Verifikation von C-Programmen

Vorlesung 1 vom 23.10.2014: Der C-Standard: Typen und Deklarationen

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Der C-Standard

C: Meilensteine

- 1965– 69: BCPL, B; Unix, PDP-7, PDP-11
- 1972: Early C; Unix
- 1976– 79: K& R C
- 1983– 89: ANSI C
- 1990– : ISO C
  - 1985: C++

Geschichte des Standards

- 1978: Kernighan & Ritchie: The C Programming Language
- 1980: zunehmende Verbreitung, neue Architekturen (80x86), neue Dialekte
- 1983: Gründung C Arbeitsgruppe (ANSI)
- 1989 (Dez): Verabschiedung des Standards
- 1990: ISO übernimmt Standard (kleine Änderungen)
- 1999: Erste Überarbeitung des Standards (ISO IEC 9899: 1999)
- 2011: Zweite Überarbeitung des Standards (ISO IEC 9899: 2011)

Nomenklatur

- "Implementation": Compiler und Laufzeitumgebung
- "Implementation-defined": Unspezifiziert, Compiler bestimmt, dokumentiert
  - Bsp: MSB bei signed shift right
- "Unspecified": Verhalten mehrdeutig (aber definiert)
  - Bsp: Reihenfolge der Auswertung der Argumente einer Funktion
- "Undefined": Undefiniertes Verhalten
  - Bsp: Integer overflow
- "Constraint": Einschränkung (syntaktisch/semantisch) der Gültigkeit
- Mehr: §3, "Terms, Definitions, Symbols"

Gliederung

- §3: Terms, Definitions, Symbols — 4 S.
- §4: Conformance — 3 S.
- §5: Environment — 20 S.
  - Übersetzungsumgebung, Laufzeitumgebung
- §6: Language — 165 S.
  - Die Sprache — Lexikalik, Syntax, Semantik
  - Präprozessor
  - Future language directions
- §7: Library — 238 S.
- Anhänge
  - Language syntax summary; Library summary; Sequence Points; Identifiers; Implementation limits; Arithmetic; Warnings; Portability — 112 S.

Eine Sprachkritik

- Philosophie: The Programmer is always right.
  - Wenig Laufzeitprüfungen
  - Kürze vor Klarheit
  - Geschwindigkeit ist (fast) alles
- Schlechte Sprachfeatures:
  - Fall-through bei `switch`, String concatenation, Sichtbarkeit
- Verwirrende Sprachfeatures:
  - Überladene und mehrfach benutzte Symbole (`*`, `()`[1]), Operatorpräzedenzen

[1] 7 Bedeutungen.

Varianten von C

- "K&R":ursprüngliche Version
  - Keine Funktionsprototypen

```
f(x, y)
int x, y;
{
  return x+ y;
}
```

- ANSI-C: erste standardisierte Version
  - Funktionsprototypen, weniger Typkonversionen
- C99, C+11: konservative Erweiterungen

Typen und Deklarationen

Verschiedene Typen im Standard

- Object types, incomplete types, function types (§6.2.5)
  - Basic types: standard/extended signed/unsigned integer types, floating types (§6.2.5)
  - Derived types: structure, union, array, function types
- Qualified Type (§6.2.5)
  - `float const *` qualified (qualified pointer to type)
  - `const float *` not qualified (pointer to qualified type)
- Compatible Types (§6.2.7)
- Assignable Types (§6.5.16, §6.5.2.2)

Namensräume in C

- Labels;
- *tags* für Strukturen, Aufzählungen, Unionen;
- Felder von Strukturen (je eines pro Struktur/Union);
- Alles andere: Funktionen, Variablen, Typen, ...
- Legal: `struct foo {int foo; } foo;`
  - Was ist `sizeof(foo);`?

Deklarationen in C

- Sprachphilosophie: Declaration resembles use
- Deklarationen:
  - *declarator* — was deklariert wird
  - *declaration* — wie es deklariert wird

Der *declarator*

Anzahl	Name	Syntax
Kein oder mehr	*pointer*	`*`
		type-qualifier `*`
Ein	*direct-declarator*	*identifier*
		identifier `[`*expression*`]`
		identifier `(`*parameter-type-list*`)`
Höchstens ein	*initializer*	`=` *expression*

Die *declaration*

Anzahl	Name	Syntax
Ein oder mehr	*type-specifier*	`void`, `char`, `short`, `int`, `long`, `double`, `float`, `signed`, `unsigned`, *struct-or-union-spec*, *enum-spec*, *typedef-name*
Beliebig	*storage-class*	`extern`, `static`, `register`, `auto`, `typedef`
Beliebig	*type-qualifier*	`const`, `volatile`
Genau ein	*declarator*	s.o.
Beliebig	*declarator-list*	`,` *declarator*
Genau ein		`;`

Restriktionen

Illegal:

- Funktion gibt Funktion zurück: `foo ()()`
- Funktion gibt Feld zurück: `foo()[]`
- Felder von Funktionen: `foo[]()`

Aber erlaubt:

- Funktion gibt Zeiger auf Funktion zurück: `int (* fun))();`
- Funktion gibt Zeiger auf Feld zurück: `int (*foo())[];`
- Felder von Zeigern auf Funktionen: `int (*foo[])();`

Strukturen

- Syntax: `struct` *identifier*$_{\text{Opt}}$ `{` *struct-declaration** `}`
- Einzelne Felder (*struct-declaration*):
  - Beliebig viele *type-specifier* oder *type-qualifier*, dann
  - *declarator*, oder
  - *declarator*$_{\text{Opt}}$ `:` *expression*
- Strukturen sind first-class objects
  - Können zugewiesen und übergeben werden.
- `union`: syntaktisch wie `struct` (andere Semantik!)

Präzendenzen für Deklarationen

A	Name zuerst (von links gelesen)
B	In abnehmender Rangfolge:
B.1	Klammern
B.2	Postfix-Operatoren:
	`()` für Funktion
	`[]` für Felder
B.3	Präfix-Operator:
	`*` für Zeiger-auf
C	*type-qualifier* beziehen sich auf *type-specifier* rechts daneben (wenn vorhanden), ansonsten auf den Zeiger `*` links davor

Beispiele

- **char**∗ **const** ∗(∗next)();
- **char** ∗(∗c[10])(**int** ∗∗ p);
- **void** (∗signal(**int** sig, **void** (∗func)(**int**)))(**int**); Lesbarer als

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Typdefinitionen mit `typedef`

- `typedef` definiert Typsynonyme
- `typedef` ändert eine Deklaration von
  - `x` ist eine Variable vom Typ `T` zu
  - `x` ist ein Typsynonym für `T`
- Braucht nicht am Anfang zu stehen (aber empfohlen)
- Nützliche Tipps:
  - Kein `typedef` für `structs`
  - `typedef` für lesbarere Typen
  - `typedef` für Portabilität (e.g. `uint32_t` in `<stdint.h>`)

Zusammenfassung

- Typen in C: Object, incomplete, function; qualified; compatible
- Deklarationen in C:
  - *declarator*, *declaration*
  - Präzendenzregeln: Postfix vor Präfix, rechts nach links

Nächste Woche

- Programmauswertung ("dynamische" Semantik)
- Wie wird folgendes Programm nach dem Standard ausgewertet:

```
int x;
int a[10];
int *y;

x= 0;
x= x+1;
y= &x;
*y= 5;
x= a[3];
y= &a[3];
*y= 5;
```

Verifikation von C-Programmen

Vorlesung 2 vom 30.10.2014: Der C-Standard: Typkonversionen und Programmausführung

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan heute

- Typkonversionen (Typwechsel ohne Anmeldung)
- Felder vs. Zeiger — Das C-Speichermodell
- Zeiger in C:
  - Dynamische Datenstrukturen
  - Arrays
  - Funktionen höherer Ordnung

Verschiedene Typen im Standard

- Object types, incomplete types, function types (§6.2.5)
  - Basic types: standard/extended signed/unsigned integer types, floating types (§6.2.5)
  - Derived types: structure, union, array, function types
- Qualified Type (§6.2.5)
  - `float const *` qualified (qualified pointer to type)
  - `const float *` not qualified (pointer to qualified type)
- Compatible Types (§6.2.7)
- Assignable Types (§6.5.16, §6.5.2.2)

Typkonversionen in C

- Integer promotion (§6.3.1.1):
  - `char`, `enum`, `unsigned char`, `short`, `unsigned short`, *bitfield*
- `float` promoted to `double`
- `T []` promoted to `T *`
- Usual arithmetic conversions (§6.3.1.8)
- Konversion von Funktionsargumenten:
  - Nur bei K&R-artigen Deklarationen, nicht bei Prototypen!
  - Moral: Stil bei Prototyp und Definition nicht mischen!

Zeiger und Felder

- Zeiger sind Adressen

```
int *x;
```

- Feld: reserviert Speicherbereich für `n` Objekte:

```
int y[100];
```

  - Index beginnt immer mit 0
  - Mehrdimensionale Felder . . . möglich
- Aber: $x \not\sim y$

Wann sind Felder Zeiger?

- Wann kann ein Array in ein Zeiger konvertiert werden?

Externe Deklaration: `extern char a[]`	Keine Konversion
Definition: `char a [10]`	Keine Konversion
Funktionsparameter: `f(char a[])`	Konversion möglich
In einem Ausdruck: `x= a[3]`	Konversion möglich

- Wenn Konversion möglich, dann durch Semantik erzwungen
- Tückisch: Externe Deklaration vs. Definition
- Größe: `sizeof`

Mehrdimensionale Felder

- Deklaration: `int foo[2][3][5];`
- Benutzung: `foo[i][j][k];`
- Stored in Row major order (Letzter Index variiert am schnellsten) (§6.5.2.1)
- Kompatible Typen:
  - `int (* p)[3][5] = foo;`
  - `int (*r)[5]= foo[1];`
  - `int *t = foo[1][2];`
  - `int u = foo[1][2][3];`

Mehrdimensionale Felder als Funktionsparameter

- Regel 3 gilt nicht rekursiv:
  - Array of Arrays ist Array of Pointers, nicht Pointer to Pointer
- Mögliches Matching:

Parameter	Argument
`char (*c)[10];`	`char c[8][10]; char (*c)[10];`
`char **c;`	`char *c[10]; char **c;`
`char c[][10];`	`char c[8][10]; char (*c)[10];`

- `f(int x[][]);` nicht erlaubt

Mehrdimensionale Felder als Funktionsparameter

- Regel 3 gilt nicht rekursiv:
  - Array of Arrays ist Array of Pointers, nicht Pointer to Pointer
- Mögliches Matching:

Parameter	Argument
`char (*c)[10];`	`char c[8][10]; char (*c)[10];`
`char **c;`	`char *c[10]; char **c;`
`char c[][10];`	`char c[8][10]; char (*c)[10];`

- `f(int x[][]);` nicht erlaubt (§6.7.5.2)
  - NB. Warum ist `int main(int argc, char *argv[])` erlaubt?

Programmausführung (§5.1.2.3)

- Standard definiert abstrakte Semantik
- Implementation darf optimieren
- Seiteneffekte:
  - Zugriff auf volatile Objekte;
  - Veränderung von Objekten;
  - Veränderung von Dateien;
  - Funktionsaufruf mit Seiteneffekten.
- Reihenfolge der Seiteneffekte nicht festgelegt!
- Sequenzpunkten (Annex C) sequentialisieren Seiteneffekte.

Semantik: Statements

- Full statement, Block §6.8
- Iteration §6.8.5

Semantik: Speichermodell

- Der Speicher besteht aus Objekten
- lvalue §6.3.2.1: *Expression with object type or incomplete type other than `void`*
- lvalues sind Referenzen, keine Adressen
- Werden ausgelesen, außer
  - als Operand von `&`, `++`,`–`, `sizeof`
  - als Linker Operand von `.` und Zuweisung
  - *lvalue (has) array tyoe*
- Woher kommen lvalues?
  - Deklarationen
  - Indirektion (`*`)
  - `malloc`
- Adressen:
  - Adressoperator (`&`)
  - Zeigerarithmetik (§6.5.6)

Ausdrücke (in Auszügen)

- Einfache Bezeichner: lvalue
  - Bezeichner von Array-Typ: Zeiger auf das erste Element des Feldes (§6.3.2.1)
- Felder: `6.5.2.1`
  - `a[i]` definiert als `*((a)+(i))`
  - Damit: `a[i]=*((a)+(i))=*((i)+(a))=i[a]`
- Zuweisung: `6.5.16`
  - Reihenfolge der Auswerung nicht spezifiziert!

Funktionsaufrufe §6.5.2.2

- Implizite Konversionen:
  - Nur wenn kein Prototyp
  - Integer Promotions, `float` to `double`
- Argumente werden ausgewertet, und den Parametern zugewiesen
  - Funktionsparameter sind wie lokale Variablen mit wechselnder Initialisierung
- Reihenfolge der Auswertung von Funktionausdruck und Argumenten nicht spezifiziert, aber Sequenzpunkt vor Aufruf.

Zusammenfassung

- Typkonversionen in C: meist klar, manchmal überraschend
- Auswertung durch eine *abstrakte Maschine* definiert
- Speichermodell:
  - Speicher besteht aus Objekten
  - Durch `char` addressiert (*byte*)
  - Referenzen auf Objekte: *lvalue*

Verifikation von C-Programmen
Universität Bremen, WS 2014/15

Lecture 03 (06.11.2014)

Was ist eigentlich Verifikation?

Christoph Lüth

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: 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:**

- Note that the scope of the directive is market harmonisation, not safety – that is more or less a byproduct.

The Norms and Standards Landscape

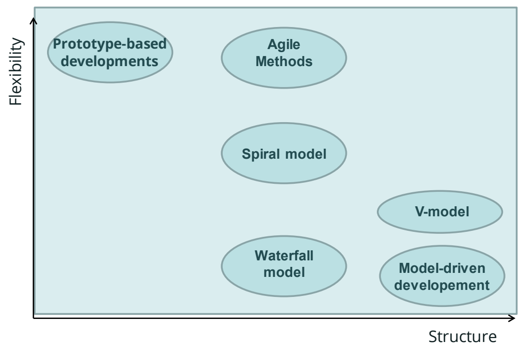
- First-tier standards (*A-Normen*):
  - General, widely applicable, no specific area of application
  - Example: IEC 61508
- Second-tier standards (*B-Normen*):
  - Restriction to a particular area of application
  - Example: ISO 26262 (IEC 61508 for automotive)
- Third-tier standards (*C-Normen*):
  - 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)

Software Development Models

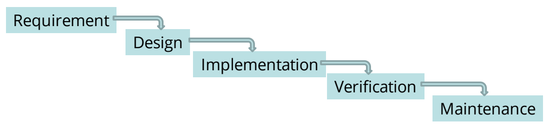
Software Development Models



from S. Paulus: Sichere Software

Waterfall Model (Royce 1970)

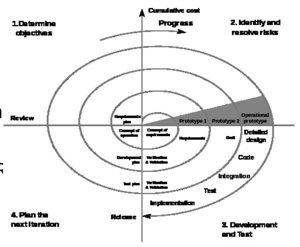
- Classical top-down sequential workflow with strictly separated phases.



- 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 (R
- 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)

$$\begin{tikzcd} \boxed{\text{CIM}} \arrow[r] & \boxed{\text{PIM}} \arrow[r] & \boxed{\text{PSM}} \arrow[r] & \boxed{\text{Code}} \end{tikzcd}$$

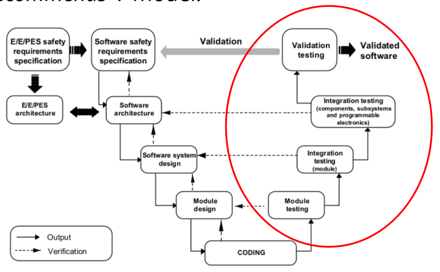
- Variety of tools:
  - Rational tool chain, Enterprise Architect
  - EMF (Eclipse Modelling Framework)
- Strictly sequential development
- Drawbacks: high initial investment, limited flexibility

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.

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.


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

Verification & Validation

V & V

- Verification
  - Making sure the system satisfies safety requirements
  - „Is the system built right (i.e. correctly)?“
- Validation
  - Making sure the requirements are correct and adequate.
  - „Do we build the right (i.e. adequate) system?“

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.

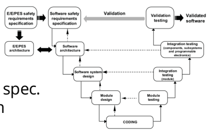
Artefacts in the Development Process

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*.

Introducing IEC 61508

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

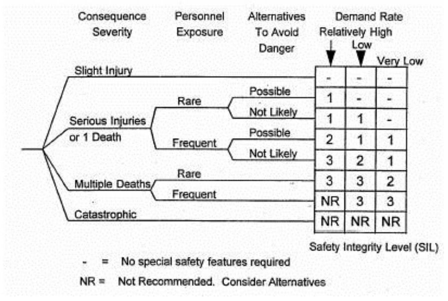
- IEC 61508 does not describe standard procedure to establish a SIL target, it allows for alternatives:
- Quantitative approach
  - Start with target risk level
  - Factor in fatality and frequency

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

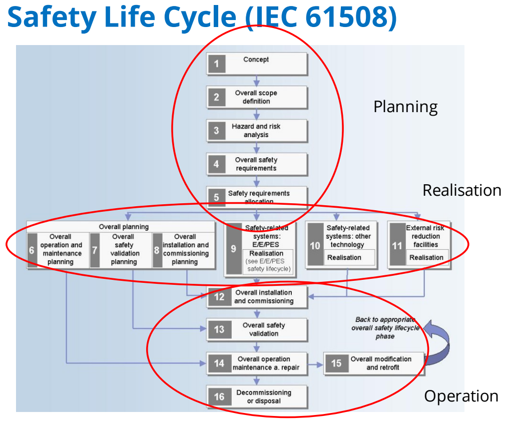
Increasing SIL by redudancy

- 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 (IEC 61508)


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

The Software Development Process

- 61508 „recommends“ V-model development process
- 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

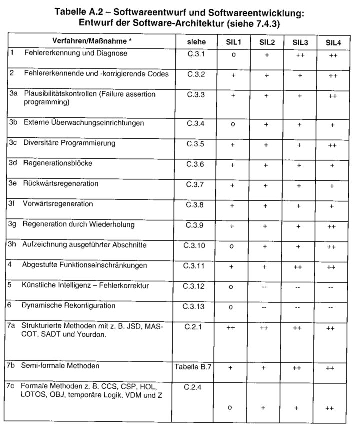


Tabelle A.2 – Softwareentwurf und Softwareentwicklung: Entwurf der Software-Architektur (siehe 7.4.3)

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, MAS-COT, 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

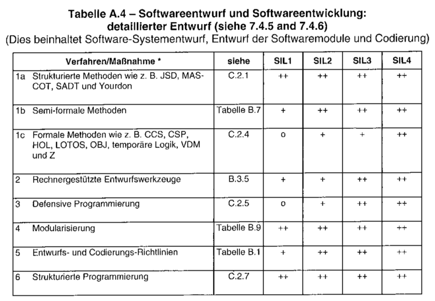


Tabelle A.4 – Softwareentwurf und Softwareentwicklung: detaillierter Entwurf (siehe 7.4.5 and 7.4.6)
(Dies beinhaltet Software-Systementwurf, Entwurf der Softwaremodule und Codierung)

Verfahren/Maßnahme *	siehe	SIL1	SIL2	SIL3	SIL4
1a Strukturierte Methoden wie z. B. JSD, MAS-COT, 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

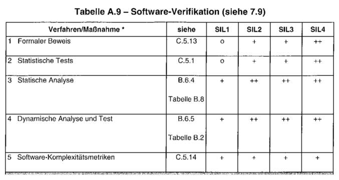


Tabelle A.9 – Software-Verifikation (siehe 7.9)

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.

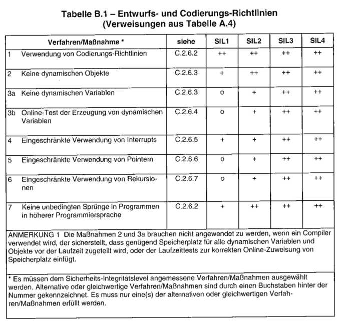


Tabelle B.1 – Entwurfs- und Codierungs-Richtlinien (Verweisungen aus Tabelle A.4)

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

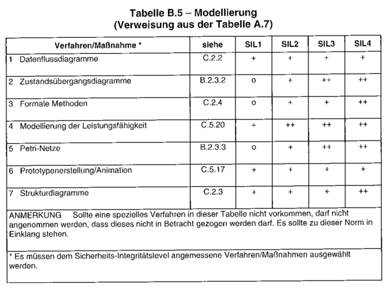


Tabelle B.5 – Modellierung (Verweisung aus der Tabelle A.7)

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'.

Basic Notions of Formal Software Development

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).

Summary

- Norms and standards enforce the application of the state-of-the-art when developing software which is
  - *safety-critical* or *security-critical*.
- Safety standards such as IEC 61508, DO-178B suggest development according to V-model:
  - **Verification** and **validation** link specification and implementation.
  - Variety of artefacts produced at each stage, which have to be subjected to external review.

Verifikation von C-Programmen

Vorlesung 4 vom 13.11.2014: MISRA-C: 2004
Guidelines for the use of the C language in critical systems

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

MISRA-Standard

- Beispiel für eine Codierrichtlinie
- Erste Version 1998, letzte Auflage 2004
- Kostenpflichtig (£40,-/£10,-)
- Kein offener Standard
- Regeln: 121 verbindlich (required), 20 empfohlen (advisory)

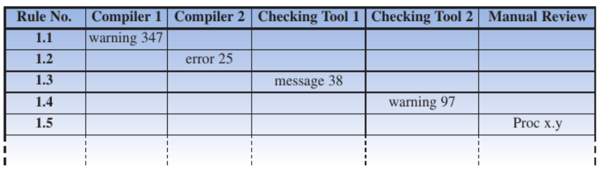
Gliederung

§1 Background: The use of C and issues with it
§2 MISRA-C: The vision
§3 MISRA-C: Scope
§4 Using MISRA-C
§5 Introduction to the rules
§6 Rules

Anwendung von MISRA-C (§4)

- §4.2: Training, Tool Selection, Style Guide
- §4.3: Adopting the subset
  - Produce a compliance matrix which states how each rule is enforced
  - Produce a deviation procedure
  - Formalise the working practices within the quality management system

MISRA Compliance Matrix



Rule No.	Compiler 1	Compiler 2	Checking Tool 1	Checking Tool 2	Manual Review
1.1	warning 347				
1.2		error 25			
1.3			message 38		
1.4				warning 97	
1.5					Proc x.y

Table 1: Example compliance matrix

Die Regeln (§5)

- Classification of rules:
  - Required (§5.1.1): "C code which is claimed to conform to this document shall comply with every required rule"
  - Advisory (§5.1.2):"should normally be followed", but not mandatory. "Does not mean that these items can be ignored, but that they should be followed as far as is reasonably practical."
- Organisation of rules (§5.4)
- Terminology (§5.5) — from C standard
- Scope(§5.6) : most can be checked for single translation unit

Environment

1.1 (req)	All code shall conform to ISO 9899:1990 "Programming languages — C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996 .	
1.2 (req)	No reliance shall be placed on undefined or unspecified behaviour .	2
1.3 (req)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.	1
1.4 (req)	The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	1
1.5 (adv)	Floating-point implementations should comply with a defined floating-point standard .	1

Language extensions

2.1 (req)	Assembly language shall be encapsulated and isolated.	1
2.2 (req)	Source code shall only use /* ... */ style comments.	2
2.3 (req)	The character sequence /* shall not be used within a comment.	2
2.4 (adv)	Sections of code should not be "commented out".	2

Documentation

3.1 (req)	All usage of implementation-defined behaviour shall be documented.	3
3.2 (req)	The character set and the corresponding encoding shall be documented	1
3.3 (adv)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	1
3.4 (req)	All uses of the #pragma directive shall be documented and explained.	1
3.5 (req)	The implementation-defined behaviour and packing of bitfields shall be documented if being relied upon.	1
3.6 (req)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation .	1

Character sets

4.1 (req)	Only those escape sequences that are defined in the ISO C standard shall be used.	1
4.2 (req)	Trigraphs shall not be used.	1

Identifiers

5.1 (req)	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	1
5.2 (req)	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	1
5.3 (req)	A typedef name shall be a unique identifier.	2
5.4 (req)	A tag name shall be a unique identifier.	2
5.5 (adv)	No object or function identifier with static storage duration should be reused.	2
5.6 (adv)	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.	2
5.7 (adv)	No identifier name should be reused.	2

Types

6.1 (req)	The plain char type shall be used only for storage and use of character values.	2
6.2 (req)	signed and unsigned char type shall be used only for the storage and use of numeric values.	2
6.3 (adv)	typedefs that indicate size and signedness should be used in place of the basic numerical types.	2
6.4 (req)	Bit fields shall only be defined to be of type unsigned int or signed int.	1
6.5 (req)	Bit fields of signed type shall be at least 2 bits long.	1

Constants

7.1 (req)	Octal constants (other than zero) and octal escape sequences shall not be used.	2

Declarations and definitions (I)

8.1 (req)	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	1
8.2 (req)	Whenever an object or function is declared or defined, its type shall be explicitly stated.	1
8.3 (req)	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	2
8.4 (req)	If objects or functions are declared more than once their types shall be compatible.	2
8.5 (req)	There shall be no definitions of objects or functions in a header file.	2

Declarations and definitions (II)

8.6 (req)	Functions shall be declared at file scope.	1
8.7 (req)	Objects shall be defined at block scope if they are only accessed from within a single function.	2
8.8 (req)	An external object or function shall be declared in one and only one file.	2
8.9 (req)	An identifier with external linkage shall have exactly one external definition.	2
8.10 (req)	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.	3
8.11 (req)	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.	3
8.12 (req)	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.	2

Initialisation

9.1 (req)	All automatic variables shall have been assigned a value before being used.	3
9.2 (req)	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.	1
9.3 (req)	In an enumerator list, the “=” construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.	1

Arithmetic type conversions (I)

10.1 (req)	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a) it is not a conversion to a wider integer type of the same signedness, or b) the expression is complex, or c) the expression is not constant and is a function argument, or d) the expression is not constant and is a return expression.	2

Arithmetic type conversions (II)

10.2 (req)	The value of an expression of floating type shall not be implicitly converted to a different type if: a) it is not a conversion to a wider floating type, or b) the expression is complex, or c) the expression is a function argument, or d) the expression is a return expression.	1

Arithmetic type conversions (II)

10.3 (req)	The value of a complex expression of integer type shall only be cast to a type of the same signedness that is no wider than the underlying type of the expression.	2
10.4 (req)	The value of a complex expression of floating type shall only be cast to a floating type which is narrower or of the same size.	1
10.5 (req)	If the bitwise operators ˜ and $<<$ are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	2
10.6 (req)	A “U” suffix shall be applied to all constants of unsigned type.	2

Pointer type conversions

11.1 (req)	Conversions shall not be performed between a pointer to a function and any type other than an integral type.	1
11.2 (req)	Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.	1
11.3 (adv)	A cast should not be performed between a pointer type and an integral type.	1
11.4 (adv)	A cast should not be performed between a pointer to object type and a different pointer to object type.	1
11.5 (req)	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	1

Expressions (I)

12.1 (adv)	Limited dependence should be placed on C’s operator precedence rules in expressions.	3 -
12.2 (req)	The value of an expression shall be the same under any order of evaluation that the standard permits.	3
12.3 (req)	The sizeof operator shall not be used on expressions that contain side effects.	3
12.4 (req)	The right-hand operand of a logical && or || operator shall not contain side effects.	3
12.5 (req)	The operands of a logical && or || shall be primary-expressions.	3
12.6 (adv)	The operands of logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, ||, !, =, ==, !=, and ?:).	3

Expressions (II)

12.7 (req)	Bitwise operators shall not be applied to operands whose underlying type is signed.	2
12.8 (req)	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.	3
12.9 (req)	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	2
12.10 (req)	The comma operator shall not be used.	1
12.11 (adv)	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	3
12.12 (req)	The underlying bit representations of floating-point values shall not be used.	3
12.13 (adv)	The increment (++) and decrement (–) operators should not be mixed with other operators in an expression.	1

Control statement expressions

13.1 (req)	Assignment operators shall not be used in expressions that yield a Boolean value.	1
13.2 (adv)	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.	3
13.3 (req)	Floating-point expressions shall not be tested for equality or inequality.	1
13.4 (req)	The controlling expression of a for statement shall not contain any objects of floating type.	1
13.5 (req)	The three expressions of a for statement shall be concerned only with loop control.	1
13.6 (req)	Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.	3
13.7 (req)	Boolean operations whose results are invariant shall not be permitted.	3

Control flow (I)

14.1 (req)	There shall be no unreachable code.	3 -
14.2 (req)	All non-null statements shall either: a) have at least one side effect however executed, or b) cause control flow to change.	3
14.3 (req)	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	3
14.4 (req)	The goto statement shall not be used.	1
14.5 (req)	The continue statement shall not be used.	1
14.6 (req)	For any iteration statement there shall be at most one break statement used for loop termination.	2

Control flow (I)

14.7 (req)	A function shall have a single point of exit at the end of the function.	1
14.8 (req)	The statement forming the body of a switch, while, do ... while or for statement be a compound statement.	1
14.9 (req)	An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	1
14.10 (req)	All if ... else if constructs shall be terminated with an else clause.	1

Switch statements

15.1 (req)	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	1
15.2 (req)	An unconditional break statement shall terminate every non-empty switch clause.	1
15.3 (req)	The final clause of a switch statement shall be the default clause.	1
15.4 (req)	A switch expression shall not represent a value that is effectively Boolean.	1
15.5 (req)	Every switch statement shall have at least one case clause.	1

Functions (I)

16.1 (req)	Functions shall not be defined with variable numbers of arguments.	1
16.2 (req)	Functions shall not call themselves, either directly or indirectly.	3
16.3 (req)	Identifiers shall be given for all of the parameters in a function prototype declaration.	1
16.4 (req)	The identifiers used in the declaration and definition of a function shall be identical.	1
16.5 (req)	Functions with no parameters shall be declared and defined with the parameter list void.	1
16.6 (req)	The number of arguments passed to a function shall match the number of parameters.	2
16.7 (adv)	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	3

Functions (I)

16.8 (req)	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	3
16.9 (req)	A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty.	1
16.10 (req)	If a function returns error information, then that error information shall be tested.	3

Pointers and arrays

17.1 (req)	Pointer arithmetic shall only be applied to pointers that address an array or array element.	3
17.2 (req)	Pointer subtraction shall only be applied to pointers that address elements of the same array.	3
17.3 (req)	$>$, $>=$, $<$, $<=$ shall not be applied to pointer types except where they point to the same array.	3
17.4 (req)	Array indexing shall be the only allowed form of pointer arithmetic.	3
17.5 (adv)	The declaration of objects should contain no more than 2 levels of pointer indirection.	1
17.6 (req)	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	3

Structures and unions

18.1 (req)	All structure or union types shall be complete at the end of a translation unit.	3
18.2 (req)	An object shall not be assigned to an overlapping object.	3
18.3 (req)	An area of memory shall not be reused for unrelated purposes.	x
18.4 (req)	Unions shall not be used.	1

Preprocessing directives (I)

19.1 (adv)	`#include` statements in a file should only be preceded by other preprocessor directives or comments.	3
19.2 (adv)	Non-standard characters should not occur in header file names in `#include` directives.	3
19.3 (req)	The #include directive shall be followed by either a `<filename>` or `"filename"` sequence.	3
19.4 (req)	C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	3
19.5 (req)	Macros shall not be `#define`'d or `#undef`'d within a block.	x
19.6 (req)	#undef shall not be used.	2
19.7 (adv)	A function should be used in preference to a function-like macro.	3
19.8 (req)	A function-like macro shall not be invoked without all of its arguments.	3

Preprocessing directives (II)

19.9 (req)	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	3
19.10 (req)	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	3
19.11 (req)	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	3
19.12 (req)	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	3
19.13 (adv)	The # and ## preprocessor operators should not be used.	3

Preprocessing directives (III)

19.14 (req)	The defined preprocessor operator shall only be used in one of the two standard forms.	3
19.15 (req)	Precautions shall be taken in order to prevent the contents of a header file being included twice.	3
19.16 (req)	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	3
19.17 (req)	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.	3

Standard libraries (I)

20.1 (req)	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	3
20.2 (req)	The names of standard library macros, objects and functions shall not be reused.	3
20.3 (req)	The validity of values passed to library functions shall be checked.	3
20.4 (req)	Dynamic heap memory allocation shall not be used.	2
20.5 (req)	The error indicator errno shall not be used.	2
20.6 (req)	The macro offsetof, in library <stddef.h>, shall not be used.	2
20.7 (req)	The setjmp macro and the longjmp function shall not be used.	2

Standard libraries (II)

20.8 (req)	The signal handling facilities of <signal.h> shall not be used.	2
20.9 (req)	The input/output library <stdio.h> shall not be used in production code.	2
20.10 (req)	The library functions atof, atoi and atol from library <stdlib.h> shall not be used.	2
20.11 (req)	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	2
20.12 (req)	The time handling functions of library <time.h> shall not be used.	2

Run-time failures

21.1 (req) Minimisation of run-time failures shall be ensured by the use of at least one of: 3

a) static analysis tools/techniques;
b) dynamic analysis tools/techniques;
c) explicit coding of checks to handle run-time faults.

MISRA-C in der Praxis

- Meiste Werkzeuge kommerziell
- Entwicklung eines MISRA-Prüfwerkzeugs im Rahmen des SAMS-Projektes
  - Diplomarbeit Hennes Maertins (Juni 2010)
- Herausforderungen:
  - Parser und erweiterte Typprüfung für C
  - Re-Implementierung des Präprozessors
  - Einige Regeln sind unentscheidbar
  - Dateiübergreifende Regeln
- Implementierung:
  - 20 KLoc Haskell, im Rahmen des SAMS-Werkzeugs (SVT)

Verifikation von C-Programmen
Universität Bremen, WS 2014/15

Lecture 05 (19.11.2013)

Statische Programmanalyse

Christoph Lüth

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.

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

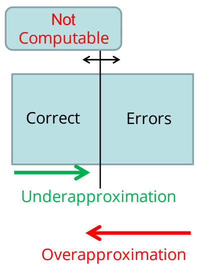
- Basic Problem:

  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
    $b \ := \mathrm{true} \mid \mathrm{false} \mid \mathrm{not}\ b \mid b_1 \, op_b \ b_2 \mid a_1 \, op_r \ a_2$
    with op_b boolean operator (e.g. and, or) and op_r a relational operator (e.g. =, <)
  - Statements given by
    $S \ ::=$
    $[x := a]^l \mid [\mathrm{skip}]^l \mid S_1; S_2 \mid \mathrm{if}\ [b]^l \mathrm{then}\ S_1 \mathrm{else}\ S_2 \mid \mathrm{while}\ [b]^l \mathrm{do}\ S$
- 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) $\mathrm{init}: S \ \to Lab$
  - The final labels (exit points) $\mathrm{final}: S \ \to \ \mathbb{P}(Lab)$
  - The elementary blocks $\mathrm{block}: S \to \mathbb{P}(Blocks)$ where an elementary block is
    - an assignment [x:= a],
    - or [skip],
    - or a test [b]
  - The control flow $\mathrm{flow}: S \to \ \mathbb{P}(Lab \times Lab)$ and reverse control $\mathrm{flow}^{\mathrm{R}}: S \to \ \mathbb{P}(Lab \times Lab)$.
- The **control flow graph** of a program S is given by
  - elementary blocks $\mathrm{block}(S)$ as nodes, and
  - flow(S) as vertices.

Labels, Blocks, Flows: Definitions

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

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

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

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

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

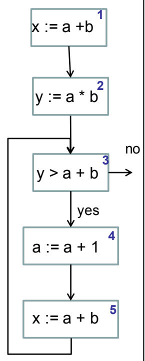
$\mathrm{labels}(S) = \{ l \mid [B]^l \in \mathrm{blocks}(S)\}$
FV(a) = free variables in a
Aexp(S) = nontrivial subexpressions of S

Another Example

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

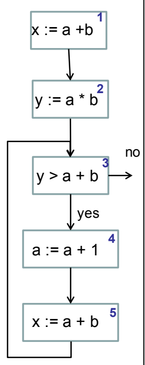


Available Expression Analysis

- 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.

S :



Available Expression Analysis

$\mathrm{gen}([x :=a]^l) = \{ a' \in \mathrm{Aexp}(a) \mid x \notin \mathrm{FV}(a') \}$
$\mathrm{gen}([\mathrm{skip}]^l) = \emptyset$
$\mathrm{gen}([b]^l) = \mathrm{Aexp}(b)$

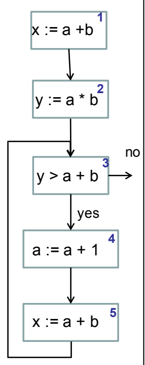
$\mathrm{kill}([x :=a]^l) = \{ a' \in \mathrm{Aexp}(S) \mid x \in \mathrm{FV}(a') \}$
$\mathrm{kill}([\mathrm{skip}]^l) = \emptyset$
$\mathrm{kill}([b]^l) = \emptyset$

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

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

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

S :



Available Expression Analysis

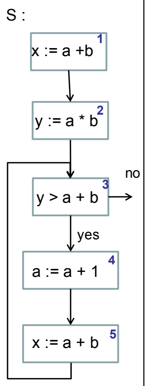
gen($[x := a]^l$) = { a' ∈ Aexp(a) | x ∉ FV(a') }
gen($[\text{skip}]^l$) = ∅
gen($[b]^l$) = Aexp(b)

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

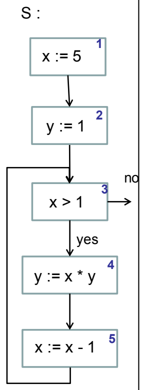
l	AE_{in}	AE_{out}
1	∅	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	∅
5	∅	{a+b}



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

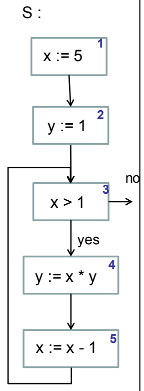
gen($[x := a]^l$) = { (x, *l*) }
gen($[\text{skip}]^l$) = ∅
gen($[b]^l$) = ∅

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($[\text{skip}]^l$) = ∅
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

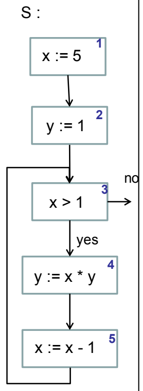
gen($[x := a]^l$) = { (x, *l*) }
gen($[\text{skip}]^l$) = ∅
gen($[b]^l$) = ∅

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($[\text{skip}]^l$) = ∅
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)}

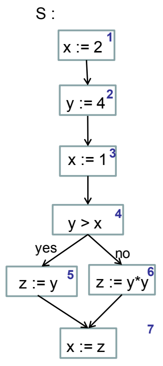


Live Variables Analysis

- 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

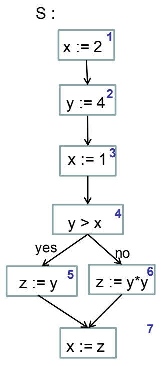
gen($[x := a]^l$) = FV(a)
gen($[\text{skip}]^l$) = ∅
gen($[b]^l$) = FV(b)

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

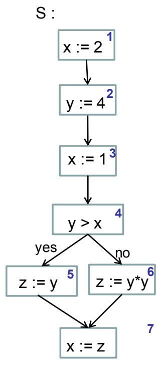
gen($[x := a]^l$) = FV(a)
gen($[\text{skip}]^l$) = ∅
gen($[b]^l$) = FV(b)

kill($[x := a]^l$) = {x}
kill($[\text{skip}]^l$) = ∅
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
- $Analyse_\circ(l) = \sqcup \{ Analyse_\bullet(l') \mid (l', l) \in$ **Flow**(S) }, otherwise
- $Analyse_\bullet(l) = f_l(Analyse_\circ(l))$

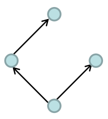
With:

- ⊔ 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 $B^l \in blocks(S)$

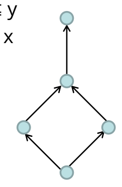
Backward analysis: F = $flow^R$, • = IN, ∘ = OUT
Forward analysis: F = flow, • = OUT, ∘ = IN

Partial Order

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



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



Lattice

A lattice (“Verbund”) is a partial order L = (M, $\sqsubseteq$) such that

- $\sqcup$X and $\sqcap$X exist for all X $\subseteq$ M
- Unique greatest element $\top$ = $\sqcup$M = $\sqcap\emptyset$
- Unique least element $\perp$ = $\sqcap$M = $\sqcup\emptyset$

Transfer Functions

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

The Generalized Analysis

- Analyse$_\circ$(*l*) = $\bigsqcup$ { Analyse$_\bullet$(*l'*) | (*l'*, *l*) $\in$ Flow(S) } $\sqcup$ ι^l_E
  with ι^l_E = EV if *l* $\in$ E and
  ι^l_E = $\perp$ otherwise
- Analyse$_\bullet$(*l*) = f_l(Analyse$_\circ$(*l*))

With:

- L property space representing data flow information with (L, $\sqcup$) 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*

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)

Verifikation von C-Programmen
Vorlesung 6 vom 04.12.2014: Abstract Interpretation

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Galois-Connections

Let L, M be lattices and

$$\alpha : L \to M$$
$$\gamma : M \to L$$

with α, γ monotone, then $\langle L, \alpha, \gamma, M\rangle$ is a Galois connection if

$$\gamma \cdot \alpha \sqsupseteq \lambda l.\, l \qquad (1)$$
$$\alpha \cdot \gamma \sqsubseteq \lambda m.\, m \qquad (2)$$

Example of a Galois Connection

$$L = \langle \mathcal{P}(\mathbb{Z}), \subseteq\rangle$$
$$M = \langle \textbf{Interval}, \sqsubseteq\rangle$$
$$\gamma_{ZI}([a,b]) = \{z \in \mathbb{Z} \mid a \leq z \leq b\}$$
$$\alpha_{ZI}(Z) = \begin{cases} \perp & Z = \emptyset \\ [inf(Z), sup(Z)] & \text{otherwise} \end{cases}$$

Constructing Galois Connections

Let $\langle L, \alpha, \beta, M\rangle$ be a Galois connection, and S be a set. Then

(i) $S \to L$, $S \to M$ are lattices with functions ordered pointwise:

$$f \sqsubseteq g \longleftrightarrow \forall s. f\, s \sqsubseteq g\, s$$

(ii) $\langle S \to L, \alpha', \gamma', S \to M\rangle$ is a Galois connection with

$$\alpha'(f) = \alpha \cdot f$$
$$\gamma'(g) = \gamma \cdot g$$

Generalised Monotone Framework

A Generalised Monotone Framework is given by

- a lattice $L = \langle L, \sqsubseteq\rangle$
- a finite flow $F \subseteq Lab \times Lab$
- a finite set of extremal labels $E \sqsubseteq Lab$
- an extremal label $\iota \in Lab$
- mappings f from $lab(F)$ to $L \times L$ and $lab(E)$ to L

This gives a set of constraints

$$A_\circ(l) \sqsupseteq \bigsqcup\{A_\cdot(l') \mid (l', l) \in F\} \sqcup \iota_E^l \qquad (3)$$
$$A_\cdot(l) \sqsupseteq f_l(A_\circ(l)) \qquad (4)$$

Correctness

Let R be a correctness relation $R \subseteq V \times L$, and $\langle L, \alpha, \gamma, M\rangle$ be a Galois connection, then we can construct a correctness relation $S \subseteq V \times M$ by

$$v\, S\, m \longleftrightarrow v\, R\, \gamma(m)$$

On the other hand, if B, M is a Generalised Monotone Framework, and $\langle L, \alpha, \gamma, M\rangle$ is a Galois connection, then a solution to the constraints $B^{\sqsubseteq}$ is a solution to $A^{\sqsubseteq}$.

This means: we can transfer the correctness problem from L to M and solve it there.

An Example

The analysis SS is given by the lattice $\mathcal{P}(\textbf{State}), \sqsubseteq$ and given a statement S_*:

- $flow(S_*)$
- extremal labels are $E = \{init(S_*)\}$
- the transfer functions (for $\Sigma \subseteq \textbf{State}$):

$$f_l^{SS}(\Sigma) = \{\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \mid \sigma \in \Sigma\} \qquad \text{if } [x := a]^l \text{ is in } S_*$$
$$f_l^{SS}(\Sigma) = \Sigma \qquad \text{if } [\texttt{skip}]^l \text{ is in } S_*$$
$$f_l^{SS}(\Sigma) = \Sigma \qquad \text{if } [b]^l \text{ is in } S_*$$

Now use the Galois connection $\langle \mathcal{P}(\textbf{State}), \alpha_{ZI}, \gamma_{ZI}, \textbf{Interval}\rangle$ to construct a monotone framework with $\langle Interval, \sqsubseteq\rangle$, with in particular

$$g_l^{IS}(\sigma) = \sigma[x \mapsto [i,j]] \quad \text{if } [x := a]^l \text{ in } S_*, \text{ and } [i,j] = \alpha_{ZI}(\mathcal{A}[\![a]\!](\gamma_{ZI}(\sigma)))$$

What's Missing?

- Fixpoints: Widening and narrowing.

Verifikation von C-Programmen

Vorlesung 7 vom 11.12.2014: Basics of Formal Verification

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

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.

A Small Imperative Language

- Arithmetic Expressions (**AExp**)

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

  with variables **Loc**, numerals **N**
- Boolean Expressions (**BExp**)

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

- Statements (**Com**)

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

Semantics of the Small Language

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

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

Semantics in a nutshell

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

Floyd-Hoare-Triples

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

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

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

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

- $\models \{\mathbf{true}\}\ \mathbf{while}\ \mathbf{true}\ \mathbf{do}\ \mathbf{skip}\,\{\mathbf{false}\}$ 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** $\quad v := n, m, p, q, k, l, u, v, x, y, z$
  - defined functions and predicates on **Aexp** $\quad n!, \sum_{i=1}^{n}, \ldots$
  - implication, quantification $\quad b_1 \Rightarrow b_2, \forall v.\, b, \exists v.\, b$
- **Aexpv**

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

- **Bexpv**

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

Rules of Floyd-Hoare-Logic

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

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

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

Rules of Floyd-Hoare Logic: Assignment

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

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

```
X := 10;
{0 < 10 ⟷ (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

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

- 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.

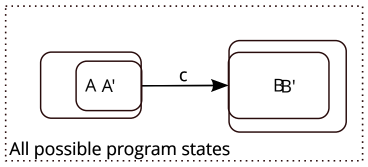
Rules of Floyd-Hoare Logic: Iteration

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

- 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

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

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

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

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

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.

A Hatful of Examples

```
{i = Y ∧ Y ≥ 0}
X := 1;
while ¬ (Y = 0) do {
  Y := Y−1;
  X := 2 × X
}
{X = 2^i}
```

```
{A ≥ 0 ∧ B ≥ 0}
Q := 0;
R := A−(B × Q);
while B ≤ R do {
  Q := Q+1;
  R := A−(B × Q)
}
{A = B ∗ Q + R ∧ 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 ∧ A < (I + 1) ∗ (I + 1)}
```

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:

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

- 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:

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

  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**)

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

Calculuation Verification Conditions

- For an annotated statement $c \in$ **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

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

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

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** $\{P = (C-1)! \wedge C - 1 \leq N\}$ **do** {
  P := P × C;
  C := C + 1
}
$\{P = N!\}$

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

Floyd-Hoare Logic and VCGen for C

The C language is much more complicated than IMP. It has structures, pointers, procedures, and many more operations. But what precisely makes things hard about C?

1. In Hoare-logic, assignment is modelled by substitution of symbols. In C, the state is not symbolic: pointers refer to other locations, so we need a sophisticated state model.
2. The state is ambient: all formulae occuring as VCs or pre/postconditions are state-dependent. This means, backwards calculations (in particular, the assignment rule) are not feasible.
3. To make verification scalable, we need to make it compositional — i.e. one function at a time. This means we have to able to restrict a functions effect on the (global) state (this is called framing).