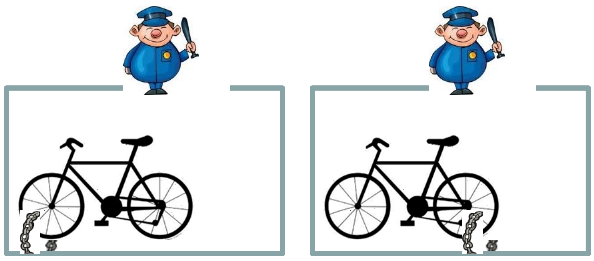
- Refinement typically preserves safety properties.
  - This means if we start with an abstract specification which we can show satisfies the desired properties, and refine it until we arrive at an implementation, we have a system for the properties hold *by construction*:

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

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

Security and Composition

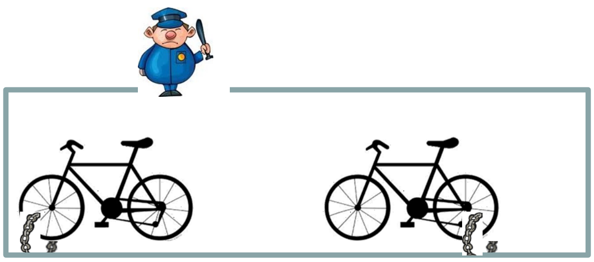
Only complete bicycles are allowed to pass the gate.



Secure ! Secure !

Security and Composition

Only complete bicycles are allowed to pass the gate.



Insecure !

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 include safety and liveness properties.
- Structuring of the development:
  - Horizontal – e.g. composition
  - Vertical – refinement (data, process and action ref.)

Systeme hoher Qualität und Sicherheit
Universität Bremen, WS 2013/14

Lecture 04 (11.11.2013)

Hazard Analysis Techniques

Christoph Lüth
Christian Liguda

Where are we?

- Lecture 01: Concepts of Quality
- Lecture 02: Concepts of Safety and Security, Norms and Standards
- Lecture 03: Quality of the Software Development Process
- Lecture 04: Requirements Analysis
- Lecture 05: High-Level Design & Formal Modelling
- Lecture 06: Detailed Specification
- Lecture 07: Testing
- Lecture 08: Program Analysis
- Lecture 09: Model-Checking
- Lecture 10 and 11: Software Verification (Hoare-Calculus)
- Lecture 12: Concurrency
- Lecture 13: Conclusions

Your Daily Menu

- Ariane-5: A cautionary tale
- Hazard Analysis:
  - What's that?
- Different forms of hazard analysis:
  - FMEA, Failure Trees, Event Trees.
- An extended example: OmniProtect

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?

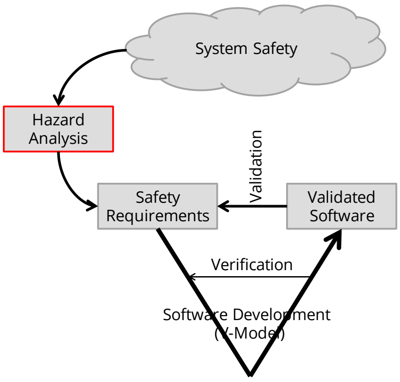
- Self-destruct triggered after 39 secs. due to inclination over 20 degr.
- OBC sent commands because it had incorrect data from IRS and tried to `adjust' trajectory.
- IRS sent wrong data because it had experienced software failure (overflow when converting 64 bit to 16 bit).
- Overflow occured when converting data to be sent to ground control (for test/monitoring purposes only).
- Overflow occured because
  - IRS was integrated as-is from Ariane 4, and
  - a particular variable (Horizontal Bias) held far higher values for the new model, and
  - the integer conversion was not protected because it was assumed that its values would never become too large.
  - This **assumption** was not **documented**.
- Because of its criticality, IRS had a backup system, but it ran the same software, so it failed as well (actually, 72 ms before the main one).

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.

Hazard Analysis i/t Development Process



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.

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

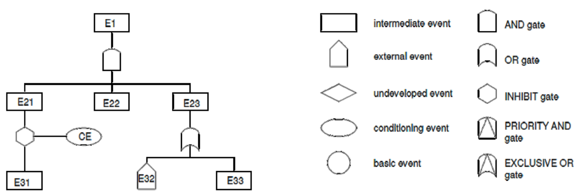
- **Top-down methods** start with an anticipated hazard and work back from the hazard event to potential causes for the hazard
  - Good for finding causes for hazard
  - Good for avoiding the investigation of "non-relevant" errors
  - Bad for detection of missing hazards
- **Bottom-up methods** consider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

Hazard Analysis Methods

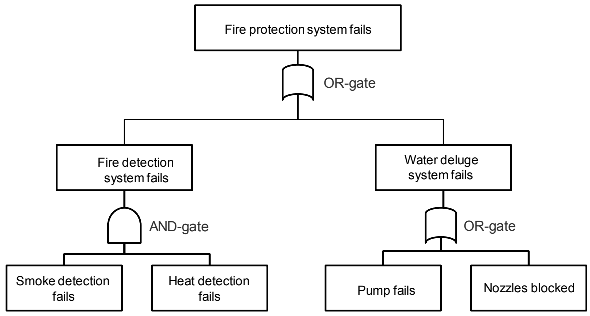
- Fault Tree Analysis (FTA) – top-down
- Failure Modes and Effects Analysis (FMEA) – bottom up
- Event Tree Analysis – bottom-up
- Cause Consequence Analysis – bottom up
- HAZOP Analysis – bottom up

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



Fault Tree Analysis: Example



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

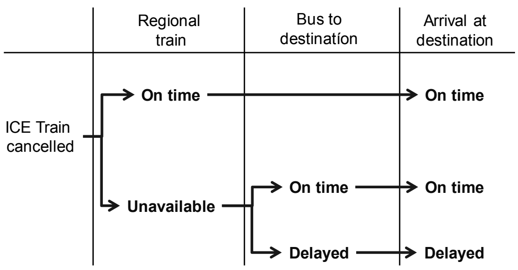
FMEA Example: Airbag Control (Funct.)

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

Event Tree Analysis

- 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



Hazard Analysis as a Reachability Problem

The analysis whether “finally something bad happens” is well-known from **property checking** methods

- Create a model describing everything (desired or undesired) which might happen in the system under consideration
- Specify a logical property *P* describing the undesired situations
- Check the model whether a path – that is, a sequence of state transitions – exists such that *P* is fulfilled on this path
- Specify as safety requirement that mechanisms shall exist preventing paths leading to *P* from being taken

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.

Verifying Requirements

- **Testing**
  - Executable specification (i.e. sort of implementation)
  - Covering individual cases
  - Functional requirements
  - Decidable
- **(Static / Dynamic) Program Analysis**
  - Executable specification
  - Covering all cases
  - Selected functional and non-functional requirements
  - Decidable (but typically not complete)

Verifying Requirements II

- **Model Checking**
  - Formal specification
  - Covering all cases
  - Functional and non-functional properties (in finite domains)
  - Decidable (in finite domains)
- **Formal Verification**
  - Formal specification
  - Covering all cases
  - All types of requirements
  - (Usually) undecidable

Our Running Example: OmniProtect

- OmniProtect is a safety module for an omnidirectional AGV such as the Kuka OmniMove.
  - *Demonstration project only.*

- It calculates **a safety zone** (the area needed for breaking until standstill).
- Documents produced:
  - Document plan
  - Concept paper
  - Fault Tree Analysis
  - Safety Requirements
  - .... more to come.

Summary

- Hazard Analysis is the **start** of the formal development.
- It produces **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)
- 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.
- Next week: High-Level Specification.

Systeme Hoher Qualität und Sicherheit
Vorlesung 5 vom 18.11.2013: High-Level Specification and Modelling

Christoph Lüth & Christian Liguda

Universität Bremen

Wintersemester 2013/14

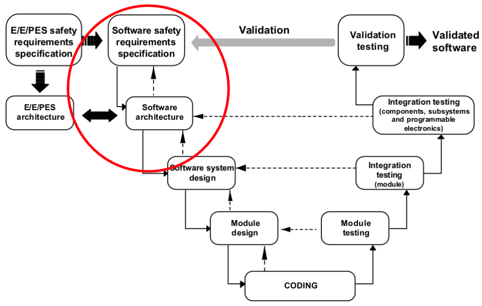
Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- **Lecture 5: High-Level Design & Formal Modelling**
- Lecture 6: Detailed Specification, Refinement & Implementation
- Lecture 7: Testing
- Lecture 8: Program Analysis
- Lecture 9: Verification with Floyd-Hoare Logic
- Lecture 10: Verification Condition Generation
- Lecture 11: Model-Checking with LTL and CTL
- Lecture 12: NuSMV and Spin
- Lecture 13: Conclusions

Your Daily Menu

- High-Level Specification and Modelling
- The Z Notation as an example of a modelling language
  - Basics, Schema Calculusm, Mathematical Library
  - Canonical Example: the Birthday Book
- Running Example *OmniProtect*
  - Modelling the safe robot

High-Level Specification and Modelling



- Here, we want to be able to express **high-level** requirements **abstractly**, **precisely** and **without** regards for the implementation.

Why look at Z?

- Z is a good example of a **modelling language**.
- It allows us to model high-level specifications in a **mathematically precise** fashion, unambigious and exact.
- Z is **easy to grasp**, as opposed to other mechanisms — we quickly get off the ground.
- Alternatives would be UML (in particular, class diagrams plus OCL), but that is on the one hand already geared towards implementation, and on the other hand there is less tool support for OCL. UML support is more geared towards code generation, not so much abstract modelling as appropriate in this phase of the design process.

The Z Notation

- Z is a **notation** based on **typed set theory**.
  - That means everything is described in terms of **set** (sets are types)
  - There is a lot of syntactic convention ("syntactic sugar")
- It is geared towards the specification of **imperative** programs
  - State and state change built-in
- Developed late 80s (Jean-Claude Abrial, Oxford PRG; IBM UK)
- Used industrially (IBM, Altran Praxis ex. Praxis Critical Systems))
- LaTeX-Notation and tool support (Community Z Tools, ProofPower)

Introducing the Birthday Book

- The birthday book is a well-known example introducing the main concepts of the Z language. It can be found e.g. in the Z reference manual (freely available, see course home page).
- It models a **birthday calendar**, where one can keep track of birthdays (of family, acquaintances, business contacts ...)
- Thus, we have names and dates as types, and operations to
  - add a birthday,
  - find a birthday,
  - and get reminded of birthdays.

Birthday Book: Types

- We start by **declaring** the types for names and date. We do not say what they are:

  $[NAME, DATE]$

- In Z, we define operations in terms of state transitions.
- We start with defining the **state space** of our birthday book system. This is our **abstract** view of the system state.
- The system state should contain **names** and **birthdays**, and they should be related such that we can map names to birthdays.

Birthday Book: The System State

- The system state is specified in form of a **Z schema**. A schema consists of two parts: **variable declarations** and **axioms**.

BirthdayBook
$known : \mathbb{P}\, NAME$
$birthday : NAME \rightarrowtail DATE$

$known = \mathrm{dom}\; birthday$

- This says that there is a set *known* of names, and a **partial map** from names to dates. (Z has a library, called the Mathematical Toolkit, of predefined types and operations, such as the power set and partial map used here.)
- The axiom is an **invariant** (meaning it has to be preserved by all operations). It says that the set of known names is the domain of the *birthday* map.

Schema Operations: Pre- and Poststate

- Operations are defined as schemas as well.
- Operations have a prestate (before the operation is applied) and a poststate (after the operation has been applied). The poststate is denoted by dashed variables.
- Here is an operation which just adds my name, *cxl*, to a set of known names:

AddMe
$known : \mathbb{P}\, NAME$
$known' : \mathbb{P}\, NAME$

$known' = known \cup \{cxl\}$

- In order to minimise repetition, schemas can comprise other schemas. We can also dash whole schemas. Further, the **schema operator** ΔS is shorthand for $\Delta S \stackrel{def}{=} S \wedge S'$, or "include S and S'".

Birthday Book: First Operation

- As a first operation, we want to add a birthday.
- This requires a name and a birthday as **input variables**.

AddBirthday
$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

- Input variables are only defined in the prestate. (Similary, output variables, written as *name*!, are only defined in the poststate.)
- The *Birthday* invariant holds both in pre- and poststate. From this, we can show that the following sensible property holds:

$known' = known \cup \{name?\}$

Birthday Book: Finding a birthday

- Finding a birthday gives the name as input, and a date as output:

FindBirthday
$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$
$date! = birthday(name?)$

- This introduces the Ξ operator. It is shorthand for $\Xi S \stackrel{def}{=} (\Delta S \wedge S = S')$ (or, "for schema S, nothing changes".)
- The *FindBirthday* operation has a precondition (the *name* must be in the set of known names); only if that holds, the postcondition is guaranteed to hold as well.

Birthday Book: Reminders

- A reminder takes a date as input, and returns the names of entries with this birthday.

Remind
$\Xi BirthdayBook$
$today? : DATE$
$cards! : \mathbb{P}\, NAME$

$cards! = \{n : known \mid birthday(n) = today?\}$

RemindOne
$\Xi BirthdayBook$
$today? : DATE$
$card! : NAME$

$card! \in known$
$birthday\; card! = today?$

- A variation of this schema just selects one name. It is an example of a **non-deterministic** operation.

Birthday Book: Putting it all together

- We need an **initial state**. It does not say explicitly that *birthday'* is empty, but that is implicit, because its domain is empty.

InitBirthdayBook
$BirthdayBook'$

$known' = \{\}$

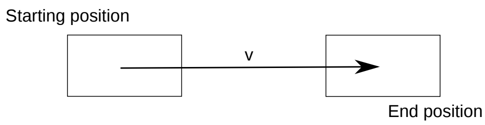
- And we put it all together by conjoining the schemas:

$System == InitBirthdayBook \wedge (AddBirthday \wedge FindBirthday \wedge Remind)$

Case Study: the *OmniProtect* Project

- The objective of the *OmniProtect* project is to develop a safety module for omnimobile **robots**.
- These robots have a behaviour which is easily describable: they move with a velocity which is given by a vector $\vec{v}$ (per time t).
- The velocity can be changed **instanteneously**, but we assume that braking is **linear**.
- The shape of the robot is described by a **convex polygon**.
- We move the robot by moving the polygon by the given velocity $\vec{v}$.
- We will first describe this movement, and the area covered by this movement (modelling). We will then (next lecture) describe the actual operations (high-level specification), and investigate how to implement them (low-level specification).

Modelling the Safe Robot: Planar Movement



- Braking time and braking distance:

$v(t) = v_0 - a_{brk}\, t \qquad s(t) = v_0 t - \frac{a_{brk}}{2} t^2 \qquad T = \frac{v_0}{a_{brk}} \qquad S = \frac{v_0^2}{2 a_{brk}}$

- Modelling in **Z**: Calculating the braking distance

$brk : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$\forall\, v, a : \mathbb{N} \bullet brk\,(v, a) = (v * v)\ \mathrm{div}\ (2 * a)$

Mathematical Modelling: Points and Vectors

- Schema for points (vectors):

$$\begin{array}{l} \textit{VEC} \\ \hline x : \mathbb{Z} \\ y : \mathbb{Z} \\ \hline \end{array}$$

- Type für Polygons und segments:

$$POLY == \{s : \mathrm{seq}\ VEC \mid \# s > 3 \wedge head\ s = last\ s\}$$
$$SEG == VEC \times VEC$$

- This introduces the type of **sequents**, seq, or finite lists, from the Mathematical Toolkit. There are a number of useful predefined functions on lists.

Mathematical Modelling: Vector Operations

- Addition and scalar multiplication of vectors

$$\begin{array}{|l} add : VEC \times VEC \to VEC \\ smult : R \times VEC \to VEC \\ \hline \forall\, p, q : VEC \bullet add(p, q) = (p.x + q.x, p.y + q.y) \\ \forall\, n : R;\ p : VEC \bullet smult\,(n, p) = (n * p.x, n * p.y) \end{array}$$

- We have slightly cheated here — Z does not really know real numbers.

More abouts Points and Vectors

- A segment defines a **left half-plane** (as a set of points)

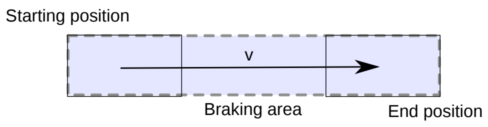
$$\begin{array}{|l} left : SEG \to \mathbb{P}\, VEC \\ \hline \forall\, a, b : VEC \bullet left\,(a, b) = \{p : VEC \mid (b.y - a.y) * (p.x - b.x) - \\ \qquad (p.y - b.y) * (b.x - a.x) < 0\} \end{array}$$

- The **area** of a (convex!) polygon is the intersection of the left half-planes given by its sides.

$$\begin{array}{|l} sides : POLY \to \mathbb{P}\, SEG \\ area : POLY \to \mathbb{P}\, VEC \\ \hline \forall\, p : POLY \bullet sides\ p = \{s : SEG \mid \langle s.1, s.2 \rangle \text{ in } p\} \\ \forall\, p : POLY \bullet area\ p = \bigcap \{s : SEG \mid s \in sides\ p \bullet left\ s\} \end{array}$$

- We should make the restriction on convex explicit (next lecture).

Moving Polygons



- Moving a polygon by a vector:

$$\begin{array}{|l} move : POLY \times VEC \to POLY \\ \hline \forall\, p : POLY;\ v : VEC \bullet move(p, v) = (\lambda\, x : VEC \bullet add(x, v)) \circ p \end{array}$$

- Area covered by this movement

$$\begin{array}{|l} cov : POLY \times VEC \to \mathbb{P}\, VEC \\ \hline \forall\, p : POLY;\ v : VEC \bullet \\ \quad cov\,(p, v) = \bigcup \{\tau : R \mid 0 \leq \tau \leq 1 \bullet area(move(p, smult(\tau, v)))\} \end{array}$$

Summary

- Z is a modelling language based on **typed set theory**
- Its **elements** are
  - axiomatic definitions
  - schema and the schema calculus
  - the Mathematical Toolkit (standard library)
- In Z, we start with modelling the **system state(s)**, followed by the **operations** (which are state transitions)
- The birthday book example can be found in the Z reference manual.
- We have started with modelling the robot.
- Next lecture: the **safe** robot, and its operations.

Systeme Hoher Qualität und Sicherheit
Vorlesung 6 vom 25.11.2013: Detailed Specification, Refinement & Implementation

Christoph Lüth & Christian Liguda

Universität Bremen

Wintersemester 2013/14

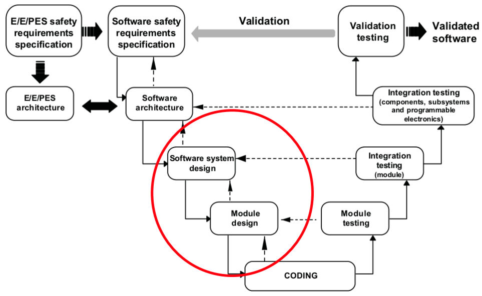
Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- Lecture 5: High-Level Design & Formal Modelling
- **Lecture 6: Detailed Specification, Refinement & Implementation**
- Lecture 7: Testing
- Lecture 8: Static Program Analysis
- Lecture 9: Verification with Floyd-Hoare Logic
- Lecture 10: Verification Condition Generation
- Lecture 11: Model-Checking with LTL and CTL
- Lecture 12: NuSMV and Spin
- Lecture 13: Concluding Remarks

Your Daily Menu

- **Refinement**: from abstract to concrete specification
- **Implementation**: from concrete specification to code
- Running examples: the safe autonomous robot, the birthday book

Design Specification



- At this point, we want to be **relate** implementation to the more abstract specifications in the higher lever, and have a **systematic** way to go from higher to lower levels (**refinement**).

Refinment in the Development Process

- Recall that we have **horizontal** and **vertical** structuring.
- **Refinement** is a **vertical** structure in the development process.
- The simplest form of refinement is **implicational**, where an implementation I implies the abstract requirement A

$$I \Rightarrow A$$

- Recall that refinement typically preserves **safety** requirements, but not **security** — thus, there is a systematic way to construct safe systems, but not so for secure ones.

The Autonomous Robot: Basic Types

- We first declare a datatype for the time:

$[Time]$

- We then declare the robot parameters, and the state of the world — these are the things which do not change.

RobotParam
$cont : POLY$

- Obstacles are just a set of points (instead of polygons)

World
$RobotParam$
$obs : \mathbb{P}\, VEC$

The Autonomous Robot: Safety Requirements

- The robot's state depends on the time, so we do not have pre/post conditions. It has a position vector, o, which determines the current contour polygon c.

Robot
$RobotParam$
$c : Time \rightarrow POLY$
$o : Time \rightarrow VEC$

$c(t) = move(cont, o(t))$

- Here is the **main safety requirement**: the robot is safe if its current contour never contains any obstacles.

RobotSafe
$Robot$

$\forall\, t . c(t) \cap obs = \emptyset$

The Autonomous Robot: Implementation

- When implementing the autonomous robot, we assume a **control loop** architecture, where a **control function** is called each T ms. It can read the**current** system state, and sets **control variables** which determine the system's behaviour over the next clock cycle.
- The cycle time ("tick") T is part of the robot parameters. We also add the braking accelaration a_{brk}.

RobotParam
$cont : POLY$
$a_{brk} : \mathbb{Z}$
$T : \mathbb{Z}$

World
$RobotParam$
$obs : \mathbb{P}\, VEC$

The Autonomous Robot: Implementation

- This specifies the **control behaviour** of the robot.
- Velocity is given by the **linear velocity** *vel*, and steering angle ω. This describes the velocity vector v in polar form.
- This does not yet describe how the velocity is controlled.

Robot
RobotParam
$vel, \omega : \mathbb{Z}$
$v, o : VEC$
$c : POLY$

$c = move\,(cont, o)$
$v = cart\,(vel, \omega)$

- The function *cart* converts a vector in polar form to the cartesian form. A simple specification in Z might be this:

$cart : \mathbb{Z} \times R \to VEC$

$\forall\, r : \mathbb{Z};\ \omega : R;\ p : VEC \bullet cart(r, \omega) = p \Rightarrow r * r = p.x * p.x + p.y * p.y$

- Unfortunately, the Mathematical Toolkit does not support trigonometric functions (or real numbers).

The Autonomous Robot: Control

- The velocity is controlled by two **input variables** $a?$ and $d\omega?$, which set the acceleration and change of steering angle for the next cycle. This determines *vel* and ω, and hence v.

RobotMoves
$\Delta Robot$
$\Xi World$
$a? : \mathbb{Z}$
$d\omega? : \mathbb{Z}$

$vel' = vel + a? * T$
$\omega' = \omega + d\omega? * T$
$o' = add\,(o, v')$

- This now describes the control loop behaviour of the robot.
- But when is it **safe**?

Moving and Driving Safely

- It is easy to say what it means for the robot to **move safely**: it will not run into any obstacles.

RobotMovesSafely
RobotMoves

$cov\,(c, v') \cap obs = \emptyset$

- Is that **enough**?
- No, this will give us a **false sense** of safety — it only fails when it is **far too late** to **initiate** braking.
- To ensure safety here we would need:

$RobotMovesSafely \Rightarrow RobotMovesSafely'$

Braking and Safe Braking

- Our safety strategy: we must **always** be able to **brake safely**
- We first need to specify **braking** and **safe braking**. Braking is safe if the braking area is clear of obstacles.

RobotBrakes
$\Delta Robot$
$\Xi World$

$vel' = vel - a_{brk} * T$
$\omega' = \omega$
$o' = add\,(o, v')$

RobotBrakesSafely
RobotBrakes

$cov\,(c, brk\,(v, \omega, a_{brk})) \cap obs = \emptyset$

- **Implementing** the overall strategy: if we can move safely, we do, otherwise we brake.
- **Invariant**: we can always brake safely.

The Safe Robot: Implementation

- We drive **safe** if we **will** be able to brake safely.

RobotDrivesSafely
$\Delta Robot$
$\Xi World$

$(cov\,(c, v') \cup cov\,(move\,(c, v'), brk\,(v', \omega', a_{brk}))) \cap obs = \emptyset$
$vel' = vel + a? * T$
$\omega' = \omega + d\omega? * T$
$o' = add\,(o, v')$

- The safe robot implements the safety strategy:

$RobotSafeImpl = RobotDrivesSafely \vee RobotBrakes$

Showing Safety

- We need to **show:**

$RobotSafeImpl \Rightarrow RobotMovesSafely$
$RobotSafeImpl \Rightarrow RobotMovesSafely'$

- The first holds directly.
- The second holds because of the following:

$RobotSafeImpl \Rightarrow RobotBrakesSafely'$
$RobotBrakesSafely \Rightarrow RobotMovesSafely$
$RobotBrakesSafely' \Rightarrow RobotMovesSafely'$

Missing Pieces

- We start off at the origin (or anywhere else), and with velocity 0.
- We need to specify that initially we are **clear of obstacles**.

InitRobot
Robot

$o = (0, 0)$
$vel = 0$
$\omega = 0$
$cont \cap obs = \emptyset$

Summing Up

- The first, abstract, safety specification was *RobotSafe*.
- We implemented this via a second, more concrete specification *RobotSafeImpl*.
- Showing refinement required several lemmas.
- The general safety argument:
  - Safety holds for the initial position: $InitRobot \Rightarrow RobotMovesSafely$
  - Safety is preserved: $RobotSafeImpl \Rightarrow RobotMovesSafely \wedge RobotMovesSafely'$
  - Thus, safety holds always (proof by **induction**).

From Specification to Implementation

- How would we **implement** the birthday book?
- We need a **data structure** to keep track of names and dates.
- And we need to **link** this data structure with the **specification**.
- There are two ways out of this:
  - Either, the specification language also models datatypes (**wide-spectrum language**).
  - Or there is fixed mapping from the specification language to a programming language.

Implementing Arrays

- In Z, arrays can be represented as functions from $\mathbb{N}_1$. Thus, if we want to keep names and dates in arrays (linked by the index), we take

$$names : \mathbb{N}_1 \to NAME$$
$$dates : \mathbb{N}_1 \to DATE$$

- To look up $names[i]$, we just apply the function: $names(i)$.
- To assignment $names[i] := v$, we change the function with the **pointwise update operator** $\oplus$:

$$names' = names \oplus \{i \mapsto v\}.$$

Implementing the Birthday Book

- We need a variable *hwm* which indicates how many date/name pairs are known.
- The axiom makes sure that each name is associated to exactly one birthday.

BirthdayBookImpl

$$names : \mathbb{N}_1 \to NAME$$
$$dates : \mathbb{N}_1 \to DATE$$
$$hwm : \mathbb{N}$$

$$\forall i, j : 1 \mathinner{..} hwm \bullet$$
$$\quad i \neq j \Rightarrow names(i) \neq names(j)$$

Linking Specification and Implementation

- We need to **link** specification and implementation.
- This is done in an abstraction or linking schema:

Abs

BirthdayBook
BirthdayBookImpl

$$known = \{\, i : 1 \mathinner{..} hwm \bullet names(i) \,\}$$

$$\forall i : 1 \mathinner{..} hwm \bullet$$
$$\quad birthday(names(i)) = dates(i)$$

- This specificies how *known* and *birthday* are reflected by the implementing arrays.

Operation: Adding a birthday

- Adding a birthday changes the **concrete state**:

AddBirthdayImpl

$$\Delta BirthdayBookImpl$$
$$name? : NAME$$
$$date? : DATE$$

$$\forall i : 1 \mathinner{..} hwm \bullet name? \neq names(i)$$

$$hwm' = hwm + 1$$
$$names' = names \oplus \{hwm' \mapsto name?\}$$
$$dates' = dates \oplus \{hwm' \mapsto date?\}$$

- We need to show that the pre- and post-states of *AddBirthday* and *AddBirthdayImpl* are related via *Abs*.

Showing Correctness of the Implementation

- Assume a state where the precondition of the specification holds, find the corresponding state of the implementation via *Abs*, and show that this state satisfies the precondition.
- Similarly, assume a pair of states where the invariant of *AddBirthdayBook* holds, find the corresponding states of the implementation via *Abs*, and show that they satisfy the invariant.

Operation: Finding a birthday

- We specify that the found day corresponds to the name via an index *i*.

FindBirthdayImp

$$\Xi BirthdayBookImpl$$
$$name? : NAME$$
$$date! : DATE$$

$$\exists i : 1 \mathinner{..} hwm \bullet$$
$$\quad name? = names(i) \land date! = dates(i)$$

- Note that we are still some way off a concrete implementation — we do not say how we **find** the index *i*.
- To formally show that an iterative loop from 1 to *hdw* always returns the right *i*, we need the **Hoare calculus** (later in these lectures); presently, we argue **informally**.

Summary

- We have seen how we **refine** abstract specifications to more **concrete** ones.
- To **implement** specifications, we need to relate the specification language to a programming language
  - In Z, there are some types which correspond to well-known datatypes, such as finite maps $\mathbb{N}_1 \to T$ and arrays of T.
- We have now reached the **bottom** of the V-model. Next week, we will climb our way up on the right-hand side, starting with **testing**.

Systeme Hoher Qualität und Sicherheit
Vorlesung 7 vom 02.12.2013: Testing

Christoph Lüth & Christian Liguda

Universität Bremen

Wintersemester 2013/14

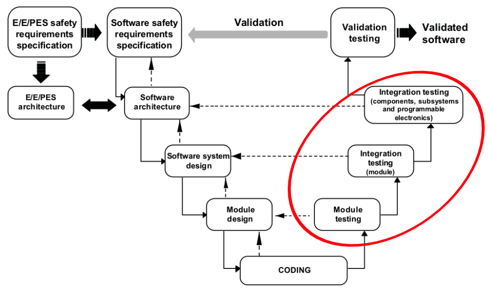
Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- Lecture 5: High-Level Design & Formal Modelling
- Lecture 6: Detailed Specification, Refinement & Implementation
- **Lecture 7: Testing**
- Lecture 8: Program Analysis
- Lecture 9: Verification with Floyd-Hoare Logic
- Lecture 10: Verification Condition Generation
- Lecture 11: Model-Checking with LTL and CTL
- Lecture 12: NuSMV and Spin
- Lecture 13: Conclusions

Your Daily Menu

- What is testing?
- Different **kinds** of tests.
- Different test methods: **black-box** vs. **white-box**.
- Problem: cannot test **all** possible inputs.
- Hence, coverage criteria: how to test **enough**.

Testing in the Development Process



- **Tests** are one way of **verifying** that the system is built according to the specifications.
- Note we can test on **all** levels of the 'verification arm'.

What is testing?

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

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

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

Testing Process

- Test cases, test plan etc.
- system-under-test (s.u.t.)
- Warning: test literature is quite expansive:

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

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.

Basic Kinds of Test

- Functional test
- Non-functional test
- Structural test
- Regression test

Test Methods

- Static vs. dynamic:
  - With **static** tests, the code is **analyzed** without being run. We cover these methods separately 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 continously, 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 employess 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 $15m/s^2$. The vertical acceleration will never be less than zero, or more than $40m/s^2$.

- Equivalence classes or limits?

Black-Box Tests

- Quite typical for GUI tests.
- 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.
- Example: consider lists in C, Java, Haskell, and list reversal.
- 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.
- To abstract from the source code (which is a purely **syntactic** artefact), we consider the **control flow graph** of the program.

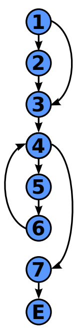
Control Flow Graph (cfg)

- 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.

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 */;
```



- A **path** through the program is a **path** through the cfg.
- Possible paths include:

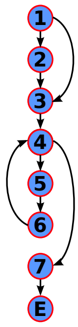
  $[1, 3, 4, 7, E]$
  $[1, 2, 3, 4, 7, E]$
  $[1, 2, 3, 4, 5, 6, 4, 7, E]$
  $[1, 3, 4, 5, 6, 4, 5, 6, 4, 7, E]$
  . . .

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 p covers all statements?

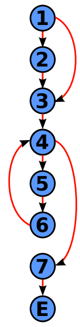
  $p = [1, 2, 3, 4, 5, 6, 4, 7, E]$

- Which state generates p?

  $x = -1$
  y *any*
  z *any*

Example: Branch Coverage

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



- Which (minimal) paths cover all vertices?

  $p_1 = [1, 2, 3, 4, 5, 6, 4, 7, E]$,
  $p_2 = [1, 3, 4, 7, E]$

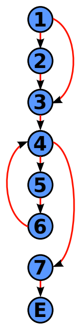
- Which states generate p_1, p_2?

p_1	p_2
$x = -1$	$x = 0$
y *any*	y *any*
y *any*	z *any*

- Note p_3 (corresponding to $x = 1$) does not add to 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 $q_1 \stackrel{def}{=} [1, 2, 3]$
  $q_2 \stackrel{def}{=} [1, 3]$
  $p \stackrel{def}{=} [4, 5, 6]$
  $r \stackrel{def}{=} [4, 7, E]$

  then all paths are given by

  $P = (q_1 \mid q_2)\ p^*\ r$

- Number of possible paths:

  $|P| = 2n_{MaxInt} - 1$

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 because of restricted practicability.

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 operational view, e.g. $x \leq y$ is elementary, but $x < y \vee x = y$ is not, even though they are equivalent.
- In 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) { ...
          T1                  T2
```

T_1	T_2	Result
true	**false**	**false**
false	**true**	**false**

T_1	T_2	Result
true	**true**	**true**
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	Result	
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.

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 2013/14

Lecture 08 (09.12.2013)

Static Program Analysis

Christoph Lüth
Christian Liguda

Universität Bremen

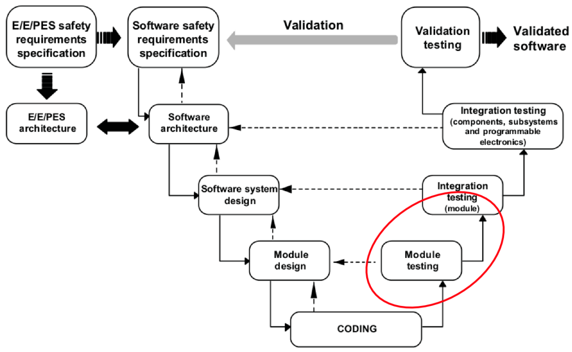
Where are we?

- Lecture 01: Concepts of Quality
- Lecture 02: Concepts of Safety and Security, Norms and Standards
- Lecture 03: Quality of the Software Development Process
- Lecture 04: Requirements Analysis
- Lecture 05: High-Level Design & Formal Modelling
- Lecture 06: Detailed Specification
- Lecture 07: Testing
- Lecture 08: Static Program Analysis
- Lecture 09: Model-Checking
- Lecture 10 and 11: Software Verification (Hoare-Calculus)
- Lecture 12: Concurrency
- Lecture 13: 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.

Static Program Analysis in the Development Cycle



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; *AbsInt* tool)

Program Analysis: The Basic Problem

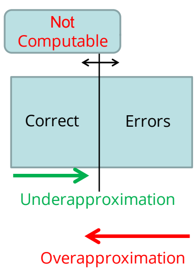
- Basic Problem:

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 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 then 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)?

A Very Simple Programming Language

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

```
[x := a+b]^1;
[y := a*b]^2;
while [y > a+b]^3 do ( [a:=a+1]^4; [x:= a+b]^5 )
```


The Control Flow Graph

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

Labels, Blocks, Flows: Definitions

final($[x := a]^l$) = { l }
final($[\text{skip}]^l$) = { l }
final($S_1; S_2$) = final(S_2)
final(if $[b]^l$ then S_1 else S_2) = final(S_1) ∪ final(S_2)
final(while $[b]^l$ do S) = { l }

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

flow($[x := a]^l$) = ∅
flow($[\text{skip}]^l$) = ∅
flow($S_1; S_2$) = flow(S_1) ∪ flow(S_2) ∪ {(l, init(S_2)) | l ∈ final(S_1) }
flow(if $[b]^l$ then S_1 else S_2) = flow(S_1) ∪ flow(S_2) ∪ { (l, init(S_1), (l, init(S_2) }
flow(while $[b]^l$ do S) = flow(S) ∪ { (l, init(S) } ∪ {(l', l) | l' ∈ final(S) }

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

blocks($[x := a]^l$) = { $[x := a]^l$ }
blocks($[\text{skip}]^l$) = { $[\text{skip}]^l$ }
blocks($S_1; S_2$) = blocks(S_1) ∪ blocks(S_2)
blocks(if $[b]^l$ then S_1 else S_2)
= { $[b]^l$ } ∪ blocks(S_1) ∪ blocks(S_2)
blocks(while $[b]^l$ do S) = { $[b]^l$ } ∪ blocks(S)

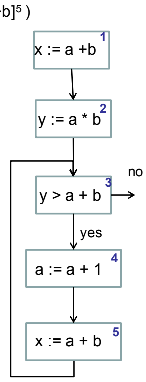
labels(S) = { l | $[B]^l$ ∈ blocks(S)}
FV(a) = free variables in a
Aexp(S) = nontrivial subexpressions of S

Another Example

P = $[x := a+b]^1; [y := a*b]^2; \text{while } [y > a+b]^3 \text{ do } ([a:=a+1]^4; [x:= a+b]^5)$

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] }
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)

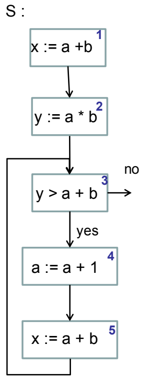
FV(a + b) = {a, b}



Available Expression Analysis

- The avaiable expression analysis will determine:

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



Available Expression Analysis

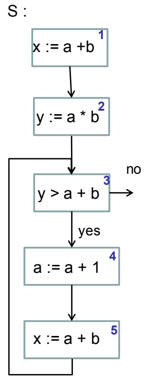
gen($[x := a]^l$) = { a' ∈ Aexp(a) | x∉FV(a') }
gen($[skip]^l$) = ∅
gen($[b]^l$) = Aexp(b)

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

$AE_{in}(l) = \emptyset$, if $l \in init(S)$ and
$AE_{in}(l) = \bigcap \{AE_{out}(l') \mid (l', l) \in flow(S)\}$, otherwise
$AE_{out}(l) = (AE_{in}(l) \setminus kill(B^l)) \cup gen(B^l)$ where $B^l \in blocks(S)$

l	kill(*l*)	gen(*l*)
1		
2		
3		
4		
5		

l	AE_{in}	AE_{out}
1		
2		
3		
4		
5		



Available Expression Analysis

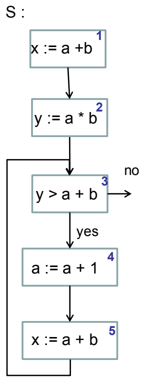
gen($[x := a]^l$) = { a' ∈ Aexp(a) | x∉FV(a') }
gen($[skip]^l$) = ∅
gen($[b]^l$) = Aexp(b)

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

$AE_{in}(l) = \emptyset$, if $l \in init(S)$ and
$AE_{in}(l) = \bigcap \{AE_{out}(l') \mid (l', l) \in flow(S)\}$, otherwise
$AE_{out}(l) = (AE_{in}(l) \setminus kill(B^l)) \cup gen(B^l)$ 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}

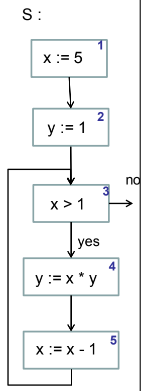
l	AE_{in}	AE_{out}
1	∅	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	∅
5	∅	{a+b}



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

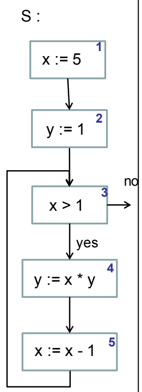
gen($[x := a]^l$) = { (x, *l*) }
gen($[skip]^l$) = ∅
gen($[b]^l$) = ∅

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, ?) } ∪ { (x, k) | B^k is an assignment to x in S }

$RD_{in}(l) = \{ (x, ?) \mid x \in FV(S) \}$, if $l \in init(S)$ and
$RD_{in}(l) = \bigcup \{RD_{out}(l') \mid (l', l) \in flow(S)\}$, otherwise
$RD_{out}(l) = (RD_{in}(l) \setminus kill(B^l)) \cup gen(B^l)$ where $B^l \in blocks(S)$

l	RD_{in}	RD_{out}
1		
2		
3		
4		
5		



Reaching Definitions Analysis

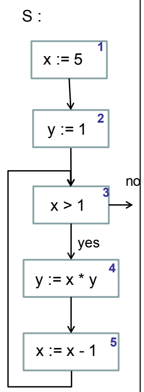
gen($[x := a]^l$) = { (x, *l*) }
gen($[skip]^l$) = ∅
gen($[b]^l$) = ∅

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, ?) } ∪ { (x, k) | B^k is an assignment to x in S }

$RD_{in}(l) = \{ (x, ?) \mid x \in FV(S) \}$, if $l \in init(S)$ and
$RD_{in}(l) = \bigcup \{RD_{out}(l') \mid (l', l) \in flow(S)\}$, otherwise
$RD_{out}(l) = (RD_{in}(l) \setminus kill(B^l)) \cup gen(B^l)$ 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)}

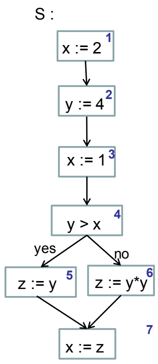


Live Variables Analysis

- 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

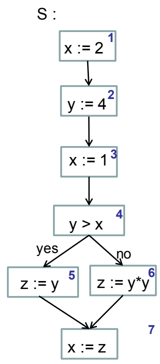
gen($[x := a]^l$) = FV(a)
gen($[skip]^l$) = ∅
gen($[b]^l$) = FV(b)

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

$LV_{out}(l) = \emptyset$, if $l \in final(S)$ and
$LV_{out}(l) = \bigcup \{LV_{in}(l') \mid (l', l) \in flow^R(S)\}$, otherwise
$LV_{in}(l) = (LV_{out}(l) \setminus kill(B^l)) \cup gen(B^l)$ where $B^l \in blocks(S)$

l	kill(*l*)	gen(*l*)
1		
2		
3		
4		
5		
6		
7		

l	LV_{in}	LV_{out}
1		
2		
3		
4		
5		
6		
7		



Live Variables Analysis

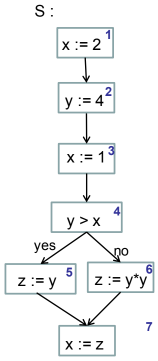
gen([x :=a]^l) = FV(a)
gen([skip]^l) = ∅
gen([b]^l) = FV(b)

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

$LV_{out}(l) = \emptyset$, if $l \in$ final(S) and
$LV_{out}(l) = \bigcup \{LV_{in}(l') \mid (l', l) \in flow^R(S) \}$, otherwise
$LV_{in}(l) = (LV_{out}(l) \setminus kill(B^l)) \cup gen(B^l)$ 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

- Analyse$_\circ$(*l*) = **EV** , if $l \in$ **E** and
- Analyse$_\circ$(*l*) = ⊔ { Analyse$_\bullet$(*l'*) | (*l'*, *l*) ∈ **Flow**(S) }, otherwise
- Analyse$_\bullet$(*l*) = $\mathbf{f_l}$(Analyse$_\circ$(*l*))

With:

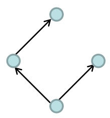
- ⊔ is either ⋃ or ⋂
- EV is the initial / final analysis information
- Flow is either flow or $flow^R$
- E is either {init(S)} or final(S)
- f_l is the transfer function associated with $B^l \in$ blocks(S)

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

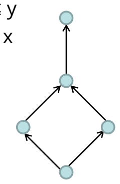
SQS, WS 13/14

Partial Order

- L = (M, ⊑) is a partial order iff
  - Reflexivity: ∀ x ∈ M. x ⊑ x
  - Transitivity: ∀ x,y,z ∈ M. *x* ⊑ *y* ∧ *y* ⊑ *z* ⇒ *x* ⊑ *z*
  - Anti-symmetry: ∀ x,y ∈ M. *x* ⊑ *y* ∧ *y* ⊑ *x* ⇒ *x* = *y*



- Let L = (M, ⊑) be a partial order, S ⊆ M.
  - y ∈ M is upper bound for S (S ⊑ y) iff ∀ x ∈ S. x ⊑ y
  - y ∈ M is lower bound for S (y ⊑ S) iff ∀ x ∈ S. y ⊑ x
  - Least upper bound ⊔X ∈ M of X ⊆ M :
    - X ⊑ ⊔X ∧ ∀ y ∈ M : X ⊑ y ⇒ ⊔X ⊑ y
  - Greatest lower bound ⊓X ∈ M of X ⊆ M:
    - ⊓X ⊑ X ∧ ∀ y ∈ M : y ⊑ X ⇒ y ⊑ ⊓X



SQS, WS 13/14

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 = ⊔∅

SQS, WS 13/14

Transfer Functions

- Transfer functions to propagate information along the execution path
  (i.e. from input to output, or vice versa)
- Let L = (M, ⊑) be a lattice. Set *F* of transfer functions of the form
  f_l : L → L with *l* being a label
- Knowledge transfer is monotone
  - ∀ x,y. x ⊑ y ⇒ f_l(x) ⊑ f_l(y)
- Space *F* of transfer functions
  - *F* contains all transfer functions f_l
  - *F* contains the identity function id, i.e. ∀ x ∈ M. id(x) = x
  - *F* is closed under composition, i.e. ∀ f,g ∈ *F*. (f ∘ g) ∈ *F*

SQS, WS 13/14

The Generalized Analysis

- Analyse$_\circ$(*l*) = ⨆ { Analyse$_\bullet$(*l'*) | (*l'*, *l*) ∈ Flow(S) } ⊔ ι^l_E
  with ι^l_E = EV if $l \in$ E and
  $\iota^l_E = \bot$ otherwise
- Analyse$_\bullet$(*l*) = f_l(Analyse$_\circ$(*l*))

With:

- L property space representing data flow information with (L, ⊔) being 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*

SQS, WS 13/14

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)

SQS, WS 13/14

Systeme Hoher Qualität und Sicherheit
Vorlesung 9 vom 16.12.2013: Verification with Floyd-Hoare-Logic

Christoph Lüth & Christian Liguda

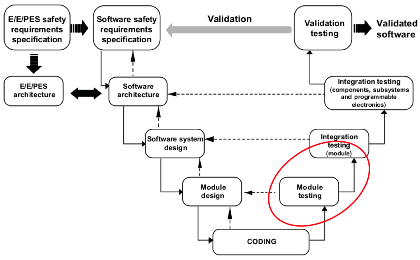
Universität Bremen

Wintersemester 2013/14

Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- Lecture 5: High-Level Design & Formal Modelling
- Lecture 6: Detailed Specification, Refinement & Implementation
- Lecture 7: Testing
- Lecture 8: Program Analysis
- **Lecture 9: Verification with Floyd-Hoare Logic**
- Lecture 10: Verification Condition Generation
- Lecture 11: Model-Checking with LTL and CTL
- Lecture 12: NuSMV and Spin
- Lecture 13: Conclusions

Floyd-Hoare logic in the Development Process



- The Floyd-Hoare calculus **proves** properties of **sequential** 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 do {
  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**.
- 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**)

$$c ::= \mathbf{skip} \mid \mathbf{Loc} := \mathbf{AExp} \mid \mathbf{if}\ \ b\ \ \mathbf{then}\ \ c_1\ \ \mathbf{else}\ \ c_2$$
$$\mid \mathbf{while}\ \ b\ \ \mathbf{do}\ \ c \mid c_1; c_2 \mid \{c\}$$

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 $N = 3$

P	?
C	?
N	3

$\leadsto$

P	1
C	?
N	3

$\leadsto$

P	1
C	1
N	3

$\leadsto$

P	1
C	1
N	3

$\leadsto \ldots$

P	6
C	4
N	3

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

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

Assertion Language

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

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

- **Bexpv**

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

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

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

- 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 ⟷ (X < 10)[X/0]}
X := 0
{X < 10}
```

```
{X < 9 ⟷ X + 1 < 10}
X := X+ 1
{X < 10}
```


Rules of Floyd-Hoare Logic: Conditional and Sequencing

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

- 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.

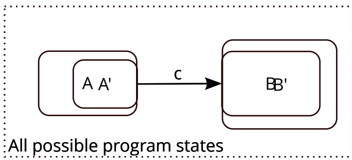
Rules of Floyd-Hoare Logic: Iteration

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

- 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[1] 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 $\models \{A'\}\, c\, \{B'\}$.

[1] If end it does.

Overview: Rules of Floyd-Hoare-Logic

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

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

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

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

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 ones are correct?

$\{1 \le N\}$
```
P := 1;
C := 1;
while C≤N do {
  C := C+1
  P := P×C;
}
```
$\{P = N!\}$

$\{1 \le N\}$
```
P := 1;
C := 1;
while C<N do {
  C := C+1
  P := P×C;
}
```
$\{P = N!\}$

$\{1 \le N \wedge n = N\}$
```
P := 1;
while 0<N do {
  P := P×N;
  N := N−1
}
```
$\{P = n!\}$

A Hatful of Examples

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

$\{A \ge 0 \wedge B \ge 0\}$
```
Q := 0;
R := A−(B×Q);
while B ≤ R do {
 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 do {
  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 Qualität und Sicherheit
Vorlesung 10 vom 06.01.2014: Verification Condition Generation

Christoph Lüth & Christian Liguda

Universität Bremen

Wintersemester 2013/14

Frohes Neues Jahr!

Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- Lecture 5: High-Level Design & Formal Modelling
- Lecture 6: Detailed Specification, Refinement & Implementation
- Lecture 7: Testing
- Lecture 8: Program Analysis
- Lecture 9: Verification with Floyd-Hoare Logic
- **Lecture 10: Verification Condition Generation**
- Lecture 11: Model-Checking with LTL and CTL
- Lecture 12: NuSMV and Spin
- Lecture 13: Conclusions

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:

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

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

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

  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**)

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

Calculuation Verification Conditions

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

Calculating Verification Conditions

$$
\begin{aligned}
\mathrm{pre}(\mathbf{skip}, P) &\stackrel{\mathit{def}}{=} P \\
\mathrm{pre}(X := e, P) &\stackrel{\mathit{def}}{=} P[e/X] \\
\mathrm{pre}(c_0; c_1, P) &\stackrel{\mathit{def}}{=} \mathrm{pre}(c_0, \mathrm{pre}(c_1, P)) \\
\mathrm{pre}(\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, P) &\stackrel{\mathit{def}}{=} (b \wedge \mathrm{pre}(c_0, P)) \vee (\neg b \wedge \mathrm{pre}(c_1, P)) \\
\mathrm{pre}(\mathbf{assert}\ Q, P) &\stackrel{\mathit{def}}{=} Q \\
\mathrm{pre}(\mathbf{while}\ b\ \mathbf{inv}\ I\ \mathbf{do}\ c, P) &\stackrel{\mathit{def}}{=} I
\end{aligned}
$$

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

Correctness of the VC Calculus

Correctness of the VC Calculus

For an annotated program c and an assertion P, let $\mathrm{vc}(c, P) = \{P_1, \ldots, P_n\}$. If $P_1 \wedge \ldots \wedge P_n$, then $\models \{\mathrm{pre}(c, P)\}\, c\, \{P\}$.

- Proof: By induction on c.

Example: Faculty

Let *Fac* be the annotated faculty program:

$\{0 \leq N\}$
```
P := 1;
C := 1;
while C ≤ N inv {P = (C − 1)! ∧ C − 1 ≤ N} do {
  P := P × C;
  C := C + 1
}
```
$\{P = N!\}$

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

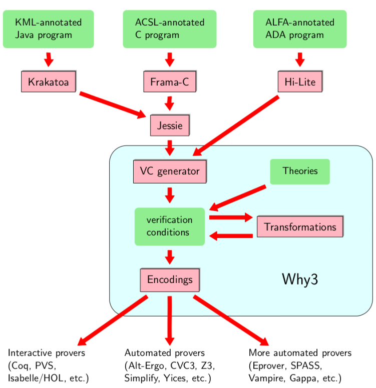
The Framing Problem

- 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 $x \in X$.

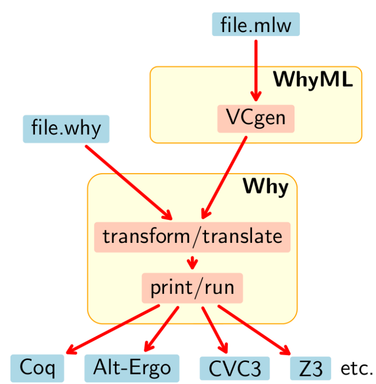
Verification Condition Generation Tools

- 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



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 Qualität und Sicherheit
Vorlesung 11 vom 13.01.2014: Modelchecking with LTL and CTL

Christoph Lüth & Christian Liguda

Universität Bremen

Wintersemester 2013/14

Organisatorisches

- Noch ein Übungsblatt?
- Prüfungen — KW 06 (4./5. Feb.)

Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- Lecture 5: High-Level Design & Formal Modelling
- Lecture 6: Detailed Specification, Refinement & Implementation
- Lecture 7: Testing
- Lecture 8: Program Analysis
- Lecture 9: Verification with Floyd-Hoare Logic
- Lecture 10: Verification Condition Generation
- **Lecture 11: Model-Checking with LTL and CTL**
- Lecture 12: NuSMV and Spin
- Lecture 13: Conclusions

Introduction

- Last lectures: verifying program properties with the **Floyd-Hoare** calculus
- In the Floyd-Hoare calculus, program verification is reduced to a **deductive** problem by translating the program into logic (specifically, state change becomes substitution).
- Model-checking takes a different approach: the system is modelled directly by a finite-state machine, and properties are expressed in some logic for FSM. Program verification reduces to state enumeration, which can be done automatically.
- The logics we will considere here are temporal logic: linear temporal logic (**LTL**) and branching temporal logic (**CTL**)

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**

Finite State Machines

Finite State Machine (FSM)

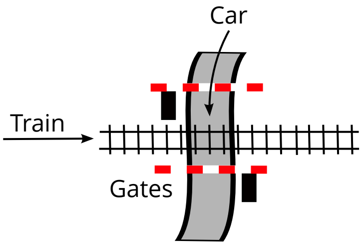
A FSM is given by $\mathcal{M} = \langle \Sigma, \to \rangle$ where

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

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

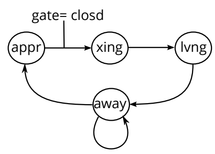
- Many variations of this definition exists, e.g. sometimes we have state variables or labelled transitions.
- Note there is no **final** state, and no input or output (this is the key difference to automata).
- If $\to$ is a function, the FSM is **deterministic**, otherwise it is **non-deterministic**.

The Railway Crossing

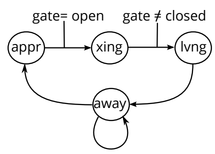


Modelling the Railway Crossing

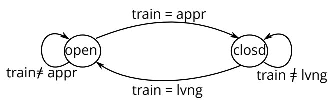
States of the train:



States of the car:



States of the gate:



The FSM

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

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

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

- The transition relation is given by e.g.

$$\langle away, open, away\rangle \to \langle appr, open, away\rangle$$
$$\langle appr, open, away\rangle \to \langle xing, open, away\rangle$$
$$\ldots$$

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)

$\phi ::= \top \mid \bot \mid p$	— True, false, atomic
$\mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \longrightarrow \phi_2$	— Propositional formulae
$\mid \mathsf{X}\,\phi$	— Next state
$\mid \mathsf{F}\,\phi$	— Some Future State
$\mid \mathsf{G}\,\phi$	— All future states (Globally)
$\mid \phi_1\ U\ \phi_2$	— Until

- 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).

Satisfaction and Models of LTL

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

$p \models True$ $p \models \phi \wedge \psi$ iff $p \models \phi$ and $p \models \psi$

$p \not\models False$ $p \models \phi \vee \psi$ iff $p \models \phi$ or $p \models \psi$

$p \models p$ iff $p(p_1)$ $p \models \phi \longrightarrow \psi$ iff whenever $p \models \phi$ then $p \models \psi$

$p \models \neg\phi$ iff $p \not\models \phi$

$p \models \mathsf{X}\,\phi$ iff $p^2 \models \phi$

$p \models \mathsf{G}\,\phi$ iff for all i, we have $p^i \models \phi$

$p \models \mathsf{F}\,\phi$ iff there is i such that $p^i \models \phi$

$p \models \phi\ U\ \psi$ iff there is i $p^i \models \psi$ and for all $j = 1, \ldots, i-1$, $p^j \models \phi$

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.

$$\mathsf{G}\,\neg(car = xing \wedge train = xing)$$

- A car can always leave the crossing:

$$\mathsf{G}(car = xing \longrightarrow \mathsf{F}(car = lvng))$$

- Approaching trains may eventually cross:

$$\mathsf{G}(train = appr \longrightarrow \mathsf{F}(train = xing))$$

- There are cars crossing the tracks:

  $\mathsf{F}(car = xing)$ 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

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

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:

$\mathcal{M}, s \models$ *True*
$\mathcal{M}, s \not\models$ *False*
$\mathcal{M}, s \models p$ iff $p(s)$
$\mathcal{M}, s \models \phi \wedge \psi$ iff $\mathcal{M}, s \models \phi$ and $\mathcal{M}, s \models \psi$
$\mathcal{M}, s \models \phi \vee \psi$ iff $\mathcal{M}, s \models \phi$ or $\mathcal{M}, s \models \psi$
$\mathcal{M}, s \models \phi \longrightarrow \psi$ iff whenever $\mathcal{M}, s \models \phi$ then $\mathcal{M}, s \models \psi$
...

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:

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

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: F $p \longrightarrow$ F 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.
- Next lecture: practical experiments with model-checkers (NuSMV and/or Spin)

Systeme Hoher Qualität und Sicherheit
Vorlesung 12 vom 20.01.2014: NuSMV and Spin

Christoph Lüth & Christian Liguda

Universität Bremen

Wintersemester 2013/14

Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- Lecture 5: High-Level Design & Formal Modelling
- Lecture 6: Detailed Specification, Refinement & Implementation
- Lecture 7: Testing
- Lecture 8: Program Analysis
- Lecture 9: Verification with Floyd-Hoare Logic
- Lecture 10: Verification Condition Generation
- Lecture 11: Model-Checking with LTL and CTL
- **Lecture 12: NuSMV and Spin**
- Lecture 13: Conclusions

Organisatorisches

- Fachgesprächstermine über Stud.IP (2./3. Februar).
- Für eine Modulprüfung: bitte zwei **aufeinanderfolgende** Termine buchen.
- Fachgespräche in der Gruppe, Prüfung alleine.
- Helft uns, die Veranstaltung zu verbessern: Nehmt an der **Evaluation** unter Stud.IP teil!

Introduction

- In the last lecture, we saw how to model systems as **finite-state machines**, and how to specify properties about these in temporal logic — namely, **linear temporal logic** (LTL) and **computational tree logic** (CTL).
- The idea was to allow **automatic** verification or disproving of the properties by **model-checkers** which enumerate the system states.
- Today, we look at two prominent model-checkers: **NuSMV2** and **Spin**. If time permits, we might also look at an interactive theorem prover.

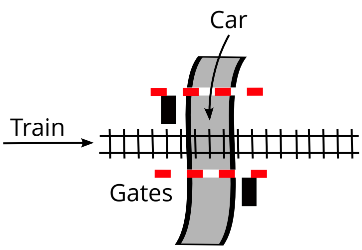
NuSMV

- **NuSMV2** originated with SMV model checker (Edmund Clarke, Ken McMillan). SMV was the first m/c to use BDDs (binary decision diagrams) to represent the transition relation, allowing for much more compact state representation (around 1990). As a result, it could represent up to 10^{20} states.
- **NuSMV2** is currently maintained by CMU, FBK-irst (Trentino, Italy), University of Genoa and University of Trentino.
- It allows simulation, tracing, and supports both LTL and CTL specifications.
- Web Site: `http://nusmv.fbk.eu/`

Spin

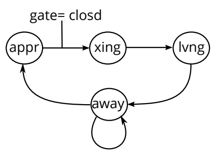
- **Spin** was written by Gerard Holzman. It originated with a protocol analyser (PAN) in 1980, which became Spin in 1989.
- Spin uses the language **Promela** for modelling. As opposed to NuSMV, it allows to model **processes** and communication between them via **channels**. The key difference is that Spin is **asynchronous**, whereas NuSMV is **synchronous**.
- Spin generates a program representing the model, which does the actual model-checking. Besides higher speed, it allows a much more flexible approach to modelling (e.g. one can inject C code into the Promela model).
- Web Site: `http://spinroot.com/`

Recall: The Railway Crossing

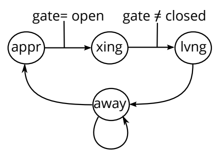


Modelling the Railway Crossing

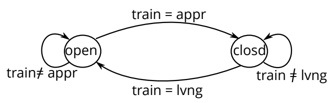
States of the train:



States of the car:



States of the gate:



Summary

- NuSMV vs. Spin:
  - Spin (Promela) is more **concrete**, closer to a programming language.
  - NuSMV supports CTL as well as LTL.
- Model-checking:
  - Can we trust the results? If it finds errors, we get **counter-examples**, but how reliable are positive results?
  - And just how good is our model?